# The study of learning mechanisms in unified theories of cognition[1]

Niels A. Taatgen

University of Groningen
Department of Cognitive Science and Engineering
Grote Kruisstraat 2/1
9712 TS Groningen

January 27, 1993

**Abstract**

Unified theories of Cognition (UTCs) aim at unifying cognitive psychology. The core of such a theory is an architecture for cognition, an open structure containing a set of mechanisms that are the building blocks of the theory. Newell has formulated a list of constraints that a UTC must satisfy, some of which also influence the mechanisms of the architecture. Two constraints will be discussed: the constraint about learning and the constraint about time. Learning and time are intertwined, since learning intends to reduce the time needed for certain tasks. Newell's proposal for a UTC, Soar, has one learning mechanism called chunking. In this paper the chunking mechanism is explored to model learning strategies in Go, a chinese board game that has a high time-complexity, thus challenging the time constraint. Some experimental material about human subjects playing Go is discussed, and a framework for a model of solving Go problems is presented.

## 1. Introduction

The field of cognition is rich in self-contained collections of facts, models, and theories about problem solving, decision making, memory, perception, language, and so on. According to Newell (1991), unified theories of cognition can provide unification of all separate theories in cognitive psychology. Unification has always been a main goal of science because of its obvious advantages. A unifying theory avoids speech confusion, so a lot of disagreement between scientists can be resolved. It also offers clues for research, and, as a common framework, gives researchers in different fields the opportunity to use each other's work. A theory of cognition is a *unified theory of cognition* (UTC) in case it is built up on a set of cognitive mechanisms and it covers the whole field of cognition, that is, problem solving, memory, perception, routine action, imagining, and so on. These mechanisms are said to collectively determine an *architecture for cognition* (Anderson, 1983; Newell, 1991). An architecture for cognition is in principle suitable for implementation on a computer. Some architectures are in fact implemented wholly or partially, enabling simulation as a means of collecting scientific evidence.
Since the architecture specifies the building blocks, it is the core of the UTC. It should be emphasized that a UTC is not just a collection of existing cognitive theories, for in that case unifying elements would be difficult, if not impossible, to identify. Re-inventing the wheel is not the goal of a UTC either. Instead, it should absorb existing theories.
Absorbing existing theories is accomplished by redefining the essence of the existing theory in terms of the mechanisms of the architecture. If the resulting new theory, cast in terms of the mechanisms of the architecture, yields the same or better predictions than the old theory, the old theory has successfully been absorbed.

### 1.1 Criteria for Unified Theories

A UTC consists of two parts: the core mechanisms of the architecture and a large body of theory expressed in these mechanisms (including absorbed theories). In this way a UTC is a research program in the sense of Lakatos (Lakatos, 1970): there is a core of undisputed theory (the architecture), but the theories on the fringes are open to the scientific debate. So what kind of mechanisms should be incorporated in this core UTC? It is easy to define a set of mechanisms that has the power of universal computation, that is, if we accept the Church-Turing thesis, a set of mechanisms that can compute any intuitively computable function. But with such a set of mechanisms everything can be implemented, so the theoretical power of the UTC would be void. This means that the mechanisms of the architecture must be stronger: since hardly any constraints can be posed on the result, they should constrain the process of reaching the result. Mechanisms should model and implement common aspects of cognition, which recur in every cognitive domain.

(*** insert table 1 ***)

---

[1] I wish to thank G. Mulder and G. Renardel de Lavalette for their comments on this paper.

Newell (1991) has formulated a list of constraints, which an eventual UTC must satisfy (Table 1). They should not be used to judge the core UTC, but are rather the long-time goals of the research program. Some of the constraints, however, may have consequences for the core theory.

### 1.2 The learning constraint

The first item to examine is the learning constraint (7). Learning occurs during nearly every cognitive process. Some properties of learning recur in every context, for example the *Power Law of Practice*. This law states, that there is a negative exponential relation between learning time (practice) and task performance. This law is quite robust: it is valid for motor tasks as well as for perceptive and problem solving tasks.
Learning can be viewed as just a by-product of cognitive processes. In this view the first goal is to model performance on some task without bother about learning: a learning 'module' can be added later on. I will show in section 1.4 that in the domain of problem solving this approach does not work. And if we follow Newell's conjecture that all cognitive processes are a form of problem solving, learning is an essential mechanism, and an ideal candidate for the core UTC. The reason why learning is essential in problem solving is Newell's third constraint, the requirement that the system has to operate in real time.

### 1.3 The time constraint

The time constraint is not easy to judge. If a UTC is implemented on a slow computer, it may behave as expected, but just too slow. It appears impossible to disqualify a computer simulation of cognition if it differs from humans with respect to the time it takes to calculate something (Anderson, 1990). There is, however, another way to make time comparisons. Although we cannot use absolute time, we can use relative time or time complexity, a notion from complexity theory.
First we must define what we mean by the notion of a problem. A *problem* is answering yes or no to the question if some input belongs to the language that identifies the problem. So if the problem is to decide if a row of numbers is sorted or not, we have to answer the question if the row of numbers belongs to the set of all sorted rows (the language). An *instance* of a problem is the question posed by the problem for a particular input. So the question "is the row 1 5 8 13 sorted?" is an instance of the problem of deciding if rows are sorted. The solution of an instance is an answer: yes or no. The solution of a problem is a method or algorithm to compute the answer yes or no.
Although these definitions are rather technical, they enable us another look at time during problem solving. Instead of measuring the time an algorithm takes to compute the answer, a function is calculated, which maps the size of the instance of the problem on the time it takes to compute the answer. If the time it takes to compute the answer increases greatly if we only increase the size of the instance by a small amount, we have an inefficient algorithm. If we use these time complexity functions, we do have a criterium for judging the real time constraint. If the time complexity functions of a human subject and the simulation of the theory are the same, ignoring constants, the real time constraint has been satisfied.
Example: it is a well known fact that in a mental-rotation task the amount of time it takes to mentally rotate the image increases linearly when the amount of rotation is increased. So the time complexity function of mental rotation with respect to the amount of rotation is a linear function. This means, that if we have a theory of mental rotation that can be simulated on a computer, the resulting algorithm must also have a linear time complexity function.

### 1.4 The consequences of complexity theory for learning

I will now show why learning cannot be seen as a non-essential by-product of problem solving. Complexity theory (see for instance Garey & Johnson, 1979) can be used to assign classes to certain problems. For example, the class P consists of problems for which there is an algorithm with a polynomial time complexity. The P-class of problems is an important class: algorithms with a polynomial time complexity are acceptable, because they find solutions within reasonable time. Such problems are called *tractable*. Examples of problems in P are searching and sorting lists or finding the shortest path between two cities.
Problems outside P have exponential time complexity functions (or worse), which means that there is no algorithm that can solve all instances within reasonable time (algorithms for problems outside P often display behavior called 'combinatoric explosion'), and are therefore called *intractable*. A superset of P, called NP, contains such intractable problems[2]. Another interesting class, called NP-complete, is a subset of NP, and contains the toughest NP problems. The NP-complete class is considered to be outside of P. As a consequence, NP-complete problems are intractable. But the interesting thing about the NP-complete class is that many problems that occur in cognitive science are in NP-complete or related classes of problems. Nearly all non-trivial planning tasks, like the traveling-salesman problem, are NP-complete. Many problems in logic and mathematics are NP-complete. Many games are

---

[2]. In this discussion we will assume that P and NP are not equal, a hypothesis that is not proved, but generally considered true by computer scientists.

intractable. A majority of grammars proposed for natural language processing is NP-complete (Barton, Berwick, & Ristad, 1987).
So how is it possible to simulate human problem solving, when all kind of cognitive problems cannot be solved efficiently by computers, while humans miraculously cope with them?
To solve this problem, we must first observe that although there may be no general efficient solutions for a problem, there can be efficient solutions for subsets of instances. But there are many possible subsets of instances, yielding many possible algorithms. So the 'solution' of an NP-complete problem is no longer a clean algorithm, but rather a messy collection of algorithms, each working on subsets of instances. This collection differs over individuals or groups of individuals. For example, looking at grammars, instances of Dutch sentences belong to the subsets of interest for Dutch people, but not for English people (unless they want to learn Dutch). If we were to model problem solving without learning, there would be no way to handle intractable problems. This means that if we would view learning as just a by-product, we could only model problem solving of tractable problems, which is obviously insufficient.
So instead of looking at all the strategies humans employ at problem solving, it is more useful to look at how they acquire these strategies, not only satisfying Newell's seventh constraint (learning), but also his second (adaptive behavior).

**2. Soar**

Newell's proposal for a UTC is a system called Soar (Laird, Newell & Rosenbloom, 1987). One of the basic assumptions of Soar is that all intelligent behavior is a form of problem solving. So the architecture of Soar consists of mechanisms to do problem solving. Figure 1 shows Soar's general layout.

(*** insert figure 1 ***)

Knowledge in Soar is represented in a production system (corresponding to human long term memory) and problem solving takes place in working memory. Working memory is organized as a stack of problem spaces, where each problem space has a goal. Problem spaces lower in de stack represent subgoals of the problem spaces immediately higher in the stack. Each problem space has four 'context slots':

- the goal, representing the goal of the problem solving process in this problem space.
- the problem space, representing the kind of problem, which the system uses to activate the right knowledge in production memory.
- the state, representing the current state of the problem solving process.
- the operator, representing the operator that is currently being applied.

Productions are allowed to fire in parallel, but cannot modify context slots. Productions can only make preferences for context slots. After all productions have fired, a decision mechanism is invoked that examines all the preferences for a certain slot, and decides which symbol will be installed. If the decision mechanism cannot make a decision, e.g. if there are two or more operators with equal preferences, an impasse occurs. Resolving this impasse becomes a new subgoal in a problem space one step lower in the goal stack.
So for example, suppose the current problem space is a blocks world, in which the goal is to obtain a stack of blocks, consisting of a red block with a yellow block on top. The current state is that both blocks are on the table. In production memory there is a rule that says:

> If there is a block with nothing on top of it and your hand is empty, then grabbing the block is an acceptable operator. (P1)

In this example the rule fires twice: once for the yellow block and once for the red block. If there are no more rules that fire, we are left with two proposed operators: grab the red block and grab the yellow block. Both operators have an acceptable preference, so the decision mechanism reaches an impasse. So the system creates a subgoal to resolve this impasse. Soar's default problem space for resolving this particular impasse is the Selection space, which implements a kind of look-ahead search.
Another possibility is that Soar already has some search-control knowledge. Soar's production memory might contain a rule like

> If your goal is to stack two blocks, and both have nothing on top, then grabbing the one that must be on top gets a best preference. (P2)

With this rule, there would have been three rules that fired in the previous example, two giving acceptable preferences to grabbing the red and the yellow block, and one giving a best preference to grabbing the yellow block. In this case the decision mechanism can reach a decision, so no impasse occurs.

**2.1 Learning in Soar**

Soar's learning mechanism is called chunking, and is tied to the impasse-subgoal mechanism. Each time a subgoal is terminated, either with success or with failure, a new production, also called a chunk, is built. The conditional part of this production contains all parts that were present in working memory before the impasse that created the subgoal occurred. The action part contains all memory elements that were added to working memory outside the context of the subgoal. This means that if Soar reaches a similar state later on, no impasse will occur, but the newly built chunk will fire, and resolve the situation at once.
In the previous blocks world example, this would mean that if we only have production P1, we reach an impasse, resulting (if Soar's defaults rules are used) in a subgoal using the Selection problem space. This problem space will evaluate both operators (grabbing the yellow block and grabbing the red block) and will eventually conclude that grabbing the yellow block will reach the goal, while grabbing the red one will not. The subgoal will add a best preference to the grab-the-yellow-block operator and will terminate. At this moment the chunker will build a new production, with as conditions the context that caused the subgoal, and as action adding a best preference to an operator. This chunk will look like production P2.
Chunking is the only learning mechanism in Soar's architecture, so chunking must satisfy all constraints posed to it, including the challenge put to it by NP-complete problems. Chunking is, however, a low-level mechanism. It is just a dumb building block of the architecture, needing the right context to function properly. In the blocks world example the chunk could look like P2, but also like:

> If your goal is to stack a yellow block on a red block, and both have nothing on top, then grabbing the yellow block gets a best preference. (P3)

This rule is obviously too specific, since the colors of the blocks are of no importance. In this case, the representation of the state influences the quality of the generated chunks.
The learning mechanism of Soar does not in itself solve the problem of development of strategies when solving NP-complete problems. It is rather a starting point, one of the basic building blocks, which will be needed. When embedded in the right structure of problem spaces, it may display the desired behavior.

## 3. The Go experiment

To get some insight into how people handle problems with a high complexity class it useful to examine their behavior when working on instances of such problems[3]. The game of Go is a Chinese and Japanese board game, in which players alternate in placing a stone of their own color (white or black) on the board. The goal is to control as large as possible regions of the board. This can be accomplished by surrounding stones of the opponent. Go is played on a 19 x 19 board, but can also be played at boards of arbitrary size (beginners usually play at a 9 x 9 board). In what is called the generalized Go problem the input consists of an arbitrarily-sized Go board with a certain configuration of black and white stones. The question asked is whether black can win the game, if it is his turn to place a stone.
It has been proven by Lichtenstein and Sipser (1978), that generalized Go is PSPACE-hard. PSPACE-hard is a class definition that will not be explained here in detail. It has one important property, which it shares with NP-complete problems: for every NP problem and every PSPACE-hard problem, there is a function of polynomial time-complexity which transforms the input of the NP problem to input for the PSPACE-hard problem. This means that any instance of an NP problem can be expressed as an instance of a Go problem.
Go is quite suitable to study novice human behavior. Although it has a high complexity class, the rules are quite simple to explain. Moreover, since Go is not played by many people in Europe, it is easy to get subjects that are complete novices.

### 3.1 Method

Eighteen instances of the Go problem were offered to subjects on a Macintosh computer using the program HyperCard. Every instance was displayed graphically on the screen. The subject could make moves by clicking on the board with a mouse, either on the place at the board where he wanted to make his move or at a button to take a move back or start again. All subjects had never played Go before. Subjects were asked to think aloud during problem solving, which was recorded on audio-tape, and to use the mouse to point at regions of the board they were discussing. The program recorded all moves and mouse movements. Synchronization of both sources, audio-tape and computer record, resulted in a protocol like in Figure 2.

(*** insert figure 2 and 3 ***)

To be able to study the development of strategies for subsets of instances, two of these subsets were hidden in the eighteen instances. Three instances could be classified as a 'Fork'-problem and five instances as an 'Eye'-problem.

---

[3] This experiment is fully described in (Taatgen, 1991).

Figure 3 shows the prototype and an example of each. The example in Figure 2 is an 'Eye'-problem. The remaining ten instances were arbitrary Go-problems.

### 3.2 Results

The goal of this experiment was to look if and how people learn from experience when solving Eye and Fork problems. Although Go is PSPACE-hard in general, both the Eye and Fork sub-problems are in P. Solving the Fork problem is a form of pattern recognition: once the problem is recognized, the same move always applies. The rule to be learned is like:

> If the instance is recognized as an instance of the Fork problem, choose the point diagonal to the stone to be captured.

Of four subjects, three appeared to have learned this rule when trying to solve the second Fork instance. While they try all kinds of alternative moves in the first instance, they immediately recognize the situation in the second instance, and choose the right move. They do, however, not explicitly mention the rule in their think-aloud protocol.
The Eye problem is more difficult: the problem must be recognized, and a rule must be invented like:

> If the instance is recognized as an instance of the Eye problem, choose the point inside the white area that has most degrees of freedom left.

Two of the four subjects had invented a rule like this, enabling them to solve the Eye problem increasingly faster. One of the subjects didn't solve any of the Eye instances at all, and the fourth subject only solved a few of them, but didn't show any improvement by learning.
The following pieces of protocol show some mastering of the rule:

(Subject 2 at the fourth Eye instance)

> I hate ones like this... and wins the white chain. Yes, these are stupid problems. At least I have to see to it that I am surrounded as late as possible or something like that. That seems to be a nice... rule.

(Subject 4 at the fourth Eye instance)

> Now, these are again those horrible ones. Now the thing has five holes in it, and it is really important where to place the first stone. You have got five choices (...). I don't think that I should move where I can get captured at once. (...) So let's take f1, I can always go two ways, no let's take f2, then I have even more ways to go.

The concern of the subjects not to get captured or surrounded directly corresponds to have as much degrees of freedom as possible.
Another interesting observation that could be made, was that subjects hardly ever worked without some plan or strategy. This is of course quite necessary in Go, because evaluating all moves on a mostly empty 9 x 9 board is too much work. This plan or strategy could be as vague as 'Try to corner the opponent', or already a clear plan, in case an instance was presented that was similar to an earlier instance (like the Eye and Fork problems). The general schema that could be seen in most of the problem solving is:

1. A strategy is selected.
2. Using the selected strategy, possible moves fitting in this strategy are proposed (possibly just one move)
3. If more than one move is proposed, the proposed moves are evaluated, using look-ahead search, or by using evaluation rules suggested by the selected strategy.
4. The move with the best evaluation is executed.

## 4. Modelling Go in Soar

From the four steps in solving Go problems, steps 2 to 4 can be handled by Soar's basic problem solving mechanisms. The main problem is the representation of strategies, and a learning system that creates new and updates old strategies.
Soar's knowledge is organized in problem spaces. So a possible way to represent strategies is by assigning a problem space to each of them. We would also need a problem space to decide which strategy to apply.
As a result, when Soar is offered an instance of a Go-problem, a problem space is instantiated to handle it. We will call this problem space Go-top. Go-top decides which strategy must be used, and selects the appropriate subgoal, which in our experiment might be called Eye or Fork. In the problem space which is instantiated to this subgoal operators for moves are proposed and evaluated, leading eventually to the selection of a move.
This structure does not yet account for learning. Initially, the novice Go-player has no strategies, only some form of representation of the rules of the game. We can only start with a problem space with rules that describe possible

legal moves, and rules that describe the consequences of a move. Also we may suppose there are some basic search mechanisms, implementing some sort of look-ahead search. We will call the problem space containing the basic Go-rules and mechanisms to implement look-ahead search Go-basic.

(*** insert figure 4 ***)

Soar is capable of creating new problem spaces. However, these new problem spaces are initially empty. Empty problem spaces can be filled by chunking from subgoals. So the learning process runs as follows:

1. An instance of a Go problem is offered to Soar.
2. The Go-top problem-space is instantiated, which tries to find a suitable strategy. Since there are no strategies yet, an impasse is reached, resulting in the creation of a new, empty problem-space, called, say Go-S1.
3. Go-S1 is instantiated as a sub-goal of Go-top. Since there are no rules yet in this problem space, a No Change impasse is reached. As a sub-goal for this impasse Go-basic is instantiated.
4. Go-basic selects through look-ahead search some move, and copies this move to its super-goal: Go-S1. After this, Go-basic terminates, and we are back in Go-S1. At this point, a chunk is built in Go-S1, which summarizes the search in Go-basic. The next time Go-S1 will be invoked, it will not be empty, but will contain one rule. If this rule than applies, it will fire, else an impasse is reached, and Go-basic used to select a move.

Figure 4 shows the layout of the problem spaces. Of course, many details in this process still need to be filled in, before an implementation can be made.

**5. Conclusions**

The approach in the previous section shows a way to implement the development of strategies for Go in the Soar architecture. It may apply also to other problems in high-complexity classes, because no assumptions were made about the specific problem. It is therefore necessary to collect more experimental data about other sub-problems of Go and about other problems with a high-complexity class. It is also necessary to fill in all the details of the model, so that it can be implemented.

**References**


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.

Barton, E. B., Berwick, R. C., & Ristad, E. S. (1987). *Computational complexity and natural language*. Cambridge, MA: MIT Press.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability, a guide to the theory of NP-completeness*. San Francisco, CA: Freeman.

Lakatos, I. (1970). Falsification and the methodology of scientific research programmes. In I. Lakatos & A. Musgrave (eds.), *Criticism and the growth of knowledge*. Cambridge, England: Cambridge university press.

Laird, J.E., Newell, A., Rosenbloom, P.S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33,* 1-64.

Lichtenstein, D. & Sipser, M. (1978). Go is Pspace hard. *Proc. 19th Ann. Symp. on Foundations of Computer Science*, Long Beach, CA: IEEE Computer Society, 48-54.

Newell, A. (1991). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Taatgen, N.A. (1991). Complexiteit-reducerende leermechanismen. University of Groningen, Groningen.

Captions

Table 1: Multiple constraints that shape mind according to Newell (1991).

Figure 1: The general layout of Soar

Figure 2: An example of the results of an 'Eye' problem.

Figure 3: Prototypes and examples of 'Eye' and 'Fork' problems.

Figure 4: Schematic representation of problem spaces for the Go-problem. Triangles represent problem spaces, while circles represent succesive states within a problem space. A line downwards to a new triangle means an impasse has occured, resulting in a subgoal with a new problem space.

1. Behave flexibly as a function of the environment.
2. Exhibit adaptive (rational, goal-oriented) behavior.
3. Operate in real time.
4. Operate in a rich, complex, detailed environment.
   a. Perceive an immense amount of changing detail.
   b. Use vast amounts of knowledge.
   c. Control a motor system of many degrees of freedom.
5. Use symbols and abstractions.
6. Use language, both natural and artificial.
7. Learn from the environment.
8. Acquire capabilities through development.
9. Live autonomously within a social community.
10. Exhibit awareness and a sense of self.
11. Be realizable as a neural system.
12. Arise through evolution.

Table 1