Learning without Limits

From problem solving towards a unified theory of learning

Beoordelingscommissie:

Prof. dr. J.R. Anderson

Prof. dr. J.A. Michon

Prof. dr. P.L.C. van Geert

Paranimfen:

Alwin Visser

Hans van Ditmarsch

Druk:

Universal Press

Rijksuniversiteit Groningen

Learning without Limits

From Problem Solving towards a Unified Theory of Learning

Proefschrift
ter verkrijging van het doctoraat in de
Psychologische, Pedagogische en Sociologische Wetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. D.F.J. Bosscher
in het openbaar te verdedigen op
donderdag 24 juni 1999
om 14.15 uur
door
Niels Anne Taatgen
geboren op 6 mei 1964
te Enkhuizen

Promotores:

Prof. dr. G. Mulder

Prof. dr. G. R. Renardel de Lavalette

ISBN: 90-367-1061-8

Voorwoord

Een van de momenten die ik me nog kan herinneren van toen ik nog heel klein was, is het moment waarop ik heb leren lezen. Ik moet een jaar of vijf zijn geweest, ik zat thuis (we woonden op een boot) en was bezig met het bestuderen van een boek. Ik begon uiteraard niet met een blanco lei: ik kende alle letters, en ook kon ik al een aantal woorden lezen (die kende ik gewoon uit mijn hoofd). Maar ik kon nog niet zomaar een willekeurig woord lezen. Terwijl ik zo bezig met het uitspreken van woorden die ik al kende en het uitspreken van klanken bij letters die ik nog niet kende, werd mij plotseling duidelijk wat de bedoeling was. Als ik de klanken van de letters van een woord snel uitsprak, en tegelijkertijd naar mezelf luisterde, dan kon ik het woord dat op papier stond verstaan, en dus lezen! Zo verguld was ik met deze ontdekking, dat ik prompt het hele boekje heb uitgelezen (het was een kinderboek, dus dat viel nog wel mee).

Deze herinnering heeft een aantal opmerkelijke kenmerken. Een eerste kenmerk is natuurlijk, dat ik me een gebeurtenis van zo lang geleden nog kan herinneren. Maar voor mij was het dan ook een belangrijk punt in mijn leven: de mogelijkheid te kunnen lezen opent zoveel nieuwe mogelijkheden, dat het beginpunt daarvan natuurlijk memorabel is. Maar een tweede, nog opmerkelijker kenmerk is het alles-of-niets karakter van de gebeurtenis. In luttele minuten beschikte ik over een complexe vaardigheid die ik daarvoor nog niet had. Uiteraard was er wel wat voorwerk nodig om dit moment te kunnen bereiken: de kennis van de letters, het kunnen lezen van enkele woorden, en natuurlijk de vaardigheid om taal te kunnen spreken en te kunnen verstaan. Maar het kwam wel samen in dat ene moment.

Dit proefschrift gaat over dit soort momenten, momenten waarna we opeens veel meer kunnen dan ervoor. Het zijn deze momenten die ons mensen in staat stellen om bijna alles was maar leerbaar is ook te leren. En dat maakt het onderzoek ernaar zo fascinerend, maar tegelijk ook zo moeilijk.

Mijn interesse voor het begrijpen van menselijk denken heeft een lange geschiedenis. Het is allemaal begonnen met Lego, waarmee je alles kon bouwen wat je fantasie je ingaf. Geleidelijk kwamen hier motortjes, schakelaars en lichtgevoelige cellen bij, en voor ik het wist was mijn interesse verschoven naar electronica. Via electronica kwam ik bij de eerste microcomputers terecht, toen nog dingen die uit een printplaat bestonden en geprogrammeerd werden door het intypen van getallen. Nadat ik mijn eerste computer had gekregen (een Commodore PET 2001), was mijn computertijdperk definitief begonnen. Tijdens mijn studie informatica werd mijn interesse gevangen door iets dat nóg ingewikkelder is dan de computer: de menselijke hersenen. Uiteindelijk heeft mij dit via psychologie bij de oprichting van Technische Cognitiewetenschap terecht doen komen, een tak van wetenschap die alles combineert wat mij interesseert.

Vele mensen hebben, direct of indirect, bijgedragen aan het tot stand komen van dit proefschrift. John Anderson wil ik bedanken voor wellicht de belangrijkste bijdrage aan dit proefschrift, de ACT-R theorie. Ook ben ik erg blij dat hij lid is van de beoordelingscommissie en bereid is voor mijn promotie naar Groningen te komen. Een ander lid van de beoordelingscommissie, John Michon, heeft niet alleen aan de wieg gestaan van de studie Technische Cognitiewetenschap, maar heeft ook in de beginperiode van mijn promotieonderzoek en daarvoor mijn afstudeeronderzoek een belangrijke invloed gehad om mijn denken over cognitie. Ik ben blij dat hij nu, aan het einde van het project, er wederom bij betrokken is. Paul van Geert mag ik in dit kader zeker ook niet vergeten te bedanken, met name omdat hij, ondanks zijn vele verantwoordelijkheden als onderzoeksdirecteur, de tijd heeft gevonden om met name mijn beweringen over de ontwikkelingspsychologie kritisch tegen het licht te houden.

Om tot een goed wetenschappelijk product te komen is het belangrijk om regelmatig met mensen te discussiëren die met hetzelfde bezig zijn als jezelf. Alexander van den Bosch, ook een Groninger ACT-R-er van het eerste uur, is een belangrijk voorbeeld van zo iemand. Daarnaast waren ook anderen uit de ACT-R groep uit Groningen een belangrijk klankbord: Mark Dekker, Ritske de Jong, Hedderik van Rijn, Pieter de Vries en Alan White. Ook wil ik in deze context Aladin Akyürek noemen, die in de beginperiode van mijn onderzoek een belangrijke discussiepartner was. Aladin was vaak zo kritisch dat ik er soms bijna moedeloos van werd. Vooral ook omdat hij meestal gelijk had. Dieter Wallach, die ik in het kader van de ACT-R workshop in Pittsburgh heb ontmoet, bleek eveneens een goede partner in het onderzoek: delen van hoofdstuk 6 zijn mede van zijn hand.

Niet alleen onderzoeksgenoten hebben belangrijke bijdragen geleverd aan mijn onderzoek. Met name ook de andere Technische Cognitiewetenschappers zijn door het creëren van een goede werksfeer onontbeerlijk gebleken. Tjeerd Andringa en Petra Hendriks, collega's van het eerste uur, maar ook Tinie Alma, Rineke Verbrugge, Gerard Vreeswijk, Ronald Zwaagstra, Esther Stiekema, Ben Mulder, Henk Mastebroek, Frans Zwarts en niet te vergeten Hans van Ditmarsch, die tijdens zijn vakantie het hele manuscript doorgelezen en becommentarieerd heeft, een taak die vooruitloopt op zijn functie van paranimf. Ook vallen in deze categorie de collega's van de sectie Experimentele en Arbeidspsychologie.

Studenten spelen in veel promotieonderzoeken een belangrijke rol. Annelies Nijdam, Richard Vos en Thijs Cotteleer hebben elk hun bijdrage geleverd. Daarnaast zijn er natuurlijk alle TCW-studenten, die met hun enthousiasme, nieuwsgierigheid en motivatie voor een continu positief achtergrondgeluid zorgen.

Niet alleen collega's, maar ook vrienden zijn van belang. Evelyn van de Veen heeft op het laatste moment binnen twee weken het hele manuscript op taalfouten gecontroleerd, en heeft daarbij een van de laatste hobbels op weg naar de drukker weggenomen. Alwin Visser, een van de paranimfen, is al tien jaar lang samen met mijn roeiploeg "Wrakhout" een belangrijke sportieve steun.

Dan kom ik nu bij Linda Jongman. Linda, je valt eigenlijk in alle categorieën. Niet alleen ben je voor mij persoonlijk heel belangrijk, je hebt ook nog een inhoudelijke bijdrage geleverd aan dit proefschrift (het experiment op pagina 192-193). Bovendien was jij altijd de eerste die mijn schrijfwerk aan een kritische blik onderwierp, en mij waarschuwde als ik met al te onbegrijpelijke schema's dingen juist onduidelijker in plaats van duidelijk maakte.

Tenslotte wil ik mijn promotoren, Bert Mulder en Gerard Renardel de Lavalette bedanken voor de tijd die ze in mijn begeleiding hebben gestoken. De gezamenlijke gesprekken waren voor mij altijd een bron van inspiratie. Met name voor Bert, die ondanks zijn ziekte nog al mijn hoofdstukken nauwkeurig bekeken heeft, heb ik grote bewondering.

Mijn taak zit erop, het is nu aan de lezer om mijn voetstappen in onderzoeksland na te lopen. Voor degenen die niet de volle tocht willen ondernemen, wil ik de verkorte route in de vorm van de Nederlandse samenvatting achter in het proefschrift aanbevelen, aangezien ik mijn best heb gedaan daar een zo begrijpelijk mogelijk verhaal van te maken.

Groningen, 23 april 1999

Niels Taatgen

Contents

CHAPTER 1

Introduction

1.1 The weak method theory of problem solving

Since the birth of cognitive science in the fifties, human problem solving has been one of its central topics. The marriage between psychology and computer science proved to be especially fruitful, since simulation of cognitive processing allowed deeper insights into the empirical data from human participants than was possible with the now old-fashioned techniques offered by behaviorists. A landmark in problem solving was Newell and Simon's 1972 book Human Problem Solving. Newell and Simon show detailed analyses of data collected from human participants, along with results from computer simulation. The main conclusion of the book is that human problem solving can be characterized by a small set of methods. These methods require very little knowledge about a particular problem, and are therefore sometimes called *weak methods*. The tie between psychology and computer science was very strong in this enterprise, since most of the weak methods were algorithms used in artificial intelligence, the sub-discipline of computer science most involved with cognitive science.

The weak-method theory pictures problem solving as search in a problem space. This problem space is a directed graph that has problem states as its nodes, and problem operators as its vertices. A state represents the current configuration of the problem, and operators manipulate these configurations. In problem-solving terms, an operator transforms a current state into a new state. Figure 1.1 shows a simple example of a problem space, the example of the blocks world. This world consists of a table and three blocks, and the only possible action is to move one uncovered block from its current spot to a new spot, either on another block or on the table. Each of the possible configurations of blocks is a state, and is represented in the figure by a rounded rectangle. There is one possible operator: moving a block. This operator can be instantiated in multiple ways, as depicted in the figure by arrows. Suppose the problem starts with the configuration depicted in the upper-left corner of the figure, and the goal is to build the pile of blocks depicted in the lower-right corner. Solving the problem involves selecting a sequence of instantiated operators that transform the start state into the goal state, in this case moving block A to the table, moving block B onto block C, and finally moving block A onto block B.

The problem-space view of problem solving transforms the abstract idea of problem solving into a concrete, easily depictable problem, the problem of deriving the right sequence of operators to transform the start state of a problem into a goal state. To actually find this sequence, one of the weak methods can be applied. Which method is most appropriate depends on the amount and type of knowledge the problem solver has about the problem. The most simple methods are blind-search methods, like generate-and-test, depth-first search and breadth-first search. These methods only assume knowledge about the set of possible states, allowed operators, and the consequences of these operators. Each method systematically searches the problem space until it stumbles over a goal state, in which case the problem has been solved.

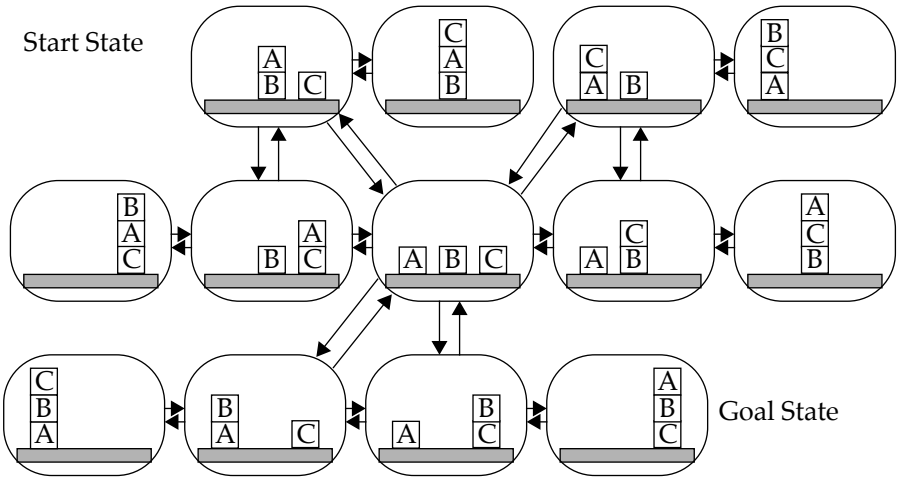


Figure 1.1. The problem space of the blocks world. Rounded rectangles represent states, and arrows represent operators.

Blind-search methods assume that the problem solver has no way of knowing whether a certain state is close to the goal or which operator can bring it closer to the goal. This kind of knowledge is called heuristic knowledge, and methods that use heuristic knowledge are called heuristic methods. The most simple heuristic method is hill-climbing. Hill-climbing assumes a heuristic function that can estimate the distance between a state and the goal state. Using this function, the operator that leads to the most promising new state can be selected. For example, in the blocks-world problem of figure 1.1 the heuristic function might be the number of blocks that are in the right place with respect to the goal state.

A more complex method is means-ends analysis. Means-ends analysis involves a comparison between the goal state and the current state, and the selection of an operator that reduces the difference. If the selected operator is not applicable in the current state, a subgoal is created to reach a state in which the desired operator is applicable. Figure 1.2 shows an example of means-ends analysis: planning a trip from Groningen to Edinburgh. The most notable difference between Groningen and Edinburgh is that they are situated in different countries. So an operator is sought that reduces this difference, in this case flying from Amsterdam to London. This operator is, however, not applicable in Groningen. So getting from Groningen to Amsterdam becomes a subgoal, and is solved by taking the train to Amersfoort and then to Amsterdam. The difference between London and Edinburgh can be found in the same way. An important advantage of means-ends analysis is its divide-and-conquer strategy. This aspect is especially important if the problem space is large or infinite, which is often the case in practice. The disadvantage of means-ends analysis

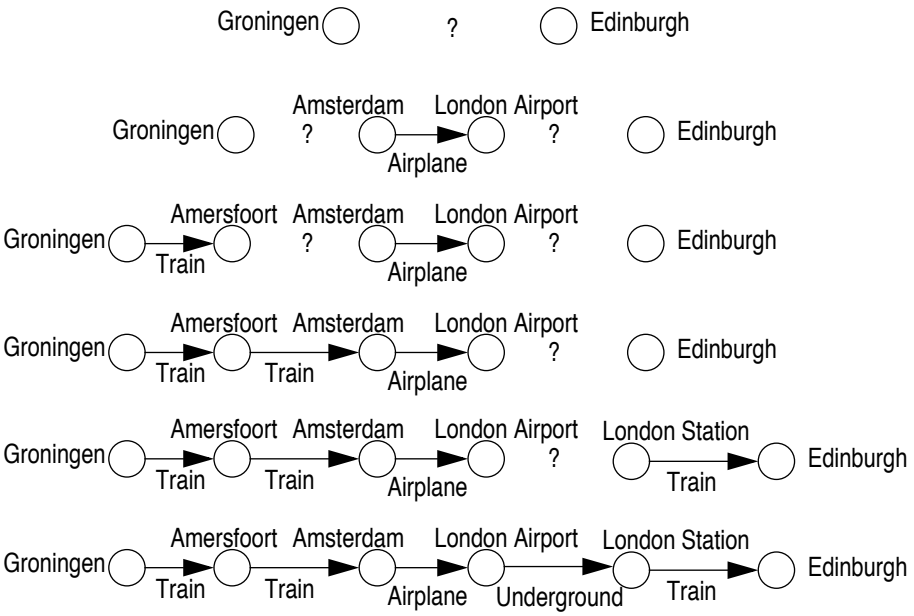


Figure 1.2. Step-by-step creation of a plan to travel from Groningen to Edinburgh

is its requirement of additional knowledge. It must be possible to find differences between states, differences must be ranked in some way (in the example: a difference in country is more important than a difference in city), and operators must be keyed to these differences.

To summarize: for each of the weak methods there is a parallel between the knowledge needed and efficiency. One would expect that as participants gain more knowledge in a certain problem domain, they will tend to use more efficient methods. Jongman (1997) has found some evidence for this hypothesis. In her study, participants have to find information on the Internet. While a majority of the participants start using a hill-climbing strategy, many of them switch to means-ends analysis as they gain experience.

Problems of the weak-method theory

Despite the fact that the weak-method theory offers a systematic framework for studying problem solving and provides explanations for many aspects of human problem solving, it leaves a number of questions unanswered. A first problem of the weak-method theory is that it assumes precise and unambiguous knowledge about problem states, operators and goals, even for the most simple blind-search methods. This assumption is correct for many problems used in problem-solving research, like the towers-of-hanoi, the eight puzzle and blocks-world puzzles.

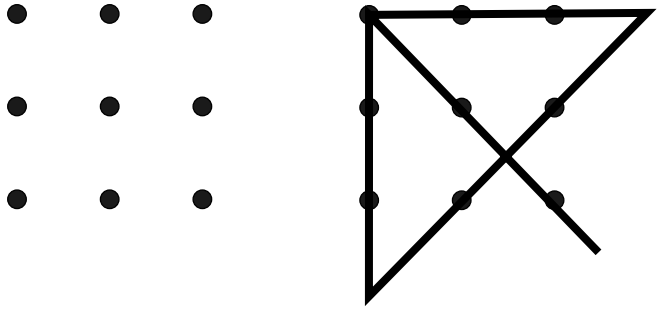


Figure 1.3. The nine-dots problem (left) and its solution (right)

Research that stresses the importance of insight in problem solving on the other hand, uses problems for which this assumption does not hold. A well-known example is the nine-dots problem (figure 1.3), in which the problem is to connect all nine dots using four connected lines. The difficult aspect of this problem is the fact that a solution is only possible if lines are used that extend beyond the borders of the 3x3 grid of points. In problem-space terms, the problem basically has an infinite number of possible operators, since there are infinitely many ways to draw a line. Participants tend to reduce the set of possible operators to operators that just draw lines between two points of the 3x3 grid. The crucial step in solving the problem is the realization that this reduction is too severe. So problem solving not only involves selecting the right sequence of operators, but also finding out what the operators are, and what they do. The example also shows that re-evaluating the operators currently used may be part of the problem-solving process.

A second problem is the fact that in many cases not all the activities of a participant can be explained in terms of clear problem-solving methods. Participants use multiple strategies for a single problem, skipping between them and inventing new ones on the fly. People tend to forget results if they can not be used immediately, or have to use memorization techniques to prevent forgetting things. Finally, and that is a criticism often quoted, people have the ability to "step out of a problem", to reason about their own reasoning (see, for example, Hofstadter, 1979, for an extensive discussion of this point). Evidence for this kind of meta-reasoning are exclamations like "This doesn't work at all", and "Let's try something different". Although it is not at all clear how extensive meta-reasoning can be, people evidently use some sort of self-monitoring to prevent them from doing the wrong thing for too long.

The third problem is that the weak-method theory does not explain how people gain a higher level of understanding in a certain problem domain. An example of this is mathematics. In order to be able to solve simple algebraic equations like $2x + 3 = 7$, one must master simple arithmetic first. Composite concepts from arithmetic form

the basic building blocks of simple algebra. Solving $2x + 3 = 7$, for example, takes at least four simple arithmetic operators. Experience allows people to collapse these operators into higher-level operators, so they can solve the equation in just one step. Mastering simple equations is a prerequisite for more complex mathematics like differential equations. The idea of several levels of understanding is quite common in developmental psychology, and stems from the stage theories of Piaget (1952).

The three problems discussed above, although somewhat different in nature, boil down to the same issue: learning. The problem solving process is not a pure search process but also includes exploration. Exploration is necessary to learn what the possible operators are and what they do or to question the operators if they fail to perform well. Exploration can also derive and refine heuristic knowledge, and find out what methods and strategies are most suitable for the current problem. To be able to do this several strategies must be tried and compared. Learning can also result in higher-level operators and an increase the level of abstraction of the problem-solving process. Exploration can also attempt to use knowledge from other domains for solving the current problem.

Problem solving from the viewpoint of skill learning

The main topic of this thesis is to study the learning aspect of problem solving. While complex problem solving will be the starting and the end point, several tasks will be discussed that are not strictly problem-solving tasks, unless one adopts Newell's claim that any task is a problem-solving task. So the topic is actually broader and extends to skill-learning in general, with complex problem solving as the main skill to be studied.

An important theme throughout the thesis will be the distinction between implicit and explicit learning (Reber, 1967; Berry, 1997). Implicit learning is often defined as unconscious learning: the learner is unaware of the fact that he or she is learning, and is unable to recall what is learned afterwards. Increased task performance is the only indication something is learned. Explicit learning, on the other hand, supposes a more active role of the problem solver. An example of this type of learning is when the participant sets explicit exploration goals, or explicitly decides to memorize aspects of a certain problem because they may be useful for another problem. Both types of learning are important for problem solving. During search the problem solver gains information in an implicit fashion, since learning is not the goal but only a by-product. Search for the solution may be alternated by setting explicit learning goals that try to combine earlier experiences, perform generalizations, explore other problem domains, or, on a more mundane level, try to keep partial results active in memory.

One of the core problems of search as a problem solving method is the fact that problem spaces are often very large or infinite. The reason for this is that in each state

there are several possible operators leading to new states. In general, the size of the problem space grows exponentially with the maximum length of the sequence of operators. For human purposes, blind, systematic search in an exponential problem space will only be successful if the sequence of operators is relatively short. If longer sequences are required, knowledge is needed to offer guidance in the choice of operators, to retrieve partial sequences used for other problems, or to collapse several operators into one composite operator. Therefore, the maximum capacity for solving problems in a certain domain is determined by the knowledge for this domain extended by a limited amount of search. Actually solving a problem using search, possibly enhanced by explicit learning, may extend the space of solvable problems.

Figure 1.4 shows an impression of this idea. The top figure represents the set of all possible problems, loosely ordered in the sense that more complex problems are at the top of the rectangle, and less complex problems at the bottom. The horizontal dimension is used to indicate that problems are related to each other. Some of these problems can be solved by a particular individual by a relatively simple procedure. This portion of the set is indicated by the black area at the bottom of the set. Problems in the grey area require more effort, and need some combinatorial search. Problems in the white area require so much search that the problem becomes practically unsolvable.

Problems in the black area take relatively little time. As soon as the grey area is entered, combinatorial search is needed, which increases the time requirements exponentially. At some point these time requirements become unpractically high, marking the beginning of the white area. Learning increases the black area in the set, sometimes by a single item, sometimes, after generalization, by a substantial area. As a consequence the border between the grey and the white area also moves outwards, as indicated by the small arrows in the graph. Take for example the left-most peak in the figure. This might represent the algebra skill of a certain individual. This individual is, for example, capable of solving equations without much effort (black area), able to solve simple problems of integration by trying out several different methods (grey area), but not proficient in doing double integrations yet (white area).

The time requirements are shown in the graphs at the bottom of the figure. Problems that can be solved in a direct fashion usually do not require much time. But once the expertise runs out and combinatorial search is needed, the grey area is entered and the time requirements increase exponentially with the amount of search needed. Due to this increase, the time requirements soon exceed practical limitations (white area). This discussion is of course still very informal. A more formal approach will be discussed later in this chapter.

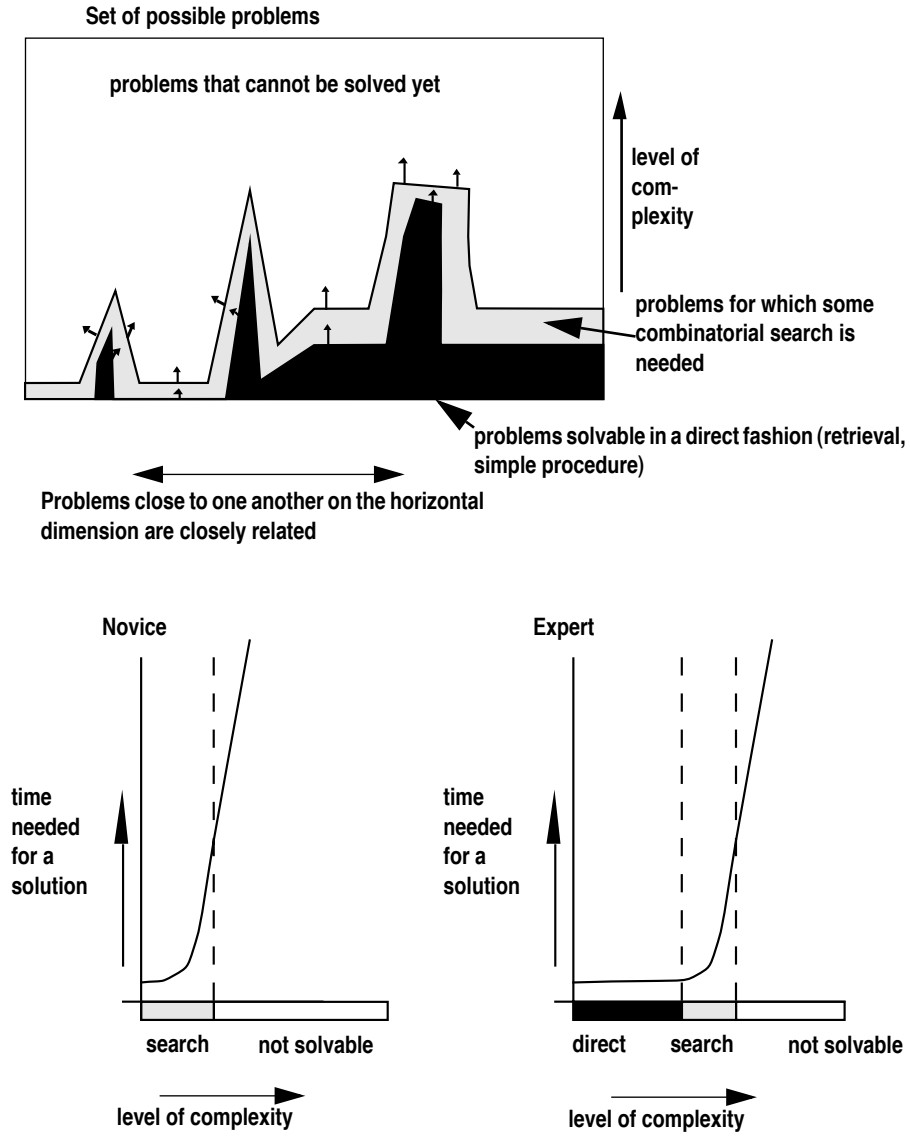


Figure 1.4. Impression of the set possible problems. Some can be solved easily (black area), some need combinatorial search to find the solution (grey), and others cannot be solved at all. The top figure outlines the expertise of a certain arbitrary individual who has three areas of expertise. The small arrows in the top figure indicate the effects of learning. The "peaks" in the figure indicate areas in which this particular individual is an expert. The two graphs at the bottom indicate the time to find the solution given the type of search needed and can be seen as a vertical cross-section of the top figure. The left graph represents a novice, who has to use search for almost everything, and the right graph represents an expert, who can solve many problems in a direct way.

1.2 How to study learning in complex problem solving?

Within cognitive science there are a number of research paradigms to study learning. The main paradigm to study learning is the experimental paradigm used in cognitive psychology. A common approach is to present participants with a sequence of similar problems, and see how their performance improves with respect to reaction time (latency) and rate of errors. One fundamental law found in this fashion is the power law of practice, a law that states that regardless what the task is, the reaction time can be described by the function:

$$T_n = bn^{-\alpha} \quad \textbf{(1.1)}$$

In this equation T_n is the reaction time for trial n, and b and α are constants.

Another method often employed in experimental learning research is the search for dissociation effects. Typical experiments first expose participants to some information, which is tested at a later time using different types of tests. Typical examples of dissociations are:

- If a participant is tested directly after learning, he or she performs equally on test A and B. If he or she is tested again after a week, performance on test A is the same, but performance on test B has decreased severely (Tulving, Schacter & Stark, 1982)
- Performance of a participant suffering from amnesia is equal to a healthy participant on test A, but much worse on test B (e.g., Morris & Gruneberg, 1994).

Dissociations are often used as evidence for the existence of different memory systems, for example a separate implicit and explicit memory.

Although experimental work offers many insights in the nature of learning and memory, the standard experimental paradigm is limited to phenomena that can be quantified easily in, for example, the power law of practice, or the hypothesis that implicit and explicit information is stored in separate memory systems. Take, for example, the power law of practice. The smooth form of the curve suggests learning is a continuous process. Although this may well be the case, this is not necessarily so. As noted by, amongst others, Siegler (1996), the smooth curve may have resulted from averaging several step-functions. Also, a hypothesis about the existence of two separate memory systems is rather crude, and offers little insight into the necessity of separate memory systems. As we will see later on in chapter 4, dissociations can sometimes also be explained using a single memory system.

Because the pure experimental paradigm can only state rather global hypotheses, it often limits itself to experiments where all participants behave roughly the same. Participants only tend to behave the same if there is only one way to do things. In terms of figure 1.4, only problems in the black area are investigated. The grey area,

however, is the area where interesting learning phenomena with respect to problem solving can be found. In that area almost all participants will behave differently due to the exponential number of choices. So it will be much more difficult to state hypotheses in the usual fashion. As a consequence, participants can no longer be studied as a group, but must be studied individually. The challenge is to still be able to make generalizations about the population, despite individual differences.

The paradigm that machine learning offers for the study of learning radically differs from what is used in experimental psychology. Complexity is the main challenge. Although many types of algorithms are used, some of which will be reviewed in chapter 2, the common goal in machine learning is to derive generalized knowledge from examples, sometimes guided by domain knowledge. The goal is to arrive at an accurate generalization using the most efficient algorithm. In a typical machine learning study to judge the quality of a new learning algorithm, a set of examples is used. For example, in a medical setting, an example contains a number of symptoms and a diagnosis. The set of examples is split in two parts, a training set and a test set. The training set is first given to the learning algorithm, which tries to generalize rules or other representations that can predict a diagnosis from the symptoms. The test set is then used to judge the correctness of these representations. A new algorithm is judged to be promising, if its performance on the test set exceeds the performance of a number of established learning algorithms. Performance is measured by the number of correct classifications the algorithm makes on the test set, and by the time it needs to learn the training set.

Machine learning algorithms are quite powerful when judged with respect to efficiency and quality of classifications. Whether or not the learning of such algorithms has any similarity to human learning is not considered important. This does not necessarily mean algorithms from machine learning are useless for studying human learning, since evolution may well have optimized human learning in the same way computer scientists try to optimize machine learning. Nevertheless, machine learning algorithms often make computational assumptions that are not easy realizable for humans. People can, for example, not learn large databases of examples easily.

A third domain of cognitive science in which learning is studied is developmental psychology. Developmental psychology studies changes in behavioral capacities in children over time. According to some theories these changes can be characterized by transitions between stages, meaning there are periods with little change and periods with large changes in capacities. Developmental psychologists are mainly interested in these changes and their characteristics, and less in the processes that cause these changes. Studying how a complex skill is learned in several steps can offer important clues about the nature of the learning processes that cause the change in skill. Possibly the most cited example is the learning of past tenses (Rumelhart & McClelland, 1986; Pinker & Prince, 1988; Elman, Bates, Johnson,

Karmiloff-Smith, Parisi & Plunkett, 1996). The literature often distinguishes three stages in this particular skill. In the first stage, all past tenses are learned as separate facts. The second stage is characterized by the discovery of a rule for regular verbs. This rule is, however, overregularized so that irregular verbs that were used correctly in the first stage are now put in the past tense using the regular rule. Only in the third stage the irregular words are recognized and used correctly. Although this description tells us little about the processes that cause change, it reveals nevertheless that an interplay between rules and examples is important. We will come back to this issue in a later chapter.

Since one of the goals of this thesis is to approach learning in problem solving from an experimental perspective, we have to deal with the problems mentioned earlier. Alan Newell already noted the limitations of the classical experimental paradigm in 1973, when he wrote his famous paper titled "You can't play twenty questions with nature and win". According to Newell, psychologists investigate cognitive phenomena. Examples of these phenomena are:

1. recency effect in free recall
2. reversal learning
3. rehearsal
4. imagery and recall

Although these are just four items from Newell's list of 59, they will discussed more extensively later in this thesis. All four of them will turn out to be important for problem solving. Newell's criticism focuses on the fact that despite the fact that all these phenomena are researched thoroughly, no clear theory of cognition emerges. The main type of structures psychology attempts to establish are binary oppositions. Among these oppositions are the following:

1. Continuous versus all-or-none learning
2. Single memory versus dual memory
3. Existence or non-existence of latent learning
4. Stages versus continuous development
5. Conscious versus unconscious

Again these examples are picked from a list of 24, and will become important at some point in the discussion later on. The point Newell tries to make is that resolution of all these binary oppositions ("20 questions") will not bring us any closer to a grand theory of cognition. Fortunately, Newell also proposes three solutions to the problem, two of which we will discuss here.

A first solution is to create a single system, a model of the human information processor that can carry out any task. He also proposed a candidate for such a

system, namely a production system. If a production system would be given the right set of rules, it should, in principle, be able to perform any experimental task. It turned out this solution became the main paradigm dominating the rest of Newell's work. In 1990, he wrote "Unified Theories of Cognition", in which he presented his final proposal for a grand theory of psychology. At that time, the single system idea had already spread, and other people had been thinking about unification as well. Anderson's 1983 book "The Architecture of Cognition" is an example, in which Anderson presents his ACT* system. The Rumelhart and McClelland 1986 books "Parallel Distributed Processing" also attempt to bring all types of cognitive phenomena together in a single paradigm. The single-system approach has two important aspects: it constrains the researcher in the type of theories he can state, in the sense that the theory has to fit in the system, and it forces the researcher to be very precise: the theory has to be simulated within the system. In this thesis I will also conform to this single-system approach. The system is the ACT-R 4.0 system, a descendant of ACT*, as described in Anderson & Lebiere (1998). ACT-R and its competitors will be discussed in detail in chapter 2.

A second solution Newell offers is to analyze a single complex task. This addresses the problem that psychology often designs its experiments according to the phenomenon studied, resulting in simple tasks. The choice for a complex task is less common, because it is very hard to relate results of a complex task to a single phenomenon. Experiments using complex problems do however offer sufficient samples of all human cognitive capacities. A possible complex problem is chess. Chess involves planning, means-ends analysis, all types of learning, mental imagery, etc. If we were able to know all there is to know how people play chess, would this not be a big step towards understanding cognition in general? I will also adopt this second recommendation in this thesis. But in stead of focussing on a single task, I will focus on a single class of problems: NP-complete problems.

1.3 NP-complete problems

What is a complex problem? There are many ways to give a subjective judgement of how difficult a problem is. Chess is difficult and tic-tac-toe is easy. Fortunately, there are more formal ways to categorize problems. A formal approach also requires us to be more precise on what a problem is. First we will examine how to formally look at problems and problem solving. Then we will look at what complexity is, and by the end of the section the class of NP-complete problems will be discussed.

In informal speech, the term problem has two different meanings. We can talk about a problem as a general category, for instance the problem of deciding the next move in chess. It is not possible to give an answer to this question, because it depends on the position on the chessboard. The term problem can also be used in a more specific

sense: what move should I make on a chess board with the black king at e1, the white king at e3, and a white rook at a8? In this case a specific answer is possible: move the white rook to a1, checkmate. A problem in the general sense is a set of problems in the specific sense. To avoid confusion, the formal term problem refers to a problem in the general sense, and a specific problem is called an instance. This distinction can roughly be compared to the terms "task" and "trial" in experimental psychology: a task is a general description of what a participant must do, a trial is a specific instance of the task.

A formal definition of a problem defines it as a set of instances and a criterion. "Solving a problem" means that we decide for a particular instance whether or not it satisfies the criterion. For example, a formal description of the informal problem of deciding whether there is a forced checkmate for white specifies the set of instances as the set of all possible configurations of chess pieces on the board, and the criterion is the yes/no-question of whether a forced checkmate is possible for white. This last characterization of the criterion is of course still informal: the formal definition involves all rules of chess. "Solving a problem" in formal terms means we have a solution for all instances in the set. If the set is finite, the solution may be an enumeration of all solutions, but usually a solution for a problem is some algorithm that can decide whether the criterion holds or not. In order to formalize an informally stated problem, like "what is the best next move in a certain chess position" it must be stated as a yes/no-question, for example "Is move X in position Y the best move?". A solution to this problem is an algorithm that computes this answer for any possible move and possible position in chess.

To be able to define the complexity of a problem in a meaningful way, it has to have an infinite set of instances and there must be some way to measure the "size" of an instance. Unfortunately, the example of chess is not infinite: although the number of positions is huge, it is nevertheless finite. The game of checkers, which is played both on an 8 x 8 board and a 10 x 10 board, can be generalized to a problem with an infinite number of instances by allowing n x n boards.

Very simple problems, however, can have infinite sets of instances. For example, the problem to decide whether a list is sorted or not has an infinite set of instances. The size can be defined by the length of the list. Summarizing, a problem can be defined in the following terms:

- A set of instances
- A criterion (a yes/no-question about instances)
- A size function on each of the instances
- A solution, i.e. an algorithm that can decide whether the criterion holds for a certain instance.

Now suppose we have some way to find the "best" algorithm to solve a problem. This "best" algorithm will use computational resources. The amount of resources the algorithm consumes is an indication of its efficiency. But since it is the best algorithm for a certain problem, the efficiency of the best possible algorithm defines the complexity of the problem. So what do we mean by "use of computational resources"? There are two computational resources, time and (memory) space. Since the use of these resources is related, time is often the resource an analysis of complexity focuses on.

Complexity theory uses relative time instead of absolute time. The time it takes a certain algorithm to solve a problem is expressed in a complexity function, which maps the size of the instance on the amount of time it takes to solve the problem. This complexity function gives a much clearer indication of the efficiency of the algorithm than absolute time can. If a small increase in the size of the instance causes a large increase in time, the algorithm is inefficient. So, an algorithm with a linear complexity function is more efficient than an algorithm with a square or exponential complexity function. Complexity functions can be calculated, if the algorithm is known, or approximated empirically when the algorithm is too messy or complicated to analyze.

If we want to know the complexity of a problem, we are looking for an algorithm that solves this problem and has the best complexity function. So the complexity of a problem is the lower bound of the complexity of all the algorithms that solve it. Some problems, like deciding whether an item is in an unsorted list, have only a linear complexity. The most efficient algorithm is to examine the items in the list one by one and compare them to the item we seek. The average number of items that has to be examined is $n/2$ if the item is in the list, and n if it is not in the list (n is the length of the list). Other problems have a higher complexity. Problems that have an exponential time complexity are called *intractable*. The source of complexity is often combinatorial: if, for example, n elements must be ordered, the number of possible ordenings is $n!$. If there is no systematic way to weed out the major part of these ordenings, the problem is intractable. In the case of checkers on arbitrarily large boards (I will not use chess, because it is finite), the number of board positions to be examined increases exponentially with the number of moves you want to look ahead. The question if white can win from the first move is decidable in principle, but not in practice, because there are more possible checkers games than atoms in the universe.

Why is exponential time complexity intractable, and polynomial complexity tractable? Because exponential functions grow so much faster than polynomial functions. This can be illustrated using part of a figure from Garey & Johnson (1979) that shows the time it takes to solve an instance of a problem of size n, given the fact that a single operation can be carried out in a microsecond (figure 1.5). One might

Complexity function	n=10	n=20	n=30	n=40	n=50	n=60
n linear	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2 polynomial	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^5 polynomial	.1 second	3.2 seconds	24.3 second	1.7 minutes	5.2 minutes	13.0 minutes
2^n exponential	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries

Figure 1.5. Comparison between a linear, two polynomial and an exponential time complexity function (from Garey & Johnson, 1979)

argue that some problems with a polynomial complexity, especially with a high exponent (e.g. n^{200}), are also intractable, but in practice these types of complexities never occur (only in contrived problems).

The consequences of intractability

Intractable problems are interesting candidates for Newell's idea of a complex problem that exposes many aspects of human cognition. Since they have an exponential time complexity, it is impossible to use an efficient procedure that solves all instances of a problem. It is, however, not always necessary to be able to solve all instances of a problem, it may be enough to be able to solve a relevant subset of them. Relevant in this case means that the system somehow has a use for them. So for any particular intractable problem, we may have a situation similar to figure 1.4: some instances of the problem, particularly instances with a small size, can be solved efficiently, some instances need additional search that may require exponential exploration of cases, and some cases are unsolvable within a reasonable amount of time. So intractable problems may serve as a miniature but faithful representative of the case of learning problem solving.

To further improve on the representativeness of example problems, we will narrow down the set of intractable problems to the set of NP-complete problems, which is in itself a subset of NP. "NP" is an abbreviation for Non-deterministic Polynomial. A problem in NP can be solved by a non-deterministic Turing Machine in polynomial time. Less technically, given an instance of an NP-problem and a path to its solution, (so not only the yes/no-answer, but also the choices that are made to reach it) it is possible to check this solution using a tractable algorithm. In summary: finding the solution may be intractable, but checking it is tractable.

Although it has technically not yet been proven that NP-complete problems really are intractable, the general consensus is that they are for all practical purposes (Garey & Johnson, 1979). In the next section some examples of NP-complete problems will be examined, showing the broad range of domains they appear in. Nevertheless they form a tight class due to their completeness-property. This completeness property means that any NP problem can be transformed into a particular NP-complete problem by an algorithm of polynomial complexity. So, take for example the travelling salesman problem, a well-known NP-complete problem. Due to its completeness property, it is possible to take an instance of another NP-complete problem, for example resolving a particular ambiguity in a sentence, and transform this instance into an instance of the travelling-salesman problem. So if you find an efficient solution for one particular NP-complete problem, you have automatically found an efficient solution for all of them. Regrettably, this doesn't mean that a partial solution (in terms of figure 1.4) will be at all helpful in this matter. Nevertheless, if it is possible to gain insight into how people partially overcome the problems of combinatorial explosion with respect to one particular NP-complete problem by learning, it carries the promise that this learning scheme may also work for other hard problems.

1.4 Examples of NP-complete problems

NP-complete problems may be very interesting problems to study, but this endeavor is purely academical if these problems have little to do with real-life situations. In this section a number of examples of NP-complete problems will be examined to show that NP-complete problems are part of everyday life. For some of these problems, for example language, almost everyone is an expert. For other problems, for example scheduling problems, extensive skill is normally thought of as the competence of an expert.

Most of the problems discussed here have been catalogued by Garey and Johnson (1979), together with their basic reference. Most examples explained here require some answer, instead of just "yes" or "no". A problem that requires an answer can almost always be converted to a yes/no question, as I have shown in the case of chess.

Examples in Planning
A plan is a sequence of actions that achieves a certain goal. Sometimes reaching the goal is enough, but in other cases additional requirements must be satisfied, like finding the most efficient sequence. Planning nearly always involves time and optimizing time. People plan every day, for example how to make coffee, a plan that requires no search. Other types of planning do require some search, for example to plan a route through town to go through a shopping list (Hayes-Roth & Hayes-

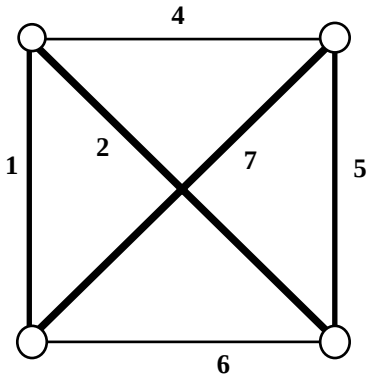


Figure 1.6. Example of the travelling-salesman problem (left)

Roth, 1979), or to plan a meal (Byrne, 1977; van den Berg, 1990). Other planning tasks involve scheduling, for example school and hospital rosters, or planning symposia (Numan, Pakes, Schuurman & Taatgen, 1990). Computer science has invested much effort in programs for planning, resulting in different approaches: hierarchical, non-hierarchical, opportunistic and script-based planners (See Akyürek, 1992 for an overview).

Most planning problems are intractable unless heavily restricted. We will look at two intractable problems that are closely related to planning. In the *travelling-salesman problem* the task is to find the shortest closed route connecting a set of cities. More precisely, a number of cities is given and a matrix stating the distance between each pair of cities. A route is a sequence of cities, and the length of the route is the sum of the distances between successive cities. Figure 1.6 shows a case of the travelling-salesman problem with four cities. The thick line indicates the shortest route, which has a length of 15.

The general problem is NP-complete, but we can imagine a particular salesman, who always visits a subset of, say, 25 cities, and who has developed his own private strategy for solving the problem. When this salesman is transferred to another part of the country, he has only limited use for his experience: he can use some of his old knowledge, but must devise some new procedures for his new environment.

The travelling-salesman problem obviously is a planning task, and shows much resemblance to other planning tasks, for example the shopping-task from Hayes-Roth & Hayes-Roth (1979). It is often easy to prove that a certain planning task is intractable, using the fact that the travelling-salesman problem is intractable.

A second planning problem is scheduling. In this problem each instance consists of a set of tasks, each of which has a certain length, a number of workers, a partial order on the tasks, and an overall deadline. The task is to create a schedule for all the tasks,

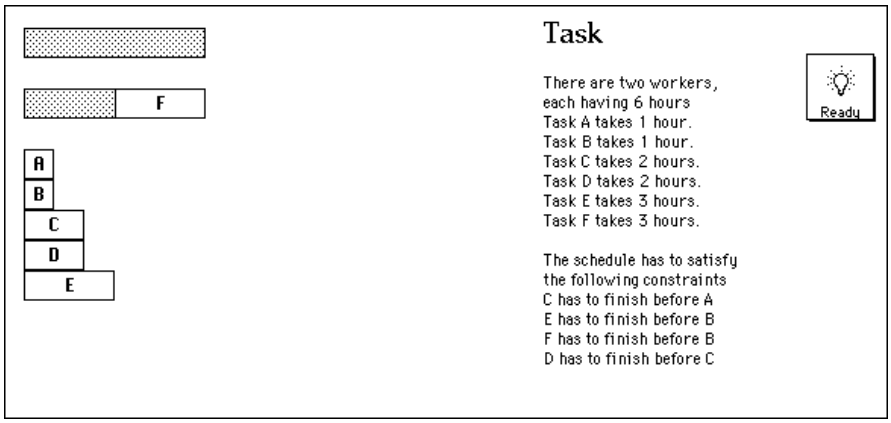


Figure 1.7. Screen shot of an instance of the scheduling problem. In this experiment (discussed in chapter 3), participants can move around the blocks to create the schedule.

obeying the precedence constraints as specified in the partial order and the deadline. Figure 1.7 shows an example of an instance of this problem.

Again the general problem is intractable, but particular sub-problems may be attainable. For example, the timetable of a certain school is always made by a particular deputy headmaster. Although it takes him two full weeks every year, he is the only one in the school who can do it at all. Previous experience is the key to successful problem solving in this case, another indication of the importance of learning.

Language

Understanding natural language is generally not considered to be problem solving. However, formal theories of language, especially with respect to grammar or syntax, use the same terminology as the formal theory of problem solving. For example, part of the natural language understanding process is concerned with the question whether a sentence is grammatically correct. In problem-solving terms, the set of instances is the set of all (finite) sequences of words. The criterion is the question whether a particular sequence of words is grammatically correct or not.

Part of research in linguistics concerns the construction of grammars and grammar systems that describe language. The goal of a grammar of a certain natural language is to be able to produce every grammatical sentence in that language, but no other, ungrammatical, sentences. A grammar system aims to provide a framework within which all grammars of natural languages can be fitted. Chomsky (Chomsky & Miller, 1963) has defined the basic types of grammars: finite-state, context-free, context-sensitive and unrestricted grammars, called the Chomsky hierarchy.

Grammars can produce language, but to parse natural language, to decide whether a certain sentence belongs to the language, an automaton is needed. It can be shown that each of the four grammar systems from the Chomsky hierarchy corresponds to a certain type of automaton: finite-state grammars to finite-state automatons, context-free grammars to push-down automatons, context-sensitive grammars to linear-bounded automatons and unrestricted grammars to Turing machines. Chomsky has shown that finite-state grammars are too restricted to be able to generate a complete natural language. Unrestricted grammars, due to their connection with Turing machines, are undecidable. This leaves context-free and context-sensitive as possible formalisms, of which context-free is always considered a more desirable alternative, because parsing a context-free grammar is tractable. The important question is whether the generative power of context-free grammars is enough to generate natural languages.

Barton, Berwick & Ristad (1987) argue this discussion has outlived its usefulness, and more modern methods must be used. They show that the fact that a grammar is context-free is no guarantee for efficiency. The generalized phrase structure grammar system (GPSG), for example, has the seemingly attractive property that any GPSG can be converted to an equivalent context-free grammar. This suggests that since context-free grammars can be parsed efficiently, a GPSG can also be recognized easily. Barton et al. show this argument is misleading, because for a GPSG G of size m the equivalent context-free grammar has in the order of $3^{m!m^{2m+1}}$ rules.

Barton et al. propose complexity theory as a replacement for the equivalence-to-context-free-grammar criterion. It is a much more precise and reliable instrument to measure the efficiency of a grammar system. They also argue efficiency is an important criterion for natural language systems: if we have a formal system of a natural language that uses combinatorial search (an intractable algorithm) where it is not really necessary, there obviously is some systematic property in the language that the formal system fails to account for. For nearly all grammar systems used in linguistics, parsing turns out to be an intractable problem. According to Barton et al., this is partly due to intractable properties of language itself, but can often also be attributed to the formalism: it simply fails to account for certain features of the language. The unnatural sources of complexity must of course be expelled from the formalism, but the natural intractable properties can not. They must be accounted for by what Barton et al. call a *performance theory*, in which they hint at least some combinatorial search takes place.

An example of an intractable property of natural language understanding is the combination of agreement and lexical ambiguity. Agreement refers to two or more words in a sentence having the same number, gender or other feature, like in subject/verb agreement. Lexical ambiguity refers to the fact that a single word can

have different functions, as with homonyms. For example, the word 'walk' can be either a noun or a verb. In the case of a verb, it can be either first or second person singular or plural. Agreement grammars are simple context-free grammars that can account for both agreement and ambiguity. However, Barton et al. prove that the problem of parsing an agreement grammar is NP-complete with respect to the length of the sentence.

The conclusion is that although care must be taken to avoid unnecessary intractability in language, it cannot be avoided altogether, and what remains must be accounted for by a so-called performance theory. This performance aspect is of course rather problematic. In Chomsky's theory the performance part of language is just a degraded version of the "ideal" competence counterpart due to human limitations. In the theory of Barton et al. performance has a function that can not be formalized but is nevertheless crucial.

So, even understanding everyday language is in itself already an intractable problem. Therefore language performance can not be explained purely by a static syntactic framework. The learning component, as is the case with other intractable problems, has to be part of the explanation of the human capacity of understanding language.

Puzzles and games

Research on problem solving is often done on toy problems. Puzzles in which letters must be replaced by numbers, missionaries and cannibals must be shipped over a river, problems where blocks must be rearranged by a robot arm, or puzzles where numbered tiles must be pushed around to get them in sequence. The problem with each of these problems is to what extent results, either empirical or by simulation, can be generalized to other domains. Especially in the case of computer simulation, the fact that a simulation solves a certain problem has no significance, because a conventional algorithm can do the same job. Even when a convincing simulation can be made, it is difficult to generalize the results.

Some games are different, however. They go beyond the toy-realm, because they keep eluding final solutions. Chess, checkers and Go are examples of games that have a long history of gradual improvement, never reaching perfection. The games of checkers and Go are intractable when generalized to an n x n board. Although chess is highly complex, it is not intractable because it can not easily be generalized to an n x n board, and standard chess games are always finite. Complexity theory needs some kind of infinity to work with. Other kind of puzzles are also intractable, for examples fitting words into an n x n crossword puzzle.

So, studying intractable problems is a far greater challenge than working with toy-problems. They pose a real challenge to problem solving, but with a larger pay-off. Since no conventional algorithms exist, the fact alone that a system simulating

human problem solving on an intractable problem can solve certain cases is significant.

Mathematics
The main and original source of intractable problems is mathematics. Many problems involving graphs, partitioning, matching, storage, representation, sequencing, algebra and number theory are intractable (Garey & Johnson, 1979).

One of the most well-known NP-complete problems stems from logic: the *satisfiability problem* (SAT) (Cook, 1971). The problem is to find, for a propositional logic formula, values for the variables so that the formula evaluates to true. A straight-forward algorithm used to solve SAT is called truth-table checking, which amounts to checking every possible combination of values for the variables. Since in propositional logic a variable can have two values, the number of combinations to be checked is 2^n, where n is the number of variables. This is obviously an exponential function, leading to an intractable algorithm.

Another nice property of the problems mentioned here is the fact that they are (with the possible exception of the language problems) knowledge-lean. That is, they are already highly complex without needing huge data banks of knowledge to work on. This makes simulation a lot easier, and the results easier to interpret.

1.5 The limits of task analysis, or: why is learning necessary for problem solving?

The picture sketched in figure 1.4 is one of gradual change in mastery of a problem due to learning. But how important is this learning aspect? Suppose we want to make a task analysis of scheduling. Wouldn't it be useful to constrain the total set of instantiations of scheduling to a manageable subset, and derive a set of rules and methods that can account for that subset? More specifically, is it possible to create an account of how an expert scheduler works, assuming an expert is someone with a set of methods that is broad enough to render learning superfluous?

Suppose we have a scheduling expert. This expert can solve some instances of scheduling, but has problems with other instances: these instances take too much time to solve. For each expert, we can divide the total set of scheduling instances into two subsets: the instances he can solve and the instances he can not solve. This boundary is not entirely clear-cut, since the amount of time the expert is willing to invest in a solution plays a role, but due to the exponential increase in solution time this willingness for extra effort pushes the boundary only very slightly. There are many experts of scheduling, each of whom has his own expertise and knowledge of

scheduling, so each has his own subset of instances he can do and subset of instances he cannot do. Now suppose we want to find the ultimate scheduling expert. If the normal expert can solve something, the ultimate expert can do it too, so the set of instances that the ultimate expert can solve is the union of all sets of solvable instances of all possible experts.

In order to find the ultimate expert, we now examine a subset of all possible experts, the experts that can only solve a single instance. If this expert is presented with its instance of expertise, it gives its memorized answer, but if another instance is presented, it says it doesn't know. So, each of these experts has a set of instances it can solve of just one member. Now, if we take the union of the knowledge of all these dumb experts, we get the ultimate dumb expert, who happens to know the answer to any instance of the problem. This is clearly in contradiction with the fact that the problem is intractable, so we must conclude that the assumption that an ultimate expert exists must be false.

The conclusion of this formal exercise is that the there are no ultimate experts for intractable problems. There is always something left to learn, always a new member, or preferably, a set of members that can be added to the set of items that can be solved. But, the reply might be, suppose we incorporate this "learning" in the algorithm. Shouldn't this algorithm be capable of solving any instance of the problem, clearly contradicting the fact that it is intractable? The answer is that a learning algorithm is not an algorithm in the normal sense. A learning algorithm changes after each instance it has or hasn't solved, so it defies the usual analyses of algorithms. A learning algorithm is not a solution to the problem of intractability. However, it can offer explanations for the fact why intractable problems are only mildly problematic for people.

The fact that learning is an essential part of problem solving also shows that the traditional art of task analysis has its limitations. For many problems a task analysis is impossible, because even experts still learn, and use learning to solve problems. The usual idea that at some point an expert knows all there is to learn is not true in general. The same point can be made with respect to linguistics. Viewing language as a static formal structure that must be discovered by linguistic research is like trying to make a task analysis of an intractable problem, so it cannot expose the full extent of language processing.

One of the research approaches to task performance is to get a full account of performance first, and worry about learning later. The previous analysis shows this approach will not work for complex tasks. As models discussed later in this thesis show, task performance is an intricate interplay between learning and performance. Just focussing on performance will only give a very limited insight into what is going on.

If traditional task analysis is an insufficient formal theory of task performance, what should replace it? Architectures of Cognition have the capability. They are formal enough to allow general analyses and making predictions, and they incorporate learning. Instead of focussing on the knowledge of an expert, the focus will be on the learning mechanisms that allow one to become an expert and that allow experts to maintain and adapt their knowledge.

1.6 Overview of the rest of the thesis

The goal of this thesis is to gain more insight into skill-learning, in particular learning of complex problem solving. The way to accomplish this goal is to use a single theory in the form a cognitive architecture, and to start with a single complex problem, the scheduling problem. In chapter 2, the discussion is centered around the topic of the architecture. There are currently four influential architectures of cognition, Soar, ACT-R, EPIC and 3CAPS. I will first establish some general criteria to compare these architectures, after which all four architectures will be discussed.

Human problem solving on the scheduling task, discussed in chapter 3, will turn out to be a puzzle with many pieces. People tend to rehearse and forget things during problem solving. People discover new strategies if old strategies don't work. Some global statistical analysis using multi-level statistics will chart the outlines of the learning process. A detailed protocol analysis will shed some more light on what is going on in the reasoning process.

The approach for chapter 4 to 6 will be to study each of the pieces of the puzzle offered by the experiment using well-known experiments from cognitive psychology. These tasks will be modeled in ACT-R to gain insight into how the particular phenomena relate to the cognitive system as a whole. Chapter 4 will pick up the issue of implicit and explicit learning in general, and rehearsal in particular. ACT-R offers a new type of explanation for the implicit-explicit distinction by removing its Newellian binary status and offering a unifying explanation of an apparent distinction. The bottom line will be that explicit learning can be explained by learning strategies, general knowledge specifically aimed at the acquisition of new knowledge.

Chapter 5 further investigates these learning strategies. It tries to offer a rationale for using a learning strategy, and investigates the representation of learning strategies in terms of ACT-R. The best domain to study learning strategies is developmental psychology. The idea is that learning strategies themselves have to be learned, so the best way to find out more about them is to compare children of different ages. The chapter ends in modeling two particular learning strategies, and seeing whether

they are applicable to multiple problems, and whether any evidence can be found for the fact that the strategies themselves are learned.

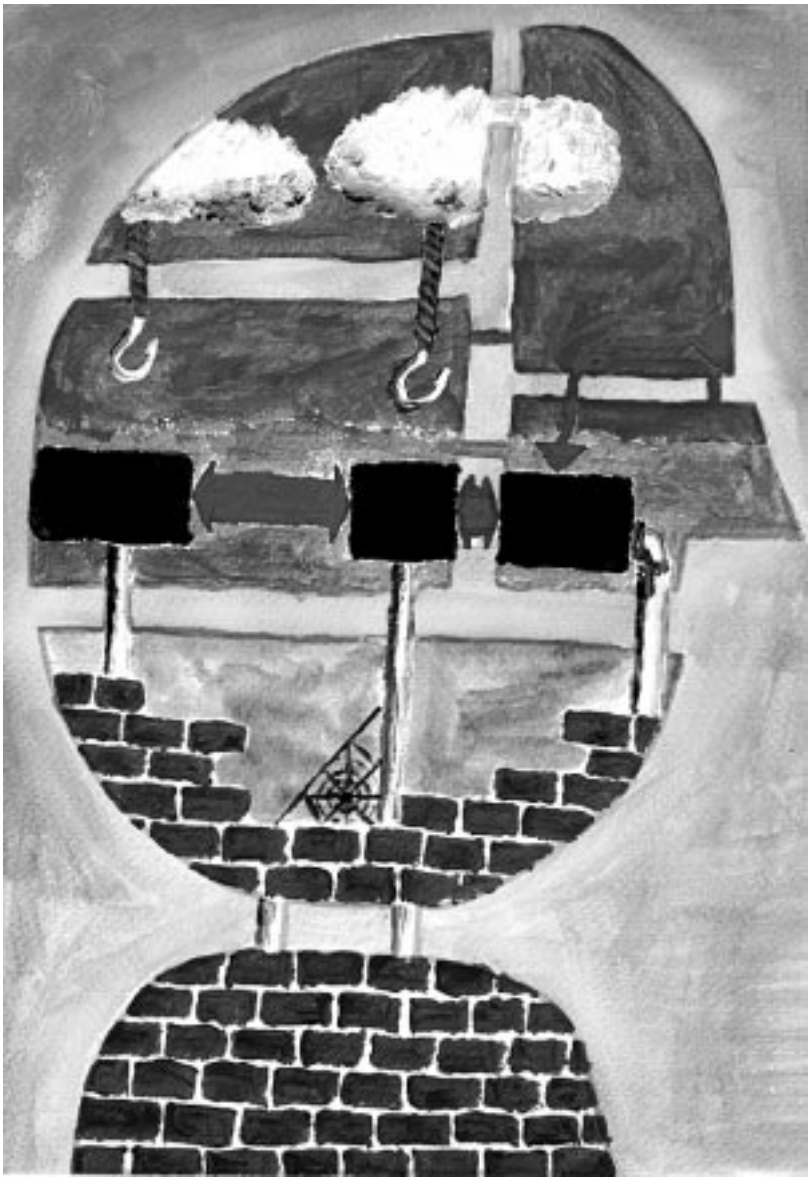
Chapter 6 focuses on another discussion with respect to skill learning, whether skills are learned by generalizing examples into rules, or by just storing and retrieving examples. The answer will turn out to be that both methods are used, and that the impact of these methods on performance depends on how useful they are.

In chapter 7, I return to the primary goal of modeling scheduling. Using all of the insights gained in the smaller projects of chapter 4 to 6, a model will be presented that is able to solve small scheduling problems and learn from this process in a human-like fashion. This model can be used to generate verbal protocols of problem solving, and is able to make some predictions with respect to individual differences.

Chapter 8, finally, is used to draw some conclusions. An overview will be given of the skill-learning theory developed during the thesis, and some applications of this theory are discussed. The usefulness and shortcomings of ACT-R will be discussed. In a sense, the approach used in this thesis will turn out to show close resemblance to the final theory we will arrive at. But this is as it should be, since figuring out how learning in complex problem solving works, is in itself also a form of complex problem solving.

CHAPTER 2

Architectures of Cognition



2.1 What is an architecture of cognition?

Chapter 1 discussed the single system approach to understanding cognition. This chapter will discuss these systems: architectures of cognition. Cognitive science has borrowed the term architecture from computer science. Computer scientists use the term architecture to refer to the aspects of a computer that are relatively fixed: the hardware and that part of the software that is fixed for all applications.

A typical computer architecture has great flexibility: it is capable of executing an infinite variety of programs. However, the architecture can pose constraints on programs. For example, if a computer has a certain amount of memory, it can not run programs that need more memory than is available. The software part of the architecture may also pose constraints. For example, in many time-sharing systems it is impossible to guarantee accurate timing.

Although these limitations may bother many users of computers, they are not interesting for theoretical computer science. In principle, any computer has the same capabilities with respect to what kind of functions it can calculate. This is due to the fact that every computer is equivalent to a universal Turing Machine with respect to the functions it can calculate, except for the fact that a Turing Machine has an infinite memory.

According to the famous Church-Turing thesis (Turing, 1936), a universal Turing Machine can calculate any function that can be calculated at all. A computer architecture is therefore a platform that is ultimately flexible: given the right program, it can calculate any function that is computable in principle, given enough time and memory. The Church-Turing thesis, together with Turing's thought experiment called the Turing Test, can be used to argue that human intelligence can be simulated on a computer (Turing, 1950; Taatgen & Andringa, 1997).

Human cognition is also very flexible. Given enough time, it is capable of learning to perform almost any task that is feasible at all for people. An important distinction between computers and people is that people are not programmed in the sense that computers are. On the other hand, people cannot learn new things out of the blue: they almost always need prior knowledge. For example, one cannot learn to add numbers without knowing what numbers are.

This analogy is the basis for the idea of an architecture of cognition. It is the fixed but versatile basis of cognition. The architecture is capable of performing any cognitive task, regardless of the domain the task is from. But where is a cognitive architecture different from a computer architecture, since a computer architecture is already capable of performing any conceivable task? A first difference is that a computer runs a program, and a cognitive architecture a model. On the surface, a model is a kind of program, written in the language of the cognitive architecture. The difference

is that a program implements an algorithm, an abstract method to solve a problem. A model is not an algorithm, however, although in some cases it may behave like one. Rather, it specifies the prior knowledge the system has. So, if the model tries to explain the behavior of an expert, the knowledge in it may resemble an algorithm, because experts have effective ways of solving problems. If the model tries to explain novice behavior on the other hand, it can only specify general knowledge. A model of a novice has to discover an effective way to do a task itself, by translating instructions into procedures it can carry out, or by discovering these procedures by itself.

Another difference concerns the way a cognitive architecture is designed. In computer science, the architecture is part of the design of a computer. The architecture is the starting point of the computer. Given the architecture, a VLSI-designer can implement the architecture on a chip, and programmers can write an operating system and other software. If you design a better architecture, you get a better computer. Human cognition is already there, so designing an architecture of cognition serves a different purpose. Designing an architecture of cognition is like specifying a theory, a theory of how cognition works. The quality of a cognitive architecture is not measured in terms of performance, but in terms of the power of the theory it implements. This difference in purpose is the same as the difference between artificial and natural languages. An artificial language is defined by its grammar, while a grammar for a natural language is a theory of the structure of that language.

The starting point for the human cognitive architecture is the brain. But many architectures are more abstract than the architecture of the brain. The main point of discussion is whether or not the grain size of individual neurons is proper for formulating a theory of cognition. According to connectionists, properties of individual neurons are crucial for understanding cognitive performance, and an understanding of how neurons cooperate and learn in different areas of the brain will be the most fruitful route to an understanding of cognition in general. Others, often called symbolists, argue that the level of individual neurons is not the right level to study cognition, and some higher-level representation should be used. The title of Anderson & Lebiere's 1998 book *The Atomic Components of Thought* directly refers to this issue. But whatever grain-size we choose, we always abstract away from the biological level of the brain, even if we model neurons in neural networks.

An architecture as a theory

What to expect from a cognitive architecture? Since human cognition is complex, a cognitive architecture will have to be able to make complicated predictions. Analytical methods such as the statistics used by most psychologists can be used to make predictions, but are often limited to linear relationships. Cognition is often non-linear, making analytical mathematical methods infeasible. If analytical methods fail, simulation is the next best method to be able to make predictions.

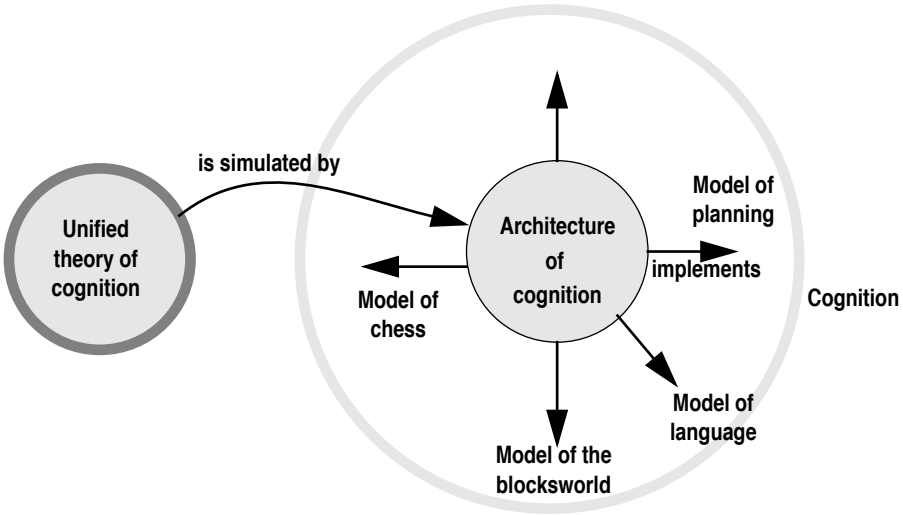


Figure 2.1. Relationship between theory, architecture, models and cognition

Generally, an architecture is an algorithm that simulates a non-linear theory of cognition. This algorithm can be used to make predictions in specific domains and for specific tasks (Figure 2.1).

To be able to make predictions about how people will perform on a specific task, the architecture itself is not enough. Analogous to the computer architecture, where a program is needed to perform tasks, a task model is needed to enable an architecture to simulate something meaningful. Prior knowledge, specified by the model, may be specific to the task, or may be more general. For example, many psychological experiments require the participants to perform some very specific task, such as adding letters as if they were numbers. Such an experiment relies on the fact that participants know how to add numbers and know the order of the alphabet. A model of adding letters would involve knowledge about adding numbers, numbers themselves, letters in the alphabet and knowledge on how to adapt knowledge from one domain to another. It should not incorporate knowledge about adding letters, since it is unreasonable to suppose an average participant in an experiment already has this knowledge. This task-specific knowledge can only be learned during the experiment, or, in the case of the model, during the simulation.

The way task knowledge is merged with the architecture depends on the nature of the architecture. In connectionist theories, all knowledge often has to be learned by a network. To be able to do this, a network has to have a certain topology, some way in which input is fed into the network, and some way to communicate the output. Some types of networks also need some supervisor to provide the network with

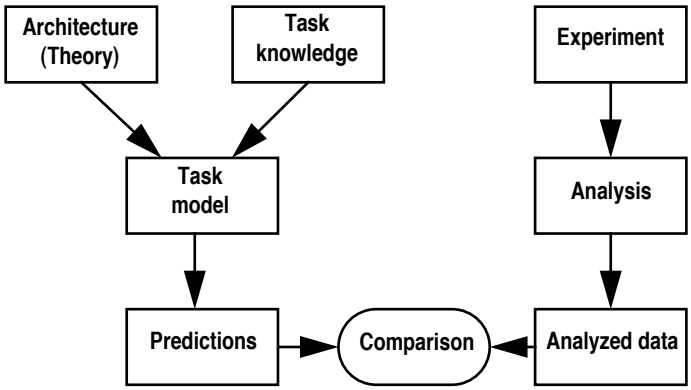


Figure 2.2. Research paradigm in cognitive modeling. Adapted from van Someren, Barnard and Sandberg (1994).

feedback. In neural networks task knowledge is not easy to identify, but is implicit in the environment the network is trained in. In symbolic architectures knowledge is readily identifiable, and consists of the contents of the long-term memory systems the architecture has. Another problem is that it is very hard to give a network any prior knowledge: one always has to start with a system that has no knowledge at all yet. In many cases, this is no problem, but it is in learning complex problem solving, since solving a problem is based to a large extent on prior knowledge.

Regardless of the details, at some point the general theory is combined with task-specific elements to create a task model. A task model is a system that can be used to generate specific predictions about behavior with respect to a certain task. These predictions can be compared to participant data. Figure 2.2 shows the layout of this paradigm. The consequence of this type of research is that the general theory cannot be tested directly. Only the predictions made by task models are tested. If the predictions made by a task model fail to come true, this may be attributed to the architecture, but it may also be attributed to inaccurate task knowledge or the way task knowledge is implemented in the architecture. To be able to judge the achievements of an architecture, there must be some way to generalize over models.

One way to judge the performance of an architecture with respect to a certain task, proposed by Anderson (1993), is to take the best model the architecture can possibly produce for that task. Although this is a convenient way, it is not entirely fair. Suppose we have two architectures, A and B. Given a set of task knowledge, architecture A can only implement a single task model, while architecture B can implement ten task models, nine of which are completely off. Although architecture B may produce the best model, architecture A provides a stronger theory since it only allows for one model.

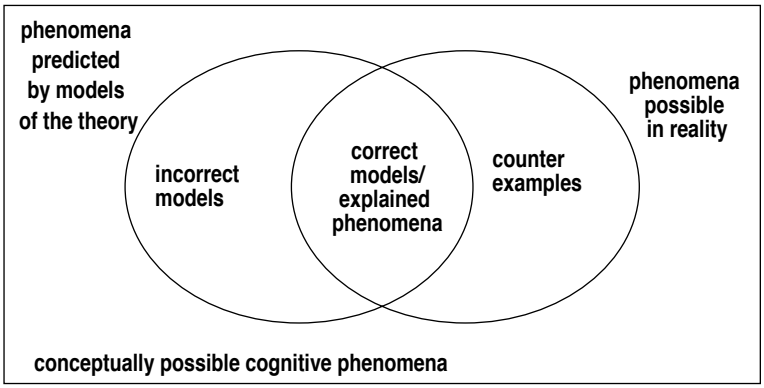


Figure 2.3. Judging an architecture

Judging the success of an architecture

Instead of just focussing on successes, an architecture also has to be judged by its failures. Figure 2.3 shows a schematic impression of this idea, based on Kuipers (Kuipers & Mackor, 1995). Imagine the set of all conceptually possible cognitive phenomena. Not all of these conceivable phenomena can actually be witnessed in reality. For example, in chapter 1 we discussed the power law of practice, but we might also hypothesize a linear law of practice, or a negative exponential law of practice. As a consequence, only a subset of the possible phenomena can actually occur in reality.

When a theory of cognition is proposed, this creates a new subset: the set of phenomena that are predicted by the theory. In terms of an architecture of cognition this means that the architecture allows an infinite set of models, each of which predicts some cognitive phenomena. The union of all these phenomena is the set of cognitive phenomena that are possible according to the theory. In order to judge the quality of the theory, we first have to look at the intersection of the "reality-subset" and the subset predicted by the theory. This intersection represents phenomena that can be predicted by some model, and can actually occur in reality. Although these successes are very important, we also have to look at the failures of the theory. Failures fall into two categories: counter examples, which are phenomena in reality that cannot be predicted, and incorrect models, phenomena predicted by the theory that cannot occur in reality. In the discussion about unified theories of cognition the emphasis is often on the counter examples: are there any phenomena the theory cannot account for? The other category, incorrect models, often gets less attention. This is unfortunate, because incorrect models pose a much bigger problem to architectures of cognition than counter examples.

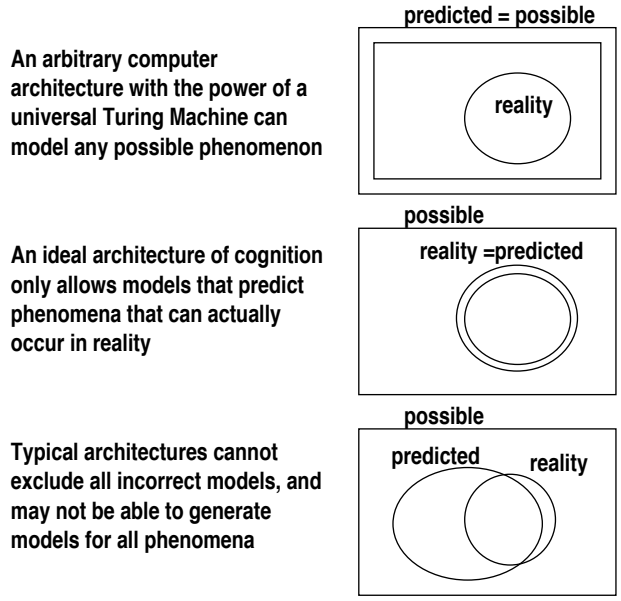


Figure 2.4. Possible instantiations of figure 2.3

The reason why incorrect models are a big problem is due to the Church-Turing thesis mentioned earlier. According to this thesis, any computable function can be computed by a general purpose machine such as the Turing Machine. This implies that, theoretically, any sufficiently powerful computer architecture can implement both all possible correct and all possible incorrect models. Figure 2.4 illustrates this implication: a general purpose architecture can, in principle, model any cognitive phenomenon. In terms of a theory of cognition: an "empty" theory can predict anything. So, the goal of designing a cognitive architecture is not to give it as much features as possible, but rather to constrain a general purpose architecture as much as possible so that it can only implement correct cognitive models. In practice, as shown in figure 2.4, a typical architecture can produce many incorrect models, but generally produces good models. Constraining the general computer architecture may have an undesired side-effect in the sense that phenomena that could previously be explained are now unreachable.

A cognitive theory in the form of an architecture is not a theory in the sense of Popper (1959), but more like a research program in the sense of Lakatos (1970). According to Popper a good theory is a theory that can be refuted. As we have seen, only predictions by models can be refuted directly. Only the claim that an architecture is an ideal architecture, in the sense of figure 2.4, can be refuted by exposing an incorrect model or producing a counter example. In Lakatos's view of science, scientists work in research programs. A research program consists of a set of

core ideas and a paradigm to do research. The core ideas of a research program are generally not disputed within the program, and researchers will continue working within a certain program as long as the paradigm keeps producing encouraging results. In the research program view, the architecture can be viewed as the core idea of a research program. Creating models of cognitive phenomena is part of the research paradigm. Another part of the research paradigm is a methodology to test models. When is a model considered to be a "correct" model?

Matching model predictions with experimental data

To consider a model of a cognitive task as a faithful model of human performance, it is not sufficient that it can perform the task. A model has to perform the task in the same manner as a participant. In order to be able to make this comparison, we have to compare data from an experiment with the output of a model. Ideally, a model produces data that can be directly compared to participant data. Measures that are used often in psychological experiments are reaction times and accuracies. Models should at least be capable of making predictions in terms of these measures. Some architectures, like ACT-R, are capable of making direct predictions about reaction times. Other architectures only indicate a correspondence between steps or cycles in the system and time. In these type of architectures only relative time between different types of problems or trials can be compared to the data. Accuracy is often measured by the rate of correct responses or by the percentage of items recalled. Not all architectures can model all aspects of accuracy. An architecture like Soar, for example, is only interested in errors that result from incomplete or inconsistent knowledge. So errors due to "slips" or forgetting are not considered interesting in the view of the Soar theory.

Since cognitive models give a detailed account of how a task is performed, they make it possible to do more elaborate testing than just reaction times and accuracies. If a trial consists of a number of operations before the response can be given, an attempt can be made to determine the individual latencies of the separate operations, for example by registering eye movement. Reaction times and accuracies tend to change over time, mainly due to learning. The influence of learning can only be disregarded in cases where the task is very simple or the participant is trained exhaustively. Most architectures can account for learning, so should be able to model effects of learning on performance.

The quality of the predictions of a model is often expressed using the R^2 measure, the proportion of variance the model can explain. Suppose we have an experiment that produces n data points, so for example a free-recall experiment in which 20 words can be recalled, we have 20 percentages, one for each of the words, so n=20. The experiment produces data points ($data_i$) that have an average of $\overline{data}$. The model makes a prediction of these data points ($model_i$). The explained variance can now be calculated using the following equation:

$$R^2 = \frac{\sum_{i=1}^{n} (data_i - \overline{data})^2 - \sum_{i=1}^{n} (data_i - model_i)^2}{\sum_{i=1}^{n} (data_i - \overline{data})^2} \quad \textbf{(2.1)}$$

An R^2 of 0.90 or higher is generally considered good, while an R^2 of 0.80 or lower is suspect. In that case there is some source of variance that is left unexplained by the model.

Although the R^2 measure gives a rough estimate of the quality of the model, it does not take into account a number of factors. A first point to consider is the relation between the number of predicted values and the number of parameters that a model uses to make its predictions. If a model needs to tweak 20 parameters in order to be able to predict 20 data points, it is clearly not a good model, regardless of the proportion of variance it can explain. A second point is that this measure only considers the data points from the experiment as averages. As a consequence, any individual differences are discarded. This is no problem if all participants basically behave the same and individual differences are only due to noise that cannot be accounted for. The R^2 measure, however, doesn't capture any systematicity within the behavior of single participants.

One way to take into account that participants differ in their choices is to use a technique called *model tracing*. Anderson, Kushmerick and Lebiere (1993) used model tracing to assess a model of a route planning task. For each individual participant at each point of the problem solving process they compared the choice of the participant to the choice of the model at that point. If both choices agreed they allowed the model to continue to the next step. If there was no agreement, the model was forced to take the same step the participant took. In this particular experiment, it turned out that there was an agreement of 67% between the participant's choice and the model's choice. In 20% of the cases, the participant's choice was the second-best choice of the model. This agreement turned out to be quite good when compared to random-choice and hill-climbing strategies, and to be quite similar to individual differences between participants.

Although model tracing allows the scoring of models in which participants have to make a number of choices in each trial, it still provides no account of individual differences. The model of the task is still a generic model. To really account for individual differences, a generic model must be made that can be instantiated for each individual participant. An example is a model of a working memory task by Lovett, Reder and Lebiere (1997). The model can explain individual differences by varying a single parameter in the generic model.

In summary, a good model is a model that can approximate as many data points as possible using as few parameters as possible. In tasks with large individual differences, a model that can explain individual differences by varying parameters is better than a model that reproduces averages.

2.2 An overview of current architectures

In this section I will review four popular architectures of cognition, all of which have been reasonably successful in modeling various cognitive phenomena. The four architectures to be discussed, Soar, EPIC, 3CAPS and ACT-R, are all either pure symbolic or hybrid architectures. This means all of them share the idea that symbols are the right grain-size to study cognition. However, a pure symbolic theory assumes the underlying neural structure is irrelevant, while a hybrid theory argues that subsymbolic processing plays an important role.

Soar

The Soar (States, Operators, And Reasoning) architecture, developed by Laird, Rosenbloom and Newell (1987; Newell, 1990; Michon & Akyürek, 1992), is a descendant of the General Problem Solver (GPS), developed by Newell and Simon (1963). Human intelligence, according to the Soar theory, is an approximation of a knowledge system. Newell defines a knowledge system as follows (Newell, 1990, page 50):

> A knowledge system is embedded in an external environment, with which it interacts by a set of possible actions. The behavior of the system is the sequence of actions taken in the environment over time. The system has goals about how the environment should be. Internally, the system processes a medium, called knowledge. Its body of knowledge is about its environment, its goals, its actions, and the relations between them. It has a single law of behavior: the system takes actions to attain its goals, using all the knowledge that it has. This law describes the results of how knowledge is processed. The system can obtain new knowledge from external knowledge sources via some of its actions (which can be called perceptual actions). Once knowledge is acquired it is available forever after. The system is a homogeneous body of knowledge, all of which is brought to bear on the determination of its actions. There is no loss of knowledge over time, though of course knowledge can be communicated to other systems.

According to this definition, the single important aspect of intelligence is the fact that a system uses all available knowledge. Errors due to lack of knowledge are no failure of intelligence, but errors due to a failure in using available knowledge are. Both human cognition and the Soar architecture are approximations of an ideal intelligent knowledge system. As a consequence, properties of human cognition that are not

part of the knowledge system approach are not interesting, and are not accounted for by the Soar architecture.

The Soar theory views all intelligent behavior as a form of problem solving. The basis for a knowledge system is therefore the *problem-space computational model* (PSCM), a framework for problem solving based on the weak-method theory discussed in chapter 1. In Soar, all tasks are represented by problem spaces. Performing a certain task corresponds to reaching the goal in a certain problem space. As we have seen in chapter 1, the problem solving approach has a number of problems. To be able to find the goal in a problem space, knowledge is needed about all possible operators, about consequences of operators and about how to choose between operators if there is more than one available. Soar's solution to this problem is to use multiple problem spaces. If a problem, "impasse" in Soar terms, arises due to the fact that certain knowledge is lacking, resolving this impasse automatically becomes the new goal. This new goal becomes a subgoal of the original goal, which means that once the subgoal is achieved, control is returned to the main goal. The subgoal has its own problem space, state and possible set of operators. Whenever the subgoal has been achieved it passes its results to the main goal, thereby resolving the impasse. Learning is also keyed to the subgoaling process: whenever a subgoal has been achieved, new knowledge is added to the knowledge base to prevent the impasse that produced the subgoal from occurring again. So, if an impasse occurs because the consequences of an operator are unknown, and in the subgoal these consequences are subsequently found, knowledge is added to Soar's memory about the consequences of that operator.

In the same sense as the PSCM is a refinement of the idea of a knowledge system, the PSCM itself is further specified at the symbolic architecture level, the Soar architecture itself. Figure 2.5 shows an overview of the architecture, in which buffers and memory systems are represented by boxes, and processes that operate on or between these systems by arrows. Except for sensory and motor buffers, which are not modeled explicitly, Soar has two memory systems: a working memory and a production memory. Working memory is used to store all temporary knowledge needed in the problem solving process. The primary data structure in working memory is the goal stack, which stores all current goals in a hierarchical fashion. Tied to each of the goals on the stack is the current state of the problem space related to that particular goal, and, if present, the current operator.

An example of the goal stack at a particular moment in a particular task is shown in figure 2.6 (Lehman, Lewis, Newell & Pelton, 1991). The task is language comprehension. Each triangle represents a goal with an associated problem space. The small squares, diamonds and circles represent states, and the arrows between them operators. The impasse-subgoal process is represented by the question mark and the dotted arrow to a subgoal. The theory behind this model assumes that sentence comprehension involves reading a sentence word-by-word. During the

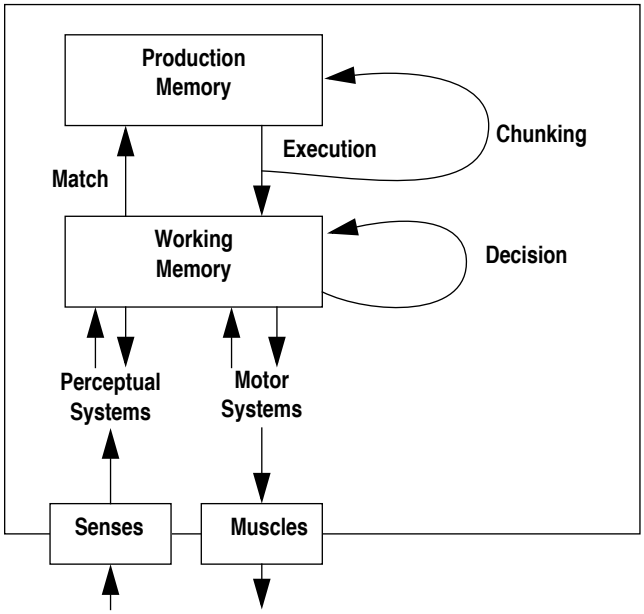


Figure 2.5. Overview of the Soar architecture (from Newell, Rosenbloom & Laird, 1989)

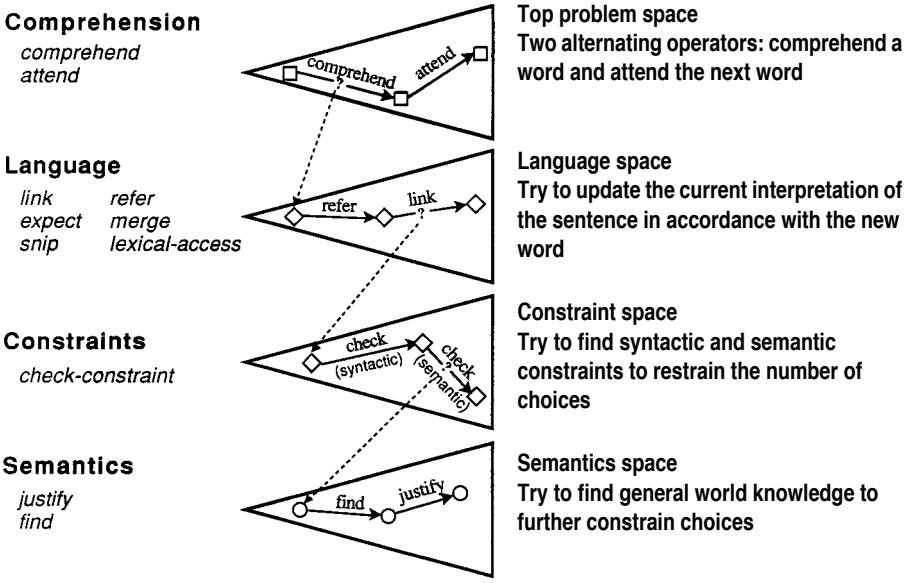


Figure 2.6. Example of the goal stack in Soar in a language comprehension model (from Lehman, Lewis, Newell and Pelton, 1991).

reading process a representation of the meaning of the sentence is assembled. So, at the top problem space, the goal is to comprehend a sentence. This goal is accomplished by alternating two operators: an attend operator, which reads the next word, and a comprehension operator, which augments or updates the current interpretation of the sentence. Comprehending a word is generally not possible in a single step, so after the comprehend operator is selected, an impasse will occur. This impasse generates the language subgoal, which tries to update the current interpretation of a sentence given a new word. The language subgoal has several operators to do this. A word can simply be linked in the interpretation. Sometimes a new word refers to a word read earlier, making it necessary to find the word referred to. In other cases the interpretation built earlier is wrong, and has to be reconstructed. The language space often offers too many choices to link words to each other, so a third subgoal, the constraint goal, is needed to create constraints on the possible linkings. This constraint space uses syntactic and semantic constraints to help making the choice. To find semantic constraints, it is sometimes necessary to use general world knowledge, which is found using the fourth and final subgoal, the semantics goal.

All knowledge needed for problem solving is stored in production memory in the form of rules. Although all knowledge is stored in production rules, they do not have the same active role production rules usually have. A rule in Soar cannot take actions by itself, it may only propose actions. So if Soar is working on a certain goal and is in a certain state, rules may propose operators that may be applied in the current state. Other rules may then evaluate the proposed operators, and may add so-called preferences to them, for example stating that operator A is better than operator B. The real decisions are made by the decision mechanism. The decision mechanism examines the proposals and decides which proposal will be executed. The decision mechanism is actually quite simple. If it is possible to make an easy decision, for example if there is just one proposal or preferences indicate a clear winner, it makes this decision, else it observes an impasse has been reached and creates a subgoal to resolve this impasse. So, the problem of choice in Soar is not handled at the level of individual production rule firings, which are allowed to occur in parallel, but at the level of the proposals of change made by these rules. The learning mechanism in Soar is called chunking.

As mentioned before, learning is keyed to impasses and subgoaling. Whenever a subgoal is popped from the goal stack, Soar creates a new production rule with a generalization of the state before the impasse occurred as the condition, and the results of the subgoal as the action. Dependent on the nature of the impasse, this new rule may propose new operators, create preferences between operators, or implement operators or do other things.

In the language comprehension example discussed earlier learning occurs at all levels of the model. At the level of the comprehension problem space, Soar may learn

a production rule that implements the comprehension operator for a specific word in a specific context. But Soar may also learn a production rule in the constraints problem space to generate a semantic constraint on possible meanings of a sentence.

The knowledge system approach of Soar has a number of consequences. Because not all aspects of human cognition are part of the knowledge system approximation, some aspects will not be part of the Soar theory, although they contribute to human behavior as witnessed in empirical data. Another property of the Soar system is that all choices are deliberate. Soar will never make an arbitrary choice between operators, it either knows which operator is best, or it will try to reason it out. Since intelligence, according to the knowledge system definition, can only involve choosing the optimal operator based on the current knowledge, it does not say much about what the system has to do in the case of insufficient knowledge.

An aspect of human memory that is not modeled in Soar is forgetting. According to the knowledge-system view this is a deviation from ideal intelligence, a weakness of the human mind. This rules out the possibility that forgetting has a function, for example to purge the memory from useless information, allowing for better access to useful information. An error such as choosing a sub-optimal strategy is also considered as aberration of rationality, and is therefore not part of Soar. To sometimes favor a sub-optimal strategy over the optimal strategy may on the other hand have advantages. Maybe one of the sub-optimal strategies has improved due to an increase in knowledge or a change in the environment, and has become the optimal theory. In many situations, the only way to discover how optimal a strategy is, is to just try it sometimes.

Since Soar's behavior deviates from human behavior with respect to aspects that are not considered rational by the Soar theory, the Soar architecture can only make predictions about human behavior in situations where behavior is not too much influenced by "irrational" aspects. Another consequence of the fact that Soar only models rational aspects of behavior is the fact that its predictions are only approximate. An example is Soar's predictions about time. A decision cycle in Soar takes "~~100 ms", where "~~" means "may be off by a factor of 10". So in a typical experiment Soar's predictions have to be determined in terms of the number of decision cycles needed, while the data from the experiment have to be expressed in terms of reaction times. If both types of data show the same characteristics, for example if both show the power law of practice, a claim of correspondence can be made.

One of the strong points of Soar is its parsimony. Soar has a single long-term memory store, the production memory, and a single learning mechanism, chunking. Soar also adheres to a strict symbolic representation. The advantage of parsimony is that it provides a stronger theory. For example, since chunking is the only learning mechanism, and chunking is tied to subgoaling, Soar predicts that no learning will

occur if there are no impasses. In a sense Soar sets an example: if one wants to propose an architecture with two long-term memory stores, one really has to show that it can not be done using just one.

ACT-R

The ACT-R (Adaptive Control of Thought, Rational) theory (Anderson, 1993; Anderson & Lebiere, 1998) rests upon two important components: *rational analysis* (Anderson, 1990) and the distinction between procedural and declarative memory (Anderson, 1976). According to rational analysis, each component of the cognitive architecture is optimized with respect to demands from the environment, given its computational limitations. If we want to know how a particular aspect of the architecture should function, we first have to look at how this aspect can function as optimal as possible in the environment. Anderson (1990) relates this optimality claim to evolution. An example of this principle is the way choice is implemented in ACT-R. Whenever there is a choice between what strategy to use or what memory element to retrieve, ACT-R will take the one that has the highest expected gain, which is the choice that has the lowest expected cost while having the highest expected probability of succeeding.

The principle of rational analysis can also be applied to task knowledge. While evolution shapes the architecture, learning shapes the knowledge and parts of the knowledge acquisition process. Instead of only being focused on acquiring knowledge per se, learning should also aim at finding the right representation. This may imply that learning has to attempt several different ways to represent knowledge, so that the optimal one can be selected.

Both Soar and ACT-R claim to be based on the principles of rationality, although they define rationality differently. In Soar rationality means making optimal use of the available knowledge to attain the goal, while in ACT-R rationality means optimal adaptation to the environment. Not using all the knowledge available is irrational in Soar, although it may be rational in ACT-R if the costs of using all knowledge are too high. On the other hand ACT-R takes into account the fact that its knowledge may be inaccurate, so additional exploration is rational. Soar cannot handle the need for exploration very well, since that would imply that currently available knowledge is not used to its full extent.

The distinction between procedural and declarative memory is studied quite extensively in psychology. Although one should be careful to map distinctions from psychology onto cognitive architectures directly, the best way to explain this distinction is to assume different representations and different memory systems. The disadvantage of this differentiation is that the architecture becomes less simple than an architecture with only a single memory system, like Soar. On the other hand, ACT-R has no separate working memory and instead uses declarative memory in conjunction with an activation concept to store short-term facts. To keep track of the

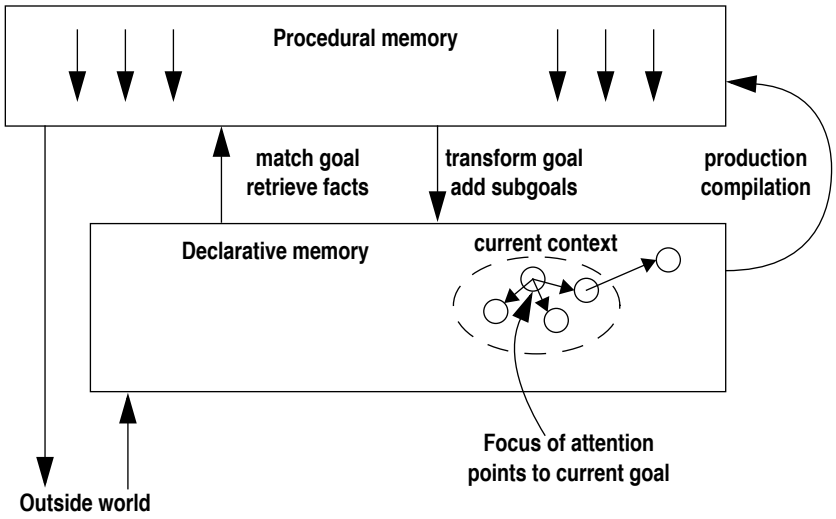


Figure 2.7. Overview of the ACT-R architecture

current context, ACT-R uses a goal stack. The top element of the goal stack is called the focus of attention, a pointer to an element in declarative memory that represents the current goal. New goals can be *pushed* onto the goal stack, and the current goal can be *popped* (removed) from the stack. Figure 2.7 shows an overview of the processes and memory systems of ACT-R. In an appendix to this chapter, some practical aspects of using the ACT-R simulation system will be discussed.

ACT-R's symbolic level

ACT-R comprises two levels of description: a symbolic and a subsymbolic level. On the symbolic level representations in memory are discrete items. Processing at the symbolic level entails the recognize-act cycle typical for production systems, with declarative memory fulfilling the role of working memory. Declarative memory uses so-called chunks to represent information. A chunk stores information in a propositional fashion, and may contain a certain fact, the current or previous goals, as well as perceptual information. An example of a goal chunk, in which two has to be added to six and the answer has not yet been found, is:

```
GOAL23
  ISA ADDITION
  ADDEND1 SIX
  ADDEND2 TWO
  ANSWER NIL
```

In this example, ADDEND1, ADDEND2 and ANSWER are slots in chunk GOAL23, and SIX and TWO are fillers for these slots. SIX and TWO are references to other

chunks in declarative memory. The ANSWER slot has a value of NIL, meaning the answer is not known yet.

Assume that this chunk is the current goal. If ACT-R manages to fill the ANSWER slot and focuses its attention on some other goal, GOAL23 will become part of declarative memory and takes the role of the fact that six plus two equals eight. Later, this fact may be retrieved for subsequent use.

Procedural information is represented in production memory by production rules. A production rule has two main components: the condition-part and the action-part. The condition-part contains patterns that match the current goal and possibly other elements in declarative memory. The action-part can modify slot-values in the goal and can create subgoals (and some other actions we will not discuss in detail here). A rule that tries to solve a subtraction problem by retrieving an addition chunk might look like:

```
IF    the goal is to subtract num2 from num1 and there is no answer
      AND there is a addition chunk num2 plus num3 equals num1
THEN put num3 in the answer-slot of the goal
```

This example also shows an important aspect of production rules, namely variables. *Num1*, *num2* and *num3* are all variables that can be instantiated by any value. So this rule can find the answer to any subtraction problem, if the necessary addition chunk is available.

ACT-R's subsymbolic level

The symbolic level provides the basic building blocks of ACT-R. Using this level only already allows for several interesting models for tasks in which a clearly defined set of rules has to be applied. The symbolic level leaves a number of details unspecified, however. The main topic that it delegates to the subsymbolic level is choice. Choices must be made when there is more than one production rule that can match, or when there is more than one chunk that matches a pattern in a production rule. Other matters that are taken care of by the subsymbolic level are accounts for errors and forgetting, as well as the prediction of latencies.

At the subsymbolic level each rule or chunk has a number of parameters. In the case of chunks, these parameters are used to calculate an estimate of the likelihood that the chunk is needed in the current context. This estimate, called the activation of a chunk, has two components: a *base-level activation* that represents the relevance of the chunk by itself, and *context activation* through association strengths with fillers of the current goal chunk. Figure 2.8 shows an example in the case of the subtraction problem 8-2=?. The fact that eight and two are part of the context increases the probability that chunks associated with eight and two are needed. In this case 2+6=8 will get extra activation through both two and eight. The activation process can be summarized by the following equation:

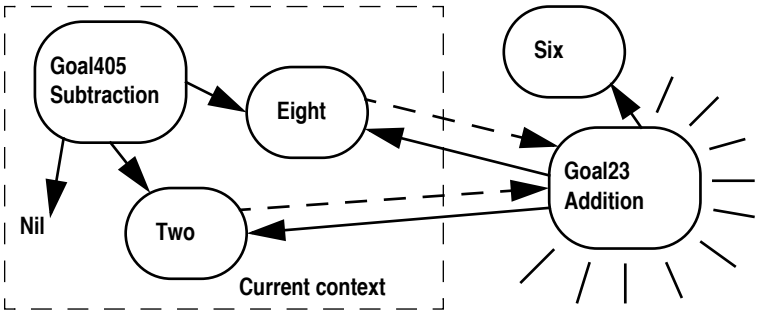


Figure 2.8. Example of spreading activation in ACT-R. The current goal is Goal405, which represents the subtraction problem 8-2=?. The context consists of Goal405 and the numbers eight and two. Context elements are sources of spreading activation. In this example they give extra activation to Goal23, an addition fact that can be used to find the answer to the subtraction. Spreading activation is indicated by dotted arrows.

$$A_i = B_i + \sum_j W_j S_{ji} + \text{noise} \quad \textbf{(2.2)}$$

In this equation, A_i is the total activation of chunk i. This total activation has two parts, a relatively fixed base-level activation (B_i) and a variable part determined by the context (the summation). The summation adds up the influences for each of the elements in the current context. Whether or not a chunk is part of the current context is represented by W_j: if a chunk is part of the context, $W_j=W/n$, otherwise $W_j=0$. n is the total number of chunks in the context and W is some fixed ACT-R parameter which usually has a value of 1. The S_{ji} values represent the association strengths between chunks.

The activation level of a chunk has a number of consequences for its processing. If there is more than one chunk that matches the pattern in a production rule, the chunk with the highest activation is chosen. Differences in activation levels can also lead to mismatches, in which a chunk with a high activation that does not completely match the production rule is selected. Such a chunk can be matched anyway, at an activation penalty, by a process called partial matching. Finally activation plays a role in latency: the lower the activation of a chunk is, the longer it takes to retrieve it. This retrieval time is calculated using the following equation:

$$\text{Time}_{ip} = Fe^{-f(A_i + S_p)} \quad \textbf{(2.3)}$$

Since a chunk is always retrieved by a production rule, this equation expresses the time to retrieve chunk i by production rule p. Besides the activation of the chunk, the strength of the production rule (S_p) also plays a role. The F and f parameters are fixed ACT-R parameters, both of which default to 1. As the sum of activation and

strength decreases, the time to retrieve a chunk grows exponentially. To avoid retrieval times that exceed the order of a second, a retrieval threshold is defined. Chunks with an activation value below the threshold cannot be retrieved.

Choices between production rules are determined by estimates of their expected gain. To be able to calculate the expected gain of a certain rule, several parameters are used to make an estimate. The main equation that governs this estimate is:

$$\text{Estimated Gain for production } p = P_p G - C_p \quad \textbf{(2.4)}$$

In this equation P_p is the estimated probability of reaching the goal using production rule *p*, *G* is the value of the goal, and C_p the estimated cost of reaching the goal using *p*. The unit of cost in ACT-R is time. Suppose we are willing to spend 10 seconds on a certain goal (*G*=10), and suppose there are two production rules *p1* and *p2*, and *p1* reaches the goal 60% of the time ($P_{p1} = 0.6$) in 2 seconds on average ($C_{p1} = 2$). Similarly, $P_{p2} = 0.8$ and $C_{p2} = 5$. In that case the expected gain of *p1* is 4, and the expected gain of *p2* is 3. So, *p1* is selected in favor of *p2*, since its expected gain is higher. To be able to estimate all these values, ACT-R maintains a number of parameters with each production rule. Besides parameters to calculate the expected gain, production rules also have a strength parameter, comparable to activation of chunks. The strength parameter is another component that determines the latency of firing a production: productions with a higher strength take less time to match (equation 2.3).

Learning in ACT-R

While ACT-R has two distinct memory systems with two levels of description, distinct learning mechanisms are proposed to account for the knowledge that is represented as well as for its parameters. At the symbolic level, learning mechanisms specify how new chunks and rules are added to declarative and procedural memory. At the subsymbolic level, learning mechanisms change the values of the parameters. Objects are never removed from memory, although they may become virtually irretrievable.

A new chunk in declarative memory has two possible sources: it either is a perceptual object, or a chunk created internally by the goal processing of ACT-R itself. ACT-R's internally created chunks are always old goals, as exemplified by the ADDITION-goal discussed earlier. Any chunk in declarative memory that has not originated from perception has once been the current goal in ACT-R.

Learning new production rules is a more intricate process. Production rules are learned from examples. These examples are structured in the form of a dependency chunk. A dependency is a special type of chunk, which points to all the necessary components needed to assemble a production rule. Figure 2.9 shows the dependency structure necessary for the subtraction rule. In this example, three slots

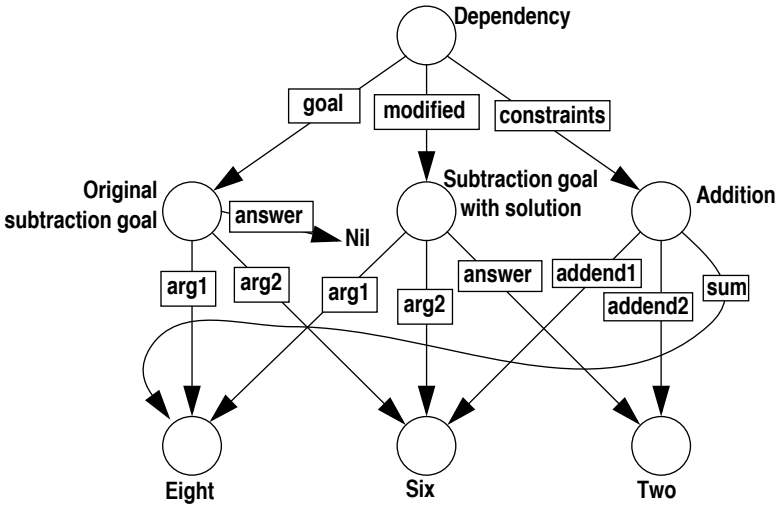


Figure 2.9. Example of the declarative structure needed to learn the subtraction production. In this case, the dependency has three filled slots: the original goal, the modified goal in which the answer slot is filled, and a constraint, an old addition goal that was used to calculate the answer

of the dependency are filled: the goal slot contains an example of a goal in which the answer slot is still empty (nil), and the modified slot has an example of the same goal, but now with its answer slot filled. The constraints slot contains the fact that has been used to transform the original goal into the modified goal. Since a dependency is a chunk that obviously is not a perceptual chunk, it must be an old goal. In order to learn a new rule, a dependency goal must be pushed onto the goal-stack. After processing, the dependency is popped and the production compilation mechanism (in former versions of ACT-R called analogy) generalizes the dependency to a production rule. This scheme for production rule learning has two important properties: it is dependent on declarative memory, and assembling a rule is a goal-driven process.

Since the parameters at the subsymbolic level estimate properties of certain knowledge elements, learning at this level is aimed at adjusting the estimates in the light of experience. The general principle guiding these estimates is the well known Bayes' Theorem (Berger, 1985). According to this principle, a new estimate for a parameter is based on its prior value and the current experience.

The base-level activation of a chunk estimates the probability that it is needed, regardless of the current context. If a chunk was retrieved a number of times in the immediate past, the probability that it will be needed again is relatively high. If a chunk has not been retrieved for a long time, the probability that it will be needed now is only small. So, each time a chunk is retrieved, its base-level activation should

go up, and each time it is not used, it should go down. This is exactly what the base-level learning mechanism does: it increases the base-level activation of a chunk each time it is retrieved, and causes it to decay over time. The following equation calculates the base-level activation of a chunk:

$$B_i(t) = \log \sum_{j=1}^{n} (t - t_j)^{-d} \quad \textbf{(2.5)}$$

In this formula, n is the number of times a chunk has been retrieved from memory, and t_j represents the time at which each of these retrievals took place. So, the longer ago a retrieval was, the less it contributes to the activation. d is a fixed ACT-R parameter that represents the decay of base-level activation in declarative memory (default value=0.5).

The other parameters are estimated in a similar fashion. For example, the probability of success of a production rule goes up each time it leads successfully to a goal, and goes down each time the rule leads to failure.

EPIC

Soar and ACT-R focus on central cognition. The EPIC (Executive-Process Interactive Control) architecture (Meyer & Kieras, 1997) instead stresses the importance of peripheral cognition as a factor that determines task performance. This stress on peripheral cognition is immediately apparent in the overview of EPIC in figure 2.10. Except for the cognitive processor with its associated memory systems, the main focus of the other three architectures discussed in this chapter, EPIC provides a set of detailed perceptual and motor processors. In order to study the role of perceptual and motor processors, it is also necessary to simulate a highly detailed task environment. The perceptual modules are capable of processing stimuli from simulated sensory organs, sending their outputs to working memory. They operate asynchronously, and the time they require to process an input depends on the modality, intensity and discriminability of the stimulus. The time requirements of the perceptual modules, as well as other modules, are relatively fixed, and serve as an important source of constraints.

EPIC's cognitive processor is a parallel matcher: in each cycle, which takes 50 ms, production rules are matched to the contents of working memory. Each rule that matches is allowed to fire, so there is no conflict resolution. It is up to the modeler to prevent this parallel firing scheme from doing the wrong thing. Whereas both Soar and ACT-R have a production firing system that involves both parallel and serial aspects, EPIC has a pure parallel system of central cognition. As a consequence, EPIC predicts that serial aspects of behavior are mainly due to communication between central and peripheral processors and structural limitations of sense organs and muscles. Corresponding to this idea that processing bottlenecks are located in the

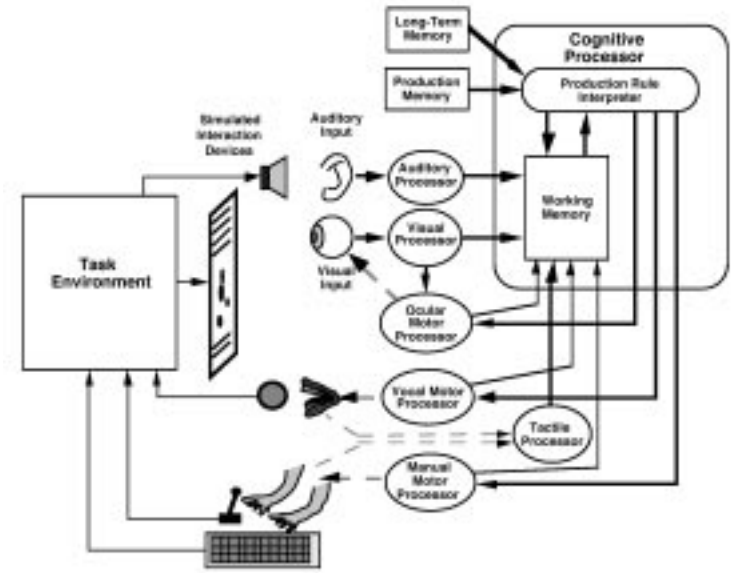


Figure 2.10. Overview of the EPIC architecture (from Meyer & Kieras, 1997)

periphery, EPIC has no goal stack in the sense of Soar and ACT-R. EPIC can represent multiple goals in a non-hierarchical fashion, and these goals can be worked on in parallel, provided they do not need the same peripheral resources. If they do, as is the case in experiments where participants have to perform multiple tasks simultaneously, executive processes are needed to coordinate which of the goals belonging to the tasks may access what peripheral processors. Because EPIC's executive processes are implemented by production rules, they do not form a separate part of the system. EPIC's motor processors coordinate muscle commands. Movements are carried out in two phases: movement preparations and movement execution. During the execution of a movement the next movement can be prepared.

An important aspect of EPIC's modular structure is the fact that all processors can work in parallel. Once the cognitive processor has issued a command to the ocular motor processor to direct attention to a spot, it does not have to wait until the visual processor has processed a new image. Instead, it can do something else. In a dual-task setting the cognitive processor may use this extra time to do processing on the secondary task. Although all the possibilities for parallel processing increase the flexibility of the architecture, it doesn't offer many constraints. The modeler has a choice between creating a very clever parallel model and a pure serial model of a task by providing other executive production rules. This can only be justified if it can be shown that participants exhibit both types of behavior. In a sense, what was one of the virtues of Soar is one of the vices of EPIC: its lack of parsimony. Another drawback of EPIC as a cognitive modeling tool, is that it does not incorporate

learning. As has been discussed in chapter 1, it can be doubted whether information processing and learning can be studied separately.

3CAPS

While EPIC proposes that most constraints posed on the architecture are due to structural limitations of sense organs and muscles, 3CAPS (Just & Carpenter, 1992) proposes limitations on working-memory capacity as the main source of constraints. 3CAPS has a working memory, a declarative memory and a procedural memory. As in ACT-R, memory elements have activation values. As long as the activation of an element is above a certain threshold, it is considered part of working memory and can be accessed by production rules. Capacity theory, 3CAPS's foundation, specifies that a certain amount of activation units is available. These activation units can be used to either keep elements active in working memory or to propagate activation by firing production rules. If the maximum amount is reached, both working memory maintenance and production firing processes get less activation units than they need. The result of activation deprivation for working memory is that memory elements may be forgotten prematurely. If processing gets less activation than needed, production rules have to fire multiple times to get the activations of target working memory elements above the threshold, effectively slowing it down.

The 3CAPS theory views the limitation in activation units as a source of individual differences. It has been successful in explaining individual differences in language comprehension, relating performance differences in reading and comprehending sentences to working memory span (Just & Carpenter, 1992). A limitation of 3CAPS is that it does not incorporate learning.

A summary of the four architectures

Figure 2.11 summarizes the properties of the four architectures discussed in this section. Each of the architectures has its own central theory, and its own roots. Most of the architectures settle on two long-term memory stores, one for procedures and one for facts. All of them have some form of working memory, although in the case of ACT-R this is only a goal stack with pointers to declarative memory. Both ACT-R and 3CAPS have an activation-based mechanism to represent availability of memory elements. Although the mechanisms behind them differ, they share some characteristics. 3CAPS poses a strict activation limit. The consequence of exceeding the limit is forgetting and longer reaction times. These consequences, however, also concur with ACT-R's effects of low activation. If the current context in ACT-R contains many elements, spreading activation is divided over all these elements, resulting in lower activation of associated elements. Although there is no explicit activation limit in ACT-R, thinning out activation may lead to a sudden decrease in performance when elements drop below the retrieval threshold. At least the predictions of both mechanisms are roughly equivalent, although it may turn out

	Soar	ACT-R	EPIC	CAPS
Central theory	Problem solving	Rational Analysis	Embedded cognition	Capacity theory
Roots	Artificial Intelligence	Cognitive Psychology	Human-Computer Interaction	Language Processing
Type	Symbolic	Hybrid	Symbolic (central cognition)	Hybrid
Learning	yes	yes	no	no
LTM systems	1 (Productions)	2 (Productions and Facts)	2 (Productions and Facts)	2 (Productions and Facts)
STM systems	Working memory	Goal stack	Working memory, several sensory stores	Limited capacity working memory
Detailed latency predictions	no	yes	yes	yes
Parallel production firing	yes	no	yes	yes
Main source of constraints	Single LTM, single learning mechanism	Small production rules, principle of rationality	Peripheral modules	Limited capacity
Parsimony	++	+/-	-	+/-
Peripheral cognition	no	extension (ACT-R/PM)	yes	no

Figure 2.11. Comparison between architectures

that they differ in subtle aspects. Not all architectures encompass learning and peripheral cognition. Only ACT-R models both, although peripheral cognition only in a recent extension (ACT-R/PM). This extension borrows many ideas from EPIC. Architectures tend to seek constraints in an area that is related to the central theory, and leave other areas unconstrained. Probably all the architectures still have too few constraints.

2.3 Neural network architectures

As of yet, there are no general neural network architectures. The four architectures discussed are either purely symbolic or hybrid. The hybrid architectures borrow some ideas from neural networks in order to calculate activation levels and other parameters, but have a symbolic production system engine as main processor.

Lebiere & Anderson (1993) have developed a neural network implementation of ACT-R. This implementation proved to be a useful exercise, since it offered additional constraints to ACT-R. One of the changes made to ACT-R due to the constraints posed by the neural network implementation is the fact that only goals are matched in parallel, and any remaining matches have to be done serially. This is curious, since other architectures, most notably 3CAPS, claim parallel matching is a "neurally inspired" feature. But a "true" neural network architecture cannot be an implementation of a symbolic architecture, since according to connectionists the level of subsymbolic elements is the right level of abstraction to study cognition. Before a "true" neural network architecture of cognition can be developed, a number of problems has to be solved.

A first problem is the *binding problem*. In a symbolic architecture it is easy to create a temporary binding between a variable in a production rule and elements in working memory. In neural networks this is much harder. The simplest way to create a temporary link between two structures is to activate a connection between the two. But allowing for connections between arbitrary concepts requires an infeasibly large number of connections. An alternative to a direct connection is to represent a temporary connection between two concepts by a synchronous activation pattern. In that way arbitrary concepts can be combined without the need for a physical connection between them. The rest of the neural architecture has to be designed to handle this kind of representation, of course, producing networks with a totally different topology from what is currently used in neural network research. Shastri & Ajjanagadde (1993) designed a network based on this idea, which is capable of representing both short-term and long-term facts, and which has the ability to reason with those facts.

A second problem is *stability*. Neural networks are famous for their capacity to learn. Maintaining this knowledge is harder though. If a standard three-layer network is trained on a certain set of data, and new information is added, the old information is forgotten, unless special care is taken to present new information along with old information. Since we cannot count on the outside world to orchestrate presentation of new and old information in the way the network would like it, McClelland hypothesizes this is a function of the hippocampus. Another solution is to design networks that are not as susceptible to forgetting as the standard networks. Grossberg's (Carpenter & Grossberg, 1991) ART-networks are an example of this idea. An ART network first matches a new input with stored patterns. If the new

input resembles one of the stored patterns closely enough, learning allows the new input to interact with the stored pattern, possibly changing it due to learning. If a new input does not resemble any stored pattern, a new node is created to accumulate knowledge on a new category. In this way, new classes of input patterns do not interfere with established concepts.

A third problem is *serial behavior*. Many cognitive tasks, most notably problem solving, require more than one step to be performed. In order to do this, some control structure must be set up to store intermediate results. Recurrent networks (see, for example, Elman 1993) have this capability in some sense. A recurrent network is a three layer network with an additional "state" that feeds back to the hidden layer of the network. If several inputs are presented in a sequence, for example in processing a sentence, this state can be used to store temporary results.

Although solutions have been found for each of the roadblocks to a fully functional neural architecture of cognition, these solutions do not add up (yet). Notably solutions to the binding problem demand radical changes in the architecture of a neural network, requiring new solutions to the other problems as well. But the fact that the brain is built out of neurons promises that there is a solution to all of the problems. But the debate on what the right grain-size of studying cognition is, has not ended yet.

2.4 Machine learning

All knowledge in the long-term memory stores of an architecture is somehow acquired at some point in time, unless it is inborn. Since only Soar and ACT-R model learning, the other architectures can not even address this issue. A model of a task that fully addresses the issue of learnability starts with a body of knowledge that is not specifically tailored for the task, but is a set of general problem solving methods and a large database of facts. Given the task instructions, it should be able to learn some initial task-specific knowledge, which is refined during practice. Both Soar and ACT-R provide the tools to do this in the form of learning mechanisms. But these mechanisms must be applied within a context of prior knowledge to be able to get a complete picture of learning.

The general problem of how to extract knowledge from examples, instruction and experience is studied in *machine learning*, a subdiscipline of artificial intelligence. Although machine learning is not primarily aimed at human cognition, it can give an overview of available methods. The task a machine learning algorithm has to carry out is often described as concept learning: given some examples of a concept and sometimes some prior knowledge, derive a knowledge representation of the

concept. A representation of a concept can be used to decide whether some new example is an example of the concept or not.

Carbonell (1990) distinguishes four machine learning paradigms: the inductive, analytic, genetic and connectionist paradigm. The *inductive paradigm* assumes a concept has to be derived from a set of examples. Examples can be positive (x is an example of the concept) or negative (y is not an example of the concept). The goal of an inductive machine learning algorithm is to find a generalization that covers all the positive examples, but excludes all negative ones. This generalization is based purely on the features of the examples themselves, and not on any other knowledge. The *analytic paradigm* has the opposite assumption that there is a rich and complete domain theory, from which the concept can be derived in principle. But since deriving things from the domain theory must be guided by some utility aspect, examples are used as a catalyst. In the analytic paradigm often only a single example is needed to create a concept description.

To take an example, suppose the concept of a swan has to be derived by an inductive paradigm. This paradigm would require a set of examples, consisting of swans and non-swans. Suppose this set contains three examples, a large white swan with a yellow beak, a large white swan with an orange beak, a small white duck with a yellow beak. Possible characterizations of a swan in this case are: large, or large and white, since both of these characterizations include both swans and exclude the duck. An analytic algorithm works in another way. It supposes we show some object to a reasoning system and ask it whether or not it is a swan. Suppose the object has the following properties: wings, white, large, orange beak, lays eggs, flies. Now the reasoning system needs to have knowledge to answer this question. It knows, for example that a swan is a large white bird that birds have wings, can fly and lay eggs. It also knows that airplanes may be white and large too, and are also able to fly. After some deduction, it may conclude that the object is indeed a swan. The analytic algorithm may now learn a new rule about swans: if the object is large, white, flies and lays eggs, it is a swan. The orange beak is not important, since it has not contributed to the decision, and the fact that the swan has wings is ignored because it is implied by the fact that it can fly.

Both the genetic and the connectionist paradigm can be seen as special cases of the inductive paradigm, since both try to generalize concepts solely using examples. But each of these approaches has grown into a separate research community. The *genetic paradigm* assumes that the choice of whether or not knowledge should be learned is based on utility instead of truth. This idea is not unique for the genetic approach, since the utility of knowledge is also central in ACT-R. The assumption the genetic approach makes, is that the mechanisms for determining the utility of a certain knowledge element are the same as the mechanisms nature uses to determine the utility of a certain organism that new knowledge is derived in the same fashion as new organisms are conceived. In genetic algorithms knowledge is represented by

strings of symbols with a fixed length and alphabet. Usually a genetic algorithm starts with a set of random strings, the initial population. For each of these strings a fitness value is determined, a value that estimates the utility of the knowledge coded by the string. Subsequently a new generation is calculated. Candidate member for the new generation are selected from the old generation using a randomization process that favors strings with a high fitness value. The new candidates are then subjected to a mutation process: some pairs of strings are mutated by a cross-over operator that randomly snaps each string in two pieces and exchanges the two pieces between strings. Other strings are mutated by a point-mutation operator that randomly changes a single token in the string. This new generation of strings is subjected to the same procedure. The idea is that the average fitness increases with each new population. To prove this idea, Holland (1975) derived the schema theorem. This theorem shows that fragments of a string (called schemas) that contribute to its overall fitness have a higher than average chance to appear in the new population, while schemas that do not contribute will gradually disappear. Consequently, in the long term the schemas with the highest fitness will survive.

The *connectionist paradigm*, although it has many flavors, can also be considered as a form of inductive learning. Take for example the popular three-layer feed-forward networks. In these networks an input is fed into the input layer of a network, which is connected to a hidden layer. The hidden layer is connected to an output layer that makes a final classification. After a classification has been made, the backpropagation algorithm is used to change the connection weights in the network based on the discrepancy between the required output and the actual output. Links which, on average, contribute to successful classifications will be strengthened, while links that do not contribute to success will be weakened. Cells in the hidden layer will often be feature-detectors, a role that shows close resemblance to Holland's schemata.

If one looks at the different paradigms, it is apparent that there is a difference in the number of examples the algorithms need before a reasonable successful generalization can be made. While an analytical algorithm sometimes only needs a single example, the connectionist and genetic algorithms often need thousands of trials before they converge. An analytical algorithm on the other hand needs to do a lot of processing and requires background knowledge to arrive at a generalization. New knowledge is often logically deduced from old knowledge, ensuring that if the domain knowledge is correct, the newly derived knowledge is also correct. This distinction is more like a dimension, since algorithms can be conceived of that use both domain knowledge and some generalization based on examining multiple examples. We will call this dimension the *rational-empirical dimension*.

Another issue in machine learning that is often left implicit, is the goal of learning. Sometimes learning is aimed at obtaining new knowledge. For example, if a neural network learns to discriminate letters on the basis of features or pixel patterns, it has learned new concepts. But learning can also be aimed at speeding up an existing

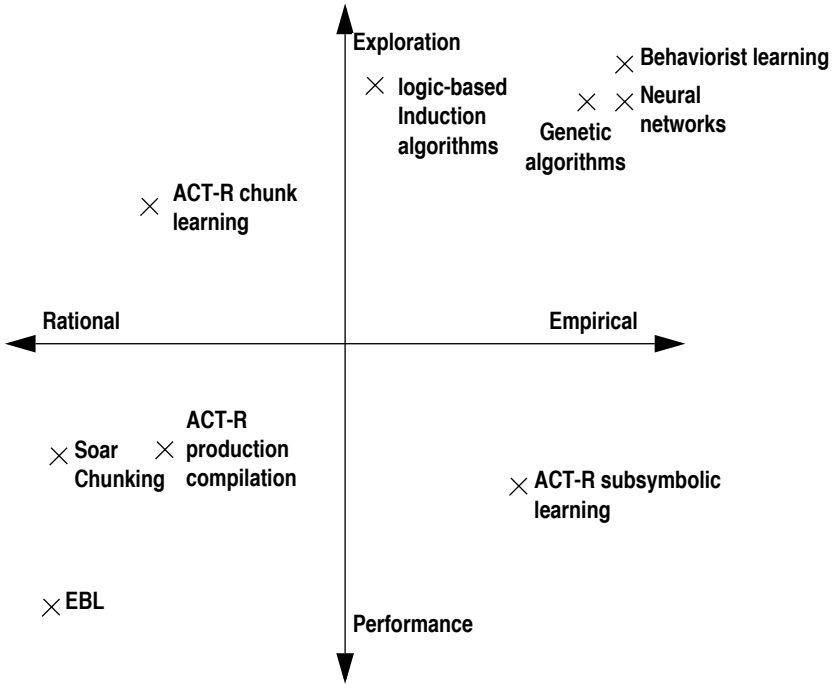


Figure 2.12. Learning algorithms and theories shown on the exploration-performance dimension and on the rational-empirical dimension.

skill, by compiling knowledge into more efficient representations. This second goal of learning is also very important in human learning, and is in general described by the power law of practice, as discussed in chapter 1. Speedup and new knowledge are not always separate goals. As is also discussed in chapter 1, a speedup in processing may make some instances of problems tractable that were previously intractable. In that case speedup opens the road to new knowledge. So this second distinction can also be seen as a dimension, which we will call the *exploration-performance dimension*.

While machine learning algorithms often take extreme positions on both dimensions, human learning has to be both rational and empirical, and aimed at both performance and exploration. Figure 2.12 shows how some current learning algorithms and theories can be positioned on the two learning dimensions. Induction algorithms tend to be aimed at exploration. The inductive algorithms based on logic often use some sort of inference to arrive at the best solution given a set of examples. So this kind of algorithm is rational in Newell's definition, in the sense that they use the available knowledge as rationally as possible, but also empirical, since they use multiple examples. Genetic algorithms and neural networks lack a rational component, and derive their generalizations from principles

inspired by genetics and neuroscience. Behaviorist principles of learning can also be found in this area: they are strictly empirical, and are not interested in performance. This may well be one of the reasons why connectionists are sometimes falsely accused of being a new breed of behaviorists. Analytical algorithms, exemplified by explanation-based learning (EBL), are on the opposite side of the figure. EBL is strictly rational in the sense that all new knowledge is specialized domain knowledge, and is based on a single example. As a consequence it can not gather any new knowledge. Soar's chunking mechanism resembles EBL in the sense that learning is based on a single example, summarizing processing in a subgoal, and its stress on rationality.

ACT-R's learning mechanisms are harder to classify, since they cannot really be considered as learning algorithms. So their positions in the diagram are approximate. The chunk learning mechanism refers to the fact that ACT-R stores past goals in declarative memory. This may serve several functions. An old goal may just help to increase performance, for example of the fact that three plus four equals seven is memorized as a result of counting it. But a chunk may also contain information gathered from the outside world, or may contain a hypothesis posed by reasoning. If exploration is considered to be a function that proposes a new knowledge element as something that may be potentially useful, the chunk-learning mechanism is more an exploration mechanism than a performance increasing mechanism. Since new chunks are single examples, and are based on reasoning, they are more a product of rational than empirical learning. The empirical aspect of learning is covered by ACT-R's subsymbolic learning mechanisms. By examining many examples, ACT-R can estimate the various parameters associated with chunks and productions. But contrary to other subsymbolic learning algorithms, parameter learning is mainly aimed at performance increase. A higher activation allows quicker retrieval of a declarative chunk, and a better estimate of expected-gain parameters allows for more accurate strategy choices. In order to compile a new production in ACT-R, a detailed example must be available in the form of a dependency structure. Although production compilation can be used in any possible fashion, it is not feasible to create large amounts of production rules that contain uncertain knowledge. So the most likely role of production compilation is to increase the efficiency of established knowledge.

Although we have discussed ACT-R's mechanisms separately, they usually work in concert. So some new knowledge may initially be learned as a chunk. The parameters of this chunk may be learned by parameter learning. If parameter learning has established that the chunk serves an intermediate function in a problem solving step, it may be transformed into a production rule. So although ACT-R's learning mechanisms are not fully fledged learning algorithms, they have the capability, in principle, to cover the whole spectrum of learning means and goals. In later chapters I will show how these primitive learning mechanisms can serve as building blocks for a theory of skill learning.

2.5 Conclusions

For the purposes of this thesis, accurate modeling of learning processes in complex problem solving, the ACT-R architecture turns out to be the clear winner with respect to the comparisons made in this chapter. Neural networks first have to solve a number of problems before they can achieve the architecture stadium, and 3CAPS and EPIC do not encompass learning. Although Soar supports learning, it is rigid in the sense that it is mainly aimed at performance increases, and gaining new knowledge is hard to model. Soar's theoretical assumptions are the main problem: by defining intelligence as using available knowledge, it discounts the importance of gaining new knowledge, and by ignoring performance aspects of behavior it makes detailed predictions of behavior impossible. When the learning mechanisms of ACT-R are examined in the context of machine learning, it turns out that they can in principle cover the whole spectrum of learning.

Although ACT-R is the vehicle I will use in the rest of this thesis, some of Soar's ideas will resurface. The idea to key learning to impasses in problem solving is not only rational in the Soar sense, but also, as we will see in chapter 5, in ACT-R's.

2.6 Appendix: The ACT-R simulation system

The ACT-R simulation system is a program written in Common Lisp. The basic version is based on a command-line interface in Lisp. Typically, a user loads Common Lisp, loads ACT-R and starts working on a model. A model in ACT-R, which is just a text file, usually consists of four areas: global parameter declarations, the contents of declarative memory, the contents of procedural memory and lisp-code to run the particular experiment.

There are two types of declarations for declarative memory: the specification of the chunk types and the initial contents of declarative memory. Although chunk types do not change during the execution of a model, the contents of declarative memory almost always does, since all the goals and subgoals ACT-R creates are added to it. In some models, a specification is added that gives the initial chunks an activation value that differs from the default value 0, for example to reflect that it is a chunk that has been in declarative memory for a long time. The next part of a model is an initial set of production rules. Sometimes initial parameters are specified for these rules. Finally some code is added to run an experiment, and to store results.

```
; Very simple ACT-R example model

; Parameter declarations: switch on
rational analysis and set Activation Noise
to 0.1
(sgp :era t :ans 0.1)

; chunk-type declarations
(chunk-type addition-problem arg1 arg2
answer)
(chunk-type addition-fact addend1 addend2
sum)

; initial contents of declarative memory

(add-dm
   (fact34
      isa addition-fact
      addend1 3
      addend2 4
      sum 7)
   (fact42
      isa addition-fact
      addend1 4
      addend2 2
      sum 6))

; contents of production memory

(p do-addition
   =goal>
      isa addition-problem
      arg1 =num1
      arg2 =num2
      answer nil
   =fact>
      isa addition-fact
      addend1 =num1
      addend2 =num2
      sum =num3
==>
   =goal>
      answer =num3)

(p do-addition-fail
   =goal>
      isa addition-problem
      answer nil
==>
   =goal>
      answer dont-know)

; Parameter declaration for do-addition-
fail

(spp do-addition-fail :r 0.2)

; Lisp code to run sample experiment

(defun do-it (n)
   (let ((result 0))
      (dotimes (i n)
      (let ((task (gentemp "goal")))
      (eval `(add-dm
         (,task isa addition-problem
            arg1 ,(random 5)
            arg2 ,(random 5))))
      (eval `(goal-focus ,task))
      (run 1)
      (when
         (equal (+
         (eval `(chunk-slot-value ,task
arg1))
         (eval `(chunk-slot-value ,task
arg2)))
         (eval `(chunk-slot-value ,task
answer)))
         (setf result (1+ result)))
               (pop-goal)))
   (format t "~%Accuracy = ~6,3F" (/
result n))))
```

Figure 2.13. Example ACT-R model

Figure 2.13 show an example of a very small model, a model that tries to solve an addition-problem. It knows only two addition-facts: 3+4=7 and 4+2=6. Whenever it tries to solve an addition-problem, two rules are applicable: the do-addition rule that tries to retrieve a matching addition-fact and the do-addition-fail rule that give "don't know" as an answer. The parameter declaration for the do-addition-fail rule makes sure that its expected gain is lower than the expected gain of the do-addition rule. ACT-R will therefore first try do-addition, and only when that rule fails will do-addition-fail be allowed to fire.

The Lisp code consists of a function that goes through *n* addition-problems. It generates a random addition-problem, which is given to the model. After one of the

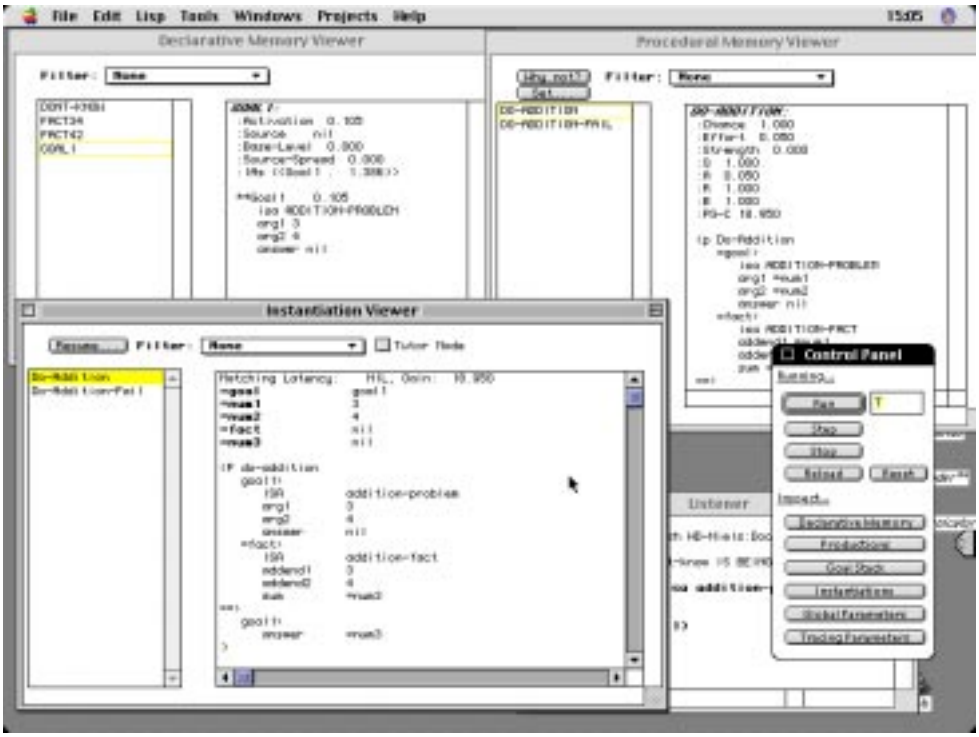


Figure 2.14. The ACT-R environment

production rules has fired, the lisp-function checks whether the answer is correct. After all n problems have or have not been solved, the function gives an accuracy score.

The following trace fragment illustrates the output of the model:

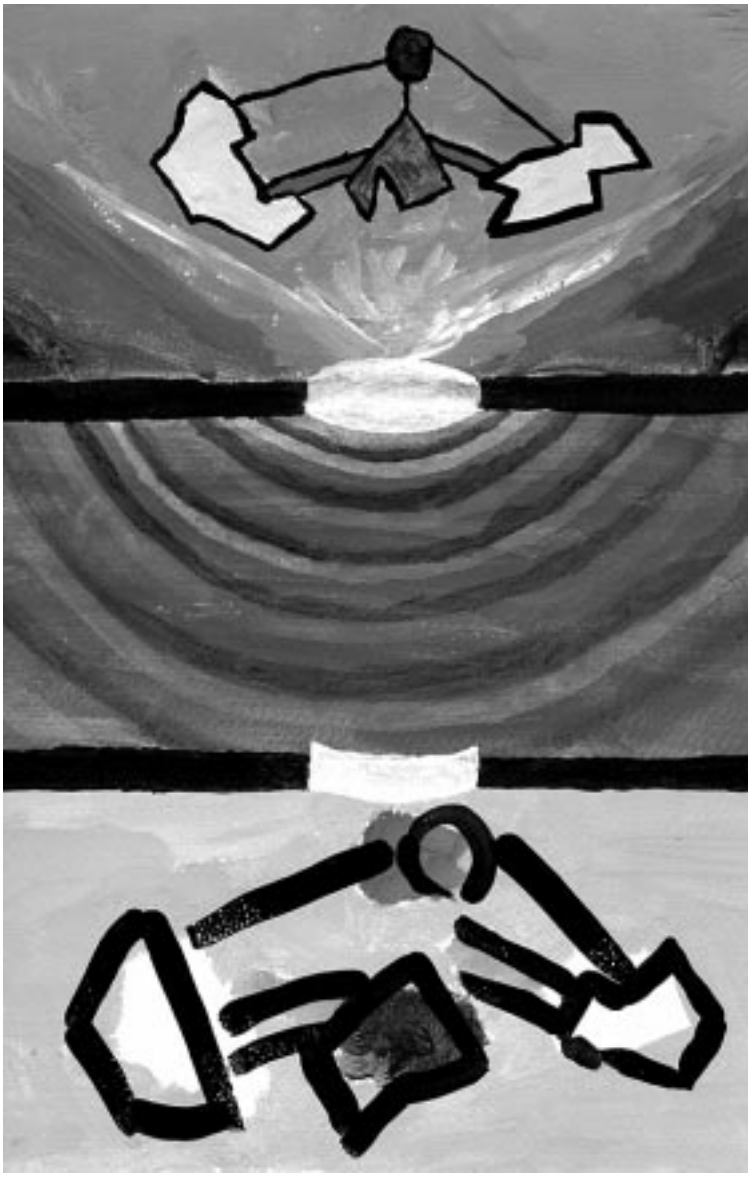
```
? (do-it 2)
 Cycle 0  Time  0.000: Do-Addition-Fail
 Matching latency:  1.000
 Action latency:  0.050

 Stopped at cycle 1
 Run latency:  1.050
 Cycle 1  Time  1.050: Do-Addition
 Matching latency:  0.950
 Action latency:  0.050

 Stopped at cycle 2
 Run latency:  1.000
Accuracy =  0.500
```

This fragment goes through two addition-problems. The first problem fails, but the second succeeds. The trace shows relatively little detail, but additional tracing options can be used to get more information.

Although ACT-R can be used from a command-line interface, an elaborate environment is also offered. Figure 2.14 shows some of the viewers available in the environment, using the addition example. In the environment, models can be executed step-by-step. At each moment the current contents of declarative and procedural memory can be viewed, as well as the rules that are applicable to the current goal. The environment also provides for a syntax-directed editor that makes it easier for novices to enter chunks and production rules. Finally, the environment supports a tutoring function that can be used in combination with a web-based tutorial. The ACT-R code, as well as the tutorial and the code for the environment, is available from http://act.psy.cmu.edu The models discussed in this thesis are listed in an appendix at the end of the thesis, and are all available from a web page as well.

CHAPTER 3

Scheduling



3.1 Introduction

In this chapter I will discuss an experiment that investigates performance changes due to learning while performing the task of precedence constrained scheduling (PCS). In PCS, a number of tasks has to be assigned to a number of workers. Each of the workers has a fixed number of hours to perform tasks. Each of the tasks can have a different (but integer) duration. Finally, a number of order constraints has to be met: sometimes a certain task has to be finished before another task may start. All workers are assumed to work in parallel. A simple example of this problem is:

There are two workers, each having 6 hours
Task A takes 1 hour
Task B takes 2 hours
Task C takes 3 hours
Task D takes 3 hours
Task E takes 3 hours
The following constraints have to met:
A before B
A before D
C before E

A solution to this example is to assign ABD to one worker, and CE to the other. This solution is straightforward, since the constraints are satisfied by the order within a single worker. So "A before B" and "A before D" are satisfied by assigning ABD to a worker, and "C before E" is satisfied by assigning CE to the second worker. A solution in which the constraints are "crossed" is to assign ABE to one worker and CD to the other. In that case the "A before D" and "C before E" constraints span both workers. Problems can be made more difficult by increasing the number of tasks and workers, but also by creating problems in which the constraints span multiple workers for any solution.

Although there are many NP-complete problems that might be used as the task in an experiment, not all of them are equally suitable. PCS has the following attractive properties:

1. The task is easy to explain, since it corresponds to a task participants may be familiar with (scheduling in general).
2. It is improbable that participants have any relevant task-specific knowledge.
3. It is relatively easy to create challenging instances.
4. The task can be presented in several different ways, one of which requires participants to solve problems completely by heart.

The version of the problem I will use in the experiment uses instances in which the tasks always take up all available time of the workers. So, the duration of all the tasks together is equal to the number of workers multiplied by the number of hours each worker has. This sub-problem will be called fully-filled precedence constrained

scheduling (FF-PCS). Restricting the selection of instances to a sub-problem is in a sense dangerous, because a sub-problem of an NP-complete problem is not necessarily NP-complete itself. Fortunately, FF-PCS is also NP-complete. A proof of this fact is given in an appendix to this chapter.

3.2 Experiment

The goal of the experiment is exploration. The general expectation is that if participants have to solve a series of scheduling problems, their performance will generally improve due to learning. But what causes these improvements? Is it a matter of gradual speed-up, or do participants make discoveries that enable them to use a more effective approach to the problem? Analysis of verbal protocols will hopefully shed some light on this issue.

To serve as experimental stimuli, a set of instances with varying difficulty was created. The main determiner for difficulty is the number of workers (m), which ranged from 1 to 3 in the experiment. The stimuli were presented to participants using two different interfaces (figure 3.1), implemented in HyperCard on the Macintosh. The direct-manipulation interface, shown in the top panel of figure 3.1, shows both a propositional representation and a visual representation of the task. The propositional representation lists the constraints of the schedule using short sentences such as "A before B". In the visual representation, tasks are represented by the white boxes with letters in them. The length of each box represents the duration of a task. Workers are represented by grey rectangles. As with the tasks, the length of the rectangle represents the number of hours a worker has. Participants can create a schedule by dragging the task boxes onto the worker rectangles. In the figure task F has already been dragged onto the bottom rectangle. In the propositional interface participants had to perform the planning process entirely by heart. The only thing the interface allows participants to do is to enter the solution by clicking on the rectangles containing the letters (A-F in the example) representing the tasks, the "next worker"-button to end the task list of a worker and move on to the next one, and a "Clear"-button to start over again. Both interfaces contain a "Ready" button which the participant has to click after entering the solution. If the answer is correct the program will move on to the next scheduling problem, else feedback will be provided and the participant has to try again.

To see whether participants develop specialized strategies for specific types of instances, approximately half of the instances has a solution that conforms to a common pattern. This pattern is outlined in figure 3.2 for instances with two and three workers. A representation similar to the direct-manipulation interface is used (the two worker example is the solution to the instance in figure 3.1). These instances

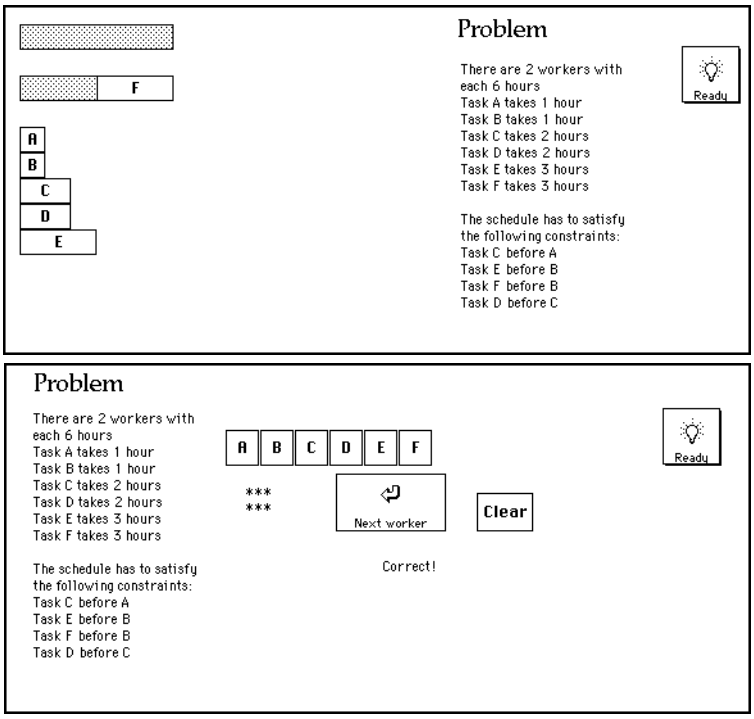


Figure 3.1. Two interfaces used in the experiment. The top panel shows the direct-manipulation interface in which participants can drag around boxes representing the tasks, while the bottom panel shows the propositional interface in which participants have to solve the problem by heart.

are particularly hard due to the fact that many of the precedence constraints in this pattern cross workers in the solution.

Although this experiment is primarily exploratory, a number of expectations can be formulated. A first expectation is that performance will increase due to experience: a learning effect. A second expectation is that there will be an effect of the type of interface: the direct-manipulation interface is easier, so will lead to better performance. A third expectation is that instances conforming to the pattern in figure 3.2 will be harder to solve than other instances. A final expectation is that participants will discover some new strategies to solve the scheduling problem. Evidence for new strategies has to found by protocol analysis, or by sudden jumps in performance.

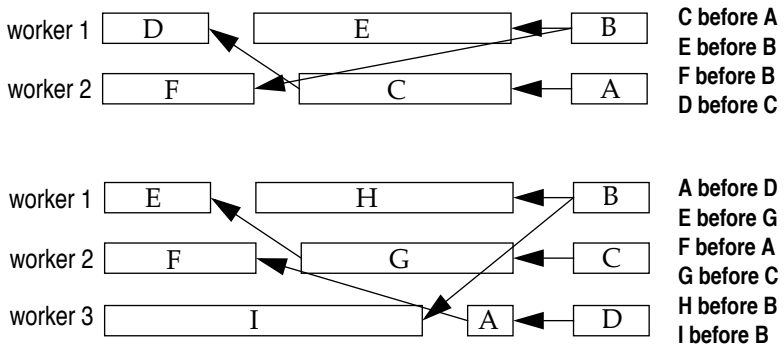


Figure 3.2. Schematic diagrams of the solutions to half of the instances presented to the participants (an example for two and for three workers is shown). The representation is similar to the direct-manipulation interface: boxes represent tasks and the length of a box represents its duration. The arrows represent precedence constraints. Note that the letters in the boxes are just examples and differ between instances.

Method

Participants. Eighteen undergraduate students of the University of Groningen were recruited to participate in this experiment. The experiment lasted 2 hours, including instructions and a small break. Participants were paid Fl. 20 for their efforts.

Materials. Sixteen FF-PCS instances were created of the following types:

- R1 (2 instances): a single worker with four or five tasks
- A2 (10 instances): two workers, conforming to the pattern in figure 3.2
- R2 (8 instances): two workers, not conforming to any specific pattern
- A3 (3 instances): three workers, conforming to the pattern in figure 3.2
- R3 (3 instances): three workers, not conforming to any specific pattern

Procedure. Participants were randomly assigned to two groups. Group 1 started the experiment with the direct-manipulation interface, and switched to the propositional interface for the second half of the experiment. Group 2 started with the propositional interface and switched to the direct-manipulation interface for the second half. Figure 3.3 shows the exact experimental procedure for each of the two groups. At the start of the experiment, participants were instructed about the task and the particular interface they started with. To ensure participants properly understood how to handle the interface, they were given an example problem with its solution, after which they had to enter the solution. After the break participants were told the task would remain the same, but the way in which they had to enter the answer had changed. They then again had to enter the solution of an example

Group 1		Group 2	
Start of experiment		Start of experiment	
Direct manipulation interface	one R1 problem	Propositional interface	one R1 problem
	one R2 problem		one R2 problem
	five A2 and three R2 problems, in random order		five A2 and three R2 problems, in random order
Break		Break	
Propositional interface	one R1 problem	Direct manipulation interface	one R1 problem
	one R2 problem		one R2 problem
	five A2 and three R2 problems, in random order		five A2 and three R2 problems, in random order
	three A3 and three R3 problems, in random order		three A3 and three R3 problems, in random order
End of experiment		End of experiment	

Figure 3.3. Experimental procedure

using the new interface. Participants were asked to think aloud during the experiment, which was recorded using an audio cassette recorder. The software registered and time-tagged all actions participants performed during the experiment.

Analysis of the results
To analyze the results of the experiment, a number of methods will be used. First, we will examine the solution times, and see if participants become faster, and whether or not there is transfer between the first and the second interface. Secondly, we will do a protocol analysis on the verbal protocols in order to get a deeper insight into what strategies participants learn during the experiment.

3.3 Analysis of solution times

An informal analysis
There are a number of potential factors that influence the solution time for each instance:

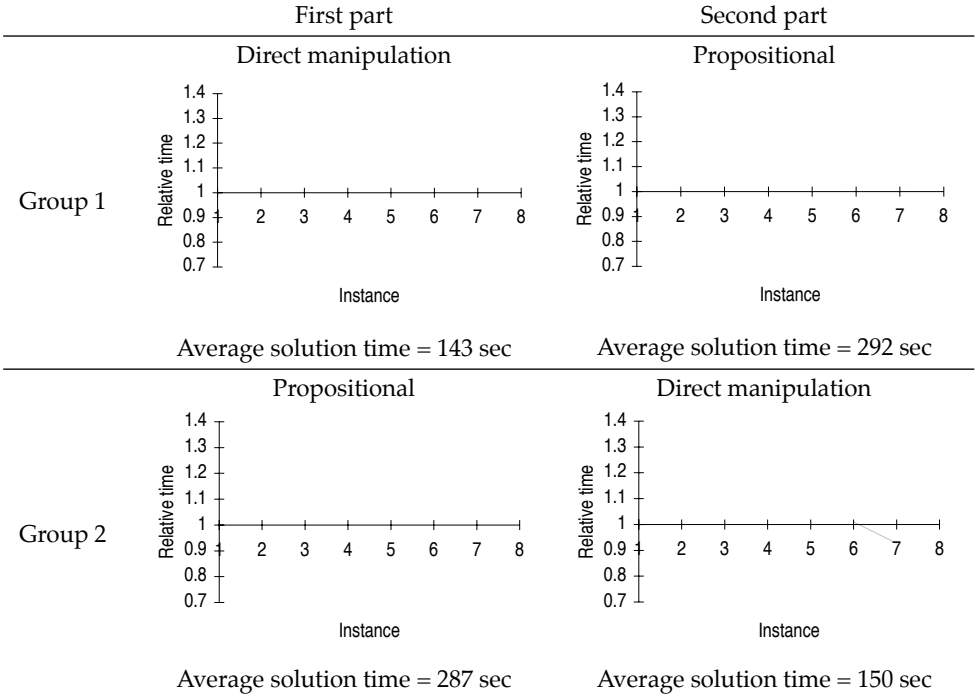


Figure 3.4. Relative time to solve each instance for the two groups and the two parts of the experiment

1. Individual differences
2. The difficulty of the instance
3. The interface (direct-manipulation or propositional)
4. Learning

To get some impression of the learning factor, which is the main factor of interest, we will first do some quick calculations. To remove the difficulty factor of items, all solution times were divided by the average solution time for that particular item. Since participants occasionally "got stuck" at a particular instance, solution times that were longer than 2.5 times the average time were removed (8 cases out of 288). Finally, the scores were averaged and plotted in figure 3.4. Only the five A2 problems and three R2 instances are plotted, since the first R1 and first R2 instance are the same for all participants, so average to 1 all the time, and the A3 and R3 instances at the end of the experiment were completed by too few participants, so were also omitted in the analysis. In the first part for each of the two groups there is a clear learning effect, since on average participants start at around 1.3 times the

average time to solve an instance, and improve to around 0.8 times the average time. In the second part of the experiment, there is no effect of learning. This cannot be explained by the fact that the learning curve has flattened out due to the fact that there is nothing left to learn, since the average solution time is not better than in the first part, but even slightly worse (average solution times are below each of the graphs.) So, there is evidence for a time-on-task effect. This effect has several possible explanations, like boredom and a decrease in motivation or mental fatigue.

An analysis using multilevel statistics

A more thorough method to analyze the data that gives an impression of the impact of the different factors on the solution time, is to make a statistical model using multilevel analysis (Bryk & Raudenbush, 1992). A model in the sense of multilevel analysis is a set of regression equations that predicts the dependent variable, the solution time in our case. The basic regression equation is as follows:

$$y_{ti} = \beta_{0i} + \beta_{1i}A_{ti} + \beta_{2i}B_{ti} + \ldots + r_{ti} \quad \textbf{(3.1)}$$

The solution time y_{ti} for participant *i* and trial *t* is predicted by an intercept β_{0i} for participant *i*, plus the influence of a number of factors. Factors, in this equation represented by A_{ti}, B_{ti}, etc., are in our case the type of interface, the difficulty of an instance and the trial number. These factors are scaled by β_{1i}, β_{2i}, etc. The final part of the equation, r_{ti}, represents the random variance for each trial.

Each of the β scaling factors may vary between participants, as indicated by the *i* index. Each of these scaling factors has its own equation:

$$\beta_{0i} = \gamma_{00} + u_{0i} \quad \textbf{(3.2)}$$

$$\beta_{1i} = \gamma_{10} + u_{1i} \quad \textbf{(3.3)}$$

Multilevel analysis in general also allows us to add factors to these equations, comparable to the A_{ti} and B_{ti} in (3.1). These factors represent characteristics of individual participants. They are omitted here, since no such information is available. The γ-coefficients are called the *fixed effects* in terms of multilevel analysis, since they do not change between either participants or trials. The *u*-coefficients are called *random effects*, since they vary between participants. Not all factors have a significant random effect on differences between individuals, so sometimes the β's are just equal to the γ's. An advantage of this method is that between-participant variance and within-participant variance can be discerned. A random effect on β_{0i} means that individuals have different starting points on the learning curve, so a random intercept. If we take the trial number as a factor, the β-coefficient that serves as its multiplication factor will become the slope of the learning curve. A random effect on this coefficient means individuals have different learning rates, so a random

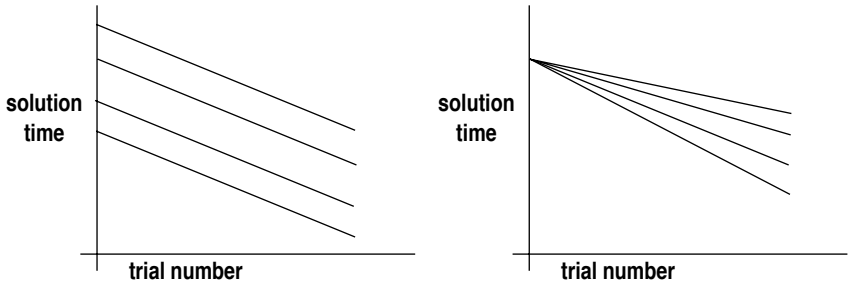


Figure 3.5. The difference between a random intercept (left), and a random slope (right). Each line represents an individual participant.

slope. Figure 3.5 illustrates the difference between a random intercept and a random slope.

The general method to find the most appropriate model for a certain set of data is to use a multilevel analysis program to estimate the coefficients in the model. In this case MLn (Rasbash & Woodhouse, 1995) is used. The analysis starts with a very simple model, after which additional factors are added incrementally. After each additional factor, the deviance (- 2 log likelihood) between the model and the data is checked, to see if the additional factor provides for a significant improvement.

Analysis of the first part of the experiment

One of the constraints on multilevel analysis is that the dependent variable has to have a normal distribution. The solution times in the scheduling experiment, however, are skewed. In order to fix this, the logarithm of solution times is used instead of plain solution times. One of the factors will be the trial number itself, in order to estimate the learning effect. The actual version of (3.1) now becomes:

$$\log y_{ti} = \beta_{0i} + \beta_{1i} t + \gamma_{20} A_{ti} + \gamma_{30} V_{ti} + \gamma_{40} A_{ti} V_{ti} + r_{ti} \quad \textbf{(3.4)}$$

In this equation t is the trial number, ranging from 1 to 9, A_{ti} equals 1 if the trial involves a type A instance (the difficult ones) and 0 otherwise, V_{ti} equals 1 if the item is presented in the propositional interface, and 0 if it is presented in the direct-manipulation interface. The $A_{ti}V_{ti}$ term represents the interaction between interface and instance type. Note that most of the β's have been replaced by γ's, indicating that no random effect on the level of individual differences has been investigated. Both β-parameters are calculated according to (3.2) and (3.3). The data used in the analysis are all the A2 and R2 instances in the first part of the experiment, nine in all. Three data points with excessively long solution times were removed (each from a different participant).

Fixed effects				
Effect	Parameter	Estimate	S.E.	p<
Intercept	γ_{00}	5.030	0.130	.000
Trial no.	γ_{10}	-0.074	0.014	.000
Propositional interface	γ_{30}	0.549	0.163	.000
Type A problem	γ_{20}	0.104	0.104	.159
Propositional * Type A	γ_{40}	0.298	0.145	.020
Random effects				
var(intercept)	u_{0i}	0.067	0.030	
var(residual)	r_{ti}	0.207	0.025	
-2 log likelihood:		224.9		

Figure 3.7. Statistical model of the log solution times

Although (3.4) represents the most elaborate model, it is not necessarily the best model. The procedure to find the best model is as follows. We start with the most simple model, in this case the model that just states that the solution time is a fixed value, and all variation is random noise. This model will leave some unexplained variance, as expressed in the -2 log likelihood estimate. The next step is to add some factor that may improve the model. Adding a factor reduces the degrees of freedom, so this reduction must be warranted by a significant decrease in unexplained variance. In this analysis, the significance threshold will be 5%. Figure 3.6 shows the search tree to find the most appropriate model. At the top of the tree the most simple model is shown. Adding the factor of time considerably improves the model, as shown by the second box. Now there are two choices: adding a random intercept or a random slope. The search tree explores both possibilities. Note that the introduction of random effects implies replacing a fixed γ in the formula by a β that has a different value for each participant. Although both new models improve the previous model, the random intercept model reduces unexplained variance most. Moreover, if a random slope is consequently added, this does not improve the model. Apparently the individual differences can be captured by just a random intercept. The next three steps add the factors of interface type, problem type and the interaction between the two. Each of these steps improve the model. Finally, a last attempt is made to add a random slope, but this still does not improve the model.

The final model is presented in figure 3.7. It turns out that the effect of trial number is very significant, so there is a clear learning effect. The type of interface also has a significant impact on the solution time: the propositional interface, not surprisingly,

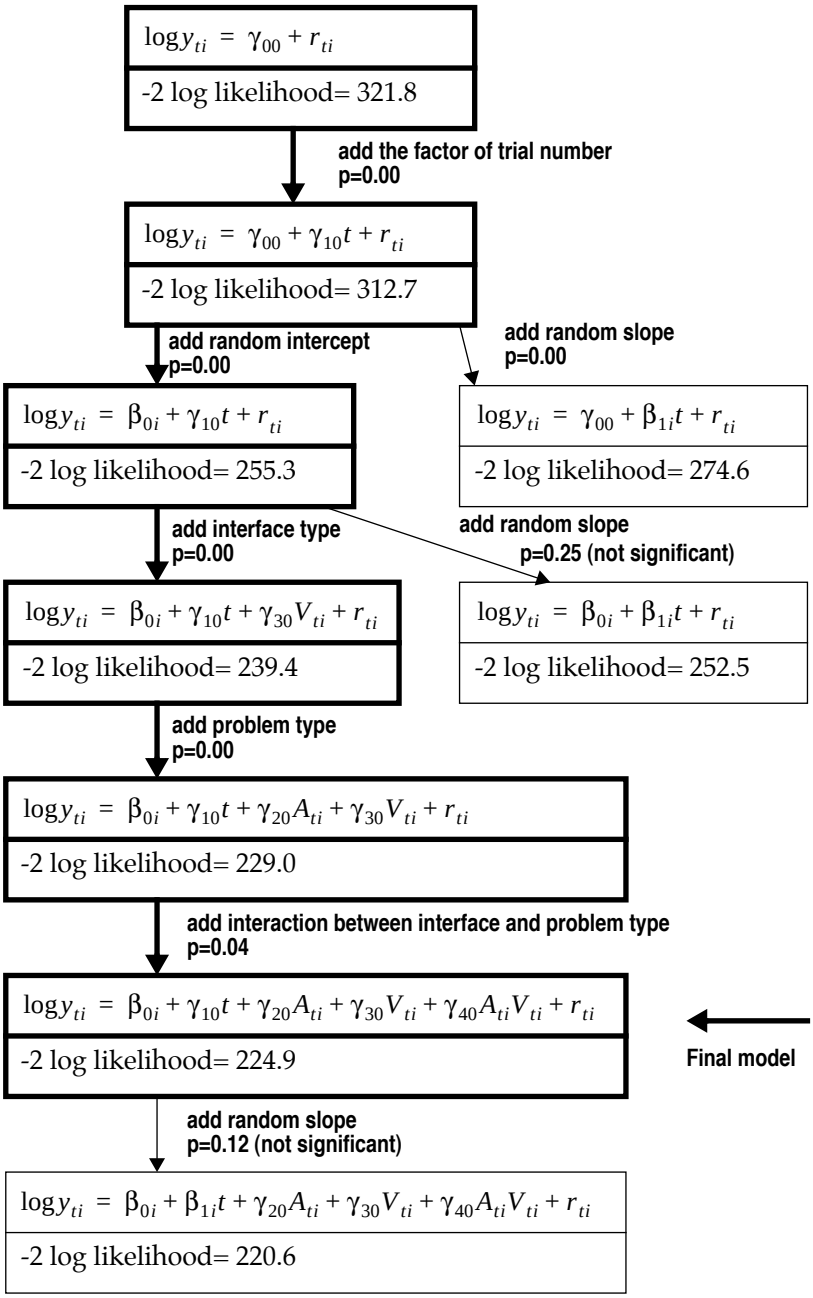


Figure 3.6. Search tree to find the best model. Each box represents a model, with the equation at the top and the -2 log likelihood at the bottom. The thick arrows and boxes indicate the optimal search path.

requires more time. Whether or not an instance is of type A is mainly significant in interaction with the type of interface. If the instance is of type A and the interface is propositional, there is an extra increase in solution time. This interaction can be explained by pointing at the difficulty of type A problems. The hard part of solving type A problems is to coordinate precedence constraints that span multiple workers. The fact that this type of instances is especially hard in the propositional interface condition is evidence for the fact that participants do not use a visual image to represent the schedule, but rather a linear string of tasks. If participants used a visual image, the type A problems would not have any additional difficulty associated with them in the propositional interface condition. If a schedule is represented as a linear string, it is easy to check constraints within a worker, but very hard to check constraints between workers.

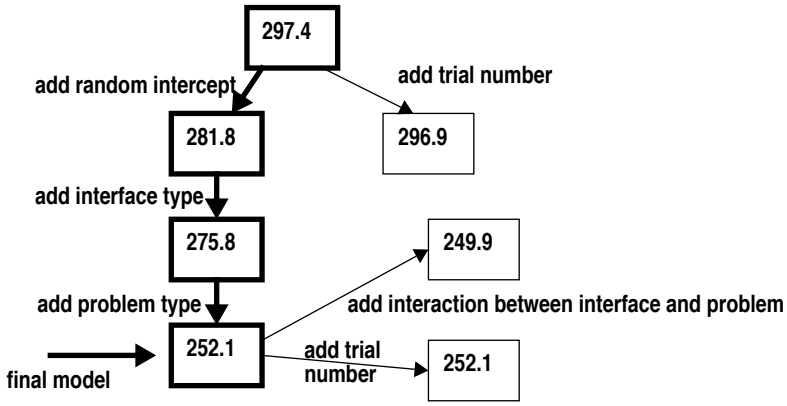
It turns out that the best fitting model only has a random intercept and no random slope. So, the main source of individual differences is the starting point of the learning curve. Individual differences in learning rate were not large enough to provide for a better fitting model.

Analysis of the second part of the experiment

Although the informal analysis already showed that there is probably no learning effect in the second half the experiment, a multilevel analysis was conducted for that part as well. Before the analysis, five datapoints with excessively long solution times were removed from the set. Figure 3.8 shows an abbreviated version of the search tree: only -2 log likelihoods are shown and the additions to the model. What is immediately apparent is that the trial number has no impact on the solution time, whether it is added to the simple model or to the final model. Although the interaction between interface and problem type is not significant in the second part of the experiment, the other effects are quite similar to effects in the first part of the experiment. The bottom part of figure 3.8 shows the final model.

Conclusions

The statistical model of the solution times confirms the expectations stated in the previous section, at least with respect to the first part of the experiment. There are strong effects of learning, interface type and problem type. None of these effects are particularly surprising. Oddly enough there is no learning effect in the second part of the experiment. An explanation for this has to be sought in the area of motivation or fatigue: perhaps participants no longer seek strategies that improve their performance. In chapter 5, I will show learning may indeed be partly dependent on motivation, because a low motivation makes learning strategies less attractive than just trying a simple strategy over and over again.



Fixed effects				
Effect	Parameter	Estimate	S.E.	p<
Intercept	γ_{00}	4.516	0.116	.000
Propositional interface	γ_{30}	0.403	0.152	.004
Type A problem	γ_{20}	0.427	0.084	.000
Random effects				
var(intercept)	u_{0i}	0.072	0.035	
var(residual)	r_{ti}	0.264	0.032	
-2 log likelihood:		252.1		

Figure 3.8. Abbreviated search tree for the model of the second part of the experiment and the final model

3.4 Analysis of verbal protocols

Inspection of the verbal protocol recordings reveals that only protocols of the propositional interface condition can be interpreted easily. The recordings of the direct-manipulation interface contain little information, and are difficult to correlate with actions of the participants in the interface. This was to be expected, since verbal protocol analysis tends to be a poor research instrument in assessing tasks with a large visual component. Since the learning effects are largest in the first half of the experiment, participants from group 2 are used primarily in the analysis. One of the recordings was unusable due to problems with the cassette recorder, and another participant did not verbalize enough in order to be intelligible. So seven

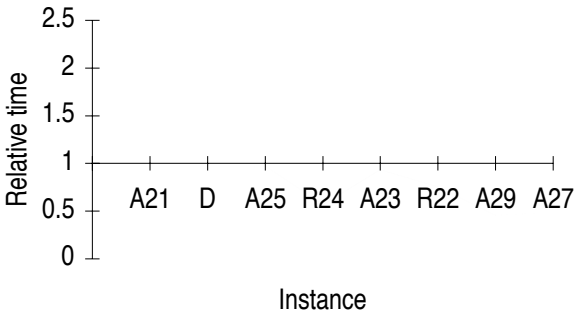


Figure 3.9. Relative solution times (similar to figure 3.4) for participant 2. Note that A21 is the third instance the participant has to solve, since the two training problems are not in the graph.

protocols were available from group 2. To get an impression of group 1 as well, two protocols from the second half of the experiment are added, giving a set of nine protocols.

We will proceed with the analysis in two steps. First, we will do a detailed analysis of a single participant. This analysis will show what kinds of processing are going on during problem solving. Secondly, a more quantitative analysis will be done on the set of nine protocols.

Analysis of participant 2

Participant 2 shows a learning curve that is similar in shape to the average learning curve (figure 3.9). The relative time to solve an instance improves from 2.3 for the first instance to 0.5 for the last, a learning effect that is even larger than the average participant. The following excerpt is from the protocol of participant 2, while she is solving problem A21 (figure 3.1). This problem is interesting, since it is followed by a large jump in performance. A possible explanation, which we will now examine in detail by protocol analysis, is that the participant discovers some new means to solve scheduling problems. The problem and the protocol are translated from Dutch, as are all other excerpts discussed. First I will repeat the problem:

There are two workers with 6 hours each
Task A takes 1 hour
Task B takes 1 hour
Task C takes 2 hours
Task D takes 2 hours
Task E takes 3 hours
Task F takes 3 hours
The schedule has to satisfy the following constraints:
C before A
E before B
F before B
D before C

The protocol is as follows:

> Yes. There are two workers with each six hours. Two. Task A, task B, task C. The schedule has to satisfy the following constraints... Task C before A, C before A, E before B, F before B and D before C. [..unintelligible..] First now D. D.. D..C..A..B.., D..C..A..B.., D.C.A.B., DCAB, and then, DCAB, [keys in DCAB] and then E... E..F, E..F. [keys in EF] [Receives feedback] Oh, task F is not before B. C.., D has to be before C. D.. No, C..D.., D has to be before C. C.. D.., C.. D.., A...B [keys in CDAB] that's one worker. E..F..., [keys in EF]. [receives feedback] Huh?! Task F is not before B and task D is not before C? Oh wait. D has to be before C, so first D... D...C..AB..AB [keys in DCAB]. Next worker, F.. yes, F..E.., ready. [keys in FE]. [receives feedback] Task E is not before B? Isn't it? Yes? [Emphasizing, keys in] D..C..A..B..E..E..F...ready. [receives feedback]. Well! Ehmm.. Task D takes two hours. [Silence] Task F is not before B, so F should be before B. Task E before... E should be before B, so E and F shouldn't be done by.... by the same worker. So we will, let's see. Task C before A, so we will first.... E before B, so we will first E..E..E..B..C. E...E..B..C.., EBC, no that's not right. EBC..F..A..B.. Ah.. start again. The D should be before C. [silence]. E... Ehm... The D should be before the C, so we put the D with worker one, and C with worker two. So we start with E with worker one... E..C..A.. E.C.A. ECA.. E.C.A. No, I don't get it... E..C..A..D..F.. Oh.. wrong again.

This is about half of the total protocol for problem A21. Participant 2 needed 793 seconds to solve the whole problem. It is obvious that the written protocol doesn't reveal much in the presentation given above. Nevertheless, we can already infer some categories into which we can classify the various elements in the protocol. First there are reading actions, in which the participant reads parts of the problem. It is also obvious that the participant incrementally builds a schedule by adding tasks one by one. So adding a task to the current schedule is also a possible action. The interesting parts of the protocol are the parts in which the participant makes complex inferences. There is one obvious example in the above excerpts, where the participants remarks "so E and F shouldn't be done by.... by the same worker." In order to reach this conclusion, five constraints of the problem need to be combined: the fact that each worker has six hours that both E and F take three hours, and the fact that both E and F must be before B. Identification of the simple steps in the problem solving process enables us to keep track of the information the participant has in working memory at a particular time. In order to analyze the above fragment, and the rest of the protocol, the following categories will be used.

Notational primitives:

- c:a denotes the constraint "C before A". Participants can connect these constraints to more advanced schemas like b:c:a (b is before c is before a) or b;c:a (b and c are both before a).
- sched(acd | bef) denotes a schedule or a schedule fragment, the vertical bar separates workers.
- a5 denotes the number of hours a task takes (in this case: task a takes 5 hours).
- work2 denotes the number of workers (in this case 2 workers).
- time7 denotes the number of hours each worker has (in this case 7 hours).
- diff(a,b) denotes the fact that task a and b must be done by different workers

- same(a,b) denotes the fact that task a and b must be done by the same worker.
- last(a) denotes the fact that task a must be done last by a worker.
- first(a) denotes the fact that task a must be done first by a worker.
- middle(a) denotes the fact that task a must be somewhere in the middle of the schedule of a worker.

Reading
When the participant reads something from the screen, this is denoted by the Read() action. The argument is the item read. For example, Read(c:a) corresponds to the participant reading "c before a". The result of a reading action is that the item read is in working memory.

Adding tasks to the schedule
When a participant adds new items to the current schedule this is represented by the Add() action. The argument is the task added to the schedule. The result of an add action is that the task is added to the current schedule in working memory.

Rehearsing working-memory items
Any items in working memory (WM) can be rehearsed, which is denoted by the Reh() action. The argument is the item rehearsed.

Inference
In general inference is denoted by Inf(p1; p2 → q), meaning q is inferred from p1 and p2. Precondition for such an inference is that p1 and p2 are available in memory. The result is that q will be in working memory.

Evaluation

- Eval+ denotes that the participant concludes the schedule is correct.
- Eval-() denotes that the participant concludes that the schedule is incorrect. If a violated constraint is mentioned, it is given as the argument, for example Eval-(c:a): the schedule is incorrect because c is not before a
- IEval+ denotes that the program accepts the solution
- IEval-() denotes that the program rejects the solution, the violated constraint(s) are again between parentheses

Other actions

- Restart denotes that the participant starts again
- Inkey() denotes that the participant keys a (possibly partial) solution into the computer
- Fill denotes miscellaneous remarks

- Meta denotes remarks about the difficulty or other aspects of the task
- Q denotes a question of the participant to the experimenter

Using this scheme, the following analysis can be made of the protocol fragment. The analysis column shows an interpretation of the fragment listed in the protocol column. The WM column shows the possible contents of working memory based on this interpretation.

Problem A21. Time 793 seconds

Protocol		Analysis	WM
1.	Yes. There are two workers with each six hours.	Read(work2); Read(time6)	work2; time6
2.	Two. Task A, task B, task C.	Reh(work2); Read(a); Read(b); Read(c)	work2
3.	The schedule has to satisfy the following constraints...	Fill	
4.	Task C before A, C before A, E before B, F before B and D before C.	Read(c:a); Reh(c:a); Read(e:b); Read(f:b); Read(d:c)	c:a; e:b; f:b; d:c
5.	[unintelligible]	?	
6.	first now D. D..	Add(d)	sched(d)
7.	D..C..A..B..,	Reh(d); Add(c); Add(a); Add(b)	sched(dcab)
8.	D..C..A..B..,	Reh(dcab)	sched(dcab)
9.	D.C.A.B,	Reh(dcab)	sched(dcab)
10.	DCAB,	Reh(dcab)	sched(dcab)
11.	and then,	Add(l)	sched(dcabl)
12.	DCAB, [keys in DCAB]	Reh(dcabl); KeyIn(dcabl)	sched(dcabl)
13.	and then E...	Add(e)	sched(e)
14.	E..F..	Reh(e); Add(f)	sched(ef)
15.	E..F.. [keys in EF]	Reh(ef); KeyIn(ef)	sched(ef)
16.	[receives feedback] Oh, task F is not before B.	IEval-(f:b); Restart	
17.	C..,	Read(c)	
18.	D has to be before C.	Read(d:c)	d:c
19.	D	Read(d)	
20.	No, C..D..,	Add(c); Add(d)	sched(cd)

There are two workers with each 6 hours
Task A 1 hour
Task B 1 hour
Task C 2 hours
Task D 2 hours
Task E 3 hours
Task F 3 hours
The schedule has to satisfy the following constraints:
C before A
E before B
F before B
D before C
Solution:
ECA
DFB

Problem A21. Time 793 seconds

Protocol		Analysis	WM
21.	D has to be before C.	Read(d:c)	sched(cd); c:d
22.	C..D..	Reh(cd)	sched(cd)
23.	C..D..,	Reh(cd)	sched(cd)
24.	A..B.. [keys in CDAB]	Add(a); Add(b); KeyIn(cdab)	sched(cdab)
25.	That's one worker	Add(l); KeyIn(l)	sched(cdabl)
26.	E..F..., [keys in EF].	Add(e); Add(f); KeyIn(ef)	sched(cdablef)
27.	[receives feedback] Huh?! Task F is not before B and task D is not before C?	IEval-(f:b;d:c); Restart	
28.	Oh wait.	Fill	
29.	D has to be before C,	Read(d:c)	d:c
30.	so first D... D...C..AB..AB [keys in DCAB].	Add(d); Reh(d); Add(c); Add(a); Add(b); Reh(ab); KeyIn(dcab)	sched(dcab)
31.	Next worker,	Add(l); KeyIn(l)	sched(dcabl)
32.	F.. yes, F..E.., ready. [keys in FE].	Add(f); Reh(f); Add(e); KeyIn(fe)	sched(dcablfe)
33.	[receives feedback] Task E is not before B?	IEval-(e:b)	sched(dcablfe)
34.	Isn't it? Yes? ?	Fill	sched(dcablef)
35.	[Emphasizing, keys in] D..C..A..B..E..E..F...ready.	KeyIn(dcablef)	sched(dcablef)
36.	[receives feedback]. Well!	IEval-	sched(dcablef)
37.	Ehmm.. Task D takes two hours.	Read(d2)	sched(dcablef); d2
38.	[Silence]	Fill	sched(dcablef)
39.	Task F is not before B,	Eval-(f:b)	sched(dcablef); f:b
40.	so F should be before B.	Reh(f:b)	sched(dcablef); f:b
41.	Task E before... E should be before B,	Read(e:b)	sched(dcablef); f:b; e:b
42.	so E and F shouldn't be done by.... by the same worker..,	Inf(f:b; e:b; e3; f3; time6 → diff(e,f))	diff(e,f)

Problem A21. Time 793 seconds

Protocol		Analysis	WM
43.	So we will, let's see. Task C before A,	Restart; Read(c:a)	diff(e,f); c:a
44.	so we will first....	Fill	diff(e,f)
45.	E before B,	Read(e:b)	
46.	so we will first E..E..E..B..C.	Add(e); Reh(e); Reh(e); Add(b); Add(c)	sched(ebc)
47.	E...E..B..C..,	Reh(e); Reh(ebc)	sched(ebc)
48.	EBC,	Reh(ebc)	sched(ebc)
49.	no that's not right.	Eval-	sched(ebc)
50.	EBC..F..A..B..	Add(l); Add(f); Add(a); Add(b)	sched(ebclfab)
51.	Ah.. start again.	Restart	
52.	The D should be before C.	Read(d:c)	d:c
53.	[silence].	Fill	d:c
54.	E... Ehm...	Read(e)	d:c
55.	The D should be before the C, so we put the D with worker one,	Inf(d:c $\rightarrow$ diff(c,d))	
56.	and C with worker two.		
57.	So we start with E with worker one...	Add(e)	sched(e)
58.	E..C..A..	Reh(e); Add(c); Add(a)	sched(eca)
59.	E.C.A.	Reh(eca)	sched(eca)
60.	ECA..	Reh(eca)	sched(eca)
61.	E.C.A.	Reh(eca)	sched(eca)
62.	No, I don't get it...	Fill	
63.	E..C..A..D..F.. Oh.. wrong again.	Reh(eca); Add(d); Add(f); Eval-	

A dissection of the protocol in terms of the analysis above reveals a bit more of what is going on during the problem solving process. Figure 3.10 shows a summary of the analysis in the form of the search tree that is traversed in the episode above.

The participant starts with reading the problem (1-4, not shown in the figure). After that, there are four episodes in which she tries to find a solution (5-16, 17-27, 28-33 and 34-36). Each of these episodes consists of a number of alternating processes: incrementally increasing the current schedule, rehearsing the current schedule, and

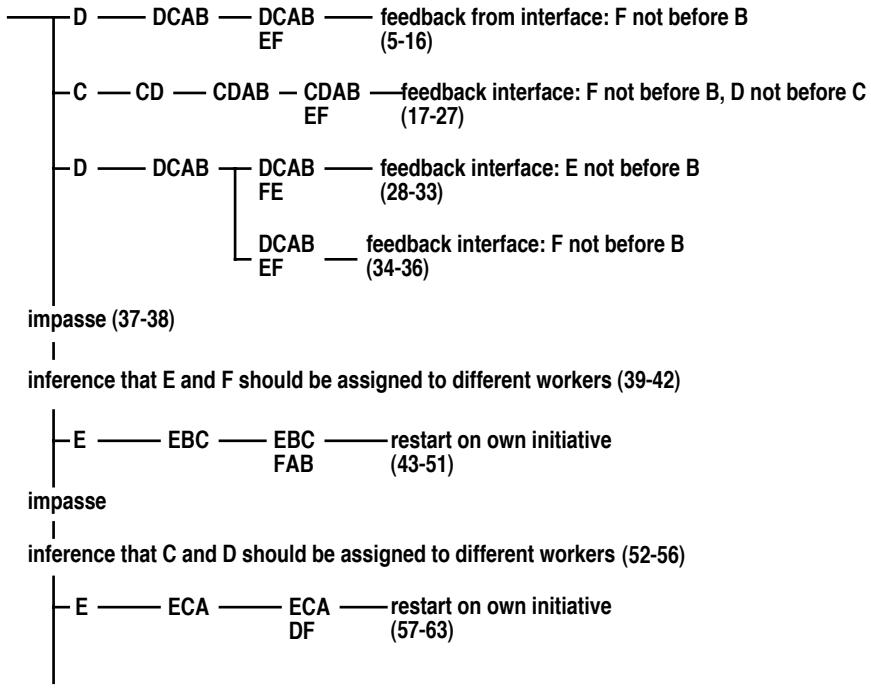


Figure 3.10. Search tree corresponding to the analysis of instance A21. Numbers in parentheses refer to line numbers in the protocol.

evaluating the current schedule. At the end of each episode, the resulting schedule is either rejected by the interface or by the participant. Although the participant does not reveal on what basis she selects tasks to add to the plan, the precedence constraints seem to be an obvious lead. The DCAB sequence, which recurs in three of the four episodes, directly reflects the "D before C" and "C before A" constraints. A possible strategy underlying this type of sequencing is to look for two constraints in which the second task in the first constraint equals the first task in the second constraint, and distill a three-task sequence out of it.

Up to line 36 in the protocol, the problem-solving process seems to follow a straight forward search pattern, although the participant only backtracks once, but rather starts again after a dead end in the search tree. Furthermore, the participant tries the same solution twice. After four unsuccessful tries, however, the participant reaches an impasse (37-38). After this impasse, a complex inference is used to infer a new constraint, the fact that task E and F should be assigned to different workers (39-42). As mentioned before, this inference is quite complex, since it involves five constraints. Using this newly inferred constraint, search is resumed, and a new unsuccessful episode follows (43-51). The solution reached in this episode differs from the previous episodes, however, in the sense that the newly derived constraint

is satisfied. Another interesting change is that the participant evaluates the solution herself, instead of relying on the interface. In the next episode the participant derives another new constraint, the fact that task D and C should also be assigned to different workers (52-56). In the final episode (57-63), the participant nearly reaches the solution. Although she only needs to add task B to her schedule, she somehow decides the solution is incorrect and starts anew. Examination of the complete protocol reveals that the participant needs several more search episodes before she solves the problem.

The problem solving fragment discussed above shows aspects of two theories of problem solving. Processing within search episodes concurs with the theory that problem solving is problem-space search, which we have discussed in chapter 1. On a more global scale, however, the fragment shows aspects of *insight theory* (Davidson, 1995). According to insight theory, which is rooted in Gestalt psychology, the interesting moment in problem solving is when the problem solver suddenly "sees" the solution, in a moment when an "unconscious leap in thinking" takes place. Instead of describing problem solving as a gradual approach of the goal, insight theory predicts the following pattern: exploration, impasse, insight and execution. The nine-dots problem is a typical example (see figure 1.3 in chapter 1): the exploration phase consists of fruitless search within the boundaries of the nine dots, after which an impasse occurs followed by the insight that lines may go beyond the boundaries of the grid. This insight allows for a final resolution in terms of a solution.

This insight problem-solving pattern can be found in the problem-solving fragment, since the four unsuccessful search attempts (5-36) can be seen as the exploration phase, after which an impasse occurs (37-38), followed by an insight (39-42). Unfortunately, the insight is only an important step in the direction of the solution, so the execution phase actually involves some more exploration. Furthermore, the insight episode isn't really an "unconscious leap in thinking", but rather an episode of solid rational reasoning. Although the fragment shows the pattern of insight problem solving, it does not share the more mystical aspects associated with some versions of insight theory.

Learning the different-worker strategy

From the viewpoint of learning problem solving it is interesting to investigate whether something is learned during an insight episode. Although it is hard to actually prove something new is learned, it is possible to find some evidence that this is the case. One way to do this is to see if the same pattern of reasoning can be found again in later instances. After problem R25, a problem in which the pattern cannot be used, it recurs in problem A25, as the following fragment shows:

Problem A25 Time 300 seconds

Protocol		Analysis	WM
20.	task A should be before D	Read(a:d)	sched(adcb); a:d
21.	A..D..C..B..E..F..ready..	Inkey(adcb); Add(l); Add(e); Add(f); Inkey(adcblef)	sched(adcblef)
22.	task F is not before B.	IEval-(f:b)	f:b
23.	so E and F cannot be done by the same worker...	Inf(e2; f2; time4; f:b; e:b → diff(e,f))	diff(e,f)

Again, the complex inference is made after two unsuccessful search attempts. In this case, however, there is no impasse: the participant immediately makes the inference. Later in the experiment, in problem A29, the same strategy is used:

Problem A29 Time:128 seconds

Protocol		Analysis	WM
15.	B..A..D..C, next worker..	Reh(badc); Add(l)	sched(badcl)
16.	E..F, ready.	Add(ef); Inkey	sched(badclef)
17.	Oh, task F is not before C, so E and F again can't go together.	IEval-(f:c); Inf(f:c; ? → diff(e,f))	diff(e,f)

The only difference with problem A25 is that the participant seems to recognize the fact that this pattern has occurred before. In the final type A problem, problem A27, the participant immediately uses the newly learned strategy without resorting to fruitless search first:

Problem A27 Time:140 seconds

Protocol		Analysis	WM
8.	So then it is eeh, A before C and D before A, E beforeB and F before B	Read(a:c); Read(d:a); Read(e:b); Read(f:b)	a:c; d:a; e:b; f:b
9.	E and F can probably not go together, since then they will not be before B	Inf(e:b; f:b → diff(e,f))	diff(e,f)

Summarizing, the four protocol fragments show how a new strategy, the "different-worker strategy", comes into existence. In A21, the strategy is discovered in a classic

insight problem-solving pattern. In A25, the pattern recurs, except that there is no impasse period. In A29, the pattern again recurs, but the participant shows evidence of recognizing the strategy. Finally, in A27, the strategy is incorporated in the normal search process.

Learning the fit-the-hours strategy

The different-worker strategy is not the only strategy the participant discovers during problem solving. The first indication of a second strategy is in problem R25, the fourth problem.

Problem R25 Time 200 seconds

Protocol		Analysis	WM
15.	ABF...D..E..C..	Reh(abfld); Add(e); Add(c); Inkey(abfldec)	sched(abfldec)
16.	Oh, no, that's not right. task C is not before B,	Eval-(c:b)	sched(abfldec)
17.	OK, one more time.	Restart	
18.	F, A and B belong together, so the first worker...	Inf(f5; a2; b2; time9 → same(a,b,f))	same(a,b,f)

2 workers
9 hours
A 2 hours
B 2 hours
C 3 hours
D 3 hours
E 3 hours
F 5 hours

A before E
C before B
E before D

Solution:
AFB
CED

After an unsuccessful search episode, the participant mentions that F, A and B belong together, and should be assigned to the first worker. Inspection of the particular problem shows why this may be inferred. For each worker, the sum of the tasks assigned must add up to nine hours. Only 2+2+5 and 3+3+3 add up to nine in this given instance, so A, B and F should go together, and C, D and E. Although this particular piece of protocol is only weak evidence for this strategy, stronger evidence for the new strategy can be found in problem R24, two problems later.

Problem R24 Time 93 seconds

Protocol		Analysis	WM
1.	two workers with each nine hours,	Read(work2); Read(time9)	work2; time9
2.	let's look at the hours...	Meta	
3.	seven plus two can again be nine,	Read(e7); Read(a2); Read(b2); Inf(e7; a2; b2 → same(2,7))	same(2,7)

2 workers
9 hours
A 2 hours
B 2 hours
C 3 hours
D 4 hours
E 7 hours

A before B
C before D
C before B

Solution:
EB
ACD

Problem R24 Time 93 seconds

Protocol		Analysis	WM
4.	Four, three and two equals nine,	Read(d4); Read(c4); Read(b2); Read(a2); Inf(d4; c3; b2; a2 → same(2,3,4))	same(2,7); same(2,3,4)
5.	So that seven has to go with A or B.	Inf(same(2,7); e7 → same(2,e7))	same(2,e7)

After problem R24, the participant uses this strategy at the start of every new problem, a clear indication that this new strategy has been incorporated in the general problem solving method. In the problem A29, the strategy has become routine, and the participant can even recognize whether or not the strategy is useful.

Problem A29 Time:128 seconds

Protocol		Analysis	WM
1.	eehm, there are two workers with each six hours.	Read(work2; time6)	work2; time6
2.	It may be the case that E and F go together, because three plus three,	Inf(e3; f3; time6 → same(e,f))	same(e,f)
3.	And DCBA, two, two, two, one hour.	Inf(d2; c3; b1; a1; time6 → same(a,b,c,d))	same(e,f); same(a,b,c,d)
4.	It can also be the case that... Well, anything can be the case.	Meta	

2 workers
6 hours
A 1 hour
B 1 hour
C 2 hours
D 2 hours
E 3 hours
F 3 hours

A before D
B before A
E before C
F before C

Solution:
EAD
BFC

Summary of the qualitative analysis

The most interesting aspect that can be found in the protocol of participant 2 is the fact that she learns two new strategies to solve scheduling problems. The first time these strategies surface is after one or more unsuccessful search attempts. It is quite probable that the participant discovers the strategy at this point. Later on, they are incorporated in the problem solving process. Since scheduling is intractable, these strategies do not provide an effective procedure to solve the general scheduling problem. Nevertheless, they are useful for a large number of instances of the problem.

Another aspect of the problem solving process is that the participant hardly uses backtracking: she just starts all over again. On the other hand, she does keep track of what she does somehow, since a renewed search attempt is almost always a variation on the previous attempt. A final very obvious aspect is the role of rehearsal.

The participant uses rehearsal quite extensively to keep partial solutions active in memory. Interleaving rehearsal with other aspects of processing requires planning as well. The participant not only has to create a schedule for the workers in the scheduling problem, she has to schedule her own activities as well.

Quantitative analysis

To get a more reliable picture of the ideas outlined in the previous section, a simplified version of the analysis has been carried out for all protocols. All main analyses have been done on the seven interpretable protocols from group 2. Occasionally we will also look at the two protocols from group 1. Two observers, a professor in computer science and a graduate student in psychology, were asked to score the protocols according to the following categories:

- Simple inferences, defined by the fact that two or less constraints are involved. Constraints are all aspects of the task, e.g. "A before C" constraints, the fact that task C takes two hours, and any constraints the participants themselves have derived.
- Complex inferences that resemble the fit-the-hours strategy
- Complex inferences that derive the fact that two tasks should be assigned to different workers
- Complex inferences that derive the fact that two or more tasks should be assigned to the same worker
- Complex inferences that derive the fact that some task should be at the beginning of the schedule, at the end of the schedule, or somewhere in the middle
- Complex inferences that do not fit in with any of the previous categories
- Counting, if the participant uses counting to do addition

The last category requires some more explanation. It turned out some of the participants sometimes used counting as a strategy to do addition. This strategy is normally found only in children who have not yet memorized all addition facts. A possible explanation is that in situations where working memory demands are high, counting is a procedure that is less likely to disrupt the contents of working memory than retrieving a fact.

After the observers had scored the protocols, the correspondence was calculated. Correspondence is expressed using the kappa-measure (van Someren, Barnard & Sandberg, 1994), which corrects for the expected correspondence. The kappa measure turned out to be 0.61. According to van Someren et al., kappa should at least be 0.70. Closer inspection of the categories, however, revealed that simple inferences and miscellaneous complex inferences were scored very unreliably. Furthermore, the fit-the-hours strategy and inferences that two or more tasks should be assigned

	Observer 1						
Observer 2	Blnk	Ch	Cdiff	Cfirst	Clast	Cmid	Cnt
No score or removed (Blnk)	2114	8	5	6	3	2	1
Fit-the-hours strategy (Ch)	11	32	1				
Different worker (Cdiff)	5		27				
Assign first (Cfirst)	24	1	3	15			
Assign last (Clast)	9				16		
Assign middle (Cmid)	1					2	
Use counting to add (Cnt)	1						14

Figure 3.11. Correspondence between the two observers

to the same worker were hard to distinguish. So, the simple inferences and the miscellaneous complex inferences were removed from the analysis, and inferences that tasks should be done by the same worker were collapsed with the fit-the-hours category. This resulted in the correspondence table in figure 3.11, and a kappa of 0.72, which is an acceptable value. The entry in the Blnk/Blnk cell (2114) of the figure is high due to the fact that most entries in the protocol were not classified, since they contained no apparent inferences (or counting events). Figure 3.11 also shows that the fit-the-hours and the different-worker strategies are most prominent among the complex inference strategies. So, the two strategies we found in participant 2 are also the main strategies found in the rest of the participants. The analyses of both observers were combined into a single analysis using only the complex inferences both observers agreed on. If both observers agreed on a complex inference, but used different categories (5 cases), the experimenter chose the most appropriate category.

One would expect that if participants learn new strategies during problem solving, the number of complex inferences increases with practice. Figure 3.12 shows this is indeed the case: in the first problem the participants use 0.5 complex inferences on average to reach the solution, which increases to more than 2 inferences in instance 8, dropping back slightly in the last two instances.

Figure 3.13 shows how the two most prominent strategies are distributed over the individual participants. The black boxes mark the use of the fit-the-hours strategy for a certain instance, while the grey boxes indicate the use of the different-worker strategy. As is evident in the figure, some of the aspects witnessed in the analysis of participant 2 are also evident in other participants. Some of the participants also integrate the fit-the-hours strategy in their standard search strategy, notably participants 2, 6, 7 and 11. The same is true for the different-worker strategy. This is less evident in the figure, since the different-worker strategy cannot be used

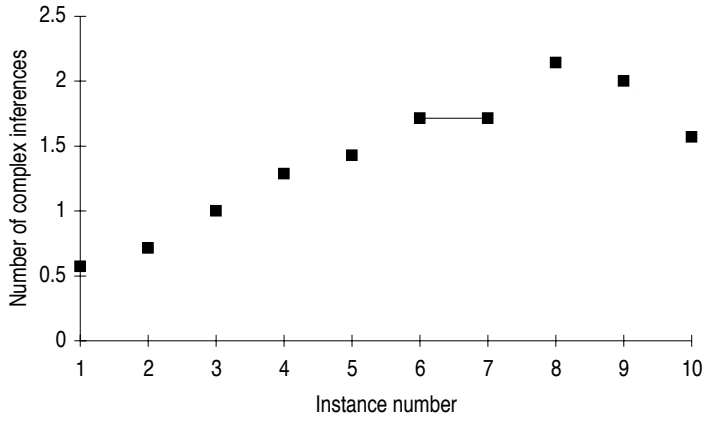


Figure 3.12. Number of complex inferences for each instance

successfully for every instance. Participants 1, 2, 4, 6 and 7, however, show consistent use of it.

The use of counting to add numbers was use extensively by participant 1, who used it 13 times to add numbers. Three participants, 6, 11 and 12, only used it once, and the other participants showed no evidence for the use of counting to do addition. Although this aspect has no relevance to the rest of the discussion here, I will return to this matter briefly in chapter 5.

3.5 Conclusions

The analyses presented in this chapter only scratch the surface of all that is going on during problem solving. But it is a study in the spirit of Alan Newell, in which we try to learn as much as possible by studying a single complex task. It is clear that learning in problem solving cannot be accounted for by a simple, one-principle theory. Nevertheless many of the aspects found in the analysis support the general outline discussed in chapter 1. There is evidence for the use of problem-space search, but also for qualitative insight-like changes in problem-solving approach. Participants discover and refine new strategies as the experiment proceeds, enabling them to eventually handle even more complex problems.

The next two chapters will examine details of the aspects of learning in problem solving that have been found in this chapter. The strategy is to formulate a model based on intuitions gained from the scheduling experiment, and test these models on more simple experiments and data from the literature.

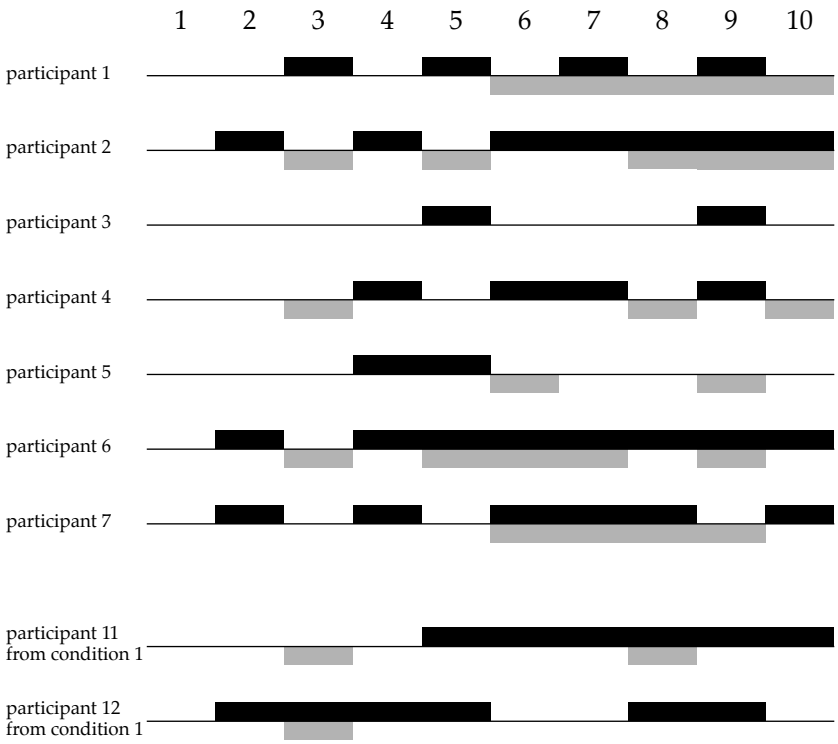


Figure 3.13. Strategy use plotted for individual participants. Black boxes indicate the fit-the-hours strategy was used for that particular trial, and grey boxes indicate the use of the different-worker strategy.

Maintaining the current problem context

One important aspect of problem solving, which becomes most apparent if problem solving has to be done entirely by heart, is to maintain the current problem context in memory. Protocols show participants have great difficulty with this aspect of problem solving, since they give a lot of attention to rehearsing their current schedule, but nevertheless make many mistakes with it. Clearly there is more going on than just pure rational search. But participants have to do more than just rehearsal, they also have to keep track of other dynamic aspects of problem solving, such as what they have already tried and what new constraints they have already derived. Coordinating knowledge in the current problem context has aspects of implicit and explicit learning. Rehearsal is clearly an intentional, explicit aspect, but

a growing sense of the inadequacy of search given the current knowledge, is more implicit. Chapter 4 will elaborate on this topic, and will discuss a model of rehearsal.

The role of insight and rule learning
Protocol analysis showed clear evidence for the emergence of two distinct problem-solving strategies, the fit-the-hours strategy and the different-worker strategy. The single protocol that was analyzed in detail also showed a learning pattern that resembled the exploration-impasse-insight-resolution scheme posed by insight theory. In chapter 5, a rational basis for this pattern of problem solving will be sought. Additionally, the problem of how new rules can be learned during an insight episode will be discussed.

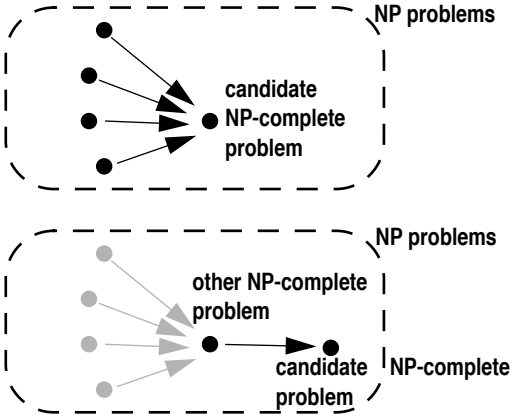
3.6 Appendix: Proof of NP-completeness of fully-filled precedence constrained scheduling

A proof of NP-completeness consists of two steps. First, the problem must be in NP, and secondly it must be possible to polynomially reduce any NP problem to the candidate NP-complete problem. Reducing a problem A to problem B means that there exists a transformation function T that takes an arbitrary instance of problem A and returns an instance of problem B, satisfying the condition that the solutions to both instances are the same. To polynomially reduce A to B means that the transformation function T must have a polynomial time complexity.

To prove that any NP problem can be reduced to a candidate NP-complete problem is very hard. Fortunately, there is a much easier method. It is sufficient to prove that an arbitrary other NP-complete problem can be reduced to the candidate NP-complete problem. Since all NP problems can be reduced to this other NP-complete problem, any NP problem can be reduced to the candidate NP-complete problem in two steps (figure 3.14).

The original definition of PCS assumes all tasks have a duration of one hour. So FF-PCS also differs in this respect, since tasks can have an arbitrary duration. The NP-completeness proof will have to take this into account as well. The formal definition of PCS is as follows (from Garey & Johnson, 1979).

Definition of PCS
An instance is a set T of tasks, each having length $l(t) = 1$, a number $m \in \mathbb{N}^+$ of workers, a partial order $<$ on T, and an overall deadline $D \in \mathbb{N}^+$. The question to be answered for each instance is: is there an m-worker schedule for T that meets the overall deadline D, i.e., a function $\sigma: T \rightarrow \mathbb{N}$ such that, for all $u \geq 0$, the number of



To prove a problem is NP-complete, one has to prove any NP problem can be reduced to it.

The easiest way to do this is through another NP-complete problem, since it requires only a single reduction.

Figure 3.14. How to prove any problem can be reduced to an NP-complete problem

tasks $t \in T$ for which $\sigma(t) \le u < \sigma(t) + l(t)$ is no more than m and such that, for all $t \in T$, $\sigma(t) + l(t) \le D$, and obeys the precedence constraints, i.e., such that $t < t'$ implies $\sigma(t') \ge \sigma(t) + l(t)$?

Definition of FF-PCS

FF-PCS differs from PCS with respect to the following two points: it allows arbitrary lengths of tasks, so $l(t) \in \mathbb{N}^+$, and it requires the schedule to be filled, so $\sum_{t \in T} l(t) = mD$.

Theorem
FF-PCS is NP-complete.

Proof
First, we have to prove that FF-PCS is NP. The problem is NP, if there is an algorithm consisting of two parts: a non-deterministic part that "guesses" a schedule, and a deterministic algorithm of polynomial time complexity that checks whether this schedule meets all the constraints. Both parts of this algorithm are easy: guessing a schedule is just filling $\sigma(t)$ with arbitrary values, and checking the schedule means checking the precedence constraints (one check for each constraint), whether all tasks end before the deadline (one check for each task), and whether there are no overlapping tasks (no more checks required than the multiplication of the number of tasks and the deadline D).

The second part of the proof involves the reduction of PCS, a known NP-complete problem, to FF-PCS. So, given an instance I of PCS, we have to show how this

instance can be transformed into an instance I' of FF-PCS, and have to prove that if there is a schedule for I, there is a schedule for I', and if there is no schedule for I, there is no schedule for I' either.

The best way to understand this transformation is to think of a PCS schedule as a schedule in which some of the workers have time left in which they have nothing to do. Now suppose we also want to schedule this "free time". This will not increase or decrease the difficulty of the process, since there are no constraints on free time, it is just that any time that is left over is now officially called a "free-time" task.

For the transformation function we will distinguish three possible cases, two of which are trivial. The first is the case in which $\sum_{t \in T} l(t) > mD$, so the total duration of all tasks exceeds the total time workers have. In that case there can never be a solution. So we can just transform all instances to a single FF-PCS instance for which we know no schedule is possible. This transformation satisfies the condition, since for all instances there is no schedule. The second trivial case is when $\sum_{t \in T} l(t) = mD$: the total duration equals the total time the workers have. In this case the PCS instance already is a FF-PCS instance, so we can just use identity as the transformation function. The third case is when $\sum_{t \in T} l(t) < mD$, the case in which there is more available time than it takes to do all the tasks. The idea is to "fill up" the rest of the schedule with tasks of length one ("free-time" tasks), on which we do not impose any precedence constraints. These extra tasks can fill in the rest of the schedule. So given an instance I of PCS, we create I' by adding $mD - \sum_{t \in T} l(t)$ tasks to T, each of which has $l(t) = 1$. No precedence constraints are imposed on these new tasks. If there is no schedule for I, neither will there be one for I', since it only has more tasks to schedule. If there is a schedule for I, it has exactly $mD - \sum_{t \in T} l(t)$ points in time left for which the schedule has less than m scheduled tasks that can be filled with the added tasks in I'. Since no precedence constraints are imposed on these tasks, they can be scheduled anywhere.

CHAPTER 4

Implicit versus Explicit Learning

The goal of this chapter is to arrive at a theory of implicit and explicit learning without introducing new theoretical entities. The basis for this theory will be the ACT-R architecture. The ACT-R theory, of course, also uses multiple theoretical entities. As we will see, none of these correspond directly to the notions of implicit and explicit learning, but together they can provide an explanation. This chapter will start with a general discussion about implicit and explicit learning. One experiment that is often quoted in the context of implicit learning is a dissociation experiment by Tulving, Schacter and Stark (1982). An ACT-R model is presented that can be used to explain their results. The model also serves as a basis for a more general discussion on how implicit learning and explicit learning can be understood in terms of ACT-R. The remainder of the chapter is used to discuss a particular example of explicit learning: rehearsal. Rehearsal is often studied using the free-recall task. By examining free-recall in several different situations, we may conclude that the primacy effect is mainly an effect of explicit learning, while the recency effect can be explained by implicit learning.

4.1 Introduction

In chapter 1 I have discussed Alan Newell's criticism of psychological research, in which he mocked the simplistic conceptualization of the complexity of human cognition in terms of binary oppositions. Since 1973 a new opposition has become popular in cognitive psychology: the distinction between *implicit* and *explicit learning* or *implicit* and *explicit memory*. Although the term implicit memory was already proposed by Reber in 1967, the topic became popular by the end of the eighties. Before implicit learning research became popular, most memory research paradigms were based on either recognition or recall. Both in recognition and recall, participants first have to study some materials, and are tested later on. These types of experiments offer many insights into the nature of human memory, but tend to bias theories of memory. For example, in the famous dual-store memory theory by Atkinson and Shiffrin (1968), a major role for storing information in long-term memory is attributed to rehearsal, the mental process of sub-vocally repeating information. The dual-store theory was able to explain many of the recognition and recall experiments. A very powerful but false prediction was however neglected: the fact that no rehearsal implies no storage in long-term memory. As we will see shortly, learning may even take place without awareness. The dual-store theory overestimated the importance of rehearsal as a memory process, because it used recognition and recall as a basis. In both types of experiments, participants were told explicitly they had to memorize certain items.

Reber's 1967 experiments departed from this experimental paradigm, and investigated what people learn without being aware of what they have to learn. The experiment he introduced, and which has been replicated many times in many

variations, is *artificial grammar learning*. In this experiment participants first study a list of strings that has been generated by a finite-state automaton based on an artificial grammar. After this study phase, participants were told the strings they had studied were words generated by a grammar. In the following test phase, they were presented with new strings generated using the same grammar, mixed with random strings and strings with subtle errors in them. Participants had to figure out which new strings were generated by the grammar, and which were not. It turned out that participants are surprisingly good at this task, and classify the new strings not perfectly, but well above chance level. Since none of the strings that were originally memorized were presented in the test phase, and participants were not aware of the fact that there was any systematicity in the learned strings, they somehow must have learned more than just the literal strings. Reber coined the term implicit learning to describe this additional, unintentional aspect of learning. Additional studies show that although participants perform well on this task, they can not explicitly state the rules of the grammar.

The idea that participants must learn to predict the behavior of a final-state automaton has been used in several other research paradigms. An example of one of these paradigms is *dynamic system control*, in which participants have to learn to control a complex system. An example is an experiment by Berry and Broadbent (1984), which involves a scenario in which participants have to learn to control a sugar factory. The Sugar Factory computer simulation they used is a dynamic system in which participants have to control sugar production by setting the number of workers. Since the relationship between input and output is highly non-linear, it is almost impossible for participants to discover the rule that governs the system. Nevertheless participants learn adequate control quite quickly, although they are not able to state the underlying rules of the system. A model of this experiment will be discussed in chapter 6.

Another type of research that deviates from traditional memory research is the *dissociation paradigm*. An example of this type of research is an experiment reported by Tulving, Schacter, and Stark (1982). In this experiment participants first had to study a list of 96 words. They were subsequently tested using two different tests, an implicit and an explicit test. The first, explicit, test was a simple recognition test, in which the participant was asked whether or not a certain word was in the study list or not. The second, implicit, test was a word-completion task. In this case participants were presented with a word fragment which they had to complete, for example A_ _ A _ _ IN (answer: ASSASSIN). Some of the fragments originated from the studied list, and others were from words not previously studied. Each participant had to do each test twice: an hour after the study phase and a week after the study phase. Figure 4.1 shows the results. One hour after studying the words, participants recognize 58% of the items correctly (this percentage is corrected for guessing). After a week, performance has dropped considerably to 24%. The implicit word-completion task, however, shows a totally different picture. Studying words

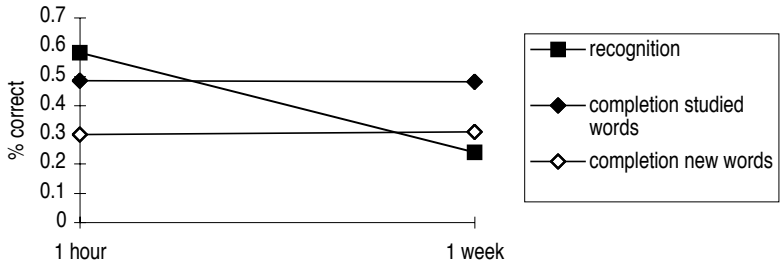


Figure 4.1. Results of the Tulving, Schacter & Stark experiment: performance on the explicit recognition test degrades in a week, while performance on the implicit word completion task remains constant.

improves performance on this test: after one hour word-completion was accurate for studied words in 49% of the cases, while new words were only completed successfully in 30% of the cases. This advantage does not degrade with time, since after a week performance on the word-completion task is still the same. The discrepancy between the two tasks is called a dissociation: while one type of information, the fact that a word has been studied in the context of the experiment, degrades over time, other, subtly different, information seems not to suffer from any decay in time at all.

In the example above the dissociation is caused by time: one type of performance did suffer due to the passage of time, while another did not. There are other types of dissociations, for example due to brain damage. A study by Warrington and Weiskrantz (1970) reveals that patients suffering from amnesia perform much worse compared to healthy people on explicit tests like recognition and recall. On implicit tests like word completion, their performance equals control participants.

What do experiments such as artificial-grammar learning and dissociation learning exactly prove? At least they show the inadequacy of the classical recognition/recall paradigms, and also show that the "no rehearsal no learning" prediction of the dual-store model does not hold. But, probably to Alan Newell's horror, psychologists turned the new phenomena into a new binary opposition, and, even worse, posed two binary opposite theories (the systems and the processing theory) to explain the distinction. Implicit and explicit learning were proposed as two distinct types of learning, each having its own mechanisms and needing its own theoretical framework. Explicit learning was associated with all the old memory research, but implicit learning, the new kid on the block, promised to be a new unexplored domain of countless experiments.

What makes implicit learning different from explicit learning? The dissociation experiments show that implicit learning is somehow more robust than explicit learning, since neither brain damage nor the passage of time seems to affect it.

Implicit learning is more robust in other aspects as well. McGeorge, Crawford and Kelly (1997) have shown that explicit learning is dependent on age and intelligence, while implicit learning is not. Participants that score higher on an IQ-test also perform better on explicit memory tests, and performance of older participants on the explicit test is worse than the performance of younger participants. Implicit learning on the other hand is hardly affected, either by IQ or age.

Another aspect of implicit learning, even used by some researchers as the defining quality, is that consciousness or awareness does not seem to play a role in it. Implicit learning is therefore sometimes called *unconscious learning*, as evidenced by the fact that although the participants can not verbalize any knowledge about the task, their performance increases nevertheless. In Reber's artificial grammar participants were not able to state any of the grammar rules, but could categorize the strings anyway. In the Tulving experiment, participants had forgotten that they had studied a particular word after a week, but managed to use them for word-completion anyway. The notion of consciousness is, however, not unproblematic, as pointed out by Shanks and St. John (1994). In the artificial grammar experiments participants were not able to express any of the rules of the grammar. But they were aware of the fact that certain combinations of letters were more likely in grammatical than in ungrammatical strings, something that could at least explain some of their increased performance. A "safer" version of the unconsciousness aspect of implicit learning is to define implicit learning as unintentional learning, learning that is not tied to goals. In artificial grammar learning and in the Tulving experiment, participants had to memorize words or strings for later recall or recognition, not with the intention to do word-completion or to figure out a grammar. In this sense implicit learning can be seen as a "by-product" of normal information processing, while in explicit learning information processing is aimed at learning, comprehending or memorizing something.

There are two opposing theories that attempt to explain the differences between implicit and explicit learning: the systems theory and the processing theory. According to the systems theory, put forward by Squire (Squire & Knowlton, 1995), there are two different memory systems, an implicit and an explicit memory system, represented in separate structures in the brain. The fact that amnesiacs perform worse than controls on explicit tasks but not on implicit tasks can simply be explained by the fact that their explicit memory is damaged but their implicit memory is intact. Explicit memory is conscious memory, implicit memory is unconscious. Information in explicit memory decays with time, while information in implicit memory stays put. This also corresponds well with the folk-psychology idea that all our experiences are stored in unconscious memory.

The processing theory of implicit learning by Roediger (1990) assumes that there is a distinction between two types of processes: data-driven processes and conceptually driven processes. Data-driven processes are triggered by external

stimuli and can be associated with tests of implicit memory. For example, in the word-completion task part of the pattern is given. This part of the data actively facilitates the retrieval of the whole pattern. In the recognition test on the other hand, a connection between a word and an episodic event must be verified, so has a more conceptual nature. Conceptually driven processes are initiated by the participant and lead to explicit learning. According to the processes theory, memory performance will be best if the processing required on the test is the same as the processing required in the learning phase.

The problem with both the systems and the processing theory is that a distinction found in empirical data is explained by proposing two different theoretical entities, either two systems or two types of processing. From a scientific point of view this is a weak explanation that furthermore offers no insights in what the difference is between implicit and explicit learning. The evidence for separate entities is not final either. There are many examples of dissociations in which explicit learning is impaired while implicit learning is intact. If each type of learning is associated with its own theoretical entity, however, a so-called crossed double dissociation has to be found. In a crossed double dissociation, two experimental variables have to be found that have opposite effects on the implicit and the explicit test. A dissociation like this has never been found (Cleeremans, Destrebecqz & Boyer, 1998). To quote Cleeremans (1997, page 215):

> With the exception of Hayes and Broadbent (1988) that has failed to be replicated so far, such a [crossed double dissociation] has never been observed in implicit learning situations. [...] the fact that no crossed double dissociation has ever been satisfactorily obtained in implicit learning research has often been used by other authors (e.g. Shanks and St John, 1994) as an argument to deny the existence of implicit learning as an independent and autonomous process.

Evidence from studies with patients isn't strong either: both patients of Huntington's disease (Heindel, Butters & Salmon, 1988) and Parkinson's disease (Saint-Cyr, Taylor & Lang, 1988) have severe difficulty in learning motor skills, while showing intact performance on recall and recognition. Motor skills are usually considered procedural skills. Since people do not have conscious access to their procedural skills the associated learning process can be considered implicit. The problems these particular patients have, however, seem to limit themselves to the motor domain, so a generalization to implicit learning in general is unwarranted.

4.2 A model of the dissociation experiment

Tulving's dissociation experiment consists of three separate activities, each of which is modeled by a small set of production rules: studying the list of words, the recognition test and the word-completion test. First, the list of words has to be

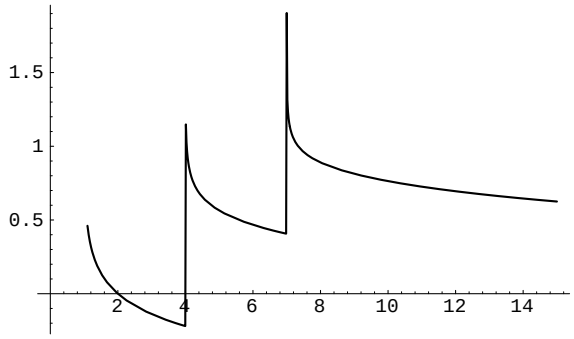


Figure 4.2. Example of the activation for a chunk accessed at time 1, 4 and 7.

studied. In the experiment, every 5 seconds a word is presented. Since participants were only told they were involved in a memory experiment, they had no direct clue on what they had to do exactly with the words. It is therefore a safe assumption that participants will just rehearse the word and the fact that they have seen the word in the current context. This is easily accomplished in ACT-R: a first production rule creates a declarative recognition chunk that points to the word to be studied and to the current context. The recognition chunk can be considered as an episodic memory trace. A second rule keeps retrieving the chunk that represents the word and the recognition chunk until the next word is presented. Due to ACT-R's base-level learning, the activation of a chunk is increased each time it is retrieved. The base-level activation at a certain time t can be calculated using the following equation:

$$B_i(t) = \log \sum_{j=1}^{n} (t - t_j)^{-d} + B \qquad \textbf{(4.1)}$$

In this formula, n is the number of times a chunk has been retrieved from memory, and t_j represents the time of each retrieval. The longer ago a retrieval was, the less it contributes to the activation. B and d are constants. Figure 4.2 shows an example of the behavior of this function, in which the activation of a chunk is plotted that is accessed at time 1, 4 and 7.

When the rehearsal production rule retrieves the recognition chunk and the chunk that represents the word itself, activations of both chunks are increased considerably, because n is increased in the formula, and the new t_j's are all still close to t. There is, however, a difference between the activation of the recognition chunk

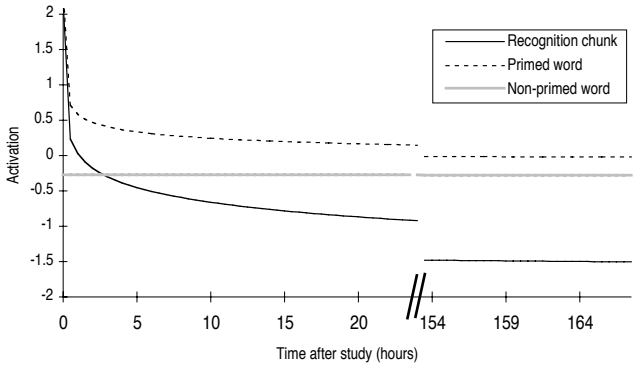


Figure 4.3. Development of activation values for recognition chunks, primed words and non-primed words in the course of the week after the study phase of the experiment.

and the word chunk. The recognition chunk has just been added to declarative memory, so has no previous history of activations. This means that the activation of the recognition chunk is based solely on the few rehearsals in the context of the experiment. The word chunk, however, was already present in declarative memory, and already has a history of past use. In the model, this is simulated by assuming that words have been accessed on average 150 times, spread evenly over the past ten years, producing a low, but stable activation value. Some fixed activation noise in the model assures that all words have slightly different activation values. The difference between recognition and word chunks means that activations will also develop differently in the time period after studying the words. As figure 4.3 shows, both the word chunks and the recognition chunks start at a high level of activation. The activation of recognition chunks, however, decays faster due to the fact that they have no previous history.

In the recognition test the question must be answered whether or not a particular word has been studied in the study phase. In terms of the model this means that given a particular word chunk and a particular context chunk, a recognition chunk must be retrieved that connects the two. This is handled by two production rules. The first rule tries to retrieve the recognition chunk and answers “yes” when it succeeds. The second rule, which may fire if the first rule fails, just answers “no”. This model is not entirely faithful, since it does not model the event in which a word that has not been studied is mistaken for one that has been studied. This can be modeled in ACT-R using partial matching, but this has not been done in the current model (partial matching has briefly been introduced in chapter 2, but will used in the Sugar-Factory model in chapter 6). Failure to recognize a word that has been studied is due to the fact that the activation of the recognition chunk has become too low, since ACT-R cannot retrieve chunks with activations below the retrieval threshold.

In a recognition test, the indices to retrieve the right chunk are clear enough: the word and the study event. This is not the case in the word-completion task, where only a part of the word is given and the rest has to be retrieved. In order to retrieve the word that fits the pattern A _ _ A _ _ IN, ideally a production rule is needed that matches the first, fourth, seventh and eighth letter, and tries to retrieve a word that fits. The problem with this solution is that a production rule is needed for any combination of letters, which would mean 256 production rules if we would restrict ourselves to just 8 letter words. A solution that only requires a few production rules is to retrieve a word using only one or two letters, and compare if the retrieved word matches the rest of the letters. If it does, a solution has been found, if it does not, the model gives up. Alternatively, the model might have a few tries before giving up, but that aspect has not been modeled. One of the matching rules is as follows:

```
IF
      the goal is to complete a word fragment AND
      the first letter of the fragment is l1 AND
      the second letter of the fragment is l2 AND
      there is a word w that has l1 as its first letter AND
      has l2 as its second letter
THEN
      mark w as a candidate solution in the goal
```

This rule tries to find a word that matches at least the first two letters of the pattern. This rule will not work for the A _ _ A _ _ IN, because the second letter is unknown, but it will work if the first two letters are given.

Although both recognition and word completion require some declarative retrieval, they differ with respect to the source of errors. In the recognition test, it may be the case that a recognition chunk is no longer retrievable due to low activation. In the word-completion test interference with other words is the major source of errors. Words that are primed in the learning phase of the experiment get an activation advantage over words that are not primed. This advantage may persist over longer periods of time, as is indicated in figure 4.3. This difference between the two tasks may well be the real explanation for the dissociation. Figure 4.4a demonstrates that the model indeed behaves in a way that is comparable to human data. The main parameter that was manipulated to achieve the fit is the base-level learning decay (parameter d in equation 4.1). The recommended value for this parameter is 0.5, but this turned out to be a poor choice to explain long-term learning, since in a week ACT-R had forgotten everything. Instead the value of 0.3 has been used. Other parameters that have been manipulated, such as the retrieval threshold and the activation noise, did have small effects on the actual values of data points, but did not change the main dissociation effect.

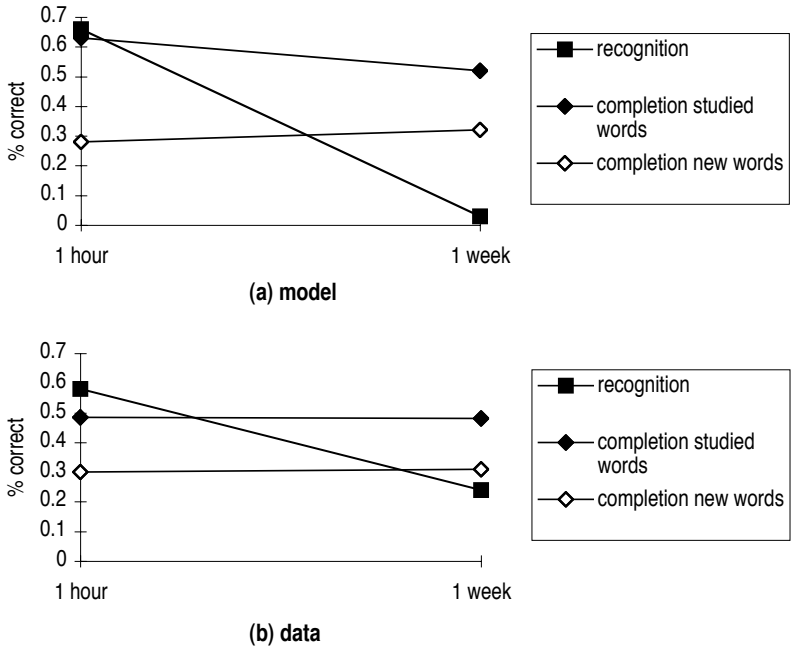


Figure 4.4. Results of the model of the dissociation experiment (a). The data are repeated in (b).

The interesting aspect of this model is the fact that although it exhibits a dissociation, it nevertheless has no separate theoretical constructs to explain this difference. Both types of information are represented in the same memory system by the same memory process. The dissociation can be explained by the characteristics of the tasks themselves, rather than by hypothesized constructs. What is the difference between recognition and word-completion? To get a broader view on this question, we first have to review the notion of activation. Activation in ACT-R is an estimate of the log odds that a certain chunk is needed in the current context. This estimate is used in ACT-R for two purposes:

- If there are two or more possible candidates for retrieval by the production rule that is currently matched, the candidate with the highest odds is chosen.
- If the odds of needing a certain chunk are too low, the potential gain of retrieving it is not worth the effort.

If we look at the study task the participants have to do, we have to compare it to the situation in which people normally read words. In normal situations, it is not useful to remember in which particular context a word has been read. It is, however, useful to keep track of how often a word is used or encountered, since high-frequency words are more important than low-frequency words. So, if someone read low-

frequency words in a normal setting, he would typically not remember the event of reading the word itself, and would probably only update the frequency information of that word. The Tulving experiment is not a normal situation, it is a memory experiment. In order to meet the, at that point, unknown criteria of the memory experiment, the participant intentionally influences the normal learning scheme by rehearsing information. Rehearsal in this context means: intentionally increasing the number of retrievals of certain chunks, thereby artificially increasing the odds-of-being-needed of the chunk. As a consequence, the recognition chunk that stores the information that the word has been studied can still be recovered one hour after the study phase. A unintended by-product of rehearsal is that the frequency information of the studied words is increased as well. Since low-frequency words are used, the extra retrievals due to rehearsal have a significant impact on this estimate. It is this frequency information that the word-completion production rules need in order to select candidates, and which can be used as an explanation why studied words are completed better than words that are not studied, even after a week.

In the previous discussion the important difference between normal situations and a memory experiment is intentionality. In the introduction I have already noted that intentionality might be a key notion in the discussion. In the next section I will explain how this idea can be worked out in terms of the ACT-R theory.

4.3 An ACT-R theory of implicit and explicit learning

In the introduction I mentioned intentionality might be a good starting point to understand the nature of the difference between implicit and explicit learning. An advantage of using intentionality is that it can easily be operationalized in terms of ACT-R. Intentionality in terms of ACT-R means: tied to a goal. In the case of learning words for later recognition, as in the Tulving experiment, the intention of the participant is to memorize the words. If we look at the learning mechanisms in ACT-R, none of them is principally tied to intentions. Although the base-level learning mechanism may be used in the context of a memorization goal, it is not its basic function. Its basic function is to keep track of the odds that chunks are needed, a function that is normally performed unintentionally and unconsciously. The same can be said about all learning mechanisms in ACT-R: they are at work all the time, and are basically not tied to intentions. In a sense all learning in ACT-R is implicit learning. This idea is consistent with other properties of implicit learning. Implicit learning does not change much by ageing, and individual differences are small. This is exactly what we want for basic mechanisms in an architecture for cognition, since it is a theory about what people have in common and not about what sets them apart. The fact that implicit learning is not easily impaired due to brain damage also favors the architectural mechanism view: the basic way the brain works shouldn't change due to damage.

What is explicit learning? The position I would like to defend is that explicit learning is a form of implicit learning. But while implicit learning is a by-product of normal processing, explicit learning is the by-product of specific learning goals. Where normal processing would retrieve a chunk representing a word only once, an explicit learning goal may retrieve it a number of times, not because it is necessary for processing, but just to put the implicit learning mechanisms to work. Although we have no direct conscious access to the base-level learning mechanisms itself, we may have found out, due to experience that repeating a word helps remembering it. Instead of being another type of learning, explicit learning is just a set of strategies to make the best possible use of the implicit mechanisms. Explicit learning is therefore not a part of the architecture of cognition, but is rather produced by knowledge that is represented in the memory systems of that architecture. This idea also corresponds well with properties of explicit learning: since the knowledge corresponding to it has to be learned itself, one can expect large individual differences due to intelligence and development. Similar observations can be made with respect to brain damage. If implicit learning is a fundamental property of the brain, it will not be easy to damage it. Explicit learning, on the other hand, consists of knowledge. Brain damage may cause this knowledge to be lost, or disrupt successful usage of this knowledge.

In the case of the Tulving experiment, the recognition task is an explicit task only because participants suspect either recognition or recall if they are told they are involved in a memory experiment. If one explained the word-completion task to participants at the start of the experiment, and told them they were supposed to do this task after the study phase, it would turn into an explicit task. The participant has several options: she can either stick to a rehearsal strategy, or attempt some more clever memory strategy, for example by explicitly memorizing characteristic fragments of words. The choice of strategy will have a large impact on performance. The original rehearsal strategy will of course still exhibit the assumed characteristics of implicit learning, while the fragment-memorization strategy, if it works at all, will probably suffer from the same fast decay that is supposed to characterize explicit learning. We might even be able to find a dissociation within the same task in healthy participants.

In Reber's artificial grammar and Berry and Broadbent's sugar factory, participants' performance increases, although they are not capable of formulating any explicit rule-like knowledge about the task. In both cases, it is very hard to find the real rules: deriving grammars from examples is a very difficult task, and the non-linear character, the randomness and the limited means of control in the sugar factory make it almost impossible for participants to derive rules within the limited time of the experiment. As a consequence, explicit strategies that are usually successful in detecting regularities will fail. Nevertheless there is also implicit learning going on. For example in the sugar factory task, which I will discuss in detail in chapter 6, each time the participant sets the controls of the factory and perceives an outcome, a chunk recording this information is added to declarative memory. This is not done

intentionally, but rather because all popped goals are stored. It will turn out that this information alone can account for the improvement participants show on the task.

In the remainder of this chapter and in the next two chapters, I will explore the implicit/explicit distinction based on the idea that implicit learning is based on mechanisms of the architecture, and explicit learning is the application of learning strategies. In chapter 5, I will discuss explicit strategies that learn new production rules, and how an increase in the number of strategies can explain the difference between small children and adults on a classification task. In chapter 6, I will describe how the implicit/explicit learning debate can be related to another debate in the learning literature: whether new skills are learned by accumulating examples, or by deriving general rules. The remainder of this chapter is devoted to one of the issues stated in the previous chapter: a model of rehearsal. This model will be discussed in the context of the free-recall task, a classical paradigm to study rehearsal.

4.4 A model of rehearsal and free recall

The model discussed in this section is the first model I made in ACT-R. As a consequence, the model is based on an old version of ACT-R (2.0), which on the one hand included features that have since been removed, but on the other hand did not include all that is currently part of ACT-R. I further chose to implement verbal rehearsal using a separate phonological loop, based on Baddeley's evidence for this kind of structure. If I were to model rehearsal again, I probably would be more hesitant to add extra structures to the architecture. Recently, the CMU group (Anderson, Bothell, Lebiere & Matessa, 1998) has also modeled free recall as part of a broader project on list learning. Their model did not use an explicit phonological loop. They, however, implemented a phonological-loop-style memory structure within declarative memory that did the same job.

As we have seen in the introduction, rehearsal has been studied extensively in the seventies in the context of the dual-store memory theory by Atkinson and Shiffrin (1968). One of the experimental tasks used for studying rehearsal is the free-recall task. In this task a list of words, typically containing fifteen to twenty items, is presented at a constant rate to a participant. After presentation, the participant has to recall as many words as possible from the list. Two effects emerge from the results, the primacy effect and the recency effect, respectively referring to the fact that the first and the last few items of the list are recalled better than the rest. The dual-store memory theory can explain both effects: the primacy effect is due to the fact that the first few items in the list are rehearsed more often because they initially don't have to compete for space in short-term memory (STM), and the recency effect is due to the fact that the last few items are still in STM at the moment they have to be recalled.

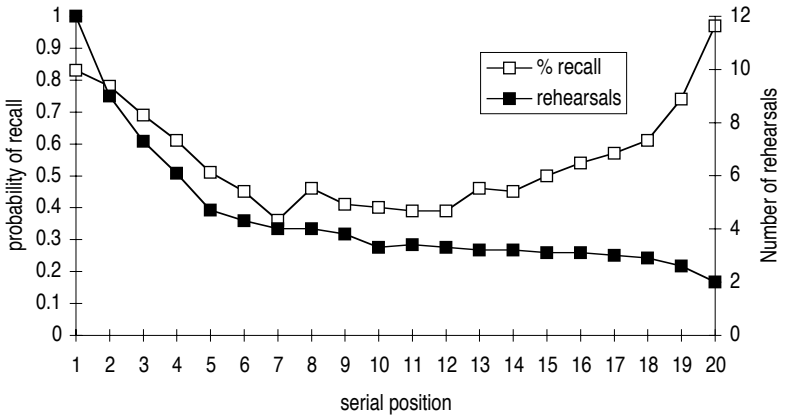


Figure 4.5. The percentage of correctly recalled items and the number of rehearsals (Rundus, 1971).

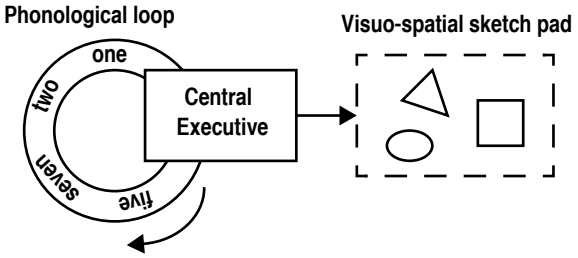


Figure 4.6. Baddeley's theory of working memory

This explanation is confirmed by Rundus (1971), who asked participants to rehearse aloud. The data show that there is a relation between the number of rehearsals and the chance of recall (figure 4.5), at least with respect to the primacy effect.

Since the popularity of the dual-store theory declined, partly because rehearsal turned out to be not the sole mechanism to store information in long-term memory (LTM), less research effort has been put into it. A theory that does involve rehearsal is Baddeley's theory of working memory (Baddeley, 1986). In Baddeley's theory, working memory has a *central executive* and two rehearsal subsystems: the *phonological loop* and the *visuo-spatial sketch pad* (figure 4.6). Both subsystems are used to temporarily store small amounts of phonological and spatial information. The phonological loop is a system that stores up to two seconds of phonological code in a serial fashion. The visuo-spatial sketch pad uses a quasi-visual representation of objects that can be used for spatial reasoning. The visuo-spatial sketch pad can be used to answer questions like: if the triangle is below the square, and the circle is to the right of the square, and the circle is above the cross, is the cross left or right from the triangle?

The phonological loop is the relevant structure for retention in free recall, at least in the overt-rehearsal version by Rundus. Instead of being the process that transfers information from STM to LTM, rehearsal has become a process necessary to maintain items in STM. Whether or not information will also be stored in LTM is not specified by Baddeley's theory, because it is a theory of working memory only. Work by Craik and Lockhart indicates that the extent to which rehearsed information is stored in LTM depends on the amount of processing that needs to be done on individual items (Craik & Lockhart, 1972). This led to the distinction between maintenance rehearsal and elaborate rehearsal. Maintenance rehearsal is used just to retain information for a short time, for example a telephone number that needs to be dialled. During elaborate rehearsal on the other hand further processing is done on the rehearsed information.

Baddeley has gathered extensive empirical evidence for the phonological loop and the visuo-spatial sketch pad. The central executive, however, is a weak point in the theory. It is supposed to be able to contain two or three items, and to control what goes into both subsystems, but it is unclear what representation it uses, and why and when it puts something in either subsystem. The central executive is almost a reference to the rest of information processing, because it not only stores information, it also makes important decisions on what to memorize in what subsystem. Some of these decisions must be deliberately planned, involving knowledge stored in LTM. The problems with the central executive have an obvious reason: somehow the theory of working memory must be tied to the rest of information processing, and the central executive is responsible for this.

The ACT-R theory can be seen as a specification of central information processing that can serve as a means to create models of rehearsal using Baddeley's phonological loop. The role of the central executive is taken care of by the ACT-R architecture.

A model of free recall in ACT-R

To be able to model free recall in ACT-R, we first need some way to do rehearsal. In order to use Baddeley's phonological loop, some assumptions have to be made about the representation of the loop and the interaction with ACT-R. According to Baddeley, the phonological loop has a phonological representation. To be able to interact with the memory of ACT-R, we must assume it is possible to activate a phonological representation given a chunk-like symbolic representation in declarative memory and vice-versa. To simplify matters, we will assume the phonological loop has the following properties:

- The phonological loop is a linear storage buffer with a capacity of 2 seconds of phonologically coded words.
- References to declarative chunks representing pronounceable words can be added to the loop. New references are added to the end of the loop.

- If the capacity of the loop is exceeded, a random word is dropped.
- At any moment the contents of the loop can be rehearsed, which involves entering a subgoal to do this.
- In the rehearsal subgoal the words can just be rehearsed (maintenance rehearsal), or further reasoning can be done with them (elaborate rehearsal).

Implementing a separate structure for rehearsal is at odds with the idea that rehearsal is just a learned strategy. But what if the phonological loop is not primarily a structure of working memory, but rather a buffer to store perceived speech in, or speech that is about to be pronounced? In that case, rehearsal would be a clever strategy of reusing a structure whose original purpose is different.

Once rehearsal is taken care of, a model of free recall is straightforward. During the study phase of the experiment words are read and added to the phonological loop one at a time. In the time between presentations the phonological loop is rehearsed. At the time of recall, words are recalled in order of activation until there are no words left above the retrieval threshold. No attempt is being made to first "empty" the phonological loop at the time of recall, only the last item of the list is retained.

The explanation this model offers for the two prominent effects in free recall, the primacy and the recency effects can now be made clear. The primacy effect can be explained in the same manner as Rundus' explanation: the first few words are rehearsed more often, on average, so are retrieved more often. The recency effect can be explained by the fact that the retrievals are relatively recent, so their impact on the activation is larger.

A positive recency effect can be considered as an implicit learning effect, since its presence is not influenced by strategy. This finding concurs with developmental data. Hagen and Kail (1973) compared free-recall behavior of 7 and 11 year-old children. Although both groups show a recency effect in recall, in the group of younger children the primacy effect is absent. Cuvo (1975) found that this difference can be attributed to strategy: younger children tend to just repeat the last item presented, while older children adhere to the adult pattern of rehearsal. These studies demonstrate that implicit learning, as witnessed in the recency effect, is not affected by age, while explicit learning is, as witnessed in the primacy effect.

Simulation 1

The goal of the first simulation was to reproduce the results of Rundus' experiment. Rundus used 25 participants, to whom 11 lists of 20 words were presented on cards with a 5 second interval. Participants were instructed to rehearse aloud.

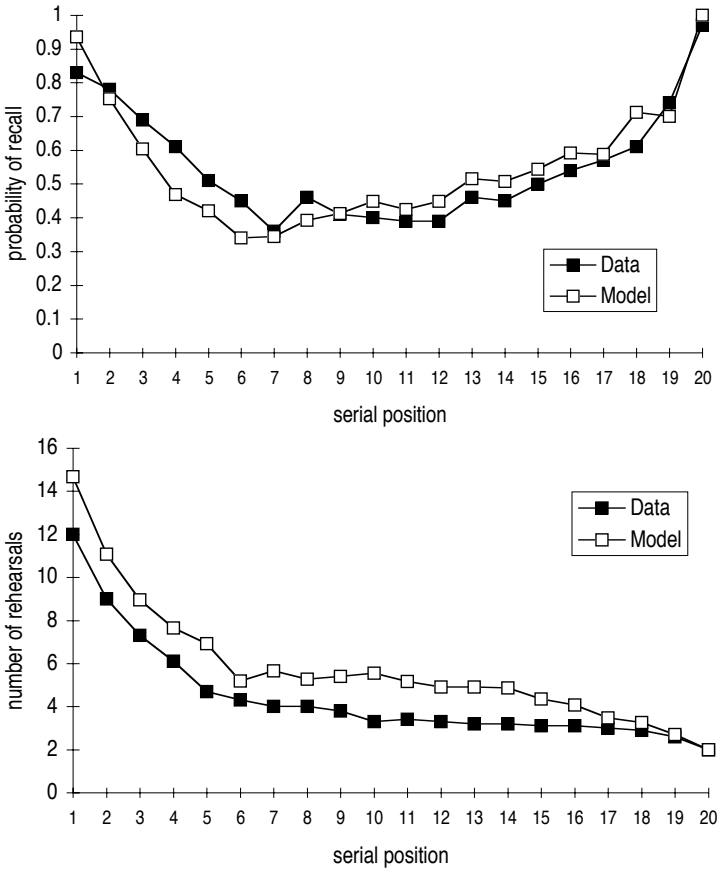


Figure 4.7. Results of the simulation compared to Rundus' data.

In the experiment the mean number of words correctly recalled was 11.12 and the mean number of rehearsals 88.3. The simulation recalls 11.15 words correctly on average, using 116.0 rehearsals. The serial position curve and the mean number of rehearsals for each item in the list are shown in figure 4.7. The fit between the data and the model is reasonably good for the probabilities of recall (R^2=0.82), and not too good for the number of rehearsals (R^2=0.57). As can been seen in the figure, the model overestimates the number of rehearsals, although the curve has the right shape.

Simulation 2

In the standard experiment, participants have to rehearse aloud, but are free in choosing which words to rehearse. Participants can be constrained in this aspect, for example if they may only rehearse the word that has just been presented. Figure 4.8

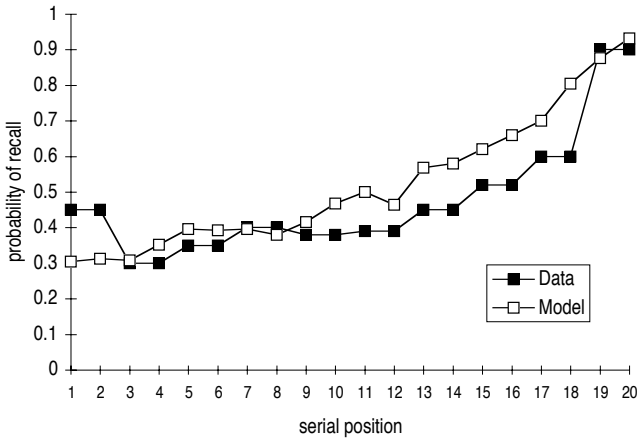


Figure 4.8. Data and model of the restricted version of free recall

shows the data (from Fischler, Rundus & Atkinson, 1970) and the results of the model (R^2=0.65). The interesting aspect is that the primacy effect largely disappears, but the recency effect remains. This finding is consistent with Hagen and Kail (1973) (no primacy effect in young children) and Cuvo (1975) (young children only rehearse the last word presented) studies.

Simulation 3

To see whether the model holds its ground in other variants of the task, a data set collected by Murdock (1962) is a good basis for comparison, since he used different list lengths (from 10 to 40 words) and different rates of presentation (1 or 2 seconds per word). Murdock did not require overt rehearsal, so only the probabilities of recall can be compared. Figure 4.9 shows the data and the results of the model. The main deviation between model and data is that the model overestimates the primacy effect. The overall explained variance is nevertheless quite high (R^2=0.91).

Simulation 4

In the standard free-recall experiment, recall starts immediately after the presentation of the words. If there is a delay between recall and presentation in which further rehearsal is prevented, the recency effect disappears. An experiment by Postman and Phillips (1965) demonstrates this effect: 18 participants were given lists of 20 words, 6 lists for which recall immediately followed the presentation, and 6 lists where participants had to count backwards for 15 seconds before recall. Words were presented at a rate of one word per second, and rehearsal was covert. The mean number of words recalled correctly was 6.20 if there was no delay after presentation, and 5.05 if there was a 15 second distraction. The serial position curves for both conditions are depicted in figure 4.10, together with the simulation data. The simulation recalls 8.6 words correct on average in the condition without

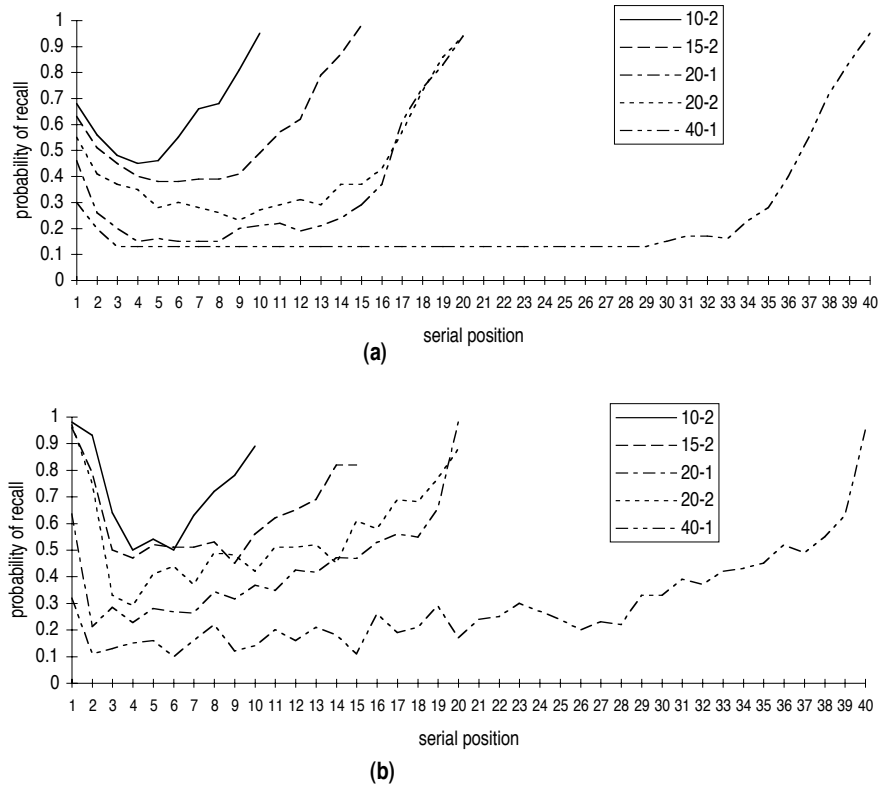


Figure 4.9. Data (a) and model results (b) for different versions of free recall. The first number is the list length and the second number the presentation rate.

delay, and 4.6 words in the 15 sec delay condition. The most interesting aspect, however, is that the recency effect has largely disappeared. This is normally explained by the fact that participants cannot use the contents of their rehearsal buffer in their answers, but the model shows that an explanation based on decay of activation is sufficient. It also predicts that due to the fact that the last few items are rehearsed fewer times than items in the middle of the experiment, the recency effect will eventually turn into a negative recency effect, as we will see in simulation 5. The primacy effect is much less affected by the delay, since it is caused by the fact that items have been rehearsed more often. The explained variance is only average: the overall R^2 has a value of 0.58.

Simulation 5

Craik (1970) discovered that the disappearance of the recency effect after a delay can even turn into a negative recency: in some situations recall for items at the end of the list is worse than for items in the middle part. In a free-recall experiment 20 participants were presented with 40 lists of 15 words at a rate of 2 seconds per

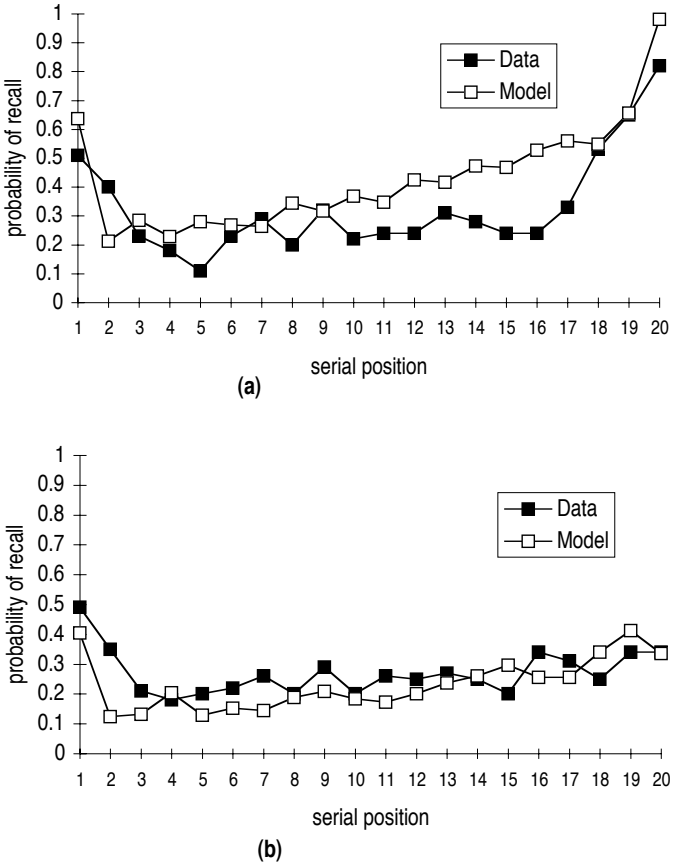


Figure 4.10. Data and model results of free recall without pause (a) and with a 15 second pause (b) after presentation.

word. After each 10 lists, participants were asked to recall as many words as possible from the previous 10 lists, giving a final-recall score. The results of this experiment are shown in figure 4.11a. To obtain a smooth curve Craik averaged each data-point with its successor and predecessor, except for the first and the last.

The free-recall model also produces negative recency, as can be seen in figure 4.11b. The same averaging technique as Craik used is used on the data. In the simulation the model has to produce as many items as possible after presentation, after which a 60 second break follows and another, final, recall session. Although the results of the model cannot directly be compared to Craik's data, since the experimental setup is different, a negative recency effect that is similar to Craik can be seen in the model.

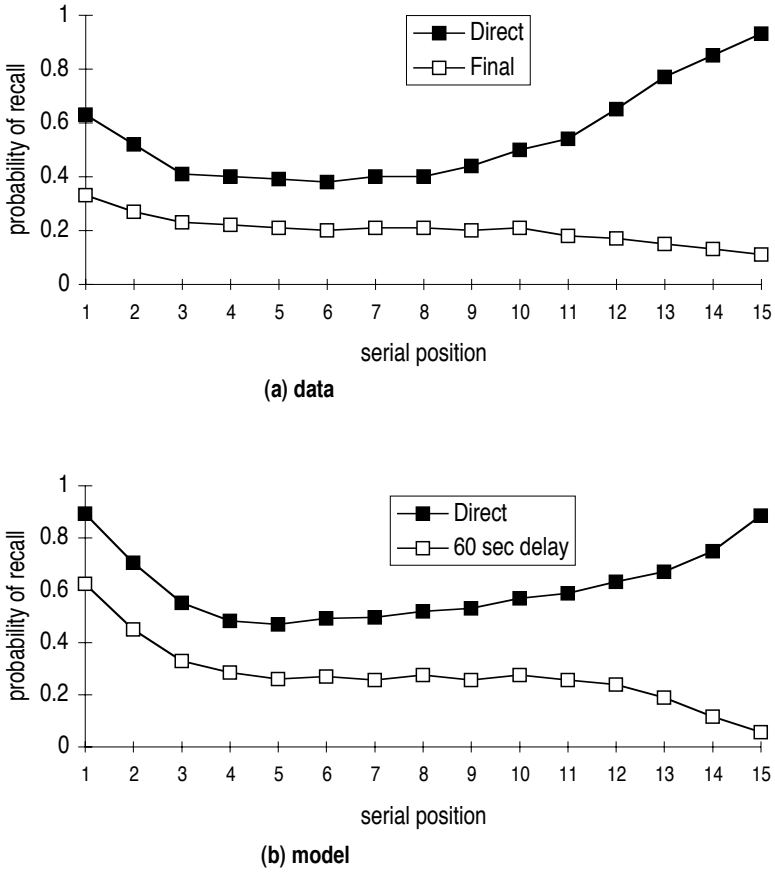


Figure 4.11. Negative recency in data by Craik (1970) (a), and in the model (b).

Discussion

The results of the simulations show that the classical effects of primacy and recency in free recall can be reproduced using a theory of rehearsal based on the ACT-R architecture and Baddeley's phonological loop. The primacy effect can be explained by the fact that items early in the list are rehearsed more often on average than other items in the list, the same explanation that was used in the dual-store theory of memory. The recency effect can be explained by the base-level activation mechanism of ACT-R: the last few items of the list have a higher activation because they have been accessed more recently.

Simulations 2, 4 and 5 show that both the primacy and the recency effect can be manipulated by changing aspects of the task. It is interesting to examine the nature

of these manipulations. In simulation 2, participants were instructed to use a certain type of rehearsal strategy, which resulted in the disappearance of the primacy effect. The learning strategy thus determines the presence or absence of the primacy effect, and can be considered as an effect of explicit learning. In simulations 4 and 5, the circumstances of the experiment were changed. Instead of changing the strategy, a time delay was used, resulting in an effect on the recency effect.

The various models presented in this section also illustrate the inadequacy of the R^2 measure to express the quality of fit between the data and the model. Although the fit with the original Rundus data is clearly the best, the model of the Murdock experiment achieves the best fit, although it overestimates the primacy effect.

The parameters in the models discussed above were set to their recommended default settings, except for the activation noise and activation threshold, which were estimated to optimize the fit to the Rundus model. The same settings were used for all the other simulations. The base-level learning decay parameter used was the recommended value of 0.5. In the Tulving model this parameter had to be set to 0.3, meaning there is an issue to be resolved here. We will return to this issue in chapter 6.

CHAPTER 5

Strategies of learning

5.1 Introduction

In the previous chapter we saw that learning is a concept with two layers. The bottom layer consists of the learning mechanisms of the architecture, while the upper layer is a set of learning strategies that manipulate the mechanisms of the bottom layer. We have already seen an example of a learning strategy in the form of rehearsal. In this chapter, the focus will be on learning strategies that try to infer new knowledge, a phenomenon that we have witnessed in the protocols of the participants in the scheduling problem. There are several questions to be answered with respect to learning strategies.

The first question is: when are learning strategies used. A learning strategy is tied to an explicit learning goal. This means that at some point during reasoning, a learning goal must be posed in favor of other processing. The protocols in chapter 3 demonstrate that several episodes can be distinguished in the problem solving process, some of which involve search, and some of which involve reflection. In the reflection episodes, participants discover new strategies, and the recurrence of these strategies in later episodes indicates that they have been learned during the first episode. But when, and for what reasons, does a participant decide to stop search and start reflection? This is a question of meta-cognition, often portrayed as a monitoring process that prevents unbounded search. An alternative, which I will pursue in sections 5.2 and 5.3, is to incorporate the function of meta-cognition without the need for a separate monitoring process. A separate process would require its own monitor, leading to endless regress.

A second question one might ask is how learning strategies themselves are learned, and what their nature is. Learning learning strategies is probably a long-term process, so it will be hard to investigate this process in a standard experimental setting. A better setting to investigate the nature of learning strategies is development. During development, a lot of learning strategies are acquired. Probably many differences between adults and children with respect to their reasoning capabilities can be explained in terms of what type of information they can represent, and what learning strategies they have available to learn this information. In section 5.4, three theories of development will be discussed, and what can be learned from them.

The third and final question is how to model strategy learning in ACT-R. New production rules have to be represented in memory. Some learning scheme has to be developed that is independent of the current task. In sections 5.5 and 5.6, I will propose some example learning strategies, and show how they can learn task-specific knowledge in two different domains. To emulate some of the developmental aspects of these strategies, I will do some "reverse development" by impoverishing the learning strategies. As we will see, this leads to behavior associated with an earlier stage of development.

5.2 Search vs. Insight

In chapter 1, I criticized the traditional approach of problem solving, in which solving a problem means no more and no less than finding an appropriate sequence of operators that transforms a certain initial state into a state that satisfies some goal criterion. The difficulty of problem solving is determined by factors as the length of the sequence needed, the number of possible operators, and the amount of knowledge available on how to choose the right operator.

The alternative insight theory stresses the moment at which the crucial step towards the solution is found. Insight can be viewed in two ways: as a special process, or as a result of ordinary perception, recognition and learning processes (Davidson, 1995). Despite the intuitive appeal of a special process, the latter view is more consistent with the modern information-processing paradigm of cognitive psychology, and is much more open to both empirical study and computational modeling. One way to look at insights from an information-processing viewpoint is that an insight involves the relaxation of constraints (see, for example, Knoblich & Ohlson, 1996). In the nine-dots problem mentioned in chapter 1, for example, the initial assumption that all lines should remain within the 3x3 square is a constraint that needs to be relaxed.

Another famous insight problem is the box-candle problem, in which a candle has to be affixed to a door, using a box of candles, a box of matches, and a box of tacks (see, for example, Mayer, 1983). The crucial constraint to be relaxed is the fact that the boxes are not just containers, but can also be used to support the candle. Knoblich & Ohlsson (1996) have shown in an experiment involving matchstick problems that once a constraint is relaxed, it stays relaxed.

Looking at insights as removing constraints is a rather negative approach: something that is there needs to be removed. A slightly different view on insight is to assume some new knowledge is gained at the moment of insight. This corresponds well with the idea that a constraint stays relaxed. Another advantage of this view on insight is that not all insights can be described as relaxing constraints. The fact that participants in the scheduling problem start using complex inferences during a reflection episode can of course be called "the relaxation of the constraint not to use complex inferences", but this stretches the original idea so much it becomes almost meaningless: it is like defining the creation of a statue as removing marble.

Both the search and the insight theory select the problems to be studied in accordance with their own view. Typical "search"-problems involve finding long strings of clearly defined operators, as in the eight puzzle, the towers-of-hanoi task and other puzzles, often adapted from artificial intelligence toy domains. "Insight"-problems, on the other hand, can be solved in only a few steps, often only one. Possible operations are often defined unclearly, or misleadingly, or are not defined

at all, as the nine-dots and candle problems illustrate. Due to this choice of problems, both evidence from insight and search experiments tend to support their respective theories. Both theories ignore some aspects of problem solving. The search theory seems to assume that participants create clear-cut operators based on instructions alone, and fails to assign a significant role to reflection. Insight theory on the other hand offers no explanation of the role of processing that happens before the "insight" occurs. An obvious alternative is to think of both search and insight as aspects of problem solving, and to try to find a theory of problem solving that combines the two (Ohlsson, 1984).

One such view sees insight as representational change, which is a more general term that includes constraint relaxation and gaining new knowledge about the task. Search is needed to explore the current representation of the problem, and insight is needed if the current representation appears not to be sufficient to solve the problem. In this view, search and insight correspond to what Norman (1993) calls *experiential* and *reflective* cognition. If someone is in experiential mode, behavior is largely determined by the task at hand and the task-specific knowledge the person already has. In reflective mode on the other hand, comparisons between problems are made, possibly relevant knowledge is retrieved from memory, and new hypotheses are created. If reflection is successful, new task-specific knowledge is gained, which may be more general and on a higher level than the existing knowledge. All these theories, however, fail to specify at what time a certain mode of thinking will be used, and due to what influences the mode of thinking changes.

In the protocol analysis of the scheduling problem in chapter 3, we saw that all participants start with an experiential search strategy, and only later on switch to a reflective strategy. As we have observed, the process reflects the explore-impasse-insight-execute pattern described in the literature about insight (Ohlsson, 1984; Davidson, 1995). Some, but not all, of the participants show some sort of impasse, during which they stop searching, just stare at the screen for a minute, and then try a new approach. Furthermore, there is no difference between the explore and the execute stage: the participant just searches on, using the knowledge gained by reflection. Sometimes further reflection is needed to reach a solution.

5.3 A dynamic growth model

In this section a model is proposed that explores the distinction between search and reflection. The model is based on Anderson's theory of rational analysis, the theoretical basis of ACT-R (Anderson, 1990). According to rational analysis, participants choose strategies based on a cost-benefit analysis: the strategy that has the lowest expected cost and the highest probability of success is selected in favor of others. The model is not an actual ACT-R model, but a *dynamic growth model*, in

which the trade-off between search and reflection is modeled in a coarse-grained way. Dynamic models are used in developmental psychology to describe developmental paths, for instance a model that describes stage-wise increases in knowledge (Van Geert, 1994; 1998). In section 5.6, the coarse-grained model will be applied in actual ACT-R models.

In order to give a rational account of insight learning, the first question is: why would participants initially prefer a search strategy in the scheduling problem? The reflective strategy seems to be much more powerful. There are several reasons for this. A first reason is that reflective reasoning has a high cost. To be successful, several aspects of the task must be combined and kept in memory. Additional knowledge must be retrieved from memory and it may be necessary to seek analogies with other problems. A second reason is that it is not immediately evident that search will be unsuccessful. In the nine-dots problem, but also in the scheduling problem, naive search alone does not work, but people generally do not know this when they start on these problems. Why not try the strategy which takes the least effort first? A third reason is that as a participant starts with a new type of problem, he has only read instructions and has seen an example problem. He first has to learn the basic rules and operators by experience, before he can attempt any higher level strategies.

Considerations like these are the basic ingredients for the model. In the model, search and reflection are two competing strategies, whose evaluations depend on expected gain. Estimates on these gains change in time, due to increasing knowledge and the successes and failures due to this knowledge.

The model

According to rational analysis (Anderson, 1990), strategies are chosen with respect to their expected outcome, according to the following equation:

$$\text{Expected outcome of strategy} s = P_s G - C_s \quad \textbf{(5.1)}$$

In this equation, P_s is the estimated probability of reaching the goal using strategy s, G is the expected value of the goal, and C_s is the estimated cost of reaching the goal using strategy s.

The model will attempt to describe how search and reflection will alternate while solving a problem. The model is coarse-grained in the sense that the knowledge of the system with respect to a certain task is summarized in two variables L_1 and L_2. L_1 is a measure for the amount of basic task-knowledge, for example, in the case of the scheduling task, knowledge about adding a task to an existing plan and knowledge to judge whether a solution is correct. L_2 corresponds to the amount of higher-level knowledge in the system, for example the fact that it is a good idea to see how the tasks add up to the amount of time the workers have available. If a

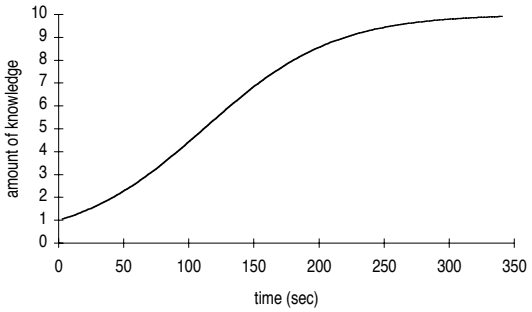


Figure 5.1. Basic growth function

participant starts with a new problem, we assume that both variables have a small value. Later on, they increase, since the participant builds up knowledge during problem solving. The assumption of the model will be that search will increase the amount of basic knowledge, represented by L_1, and reflection will increase the amount of higher-level knowledge, represented by L_2. The choice of two knowledge levels is somewhat arbitrary, as are some of the choices of parameters in the equations below. The reader should keep in mind that the goal is to produce a rational account of the alternation between search and reflection.

The following equations show how L_1 and L_2 grow in time, and are inspired by the growth equation used by Van Geert (1994):

If the strategy in step *i*-1 is search, then

$$L_1(i) = L_1(i-1) + R_1 L_1(i-1)\left(1 - \frac{L_1(i-1)}{L_{1max}}\right) \quad \textbf{(5.2)}$$

else L_1 keeps its value, so $L_1(i) = L_1(i-1)$. R_1 is a constant that controls the rate of growth, and L_{1max} is the maximum possible value for L_1. The fraction at the end of the equation ensures that L_1 doesn't exceed its maximum value. Assuming only search is used, the value of L_1 grows gradually and levels off once it approaches the maximum. Figure 5.1 shows an example of the growth of L_1 knowledge if only search is used, and L_{1max} equals 10.

The equation for L_2 is slightly more complicated, because the increase in value depends on the current value of L_1, reflecting the fact that we can only gain higher-level knowledge if we have enough basic knowledge.

If the strategy at step *i*-1 is reflection, then

$$L_2(i) = L_2(i-1) + S_{12} \cdot L_1(i-1)\left(1 - \frac{L_2(i-1)}{L_{2max}}\right) \quad \textbf{(5.3)}$$

else $L_2(i) = L_2(i-1)$. L_{2max} is the maximum possible value for L_2. The parameter S_{12} (support) controls the influence of basic knowledge on the increase of higher level knowledge.

Now that we have described how knowledge grows depending on the type of strategy, we have to describe the process by which a strategy is chosen. At this point, Anderson's expected gain equations are introduced into the model. Whether the strategy at step *i* will be search or reflection is determined by their respective expected outcomes:

$$\text{Expected outcome of search} = P_{search}(i) \cdot G - C_{search} \quad \textbf{(5.4)}$$

$$\text{Expected outcome of reflection} = P_{ref} \cdot G - C_{ref}(i) \quad \textbf{(5.5)}$$

The strategy with the highest expected outcome will be chosen. In these equations G, C_{search} and P_{ref} are fixed parameters. G, the expected value of the goal, is assumed to be fixed as long as the goal is not reached. C_{search}, the cost of search, may change in actual problem-solving situations, for example due to the fact that search becomes more complicated once more knowledge is involved. But since these fluctuations are task-dependent, the current model assumes that the costs of search remain constant. The influence of P_{ref}, the chance of success of reflection, will be taken into account in the specification of the costs of reflection. $P_{search}(i)$ and $C_{ref}(i)$ are variable in time, and rise and fall due to the chosen strategy and the growth in knowledge.

The probability that search will reach the goal depends on the amount of knowledge and the current evaluation of this knowledge:

$$P_{search}(i) = \frac{L_1(i)P_1(i) + wL_2(i)P_2(i)}{L_1(i) + wL_2(i)} \quad \textbf{(5.6)}$$

The constant *w* determines how much more useful higher-order knowledge is than basic knowledge. $P_1(i)$ is the contribution to the probability of success of L_1 knowledge, and $P_2(i)$ the contribution of L_2 knowledge. The probability of success increases as knowledge increases, but decreases over time if the goal is not reached. The decrease in knowledge is calculated by multiplying the probability of success by a decay parameter on each time-step search is used as strategy. New knowledge is given the benefit of the doubt, and is assigned an initial probability of success of 1. Both $P_1(i)$ and $P_2(i)$ can be calculated using:

$$P_j(i) = \frac{p_{decay}P_j(i-1) \cdot L_j(i-1) + (L_j(i) - L_j(i-1))}{L_j(i)}; (j = 1, 2) \quad \textbf{(5.7)}$$

p_{decay} represents the decay in probability of success, and has typical values between 0.95 and 0.99 if the strategy in step *i* was search and the goal has not been reached. In the case of reflection in step *i*, $p_{decay} = 1$. The $p_{decay}P_j(i-1)$ part of the equation takes care of the decay of existing knowledge. However, new knowledge is added to the model as well, and this new knowledge starts out with the "optimistic" probability of success of 1. The $(L_j(i) - L_j(i-1))$ part of the equation takes care of that aspect. So on each search step, the probability of success decreases due to decay, and increases due to the addition of "fresh" knowledge.

The costs of reflection depend on two factors. The first is that the costs are higher if there is less basic knowledge, since higher level knowledge has to be based on more primitive knowledge. The second factor is that the costs are higher if there is already a lot of higher level knowledge. This reflects the idea that there is only a limited number of good ideas to come up with, and that it will be more difficult to discover a new idea if there is less to discover.

$$C_{ref}(i) = C_{base} + \left(c_1\frac{L_{1max}}{L_1(i)}\right) + \left(c_2\frac{L_2(i)}{L_{2max}}\right) \quad \textbf{(5.8)}$$

This equation assumes reflection has a certain base cost (C_{base}) that is increased by two factors: $c_1\frac{L_{1max}}{L_1(i)}$ which decreases as level 1 knowledge increases, and $c_2\frac{L_2(i)}{L_{2max}}$, which increases as level 2 knowledge increases.

Finally we have to say something about time, since we have talked about "steps" in the previous discussion. Each step takes an amount of time which can vary. So, following the ACT-R intuition that cost and time are related to each other, we take the estimated cost of the strategy at step *i* as the amount of time step *i* takes:

$$T(i) = T(i-1) + C(i) \quad \textbf{(5.9)}$$

where $C(i)$ is either C_{search} or $C_{ref}(i)$, depending on the strategy at step *i*.

Results

If the appropriate constants and starting values are chosen for the variables described above, we can calculate the increase in knowledge over time. The model is simulated using a spreadsheet program, in this case Microsoft Excel. Note that the model assumes that the goal is never reached, so the results simulate a participant that never succeeds in reaching the goal. Figure 5.2 shows the value of

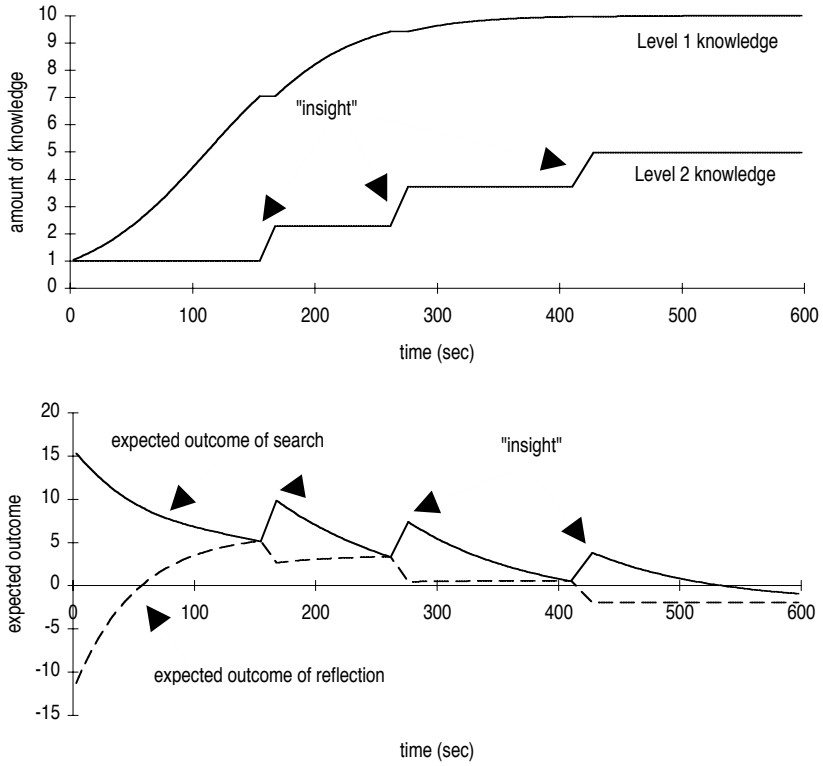


Figure 5.2. Value of level 1 and level 2 knowledge (top) and the expected gains for search and reflection (bottom) for G=20

L_1 and L_2 with respect to T, and the corresponding evaluations for search and reflection. At the start of the task, search is superior to reflection, but as search fails to find the goal, and the basic (level 1) knowledge increases, reflection becomes more and more attractive up to the point (at T=155) where reflection wins from search. Since reflection leads to an increase of level 2 knowledge, search again becomes more attractive (using the newly gained knowledge), and since the cost of reflection increases with the amount of level 2 knowledge already present, reflection becomes less attractive. As a result search will again dominate for a while, up to T=262 where reflection wins again. We assume problem solving continues until both expected outcomes drop below zero, since then neither strategy has a positive expected outcome. In the example, this is the case at T=533.

As noted, G is the value of the goal. Using a lower value for G corresponds to the fact that a participant values the goal less, and is less motivated to reach it. If we calculate the model for G=15 instead of G=20, we get the results as depicted in figure 5.3. The

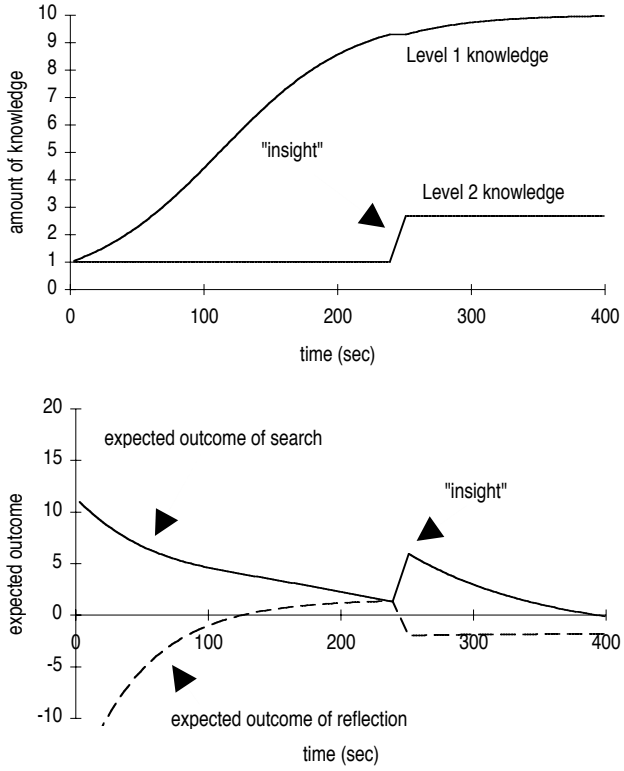


Figure 5.3. Value of level 1 and level 2 knowledge (top) and the expected gains for search and reflection (bottom) for G=15

result is that reflection occurs only once, and later (at T=239). Furthermore, at T=393 both evaluations drop below zero, so a less motivated individual gives up earlier. If G is further decreased to 12, no reflection at all takes place, and the give-up point is at T=277.

5.4 The nature of learning strategies

The dynamic growth model nicely describes the phenomena around insight in the literature and in the scheduling experiment. Furthermore, it explains why this behavior is rational. It also predicts changes in strategy due to motivational factors. It however poses new questions. What is the nature of the basic and higher-level knowledge? How will the model behave if the goal is reached at some point? What mechanism is responsible for gaining new knowledge, and how is it represented?

In the previous chapter, I proposed to define implicit learning in terms of learning by the mechanisms of the architecture, and to define explicit learning by activity of explicit learning strategies. In this sense, learning that occurs during search is implicit, since during search the goal is to solve the problem, not to learn something new. During reflection, on the other hand, the goal is to find a new way to approach the problem, so the goal is to discover something new. In this sense, reflection can be seen as explicit learning. As I have argued, there is no principal distinction between the knowledge learned by implicit learning and the knowledge learned by explicit learning, hence there is no real distinction between level 1 and level 2 knowledge in the dynamic growth model. It is just that level 2 knowledge might be more useful, because it has been constructed in a more clever way.

How to get more insight into learning strategies? As we have seen, they are a source of individual differences. On the other hand, there are explicit strategies that at least all adults share, as we have seen in the case of rehearsal. But even in the area of rehearsal, some people prefer to memorize items by verbal rehearsal, while others prefer memorizing information by visualizing it in some fashion. Since learning strategies that are unique for certain individuals are hard to investigate, I will focus on strategies that most adults share, and see how they develop in children.

Piaget's stage theory

The first to acknowledge the fact that children reason in a different way than adults do was Jean Piaget (1952). Based on many experiments, among which the famous conservation experiments, Piaget concluded that children from different ages solve problems in different ways. He proposed a theory of stages, in which children in higher stages can reason more abstractly than children in lower stages. An example is the fact that very young children, who are in the first sensorimotor stage, only reason about objects that are in their field of perception. Once an object is hidden it is considered non-existent. In the second, pre-operational stage, children have mastered the concept of object permanence, and know an object is still there, although it cannot be seen at the moment. The transition between stages is a discontinuous jump: a child either has or hasn't mastered the concept of object permanence. Piaget's four stages are very strict: if a child moves to a new stage, they do so for all skills in all domains at once. It turned out that Piaget's theory was too strong. Children can be taught skills that belong to a stage they have not reached yet, and children may be in different stages in different cognitive domains. Piaget was well aware of this problem, to which he referred to as "horizontal décolage".

The mechanism that causes these discontinuous jumps is *adaptation*, which, according to Piaget, is a result of *assimilation* and *accommodation*. During assimilation elements from the external world are added to the knowledge of the child. Accommodation, on the other hand, is an internal process that modifies the assimilatory scheme on the basis of the assimilated experiences. So accommodation

Level	Representation	Examples	Age
S4/Rp1: Single Representations	$[YOU_{MEAN}]$ or $[ME_{NICE}]$	Child pretends that doll is hitting someone.	18-24 months
		Child says, "Doll mean."	
Rp2: Representational Mappings	$[YOU_{MEAN}—ME_{MEAN}]$	Child makes one doll's mean actions produce reciprocal mean actions in the other doll.	3.5-4.5 years
		Child makes two dolls act as Mom and Dad in parental roles.	
Rp3: Representational Systems	$\left[YOU \begin{matrix} NICE \\ MEAN \end{matrix} \leftrightarrow ME \begin{matrix} NICE \\ MEAN \end{matrix}\right]$	Child makes two dolls interact in reciprocally nice and mean ways.	6-7 years
		Child makes two dolls act as Mom and Dad as well as doctor and teacher simultaneously.	
Rp4/A1: Single Abstractions	$\left[\begin{matrix} YOU1 \begin{matrix} NICE \\ MEAN \end{matrix} \leftrightarrow ME \begin{matrix} NICE \\ MEAN \end{matrix} \\ \updownarrow \\ YOU2 \begin{matrix} NICE \\ MEAN \end{matrix} \leftrightarrow ME \begin{matrix} NICE \\ MEAN \end{matrix} \end{matrix}\right]$	Person explains that intentions matter more than actions.	10-12 years
		Person sees Dad as having general personality characteristics, such as conformity, emotionality, or secretiveness	
	$= [\mathbb{INTE}_{POS}]$		

Figure 5.4. Example of stage 7-10 in Fischer's theory. Adapted from Fischer & Ayoub (1994)

can be seen as the process that produces "new" knowledge, and causes the sudden jumps. In order to do so, it needs the accumulated knowledge gained by the assimilation process.

Fischer's levels

A modern version of Piaget's theory by Kurt Fischer (1980) tries to remedy the flaws in the original theory. His theory has no less than thirteen stages or levels as he calls them, grouped into four tiers. He distinguishes between two levels of performance: the *functional level* and the *optimal level*. The functional level is the level a child performs at in a "normal" situation. There may be large variations in this level across domains. At the functional level, a child is no longer in a single stage, but has a different level of development for each cognitive domain. The optimal level, on the other hand, is the highest level that an individual can produce, and is attained

when given high levels of support and opportunities for practice. The fact that levels of development can differ across domains makes Fischer's theory more realistic, but weaker than Piaget's. A strong point of the theory is however that Fischer describes the kind of representations that are used at each level, and how they can be combined to reach a higher level. In that sense, the theory is much more precise than the original Piaget theory.

From the viewpoint of learning strategies, the optimal level can be associated with the learning strategies that are available to a child. A skill that is beyond the child's optimal level is a skill for which it lacks the right learning skills. That does not imply that the child has already learned everything it could possibly learn given its current learning skills. For each domain, the child has acquired some of the domain-specific knowledge it can possibly gain given its current learning skills. This level can be associated with the functional level. To get from the current functional level for a skill to the optimal level, the child just has to learn additional domain-specific knowledge using its current learning skills. To go beyond the optimal level, new learning skills have to be acquired first.

Figure 5.4 is an illustration of some of the levels, in this case the third tier applied to the topic of what type of behavior agents can carry out. At the level of single representations, the top level in the table, children can represent that people or animate objects can carry out concrete actions and have concrete characteristics. They cannot yet combine these representations. At the next level, simple combinations of agent-behavior tuples can be made, for example: if you are mean, I will be mean. These combinations remain isolated, however, so there is no generalization of relationships between agent-behavior tuples. At the level of representational systems, combinations of representations are no longer isolated, but generalized. Instead of having a collection of combinations of representations, the actual mapping between representations is understood. At the final level of this example, the level of single abstractions, mappings between representations are combined, leading to concepts like intentions: the intention of a person influences the actual behavior they show while interacting. The complex pattern of interactions between mappings between representations are collapsed into new units: abstractions. In the next tier, abstractions are combined in the same manner as representations in this tier: first by simple combinations, later by systems, and finally by systems of systems.

An important property of Fischer's theory is that the representations used at a certain level are combined in the next level, either by forming combinations, as in the shift from single units to mapping, or by generalization, by combining a set of mappings into a system. So, the end-products of a level are the building blocks for the next level. A simple experiment that shows that young children cannot combine representations in the same way older children can is the discrimination-shift task by Kendler & Kendler (1959). In this experiment, children are presented with blocks

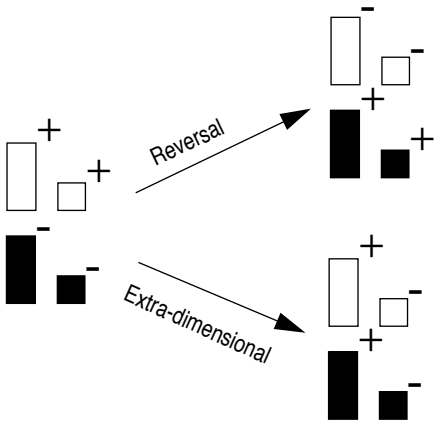


Figure 5.5. Example of a discrimination-shift task. Stimuli indicated by the "+"-sign are stimuli on which the participant has to answer with "yes".

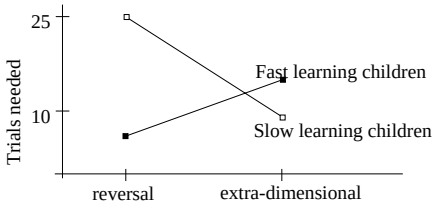


Figure 5.6. Results of the discrimination-shift experiment

that are either white or black, and either small or large. The children have to say either "yes" or "no" to each block. For example, they have to say "yes" when a white block is shown, or "no" when a black block is shown. The children do not know this, but have to discover this on the basis of feedback. After a child has made 10 consecutive correct predictions, the criterion is changed, unbeknownst to the child. Either a reversal shift is made, in which "yes" has to answered in response to black blocks, or an extra-dimensional shift is made, in which the dimension is changed, and the child has to answer "yes" when a large block is presented (figure 5.5). After the shift, the number of trials the child needs in order to be able to do ten consecutive correct trials again is counted. Figure 5.6 shows the results of a discrimination-shift experiment in which participants were children of 6-7 years old (Kendler & Kendler, 1959). Fast-learning children discover reversal shifts quickly, but need a lot more trials to discover an extra-dimensional shift. Slow-learning children show a pattern that is entirely opposite: they are faster at an extra-dimensional shift, while needing much more time for a reversal shift. Similar experiments have shown that adults are also faster at reversal shifts (for example, Harrow & Friedman, 1958), while small

children and animals (for example rats in Kelleher, 1956) are faster at extra-dimensional shifts.

In terms of Fischer, the knowledge needed to successfully do this particular discrimination-shift task can now be stated. The most compact representation is an Rp3-system (the third row in figure 5.4), in which the color (or size) of the block has to be mapped onto the response (yes or no). Before a shift takes place, the following system has to be learned:

$$\left[COLOR \begin{matrix} WHITE \\ BLACK \end{matrix} \leftrightarrow RESPONSE \begin{matrix} YES \\ NO \end{matrix} \right] \tag{5.10}$$

A property of the block, its color, has to be used to select a response. If a child has not mastered Rp3-representations yet, it has to use representations of one of the lower stages of development, for instance the S2 stage of sensorimotor mappings. This stage is not shown in figure 5.4, but is two levels below the S4/Rp1-level. At this stage, it is not yet possible to reason about individual properties of an object, but just about the object as a whole. The knowledge needed before the shift has to be represented by a set of four sensorimotor mappings:

$$\begin{array}{l} \{\, [SMALL\text{-}WHITE\text{-}BLOCK — REPONSE\text{-}YES], \\ [LARGE\text{-}WHITE\text{-}BLOCK — RESPONSE\text{-}YES], \\ [SMALL\text{-}BLACK\text{-}BLOCK — RESPONSE\text{-}NO], \\ [LARGE\text{-}BLACK\text{-}BLOCK — RESPONSE\text{-}NO] \,\} \end{array} \tag{5.11}$$

If we now look at the changes required in each of these representations to accommodate the different types of shift, we can understand why reversal shifts are easier if you use Rp3 representations, and extra-dimensional shifts are easier if you use just S2 representations. In the Rp3 case (figure 5.7a), the reversal shift is easier, because the system remains the same: only the mapping within the system changes. In the S2 case (figure 5.7b), the extra-dimensional case is easier, since only two out of four mappings change, while two mappings remain the same. In the reversal shift all four mappings change.

In the introduction to this section I remarked that reflection corresponds to the use of explicit learning strategies. Since learning strategies themselves have to be acquired as well, it interesting to look at the development of reflection and the relation with Fischer's theory. Kitchener, Lynch, Fischer and Wood (1993) have done a study in which they relate Fischer's skill levels to reflective judgement. Each level from Rp1 upwards can be related to an increased capacity of reflection. While children at the Rp1 level can only reason about concrete propositions, like "I know the cereal is in the box", children at the Rp3 level can reason about the uncertainty of knowledge. Kitchener et al. developed the Reflective Judgement Interview to assess the level of reflection, and used participants who were between 14 and 28 years old. The results show a steady increase in reflective capacity. Moreover, a specific version of the test was used to assess the optimal level of performance by giving maximal

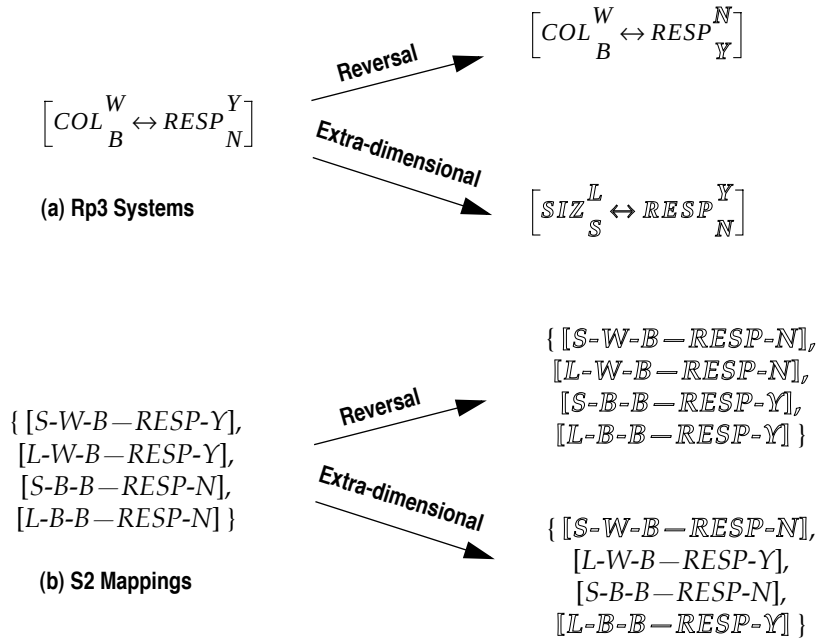


Figure 5.7. Changes in representation (indicated in an outlined font) due to reversal and extra-dimensional shifts using different types of representation. Abbreviated versions of (5.10) and (5.11) are used.

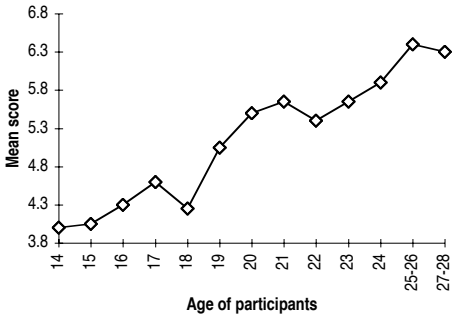


Figure 5.8. Increase in reflective judgement with age. From Kitchener et al. (1993)

contextual support. In this version of the test the growth curve shows some evidence for growth spurts, as predicted by Fischer's theory (figure 5.8).

In summary, Fischer's theory is weaker than Piaget's with respect to the predictions it makes. This is not a big problem, since Piaget's original theory is not completely accurate. On the other hand, Fischer provides representations that can be used to analyze skills in different stages of development. These representations can also be

used to describe developmental paths that lead from one stage to the next stage. In this sense Fischer's theory is stronger than Piaget's theory: it can specify how knowledge is represented, and how a higher-order representation can be built out of lower-order representations. It still lacks a real processing component, however, a specification of the processes that actually change the representations. Furthermore, Fischer's representations in their current form are not precise enough to support a detailed processing theory. This is also the main criticism of stage theories of development, the fact that they put too much stress on the state of knowledge at a certain age, thereby neglecting the importance of what some researchers see as the main issue of interest in development: the process of change.

The dynamics of change in Fischer's theory can be described by dynamic systems theory. Van Geert (1994) has developed models of the increase in knowledge on different levels, using growth equations similar to those presented in section 5.3. An interesting feature of van Geert's model is that it can model the shape of the growth spurts, such as the slight regression in performance between age 17 and 18 in figure 5.8, followed by a fast increase between age 18 and 20. As the model is coarse-grained, it does not describe the changes in representations, nor can it explain by what changes a new level starts. Nevertheless, a dynamic growth model may be a good starting point for constructing a fine-grained model that does model knowledge representations.

Karmiloff-Smith's representational redescription

A theory that puts more stress on the process of change than on levels of knowledge is Annette Karmiloff-Smith's (1992) theory of *representational redescription* (RR). The RR theory is concerned with mastering skills in specific domains, so it has no global Piaget-like stages or Fischer-like optimal levels. An interesting feature of the theory is that it discriminates an implicit learning phase for a new skill, followed by several explicit learning phases. In each new phase, the representations of the previous phases are redescribed into a new representation. The phases are called I (implicit), E1 (explicit 1), E2 (explicit 2) and E3 (explicit 3). The last two phases are often collapsed into a single E2/3 phase. The difference between a phase and a stage is that phases are not related to age, and the cycle of four phases recurs for every domain that has to be mastered during development.

According to the RR theory, the I-phase in learning a new skill involves implicit, data driven processing. In this phase, the child creates "representation adjunctions" out of the external data, which are just stored in memory. No further processing is done on these representations, but they can contribute to successful performance. If the child has accumulated enough adjunctions, performance becomes consistently successful. The RR theory defines this as *behavioral mastery*. Although the child can perform the skill, it does not have conscious access to it, since the examples are not generalized into rules. Generalization takes place in the E1 phase, in which the focus is moved from external data to internal representations. Features from the

environment are disregarded in favor of the internal generalization process. This may lead to a decrease in performance, since generalizations may be wrong. In E2/3, the internal representations are made consistent with the external data, leading to a representation that supports successful performance, and offers the building blocks for new skills.

Each phase produces its own type of representations. The "representational adjunctions" are stored in procedural form. This procedural form is not the same as production rules in ACT-R, but shows a strong resemblance to popped goals that are stored in declarative memory. In the E1 phase, the representational adjunctions are redescribed into more compact abstractions that can be related to other domains. These abstractions are recoded in E2/3 into a representation that is available for conscious manipulation, and that can be verbalized. An important feature of these representations is that they all remain available, so even if a child has reached phase E2/3, the representational adjunctions are still available. In chapter 6 we will discuss some ACT-R models in which the ideas of representational redescription will be used and made precise in terms of ACT-R representations.

Siegler's overlapping-waves theory

Siegler (1996) criticizes the stage, level and phase models by pointing out that the idea of a stage may well be an artifact of the way developmental psychologists collect their data. Typical experiments involve studying how two or more age groups of children perform a certain task, and contrasting their respective approaches. According to Siegler, however, it is a mistake to think about *the* way children think about a certain problem at a certain age. The result of these approaches are *staircase models*. For example, several strategies to do simple additions have been identified in children: small children tend to count both addends from 1, slightly older children start with the largest addend (the min strategy), and even older children retrieve the answer from memory (Ashcraft, 1987). A "staircase" interpretation of these differences is depicted in figure 5.9: first children use the sum strategy, then they switch to the min strategy, and finally to the retrieval strategy. Closer inspection of what strategies children use reveals that children do not use a single strategy to solve addition problems, but instead use several strategies. What changes with age is the frequency with which they use a certain strategy. The bottom graph of figure 5.9 illustrates this aspect using a study from Svenson and Sjoberg (1983). In this longitudinal study, the strategy use of 13 children was followed from first to third grade. As can be seen in the graph, at each point in time children use several strategies, and the frequencies of particular strategies fluctuate over time.

The main point Siegler makes is that children do not change strategies overnight. When a child discovers or learns a new strategy to do addition, it does not exclusively switch to this strategy but adds it to the set of existing strategies with

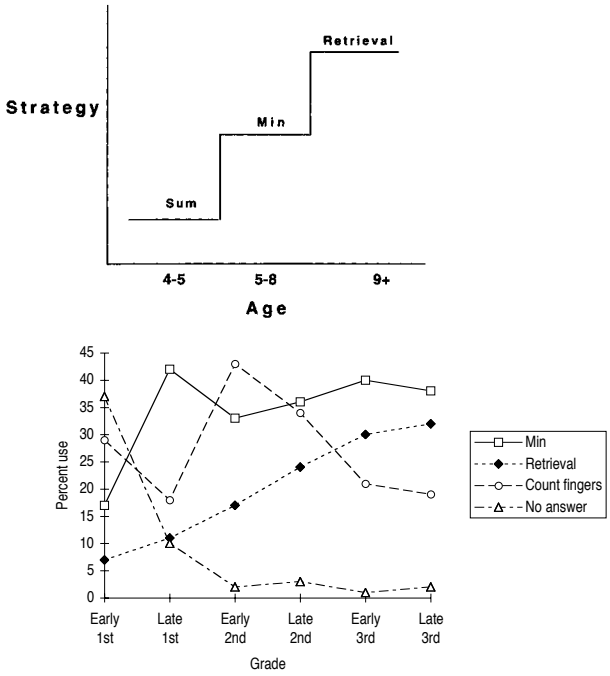


Figure 5.9. Staircase model (top) versus data supporting the overlapping-waves model (bottom) of addition strategies (from Siegler, 1996)

which it has to compete. If a strategy proves to be sound in the long run, and has an edge over other strategies, it will be used more often.

In chapter 3, we saw that some participants in the scheduling experiment sometimes use counting to do addition, which corresponds to the min strategy. This corresponds well with the overlapping waves model: even adults have all strategies available, but most adults just use retrieval as their sole strategy. Some individuals may however use other strategies occasionally. The fact that addition had to be performed in a situation where working memory load was already high may also have contributed to a shift in strategy. The matter of working memory load will return in chapter 7.

Discussion

The goal of this section was to get some idea of what learning strategies are by looking at development. Each of the four theories discussed offers some parts of the puzzle. Unfortunately, all four theories are mainly descriptive, and are not very specific about exact representations or processes acting on these representations.

An important topic in development is domain specificity. Although Piaget's theory of pure global development has turned out to be too strong, the presence of some global factor is still under debate. Fischer and Karmiloff-Smith seem to contradict each other on this point. Fischer defines a global optimal level of performance at a certain age. When this level goes up, there is a global increase in development. This global increase is not witnessed in the way Piaget envisions it, because performance in specific domains may still be lagging behind. Karmiloff-Smith's RR theory only describes development within a domain, without any need for global development.

One might ask whether it is at all possible to settle this debate on the basis of empirical evidence. In Fischer's theory, it is always possible to define an optimal level: it is just the level of the domain that has progressed most. In order to assert an optimal level that is really meaningful, it has to offer some additional support to the learning process. Although it may be very hard to find empirical evidence, a modeling perspective may offer some sort of support.

One issue a model may resolve is whether it is at all possible to have knowledge that is useful for all domains. If such knowledge can be defined and represented, for example in ACT-R's representations, the next step is to find a developmental path through this knowledge, and to specify how a more refined strategy can be learned from a more primitive one. If a system like this can be developed, and is capable of offering new explanations for old phenomena, it might offer a new type of evidence in the discussion. But in order to build such a system, the mechanisms of change have to be understood. The theories discussed here can offer some clues.

Karmiloff-Smith suggests the first (I) phase in learning a new skill is to store representational adjunctions. This phase only involves storing, retrieving and applying these adjunctions. Only when this set is sufficiently stable in the sense that behavioral mastery is reached, the explicit phases in which the information is integrated can be entered. This idea closely matches Piaget's idea of assimilation and accommodation: during assimilation external experiences are stored, while during accommodation these experiences are integrated into a qualitively new behavior.

Siegler's theory of overlapping waves shows that the discovery of a new strategy does not necessarily imply that this strategy will completely dominate behavior. A new strategy first has to prove it is better than the existing strategies. This illustrates the need for an evaluation mechanism: any new strategy has to be assessed with respect to the question whether it really is useful and better than the alternatives.

What have we learned with respect to learning strategies? Take Fischer's theory as a starting point. Each new level in the theory involves a type of representation in which a single representation replaces a combination of representations from the previous level. Assuming these representations are mainly declarative, one needs accompanying procedural knowledge in order to handle these representations.

Which of these comes first? In terms of ACT-R, the declarative representations have to be first, because a declarative example is needed to learn a new production rule. This also concurs with the RR model in which a set of representations is acquired and stored in the first phase. Only when a suitable set of knowledge is collected can generalization be attempted. Probably many generalizations are possible, so sorting them out may take some time, and may cause the rise and fall of certain strategies as Siegler has shown. Summarizing,

- learning strategies have to be general, so they can be used in several domains
- it has to be possible to find some developmental path through these learning strategies
- representing, storing and retrieving examples is an important first step in acquiring a new strategy
- since several generalizations are possible, an evaluation mechanism is needed to select the most useful strategies

In the remainder of this chapter, I will show a potential example of a general learning strategy, thus addressing the first point on the list. This strategy will be explored in models of two separate tasks. An interesting property of the strategy is that once it is impoverished by removing some of the production rules, it exhibits behavior consistent with a lower level of development. This property is important for the second point: the developmental path through strategies. The models in the remainder of this chapter will model the discovery of new rules, so accommodation in terms of Piaget, or the E1-phase of Karmiloff-Smith. The aspect of assimilation or I-phase, i.e. the use of examples, will be an important topic in the next chapter, as well as the evaluation mechanism.

5.5 Modeling explicit learning strategies in ACT-R

The goal of an explicit learning strategy is to learn new knowledge that is necessary for some new task or domain, or to improve the knowledge already available for an existing task or domain. In order to model this in terms of ACT-R, general learning goals have to be defined, and production rules that operate on these goals. The starting point for learning goals is the predefined dependency chunk-type (see figure 2.9 in chapter 2). Dependency chunks form the basis for new production rules: once a dependency is popped from the goal stack, it is compiled into a production rule. Intuitively, the best way to think of a dependency is to consider it as an example of how to do something. The goal of coming up with such an example can therefore be seen as an explicit learning goal. Eventually, this learning goal will produce a new production rule. In ACT-R, the dependency learning goal needs production rules that matches it. These rules are therefore also part of explicit

learning, and have to be domain independent. So at least the production rules that operate on dependencies are explicit learning strategies for learning new procedural knowledge.

When are explicit learning goals needed? As we have seen earlier in this chapter, we need them if the current approach to the task does not work well. But they are also needed, in the case of a psychological experiment, when participants have to do a task they have never done before, as is often the case. Participants in a psychological experiment need explicit learning strategies to set up initial knowledge structures to perform the task. These strategies need some domain-specific information to work with, for example the following types of information:

Task instructions and examples. In the case of an experiment or educational setting, a task or problem is explained by the experimenter or teacher, and sometimes a few examples are shown.

Relevant facts and biases of other domains in declarative memory. New tasks often build on existing knowledge. Knowledge from related domains can therefore be retrieved and adapted to the task at hand.

Facts and biases in declarative memory from the current domain. As someone gains experience in a new domain, popped goals are accumulated in declarative memory, while declarative learning maintains activation levels and associations with other chunks. This declarative knowledge, similar to the RR model's implicit I-phase knowledge, may serve as a basis for new production rules.

Feedback. If a wrong answer is given based on the current knowledge, and feedback is provided on what the right answer is, this may also be used as a basis for new rules.

Figure 5.10 outlines how a learning strategy works: given initial information in declarative memory, a set of general production rules creates an example of how to do something, a dependency. This dependency is compiled into a new production rule, which has to compete with the rules that have created it. If the task-specific rule performs too poorly, the explicit learning strategies win the competition, and propose new rules, taking into account the feedback (if any) received on the faulty rule. The competition between the task-specific rules and the general learning strategies is the same competition as the competition between search and reflection modeled in the dynamic systems model earlier this chapter.

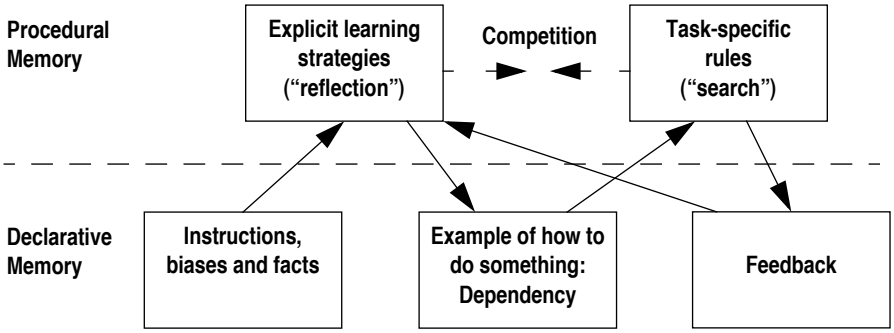


Figure 5.10. General schema of learning strategies in ACT-R

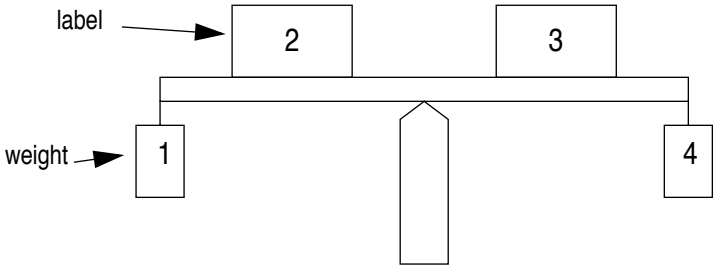


Figure 5.11. Example of the beam task

5.6 An ACT-R model of a simple explicit strategy

The beam task

The task we will start with is a beam task. It is a simplified version of the balanced-beam task, a task of used in developmental studies (Siegler, 1981). The problem is relatively easy: a beam is given, with weights on the left and the right arm. Attached to the arms of the beam are labels, each with a number on it. The task is to predict whether the beam will go left, right, or remain in balance. The numbers on the labels have no influence on the outcome. Figure 5.11 shows an example of a beam. Although the task is easy if we know something about weights and beams, it is much more difficult if we know nothing at all.

The assumption is that the model initially has no task-specific rules about beam-problems. The only procedural knowledge the model has is a set of general rules. Later on, we will use the same general rules for other tasks. The general rules used to learn this task are the following:

Property-retrieval. If there is a task that has a number of objects, create a dependency that contains an example of retrieving a certain property of each of the objects. In the case of the beam task, the objects are the arms of the beam, and weight and label are possible properties. This rule creates a rule that directs attention to a certain aspect, attribute or dimension of the task.

Find-fact-on-feedback. If feedback indicates that the answer is incorrect, and also contains the correct answer, set up a dependency that uses the goal and the answer as examples. Also, retrieve some fact that serves as a constraint in the dependency. The resulting rule will, given a goal, try to fill in the answer using some retrieved fact from declarative memory. To be able to generate correct rules for the beam task, we need to retrieve the fact that a certain number is greater than another number, in order to predict correctly whether the beam will go left or right.

Both general rules involve retrieving an arbitrary chunk from declarative memory, either a property or a fact. Normally the retrieval of arbitrary chunks will not produce the right rules. The chunks retrieved are however not arbitrary, since ACT-R's activation mechanism ensures that the chunk with the highest activation is retrieved. Since activation represents the odds that a chunk is needed, the chunk with the highest odds of being needed is retrieved. This activation can itself again be manipulated by explicit declarative memory strategies such as rehearsal.

In the model, this is reflected by the fact that both property-retrieval and find-fact-on-feedback can be influenced by prior knowledge. If there is an association strength between beam and weight, indicating knowledge that a beam has something to do with weight, property-retrieval will choose weight in favor of label. If there is an association strength between beam and greater-than, a greater-than fact will be retrieved by find-fact-on-feedback. Although this is not part of the model presented here, a possible source of the relevant associations is an implicit learning phase in the sense of the RR theory as discussed in section 5.4.

Since the general rules are just production rules, they can be in direct competition with the task-specific rules they generate. If property-retrieval generates a rule X to retrieve the label, X will compete with property-retrieval. If X is not performing well, for example if it retrieves the irrelevant label, its evaluation will decrease, and it will eventually lose the competition, in which case property-retrieval will create an example of retrieving weight. Although find-fact-on-feedback is only activated if feedback indicates an incorrect answer (i.e., when an expectation-failure occurs), the rules it produces are in competition with each other. The rule with the highest success rate will eventually win.

Figure 5.12 summarizes the property-retrieval rules, and figure 5.13 summarizes the find-fact-on-feedback rules. Both are instantiations of figure 5.10. Figure 5.13 shows

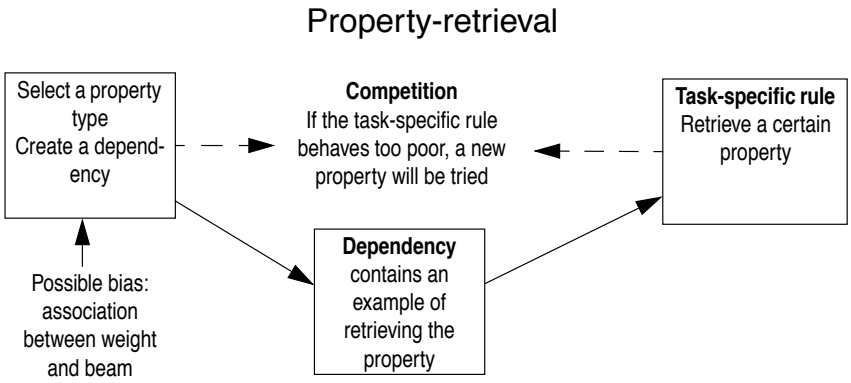


Figure 5.12. How property-retrieval works

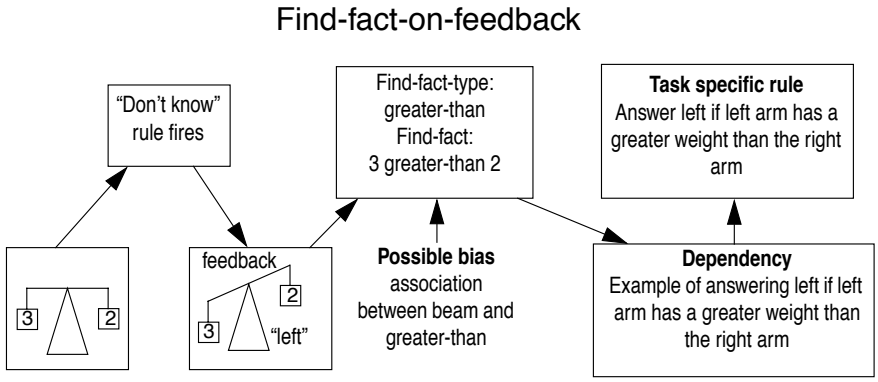


Figure 5.13. How find-fact-on-feedback works

the case in which a "Don't know" rule fires. If instead an incorrect answer is predicted, a dependency is created in the same manner. Apart from the general rules, the model contains lisp functions to generate random beams, and production rules to give feedback. When the model produces an incorrect answer, it will try the same beam again until it can predict the right outcome.

Simulation results

The general rules turn out to be sufficient to learn the task. The following rules are examples of (correct) rules learned by the model. The rule generated by property-retrieval is a rule that retrieves the weight property for both arms of the beam, and stores them in the goal:

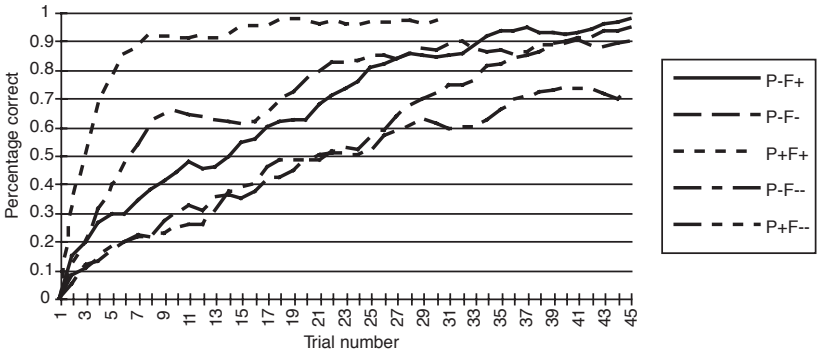


Figure 5.14. Results of the beam model

```
IF    the goal is of type SOLVE-BEAM and refers to two objects O1 and O2
      of which no properties have been retrieved yet
      AND there is a property of O1 of type weight and value V1
      AND there is a property of O2 of type weight and value V2
THEN  add V1 and V2 as properties of type weight to the goal
```

One of the rules generated by find-fact-on-feedback is a rule that predicts when the left arm of the beam will go down.

```
IF    the goal is of type SOLVE-BEAM and two properties V1 and V2 of
      type weight have been identified
      AND there is a fact of type greater-than that specifies V2 is
      greater than V1
THEN  set the answer slot of the goal to LEFT
```

The model was tested in several conditions, differing in the bias given for the properties (P) and the fact-type (F). The following table summarizes the conditions:

- P+ Association between beam and weight
- P- Association between beam and label, a bias for the wrong property
- F+ Association between beam and greater-than
- F- Association between both beam and greater-than, and beam and number, so two possible fact-types were favored.
- F-- No associations between beam and fact-types, four fact-types are possible.

Each experiment has both a P condition and an F condition. Each experiment was run 30 times for 45 trials. Figure 5.14 shows the results. As can be seen in the graph, in the P+F+ condition ACT-R learns to solve the task quite rapidly, and the fact that the model does not reach a 100% score within a few trials is only due to the fact that beams are generated randomly, only occasionally producing a beam in which

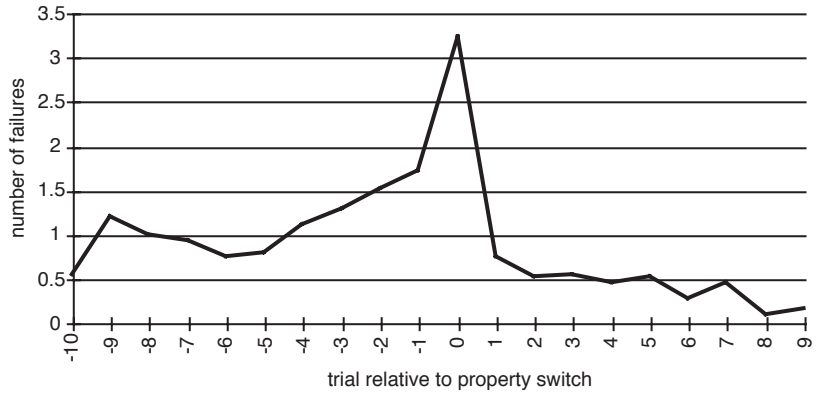


Figure 5.15. Average number of failures for trials relative to a property switch

balance is the correct answer. Performance decreases if the model has less initial information. In the case of the P-F-- condition, the model often fails to find the correct rules for the task. Success depends heavily on the quality of the declarative information. This information does not have to be completely accurate, but some declarative stage before proceduralization is important for success.

The results in figure 5.14 suggest a gradual increase of performance. However, this impression is misleading, as it is caused by averaging 30 runs. If individual runs are examined, each has a certain point where performance increases dramatically. To get a better perspective on this increase, it is necessary to find the exact point at which the increase in performance starts. In one of the conditions, the P-F+ condition, this point is the most obvious: the moment the model switches from examining the label property to examining the weight property. Since this moment is easy to identify in an individual run of the model, it is possible to average results with respect to this point in time. An interesting aspect to average is the number of failed predictions the model makes before it makes the right predictions. Remember the model keeps trying to predict the right answer until it is successful. The result is shown in figure 5.15. It shows the average number of incorrect tries for each trial in the P-F+ condition. At x=0 the model creates a production rule that retrieves the weight properties. As is apparent from the graph, before ACT-R creates this rule, on average three failed predictions are made. Since this clearly establishes that the current task-specific rules are not correct, the general rules can take over and propose new task-specific rules. This process resembles the impasse-insight stages of insight problem solving, and is based on the same mechanisms of the dynamic growth model.

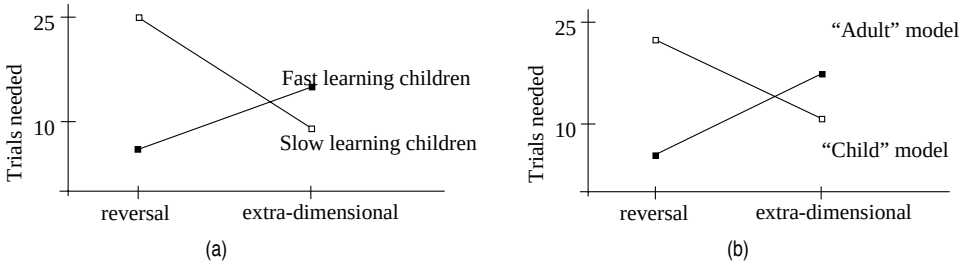


Figure 5.16. Trials needed to learn the discrimination-shift task, (a) from the Kendler & Kendler experiment, (b) by the ACT-R model

Discrimination-shift learning

One of the advantages of explicit learning strategies compared to implicit learning is that they can handle change more easily. If something changes that has been stable for a while, an explicit strategy may react by proposing new knowledge to replace the old. An example of a task in which the rules change is discrimination-shift learning, which I have explained in section 5.4. The ACT-R model of adult behavior uses the same 8 general production rules used in the beam-task, implementing the property-retrieval and find-fact-on-feedback strategies. It learns rules that are quite similar to the rules for the beam task: a rule that focuses on one of the properties of the blocks, either the size or the color, and rules that map specific colors or sizes onto the answers yes and no. This knowledge is closely related to the Rp3-representation of Fischer's theory (figure 5.7). The small-child/animal model uses only 2 of the 8 general production rules, implementing a limited find-fact-on-feedback strategy. The latter model hardly uses any explicit reasoning at all, but rather stores regularities in the environment in production rules. This representation closely resembles Fischer's S2-representation. The results of both ACT-R models are shown in figure 5.16b, producing results quite similar to the Kendler & Kendler data in figure 5.16a.

Despite the fact that the discrimination-shift task is generally not considered to be an insight problem, it nevertheless requires the participant to notice that something has changed, and to discover the new relations. So it can be seen, in a sense, as an elementary insight problem.

5.7 Discussion

The goal of cognitive modeling is to create computer simulations of cognitive processes. A criterion for a good model is whether the results of the simulation match the empirical data. A second criterion that becomes increasingly more important, is the question whether the model can learn the knowledge it needs. A model that uses a large set of specialized production rules is less convincing than a model that gathers its own knowledge. The learning mechanisms which are part of the architecture, are often not capable of doing this job by themselves, so they need augmentation. In the previous chapter I have argued that these mechanisms correspond to implicit learning. The mechanisms can be augmented by explicit learning, that is, implemented by knowledge in memory that directs the implicit learning mechanisms.

Implicit mechanisms are fixed, but explicit strategies have to be acquired. Individuals probably differ in their explicit strategies, although they may well have many in common. Rehearsal, for example, is a strategy used by almost all adults, though it is clearly not something we were born with. An interesting question is whether the same property is also true for other learning strategies. Is there a sequence of rules that unfolds during development? The model of the discrimination-shift task at least hints in this direction, as does Fischer's theory. On the other hand we may well expect large individual differences. Experiments in which participants have to solve difficult problems often show that every participant solves a problem in a different way.

An interesting question is, how the issues discussed here can be related to other architectures. The emphasis on learning models is often attributed to the ascent of neural network models. A neural network model typically starts out with an untrained network, gaining knowledge by experience. Neural networks are powerful in the sense that a three-layer network can learn any function if properly configured. This power is also a weakness, especially if the time taken to learn something is taken into account. Neural networks usually do not have any goal structures, so they lack the mechanisms that are able to focus learning. Karmiloff-Smith, for example, states that neural networks model implicit I-phase learning very well, but are not yet capable of modeling the more explicit phases of skill learning. Raijmakers, van Koten and Molenaar (1996) have shown that a standard feed-forward neural network always behaves like a small child or animal in the discrimination-shift task, being faster at the extra-dimensional shift. To summarize: neural networks do a very good job at implicit learning, but the step towards explicit learning is difficult to make because of the absence of goals and intentional structures.

In the Soar architecture (explained in chapter 2), goals and deliberate reasoning are even more important than in ACT-R (Newell, 1990; see for an extensive comparison

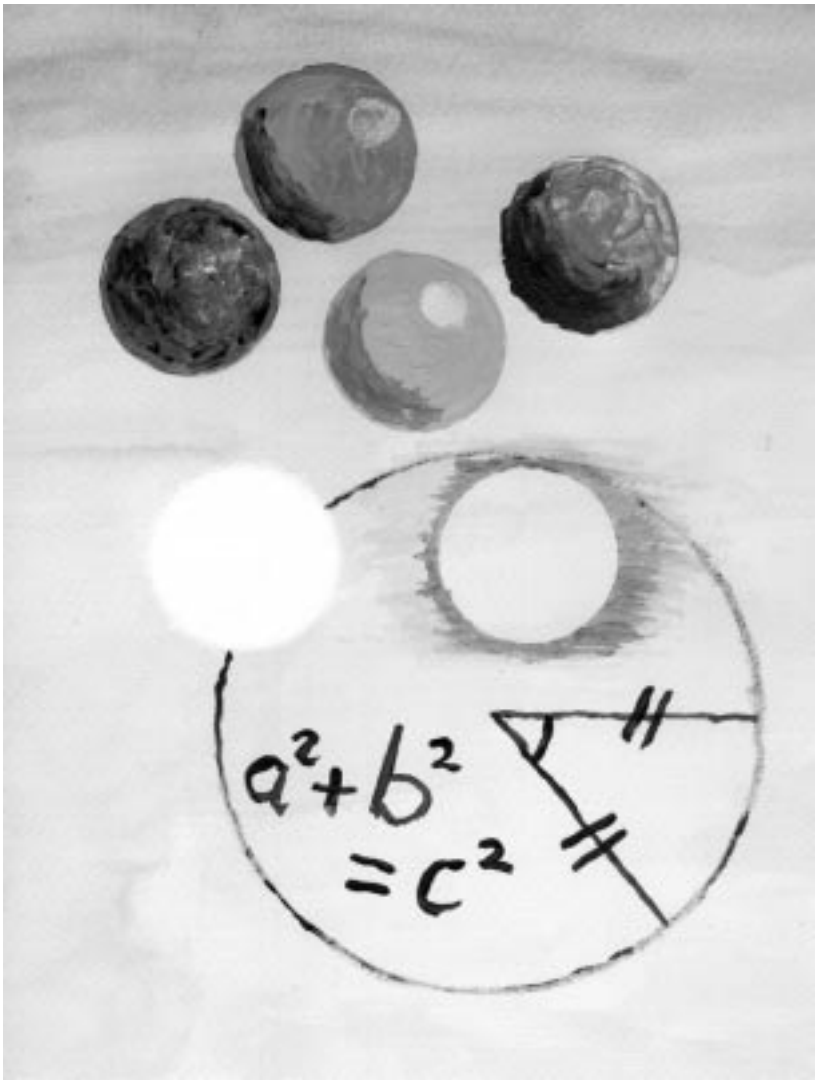
of ACT-R and Soar: Johnson, 1997). The ACT-R models presented in this chapter only use deliberation when existing simple rules prove to be insufficient and, more importantly, if there is any knowledge present on how to deliberate. If ACT-R has to choose between actions A and B, a cost benefit analysis between the rule "do A" and the rule "do B" will decide. Only if both rules prove to perform badly, explicit learning strategies will try to find a more sophisticated rule. A Soar model on the other hand will always try to make a deliberate and rational choice between A and B, a process that may require a lot of processing and specific task knowledge. A Soar model that has to choose between A and B, and has no particular additional knowledge, will get into a infinite sequence of impasses. Soar's single learning mechanism is chunking, which summarizes the processing done between an impasse and its resolution into a new production rule. Although chunking is a mechanism, it is only activated after an impasse has been resolved, so after a deliberate problem solving attempt. Since chunking is Soar's only learning mechanism, this may cause trouble. For example, to learn simple facts, Soar needs the elaborate scheme of data-chunking. Data-chunking eventually produces rules like "IF bird THEN note it has wings". To be able to learn this, however, a lot of deliberation has to be done by production rules *that are not part of the architecture*. In a sense, Soar walks the reverse way: instead of building explicit learning on top of implicit learning, it accomplishes typical implicit learning tasks by elaborate explicit schemes. The critical reader will be able to find more examples of Soar's problems with simple satisficing behavior in Johnson (1997).

Since many other architectures, like EPIC and 3CAPS, currently support no learning at all, ACT-R presently seems to be the best platform to support explicit learning strategies on a basis of implicit learning. To be able to fully sustain explicit learning though, some technical issues in ACT-R must be resolved. Most notably a mechanism must be included to create new chunk-types. The models discussed in this chapter circumvent this problem by using a generic goal type for all goals, but this is hardly a satisfactory solution in the long run.

This chapter may be a starting point for several strands of further research. A more thorough inventory of possible general rules has to be made. This leads to a further question: where do the general rules themselves originate? This question is best studied in a developmental setting. Is it possible to specify a sequence of general rules that are learned during development that can account for the fact that older children can handle more abstract concepts? Unfortunately, I will not answer this question in this thesis: in the next few chapters we will focus on adult problem-solving behavior only.

CHAPTER 6

Examples versus Rules



Acknowledgment
This chapter is written in collaboration with Dieter Wallach, who has developed the model in section 6.3.

An important topic in skill learning is the question of what type of knowledge is learned. Two explanations dominate the discussion. The rule-learning explanation assumes rules are learned by generalizing examples. The instance-based explanation assumes a set of examples is retained. This explanation assumes improved performance can be explained by the fact that a solution is retrieved from memory instead of being calculated again. Both types of explanation are compatible with ACT-R, and this chapter will explore the question of how to choose between the two. The central idea will be that the type of learning with the best expected gain will dominate performance. This will be demonstrated using two models. The first of these models the Sugar Factory task, a task in which performance can be explained by instance learning alone. The second models the Fincham task, in which the expected gain of both the use of instances and the use of rules can be examined in detail.

6.1 Introduction

The models in the previous chapter made an important assumption about learning new skills, the assumption that they are represented as production rules. An alternative account of skill learning is that people store examples, and later retrieve these examples if they encounter the same or a similar situation.

The question whether skills are realized as abstract rule-like entities or as sets of concrete instances is one of the central distinctions in cognitive science, spreading across fields as diverse as research on memory, problem solving, categorization or language learning (Logan, 1988; Hahn & Chater, 1998; Redington & Chater, 1996; Plunkett & Marchman, 1991; Lebiere, Wallach, & Taatgen, 1998). Hahn and Chater (1998) proposed that the distinction between instance- and rule-based learning mechanisms cannot be based on different types of representations, but must be seen within the framework of their use in problem solving. We extend their argument and emphasize the necessity of an integrated investigation of human skill acquisition using a comprehensive theory of cognition.

The view of skill acquisition as learning and following abstract rules has dominated theories of skill acquisition over the last decades, whether encoded in production systems (Newell & Simon, 1972; Anderson, 1993), stored as logical implications or represented in classifier systems (Holland, Holyoak, Nisbett & Thagard, 1986). While these approaches differ in many aspects, they share the assumption that cognitive skills are realized as abstract rules that are applied to specific facts when

solving problems. In ACT-R, it is assumed that people start out with concrete examples of previous problem solving episodes that are generalized to abstract rules. These rules can be applied in subsequent problem solving and can thus account for increased performance. Discontinuous improvements in cognitive performance (Blessing & Anderson, 1996) can be taken as further evidence for the acquisition of rules. While Anderson (1993) describes the view that cognitive skills are realized as (production) rules as "one of the most important discoveries" in cognitive psychology, Logan (1988) argues for domain-specific instances as the basis for cognitive skills. According to this instance theory, general-purpose procedures or algorithms are applied to solve novel problems. Each time such a procedure is used in problem solving, its solution is retained as a separate instance. For new problems, the solution can be calculated, or a previous one can be retrieved and applied to the current problem. The retrieved solution can be used as a whole, in part, or in an adapted version to obtain the solution of the new problem.

An important source of evidence for the instance-based approach is the fact that repeating a certain specific example of a problem increases performance on this example, but not on other ones. The fact that participants cannot verbalize abstract knowledge about the problems solved is frequently cited as further evidence against some form of generalization, as implied by rule-based skill theories. ACT-R, however, assumes that rules themselves cannot consciously be inspected, so this second source of evidence is not as convincing as the first.

Evidence for the fact that knowledge is represented as production rules comes from research on the *directional asymmetry* of rules. A production rule has two parts, a condition and an action, which we informally denote as 'IF condition THEN action'. In a production system, control always flows from the condition to the action. In many practical cases, the condition and the action are both part of a pattern, for example the pattern AB. A rule like 'IF A THEN B' can be used to complete the pattern given A. In an instance approach, the pattern AB can be stored as an instance, and retrieved given either A or B. If participants are trained to complete some pattern AB on the basis of A, a rule approach predicts that they learn the rule 'IF A THEN B', and the instance approach predicts that they learn the instance AB. If participants are consequently asked to complete AB on the basis of B, the instance approach would not predict a decrease in performance. The rule-based approach, however, suggests that a new rule would have to be learned for the 'IF B THEN A' case, resulting in worse performance.

Another apparent source of evidence stems from the fact that rules are more general than instances, which are assumed to be represented in a relatively unprocessed form (Redington & Chater, 1996). If participants show increased performance on examples they have not encountered before, some generalized knowledge can be postulated as the basis of the observed performance. This second source of evidence is, however, unreliable. It assumes that stored examples can only be used when the

new example is literally identical to one of the stored examples. If one or more old examples (or fragments of them) can be used to improve performance on a new example in a less direct fashion, generalization is also possible in an instance-based setting. Consequently, if generalization in transfer experiments is used as evidence against instance theory, it must be made clear that the answer to a certain problem cannot easily be derived from answers to previous problems. As Redington and Chater (1996) have pointed out, surprisingly simple models, relying on represented fragments of observed stimuli, can perform exceedingly well in transfer tasks without acquiring any abstract knowledge. An example of such a model will be discussed in section 6.3 when we demonstrate the scope of a purely instance-based approach in accounting for data that Broadbent and his colleagues (Broadbent, 1989) have interpreted as evidence against ACT-R's claim that production rules are learned on the basis of examples. Their results on dissociations between knowledge and performance seem to imply that participants can acquire rules to successfully operate complex systems without showing an increased performance in answering questions about the system's behavior. Our instance model will provide a very simple explanation for this dissociation result.

6.2 Learning strategies

The learning mechanisms in ACT-R are all quite basic, and can be used in several different ways to achieve different results. In chapter 4, it is argued that the learning mechanisms of ACT-R correspond to the psychological notion of implicit learning, since they are always at work, do not change due to development and show few individual differences. Explicit learning, on the other hand, is tied to intentions — to goals in ACT-R terms — and can better be explained by a set of learned strategies.

In this chapter we will discuss a paradigm for skill learning that involves both implicit learning and an explicit strategy. Figure 6.1 shows an overview of this paradigm. First we assume people have some initial method or algorithm to solve the problem. Generally this method will be time-consuming or inaccurate. Each time an example of the problem is solved by this method, an instance is learned. In ACT-R terms, an instance is just a goal that is popped from the goal stack and is stored in declarative memory. Since this by-product of performance is unintentional, it can be considered as implicit learning.

Other types of learning require a more active attitude. If the initial method is too time consuming, one may try to derive an abstraction to increase efficiency. If the initial method leads to a large number of errors, new relationships in the task may be deduced or guessed in order to increase performance. The next step, from abstraction to production rule, can only be made if the abstraction is simple enough to convert to a production rule. Since proceduralization is usually not considered

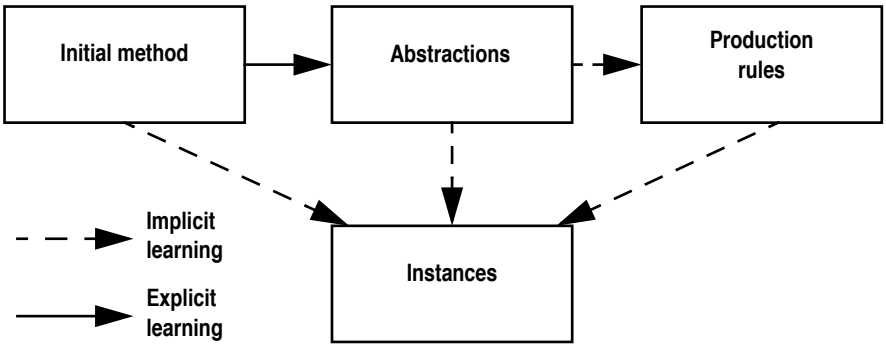


Figure 6.1. Overview of the proposed skill learning paradigm

something that is under conscious control, it is a form of implicit learning as well. This idea is not entirely consistent with ACT-R's production compilation mechanism. We will return to this issue in the discussion at the end of the chapter. Both the application of abstractions and the firing of new production rules will create new instances. Regardless of what is going on due to explicit learning, implicit learning keeps accumulating knowledge.

If we have that many ways of learning, what type of learning will we witness in a particular experiment? To be able to answer this question we go back to the principle of rational analysis. According to this principle, the type of learning that will be principally witnessed is the type that will lead to the largest increase in performance. If we have a task in which it is very hard to discover relationships or abstractions, learning will be characterized primarily by implicit instance learning. In tasks where each instance is different from the others, but where generalization is relatively easy, the best explanation of performance will probably involve the learning of rules.

Before discussing specific models, both learning instances and production rules will be examined in more detail. The abstractions used in this chapter are still very simple structures, and will be elaborated in the next chapter.

Instance-based learning

The last thirty years have seen a number of different experimental paradigms investigating the concept of implicit learning in domains as diverse as learning artificial grammars (Reber, 1967), sequence learning (Willingham, Nissen, & Bullemer, 1989) or learning to control complex systems (Berry & Broadbent, 1984). All these studies share the claim that participants learn more about structural properties of the tasks than they are able to verbalize. To explain these findings, an implicit mode of learning has been distinguished from an explicit mode. Berry and Broadbent (1995) characterize the implicit mode as

> […] a process whereby a person learns about the structure of a fairly complex

> stimulus environment without necessarily intending to do so, and in such a way that the resulting knowledge is difficult to express.

In opposition to this characterization they refer to explicit learning as involving

> […] deliberate attempts to solve problems and to test hypotheses, and people are usually aware of much of the knowledge that they acquired.

The distinction between two learning modes has not remained unchallenged (c.f. Perruchet & Amorim, 1992; Perruchet & Pacteau, 1990; Buchner, 1994) but is cited frequently as evidence against the conception of declarative knowledge as the source for the acquisition of procedural knowledge as is assumed in the ACT-framework. Broadbent (1989) argues that the study of Berry and Broadbent (1984) contradicts the ACT claim since participants seem to learn rules for successfully operating a complex system without being able to consciously state these rules. Berry and Broadbent (1984) even found negative correlations between task performance and the ability to answer specific questions about the system's behavior.

In section 6.3 we propose an explanation for the reported dissociation between knowledge and performance by analyzing instance-based learning in an ACT-R model and comparing it to Logan's instance theory.

Learning production rules

In the previous chapter some strategies for learning task-specific rules were discussed. We will now extend those methods to a general scheme for procedural learning in ACT-R. Both the property-retrieval and the find-fact-on-feedback strategy have the desirable property that they can be used for several different tasks. The implementation of these strategies in terms of production rules is, however, rather ad hoc. This becomes an issue if the question of how these strategies themselves are learned is raised. In this chapter we will, therefore, propose a more general approach to learning new production rules. The idea is to have a standard method to construct a dependency, the declarative memory structure needed for a new production rule. Explicit learning strategies can extend this standard method. The advantage is that a learning strategy no longer has to take care of the whole process of creating a dependency, but only modifies some of the details.

It is important to note that the method of learning new productions presented here has two aspects. On the one hand, some principled decisions are made that have psychological relevance. On the other hand, there is a "programming" aspect involved: the method must produce the right rules. As a consequence, some, but not all steps in the production learning process are defendable in psychological terms.

The many constraints ACT-R poses on production rules actually simplify the problem of finding this basic method of production rule learning. Consider the most common type of production rule: a rule that matches the goal, retrieves a fact from declarative memory, and modifies the goal:

```
IF    the goal has a certain type and satisifies certain
      properties
      AND there is a fact in declarative memory that satisfies
      certain constraints
THEN  modify one or more slots of the goal
```

The dependency necessary to learn this rule requires four principal components: the dependency itself, an example goal before the desired rule is executed, an example solution after the desired rule is executed, and the fact that is retrieved. Let us examine these four components and investigate how they may be derived.

The easiest component is the example goal. Assuming rules are derived at the point they are needed, the example goal is actually the current goal at the moment the assembly of a dependency is started. The next component is the dependency itself. Since ACT-R requires that all elements in declarative memory are former goals themselves (apart from chunks acquired through perception), the dependency must be pushed onto the goal stack at some point. The best time to do this is right at the beginning, in order to change the context from normal processing to a production learning setting. Since any goal setting may be appropriate for learning new rules, a rule is needed that pushes a dependency as a subgoal regardless of the current goal. As we already mentioned, the current goal is one of the four components needed, so we immediately stick it into its rightful place: the goal-slot of the dependency:

```
IF    the goal is anything
THEN  push as a subgoal a dependency with the original goal in
      the goal slot of the dependency
```

This rule always matches, and can interrupt normal information processing at any moment. The rule has a high cost associated with it, since it will be followed by extra processing that is not directly necessary for normal performance. The rate at which this rule will fire is directly related to the rules it competes with. If competing rules have high expected gain values, this rule will fire rarely. If competing rules have low expected gains, due to the fact that they are inaccurate or costly, this rule will fire more often. So the frequency with which dependencies are produced depends on the amount and quality of the knowledge that is already available. This is the same mechanism as the search-reflection trade-off discussed in the previous chapter.

After the dependency-pushing rule has fired, we end up with a dependency on top of the goal stack. This is illustrated in figure 6.2a: on top of some arbitrary task goal X, a dependency has been pushed as a subgoal. Only one slot of the dependency is

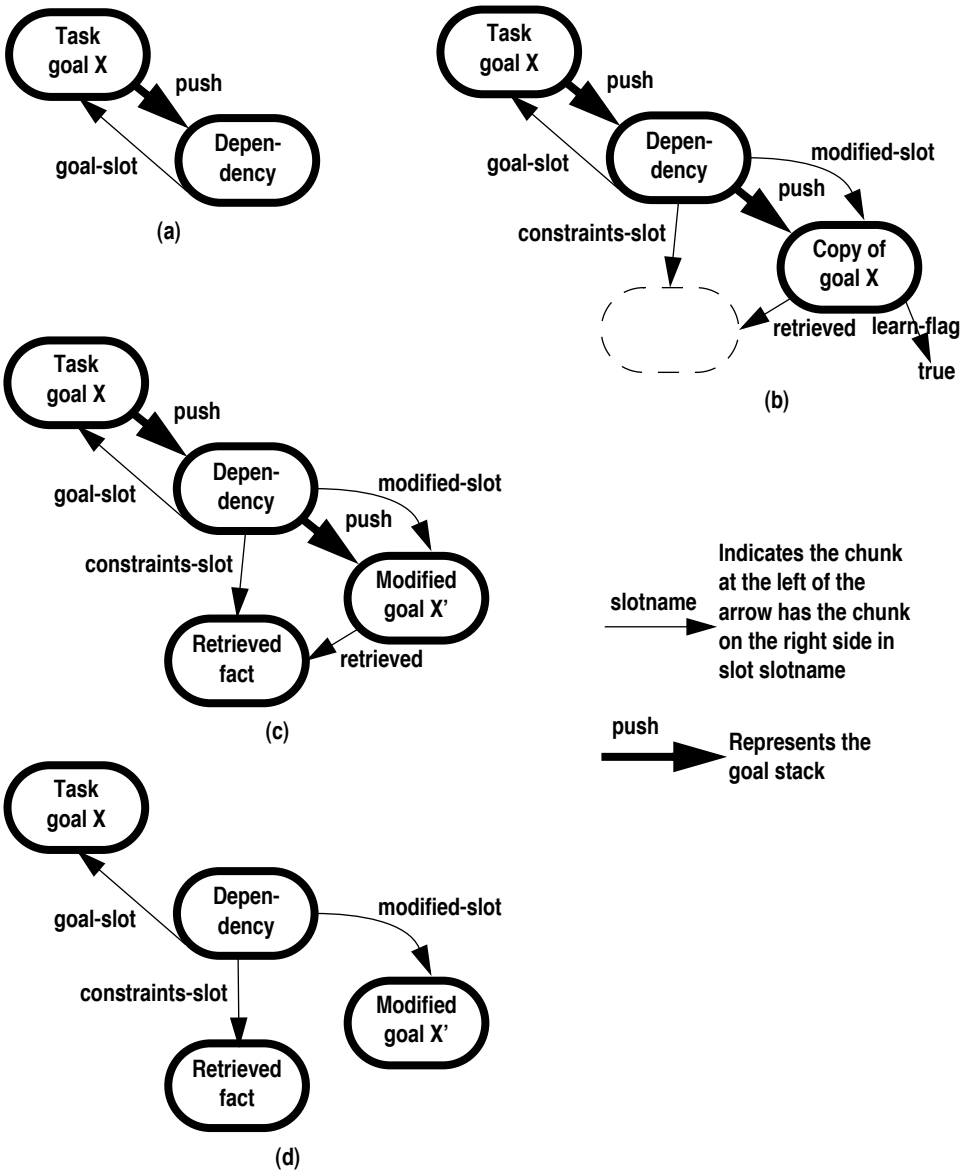


Figure 6.2. General method to create dependencies on the fly. (a) a dependency is pushed. (b) the a copy of the original goal is pushed with a place holder for the retrieved fact. (c) the goal is modified using some retrieved fact. (d) both the modified goal and the dependency are popped, leaving a completed dependency structure.

filled: the goal slot. The next step is to fill in the remaining slots of the dependency, as far as necessary. The main two slots to fill are the modified slot and the constraints slot. Some way has to be found to propose some modified goal. At this point we need some explicit learning strategy that can reason out the next step, take a guess or whatever. In order to take this next step, however, we need to restore the original goal context. This is accomplished by pushing a copy of the original goal as a new subgoal, and creating a placeholder for the retrieved fact in that subgoal.

```
IF    the goal is a dependency and the modified slot is nil
      and G is in the goal slot of the dependency
THEN  push a copy GC of G as a subgoal, set the learn flag of GC
      to true, and create a place holder in the retrieved
      slot of GC
      AND put GC in the modified slot of the dependency
      and set the constraints slot of the dependency to the
      place holder
```

After this rule has fired, the goal stack contains three items: the original goal, a dependency, and a copy of the original goal (figure 6.2b). The copy of the original goal has its learn flag set to true, so rules that implement explicit learning strategies are allowed to fire. The next step is that the copy of the goal is modified. This may be due to explicit learning strategies, but may also be 'regular' problem-solving steps (figure 6.2c). Once the goal is modified using some fact that is retained in the retrieved slot, it is popped while removing the learn flag:

```
IF    the goal is has its learn-flag set to true
      and the retrieved slot the goal is not nil
THEN  set the learn-flag to nil and pop the goal
```

At that point, further slots of the dependency may be filled, the dependency itself is popped, and ACT-R's production compilation mechanism creates a new production rule. Now we are back in the original situation in the original goal (figure 6.2d), but with a new production rule that can modify it.

The advantage of the method outlined above is that learning strategies do not have to handle dependencies themselves, which is a big hassle. A learning strategy now only needs to recognize the learn-flag, and modify the goal while putting some fact in the retrieved slot of the goal. The method can also be modified slightly to produce production rules that push a subgoal instead of retrieving a fact. This is accomplished by simply using the stack slot of the dependency instead of the constraints slot.

The important thing to note in the method above is that procedural learning is part of normal processing, in the sense that it can be initiated at any moment. The fact that the goal needs to be copied in the subgoal, and some of the manipulations in this subgoal, are a bit awkward from a cognitive perspective. In the next chapter we will

pull out all knowledge-based processing from the dependency subgoal. In that way, the actual process of learning a production rule becomes more like an implicit-learning mechanism.

6.3 Sugar Factory

In contrast to rule-based approaches that conceptualize skill acquisition as learning of abstract rules, theories of instance-based learning argue that the formation of skills can be understood in terms of the storage and deployment of specific episodes or instances (Logan, 1988; 1990). According to this view, abstraction is not an active process that results in the acquisition of generalized rules, but rule-like behavior emerges from the way specific instances are encoded, retrieved and deployed in problem solving. While ACT-R has traditionally been associated with a view of learning as the acquisition of abstract production rules (Anderson, 1983; 1993), we present a simple ACT-R model that learns to operate a dynamic system based on the retrieval and deployment of specific instances (i.e. chunks) which encode episodes experienced during system control. The ACT-R model will be compared to a model by Dienes and Fahey (1995). This comparison will involve both the accuracy of the predictions and the assumptions made by each of the models.

The Task

Berry & Broadbent (1984) used the computer-simulated scenario Sugar Factory to investigate how subjects learn to operate complex systems. Sugar Factory is a dynamic system in which participants are supposed to control the sugar production *sp* by determining the number of workers *w* employed in a fictional factory. The behavior of Sugar Factory is governed by the following equation:

$$sp_t = 2w_t - sp_{t-1} + \text{random component (-1, 0, or 1)} \quad \textbf{(6.1)}$$

The number entered for the workers *w* can be varied in 12 discrete steps $1 \leq w \leq 12$, while the sugar production changes discretely between $1 \leq sp \leq 12$. To allow for a more realistic interpretation of *w* as the number of workers and *sp* as tons of sugar, these values are multiplied in the actual computer simulation by 100 and 1000, respectively. If the result according to the equation is less than 1, *sp* is simply set to 1. Similarly, a result greater than 12 leads to an output of 12. Participants are given the goal to produce a target value of 9000 tons of sugar (so *sp*=9) on each of a number of trials. They are given no information at all about the relationship between present output, number of workers and previous output.

The models

Based on Logan's instance theory (1988; 1990) Dienes & Fahey (1995) developed a computational model (the D&F model) to account for the data they gathered in an experiment using the Sugar Factory scenario. According to instance theory, encoding and retrieval are intimately linked through attention: encoding a stimulus is an unavoidable consequence of attention, and retrieving what is known about a stimulus is also an obligatory consequence of attention. Logan's theory postulates that each encounter of a stimulus is encoded, stored and retrieved using a separate memory trace. These separate memory traces accumulate with experience and lead to a "gradual transition from algorithmic processing to memory-based processing" (Logan, 1988, p. 493). The ACT-R model is also based on Logan's ideas, but differs in the way they are worked out.

Both models assume some algorithmic knowledge prior to the availability of instances that could be retrieved to solve a problem. Dienes & Fahey (1995, p. 862) observed that 86% of the first ten input values that subjects enter into Sugar Factory can be explained by the following rules:

1. If the sugar production is below (above) target, then increase (decrease) the amount of workers with 0, 100, or 200.
2. For the very first trial, enter a work force of 700, 800 or 900.
3. If the sugar production is on target, then respond with a workforce that is different from the previous one by an amount of -100, 0, or +100 with equal probability.

While this algorithmic knowledge is encoded in the D&F model by a constant number of prior instances that could be retrieved in any situation, ACT-R uses simple production rules to represent this rule-like knowledge. The number of prior instances encoded is a free parameter in the D&F model that was fixed to give a good fit to the data reported below. There is no equivalent parameter in the ACT-R model.

Logan's instance theory predicts that every encounter of a stimulus is stored. The D&F model, however, only stores instances for those situations in which an action successfully leads to the target. All other situations are postulated to be forgotten immediately by the model. ACT-R, on the other hand, encodes every situation, irrespective of its result. The following chunk is an example of an instance stored by the ACT-R model:

```
transition1239
      ISA transition
      STATE 3000
      WORKER 800
      PRODUCTION 12000
```

The chunk encodes a situation in which an input of 800 workers, given a current production of 3000 tons, led to subsequent sugar production of 12000 tons.

The assumption that only successful instances are stored is not problematic in itself. The problem is that the D&F model uses a "loose" definition of what is successful. Due to the random component in the equation the outcome may be 1000 more or less than expected. Therefore an output of between 8000 and 10000 was considered successful by the model. This generous scheme of success was not available to participants: for them only an outcome of 9000 meant success.

Retrieving instances

In the D&F model each stored instance "relevant" to a current situation races against others and against prior instances representing algorithmic knowledge. The fastest instance determines the action of the model. An instance encoding a situation is regarded to be "relevant", if it either matches the current situation exactly, or does not differ from it by more than 1000 tons of sugar in either the current output or the desired output, analogous to the loose range discussed above. Retrieval in the ACT-R model, on the other hand, is governed by similarity matches between a situation currently present and encodings of others experienced in the past (see Buchner, Funke & Berry, 1995 for a similar position in explaining the performance of subjects operating Sugar Factory). On each trial, a memory search is initiated based on the current situation and the target state '9000 tons' as cues in order to retrieve an appropriate intervention or an intervention that belongs to a similar situation. The following production rule is used to model the memory retrieval of chunks based on their activation level:

```
IF    the goal is to find a transition from the current state with
      output current to a state with new output desired
      AND there is a transition in declarative memory, with
      current output current and new output desired and a number
      of workers equal to number
THEN  set the number of workers in the goal to number
```

This rule will normally only retrieve an old situation that exactly matches the current situation. However, ACT-R can also match chunks that do not exactly match the rule by a process called *partial matching*, which was mentioned briefly in chapter 2. This means that an old situation may also be retrieved if it is slightly different from the current situation. Instances which only partially match the retrieval pattern, i.e. which do not correspond exactly to the current situation will be penalized by lowering their activation proportional to the degree of mismatch. Activation noise is introduced to allow for some stochasticity in memory retrieval.

As figure 6.3 shows, the use of instances instead of the initial algorithmic knowledge increases over time, resulting in the gradual transition from algorithmic to memory-based processing as postulated by Logan (1988, p. 493).

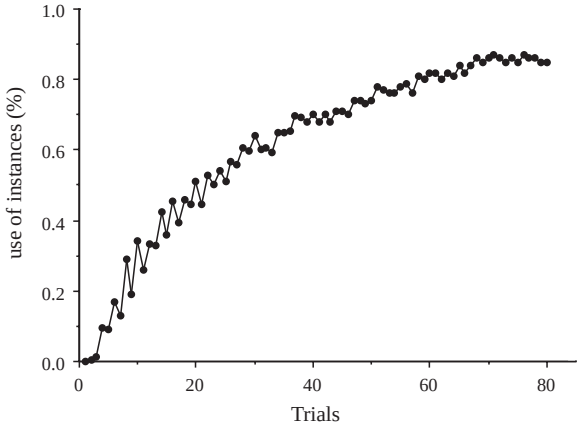


Figure 6.3. Relative use of instance retrieval per trial by the ACT-R model

Theoretical Evaluation

While the two models of instance-based learning share some striking similarities, the D&F-model makes unrealistic assumptions with respect to the storage and the retrieval of instances. Dienes & Fahey (1995) found out that these critical assumptions are essential to the performance of the D&F model (p. 856f):

> The importance to the modeling of assuming that only correct situations were stored was tested by determining the performance of the model when it stored all instances. (…) This model could not perform the task as well as participants: the irrelevant workforce situations provided too much noise by proscribing responses that were in fact inappropriate (…) If instances entered the race only if they exactly matched the current situation, then for the same level of learning as participants, concordances were significantly greater than those of participants.

Since the ACT-R model does not need to postulate these assumptions, this model can be regarded as the more parsimonious one, demonstrating how instance-based learning can be captured by the mechanisms provided by a unified theory of cognition.

Empirical Evaluation

While the theoretical analysis of the assumptions underlying the two models favors the ACT-R approach, we will briefly discuss the empirical success of the models with respect to empirical data reported by Dienes and Fahey (1995). Figure 6.4 shows the trials on target when controlling Sugar Factory over two phases, consisting of 40 trials each. ACT-R slightly overpredicts the performance found in

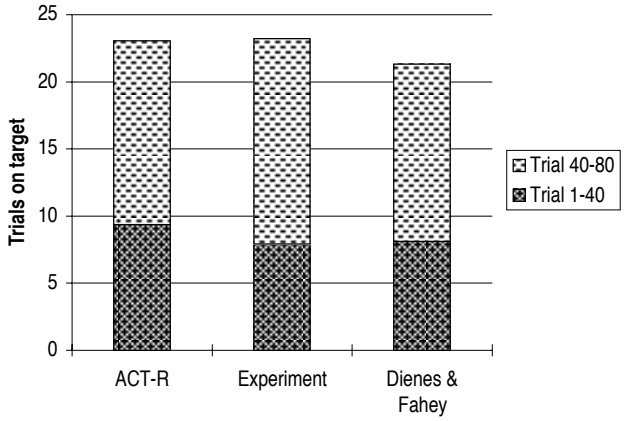


Figure 6.4. Number of trials on target in the experiment, the ACT-R model and the D&F model for the first and second half of the experiment conducted by Dienes & Fahey (1995)

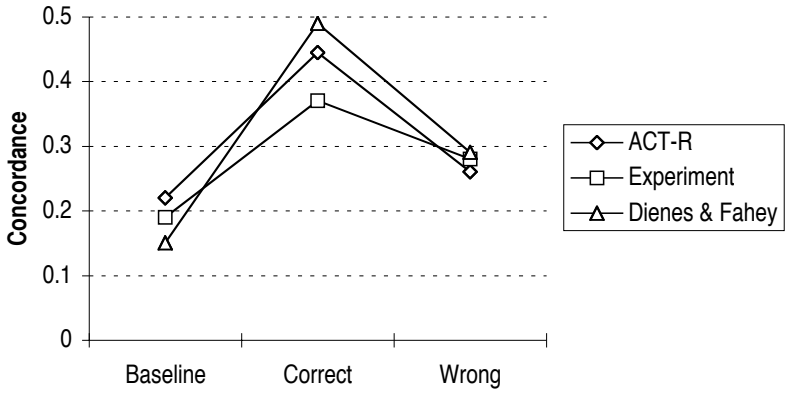


Figure 6.5. Concordances for the experiment and both models

the first phase, while the D&F model slightly underpredicts the performance of the subjects in the second phase. Since both models seem to explain the data equally well, we cannot favor one over the other.

After the participants had controlled the Sugar Factory for 80 trials, they had to do a slightly different task. Again they had to determine the work force in 80 situations, but now they did not receive feedback, but just moved on to a new, unrelated situation. The 80 situations presented were the last 40 situations from the first part of the experiment mixed with 40 new situations.

Figure 6.5 shows how the percentage of times (concordance) participants chose the same work force in this second task as they did in the first. The baseline level

represents the chance that both choices are equal due to random choice. This chance is higher than 1/12, because some choices are made more often during the experiment than others. The correct column shows how often the same work force is chosen if this leads to a correct output, and the wrong column shows the same for the incorrect outputs. Again, both models seem to do a similarly good job in explaining the data, with neither model being clearly superior.

Conclusion

We discussed and compared a simple ACT-R model to an approach based on Logan's instance theory with respect to their ability to model the control of a dynamic system. While both models were similar in their empirical predictions, the ACT-R model was found to require fewer assumptions and is thus preferred over the model proposed by Dienes & Fahey (1995). Generally, ACT-R's integration of an activation-based retrieval process with a partial matcher seems to be a very promising starting point for the development of an ACT-R theory of instance-based learning and problem solving.

6.4 The Fincham task

An example of a task in which both rule learning and instance learning are viable strategies is described by Anderson & Fincham (1994). In this task, participants first have to memorize a number of facts. These facts look like this:

> "Hockey was played on Saturday at 3 and then on Monday at 1."

We will refer to these facts as "sports-facts" to prevent confusion with facts and rules in the model. A sports-fact contains a unique sport and two events, each of which consists of a day of the week and a time. After having memorized these facts, participants were told they really are rules about the time relationships between the two events. So in this case "Hockey" means you have to add two to the day, and subtract two from the time. In the subsequent experiment, participants were asked to predict the second event, given a sport and a first event, or predict the first event, given the sport and the second event. So participants had to answer questions like: "If the first game of hockey was Wednesday at 8, when was the second game?" Figure 6.6 shows an example of the interface used in the experiment. In this paradigm, it is possible to investigate evidence for both rule-based learning and instance-based learning.

Directional asymmetry, evidence for rule-based learning, can be tested for by first training participants to predict events in one direction for a certain sports-fact, and then reverse the direction and look how performance in the reverse direction relates to performance on the trained direction. Evidence for instance learning can be

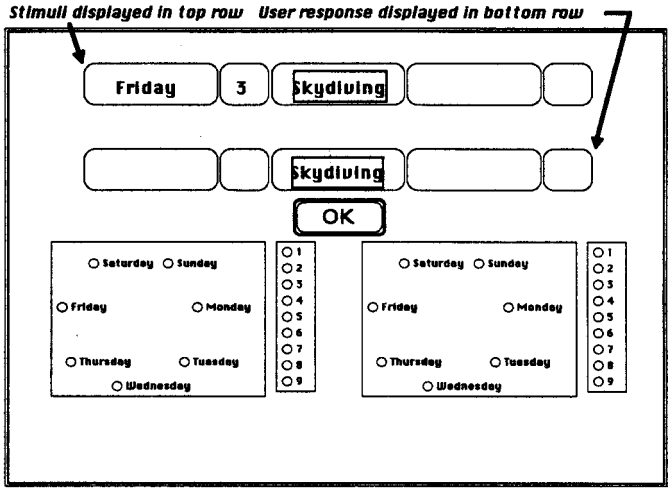


Figure 6.6. Example of the interface used in Anderson & Fincham (1994) and Anderson, Fincham & Douglass (1997). From Anderson, Fincham & Douglass (1997).

gained by presenting specific examples more often than other examples. Better performance on these specific examples would indicate instance learning. Anderson & Fincham (1994), and later Anderson, Fincham & Douglass (1997) performed five variations on this basic experiment, three of which we will discuss here. But before discussing the specific experiments, we will first take a look at the ACT-R model we have developed.

The ACT-R model

The central assumption of our model of the Fincham task is that the data can only be explained by multiple strategies. We will use the four strategies discussed by Anderson, Fincham & Douglass (1997): *analogy*, *abstraction*, *rule* and *instance*. These strategies have different cost-success profiles (summarized in figure 6.9), which determine at what stage of the learning process they will be most prominent. Figure 6.7 shows schematic representations of each of the strategies. Since each problem involves calculating a day and a time, two separate sub-problems have to be solved. Each of these strategies corresponds to one of the boxes in figure 6.1.

The analogy strategy (figure 6.7a) has the highest cost, but only needs the sports-facts learned initially. Starting at the top goal, a subgoal is pushed onto the goal stack to either find the day or the time. To be able to do this, the original example must first be retrieved, and the appropriate elements (days or times) must be extracted. Another subgoal takes care of this stage. After retrieving the example, this second subgoal is popped, and a new subgoal is pushed to make an analogy between the example and the current problem. First the relation in the example is determined, for example the fact that two has to be subtracted from the day. Most of the time, this

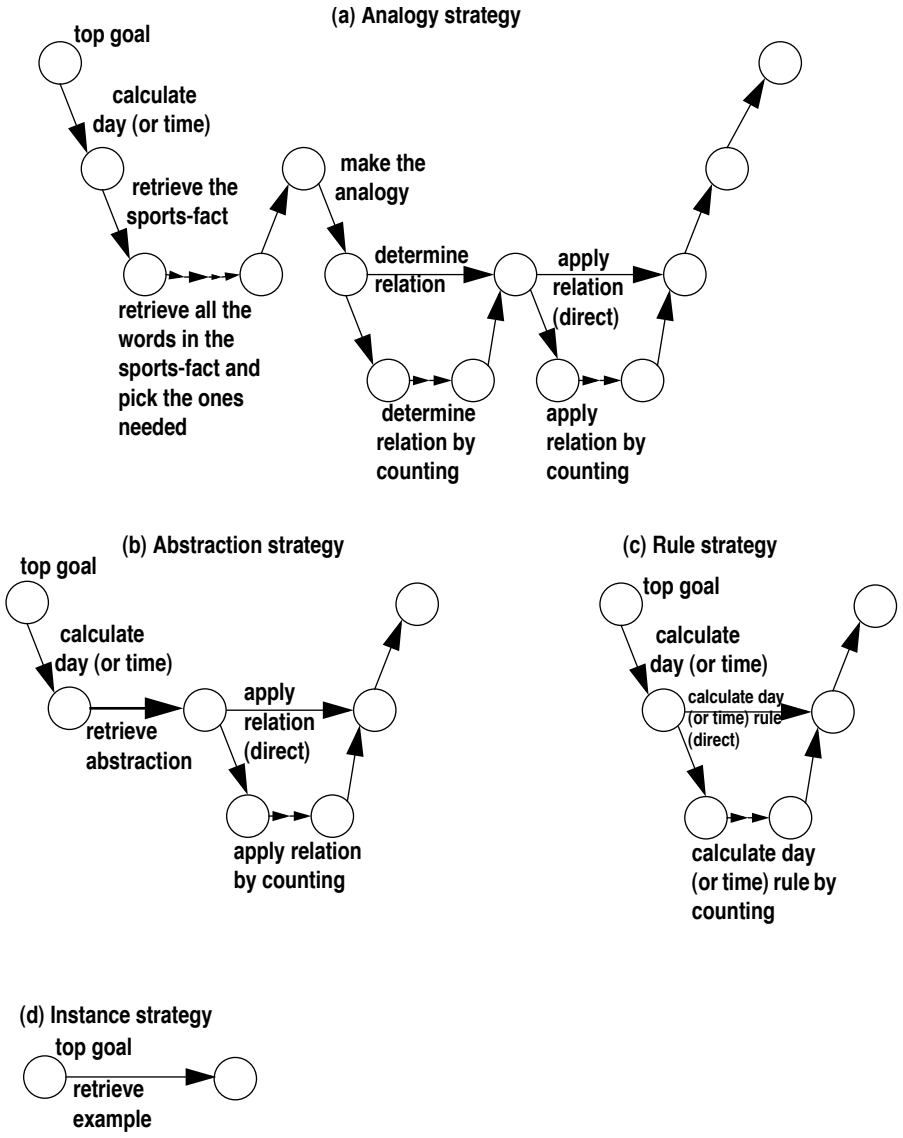


Figure 6.7. Schematic representation of the four possible strategies used in the model. Note that two strategies (possibly the same) are needed to solve the whole problem: one for the day of the week and one for the time of the day

relationship can be determined directly by retrieval, for example the relationship between four and six. But sometimes, as in the case of days of the week, this has to be done by counting. To determine the relationship between Sunday and Friday, one has to count two steps back from Sunday. Counting is taken care of by an additional subgoal, with the advantage that this subgoal is added to declarative memory and can be retrieved during later trials to determine the relation directly. After determining the relationship in the example, this relation is applied to the current problem. This can again be direct, or through a counting subgoal.

The analogy strategy requires prior knowledge. The model assumes that people already know how to make simple analogies, how to memorize and recall strings of words, and that they know relationships between numbers and days of the week, and are able to calculate these relations if they cannot be retrieved from memory. The rest of the necessary knowledge, mainly involving perceptual-motor operations like reading the information on the screen and entering the answers, has to be learned by the participants during the instructions. This aspect of the task is not modeled.

The abstraction strategy (figure 6.7b) assumes knowledge about the relation between the two days or two times for a certain sport. For example, "Hockey" means "add two to the days". An abstraction in the model is a declarative fact that stores this information, for example:

```
ABSTRACTION234
      ISA ABSTRACTION
      SPORT HOCKEY
      TYPE DAY
      RELATION PLUS2
```

Using an abstraction to find the answer only requires two steps: retrieve the abstraction and apply it to the current problem. The second step, application, may involve another counting subgoal, similar to the analogy strategy. Although the abstraction strategy is more efficient than the analogy strategy, it requires knowledge participants initially do not have: abstractions.

The rule strategy (figure 6.7c) uses production rules to find the answer. Each of the rules has two versions, one that retrieves the answer, and one that calculates the answer. An example of a retrieve rule is:

```
IF    the goal is to find the day of the second event, the sport
      is hockey and the day of the first event is day1
      AND day1 plus two days equals day2
THEN  put day2 in the second event slot of the goal
```

The calculate version pushes this calculation as a subgoal, which is handled by the same production rules that determine and apply the relations in the analogy strategy. An example of this second version is:

```
IF    the goal is to find the day of the second event, the sport
      is hockey and the day of the first event is day1
THEN  push as a subgoal to find the answer to day1 plus two days
      AND put the answer in the second event slot of the goal
```

The advantage of the rule strategy is that its costs are much lower than those of the analogy strategy, and also slightly lower than the costs of the abstraction strategy, since the answer can be found in a single step. However, in order to use it, the necessary production rules must be learned. Furthermore, the two example rules given only calculate the second event given the first. To calculate the first event given the second, two additional rules are needed.

The strategy with the lowest costs is the instance strategy (figure 6.7d). It can be applied to the top-goal, since it retrieves the answer from past subgoals directly. This strategy will only work if the appropriate instance is available. An example of an instance is:

```
ITEM434
      ISA ITEM
      SPORT HOCKEY
      TYPE DAY
      LEFT SUNDAY
      RIGHT TUESDAY
```

To be able to fully depend on this strategy, all possible examples have to be learned. For each sports-fact, seven to nine examples are needed.

The abstraction, rule and instance strategy are actually short-cuts for the original analogy strategy. The abstraction and rule strategy make short-cuts at the subgoal level of the analogy strategy, and the instance strategy directly at the top level. The knowledge needed for the instance short-cut is gained automatically, since the popped subgoals serve as examples. To be able to use an example, its activation must be high enough, so it has to be repeated a number of times before it can successfully be retrieved. Abstractions and rules, on the other hand, have to be learned more explicitly.

To create an abstraction and use it for later problems, information from different levels of the goal stack has to be used. The relation is determined in the analogy subgoal, while the name of the sport is stored higher in the goal stack. As a consequence, old goals created by the analogy strategy cannot be used as abstractions. An explicit goal is necessary to assemble it. An appropriate moment to do this is at the end of the analogy strategy, as illustrated in figure 6.8a. The goal is not popped, but is replaced by a goal to build an abstraction. Alternatively, the abstraction could be derived first and be subsequently applied. Since this alternative will produce the same predictions, it is not further investigated.

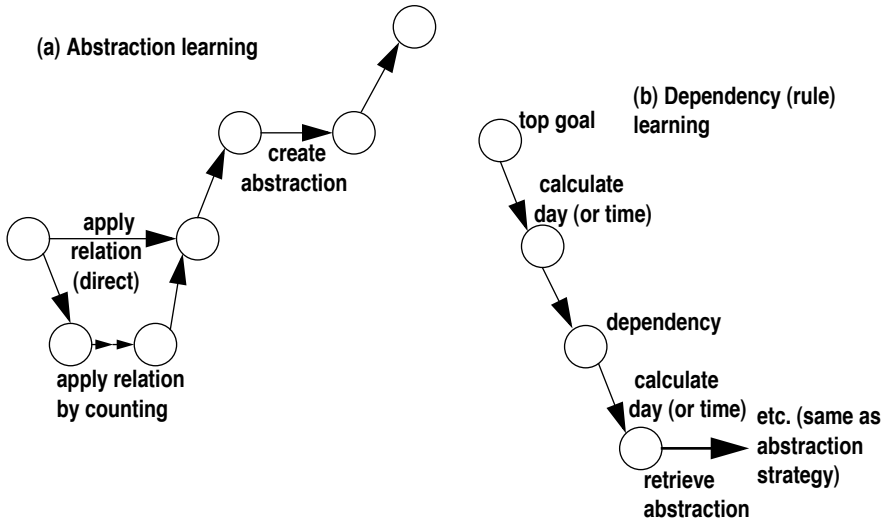


Figure 6.8. Explicit learning used by the model. (a) Learning abstractions: an additional goal in the analogy strategy (figure shows the right-hand side of figure 6.7a). (b) Learning dependencies that are compiled into production rules, as outlined in figure 6.7 (figure shows the left-hand side of figure 6.7b).

Learning a new production rule presupposes a dependency that must be created explicitly. As discussed earlier in this chapter, a dependency and a copy of the goal may be pushed as a subgoal to accomplish this (figure 6.8b). The subgoal that calculates a day or a time is replaced by a dependency. Further processing is done on a copy of the original subgoal. Assuming some other strategy has found the answer, the subgoal is popped and the dependency is completed. After the dependency has been popped from the goal stack, ACT-R's production compilation mechanism will compile the dependency into a production rule. In this particular model, pushing a dependency can only be successfully completed if it is followed by the abstraction strategy, since only the abstraction strategy can provide for the necessary constraint (for example, the appropriate plus2 fact in the hockey case). In the case of the analogy strategy, this constraint is buried deeper in the goal-structure, and cannot easily be recovered.

For both abstraction and rule learning, additional steps in the reasoning process are necessary that are irrelevant to the immediate solution. The production rule that proposes to create an additional abstraction goal has to compete with the rule that proposes to just pop the goal and be done. Similarly, the rule that proposes to replace the original goal with a dependency has to compete with rules that try to solve the problem immediately. Since the rules that propose additional processing imply

Strategy	Cost	Additional knowledge needed for each rule	Is knowledge necessary for this strategy gained implicitly?	Uses knowledge gained from
Analogy	High	None	No	Instructions
Abstraction	Medium	1 instance	No	Analogy
Rule	Low	2 rules for each direction	Officially no, but see discussion at the end of chapter	Abstraction
Instance	Very low	7-9 instances	Yes	Analogy, Abstraction or Rule

Figure 6.9. Summary of cost and learning aspects of the four strategies

additional costs, they will only occasionally win the competition. Building up abstractions and production rules may therefore be a slow process, and may well be a source of individual differences. Figure 6.9 summarizes cost and learning aspects of the four strategies.

In the Fincham task, learning of abstractions, instance learning and rule learning are all viable strategies from the viewpoint of rational analysis. Abstraction and rule learning will lead to quicker results but need more effort initially, since rules are not learned automatically. Instance learning is eventually the best strategy, but requires much more training to be fully effective.

Empirical evaluation of the model

In order to test the predictive power of the model, three experiments conducted by Anderson, Fincham and Douglass have been modeled. The first experiment was used to determine all the parameters, so the second and the third experiment can be considered as predictions based on the first. Each of the experiments tries to gain insights into the learning process by seeking evidence for the use of rules and the use of instances. The data discussed in the experiments all come from Anderson, Fincham and Douglass (Anderson & Fincham, 1994; Anderson, Fincham & Douglass, 1997), the model outputs are produced by 100 runs of our model.

Experiment 1

In the first experiment (experiment 2 in Anderson & Fincham, 1994), participants had to learn eight sports-facts. In the first three days of the experiment, four of these sports-facts were tested in a single direction: two from left to right and two from right to left. On each day 40 blocks of trials were presented. In each block, each of

the four sports-facts was tested once. On the fourth day all eight sports-facts were tested in both directions. On this day 10 blocks of trials were presented, in which each of the eight sports-facts was tested twice, once for each direction.

The model uses the following parameters: base-level decay is set to 0.3, in accordance with the findings in the Tulving-model in chapter 4, both permanent activation noise and normal activation noise are set to 0.05, the expected gain noise is set to 0.2, the retrieval threshold is set to 0.3 and both the latency factor and latency component are set to their default values of 1.0. Except for the base-level decay, all these values are close to their recommended values. Furthermore, the same parameter values will also be used for experiment 2 and 3.

Figure 6.10 shows the latencies in the first three days of the experiment, both the data from the experiment and from the model. Although the results of the model are the product of four interacting strategies, this produces no discontinuities: the learning curve of the model resembles a power-function, except for a slight decrease in performance at the beginning of each new day. The fit between the model and data is quite good: R^2=0.94. Figure 6.11 shows the results for day 4. Both in the data and in the experiment there is a clear directional asymmetry, since items in the practiced direction are solved faster than reversed items. Items that are completely new and have been practiced in neither direction, however, are performed even more slowly than the reversed items, indicating rule learning cannot be the whole explanation for all of the learning in the first three days of the experiment.

Figure 6.12 shows how the model uses the four strategies in the course of the experiment. At the start of the experiment, analogy is used most of the time, but both the abstraction and the instance strategy gain in importance after a few blocks of trials. The rule strategy appears later, and only plays a minor role during the first day. At the start of the second day, there is a large shift toward using rules at the expense of instances. This can be explained by the fact that the activation of a large portion of the instances has decayed between the two days, so that they cannot be retrieved anymore. Since only a few rules are needed for successful performance, they receive more training on average and are less susceptible to decay. Note that the abstraction strategy remains relatively stable between the days since it also less susceptible to decay than the instance strategy. This pattern is repeated at the start of the third day, although the instance strategy loses less ground due to more extended training of the examples. At the start of the fourth day, the frequency of use of the analogy strategy goes up again, since there are no production rules for the new four sports-facts. The abstraction strategy can take care of the reversed items though, so in that case the expensive analogy strategy is not needed. This explains the fact that reversed items are still faster than completely new items.

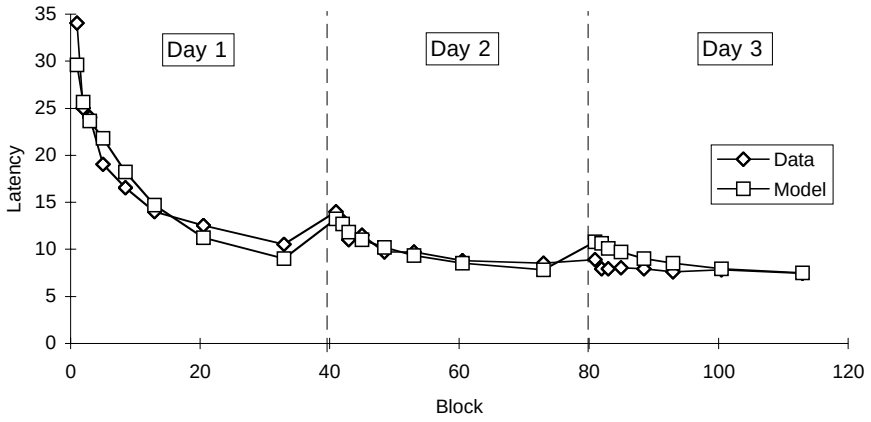


Figure 6.10. Latencies for day 1 to 3 in experiment 1

	Data	Model
Same direction, practised	8.9	8.4
Reverse direction, practised	10.9	9.3
Not practised	13	16

Figure 6.11. Effect of direction of practice and whether a rule has been trained on time to respond (in seconds) from day 4 of experiment 1.

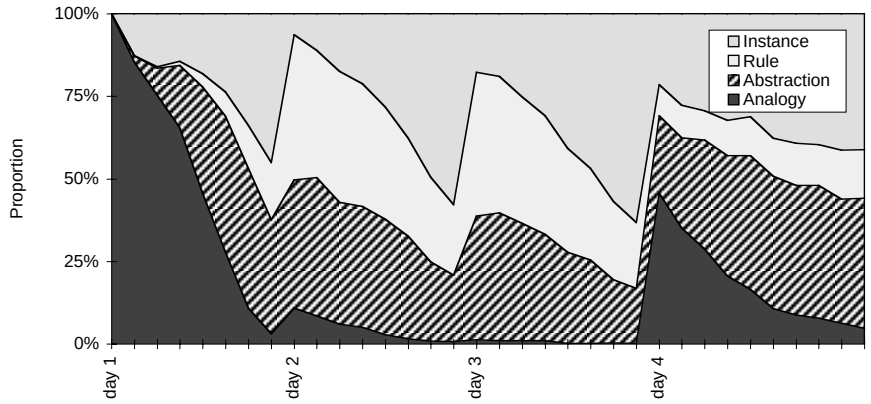


Figure 6.12. Proportion of the trials a certain strategy is used by the model in experiment 1

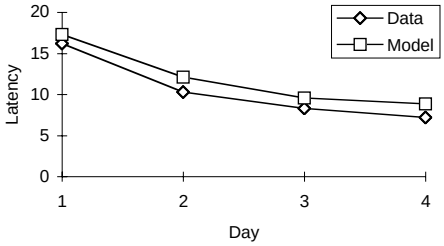


Figure 6.13. Latencies for experiment 2

Experiment 2

In experiment 2 (experiment 1 in Anderson, Fincham & Douglass, 1997) the directional asymmetry was explored further. Instead of having only a single transfer day, two rules were reversed each day of the experiment. This requires quite a complicated experiment, since on each day a rule has to be presented in two directions that was presented in one direction previously. So, on day 1 of the experiment, two out of eight rules were presented in two directions, while the remainder was only tested in one direction, on day 2 four out of eight rules, up to day 4 where all rules were presented in both directions. On each day participants had to do sixteen blocks of ten to sixteen trials, ten trials on day 1, twelve trials on day 2, fourteen trials on day 3, and sixteen trials on day 4. To further investigate the difference between rule and instance based performance, participants were asked after each trial whether they solved it using a rule or an example. Finally, on each day one of the sports-facts studied originally was offered as a trial somewhere between block 7 and 10. If performance on this original sports-fact is better than on other sports-facts, this indicates the participant retrieves the answer instead of calculating it.

The latencies for day 1 to 4 are shown in figure 6.13 for both the data and the model. Although the model is slightly slower than the participants, the learning curves are parallel. Directional asymmetries are calculated using the two rules that are presented in two directions for the first time that day. The solution time for the practised direction is subtracted from the solution time for the reversed direction. The result is the extra time needed for the reversal, and is shown in figure 6.14. Both the data and the model show a gradual increase in asymmetry over the days, although asymmetry for the model is slightly larger than for the data. To be able to map the participants' reports of using either a rule or an example onto the model, we first have to decide when the model uses a rule or an example. The most logical choice is to assume that both the analogy and the instance strategy are strategies that use examples, and that the abstraction and the rule strategy are strategies that use rules. Figure 6.15 shows the results of both the model and the data on this aspect of the task. Since the "solve by example"-category includes both the slowest (analogy) and the fastest (instance) strategy, it eventually becomes faster than the rule strategy

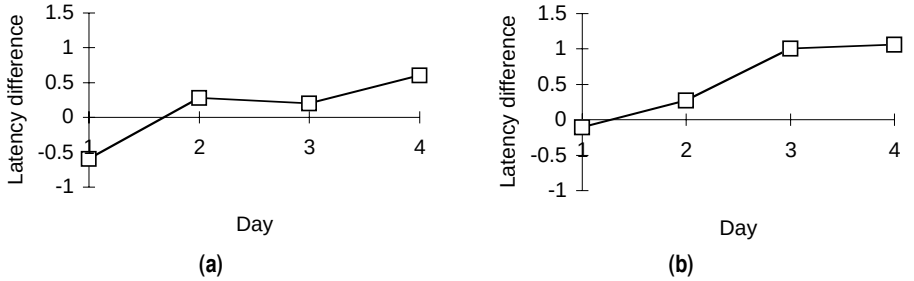


Figure 6.14. Directional asymmetry in experiment 2, (a) data (b) model

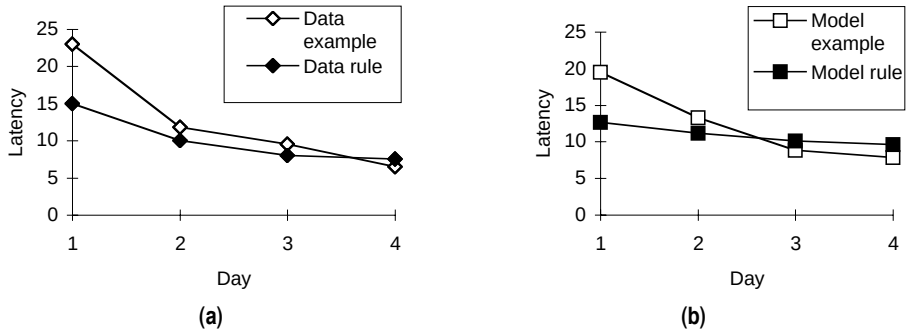


Figure 6.15. Time to respond as a function of whether a rule or example is reported, (a) data (b) model

as analogy is not used anymore. Both the data and the model show this phenomenon.

The latencies for the original sports-fact that was presented between block 7 and 10 are shown in figure 6.16, and are compared with the average latencies between block 7 to 10. Performance on original examples is clearly superior to other examples, indicating instance learning. Figure 6.17, finally, shows the strategies that were used by the model in the course of the experiment. It shows a pattern that is similar to the pattern in experiment 1.

Experiment 3

In experiment 3 (experiment 3 in Anderson, Fincham & Douglass, 1997), the effect of repeated examples is further explored. The same experimental setup as in experiment 2 was used, except that the experiment now took five days and each day consisted of 32 blocks of trials. On the first day eight rules were tested in only one direction. On each subsequent day, a new pair of rules was also tested in the reversed direction. So, on day 2 eight rules were tested in the practiced direction,

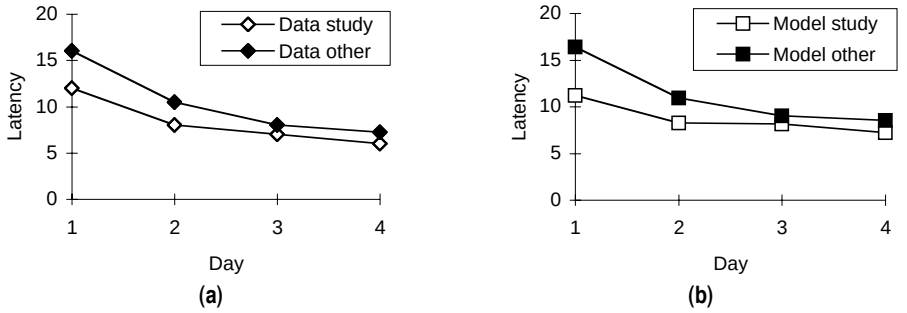


Figure 6.16. Time to respond for the studied example and other example, (a) data (b) model

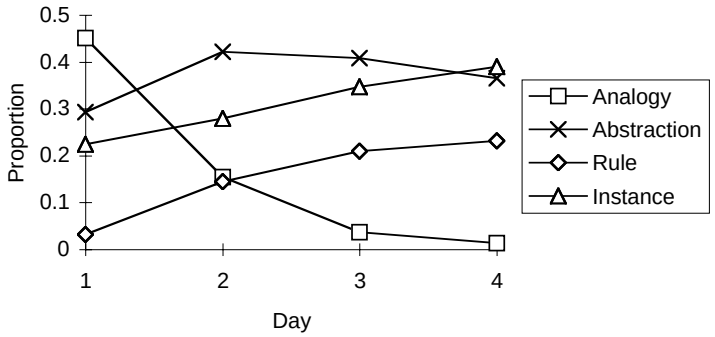


Figure 6.17. Proportion of the trials a certain strategy is used in experiment 2 by the model

and two rules in the reverse direction, on day 3 eight rules were tested in the practiced direction and four rules in the reverse direction, etcetera. To see if instances that are repeated more often than others are solved faster, half of the instances presented for a certain sport were identical, while the other half were generated in the usual way.

Figure 6.18 shows the results for both the data and the model. Repeated instances have a clear advantage over unique instances, further evidence for instance-based learning. Figure 6.19 shows the directional asymmetry results. After a steady increase between day 2 and 4, it decreases on day 5, both in the model and the data. On day 5 however, both the data and the model show a decline in asymmetry, indicating that instance-based reasoning, which has no asymmetry, takes over from rule-based reasoning.

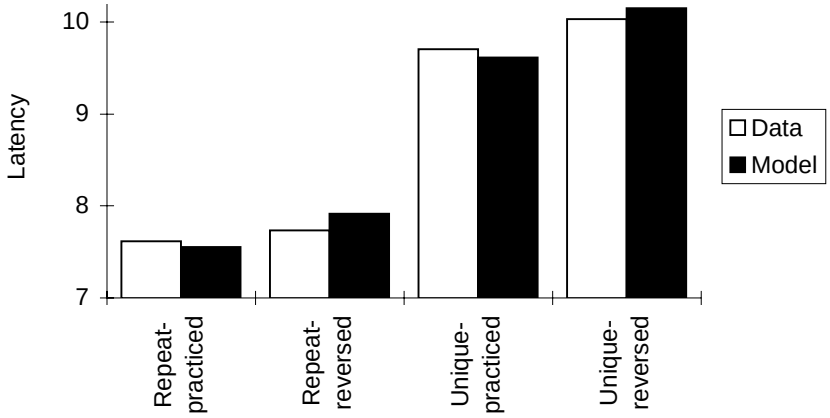


Figure 6.18. Latency (in seconds) in experiment 3 as function of condition. Only items that are reversed that day are used for these results

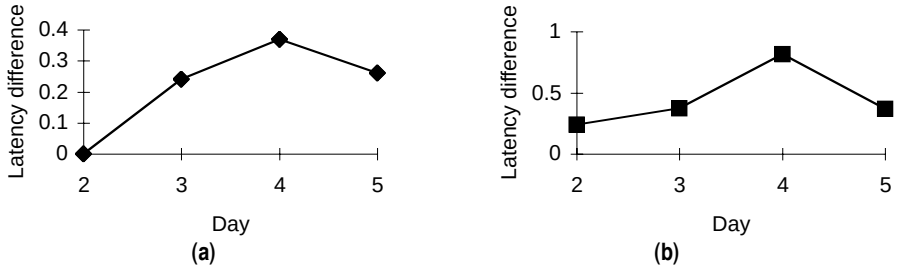


Figure 6.19. Directional asymmetry in experiment 3, (a) data (b) model

6.5 Discussion

The two models discussed in this chapter demonstrate that understanding skill acquisition is not just a matter of answering the question whether skills are represented by rules or examples. People apparently have the capacity to store previous results and the capacity to generalize rules. Whether or not both types of learning show up in the results of experiments depends on their successfulness. In the Sugar Factory experiment, rules were very hard to generalize, so behavior can be explained by learning examples only. The Fincham experiment, on the other hand, shows clear evidence for both types of learning, since there is a balance between the usefulness of learning rules and examples.

A theory that just states that skill learning is a matter of both instance learning and rule learning is rather weak, and will certainly not end the debate. That is why cognitive modeling is so useful: the theory proposed in this chapter is not just the conjunction of two existing theories, but adds the constraint that the structure of the task is a main determinant of which types of learning will have an impact on performance.

The subject of the previous chapter, implicit versus explicit learning, is also tied to the discussion of rule versus instance learning. If we consider a rule as a generalization of one or more examples, creating abstractions is the most important step of rule learning. Proceduralizing this abstraction later on is just an efficiency improvement. This brings up another issue, namely whether the proceduralization of abstractions is a form of explicit or implicit learning. Technically, it is explicit in ACT-R, since a dependency has to be pushed on the top of the goal stack, so is the focus of attention for a while. But is learning a production rule really an intentional act? This is at odds with our intuitions about production rules, especially since we have no conscious access to production rules. How can we intentionally learn things we cannot directly access?

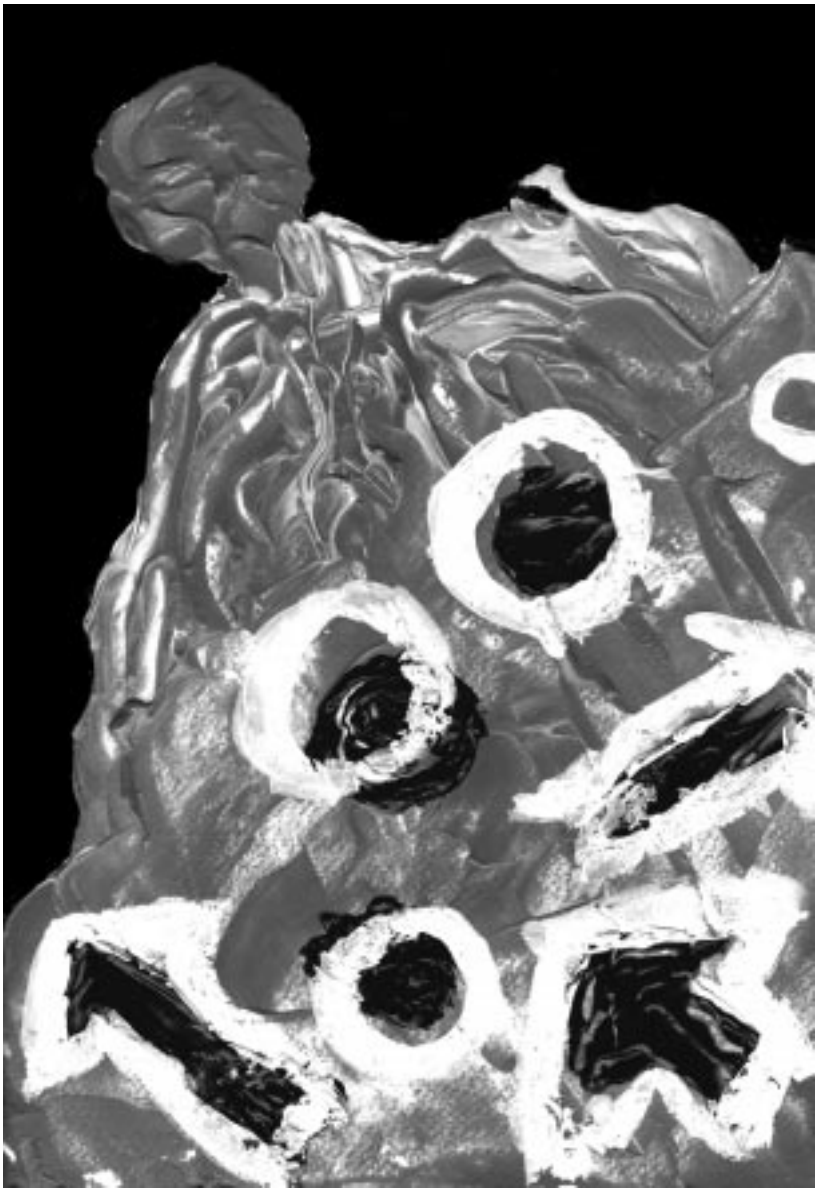
An alternative is to suppose production learning is a more or less automatic mechanism, along the lines sketched in section 6.2. The assumption that production-rule learning is an implicit learning mechanism implicates another stance towards explicit learning strategies. Instead of depicting explicit knowledge as dependency manipulators, explicit strategies are clever abstraction builders and interpreters. Although the Fincham model needs an abstraction before a production rule can be compiled, we might imagine more simple situations in which a rule can be learned without explicit declarative intervention (for example as in the child model of discrimination-shift learning in chapter 5). Eventually it may be possible to develop a learning mechanism that does not need the dependency structure at all. It must be noted that ACT-R's developers still consider production compilation as a tentative proposal (Anderson & Lebiere, 1998, pp. 109-110)

If learning a production rule is an implicit learning process, the explicit part of learning rules lies in constructing abstractions, which can be considered declarative rules. Now that rules and instances are both declarative representations in ACT-R, we might ask the question whether there really is a distinction between the two. Instances in the Sugar Factory model are used for situations that are different from the situation in which they were created, so some sort of generalization occurs at the moment an instance is applied. Abstractions in the Fincham model are used as rules, but are just declarative facts in memory. The main difference is not their representation, but the way in which they were learned, either implicitly or explicitly.

Again a traditional distinction in cognitive psychology is not what it seems when analyzed in detail within a cognitive architecture. The mapping from implicit memory to procedural memory and explicit memory to declarative memory turned out to be invalid, and now the mapping from procedural memory to rules and declarative memory to examples is not valid either. Although the concepts themselves are quite meaningful, we have to learn to live with the fact that there are no direct mappings between them and the underlying cognitive architecture. But this may just make them more interesting.

CHAPTER 7

Models of Scheduling



7.1 Introduction

When I first started thinking about a model of a problem as complicated as the scheduling problem, I didn't have a clue where to start. One option, which I quickly dismissed, was to specify a set of production rules that implemented a scheduling algorithm. Such a model might display some expert-like behavior on scheduling, but would not expose any learning. It would also contradict the ideas I started out with in chapter 1, namely that the process of learning is more interesting than the actual problem-solving behavior itself. While carrying out the projects discussed in chapters 4 though 6, however, the contours of a scheduling model started to take shape. In this chapter, I will present a model of scheduling that integrates many of the aspects of learning discussed in the last few chapters.

The starting point for the model of scheduling will be the learning paradigm presented in chapter 6 (figure 6.1). The central idea is that problem solving on a new task starts with some initial method. This method produces instances, examples of how the task is solved that can be retrieved later on. Another possible product of the initial method are abstractions, declarative representations of how the problem can be solved. Abstractions can be retrieved and applied to new cases of the problem, and during this application, production rules can be compiled. In the model of the Fincham task in chapter 6, abstractions used a representation that was specific to the task, and always involved the same kind of operation: adding or subtracting days or times. In this chapter, I will develop a generalized abstraction representation, to be used for any type of operation. This is necessary, since the strategies for the scheduling task are not fixed in advance. Another advantage of generalized abstractions is that explicit learning strategies can now operate on these abstractions. As a consequence, production rules themselves are no longer a product of explicit learning, and production compilation can be seen as a truly implicit learning mechanism.

Besides learning, there is another important aspect of the data discussed in chapter 3 that I will investigate in the model, the issue of individual differences. Individual differences have many sources. One source is differences in knowledge. If an individual does not know all the addition facts, they have to do addition by counting. This slows down the problem solving process, or may even disrupt it if working-memory capacity is exceeded. Or an individual may use a particular trick to solve a certain problem, which is not available to other individuals. I will not explore this source of individual differences in this chapter, although the reader might want to refer to the discussion of discrimination-shift learning in chapter 5 for an example. Another source of individual differences is the ability to retain elements in working memory. Although ACT-R does not model working memory explicitly, what is normally referred to as working-memory capacity is closely related to the source activation parameter in ACT-R (Lovett, Reder, & Lebiere, 1997). Source activation is the amount of activation that spreads from the chunks that are part of

the goal context to other chunks (figure 2.8). Lovett et al. have shown that varying the parameter between 0.7 and 1.4 with a mean of 1.0 can explain individual differences in the digit working memory task (Yuill, Oakhill, & Parkin, 1989). In this task, participants have to read aloud a number of strings of digits that appear on the screen. Their goal is to memorize the last digit in each string, and reproduce them after all strings have been read. Both the number and the length of the strings can be varied. Working-memory capacity is quite relevant in the scheduling task, since many aspects of the task have to be retained in memory at the same time. We will therefore look at changes in performance in the model when the source activation parameter is varied.

A final factor that has to be taken into consideration is randomness in choice. In ACT-R, noise is involved in almost any choice that is made. This means that ACT-R predicts that even if participants could be brought into exactly the same situation twice, they would not necessarily make the same choice twice.

7.2 Generalized abstractions

Abstractions in the Fincham model consist of two parts: a specification of what the goal has to be like, for example the sport is hockey and we are looking at the day of the week, and the operation that has to be performed, for example plus2. This operation has two aspects: on the one hand the plus2 operation has to be performed, involving either retrieval or subgoaling, and the answer has to be stored in the goal. The generalized version of abstractions will have the same components, but will separate out the two aspects of the operation. Furthermore, in the Fincham model abstractions relied on task-specific rules to retrieve and apply them. Generalized abstractions will need no task-specific knowledge, but are retrieved and applied by general purpose productions only. In this section, I will describe the representation and use of abstractions in general terms. A more elaborate discussion, which will take care of all the details, can be found in section 7.8.

Representation of an abstraction
The main four components of a generalized abstraction are the following:

1. The type of goal the abstraction can be used for
2. The type of fact that needs to be retrieved
3. A test that is performed on the goal and the retrieved fact
4. An action, which specifies what to do with the retrieved fact and the goal

A generalized version of a Fincham abstraction may therefore look like this:

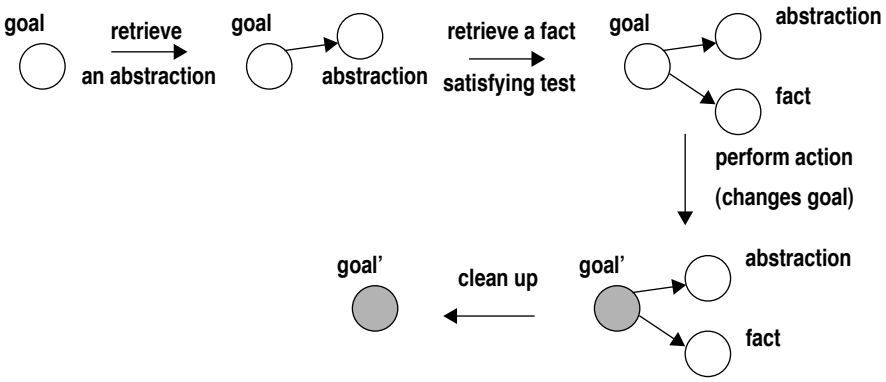


Figure 7.1. Diagram that illustrates retrieval and application of an abstraction.

```
EXAMPLE-ABSTRACTION
        ISA ABSTRACTION
        GOAL FINCHAM-HOCKEY-DAY-GOAL
        RETRIEVE "x PLUS2 equals y"
        TEST "the day of the first event must be equal x"
        ACTION "set the day of the second event to y"
```

In order to use an abstraction, it has to be interpreted by production rules. This involves a number of steps, the main ones of which are depicted in figure 7.1. The first step is to retrieve an abstraction that is applicable to the current goal. This abstraction is stored in the current goal. The second step is to retrieve a fact as specified in the abstraction, satisfying the test in the abstraction. In the example Fincham abstraction, a fact of type plus2 is needed in which the argument matches the day in the goal. Finally, the action is carried out: the retrieved fact has to be used to modify the current goal. In the Fincham example, the answer of the plus2 fact needs to be stored in the second-day slot of the goal.

This description looks conspicuously like a description of a production rule, but this is intentional. An abstraction is more or less the declarative counterpart of a production. But since it is declarative, it can be inspected, reasoned with explicitly, and manipulated. On the other hand it has to be interpreted by production rules in order to be executed. While abstractions offer flexibility, production rules offer speed: the whole cycle in figure 7.1 can be done in one step by a task-specific production rule. If both speed and flexibility are needed, both representations can be retained, but if flexibility is no longer necessary, the abstraction may be forgotten.

Using this dual representation of knowledge corresponds directly with theories about skill learning. For example Fitts (1964, cited in Anderson, 1995) discerns three stages in skill learning: a cognitive stage, an associative stage and an autonomous stage. In the cognitive stage, declarative representations (in our case abstractions),

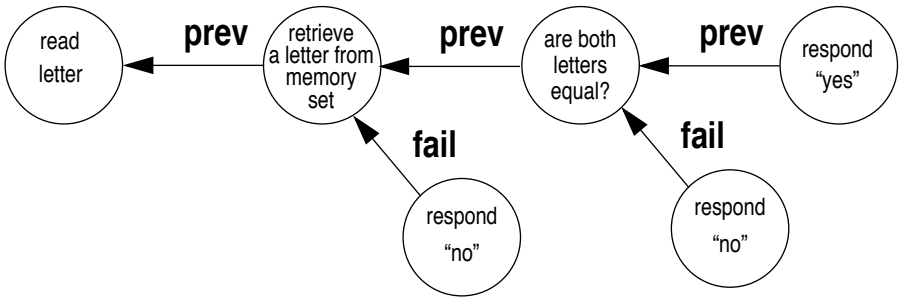


Figure 7.2. Example of chaining abstractions, in which the participant has to decide whether a letter presented on the screen is part of a previously memorized memory set. Each circle represents an abstraction.

acquired through instructions or examples, are interpreted. In the associative stage, the skill is in transition between a declarative and a procedural representation (abstractions are available, productions only partially). In the autonomous stage the skill is proceduralized completely, and sometimes the ability to verbally describe the skill is lost (all productions are learned, abstractions are forgotten). Anderson has also adapted Fitts' general skill learning theory for the ACT theories when he developed ACT* (Anderson, 1983). In the chapter about procedural learning in *The Architecture of Cognition* he already discusses the need for *general interpretive productions* in a description of how skill learning in ACT* can be accomplished. This skill acquisition aspect has, however, not been elaborated yet in terms of the ACT-R theory.

Chaining abstractions

An abstraction can be considered as a sort of plan for what to do. The example in the previous section was a simple one-step plan. But sometimes a number of steps have to be carried out in a certain order. To allow multi-abstraction plans, two extra slots have been added to the abstraction: a prev slot and a fail slot. These two slots are used to link abstractions into lists of abstractions. Each time an abstraction is completed successfully, a next abstraction is retrieved following the prev links. If an abstraction somehow fails, the next abstraction is retrieved following the fail links.

Figure 7.2 shows an example of a plan that a participant might have in a standard Sternberg memory experiment (Sternberg, 1969). In this type of experiments, the participant first has to memorize a set of letters, the memory set. Subsequently, new letters are presented to the participant, and they have to decide as quickly as possible whether or not the letter is in the memory set. Figure 7.2 shows the plan for this decision process. Each circle represents an abstraction, and the arrows show how they have been linked. In the first abstraction, the letter is read and stored in the goal. In the next step, the second abstraction from the left, a letter from the memory set is retrieved (hopefully the right one). If this already fails, a response of “no” is given,

following the "fail"-link. If a letter is retrieved, the third abstraction checks whether both letters are equal. If this succeeds, the response is "yes", else it is "no".

Proceduralizing abstractions

Since an abstraction has a function that is quite similar to a production rule, it is not so hard to proceduralize. The same method is used as discussed in the previous chapter (figure 6.2). Each time an abstraction is retrieved, there is a possibility that a dependency will be pushed as a subgoal, and the four steps in which the abstractions are carried out will be compiled into a single production rule. Due to the use of generalized abstractions, it is no longer necessary to have explicit learning strategies that are activated when a dependency goal has been pushed. The explicit strategies can now operate at the level of abstractions, independently of the production-compilation process. By pulling explicit learning strategies out of the dependency subgoal, the actual process of building dependencies can be carried out by a fixed set of production rules, more or less as a mechanism of the architecture.

7.3 A first model

A first approximation of a model of scheduling has the following components:

1. Production rules that interpret and proceduralize abstractions as outlined in the previous section
2. A top-goal that reads the constraints for the current problem from the screen and pushes a task subgoal upon the goal stack. After the subgoal has successfully terminated, it outputs the answer found in the subgoal.
3. Productions that store elements in a list, and implement rehearsal, both maintenance rehearsal and elaborate rehearsal.
4. A set of abstractions that implements a simple strategy for scheduling.
5. Productions that produce some sort of verbal protocol.

The first item on the list has already been discussed, and the top-goal productions are quite trivial, so I will only elaborate on the last three items of the list.

Storing elements in a list and doing rehearsal

In chapter 4, I discussed a model of rehearsal based on Baddeley's phonological loop. As we have seen in the protocols of scheduling, participants maintain a list of the partial solution, which they rehearse from time to time. Rehearsal can have two functions: maintenance rehearsal to keep the activation of the list high enough, and elaborate rehearsal to do additional processing on the items in the list. The first model will use elaborate rehearsal to calculate the total duration of the tasks in the

list. Instead of using an explicit phonological loop, as in chapter 4, ordinary ACT-R chunks are used to represent the list. Details of the implementation can be found in section 7.8.

Abstractions that implement a simple strategy

The first model uses a simple, one shot strategy that involves the following steps:

1. A first task for the schedule is selected by retrieving an order-constraint and picking the first task in this constraint. For example, if 'D before C' is a constraint, D is picked as a possible first task. It is then checked if there is no earlier task, indicated by a constraint like 'A before D'. If that is the case, the earlier task is substituted as candidate first task, else D is accepted as first task.
2. The next task is determined by finding an order constraint that specifies a fact that is later than the task we have just added to the schedule. So if the schedule starts with D, and 'D before C' is a constraint, we add C. Repeat this step until no more tasks can be added using this method.
3. Now count how many hours the tasks in the current schedule take (using elaborate rehearsal, as explained above).
4. Calculate how many hours are left for one worker. So, if the tasks currently in the schedule take four hours, and each worker has six hours, two hours are left. If the number of hours left is greater than zero, find a task that has a duration of exactly that number of hours and add it to the schedule.
5. Move to the next worker.
6. Go through the list of all the tasks, and add those to the schedule that are not already allocated to the first worker.

Verbal protocol

An assumption about abstractions is that they can be reasoned about, so they are available to verbalization in a think-aloud experiment. To avoid writing a language-production model, a "verbalization" string is added to each abstraction that describes the action performed by the expression. Whenever an abstraction is executed, this string is added to the verbal protocol. Rehearsal actions also produce verbal protocol, as do reading actions. The verbal protocol not only enables producing "Turing Test"-like results, but is also very useful in debugging the model. Although a fully-fledged language production module will probably require a formidable modeling effort, it may be a very useful tool in a continued research effort on declarative rules.

Results of the model

The model was tested using a set of ten example problems, all of which consisted of two workers and six or seven tasks. Although the problems are not particularly hard, this is not yet important since the answer given by the model is not checked.

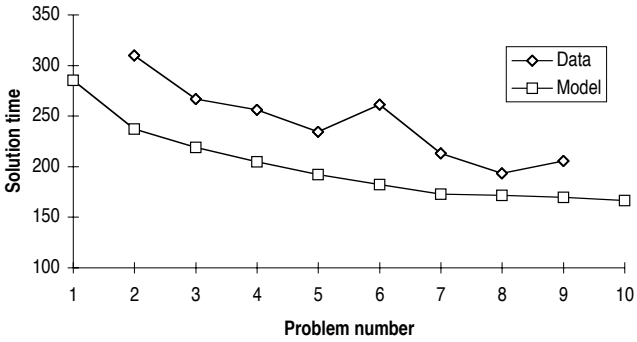


Figure 7.3. Learning curve of the first model. The data are plotted on an idealized curve based on the data discussed in chapter 3.

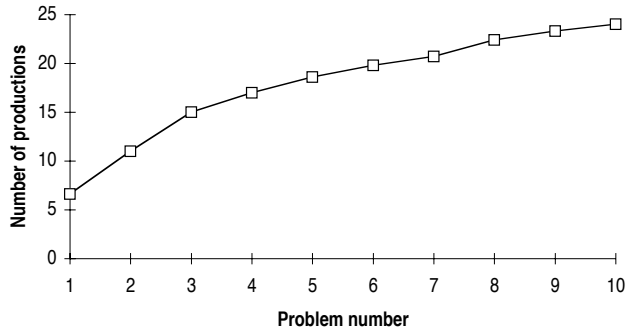


Figure 7.4. Growth in the number of production rules

The model uses only symbolic learning, and has all subsymbolic learning turned off. New chunks in declarative memory do not have a role in the problem solving process yet. Improvements in performance can therefore be attributed to production compilation. Figure 7.3 shows the learning curve of the model. The graph also shows the data from chapter 3 in comparison (actually the lower-left panel of figure 3.4 multiplied by the average solution time; the data start at problem 2, because participants have already solved one two-worker example problem). Although the data from the model and the experiment cannot be compared properly because different problems have been used, the graph shows the same logarithmic curve for both the model and the data. To get some idea of the rate of learning, the growth in the number of productions is plotted in figure 7.4. The more interesting part is the pseudo verbal protocol produced by the model. To see the impact of proceduralization, examples of the output of the first and the tenth problem have been printed in figure 7.5. Clearly, the protocol of the first problem is a protocol analyst's dream, because participants are hardly ever that precise. But the tenth protocol looks more familiar: many steps in the process are omitted, and we

Protocol of first problem

There are two workers. Each of the workers has seven hours. Task A takes two hours. Task B takes two hours. Task C takes two hours. Task D takes two hours. Task E takes three hours. Task F takes three hours. Task B before F. Task F before A. First I will find a task to begin with. Let's look at an order constraint. B before F. Let's see if there is no earlier task. There is no earlier task. Begin with B. B Can we find a next task just by looking at the order? B before F. B.. F.. Can we find a next task just by looking at the order? F before A. B.. F.. A.. Can we find a next task just by looking at the order? Is this a schedule for one worker or for more? Now I am going to count how many hours we already have B.. How long does this one take? Task B takes two hours. Add this to what we have. nothing plus two equals two. F.. How long does this one take? Task F takes three hours. Add this to what we have. Two plus three equals five. A.. How long does this one take? Task A takes two hours. Do we have enough for one worker? Each worker has seven hours. We can move to the next worker.. B.. F.. A.. next.. Let's do the rest Now we are going to look at all the tasks, and see which ones are not yet in the schedule. Let's start with A. Task A takes two hours. Let's try to put it in the schedule. A is already in the schedule. OK, what is the next letter? B comes after A. Task B takes two hours. Let's try to put it in the schedule. B comes after A. B is already in the schedule. OK, what is the next letter? C comes after B. Task C takes two hours. Let's try to put it in the schedule. C comes after B. B.. F.. A.. next.. C.. OK, what is the next letter? D comes after C. Task D takes two hours. Let's try to put it in the schedule. D comes after C. B.. F.. A.. next.. C.. D.. OK, what is the next letter? E comes after D. Task E takes three hours. Let's try to put it in the schedule. E comes after D. B.. F.. A.. next.. C.. D.. E. Task F takes three hours. Let's try to put it in the schedule. F comes after E. F is already in the schedule. OK, that was the last task, we're done! The answer is B F A next C D E

Protocol of tenth problem:

There are two workers. Each of the workers has six hours. Task A takes one hours. Task B takes one hours. Task C takes two hours. Task D takes two hours. Task E takes three hours. Task F takes three hours. Task D before E. Task E before A. First I will find a task to begin with. Let's see if there is no earlier task. Begin with D. D.. D.. E.. D.. E.. A.. Can we find a next task just by looking at the order? Is this a schedule for one worker or for more? Now I am going to count how many hours we already have D..E..A.. D..E..A.. next Now we are going to look at all the tasks, and see which ones are not yet in the schedule. Let's start with A. A is already in the schedule. D..E..A.. next.. B D..E..A..next..B..C D is already in the schedule. E is already in the schedule. D..E..A.. next.. B..C..F. OK, that was the last task, we're done! The answer is D E A next B C F

Figure 7.5. ACT-R protocol of the first and the tenth problem of a sample run

can only guess why some decisions have been made. This concurs with the general idea that proceduralized skills produce no verbal protocol (Ericsson & Simon, 1984; van Someren, Barnard & Sandberg, 1994).

Although this first model shows some interesting properties similar to real problem-solving behavior, it is far from complete. The current model just takes a single shot at the solution, and does not retry if it is incorrect. Only production compilation had been turned on, so the model will never forget any intermediate results it has found. And finally, the model starts out with a set of task-specific abstractions. One of the desired properties of the model was to start without any task-specific knowledge. These issues will be addressed in the second version of the model. But first the most important of these issues will be discussed separately: where do abstractions themselves come from?

7.4 Learning new abstractions

Figure 6.1 specifies the first step of learning a new skill as the 'initial method'. In the Fincham task the initial method was analogy, because this method had been explained as part of the experiment. In general, analogy is a strategy that takes knowledge from another domain, and modifies this knowledge to suit the current task.

Work by Sander and Richard (1997) indicates that people use the analogy strategy in discovering knowledge for a new domain. In their experiment, participants without any computer experience had to learn to operate a word processor. Since participants did not get any instructions on how to operate the word processor, they had discover the functions by themselves. The tasks participants had to do was to modify a given text so that it would be identical to another text. The total set of possible operations to change text was classified into four levels. Level 1 consists of operations that are also possible on a standard typewriter, a device participants were all familiar with, for example typing new letters, and deleting the last character typed. Level 2 operations are operations that are not possible on a typewriter, but can be considered as normal in the domain of writing, for example inserting a word in a sentence. Level 3 operations come from the even more general domain of object manipulation, for example copying a word and pasting it somewhere else. Finally, level 4 operations are operations not related to any domain. An example of a level 4 operation is to copy strings of spaces. A space is not an object in the real world, so the specific knowledge that a space in a word processor is like any other character is required.

In the experiment participants were strongly encouraged to discover new methods, since each time they tried a method they used before, they were prompted to attempt another method to solve that particular problem. As it turned out, all participants used level 1 operations immediately from the start of the experiment. As the experiment progressed, they gradually discovered level 2 operations, followed by level 3 operations. Level 4 operations were only discovered by a minority of the participants, and only in the last few sessions of the experiment.

The results of this experiment support the idea that when people are in a new situation, they adapt knowledge from a similar domain to initially guide their actions. In word processing, knowledge of a typewriter is the most immediate source. If that source of knowledge is exhausted, knowledge of writing in general can be used, followed by the even more general knowledge source of object manipulation.

In the scheduling task, analogy is also a good starting point. People may not know anything about schedules, but they do know something about lists, and how to construct them. Suppose we need to make a schedule. We may use knowledge about

lists to start with. How do we make a list? First we have to find a first item for the list, a beginning. Once we have a beginning, we find a next task until we are done. But how do we find something to begin with, and how do we find a next task? We may choose to handle these problems by making them subgoals, or we may try to find mappings between 'beginning' and 'next' and terms in the scheduling problem. For example, a mapping can be made between 'next' and an order-constraint in the scheduling problem. The result is a modified version of the list-building abstractions, with 'list' substituted by 'schedule' and 'next' substituted by 'order'. Note that for sake of the explanation, the terms 'list', 'beginning', 'before' and 'next' will be used to refer to general terms, and 'schedule' and 'order' to refer to task-specific terms. Except for knowledge on how to build a list, the analogy between a schedule and a list may also offer knowledge on how to retain a list in memory by rehearsal.

Although these new abstractions may find a start for a schedule, they are not sufficient to build a complete schedule, mainly because the mapping between 'next' and 'order' is inadequate. When this abstraction fails to make a complete schedule, another plan may take over and contribute to the schedule.

An idea that may take over if the list-building plan fails to add any more tasks to the schedule is the plan that tries to complete the first worker. A useful general plan may state that whenever something has to be completed, the difference between the desired size and the current size has to be calculated, after which an object has to be found with a size equal to this difference.

The central emerging idea is therefore that several strategies from similar domains are adopted and patched together. This method of adapting old strategies to new purposes is similar to the *script* and *schema* theories, as proposed by Schank (Schank & Abelson, 1977). Traditional script and schema theories assume that a complete script is first adapted to fit the current task, and then carried out. The ACT-R model uses a more on-demand style of adaptation: a new abstraction is created at the moment it is needed. Again, the details may be found in section 7.8.

7.5 The second model

The second model solves some of the shortcomings of the first. It learns new abstractions as outlined in the previous section. Furthermore, the following aspects have been added to the model:

1. After a solution has been produced by the model, it receives feedback from the interface. If the solution is incorrect, the model has the opportunity of reading the violated constraint, and has to attempt a new solution. If no solution has been found after 300 seconds, the model has to move on to the next problem.

This 300 second boundary is somewhat arbitrary, but gives an opportunity to assess the accuracy of the model: if the model cannot solve it within the allotted time, it is counted as a failure.

2. Base-level learning is turned on. As a consequence, the model can forget all kinds of partial results it derives, most notably the list that contains the partial solution, but also read constraints (which have to be reread in that case), newly derived abstractions, etc. To recover gracefully from all kinds of errors that can occur due to forgetting, the robustness of the model had to be increased. The result of an error is often that a subgoal is popped in failure. This means that if a goal pushes a subgoal, it sometimes has to check whether or not this subgoal has actually succeeded. This is especially important if production compilation is involved, since this may result in learning a faulty production rule that gets ACT-R into endless loops. A base-level decay of 0.5 is used, the recommended value, from which I diverged in the Tulving and Fincham model. No long-term effects of learning were investigated in this model, so there was no need for a smaller decay.
3. The model uses the order in which constraints are presented on the screen. For example, if a task has to be found that takes 3 hours and is not yet present in the current schedule, the list on the screen is used to find the first task taking 3 hours. If that task is already in the schedule, the next 3 hour task is looked for on the screen, etc.
4. Several extra abstractions have been added to ensure that correct solutions are eventually found by the model. The model now tries to satisfy the order constraints for the second worker as well, and uses the feedback it gets when it makes an error as a starting constraint for the next try.

Example verbal protocol

The following protocol fragment, produced by the model, gives an impression of the additional aspects of the model:

> There are two workers. Each of the workers has six hours. Task A takes one hours. Task B takes one hours. Task C takes two hours. Task D takes two hours. Task E takes three hours. Task F takes three hours. Task B before C. Task F before A. I have to think of some new way to find a schedule. Let's use what I know about lists. First I will find something to begin with. Let's look at a before constraint. I have to think of some new way to find a before. Let's use what I know about order. Let's use a order fact as a before fact. F before A. I have to think of some new way to find a before following a fail-abs12. F before A. I have to think of some new way to find a before following abstraction12. Let's look at a before constraint. Let's see if there is no earlier element. Let's use a order fact as a before fact. There is no earlier element.
>
> This doesn't work. Let's start again. First I will find something to begin with. Let's look at a before constraint. Let's see if there is no earlier element. Let's use a order fact as a before fact. There is no earlier element. Begin with F. F.. I have to think of some new way to find a schedule following abstraction10. Now I have to find the next thing. F before A. A.. I have to think of some new way to find a schedule following abstraction17. Now I have to find the next thing. No more items for the list, let's check whether we're done. F.. A.. Is this a schedule for one worker or for more? Now I am going

to count how many hours we already have F.. How long does this one take? Task F takes three hours. Add this to what we have. nothing plus three equals three. A.. Add this to what we have. Three plus one equals four. Do we have enough for one worker? No, the schedule is not full, yet. I have to think of some new way to find a total. Let's use what I know about hours. Let's use a hours fact as a total fact. Each worker has six hours. I have to think of some new way to find a total following fail-abs23. Each worker has six hours. I have to think of some new way to find a find-remain following a failure. This doesn't work. Let's start again.

First I will find something to begin with. Let's look at a before constraint. Let's see if there is no earlier element. Let's use a order fact as a before fact. Let's see if there is no earlier element. Let's use a order fact as a before fact. There is no earlier element. Begin with F. F.. Now I have to find the next thing. F before A. A.. Now I have to find the next thing. Now I have to find the next thing. No more items for the list, let's check whether we're done. F.. A.. Is this a schedule for one worker or for more? Now I am going to count how many hours we already have F.. Add this to what we have. Nothing plus three equals three. A.. Add this to what we have. Three plus one equals four. Do we have enough for one worker? No, the schedule is not full, yet. Let's use a hours fact as a total fact. Each worker has six hours. Each worker has six hours. How many hours are there left? Two plus four equals six. F.. A.. Now find the task that fits in. Task C takes two hours. C.. We can move to the next worker.. NEXT-WORKER Let's do the rest.. F.. A.. C.. NEXT-WORKER.. I now try to find any unused order constraints. B before C. This one hasn't been used, so the constraint has been found. B.. Now we are going to look at all the tasks, and see which ones are not yet in the schedule. Let's start with A. A is already in the schedule. B is already in the schedule. Let's move on to the next task. C is already in the schedule. Task D takes two hours. D.. Task E takes three hours. E.. Task F takes three hours. OK, what is the next task? OK, that was the last task, we're done! This doesn't work. Let's start again.

[one more failed search episode]

First I will find something to begin with. Begin with F. F.. Now I have to find the next thing. F before A. A.. Now I have to find the next thing. No more items for the list, let's check whether we're done. F.. A.. Is this a schedule for one worker or for more? Now I am going to count how many hours we already have F.. Add this to what we have. Nothing plus three equals three. A.. Add this to what we have. Three plus one equals four. Do we have enough for one worker? No, the schedule is not full, yet. F.. A.. Now find the task that fits in. Task C takes two hours. C.. We can move to the next worker.. NEXT-WORKER Let's do the rest F.. A.. C.. NEXT-WORKER.. I now try to find any unused order constraints. B before C. B before C. This one hasn't been used, so the constraint has been found. B before C. B before C. B.. Now we are going to look at all the tasks, and see which ones are not yet in the schedule. Let's start with A. Task A takes one hours. A is already in the schedule. OK, what is the next task? Task B takes one hours. B is already in the schedule. Let's move on to the next task. OK, what is the next task? Task C takes two hours. C is already in the schedule. Let's move on to the next task. OK, what is the next task? Task D takes two hours. D.. Let's move on to the next task. OK, what is the next task? Task E takes three hours. E.. Let's move on to the next task. OK, what is the next task? Task F takes three hours. F is already in the schedule. Let's move on to the next task. OK, what is the next task? OK, that was the last task, we're done! F.. A.. C.. NEXT-WORKER.. B.. D.. E.. The answer is F A C NEXT-WORKER B D E

The particular fragment contains five search episodes, only four of which are shown: the first three and the final, successful episode. In the first two fragments, the model is busy figuring out how aspects of the problem can be mapped onto things it knows something about. Unfortunately, the primitive protocol generating part of the model produces some awkward sentences with references to internal symbols. Somewhere along the line the model gets stuck, because it can not keep track of all the constraints

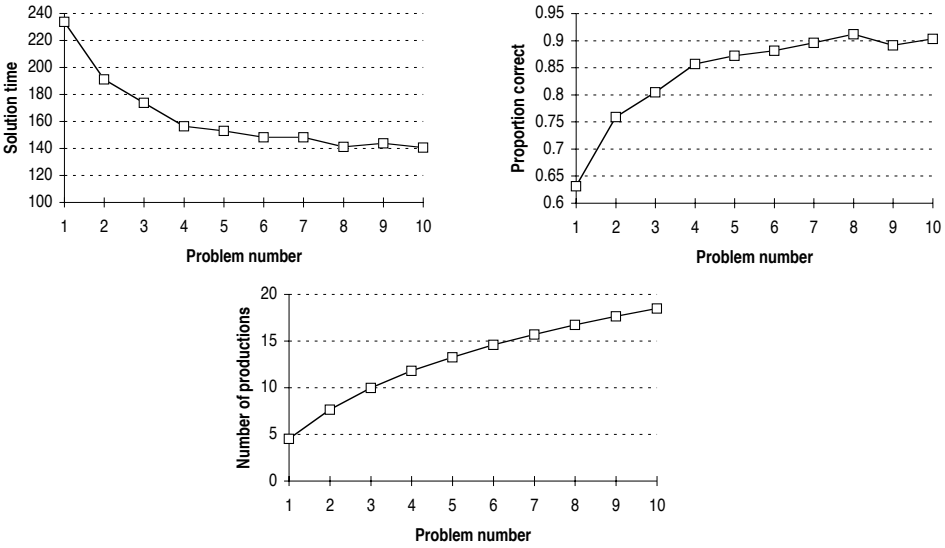


Figure 7.6. Solution times (top left), proportion solved (top right) and number of learned production rules (bottom) for the second model

in the task and all the newly derived abstractions. The third search episode is slightly more successful: it can use the abstractions derived in the first two episodes. Unfortunately, it fails just when it is done, because it cannot retrieve the start of the list anymore for typing in the answer. In the fifth, successful episode some of the earlier derived results can be retrieved. For example, the model immediately starts with "Begin with F" instead of deriving this fact.

The separate search episodes are similar to the episodes participants showed in the experiment (see chapter 3). Once the model gets stuck, it often does not have knowledge to repair the situation other than starting again. In the new search episode knowledge derived in the earlier episode is sometimes retrieved, so failed episodes do contribute to eventual success. This concurs with the behavior of participants, since they also are hardly ever able to recover from an error in their reasoning process.

Results of the model

Figure 7.6 shows the basic, averaged, results of the model. The solution times and the number of learned production rules are similar to the results of the first model. The improvement in solution time is accompanied by an improvement in the proportion of the problems that is solved correctly within 300 seconds, which I will refer to as 'accuracy' in the rest of the chapter.

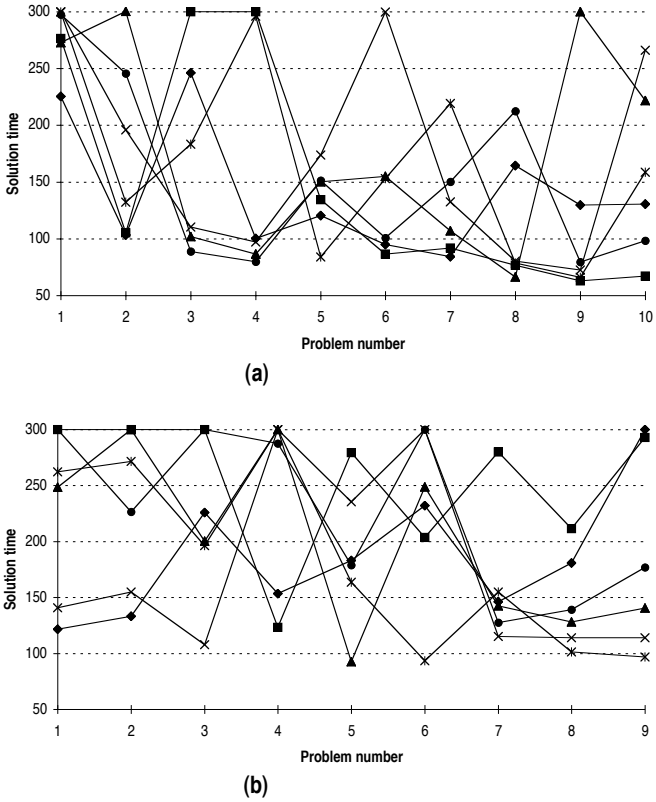


Figure 7.7. Examples of individual data: (a) six runs of the model, (b) six subjects from the experiment.

Individual differences

Figure 7.6 shows a gradually improving learning curve. But as we have seen in chapter 5, the averaging process may smooth out discontinuities that may be present in the data of individuals. Figure 7.7 shows solution times of six individual runs of the model, and the results of six participants from the experiment in chapter 3. Neither the model nor the data shows a smooth improvement of performance, only after averaging results is such a result obtained. Again, the comparison between model and data is only an approximation, since different problems were used.

As mentioned in the introduction of this chapter, the source activation (W) parameter is associated with individual differences in working-memory capacity. Since the scheduling task requires participants to keep many aspects of the task in memory at the same time, it should be quite sensitive to changes in this parameter. Working-memory capacity, as modeled by source activation in ACT-R, is not a

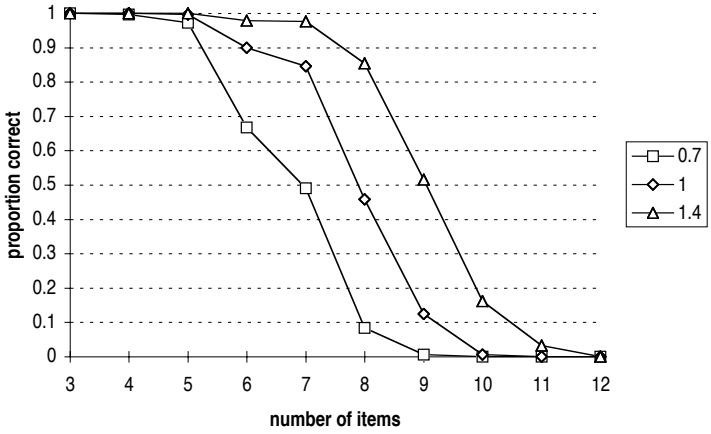


Figure 7.8. The proportion of items retrieved correctly decreases non-linearly with the number of items currently relevant. Three values for the W parameter are shown: low (0.7), average (1) and high (1.4)

buffer of limited size, but rather the capacity to increase activation of currently relevant memory chunks. As the number of relevant chunks increases, the potential for errors increases. Sometimes it is possible to recover from an error, by recalculating the lost fact, but sometimes the information is lost. The number of errors is a non-linear function of the number of currently relevant chunks. Figure 7.8 illustrates this aspect: it shows the results of a small ACT-R model that stores between three and twelve items and then tries to retrieve them. The graph shows the proportion of correct retrievals for three different values of W. The model is allowed a single rehearsal for each item. As can be seen in the graph, at some point the probability of correct retrieval decreases dramatically. For the average W=1 case, this point is around the "magical number seven", and the low and high W cases roughly represent "plus or minus two." This decrease in performance is not caused by the fact that some activation resource must be distributed over a number of chunks "in working memory", but is rather an emergent fact of several aspects of processing at the same time. The real limited resource is time: as more chunks are relevant, less time can be spent on each of them individually. A higher source activation just makes it possible to retrieve chunks that were accessed longer ago. The result is a model of a limited capacity without resources.

The following metaphor may clarify this issue. Suppose you are a baby-sitter and you look after number of small children. To prevent children from getting up to mischief, you have to pay attention to them. You can only pay attention to one child at a time. As long as a child has had your attention not too long ago, it will behave properly. But if ignore a child too long, it will start misbehaving. If you only have a few children too look after, you will have no problems. Any mischief can be corrected easily by giving a little more attention to the particular rascal. But as the

number of children rises, giving more attention to one child means neglecting the others, causing more and more trouble. So, at a certain number of children, it becomes almost impossible to keep them all happy. Now if you are a particular good baby-sitter, you can give them impressive talkings-to, so they will keep from mischief just a little longer. Or, if you know the particular children, you might know a few tricks to keep a particular child happy.

In the baby-sitter example, baby-sitters have no particular hard limit or capacity of children they can keep happy. Neither do they spread attention to all children at the same time. They just go from child to child and hope for the best. Individual differences between baby-sitters are reflected by the impact their attention has on the children: better baby-sitters don't have more time, they just use it more effectively.

To see the impact of the W parameter, the model was run several times with source activations ranging from 0.7 to 1.4, the range that covers all subjects in the Lovett et al. experiment. Figure 7.9 shows that source activation has indeed a high impact on performance. A low source activation implies longer solution times and a lower accuracy. The interesting thing about the accuracy is, however, that the differences are initially very large: for the first problem, the accuracy of the high source activations is almost perfect, as opposed to the very poor accuracies for low source activations. But as learning progresses, these poor accuracies improve dramatically and by the tenth problem are almost as good as the higher source activations. This corresponds well with the experiment, in which almost every participant eventually managed to solve the problems, although the time they needed to do this (so the number of opportunities for learning), differed tremendously. A tentative conclusion of this model may therefore be that practice eventually overcomes poor working-memory capacity.

Is proceduralization necessary for mastering complex skills?
In chapter 1, the hypothesis was posed that mastering a complex skill is a gradual process, in which some cases of a problem can be solved directly, some need additional search, and some cannot be solved due to the fact that this would take too much time. In the scheduling model, a similar issue turns up: if part of a skill is not proceduralized, it puts extra demands on working-memory capacity, and limits the amount of other non-proceduralized activity. As a consequence, as working-memory capacity is lower, more proceduralization (i.e., practice) is needed before a task can be performed successfully. Working-memory capacity more or less defines how broad the small grey band in figure 1.4 is. The results in figure 7.9 show that the accuracy for the higher source activations is close to 1 for the very first problem. If source activation is lower, practice is needed before a high accuracy is reached.

Figure 7.10 shows a graphical impression of the consequences of limited working-memory capacity for the scheduling task, analogous to figure 1.4. The rectangle

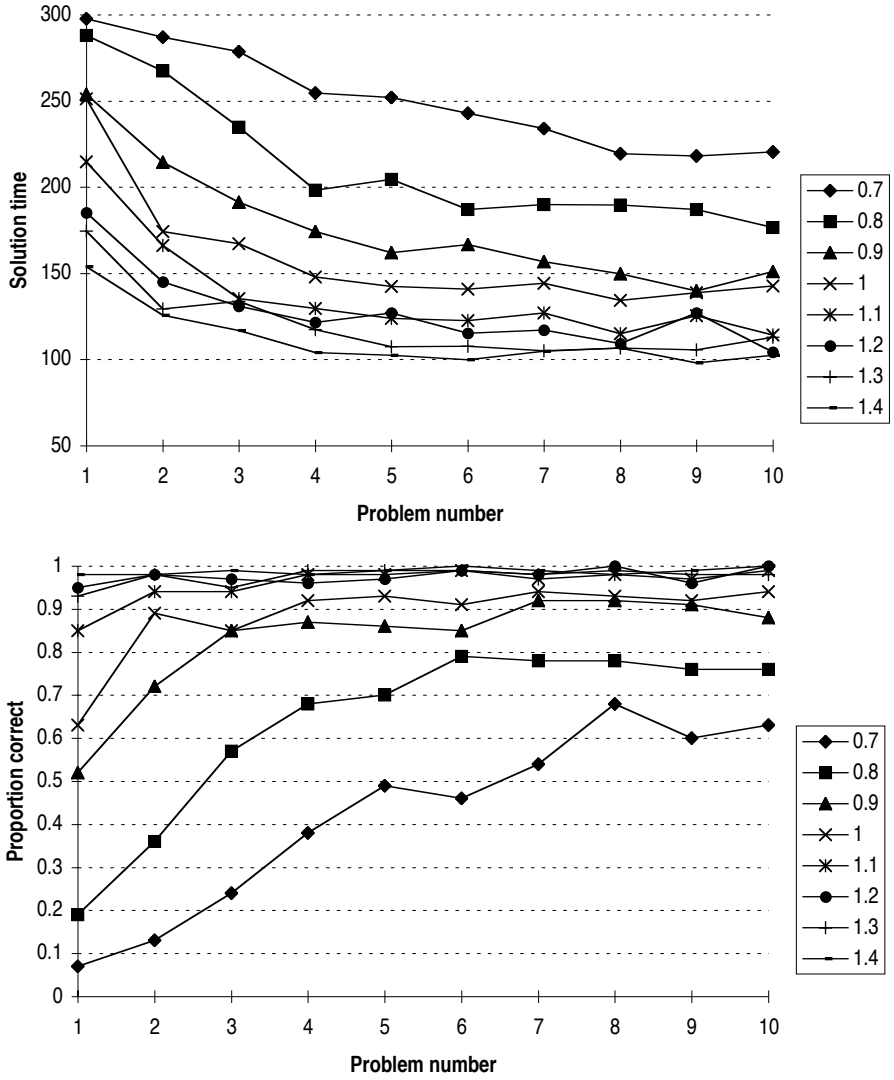


Figure 7.9. Solution time and proportion solved correctly for source activations ranging from 0.7 to 1.4

represents all skills involved in scheduling. At the bottom of the rectangle is a black region which represents the some basic skills that even novices in scheduling already possess, such as reading the screen, building lists and doing rehearsal. Using these skills does not require any extra working-memory capacity. Skills in the light grey area do require working-memory capacity. In terms of the model, these skills use chunks that represent the list, but also abstractions that are used in the reasoning

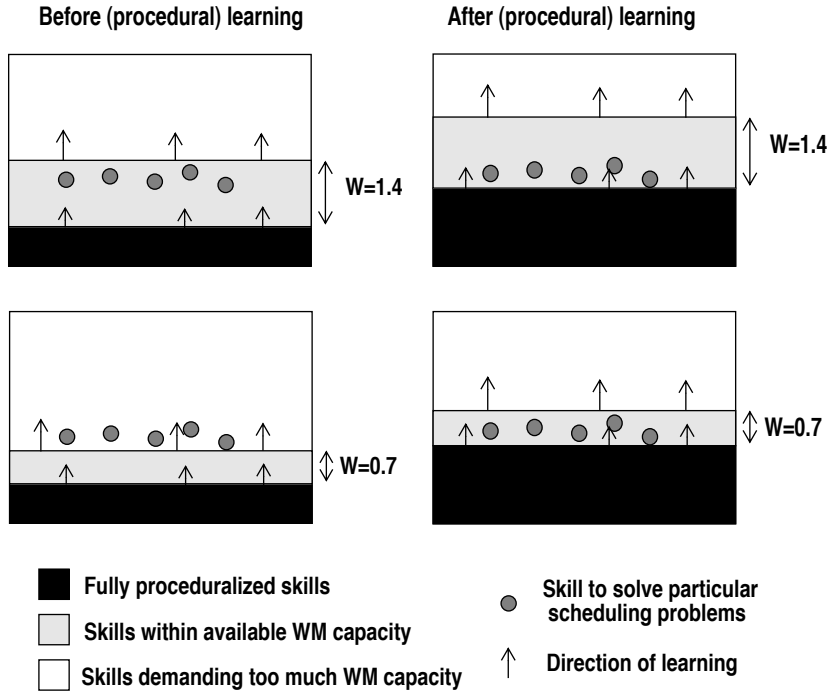


Figure 7.10. Graphical impression of the role of source activation in solving scheduling problems

process. The white area represents skills, or groups of skills that take too much working-memory capacity. The dark grey circles, finally, represent the skills of doing the scheduling problems the model has to solve. The top part of the figure shows a case of high working-memory capacity (W=1.4). The scheduling problems are already within the grey area, so can immediately be solved by the model. The only advantage of procedural learning is that the solution time decreases. When source activation is low, on the other hand, the skill of solving the scheduling problems is still in the white area, as shown in the bottom-left part of the figure (W=0.7). In order to be able to solve the problem at all, procedural learning is necessary to reach to get the dark grey circles within the grey band.

To examine more closely whether this is the case in the model, a comparison is made between runs with production compilation turned on and turned off. Figure 7.11 shows the results for source activations 0.6, 0.65, 0.7 and 1.4. Clearly for the lower source activations, production learning is essential for successfully mastering the skill. For W=1.4, on the other hand, procedural learning does not contribute to accuracy at all.

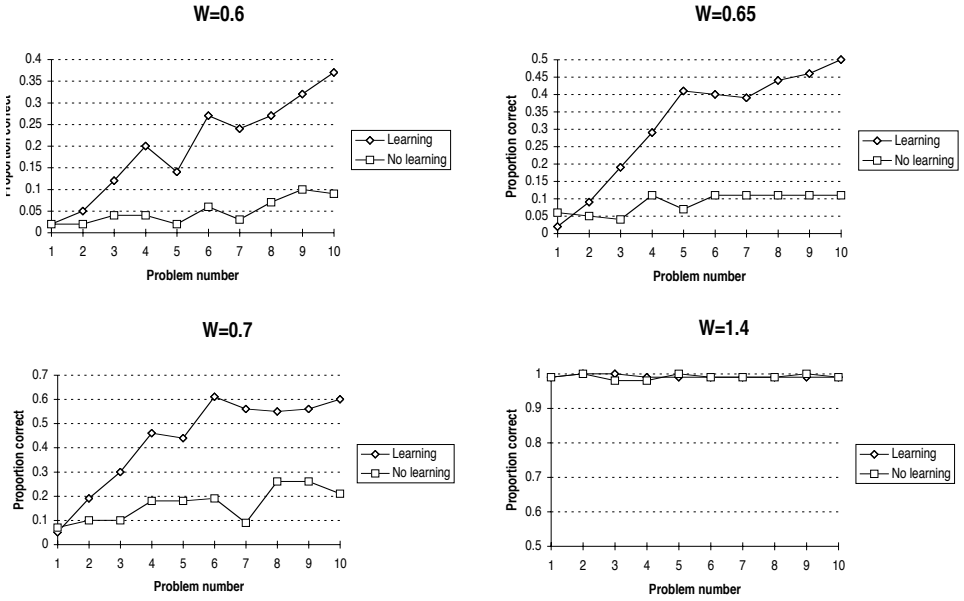


Figure 7.11. Comparison between procedural learning turned on and off, for different values of source activation

7.6 Some empirical evidence for the scheduling model

Although the models of scheduling presented in this chapter address most of the issues raised in chapter 3, the predictions made by the model have not actually been tested yet. Fortunately, Linda Jongman has recently performed an experiment that provides some experimental support for the model. In a study on mental fatigue, she used the scheduling task as discussed in chapter 3, and the digit working memory task that has been modeled in ACT-R by Lovett, Reder and Lebiere (1997).

The digit working memory task was used to make an estimate of the working-memory capacity of a participant, expressed in the ACT-R source activation parameter. This working-memory capacity was related to the performance on the scheduling task. Unfortunately, the scheduling task as it was used in this particular experiment was a mixture of problems with two and three workers with varying difficulty and varying time limitations. It is therefore hard to compare the results directly to the model predictions. Nevertheless some of the more qualitative predictions of the model can be tested with respect to individual differences.

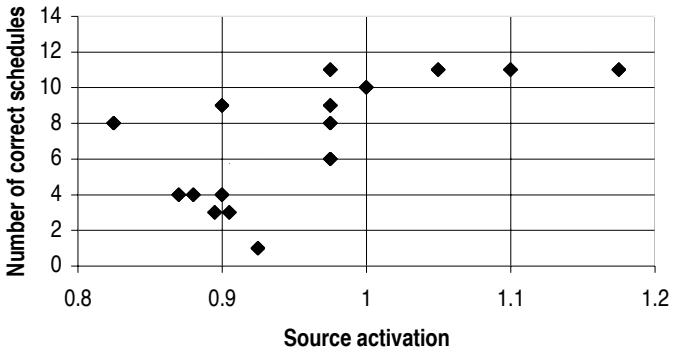


Figure 7.12. Relationship between estimated source activation and the number of correctly solved three-worker schedules. Each diamond represents one participant. The dashed line indicates the regression line.

The model predicts a strong correlation between working-memory capacity and the performance on the scheduling task. This proved to be the case in the experiment: the correlation between the estimated source activation and the number of successfully solved schedules is 0.56 (with n=16). This correlation increases to 0.66 if the analysis is restricted to the three-worker schedules, the schedules that require most working-memory capacity. Figure 7.12 shows the scatter plot for this latter relation. A more specific prediction of the model is that the effect of working-memory capacity on performance will diminish due to proceduralization. To investigate this prediction, the group of participants is split into eight low source-activation participants (W<0.95) and eight high source-activation participants (W>0.95). The proportion of correct solutions for each of the groups is plotted in figure 7.13. In this graph only three-workers problems are shown, and to average out part of the noise each data point is averaged with its predecessor and its successor. There is a clear convergence between the two curves, as can be seen in the bottom graph that depicts the difference.

7.7 Discussion

At the end of the previous chapter it seemed that production compilation was an uninteresting optimization of declarative knowledge. The scheduling model shows that this was a false impression. Complex reasoning processes in declarative memory can only become more complex because production compilation decreases demands on working-memory capacity.

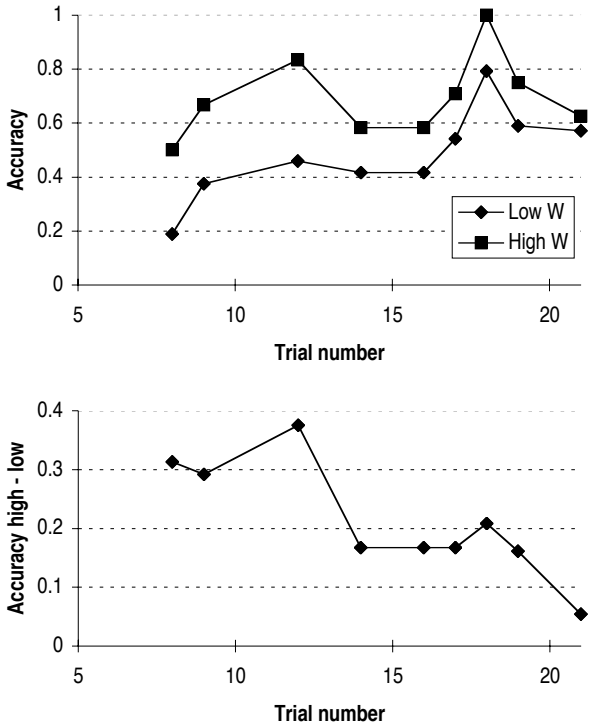


Figure 7.13. Proportion of correctly solved schedules for the 8 highest source activations and the 8 lowest source activations. Only three worker schedules are shown, and each point represents the average of three problems. The top graph shows the accuracy curves for both groups, the bottom graph shows the difference between the two.

The scheduling model also reveals insights into a part of the problem-solving process that is usually not part of cognitive models: the acquisition of task-specific rules from instructions. Although the model does not encompass a natural language parser, it is easier to imagine translating an instruction into a list of abstractions than into a set of production rules.

The abstraction representation chosen for this model is not the only possibility: probably a more general and efficient representation is possible. Optimizing the representation might be a good topic to study in conjunction with a more extensive system for creating new plans using old plans. A more general issue of representation that has become clear in this model is the fact that the degree of freedom ACT-R provides in choosing different types of chunks is probably too great. When general rules have to reason with declarative facts, having too many distinct types is a hindrance. The scheduling model uses only a few chunk types. The

downside of a chunk type that can be used for many purposes is that the number of slots becomes very large. Many of the slots are only needed at the moment that the chunk is the current goal, and are irrelevant for retrieval later on. For example, the generic goal type contains slots to store the current abstraction and the retrieved fact, and other bookkeeping slots. These slots are not needed anymore once the goal is popped.

Unfortunately, the model cannot yet solve the hard problems that participants had to solve in the experiment. The current model, however, shows many aspects also found in the experiment:

- Separate search episodes
- Elaborate and maintenance rehearsal
- Errors due to limited working-memory capacity
- Large individual differences
- Deliberate reasoning about the task
- Proceduralization of declarative knowledge

A number of issues are still unresolved. The way in which new abstractions are learned is a good starting point for discovering new strategies, but it is not yet clear whether that is sufficient to discover complicated strategies like the different-worker and fit-the-hours strategies in chapter 3. Another issue is the fact that ACT-R only maintains expected gains of production rules, and that it has no mechanisms to keep track of the quality of declarative knowledge. Some way has to be found to represent that, for example, a particular abstraction does not work most of the time.

Although these issues may involve even more explicit strategies, there is no fundamental problem in resolving them within the current framework. The main problem lies in the fact that people have a lot of relevant knowledge, even for an abstract task like the scheduling problem, and it is hard to specify all this knowledge and put it in a model.

7.8 Appendix: Implementation of abstractions in ACT-R

In this section I will discuss in detail how abstractions work. Readers not interested in the technical details may skip this section.

The basic generalized abstraction
The basic structure of a generalized abstraction is as follows:

```
GENERALIZED-ABSTRACTION
      ISA ABSTRACTION
      GOAL the type of goal this abstraction applies to
      RETRIEVE the type of fact that needs to be retrieved from declarative memory
      TEST constraints the retrieved fact has to satisfy
      ACTION how the goal is modified using the retrieved fact
```

The generalized abstractions have many properties of a typical production rule in ACT-R: the goal has to be of a certain type (GOAL), some fact is retrieved from declarative memory (RETRIEVE) satisfying some condition (TEST), and this fact is used to modify the goal (ACTION). Note that tests on the goal itself are part of the condition in the TEST slot. Suppose the goal of the Fincham task looks like:

```
EXAMPLE-FINCHAM-GOAL
      ISA HOCKEY-DAY-GOAL
      DAY1 WEDNESDAY
      DAY2 NIL
```

Further assume there are plus2 facts available of the following form:

```
EXAMPLE-PLUS2-FACT
      ISA PLUS2
      ARGUMENT WEDNESDAY
      ANSWER FRIDAY
```

An abstraction that specifies that plus2 facts are needed for the hockey-day-goal looks as follows:

```
EXAMPLE-FINCHAM-ABSTRACTION
      ISA ABSTRACTION
      GOAL HOCKEY-DAY-GOAL
      RETRIEVE PLUS2
      TEST DAY1=ARGUMENT
      ACTION DAY2:=ANSWER
```

In English, the interpretation of this abstraction is:

> If the goal is of type hockey-day-goal, retrieve a plus2 fact, so that the content of the day1 slot of the goal is equal to the argument slot of the plus2 fact, and put the contents of the answer slot of the plus2 fact in the day2 slot of the hockey-day-goal.

The representation presented above cannot be used directly. It needs to be interpreted, and this interpretation has to be done by production rules. Production rules, however, cannot inspect the names of slots, nor can the type of the goal (i.e., the contents of the ISA-slot) be variabilized. In order to circumvent this problem, some generalized goal representation is necessary with a fixed amount of slots. The representation I will use is as follows:

```
EXAMPLE-GENERAL-GOAL
      ISA GENERIC
      TYPE the type of the goal
      SLOT1 a general purpose slot to store results, arguments etc
      SLOT2 another general purpose slot
      SLOT3 a third general purpose slot
      ANSWER the answer, or true to indicate a goal that has succeeded
      ABSTRACTION slot to store a retrieved abstraction
      RETRIEVE slot to store the retrieved fact
      TEST slot to store the test
      ACTION slot to store the action
```

Unfortunately, the general purpose goal has many slots. Especially the abstraction, retrieve, test and action slots are necessary for processing purposes and are useless once the goal is popped.

The generic goal makes it possible to interpret abstractions using ordinary production rules. Let's look at our Fincham example again, and translate the goals into the generic goal:

```
EXAMPLE-FINCHAM-GOAL
      ISA GENERIC
      TYPE HOCKEY-DAY-GOAL
      SLOT1 WEDNESDAY
      ANSWER NIL
(all other slots are nil)

EXAMPLE-PLUS2-FACT
      ISA GENERIC
      TYPE PLUS2
      SLOT1 WEDNESDAY
      ANSWER FRIDAY
(all other slots are nil)
```

The example abstraction now becomes:

```
EXAMPLE-FINCHAM-ABSTRACTION
      ISA ABSTRACTION
      GOAL HOCKEY-DAY-GOAL
      RETRIEVE PLUS2
      TEST SLOT1=SLOT1
      ACTION ANSWER:=ANSWER
```

As we can see, slot names no longer label what is in a slot, making it slightly harder for us (but not for ACT-R) to interpret the meaning of abstractions and rules. The convention for tests and actions is that in a slotx=sloty or slotx:=sloty construction the slotx part refers to the goal, and the sloty part to the retrieved fact. When used in the test slot of the abstraction, it means that slotx of the goal has to match sloty of the

fact, and in the action slot of the abstraction, it means the contents of sloty of the retrieved fact have to be copied to slotx of the goal. It is important to note that ACT-R does not interpret these test and action instructions: they are just labels that are matched by the appropriate production rules.

Interpretation of an abstraction can be handled by four consecutive production firings. I will present the rules, and step through the Fincham example to illustrate it. First, an abstraction needs to be retrieved:

```
RETRIEVE-ABSTRACTION
IF    the goal is a generic goal of type type, and the abstraction
      slot of the goal is nil
      AND there is an abstraction with goal type
THEN  put the abstraction in the abstraction slot of the goal
```

This rule will be competing with ordinary task-specific rules, so it should have an expected gain that is not too high. In that case, when task-specific rules perform well and have a high expected gain, they will win most of the time, but when the task-specific rules have a low expected gain, abstraction retrieval will be preferred. This competition is comparable to the competition between search and reflection, as discussed in chapter 5. After the abstraction has been retrieved, the contents of its slots are copied to the goal.

```
COPY-ABSTRACTION-TO-GOAL
IF    the goal is a generic goal and some abstraction is in the
      abstraction slot of the goal
THEN  copy the contents of the retrieve, test and actions slots of
      the abstraction to their respective slots in the goal
```

In the Fincham example, these two rules will retrieve the example-fincham-abstraction and store it in the goal, so the goal will now become:

```
EXAMPLE-FINCHAM-GOAL
      ISA GENERIC
      TYPE HOCKEY-DAY-GOAL
      SLOT1 WEDNESDAY
      ANSWER NIL
      ABSTRACTION EXAMPLE-FINCHAM-ABSTRACTION
      RETRIEVE PLUS2
      TEST SLOT1=SLOT1
      ACTION ANSWER:=ANSWER
```

Now that the abstraction has been selected, the retrieval specified in its “retrieve” slot has to be carried out:

```
APPLY-ABSTRACTION-RETRIEVE-SLOT1-SLOT1
IF    the goal is a generic goal with an abstraction in the
      abstraction slot, and the retrieve slot has type retrieve
      and the test slot equals slot1=slot1 and slot1 has value slot1
      AND there is a fact of type retrieve and value slot1 in slot1
THEN  put this fact in the retrieved slot of the goal
```

This rule is specific to the slot1=slot1 test, so a similar rule is necessary for every possible test. In the Fincham example, this rule will look for a plus2 fact with wednesday as slot1 value, and will find our example-plus-fact, transforming the goal to:

```
EXAMPLE-FINCHAM-GOAL
      ISA GENERIC
      TYPE HOCKEY-DAY-GOAL
      SLOT1 WEDNESDAY
      ANSWER NIL
      ABSTRACTION EXAMPLE-FINCHAM-ABSTRACTION
      RETRIEVE EXAMPLE-PLUS2-FACT
      TEST SLOT1=SLOT1
      ACTION ANSWER:=ANSWER
```

Sometimes the fact that needs to be retrieved is not available in declarative memory. An alternative method to get a fact is to push it as a subgoal. The following rule accomplishes this for the slot1=slot1 case:

```
APPLY-ABSTRACTION-SUBGOAL-SLOT1-SLOT1
IF    the goal is a generic goal with an abstraction in the
      abstraction slot, and the retrieve slot has type retrieve
      and the test slot equals slot1=slot1 and slot1 has value slot1
THEN  push as a subgoal a goal of type retrieve and set slot1
      to slot1
      AND store this subgoal in the retrieved slot of the goal
```

In the Fincham example, this rule would create the following subgoal:

```
EXAMPLE-SUBGOAL-PLUS2
      ISA GENERIC
      TYPE PLUS2
      SLOT1 WEDNESDAY
      ANSWER NIL
(the rest of the slots also nil)
```

Resolving this subgoal of course needs knowledge in the form of other abstractions or productions to find the answer.

The final step is to carry out the action and remove the abstraction:

```
ABSTRACTION-DO-ANSWER-ANSWER
IF    the goal is a generic goal and fact retrieved is in the
      retrieve slot of the goal and the action slot equals
      answer:=answer
      AND retrieved has answer in the answer slot
THEN  put answer in the answer slot of the goal, set the
      abstraction, action and retrieved slots to nil, and put the
      original abstraction in the test slot
```

This production rule takes the answer from the retrieved slot and puts it in the answer slot of the goal, and resets the rest of the slots:

```
EXAMPLE-FINCHAM-GOAL
      ISA GENERIC
      TYPE HOCKEY-DAY-GOAL
      SLOT1 WEDNESDAY
      ANSWER FRIDAY
      ABSTRACTION NIL
      RETRIEVE NIL
      TEST EXAMPLE-FINCHAM-ABSTRACTION
      ACTION NIL
```

The abstraction that has just been used is retained in the test slot. Although this has a specific purpose that I will discuss in the next section, it also allows access to the abstraction even when the abstraction is proceduralized. The proceduralized version of the example is:

```
IF    the goal is of type hockey-day-goal and slot1 equals day1
      and the answer is nil
      AND there is a fact that day1 plus2 equals day2
THEN  put day2 in the answer slot of the goal and put
      example-fincham-abstraction in the test slot of the goal
```

This rule does exactly what we expect it to do, an leaves behind a reference to the example-fincham-abstraction. Even when the proceduralized version of the abstraction is fired, the declarative version is still available for retrieval, provided that the abstraction still has an activation that is high enough for retrieval. If the activation of an abstraction drops below the retrieval threshold, it becomes a meaningless symbol in the production rule and declarative conscious access is lost. Although the symbol has become meaningless, it still has a function, as we will see in the next section.

Chaining abstractions

In order to implement the chaining of abstractions, a few more production rules are needed. The following rule implements handling the prev-links:

```
ABSTRACTION-DO-NEXT
IF    the goal is a generic goal of type type, and the abstraction
      slot of the goal is nil and the test slot has value prev-abs
      AND there is an abstraction with goal type and prev prev-abs
THEN  put the abstraction in the abstraction slot of the goal
```

The rule for handling fail links is slightly different, since it has to fire whenever the current abstraction somehow gets stuck. The rule has to remove the current abstraction, and replace it by an abstraction that points to it using a fail link:

```
ABSTRACTION-REPLACE-FAIL
IF    the goal is a generic goal of type type and the abstraction
      slot contains some abstraction abs1
      AND there is an abstraction with goal type and fail abs1
THEN  put this abstraction in the abstraction slot of the goal
```

This production may of course only fire if we are really stuck, so we give it a low expected gain.

Proceduralizing abstractions

To proceduralize an abstraction, the same method is used as outlined in chapter 6. After an abstraction has been retrieved, but before its contents have been copied to the goal (so in between retrieve-abstraction and copy-abstraction-to-goal), a push-dependency rule may fire that pushes a dependency onto the goal-stack. The remaining steps are exactly the same as outlined in figure 6.2. The resulting production rules use the test slot of the goal to make sure steps are carried out in the right order. For example, the rule that results from proceduralizing abs 2 in figure 7.2 checks in its condition part whether abs 1 is in the test slot, and puts abs 2 in the test slot in the action part.

Building lists and doing rehearsal

Each item in a list is represented by a separate chunk, using the following chunk type:

```
EXAMPLE-LIST-ITEM
      ISA LIST-ITEM
      LIST-ID Reference to the first item in the list (to itself if it is the first item)
      VALUE The item that is stored in the list
      PREV Reference to the previous item in the list (nil if it is the first item)
```

In the scheduling goal, slot3 points to the last item of the current list. Tasks that are put in slot1 of the goal are added to the list by a production rule that creates a new list-item that replaces the current last item in slot3. As an example, figure 7.14 shows how the list "ABD" is represented.

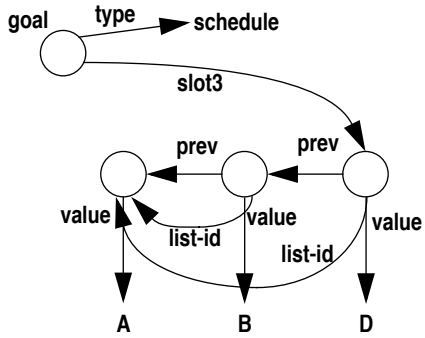


Figure 7.14. Example of the representation of the partial solution "ABD"

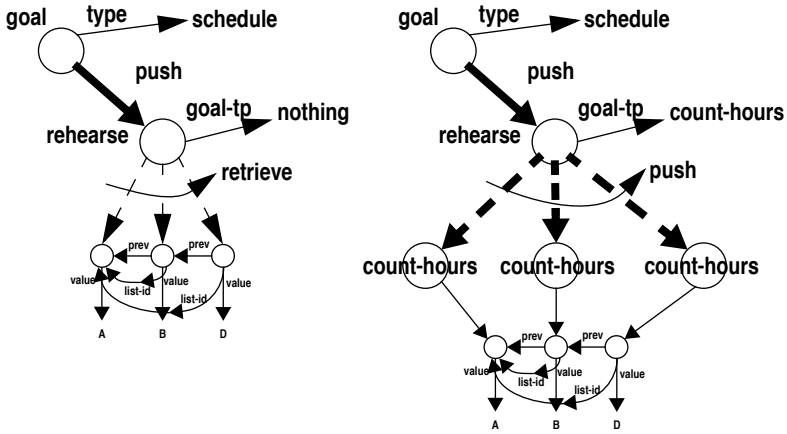


Figure 7.15. Schematic representation of maintenance rehearsal (left) and elaborate rehearsal (right)

Rehearsal is implemented by a subgoal that retrieves the list-items one at a time. If additional processing on items is required, in the case of elaborate rehearsal, a further subgoal is pushed in which the elaboration is carried out. Figure 7.15 shows a schematic representation of both types of rehearsal. In the case of maintenance rehearsal the content of the goal-tp slot in the rehearse-goal is 'nothing', and the list-items are retrieved one at a time without further processing. In elaborative rehearsal, the goal-tp slot of the rehearsal goal stores the goal type of the goal that has to do the elaboration ('count-hours' in the figure). For each item in the list a subgoal of that type is created and pushed onto the goal stack. Results of this processing are passed on from subgoal to subgoal, and are eventually passed on to the main goal.

Rehearsal is initiated by abstractions that have 'rehearsal' in their retrieve slot, and the type of the elaboration subgoal in the test slot (or 'nothing' in the case of maintenance rehearsal). The action slot specifies what has to be done with the results

of the elaboration. The following set of abstractions implements the strategy that calculates the total duration of the tasks already in the list. To do this, the durations of individual tasks in the list have to be retrieved and added. The first abstraction initiates elaborate rehearsal:

```
START-COUNT-REHEARSAL
      ISA ABSTRACTION
      GOAL SCHEDULE
      RETRIEVE REHEARSE
      TEST COUNT-STEP
      ACTION SLOT2:=ANSWER
```

When this abstraction is retrieved, a rehearsal subgoal (retrieve=rehearsal) is pushed with its goal-tp set to count-step (test count-step). The final result will be stored in slot2 of the goal (action slot2:=answer). Although the rehearse subgoal is implemented by production rules, the processing in the count-step goal still has to be specified:

```
COUNT-GET-TIME
      ISA ABSTRACTION
      GOAL COUNT-STEP
      RETRIEVE TIME
      TEST SLOT1=SLOT1
      ACTION SLOT3:=SLOT2
      PREV NIL
```

```
COUNT-ADD-TIME
      ISA ABSTRACTION
      GOAL COUNT-STEP
      RETRIEVE ADDITION
      TEST SLOT2=SLOT1*SLOT3=SLOT2
      ACTION ANSWER:=ANSWER
      PREV COUNT-GET-TIME
```

The rehearse subgoal puts the currently rehearsed item in slot1, and the current results of elaboration in slot2. It expects the result of the elaboration step in the answer slot. So, at the moment the subgoal of type count-step is pushed, slot1 contains a task, and slot2 contains the running total of the duration. The first step is to retrieve the duration of the task that is currently rehearsed. These durations are stored in chunks of type time, which have the task in slot1, and the duration of the task in slot2. Count-get-time retrieves a fact of type time which matches the task in slot1 (test slot1=slot2). It then stores the duration of the task in slot3 (slot3:=slot2). The next step is to add this duration to the running total in slot2. Count-add-time retrieves an addition fact with the first addend (which is in slot1 of the addition fact) equal to the running total and the second addend (in slot2) equal to the duration of the current task (slot2=slot1*slot3=slot2, a conjunction of two tests: slot2=slot1 and slot3=slot2), and stores the sum in the answer slot of the subgoal (answer:=answer). Whenever something is put in the answer slot of a goal it is automatically popped, so the elaboration subgoal is popped after count-add-time has finished.

Learning new abstractions

Suppose we need to make a schedule. We may use knowledge about lists to start with. How do we make a list? First we have to find a beginning. Once we have a beginning, we find a next task until we are done. A general set of abstractions to create a list might look like:

```
FIND-BEGINNING                    FIND-NEXT
      ISA ABSTRACTION                   ISA ABSTRACTION
      GOAL LIST                         GOAL LIST
      RETRIEVE BEGINNING                RETRIEVE NEXT
      TEST SLOT3=NIL                    TEST SLOT2=SLOT1
      ACTION SLOT1:=SLOT1               ACTION SLOT1:=SLOT2
      PREV NIL                          PREV FIND-BEGIN
```

This representation assumes that the list is stored in slot3 of the goal (as illustrated in figure 7.14), and that items in slot1 are added to the list and copied to slot2. Find-beginning specifies that if there is no list yet (test slot3=nil), a beginning has to be found, and this beginning has to be stored in slot1 (action slot1:=slot1). Once list-building productions have added the item in slot1 to a new list in slot3, and have transferred this item to slot2, the find-next abstraction specifies that a next relation has to be found between the item in slot2 (test slot2=slot1), the item we have just added to the list, and some new item, which will be stored in slot1 (action slot1:=slot2).

In the ACT-R model, the process of adaptation does not precede the rest of processing, but rather is part of it. A new strategy is initiated by a production rule that pushes an abstraction as a subgoal. This subgoal only produces the first step in the solution plan: later parts of the plan are generated when needed later on. The rule that pushes an abstraction goal is the subgoaling version of the rule that retrieves abstractions:

```
SUBGOAL-ABSTRACTION
IF    the goal is a generic goal of type type, and the abstraction
      slot of the goal is nil
THEN  set as a subgoal an abstraction with goal type
      AND put this abstraction in the abstraction slot of the goal
```

This rule has to compete with the retrieve-abstraction rule, but since it has a higher cost, it will only occasionally win the competition if an abstraction is already available in the current situation (or, it will almost never win if there is a task-specific production rule available with a high expected gain). If there is no abstraction available, the retrieve version of the rule will fail, and subgoal-abstraction will be chosen automatically. Once an abstraction has become the goal, the first step is to find a goal that is similar to the desired goal-type in the goal slot of the abstraction. For example, in the case of scheduling, the abstraction subgoal becomes:

```
EXAMPLE-ABSTRACTION-SUBGOAL
      ISA ABSTRACTION
      GOAL SCHEDULE
      RETRIEVE NIL
      TEST NIL
      ACTION NIL
```
etc.

The current model uses an explicit representation to store relations between goal types, for example, it represents that schedule is related to list, and next is related to order. An alternative, but less reliable, option is to use implicit association strengths to find related goals. The following rule implements the explicit version:

```
FIND-ASSOCIATED-GOAL-TYPE
IF    the goal is an abstraction for goal type goal-tp1 and no
      associated goal-type has been found yet.
      AND goal-tp1 is related to goal-tp2
THEN  put goal-tp2 in the test slot of the goal
```

In our example, this rule stores 'list' in the test slot of the abstraction. In the next few steps (for reasons of brevity, I will omit the production rules), an abstraction of the associated type is retrieved and its slots are copied to the abstraction, producing (assuming find-beginning is retrieved):

```
EXAMPLE-ABSTRACTION-SUBGOAL
      ISA ABSTRACTION
      GOAL SCHEDULE
      RETRIEVE BEGINNING
      TEST SLOT3=NIL
      ACTION SLOT1:=SLOT1
      PREV NIL
```

No more adaptations are possible for this abstraction, since the beginning type in the retrieve slot cannot be related to any task-specific aspect. As a consequence, applying this abstraction will lead to a subgoal of type beginning. Once the next step in the plan, the find-next abstraction, has been adapted to the schedule goal, the retrieve type can be filled with a task-specific term. The situation is as follows:

```
EXAMPLE-FOLLOW-UP-ABSTRACTION
      ISA ABSTRACTION
      GOAL SCHEDULE
      RETRIEVE NEXT
      TEST SLOT2=SLOT1
      ACTION SLOT1:=SLOT2
      PREV EXAMPLE-ABSTRACTION-SUBGOAL
```

In this case a fact of type next has to be retrieved. But since next facts are related to order constraints, next can be substituted by order, producing:

```
EXAMPLE-FOLLOW-UP-ABSTRACTION
      ISA ABSTRACTION
      GOAL SCHEDULE
      RETRIEVE ORDER
      TEST SLOT2=SLOT1
      ACTION SLOT1:=SLOT2
      PREV EXAMPLE-ABSTRACTION-SUBGOAL
```

Although there are probably more ways to adapt abstractions, this are sufficient for the second model.

CHAPTER 8

Concluding remarks

The goal of this thesis, as stated in chapter 1, is the development of a theory of problem solving that is psychologically plausible, and does justice to its complexity. The weak-method theory that stems from artificial intelligence acknowledges this complexity, but shows only very limited correspondence with human data. Theories from experimental psychology, on the other hand, neglect the complexity of problem solving, and theorize about how complex skills can be explained in terms of memory systems and basic information processing. In the course of this thesis, the subject of problem solving has been broadened to cognitive skills in general, since there is no principled distinction between the two. The approach I have chosen is to focus on learning cognitive skills. In chapters 5 to 7, a view of skill learning has taken shape, which I will try to explicate in this final chapter. This view is built upon the foundation the ACT-R architecture offers, ensuring a plausible system of learning and memory, and it exhibits the complexity of behavior that artificial intelligence is interested in. The theory of skill learning that emerges has a number of areas that need more investigation, and a number of possible applications. Both topics will be discussed in the final sections of this chapter.

Have I solved the problem of NP-completeness? The answer to this question is of course "no". But I have tried to draw a picture of how humans can acquire the knowledge to at least partially solve hard problems. This is not a simple picture, and cannot be summarized in a clear-cut algorithm. When people start with a new task, they do not use a general machine learning algorithm to acquire knowledge about this task. Rather, they use a set of strategies, knowledge about other domains and tasks, instructions about the current task to start with, and keep adapting and refining their knowledge while they are working on the task. A better understanding of these processes is the key to understanding complex human problem solving.

8.1 The skill of learning

Skills are not separate entities, they almost always rely on other skills. In order to learn multiplication, one has to master addition first. In order to be able to add numbers, one has to be able to count, etc. The diagram I used in chapter 1 to outline the growth of knowledge for a certain problem (figure 1.4) is useful to intuitively sketch skill learning in general. Let's visualize the space of possible tasks in a two-dimensional diagram (figure 8.1). Again, the idea is that tasks that are similar are close to each other in the diagram. The vertical axis is used to indicate complexity. A task is higher in the diagram if the skill to perform this task relies on skills associated with a task lower in the diagram. The result is a partial ordering. Each task can be instantiated in numerous ways, so each of them is shown as a small set (a box). Now, if the set boundaries are dissolved, we're back at figure 1.4, except that the diagram now represents the set of instances of all possible tasks.

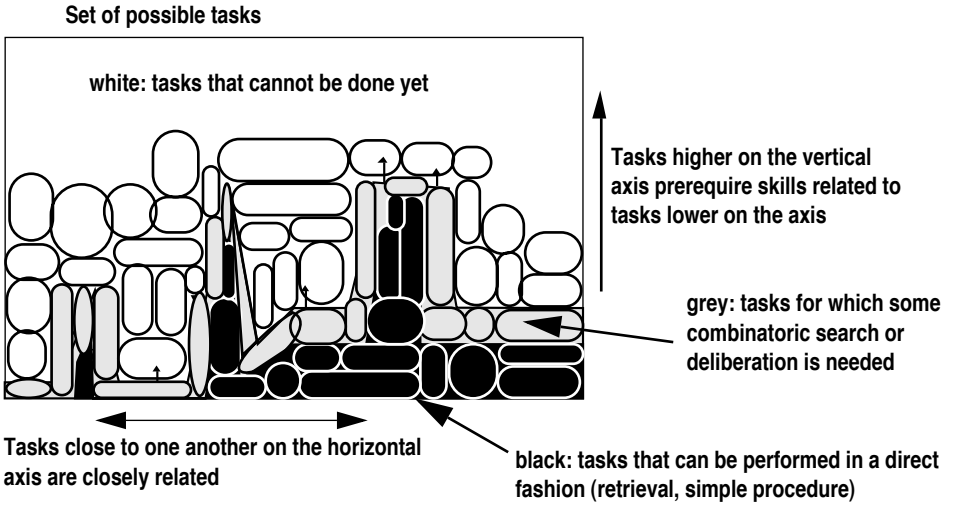


Figure 8.1. The set of possible tasks

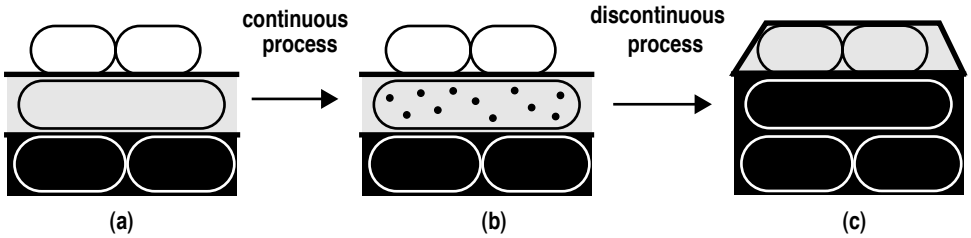


Figure 8.2. Summary of the skill acquisition process. The figure shows a detail of figure 8.1. (a) The rectangle in the middle represents the task to be learned. The two black rectangles at the bottom represent prerequisite skills, which already have been mastered. Experience with the task produces examples or instances (b). Once there are enough examples, generalization can be successful (rule learning), producing situation (c).

Figure 1.4 gave the impression of a gradual learning process, in which the boundaries between the black, grey and white areas slowly move upwards. On the basis of chapter 5 to 7, it is possible to revise this picture, and to show how discontinuous learning can occur. Suppose a new skill is being learned. To be able to start at all, the prerequisite skills have to be mastered first (i.e., the have to be in the black area). This means that the task itself has to be in the grey area (figure 8.2a). Experience with the task will produce examples of solutions. These examples can be retrieved later on, producing "specks of black" in the grey area that represents the new skill (figure 8.2b). This process is similar to instance learning, Piaget's assimilation, implicit learning and the I-phase of the RR-theory. Although it seems that the black specks will eventually fill up the whole set, this is only true for tasks with a finite number of instantiations. Mastery of an infinite set of instantiations

would require an infinite number of examples. To master an infinite set as a whole, one or more generalization steps are necessary. Once a successful generalization is made (possibly after several unsuccessful ones), it is suddenly possible to efficiently solve the set as a whole (figure 8.2c). The process of generalization is similar to rule learning, Piaget's accommodation, explicit learning and the E1-phase of the RR-theory. A side effect of successful mastery of a skill is that it is now possible to learn skills that were previously unreachable. This means a sudden shift in the border between the grey and the white area (figure 8.2c).

An important aspect of this process is that the generalization step is a skill itself. As we have seen in chapter 5, adults use a more elaborate form of generalization than children do. It is this aspect of learning that makes human learning virtually limitless. Understanding this particular set of skills may well be the key to understanding many aspects of individual differences and cognitive development. In the next section, an ACT-R implementation of this idea will be outlined, based on the models from chapter 6 and 7. In section 8.3, I will elaborate on the issue of individual differences.

8.2 Processes involved in skill learning

In chapter 6, I proposed a first version of a general schema of skill learning (figure 6.1). Although this schema offered a good description for the two models discussed, the Sugar Factory and the Fincham task, it did have a problem: the vague notion of the initial method. In both the Sugar Factory model and the Fincham model these initial methods were hard-coded into the model. The origin of this knowledge was left unspecified. The scheduling model in chapter 7 showed how this problem can be solved. In this section, I will update figure 6.1 according to the modifications introduced in chapter 7. The final skill learning theory I propose encompasses two currently dominant theories, the theory of rule learning, rooted in the original ideas by Fitts (1964) and further extended by Anderson (1982), and the idea of instance learning, as posed by Logan (1988).

Now how can this theory produce the kind of learning discussed in the previous section? Suppose we have a task in the grey area (as in figure 8.2a). This means some way of doing this task is available, although it is inefficient. This initial method may involve a set of declarative rules that has been adapted through analogy to suit the current task, or is the result of interpreting instructions. Doing the task using these initial rules produces examples that are stored as instances. At some point, the explicit learning strategies will attempt some generalization (in terms of chapter 5: a reflection episode). As soon as the generalization is successful, and the new, efficient declarative rules are subsequently proceduralized, the skill is mastered.

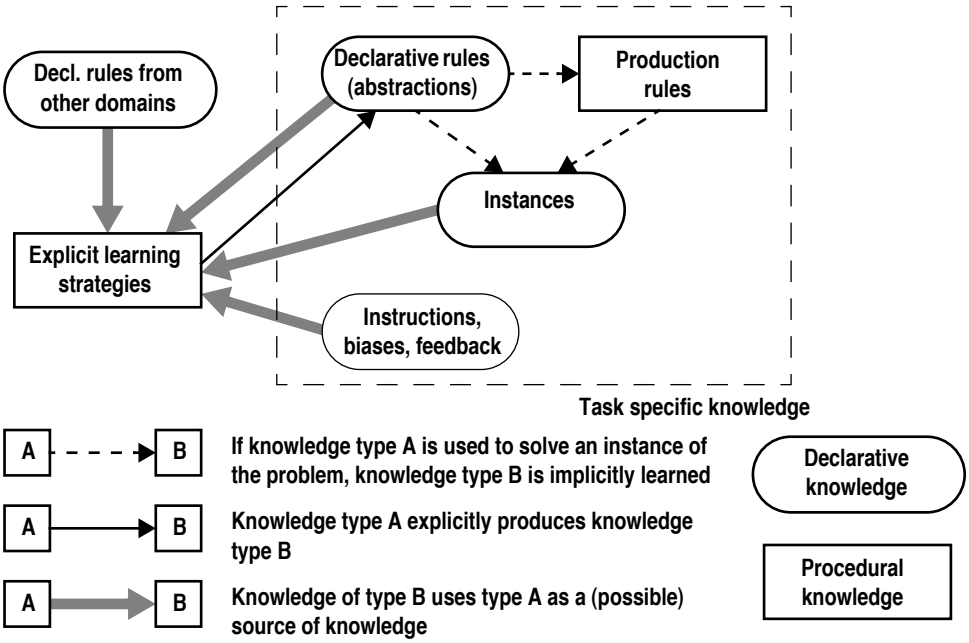


Figure 8.3. Overview of the proposed skill-learning theory

Figure 8.3 shows a revised overview of the theory, an expansion of figure 6.1. Each of the boxes in the diagram represents a type of knowledge. The dashed boundary indicates which of these knowledge types are task specific. Each rectangle represents a type of knowledge that is associated with a certain strategy. Each of these strategies needs knowledge associated with it to function properly. Within the task-specific knowledge, there are three possible strategies, each of which represents a possible way to perform the task: using a declarative rule, using a production rule, or retrieving an instance. According to ACT-R, the strategy with the highest expected outcome will win the competition. The instance strategy is generally the best strategy, followed by production rules, declarative rules and using an explicit learning strategy.

Using one type of knowledge to perform the task results in learning: not only is the knowledge itself reinforced, but it may also produce knowledge necessary for other strategies as a by-product. This implicit learning is represented by the dashed arrows. The strategy outside the task-specific knowledge represents explicit-learning strategies. The goal of this strategy is to produce task-specific knowledge in the form of declarative rules. In order to do this, it can use different knowledge sources, both within and without the task domain. The use of these sources is represented by the grey arrows.

Explicit learning strategies
There are many possible learning strategies, and they are an important source of individual differences. Explicit learning strategies themselves can also be considered skills, and can be learned in the same way as other skills. Eventually, some bootstrapping problem must be solved, so some initial strategy should probably be part of the architecture itself.

Some possible strategies are:

- Use analogy to apply declarative rules from other domains to the current domain
- Generalize instances into declarative rules
- Create a direct representation of the instructions
- Refine current declarative rules based on feedback

Both the Fincham model (chapter 6) and the scheduling model (chapter 7) use analogy initially, although the Fincham model does so in a task-specific fashion (i.e., the method of analogy was already part of the instructions). In some cases instructions provide for an initial method. In that case the main task is to translate the instructions into declarative rules that can be carried out. But in many cases instructions will also draw on knowledge people already have.

Declarative rules and Instances
Declarative rules have to be interpreted in order to be carried out. As such, they require attention and working-memory capacity. Since they are declarative, these rules can be subjected to deliberate reasoning, which may lead to new declarative rules. Whether or not learning of declarative rules will contribute to performance in the long run, not only depends on the available knowledge, but also on the task. The Sugar Factory experiment in chapter 6 illustrates this: the possibility of control is limited, the behavior of the system is non-linear, and there is random noise involved. Therefore, learning of declarative rules will fail for most participants, so performance can be explained by instance learning alone. In other cases learning of declarative rules may be successful, but will not prove to be the most cost-efficient strategy. This is the case when the set of instances is small, so that instance retrieval can dominate performance. In the Fincham task, the use of declarative rules was a very useful strategy, and in the scheduling task it was crucial, since the instructions alone are clearly insufficient to do the task and examples are not very useful.

Production rules
Production rules serve the same function as their declarative counterparts. But since they can be executed in one step instead of being interpreted, they are much faster. Another advantage is that they use less working memory capacity due to the fact that there is no longer a declarative rule that has to be kept active. Although

production rules provide no new knowledge when compared to their declarative cousins, they make it possible to solve problems that could not be solved before because of limitations in working memory capacity or available time. This may even open the way to learning more difficult skills.

In ACT-R, production rules are learned through a specific type of goals (dependencies). In that sense production-rule learning requires deliberate planning. The encapsulation explained in chapter 6 and 7 turns this goal-directed type of learning into an implicit learning mechanism. In the current models, each time a declarative rule is retrieved, there is a chance that a production rule will be learned. Although this method fits the data quite nicely, there are alternatives, such as a gradual transfer from the declarative to the procedural version of a rule. This raises the question whether procedural memory and declarative memory are really distinct in principle. Is proceduralization not a process of gradually speeding up the process of interpreting declarative rules while reducing interference?

Implicit and explicit learning
The distinction between implicit and explicit learning follows naturally from this theory. Implicit learning is a result of normal processing, producing new instances and proceduralization. Explicit learning is based on strategies. Using these strategies is also a form a normal processing, except with another goal. Instead of performing the task, the goal is to generalize new knowledge. Due to this fact, explicit learning is "conscious", since the episode can be recalled later on by retrieving the learning goal, while implicit learning is not. Explicit learning is a skill that has to be learned. This fact can explain the large individual differences in explicit learning whether due to age, intelligence or brain damage.

8.3 Individual differences

In this thesis two sources of individual differences have been discussed in detail. In chapters 4 and 5, the idea has been put forward that individual differences stem from differences in explicit learning strategies. Experiments in discrimination-shift learning, for example, exhibit a qualitative difference in learning between young children and adults. The model in chapter 5 shows how this difference can be explained in terms of different strategies. The scheduling model in chapter 7 further suggests that these explicit learning strategies can be subdivided into a set of declarative rules that can be reused, and production rules that adapt these declarative rules to new situations.

Chapter 7 introduces another source of individual differences: working memory capacity, which corresponds to source activation in ACT-R. Correlational studies, for example by Kyllonen and Christal (1990), find correlations of between 0.80 and 0.90

between reasoning ability factors and working-memory capacity. All the tests in their experiment, however, were relatively short. The model in chapter 7 and the results in the previous section suggest that the advantage of working-memory capacity diminishes due to procedural learning. With respect to individual differences on a larger time scale, explicit learning strategies may become more important. Nevertheless, a high working-memory capacity may imply more flexibility, leading to a larger range of problems that can be solved in a declarative fashion. The evidence up to now is inconclusive with respect to the interaction between working memory capacity and reasoning ability.

Although the supporting evidence is still scarce, the modeling approach to individual differences has great advantages over the traditional approach in IQ-tests. Working-memory capacity and explicit learning strategies are information processing notions. The exact impact of these aspects on performance can be predicted by an ACT-R model. For example, Kyllonen and Christal classify a particular test as a working-memory test on the basis of intuition alone. In that case, working-memory capacity is just a factor that is defined as "what all the working-memory tests measure". In the case of the digit-working memory task, on the other hand, working-memory capacity is a parameter in the model that can be separated out from, for example, the effect of learning. A further advantage of a parameter is that it leads to precise predictions about individual differences in other tasks. As a result, research in individual differences may move from a descriptive to a predictive stance.

Binet's (1962; originally published in 1911) original IQ concept was based on ideas about development, in the sense that IQ represents the "mental age" divided by the real age of a child. This idea is, however, no longer valid in modern IQ tests. As was argued in chapter 5, individual differences due to development can best be explained by differences in available explicit strategies. I do not believe that the source activation parameter in ACT-R, the main determinant of individual differences in working-memory capacity, is susceptible to development. Anyone who has played the game of memory with children may attest this. Tests of working-memory span do not measure working-memory capacity only, but also memorization strategies like rehearsal.

The skill-learning theory presented in this thesis may also be useful in the study of cognitive development. Explicit learning strategies are themselves skills, and their development may resemble the development of skills, but on a larger time scale. The table in figure 8.4 shows how we may think about this issue in terms of a time scale of human learning. This table is analogous to the time scale of human action, as originally conceived by Alan Newell (1990). Individual skills are learned and proceduralized in terms of minutes and hours, while general intelligence develops in terms of weeks and years. This is not because a general learning strategy is fundamentally different from a skill. Skills are based on generalizing knowledge to

Time scale of human learning				
Scale (sec)	Time Units	Type of representation	Memory system used	
10^8	years	Learning strategies	Procedural	Develop–mental band
10^6	weeks	Generally useful declarative rules	Declarative	
10^4	hours	Task-specific production rules	Procedural	Skill band
10^2	minutes	Task-specific declarative rules	Declarative	
10^0	seconds	Task-specific facts	Declarative	Instance band

Figure 8.4. Time scale of human learning, analogous to Newell's (1990) time scale of human action

produce specific facts. To go from a fact to a skill takes two orders of magnitude on the time scale (from 10^0 to 10^2). It takes another two orders of magnitude to proceduralize this knowledge. An assumption behind these learning steps is that it includes all aspects that lead to an optimally useful representation, such as learning parameters, discrediting wrong generalizations, etc. To learn knowledge that is generally useful, knowledge related to specific skills needs to be generalized, which takes another two orders of magnitude.

8.4 Evaluation of ACT-R

This thesis could not have been written without the ACT-R cognitive architecture. It has provided both a theory and a modeling environment with just the right level of constraints. ACT-R's constraints actually help the modeler in designing a proper model for his data. Too few constraints leave too many choices, while too many constraints may force the modeler to spend too much time overcoming the limitations of the architecture. ACT-R offers a solid basic system of learning and memory, grounded in empirical data. An ideal architecture would be a system in which it is just sufficient to specify some necessary knowledge, after which the architecture exhibits psychologically plausible behavior. ACT-R comes close to this ideal.

In the course of the research described in this thesis, I often pushed the envelope of the capabilities of ACT-R. As a consequence, I have encountered some aspects of ACT-R that are still underspecified. I think this thesis can contribute to ACT-R by helping to point out and sometimes fill in some of the gaps.

Production compilation

One of the cornerstones of the ACT-R theory is the distinction between declarative and procedural memory. This distinction has proved to be very useful, not only in terms of the theory, but also in providing users of the theory with a relatively easy-to-learn modeling environment. Having two long-term memory stores, however, also produces additional complications. More specifically: ACT-R's current procedural learning is not yet completely adequate. Using dependencies in ACT-R is still too much like programming, and some of the productions that fill slots in a dependency goal lack any psychological plausibility. The method for creating dependencies introduced in chapter 6 and extended in chapter 7 abstracts away from this process by focusing on declarative rules. The actual production rules are learned more or less automatically if their declarative counterparts are used often enough. In order to use these declarative rules, and stay consistent with earlier ACT systems like ACT* (Anderson, 1983), a set of general interpretive productions is necessary, or at least a framework in which they can be defined. The scheduling model in chapter 7 offers some kind of solution, which may need some more elaboration to be useful in any setting.

An issue we should keep in mind, in the spirit of Newell, is not to be too dogmatic about the declarative/procedural distinction. From the fact that declarative knowledge apparently gradually changes into procedural knowledge we may deduct that at a deeper level declarative and procedural knowledge are similar after all. Finding this deeper equivalence not necessarily contradicts the ACT-R theory. Rather, it may strengthen the theory by pointing out how a conceptually useful distinction can be grounded in a more parsimonious, but probably less usable, system.

Chunk types

Another unresolved issue in ACT-R concerns chunk types. There is no mechanism that can learn new types, neither will it be easy to specify one. In most models, this problem does not surface, since most models are only concerned with a specific task. Production rules that implement explicit learning strategies need to be able to operate on several different types of chunks. The solution I have chosen, to use a generic goal type, is not entirely satisfactory. It makes chunks and production rules harder to read and understand. Because the generic chunk type must be multi-purpose, it contains too many slots. Apart from aesthetic reasons, large chunk-types have two serious disadvantages: spreading activation is diluted over more chunks, and collapsing two chunks with the same information is more likely to go wrong due to the fact that irrelevant bookkeeping slots have different contents.

Ideally, a model uses small production rules and chunks with only a few slots. It is almost impossible to satisfy both of these constraints at the same time in a model in which new production rules are learned. There are probably no easy solutions to this

problem, but it should be a consideration if one attempts to specify a new production compilation mechanism.

Base-level decay

The models in this thesis have used either 0.5 or 0.3 as the base-level decay parameter. A fast decay of 0.5, which is the official recommended value of the parameter, turns out to work best for decay within an experimental session. A slow decay of 0.3 is necessary for experiments in which an hour, a day or a week passes between experimental sessions, else ACT-R will forget all it has learned. This problem has also been noted by Anderson, Fincham and Douglass (submitted). A decay of 0.25 is even necessary to account for some of their data. Since base-level decay is a parameter that is supposed to remain constant, this poses a problem that has to be resolved. A possible solution, explored by Anderson et al., is to change the decay function, so that it decays fast at the start, but more slowly as time progresses. Another possible solution is to suppose that base-level decay is slow all the time, but that the apparently high decay during an experiment is due to interference. This interference, for which association strength learning in ACT-R can account in principle, may also be the key to resolving this issue.

Production-strength learning

The learning mechanism I haven't used in any of the models in this thesis is production-strength learning. Strength is a parameter maintained with each production rule, reflecting its past use in the same manner as base-level activation for chunks. Strength is a parameter in the equation that determines the time it takes to retrieve a chunk (equation 2.3). Since strength influences the retrieval time of chunks, production rules that do not retrieve chunks other than the goal are not affected at all by strength. A second problem is that the strength of the production and the activation of the chunk are added together in the retrieval time equation. As a consequence, if the strength of a production rule is high enough, it can retrieve almost any chunk in almost zero time.

Why would ACT-R need production-strength learning? Generally, strength learning is used in ACT-R models to account for the fact that there is a speed-up in performance on new tasks. These models often assume that participants have already learned the necessary task-specific production rules at the start of an experiment. Using these rules improves their speed. An alternative account would be along the lines of the scheduling model: at the start of an experiment, most task-specific knowledge is still declarative. This declarative knowledge is only gradually compiled into production rules, providing the speed-up normally explained by strength learning.

Assessing model fits

At several points in the thesis, I have criticized the R^2 measure as a method to assess the quality of fit between the data and the model. This measure is not sensitive to the spread of the data, and is not suitable if there are only a few data points to compare. Moreover, the number of free parameters in the model is not taken into account in the measure. So, if several models are compared with respect to the R^2-measure, the model with the highest value is not necessarily the best model. It would be very desirable to have a method similar to the multi-level methods described in chapter 3, in which addition of a parameter to the model has to be defended by showing it provides for a significantly better fit. Of course, this problem is not specific to ACT-R, but applies to cognitive modeling in general.

A look back at Soar

One of the concerns in the research of cognitive architectures has always been: is it not possible to implement any model you want in any cognitive architecture? For example, would it not have been possible to model all the data discussed in this thesis by Soar models? This is tough to answer, since it is very hard to prove something cannot possibly exist. But let us take a very simple example, the Tulving model discussed in chapter 4. The important issue in that model is the notion of forgetting, and the fact that certain information is forgotten in a week and other information apparently not. The ACT-R model shows that this dissociation naturally follows from a rationally organized declarative memory, without the need to resort to an explanation that assumes separate memory systems for implicit and explicit memory. If one were to model this experiment in Soar, the process of forgetting would have to be part of the model. Although it is possible to come up with such a model eventually, the fact remains that the aspect of forgetting information is not part of the specific task, so should not be part of a model of that task.

Nevertheless, Soar has been a source of inspiration for many of the models in this thesis. The idea, introduced in chapter 5, of pushing a dependency as a subgoal in situations where no promising other rules apply corresponds closely to the Soar notion of pushing a subgoal in case of an impasse. The interpretation process of declarative rules, as discussed in chapter 7, also has close ties to the way Soar handles operators. There are many good ideas in the Soar architecture, and its failure to penetrate main-stream psychological research is probably due to the fact that the area in which it excels, complex problem solving, is a topic that is not as central in cognitive psychology as it should be.

8.5 Practical implications

In this thesis I have shown that problem solving cannot be studied properly without taking learning into account. Although this idea may not be too controversial in the domain of problem solving, many practical applications still assume that non-learning reasoning systems can be built that reason in a human-like fashion. The main applications of rule-like systems are expert systems, human-computer interaction and education.

Application in the domain of expert systems
The assumption of expert-system design is that it is possible to specify all the relevant task-specific knowledge for a certain task. This may be true in the case of simple tasks, but not of all tasks in general. It is impossible to make a non-learning expert system for scheduling. For tractable problems, one might also wonder whether the expert-knowledge approach is the best. Since it is estimated that the number of rules an expert has on a certain domain is around 50.000, it is highly impractical to try to specify all of them. Even if it is possible to specify all these rules, the subsymbolic knowledge associated with these rules also has to be defined. This subsymbolic information is crucial in finding the right information at the right moment. Expert problem-solving behavior is probably not the invocation of stored knowledge, but an active process of constructing new knowledge for the current purpose. Apart from this explicit learning aspect of expert behavior, implicit learning, by means of the ACT-R learning mechanisms, keeps organizing the subsymbolic aspects of the knowledge.

A promising alternative method for constructing expert systems is to use the skill-learning theory presented in this thesis. Knowledge the system has to use can be supplied in a declarative fashion, after which the system is submitted to a training program. As a result, the system will organize the knowledge in the most profitable way, either as production rules, or as examples, or it will forget knowledge that does not prove to be useful altogether. Some issues have to be resolved before this can be a viable method: the right set of learning strategies has to be found, and a set of generally useful declarative rules.

Application in the domain of cognitive ergonomics
The same point made about expert-system design can also be made with respect to task analysis for the purpose of interface design. For difficult tasks it will be impossible to make a complete task-analysis. Task analysis has another drawback: since it always investigates the knowledge of an expert user, it can only make approximate predictions about novice users. Instead of trying to identify all the knowledge an expert user has, a model can be supplied with the instructions a

novice user gets, and be submitted to the same training program novices have to go through. Again, the skill learning theory, when properly extended, can be very useful for this purpose.

The theory also allows for the study of the integration of task-specific knowledge with knowledge about other tasks (the assimilation paradox, see Mulder, Lamain & Passchier, 1992). A general guideline in interface design is that the interface should help the user to get an adequate mental model of the system (Norman, 1988). Although this guideline is considered very useful, the notion of a mental model is rather vague. Neither is it clear when a mental model is adequate enough, and to what kinds of mistakes a certain mental model may lead. These kinds of questions can all be studied in the proposed modeling framework of skill acquisition. The notion of explicit learning of declarative rules is closely related to the concept of a mental model. Both are consciously inspectable knowledge structures that can be used in an interpretive fashion to make decisions about what actions to take. When used within the skill-learning framework, it is possible to make predictions about how knowledge from a mental model is proceduralized.

Application in the domain of education

The idea that cognitive development involves explicit learning strategies can also have implications for education. The goal of education is not just to teach children specific skills, but also to teach children how to approach problems in general. This latter goal can only be achieved indirectly, since general strategies can only develop by learning specific skills. But if we know more about this process of strategy learning, we might be able to select a set of skills to teach that is optimal for general strategy development. This is not only applicable to children. One of the main goals of university education is to teach "an academic way of thinking", although this is not taught in any particular individual course.

8.6 A Unified Theory of Learning?

In this final chapter I have outlined a theory of skill learning. This theory uses existing theories of learning, glued together by elements inspired on the principle of rational analysis. This theory is supported by several models discussed during the course of this thesis. These ingredients may eventually be parts of a unified theory of learning, which is itself a piece of the puzzle for a unified theory of cognition. In order to specify such a unified theory of learning, a simulation environment is needed that implements it. It might take the form of an extension to ACT-R, and be capable of learning its own task-specific knowledge from instructions. This implementation would strengthen the theory, and enable many new predictions and applications.

Web documents and Publication list

The following documents are available from the webpage
http://tcw2.ppsw.rug.nl/~niels/thesis/

The ACT-R architecture itself can be obtained from:
http://act.psy.cmu.edu/

Chapter 3

- Verbal protocols of all participants (Dutch)
- The detailed protocol analysis of participant 2 (Dutch)

Chapter 4

- Model of the dissociation experiment
- Models of rehearsal and free recall

Chapter 5

- A Macintosh Microsoft Excel 4.0 file with the dynamic growth model
- Model of the beam task
- Model of discrimination-shift learning

Chapter 6

- Model of the Fincham task

Chapter 7

- Model that demonstrates the influence of W on working memory span
- Two models of scheduling

Publication list

Taatgen, N.A. (1993). Complexiteitsaspecten van leermechanismen. *Proceedings of the Benelearn-93 conference*. Artificial Intelligence Laboratory, Vrije Universiteit Brussel, Belgium.

Taatgen, N.A. (1994a). The study of learning mechanisms in unified theories of cognition. In F.J.Maarse, A.E. Akkerman, A.N. Brand, L.J.M. Mulder, M.J. v.d. Stelt, *Computers in psychology 5*, Lisse, the Netherlands: Swets & Zeitlinger.

Taatgen, N.A. (1996a). A model of free-recall using the ACT-R architecture and the phonological loop. In H.J van den Herik & T. Weijters (Eds.), *Proceedings of Benelearn-96* (pp. 169-178). Maastricht, the Netherlands: Universiteit Maastricht, MATRIKS.

Taatgen N.A. (1996b). Learning and revising task-specific rules in ACT-R. In U. Schmid, J. Krems & F. Wysotzki (Eds.), *Proceedings of the First European Workshop on Cognitive Modeling* (report no. 96-39, pp. 31-38). Berlin, Technische Universitaet Berlin, Fachbereich Informatik.

Taatgen, N.A. (1997a). A rational analysis of alternating search and reflection in problem solving. *Proceedings of the 19th Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum.

Hendriks, P., Taatgen, N.A & Andringa, T.C. (Eds.) (1997b). *Breinmakers & Breinbrekers, inleiding cognitiewetenschap*. Amsterdam: Addison-Wesley Longman.

Lebiere, Christian, Dieter Wallach & Niels Taatgen (1998). Implicit and explicit learning in ACT-R. In Frank Ritter & Richard Young (eds.), *Proceedings of the Second European Conference on Cognitive Modelling*. Nottingham, UK: Nottingham University Press.

Taatgen, N.A. (1999). Explicit Learning in ACT-R. In U. Schmid, J.F. Krems, F. Wysotki (Eds.), *Mind Modelling: A Cognitive Science Approach to Reasoning, Learning and Discovery*. Berlin: Pabst Science Publishers.

Taatgen, N.A. (1999). Review of 'The atomic components of thought'. *Trends in Cognitive Sciences, 2*, 82-82.

Taatgen, N.A. (submitted). A model of learning task-specific knowledge for a new task. Submitted to the cognitive science conference.

Taatgen, N.A. (submitted). Implicit versus explicit: an ACT-R learning perspective; Commentary on Dienes & Perner: A theory of implicit and explicit knowledge. Submitted to Behavioral and Brain Sciences.

Taatgen, N.A. (submitted). Cognitief Modelleren: Een nieuwe kijk op individuele verschillen. Submitted to Nederlands Tijdschrift voor de Psychologie.

Taatgen, N.A. & Wallach, D. P. Wallach (in preparation). Model of rule and instance-based skill acquisition in complex task domains.

References

Akyürek, A. (1992). On a computational model of planning. In J. A. Michon & A. Akyürek (Eds.), *Soar: A cognitive architecture in perspective* (pp. 81-108). Dordrecht, The Netherlands: Kluwer.

Anderson, J. R. (1976). *Language, memory, and thought*. Hillsdale, NJ: Erlbaum.

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review*, *89*, 369-406.

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard university press.

Anderson, J. R. (1990). *The adaptive character of thought*. Hillsdale, NJ: Lawrence Erlbaum.

Anderson, J. R. (1993). *Rules of the Mind*. Hillsdale, NJ: Lawrence Erlbaum.

Anderson, J. R. (1995). *Learning and memory*. New York: Wiley.

Anderson, J. R., Bothell, D., Lebiere, C., & Matessa, M. (1998). List memory. In J. R. Anderson & C. Lebiere (Eds.), *The atomic components of thought* (pp. 201-254). Mahwah, NJ: Erlbaum.

Anderson, J. R. & Fincham, J. M. (1994). Acquisition of procedural skills from examples. *Journal of experimental psychology: learning, memory, and cognition*, *20*(6), 1322-1340.

Anderson, J. R., Fincham, J. M., & Douglass, S. (1997). The role of examples and rules in the acquisition of a cognitive skill. *Journal of experimental psychology: learning, memory, and cognition*, *23*(4), 932-945.

Anderson, J. R., Fincham, J. M., & Douglass, S. (submitted). Practice and Retention: A Unifying Analysis.

Anderson, J. R., Kushmerick, N., & Lebiere, C. (1993). Navigation and conflict resolution. In J. R. Anderson (Eds.), *Rules of the Mind* (pp. 93-116). Hillsdale, NJ: Erlbaum.

Anderson, J. R. & Lebiere, C. (1998). *The atomic components of thought*. Mahwah, NJ: Erlbaum.

Ashcraft, M. H. (1987). Children's knowledge of simple arithmetic: a developmental model and simulation. In J. Bisanz, C. J. Brainerd, & R. Kail (Eds.), *Formal methods in developmental pyschology* New York: Springer verlag.

Atkinson, R. C. & Shiffrin, R. M. (1968). Human memory: A proposed system and its control processes. In K. W. Spence & J. T. Spence (Eds.), *The psychology of learning and motivation* New York: Academic Press.

Baddeley, A. D. (1986). *Working Memory*. Oxford: Oxford university press.

Barton, G. E., Berwick, R. C., & Ristad, E. S. (1987). *Computational complexity and natural language.* Cambridge, MA: MIT Press.

Berger, J. O. (1985). *Statistical decision theory and Bayesian statistics*. New York: Springer-Verlag.

Berry, D. C. (1997). Introduction. In D. C. Berry (Eds.), *How implicit is implicit learning?* Oxford: Oxford university press.

Berry, D. C. & Broadbent, D. A. (1984). On the relationship between task performance and associated verbalisable knowledge. *Quarterly journal of experimental psychology*, *36*, 209-231.

Berry, D. C. & Broadbent, D. E. (1995). Implicit learning in the control of complex systems. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European perspective* (pp. 131-150). Hillsdale, NJ: Erlbaum.

Binet, A. (1962). The nature and measurement of intelligence. In L. Postman (Eds.), *Psychology in the making: histories of selected research programs* New York: Knopf.

Blessing, S. B. & Anderson, J. R. (1996). How people learn to skip steps. *Journal of experimental psychology: learning, memory, and cognition*, *22*(3), 576-598.

Broadbent, D. (1989). Lasting representations and temporary processes. In H. L. Roediger & F. I. M. Craik (Eds.) *Varieties of memory and consciousness: Essays in honour of Endel Tulving* (pp. 211-227). Hillsdale, NJ: Erlbaum.

Bryk, A. S. & Raudenbush, S. W. (1992). *Hierarchical linear models: applications and data analysis methods*. Newbury Park, CA: Sage publishers.

Buchner, A. (1994). Indirect effects of synthetic grammar learning in an identification task. *Journal of experimental psychology: learning, memory, and cognition*, *20*(3), 550-566.

Buchner, A., Funke, J., & Berry, D. C. (1995). Negative correlations between control performance and verbalizable knowledge: Indicators for implicit learning in process control tasks? *Quarterly journal of experimental psychology: Human experimental psychology*, *48*(1), 166-187.

Byrne, R. (1977). Planning meals: Problem-solving on a real data-base. *Cognition*, *5*, 287-332.

Carbonell, J. G. (1990). Introduction: paradigms for machine learning. In J. G. Carbonell (Eds.), *Machine learning, paradigms and methods* (pp. 1-9). Cambridge, MA: MIT Press.

Carpenter, G. A. & Grossberg, S. (1991). *Pattern recognition by self-organizing neural networks*. Cambridge, MA: MIT Press.

Chomsky, N. & Miller, G. (1963). Introduction to the formal analysis of natural languages. In R. Luce, R. Bush, & E. Galanter (Eds.), *Handbook of mathematical psychology* Wiley.

Cleeremans, A. (1997). Principles for implicit learning. In D. C. Berry (Eds.), *How implicit is implicit learning?* (pp. 195-234). Oxford, England: Oxford university press.

Cleeremans, A., Destrebecqz, A., & Boyer, M. (1998). Implicit learning: news from the front. *Trends in cognitive sciences*, *2*(10), 406-416.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *3rd Ann. ACM Symp. on Theory of Computing* (pp. 151-158). New York: Association for Computing Machinery.

Craik, F. I. M. (1970). The fate of primary memory items in free recall. *Journal of verbal learning and verbal behavior*, *9*, 143-148.

Craik, F. I. M. & Lockhart, R. S. (1972). Levels of processing: a framework for memory research. *Journal of verbal learning and verbal behavior*, *11*, 671-684.

Cuvo, A. J. (1975). Developmental differences in rehearsal and free recall. *Journal of experimental child psychology*, *19*(2), 265-278.

Davidson, J. E. (1995). The suddenness of insight. In R. J. Sternberg & J. E. Davidson (Eds.), *The nature of insight* (pp. 125-155). Cambridge, MA: MIT Press.

Dienes, Z. & Fahey, R. (1995). Role of specific instances in controlling a dynamic system. *Journal of experimental psychology: learning, memory, and cognition*, *21*(4), 848-862.

Elman, J. (1993). Learning and development in neural networks: the importance of starting small. *Cognition*, *48*, 71-99.

Elman, J. L., Bates, E. A., Johnson, M. H., Karmiloff-Smith, A., Parisi, D., & Plunkett, K. (1996). *Rethinking innateness. A connectionist perspective on development*. Cambridge, MA: MIT Press.

Ericsson, K. A. & Simon, H. A. (1984). *Protocol analysis. Verbal reports as data.* Cambridge, MA: The MIT Press.

Fischer, K. W. (1980). A theory of cognitive development: the control and construction of hierarchies of skills. *Psychological Review*, *87*(6), 477-531.

Fischer, K. W. & Ayoub, C. (1994). Affective splitting and dissociation in normal and maltreated children: developmental pathways for self in relationships. In D. Cicchetti & S. L. Toth (Eds.), *Disorders and dysfunctions of the self* (pp. 149-222). Rochester, NY: University of Rochester Press.

Fischler, I., Rundus, D., & Atkinson, R. C. (1970). Effects of overt rehearsal procedures on free recall. *Psychonomic Science*, *19*(4), 249-250.

Fitts, P. M. (1964). Perceptual-motor skill learning. In A. W. Melton (Eds.), *Categories of human learning* New York: Academic Press.

Garey, M. R. & Johnson, D. S. (1979). *Computers and intractibility, a guide to the theory if NP-completeness*. San Fransisco, CA: Freeman.

Hagen, J. W. & Kail, R. V. (1973). Facilitation and distraction in short-term memory. *Child development*, *44*, 831-836.

Hahn, U. & Chater, N. (1998). Similarity in rules: distinct? exhaustive? empirically distinguishable? *Cognition*, *65*, 197-230.

Harrow, M. & Friedman, G. B. (1958). Comparing reversal and nonreversal shifts in concept formation with partial reinforcement control. *Journal of experimental psychology*, *55*, 592-598.

Hayes-Roth, B. & Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science*, *3*, 275-310.

Heindel, W. C., Butters, N., & Salmon, D. P. (1988). Impaired learning of a motor skill in patients with Huntington's disease. *Behavioural Neuroscience*, *102*, 141-147.

Hofstadter, D. R. (1979). *Gödel, Escher, Bach: An eternal golden braid*. New York: Basic Books.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press.

Holland, J. H., Holyoak, K. J., Nisbett, R. E., & Thagard, P. R. (1986). *Induction: Processes of inference, learning, and discovery*. Menlo Park, CA: Addison-Wesley.

Johnson, T. R. (1997). Control in ACT-R and Soar. In M. G. Shafto & P. Langley (Ed.), *Nineteenth annual conference of the cognitive science society* (pp. 343-348). Stanford: Erlbaum.

Jongman, L. (1997). *Zoeken in informatiesystemen.* University of Groningen, the Netherlands.

Just, M. A. & Carpenter, P. A. (1992). A capacity theory of comprehension: individual differences in working memory. *Psychological Review*, *1*, 122-149.

Karmiloff-Smith, A. (1992). *Beyond modularity. A developmental perspective on cognitive science*. Cambridge, MA: MIT-Press.

Kelleher, R. T. (1956). Discrimination learning as a function of reversal and nonreversal shifts. *Journal of experimental psychology*, *51*(6), 379-384.

Kendler, T. S. & Kendler, H. H. (1959). Reversal and nonreversal shifts in kindergarten children. *Journal of experimental psychology*, *58*, 56-60.

Kitchener, K. S., Lynch, C. L., Fischer, K. W., & Wood, P. K. (1993). Developmental range of reflective judgment: the effect of contextual support and practice on developmental stage. *Developmental psychology*, *29*(3), 893-906.

Knoblich, G. & Ohlsson, S. (1996). Can ACT-R have insights? In U. Schmid, J. Krems, & F. Wysotzki (Ed.), *First European Workshop on Cognitive Modeling* (pp. 161-169). Berlin: TU Berlin, fachbereich informatik.

Kuipers, T. A. F. & Mackor, A. R. (1995). *Cognitive patterns in science and common sense*. Amsterdam: Rodopi.

Kyllonen, P. C. & Christal, R. E. (1990). Reasoning ability is (little more than) working-memory capacity. *Intelligence*, *14*(4), 389-433.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, *33*, 1-64.

Lakatos, I. (1970). Falsification and the methodology of scientific research programmes. In I. Lakatos & A. Musgrave (Eds.), *Criticism and the growth of knowledge* (pp. 91-196). Cambridge, England: Cambridge university press.

Lebiere, C. & Anderson, J. R. (1993). A connectionist implementation of the ACT-R production system. In *Fifteenth Annual Conference of the Cognitive Science Society* (pp. 635-640). Erlbaum.

Lehman, J. F., Lewis, R., Newell, A., & Pelton, G. (1991). NL-Soar as a cognitive model. In *EuroSoar Workshop-4* . Enschede, the Netherlands:

Logan, G. D. (1988). Toward an instance theory of automization. *Psychological Review*, *22*, 1-35.

Logan, G. D. (1990). Repetition priming and automaticity: Common underlying mechanisms? *Cognitive Psychology*, *22*(1), 1-35.

Lovett, M. C., Reder, L. M., & Lebiere, C. (1997). Modeling individual differences in a digit working memory task. In M. G. Shafto & P. Langley (Eds.), *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society* (pp. 460-465). Hillsdale, NJ: Erlbaum.

Mayer, R. E. (1983). *Thinking, problem solving, cognition*. New York: Freeman.

McGeorge, P., Crawford, J. R., & Kelly, S. W. (1997). The relationships between psychometric intelligence and learning in an explicit and an implicit task. *Journal of experimental psychology: learning, memory, and cognition*, *23*(1), 239-245.

Meyer, D. E. & Kieras, D. E. (1997). A computational theory of executive cognitive processes and multiple-task performance: I. Basic mechanisms. *Psychological Review*, *104*(1), 3-65.

Michon, J. A. & Akyürek, A. (Ed.). (1992). *Soar: A cognitive architecture in pespective*. Dordrecht, the Netherlands: Kluwer.

Morris, P. & Gruneberg, M. (1994). *Theoretical aspects of memory* (2 ed.). London: Routledge.

Mulder, G., Lamain, W., & Passchier, P. (1992). De cognitieve interface. In R.J. Jorna & J.L. Simons (Eds.), *Kennis in organizaties: toepassingen en theorie van kennissystemen*. Muiderberg, the Netherlands: Coutinho.

Murdock, B. B. (1962). The serial position effect of free recall. *Journal of experimental psychology*, *64*(5), 482-488.

Newell, A. (1973). You can't play 20 questions with nature and win. In W. G. Chase (Eds.), *Visual information processing* New York: Academic Press.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard university press.

Newell, A., Rosenbloom, P. S., & Laird, J. E. (1989). Symbolic architectures for cognition. In M. I. Posner (Eds.), *Foundations of cognitive science* (pp. 93-131). Cambridge, MA: MIT Press.

Newell, A. & Simon, H. (1963). GPS, a program that simulates human thought. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought* New York: McGraw-Hill.

Newell, A. & Simon, H. A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice-Hall.

Norman, D. A. (1988). The psychology of everyday things. New York: Basic Books.

Norman, D. A. (1993). *Things that make us smart*. Reading, MA: Addison-Wesley.

Numan, H., Pakes, F., Schuurman, J. G., & Taatgen, N. A. (1990). *Strategies of human planning: modelling the organisation of a symposium*. University of Groningen, the Netherlands.

Ohlsson, S. (1984). Restructuring revisited II. An information processing theory of restructuring and insight. *Scandinavian journal of psychology*, *25*, 117-129.

Perruchet, P. & Amorim, M. A. (1992). Conscious knowledge and changes in performance in sequence learning: Evidence against dissociation. *Journal of experimental psychology: learning, memory, and cognition*, *18*(4), 785-800.

Perruchet, P. & Pacteau, C. (1990). Synthetic grammar learning: Implicit rule abstraction or explicit fragmentary knowledge? *Journal of experimental psychology: General*, *119*(3), 264-275.

Piaget, J. (1952). *The origins of intelligence in children*. New York: International University Press.

Pinker, S. & Prince, A. (1988). On language and connectionism: Analysis of a distributed processing model of language acquisition. *Cognition*, *28*, 73-193.

Plunkett, K. & Marchman, V. (1991). U-shaped learning and frequency effects in a multi-layered perceptron: Implications for child language acquisition. *Cognition*, *38*, 43-102.

Popper, K. R. (1959). *The logic of scientific discovery*. New York: Basic Books.

Postman, L. & Phillips, L. W. (1965). Short-term temporal changes in free recall. *Quarterly journal of experimental psychology*, *17*, 132-138.

Raijmakers, M. E. J., Maas, H. v. d., & Molenaar, P. C. M. (1996). On the validity of simulating stagewise development by means of PDP-networks: application of catastrophe analysis and an experimental test for rule-like network performance. *Cognitive Science*, *20*, 101-136.

Rasbash, J. & Woodhouse, G. (1995). *MLn: Command reference*. London: Institute of Education, University of London.

Reber, A. S. (1967). Implicit learning of artificial grammars. *Journal of verbal learning and verbal behavior*, *5*, 855-863.

Redington, M. & Chater, N. (1996). Transfer in artificial grammar learning: a reevaluation. *Journal of experimental psychology: general*, *125*(2), 123-138.

Roediger, H. K. (1990). Implicit memory: retention without remembering. *American psychologist*, *45*, 1043-1056.

Rumelhart, D. E. & McClelland, J. L. (1986). *Parallel distributed processing: Explorations in the microstructure of cognition.* Cambridge, MA: MIT Press.

Rundus, D. (1971). Analysis of rehearsal processes in free recall. *Journal of experimental psychology*, *89*(1), 63-77.

Saint-Cyr, J. A., Taylor, A. E., & Lang, A. E. (1988). Procedural learning and neostriatal dysfunction in man. *Brain*, *111*, 941-959.

Sander, E. & Richard, J. F. (1997). Analogical transfer as guided by an abstraction process: The case of learning by doing in text editing. *Journal of experimental psychology: learning, memory, and cognition*, *23*(6), 1459-1483.

Schacter, D. L. (1987). Implicit memory: history and current status. *Journal of experimental psychology: learning, memory, and cognition*, *13*(3), 501-518.

Schank, R. C. & Abelson, R. (1977). *Scripts, plans, goals, and understanding*. Hillsdale, NJ: Erlbaum.

Shanks, D. R. & John, M. F. S. (1994). Characteristics of dissociable learning systems. *Behavioral and Brain Sciences*, *17*, 367-395.

Shastri, L. & Ajjanagadde, V. (1993). From simple associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioral and Brain Sciences, 16*(3), 417-494.

Siegler, R. S. (1981). Developmental sequences within and between concepts. *Monographs of the Society for Research in Child Development, 46* (2).

Siegler, R. S. (1996). *Emerging minds. The process of change in children's thinking.* New York: Oxford university press.

Squire, L. R. & Knowlton, B. J. (1995). Memory, hippocampus, and brain systems. In M. S. Gazzaniga (Eds.), *The cognitive neurosciences* (pp. 825-838). Cambridge, MA: MIT press.

Sternberg, S. (1969). Memory scanning: Mental processes revealed by reaction time experiments. *American Scientist, 57*, 421-457.

Svenson, O. & Sjoberg, K. (1983). Evolution of cognitive processes for solving simple additions during the first three school years. *Scandinavian journal of psychology, 24*, 117-124.

Taatgen, N. A. & Andringa, T. C. (1997). De computer. In P. Hendriks, N. A. Taatgen, & T. C. Anderinga (Eds.), *Breinmakers & Breinbrekers* (pp. 65-91). Amsterdam: Addison-Wesley Longman.

Tulving, E., Schacter, D. L., & Stark, H. A. (1982). Priming effects in word-fragment completion are independent of recognition memory. *Journal of experimental psychology: learning, memory, and cognition, 8*(4), 336-342.

Turing, A. M. (1936). On computabel numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc. (ser. 2)*(42), 230-265.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind, 59*, 433-460.

van den Berg, A. E. (1990). *Aspects of case-based reasoning in planning meals.* University of Groningen, the Netherlands.

van Geert, P. (1994). *Dynamic systems of development: Change between complexity and chaos.* London: Harvester Wheatsheaf.

van Geert, P. (1998). A dynamic systems model of basic developmental mechanisms: Piaget, Vygotsky, and beyond. *Pychological Review, 105*, 634-677.

van Someren, M. W., Barnard, Y. F., & Sandberg, J. A. C. (1994). *The think aloud method*. London: Academic Press.

Warrington, E. K. & Weiskrantz, L. (1970). Amnesia: consolidation or retrieval? *Nature, 228*, 628-630.

Willingham, D. B., Nisen, M. J., & Bullemer, P. (1989). On the development of procedural knowledge. *Journal of experimental psychology: learning, memory, and cognition, 15*, 1047-1060.

Yuill, N., Oakhill, J., & Parkin, A. (1989). Working memory, comprehension ability and the resolution of text anomaly. *British journal of psychology, 80*, 351-361.

Samenvatting

Hoe denken mensen?

Een van de nog grotendeels onbeantwoorde vragen in de wetenschap is hoe het komt dat mensen zo slim zijn als ze zijn. Hoe is het mogelijk dat mensen zich kunnen handhaven in alle ingewikkelde situaties die het dagelijks leven aan ze voorschotelt? Tegenwoordig wordt deze vraag bestudeerd door de cognitiewetenschap, een inter-disciplinaire wetenschap die voortgekomen is uit psychologie, filosofie, kunstmatige intelligentie, neurowetenschap en taalkunde. Elk van deze disciplines heeft een eigen beginpunt voor de beantwoording van deze vraag. In dit proefschrift staan de benaderingen van de psychologie en de kunstmatige intelligentie centraal, met hier en daar wat verwijzingen naar de andere drie.

De invalshoek van de kunstmatige intelligentie is die van het oplossen van complexe problemen. Aanname is, dat menselijke intelligentie hierin maximaal tot uitdrukking komt. De nadruk van het onderzoek naar de aard van de intelligente processen ligt hierbij vooral op het resultaat: het vinden van een computerprogramma dat een bepaald probleem zo efficient mogelijk kan oplossen. De kunstmatige intelligentie ziet denken als een combinatie van *zoeken* en *kennis*. Een probleem is als het ware een soort doolhof. Ergens in het doolhof ligt de oplossing van het probleem. Om deze oplossing te bereiken, moet telkens gekozen worden tussen verschillende paden. Dit is het zoek-aspect van probleemoplossen. De *kennis* die iemand over een probleem heeft bepaalt hoe het doolhof er precies uitziet. Iemand met weinig kennis moet door

een ingewikkeld doolhof, terwijl iemand met veel kennis slechts een eenvoudig doolhof hoeft te bewandelen. Dit komt doordat voor degene met veel kennis veel van de doodlopende wegen niet bestaan.

Als theorie van menselijke cognitie heeft de zoeken-en-kennis benadering een probleem. Aangezien mensen leren, is hun kennis niet statisch, maar continu aan verandering onderhevig. Het doolhof verandert als het ware terwijl iemand er door heen loopt. Op zich hoeft dit niet zo erg te zijn: de wetenschap kan niet alles tegelijk onderzoeken. Een mogelijkheid is, om eerst het zoeken-en-kennis aspect goed in kaart te brengen, en later het aspect van leren toe te voegen. Een voorbeeld van een dergelijke benadering is de benadering van de expert-systemen. Expert-systemen hebben de pretentie het redeneren van een expert na te bootsen. Aangezien experts mensen zijn met al heel veel kennis, is de aanname dat leren geen invloed meer heeft op het redeneren van een expert. Het tot nu toe geringe succes van expert-systemen doet de vraag rijzen of deze aanpak wel klopt. Is het wel zo dat experts niet meer leren, en dat menselijk redeneren los van leren bestudeerd kan worden? Het antwoord op deze vraag is ontkennend. Voor complexe problemen geldt, dat ze in het algemeen niet efficient oplosbaar zijn met een computerprogramma, hetgeen met behulp van de complexiteitstheorie kan worden aangetoond. Complexe problemen maken deel uit van het dagelijks leven: niet alleen het maken van ingewikkelde roosters is een complex probleem, maar ook het kunnen interpreteren van alledaagse taal. De conclusie is, dat redeneren en leren onlosmakelijk met elkaar verbonden zijn, en dat ze niet goed los te bestuderen zijn.

In de psychologie is het onderzoek naar leren lange tijd gedomineerd door experimenten, waarbij proefpersonen een lijst woorden of andere dingen moesten onthouden, die ze later in het experiment weer moesten reproduceren of herkennen. Hoewel dit belangrijke inzichten verschafte in de werking van het geheugen, presenteerde deze manier van onderzoek ook het beeld van een leerproces dat los staat van het redeneren. Pas de afgelopen vijftien jaar is er belangstelling onstaan voor leren tijdens het uitvoeren van een taak, het zogenaamde *impliciete leren*. In de experimenten die in het kader van dit soort onderzoek worden uitgevoerd verbetert de prestatie van proefpersonen zonder dat ze kunnen aangeven welke kennis ze hebben geleerd. Leren blijkt ook vanuit psychologisch oogpunt een continu proces te zijn, dat niet ophoudt als de leerfase voorbij is.

Een theorie over menselijk redeneren

Om tot een theorie over menselijk redeneren te kunnen komen, is meer nodig dan de zoeken-en-kennis theorie van de kunstmatige intelligentie. De ontbrekende component, het leren, moet geïntegreerd zijn in de theorie, en niet als component later toegevoegd zijn. Het onderzoek naar architecturen voor cognitie probeert zo'n geïntegreerde theorie te formuleren. Een architectuur voor cognitie is een theorie die tegelijk ook een simulatieprogramma is.

In dit proefschrift staat de ACT-R (Adaptive Control of Thought, Rational) architectuur centraal. Als theorie stelt het, dat menselijk denken een rationele grond heeft. Rationeel in de ACT-R visie is een soort economische rationaliteit: de hersenen zijn zo georganiseerd dat bij het maken van elke keuze een kosten-baten analyse gemaakt wordt. Welke keuze levert het meeste op bij zo min mogelijk risico en zo laag mogelijke kosten? Verder stelt de ACT-R theorie dat mensen over twee soorten geheugen beschikken: een geheugen voor feitenkennis (het z.g. declaratief geheugen) en een geheugen voor procedures (het z.g. procedureel geheugen), met kennis over hoe dingen gedaan moeten worden. De inhoud van deze geheugens is continu aan verandering onderhevig: niet alleen worden er dingen toegevoegd, maar ook wordt de waardering en ordening van de kennis telkens veranderd onder invloed van ervaring en context.

Behalve een theorie is ACT-R ook een simulatieprogramma. Voor een gegeven psychologische taak is het mogelijk beginkennis in de simulatie te brengen, en het programma voorspellingen te laten maken over menselijk gedrag. In die zin is het dus een theorie die niet alleen achteraf verklaringen kan geven voor bepaalde verschijnselen, maar ook voorspellingen kan doen. Wat we dus eigenlijk doen is aspecten van menselijk denken nabootsen op een computer, om ze zo beter te begrijpen. Hierbij moet niet meteen gedacht worden aan een "denkende computer" maar meer aan een programma dat voorspellingen kan doen over hoe snel mensen dingen kunnen doen, wat voor soort fouten ze maken, en welke keuzes in welke omstandigheden worden gemaakt. Voor een echt denkende computer zou het nodig zijn om alle kennis van een bepaald individu in ACT-R te brengen, niet alleen feitenkennis, maar ook kennis over hoe je dingen moet doen (procedures).

In praktijk maken we zogenaamde modellen in ACT-R, hetgeen simulaties zijn van hoe mensen zich gedragen in een specifieke context, namelijk de context van een bepaald psychologisch experiment. Dit proefschrift bevat een groot aantal voorbeelden van dergelijke modellen. Een voorbeeld van zo'n taak is "free-recall", waarbij proefpersonen een lijstje woorden moeten leren, waarvan ze er later zo veel mogelijk moeten herinneren. Het model van free-recall bevat kennis over het lezen van woorden, het mentaal repeteren van deze woorden, en het reproduceren ervan. Ook bevat

het uiteraard de woorden zelf. Dit model kan een aantal zaken verklaren die in experimenten gevonden zijn. Zo is bekend uit de literatuur dat de eerste woorden uit de lijst en de laatste woorden uit de lijst beter herinnerd worden dan de woorden uit het midden van de lijst. Dit worden het primacy-effect en het recency-effect genoemd. Beide effecten zijn normaal aanwezig, maar kunnen in bepaalde varianten van het experiment verdwijnen. Zo verdwijnt het primacy-effect als proefpersonen een bepaalde instructie krijgen over hoe ze woorden moeten repeteren. Het recency-effect verdwijnt als er tijd zit tussen het leren van de lijst en het reproduceren. Als deze tijd lang is, kan het recency effect zelfs negatief worden, wat betekent dat de laatste woorden uit de lijst zelfs slechter herinnerd worden. Het ACT-R model kan al deze verschijnselen nabootsen en verklaren.

Bovenstaand voorbeeld illustreert een aantal belangrijke aspecten van een model. Het feit dat het model een lijstje woorden kan reproduceren is op zich niet zo interessant. Waar het om gaat is dat de specifieke effecten die gevonden worden in het experiment, primacy en recency, voorspeld kunnen worden, en dat ook de omstandigheden kunnen worden nagebootst waarin beide effecten niet optreden.

Maar hoe zit het dan met echt complexe problemen?

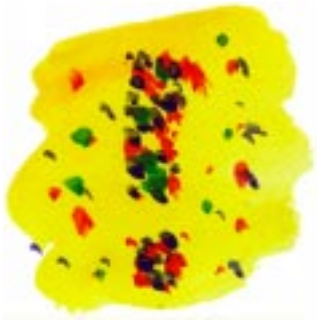
Bij het voorbeeld van free-recall mogen we aannemen dat mensen over de kennis beschikken die ze nodig hebben om het experiment te doen, zoals woorden lezen, ze mentaal repeteren en later reproduceren. Bij echt complexe problemen moeten mensen echter nog uitzoeken hoe deze in elkaar zitten, en wat de methoden zijn om ze op te lossen. Een voorbeeld van zo'n complex probleem is roosteren. Roosteren is een probleem dat volgens de complexiteitstheorie in zijn algemeenheid onoplosbaar is voor computers. Als mensen roosterproblemen moeten oplossen, dan moeten ze dit eerst leren. Om beter inzicht in dit proces te krijgen, heb ik een experiment gedaan waarin proefpersonen roosterproblemen moesten oplossen. Tijdens het oplossen moesten ze hardop denken. Bij het begin van het experiment krijgen ze instructies, maar deze zijn onvoldoende om het probleem op te lossen. Dus moeten proefpersonen een beroep doen op kennis die ze al hebben, en moeten ze door dingen te proberen het probleem in de vingers krijgen.

Uit het experiment bleek, dat proefpersonen soms heel erg vast kwamen te zitten in een probleem. Dit wordt wel een impasse genoemd. In een aantal gevallen kwamen proefpersonen vervolgens met een nieuwe idee, een nieuw inzicht om het probleem op te lossen. Ook bleek dat als proefpersonen zich eenmaal zo'n inzicht hadden verworven, ze dit bij latere problemen ook gingen toepassen. Behalve dit verwerven van inzichten zijn er nog een aantal interessante verschijnselen uit het experiment te

halen. Zo moest in één versie van het experiment alles uit het hoofd gedaan worden, met als gevolg dat proefpersonen hun aandacht moesten verdelen over enerzijds het repeteren van de al gevonden oplossing, en anderzijds het redeneren hierover.

Al met al wijst het roosterexperiment erop dat er een breed scala van leerverschijnselen plaatsvindt bij het verwerven van een complexe vaardigheid. Proefpersonen interpreteren instructies en stippelen op grond hiervan een strategie uit, tijdens het experiment ontdekken ze nieuwe strategieën, en ook worden ze beter in de meer elementaire stappen van het proces, zoals de coördinatie van het repeteren van deeloplossingen. De conclusie moet dus zijn, dat prestatieverbeteringen niet aan één leereffect toe te schrijven zijn, maar aan een aantal. Om beter inzicht te krijgen in de leereffecten die een rol spelen, is het nuttig om deze effecten afzonderlijk te bestuderen.

Impliciet en expliciet leren



In de psychologische literatuur wordt vaak het onderscheid gemaakt tussen impliciet en expliciet leren. Impliciet leren is het "leren door te doen". Zonder actief bezig zijn met het leerproces, en vaak ook onbewust, worden mensen beter in het uitvoeren van een taak. Bij expliciet leren daarentegen zijn mensen actief bezig met iets te leren, zoals het leren van een lijst woorden, of het uitdenken van een strategie voor een moeilijk probleem. Psychologische experimenten hebben uitgewezen dat er interessante verschillen zijn tussen beide soorten leren. Zo is impliciet leren veel robuuster dan expliciet leren: de geleerde kennis wordt minder snel vergeten, zowel kinderen, ouderen, en minder intelligente individuen zijn even goed in impliciet leren als ieder ander. Zelfs mensen die leiden aan geheugenverlies (amnesie) kunnen nog wel impliciet leren. Bij expliciet leren zijn er wel grote verschillen tussen individuen, en de geleerde kennis wordt soms weer snel vergeten. Op grond van deze verschillen wordt door een aantal onderzoekers aangenomen dat impliciet en expliciet leren daarom plaatsvindt in verschillende hersendelen. Dit wordt de systems-theorie genoemd.

Hoewel deze systems-theorie vrij aannemelijk klinkt, kent hij toch een aantal problemen. Zo kan deze theorie niet goed verklaren waarom impliciet leren zoveel robuuster is dan expliciet leren, dus waarom het eigenlijk bijna nooit voorkomt dat mensen nog wel expliciet kunnen leren, maar niet impliciet. Ook kan de theorie niet verklaren, waarom er zoveel individuele verschillen zijn in expliciet leren, maar niet in impliciet leren. In dit proefschrift behandel ik een alternatieve theorie. Eerst laat ik door middel van een model zien, dat het verschil tussen impliciet en expliciet leren ook prima met behulp van één geheugensysteem verklaard kan wordt, namelijk het declaratief geheugen van ACT-R. Het verschil tussen impliciet en expliciet leren in

dit model is, dat impliciet leren verklaard kan worden vanuit de leermechanismen die in de ACT-R architectuur verankerd zijn. Je zou ze dus kunnen vergelijken met de leerprocessen die deel uitmaken van onze hersenen. Expliciet leren kent echter specifieke leerdoelen. Een eenvoudig voorbeeld van een leerdoel is het onthouden van een lijstje woorden door ze te repeteren. Door dit repeteren zet je als het ware de impliciete leermechanismen van de hersenen aan het werk, net als wanneer je door te trainen je spieren sterker maakt. Leerdoelen kunnen ook complexer zijn, zoals het bedenken van een nieuwe strategie voor een complex probleem. Al deze leerdoelen hebben als gemeenschappelijk kenmerk, dat je kennis nodig hebt om ze te kunnen vervullen. Het blijkt bijvoorbeeld, dat kinderen niet of anders lijsten repeteren dan volwassenen. Aangezien expliciet leren gebaseerd is op kennis, zijn er daarom ook veel grotere verschillen tussen individuen. Ook is het daarom minder robuust: als de kennis om leerdoelen uit te voeren vergeten wordt, of door hersenbeschadiging verloren is gegaan, werkt het expliciet leren niet meer, terwijl de basiseigenschappen van de hersenen die ten grondslag liggen aan impliciet leren niet zomaar veranderen.

Hoe werken dan die leerstrategieën, en hoe komen we eraan?

Aangezien impliciet leren al onderdeel is van de architectuur, is het met name interessant om te kijken naar het expliciete leren, waarvoor kennis nodig is. Hoe ziet zo'n leerstrategie er in praktijk uit? En wanneer is het nuttig om zo'n strategie toe te gaan passen? En hoe leren we de leerstrategieën zelf? Om een beter inzicht in deze materie te krijgen is het nuttig om naar de cognitieve ontwikkeling van kinderen te kijken. Immers, als expliciet leren bestaat uit aangeleerde kennis, hebben jonge kinderen minder van deze kennis dan oudere kinderen en volwassenen. Een aantal theorieën over ontwikkeling lijken deze visie te ondersteunen. Jonge kinderen hebben grote moeite om om te gaan met abstracte begrippen. Volgens de theorie van Fischer, bijvoorbeeld, gaan kinderen door een aantal stadia waarin ze over steeds complexere structuren en concepten kunnen redeneren. Een voorbeeld hiervan is dat jonge kinderen nog niet kunnen redeneren over een concept als "rood". Ze kunnen wel zien dat een bepaald object rood is, maar ze kunnen nog niet over "rood" als abstract begrip, dus los van een object, redeneren. Gevolg hiervan is, dat ze zich anders gedragen dan volwassen in een experiment waarbij zowel abstraheren als niet-abstraheren tot een oplossing leidt (het z.g. discrimination-shift experiment). In ACT-R is dit goed te modelleren door twee leerstrategieën te definiëren. Met beide leerstrategieën gedraagt het model zich als een volwassene, terwijl het zich met een van beide leerstrategieën als een kind gedraagt.

Wanneer gebruiken we een leerstrategie eigenlijk? Uiteraard zijn we niet de hele dag bezig met het stellen van leerdoelen. Deze moeten alleen gesteld worden op momenten dat dat nodig is. Een mogelijke theorie hierover is die van meta-cognitie. Deze theorie veronderstelt dat we een "gewoon" redeneersysteem hebben en een systeem dat over het gewone redeneersysteem waakt, het meta-systeem. Het meta-systeem houdt het gewone systeem in de gaten en grijpt in wanneer dat nodig is. Het meta-systeem kan dus bijvoorbeeld een leerdoel stellen op het moment dat het gewone redeneersysteem vast loopt.

De meta-cognitietheorie heeft echter een aantal problemen. Het toevoegen van een extra systeem aan een theorie is wetenschappelijk gezien nooit zo'n aantrekkelijke optie, omdat het de theorie zwakker maakt. Maar ook is het de vraag waardoor het meta-systeem dan gecontroleerd wordt. Een meta-meta-systeem? Gelukkig is een meta-systeem niet echt nodig, hetgeen te zien is aan de hand van een zogenaamd dynamisch-groeimodel dat gebaseerd is op de kosten-baten analyse uit ACT-R. Het centrale idee in dit model is dat er een competitie plaatsvindt tussen het denkproces dat gewoon de taak wil uitvoeren en het denkproces dat een leerdoel wil stellen. Beide processen hebben kosten en opbrengsten: taakuitvoeringsprocessen hebben meestal lage kosten en leiden meestal tot de oplossing van een probleem. Leerdoelen kosten meestal meer tijd op uit te voeren, en leiden slechts indirect tot een oplossing van een probleem. Normaal gesproken is het uitvoeren van de taak aantrekkelijker. Als dit proces echter regelmatig fout loopt, doordat de kennis niet klopt of het proces te omslachtig is, zal de kosten-baten analyse van de taakuitvoeringsprocessen niet meer zo gunstig uitvallen, en zullen de leerstrategieën de competitie winnen.

De vraag die blijft liggen is hoe leerstrategieën zelf geleerd worden. Leerstrategieën zelf zijn ook vaardigheden. Een mogelijkheid is dat leerstrategieën net zo geleerd worden als andere vaardigheden, maar dan op een langere tijdschaal. In dit proefschrift zal ik hiervan echter geen concrete voorbeelden behandelen.

De rol van het formuleren van regels en het onthouden van voorbeelden

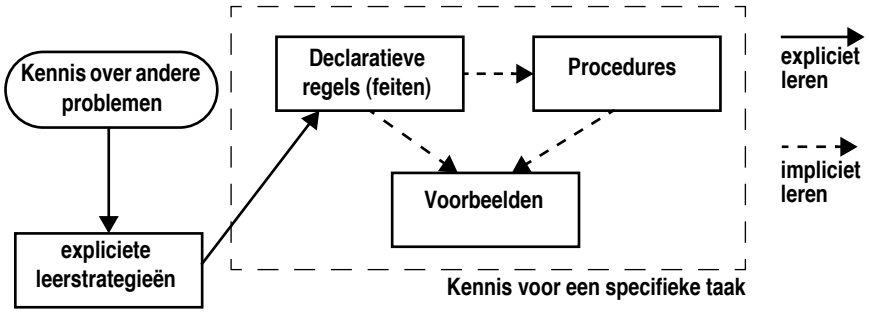
Wat is precies nieuwe kennis voor een nieuw probleem? Het gebruikelijke idee is, dat mensen algemeen geldige regels proberen af te leiden. Een algemene regel zou bijvoorbeeld kunnen zijn: "als je een voorwerp in de lucht houdt en het loslaat, dan valt het op de grond". Deze regel is algemeen, omdat deze voor elk voorwerp geldig is. Je zou een dergelijk regel kunnen afleiden op grond van het feit dat je al een aantal voorbeelden hebt gezien van voorwerpen die op de grond vallen, en nog (bijna) nooit een, die in de lucht blijft hangen. Er is echter ook een alternatieve theorie, die stelt dat we helemaal geen algemene regels leren, maar voornamelijk voorbeelden onthouden. Deze voorbeelden kunnen we gebruiken voor nieuwe

voorspellingen. Als je dus moet voorspellen of een bal valt als je hem loslaat, zou je je een voorbeeld kunnen herinneren van het vallen van een steen. Deze strategie van het onthouden van voorbeelden is zeer krachtig. Uit een experiment waarin proefpersonen een suikerfabriek moeten besturen blijkt bijvoorbeeld, dat hun gedrag volledig verklaard kan worden door het feit dat ze voorbeelden onthouden. Dit blijkt uit een ACT-R model van deze taak, dat uitsluitend voorbeelden onthoudt en weer terughaalt, en hetzelfde gedrag vertoont als proefpersonen. Een mogelijke verklaring voor het feit dat de prestaties zo aan voorbeelden toe te schrijven zijn, is dat de suikerfabriek zo in elkaar zit dat het bijna onmogelijk is om de echte regel achter de fabriek te ontdekken.

Om een betere afweging te maken tussen de regel- en de voorbeeldtheorie, is een taak nodig waarbij zowel het afleiden van regels als het onthouden van voorbeelden een mogelijke strategie is, en waarbij de keuze in de resultaten terug te vinden is. De Fincham-taak voldoet aan deze criteria. In deze taak krijgen proefpersonen een gebeurtenis en een tijdstip, en moeten ze voorspellen wanneer de gebeurtenis nog een keer plaatsvindt. Dit kan zowel geleerd worden door het onthouden van voorbeelden ("Als hockey of maandag is, dan is het de tweede keer op woensdag.") als het afleiden van regels. ("De tweede keer hockey is altijd twee dagen na de eerste keer.") Uit de resultaten van de experimenten met deze taak en het ACT-R model is af te leiden, dat mensen van beide strategieën gebruik maken, dus zowel van regels als van voorbeelden. De keuze van strategie hangt wederom af van een kosten-baten analyse: als een bepaald voorbeeld vaker voorkomt, zal de voorbeeld-strategie vaker worden gekozen. Voorbeelden worden echter ook weer snel vergeten, dus als er een dag tussen de testafnames zit, dan wordt het gedrag aan het begin van de volgende dag gedomineerd door de regel-strategie. Ook hier blijkt, net als bij de keuze tussen gewoon redeneren en het stellen van leerdoelen, dat de kosten-baten analyse die in ACT-R is ingebouwd een goede voorspeller is van gedrag.

Alle puzzelstukjes weer bij elkaar

De verschillende modellen van verschillende aspecten van leren kunnen worden samengebracht in één model, dat gebruikt kan worden voor een model van roosteren. Dit model is weliswaar nog niet zo slim als mensen in het oplossen van roosterproblemen, maar vertoont wel een groot aantal kenmerken die ook in menselijk leren worden aangetroffen. Zo probeert het model in eerste instantie om regels te formuleren die het probleem oplossen (zie figuur bovenaan de pagina). Deze regels worden in eerste instantie gewoon opgeslagen als feiten in het declaratief geheugen. Hierbij wordt gebruikt gemaakt van een expliciete leerstrategie, die probeert analogieën te vinden tussen de gezochte kennis en andere kennis waarover



het model al beschikt. Het model weet bijvoorbeeld niet zoveel af van roosters, maar wel van het maken van lijsten. Een mogelijkheid is dus om kennis over lijsten te gebruiken voor het maken van roosters. Declaratieve regels zijn heel flexibel, maar hebben als nadeel dat ze de volle aandacht vereisen om ze te kunnen gebruiken. Bovendien kunnen ze ook makkelijk weer vergeten worden. Daarom worden deze regels langzaam omgezet in procedures in het procedureel geheugen. Daardoor worden ze veel sneller, en worden minder snel fouten gemaakt. Naast regels onthoudt het model ook voorbeelden van hoe iets is opgelost, die later weer gebruikt kunnen worden. Tussen de bedrijven door moet het model ook nog de tussenresultaten repeteren die het tot dan toe heeft afgeleid.

Het moge duidelijk zijn dat er nogal wat mis kan gaan in dit proces, en dit gebeurt ook in het model. Het aardige hiervan is, dat de fouten die het model maakt, overeen komen met het soort fouten dat proefpersonen maken. Tevens is het vóórkomen van fouten een bron van individuele verschillen: sommige individuen hebben veel problemen met het onthouden van tussenresultaten, en andere veel minder. Deze verschillen kunnen gerelateerd worden aan verschillen in de capaciteit van het werkgeheugen. Het model laat echter zien, dat deze verschillen slechts tijdelijk zijn: na voldoende oefening zijn de prestaties van individuen met een klein werkgeheugen bijna net zo goed als die van individuen met een groot werkgeheugen.

Hoewel het model van roosteren nogal uitgebreid is, biedt het veel aanknopingspunten voor generalisatie. In tegenstelling tot veel cognitieve modellen is de kennis niet echt voorgeprogrammeerd, en slechts indirect beschikbaar in de vorm van feitenkennis. Het model leert dus zelf de benodigde regels. Door de feitenkennis van het model te veranderen wordt ook het gedrag veranderd, en kan hetzelfde model ook heel andere taken uitvoeren.

Conclusies

Menselijk leren van nieuwe vaardigheden is een complex cognitief fenomeen, dat niet met een enkelvoudige theorie in kaart te brengen is. Niettemin is het mogelijk op grond van de modellen uit dit proefschrift een redelijk consistent totaalbeeld te scheppen, dat goed aansluit op bestaande ideeën in de cognitiewetenschap. Ook maakt deze theorie bepaalde constructies overbodig, zoals meta-cognitie en aparte impliciete en expliciete geheugensystemen. Wel is duidelijk dat het menselijk leervermogen geen gesloten systeem is: het is altijd mogelijk nieuwe leerstrategieën te ontdekken of van iemand anders te leren. Het precies in kaart brengen van hoe leerstrategieën zelf geleerd worden is een uitdaging voor met name de ontwikkelingspsychologie.

De gepresenteerde theorie van het leren van nieuwe vaardigheden kent ook ruime mogelijkheden voor het in kaart brengen van individuele verschillen. Niet alleen verschillen individuen in beschikbare leerstrategieën, maar ook in eigenschappen die betrekking hebben op de onderliggende architectuur, zoals de capaciteit van het werkgeheugen. Het is daarom niet zo verwonderlijk dat elke proefpersoon complexe problemen als roosteren weer anders oplost.

Hoewel het onderzoek in dit proefschrift voornamelijk theoretisch van aard is, zijn er een aantal toepassingsvelden mogelijk. In de cognitieve ergonomie kan het inzicht van hoe nieuwe vaardigheden worden geleerd gebruikt worden in het ontwerp van computerprogramma's. Deze inzichten zijn om dezelfde reden van belang in het ontwikkelen van computer-ondersteund onderwijs.

Een geünificeerde theorie van leren?

In dit proefschrift heb ik een theorie van leren geschetst, die bestaat uit delen van bestaande theorieën, aangevuld met verbindende elementen die met name geïnspireerd zijn door de rationele theorie achter ACT-R. Vooralsnog bestaat deze theorie uit een algemeen verhaal en een verzameling computermodellen die deelaspecten illustreren. Het roostermodel levert daarnaast een voorbeeld van de theorie in zijn geheel. Een echte geünificeerde theorie van leren is echter nog wat concreter: daarvoor is een simulatieprogramma nodig dat alle elementen uit de theorie bevat. Een dergelijk programma zou een uitbreiding op ACT-R moeten zijn, en zou in staat moeten zijn uit instructies zelf de kennis te leren die nodig is voor het uitvoeren van een taak.Wellicht kan dit proefschrift een eerste stap zijn in de ontwikkeling van zo'n theorie.

Index