Chapter 5

THE ALIGNMENT STUDY

Introducing the simulation experiment

5.1 Introduction

The alignment study has been conducted in order to contrast two different agent types: a cognitive model that implements the learner characteristics in what is considered a cognitive plausible way (the Soar agent) and one that implements the learner characteristics in what is considered a cognitive implausible way (the Cohen agent). Both the task environment in which the agents operate and the tasks they have to perform, are mimicked as closely as possible. The alignment study is designed and conducted in such a way that the differences found in agent performance can be reduced, and related to functional and representational differences between the two agent types, i.e. are reduced to the architectural differences.

This chapter is structured as follows. Section 5.2 introduces and describes the simulation experiment and describes the differences and the similarities between the agent types contrasted. Section 5.3 discusses the experimental design of the alignment study. It explains the data used as the basis for a performance assessment. It discusses the two experimental conditions as well as how the two experimental versions are compared. The section closes by explaining how the results found will be related to the issue of cognitive plausibility. Section 5.4 explains how the alignment experiment has been set up and implemented using the Soar architecture.

5.2 The simulation experiment

This section introduces the simulation experiment as it has been described by Cohen (1992). The computer simulation experiment conducted by Cohen consists of:

1. An agent, capable of interacting with
2. A task environment in order to carry out
3. A task.
These three elements will be introduced and explained next.

5.2.1 The task

The task for the agents is to find an appropriate sequence of actions in order to move a simulated organization from some initial state into a predefined goal state. The graphical interface depicted in figure 5.1 presents the information available within the task environment and, hence, is all the information that the agents can include in their decision-making process. The interface represents the goal state, the current state, the actions that can be applied, the feedback that the task environment provides, and the number of state transitions. The latter number is strictly speaking not included within the agent decision-making process but is applied to make monitoring easier.

Figure 5.1. The goal state is modeled as desirable, A, B, C, D, and E represent actions that can be applied. State changes occur as changes in the detectable features. No state change occurs when an action is not applicable. The feedback indicates whether progress has been made.

The task modeling effort is based on the familiar state-space approach. The simulated organization is viewed as being in a certain state. After the passage of a discrete unit of time, it will be in some other state\(^1\) in the space of possible states. The organization’s movement among states is the result of actions it may take and of regularities governing the modeled situation, such as laws of physics and product demand.

The current state of the simulated organization is represented through the highlighted features as \{4 8 16 24\}. The actions that can be applied to the current state are A, B, C, D, and E. After applying an action, by pressing a button, one of two things can happen. One is that the current state of the organization changes, an event that will be reflected as changes in the detectable features (some highlighted numbers are turned off, others turn on). The second is that the current state does not change, i.e. the detectable features do not change. It means that the applied action is somehow not allowed, or simply does not work for that problem-solving state. Of course, such an event is also a source of information for the agent performing the task, as will be explained below.

\(^1\) Or perhaps, in the same state again
Some subset of the states is modeled as desirable. Moving to them rewards the organization, or achieves some of its goals and therefore will be called the goal state. When the features of the current state match those of the goal state, the task for the agent is accomplished, an event called goal state attainment.

The assumption of the Cohen simulation model is that “we do not deal with states directly, but rather with the features that states represents to us” (p. 176). In figure 5.1, the goal state is represented through the highlighted features as \{1 8 13 20\}. The numbers 1-24, representing 24 different features, have no meaning as such; the features could also have been represented as different colored lights or any other collection of symbols for that matter. The meaning is to be found in the particular combination of four different features, which, taken together, represent a state. Hence, within this model, features are viewed as the relevant detectable indicators that identify the state of the simulated organization. In real world terms, relevant features could be the indicator of profit, turnover, workforce, shareholder value, market share, product demand, etc.

Any agent that performs the task needs feedback in order to be able to determine whether an action contributes to the goal attainment. Feedback about whether or not the new state is closer to the desired goal state has been explicitly incorporated within the Cohen model. The feedback is explicit because the agents simply receive information about whether or not the new state is closer to the desired goal. The task environment provides the feedback by highlighting either a “+” or a “-” sign, indicating that the new state is respectively closer to the goal state or further away. A more detailed discussion of the feedback mechanism and its implementation will take place in section 5.2.5.

The elements incorporated within the model—states, features, actions, events, and feedback—are the principal elements of a class of problems that an agent could be asked to learn to solve. The problem of learning, for an individual agent or group, is to enable itself to act in situations which contain some novel aspects, by using features experienced in a previous state as a guide to action.

The task environment can be characterized in terms of Russell and Norvig (1995) as follows. The task environment is regarded deterministic because the next state of the environment is completely determined by the current state and the actions selected by the agent. The task environment is also fully accessible in the sense that the agent’s sensory apparatus gives access to the complete state of the environment. The task environment is also episodic because the agent’s experience is divided into “episodes” each consisting of the agent perceiving, deliberating (decision-making), and then acting. Episodic environments are much simpler because the agent does not need to think ahead. The task environment is also characterized as static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. The last
characterization that can be made is that the task environment is *discrete*, because of the limited number of distinct clearly defined precepts and actions.

### 5.2.2 The task environment

The task environment is deterministic because the next state is completely determined by the current state and the actions selected by the agent. The mechanism upon which this is based will be explained next.

The four features representing a state are systematically distributed over the states. This systematic relation between features determines which state follows the other and determines whether states are directly related or indirectly. Indeed, it is the system of feature distribution that the agents must detect and model by applying actions and receiving information. Figure 5.2 shows the systematic relation between the features and the actions. Each square can be interpreted as corresponding to a state.

![Figure 5.2. Mapping features onto states. The model states are represented as squares. The thick interior lines represent barriers between states. Actions applied that would cross the simulation barriers produce no state transition, i.e. the action is not applicable.](image)

In the ontology of figure 5.2, the four available actions (A, B, C, D) now can be interpreted as respectively “up”, “down”, “left”, “right”. The fifth action becomes functional only within the multiagent task environment where it is used for passing

---

2 Although the fifth action “E” has only functional relevance in the multiagent task environment, the action will be applied within the background by passing control over to the only agent available. This is done for data collection reasons only because in this way, we are able to compare the single and the multiple agent condition in terms of the number of actions needed to attain the goal state.
control between the agents. When an agent has brought about a state transition within the multiagent task environment, the action control will be passed over to the other agent, which in turn brings about a state transition, etc. The thick interior lines represent barriers between the particular states. The barriers are to be interpreted as constraints on the action potential, i.e., not all actions are allowed or have an impact on all states. Actions applied that would cross the simulated barriers produce no state transition, i.e. when the particular action is applied, nothing happens.

Of course, an agent performing the model task is not, a priori, in the possession of this topological map, instead the agent has to create a model of the task environment. That is where “learning” comes into play. The agent must create this model mentally based solely on the information as presented by the interface, depicted in figure 5.1. The mental model of the task environment formed in this way is based on individual learning.

The two versions within the alignment study differ as to how learning takes place within the model. The alignment experiment implements a cognitive plausible agent that learns and represents the task environment in a different way than the Cohen agent does. We will describe both learning models.

5.2.3 The Cohen agent

In the terms of Cohen, the “learning engine” used to implement the agent in the original model concerns a matrix updating mechanism. The matrix summarizes previous experiences in that the entries of the matrix reflect the earlier reinforcement of associations between features and actions. Whenever an action moves the system to a new state that is closer to its goal, the association of that action with the features that were present in the prior state is strengthened; if this is not the case, it is weakened. The action taken is the one that has the strongest total of associations across all the features that the agent detects in the current state. The agent has no knowledge of how many states there are or how they are affected by the five available actions (Cohen, 1992: 180). Actions that have the best potential within a state will be selected and applied. The potential of actions in relation to features is based on previous experience.

5.2.4 Soar learning

The learning done by the Soar agents is based on the same “learning from the environment” principle as the Cohen agent. Hence, both agents do not differ as a learner at the functional level. However, the way they represent the task environment and the actual learning (i.e., the internal cognitive mechanism) is different. The action-feedback cycles of the Soar agent contains the following elements:

1. An agent starts its problem-solving task with a model of the environment holding only some initial knowledge about the task environment. For instance, the agent knows that it has four actions to choose from.
2. Actions will be selected based on the actual knowledge available of the task environment and applied within the task environment.

3. The agent observes the effects of the applied action.

4. Based on the observation, knowledge will be added to the long-term memory, i.e. the mental model of the task environment will be extended when information is encountered which was previously not available.

5. Return to step two until the goal state has been reached.

The Cohen agent goes through a similar learning cycle, however the main difference with Soar is the way in which environmental knowledge is represented within the agent. This subject will be addressed in detail in section 6.4.

The initial knowledge about the task environment consists of the following elements:

- A representation of the goal state and of the current state of the problem-solving process (when the two states match, the problem-solving process ceases).

- A representation of the actions that can be applied within the task environment and the correlation between the actions. For instance, action A (“up”) and action C (“down”) are antagonistic. It means that when action C is applied immediately after action A, the problem-solving state returns to the state before action A was applied, and vice versa. Each action neutralizes the other because both operate on the same dimension. The other action pair consists of action B (“left”) and D (“right”).

- The implementation is based on the fact that the relation between states on either dimension is based on two pairs of features, respectively the first and third, and the second and fourth feature.

The knowledge that will be acquired within the learning cycle concerns:

- States visited. Remember which states have been visited.

- Which actions are not allowed (i.e. not effective) to which states. From figure 5.2 we learn that transitions between certain states are not possible. The agent detects, by a process of trial-and-error, which actions in which states are not allowed. Actions that proved to have no effect on a state are not included for deliberation the next time the state is visited.

- A preference order for actions. Based on feedback information of whether the new state is closer to some desired goal or not, a preference order between actions will be established (this will be explained in more detail later).

- Knowledge about state transitions; i.e. which states are related and which actions bring about the state transition. When the problem-solving state is the initial state {4 8 16 24} and the agent applies action A, the result is a state change to {3 8 15 24}. The observation is that two features changed (4 and 16...
into 3 and 15) and two features did not change (8 and 24). The deduction is that action A has only an effect on the first and the third feature. The knowledge that is established is that state \{4 8 16 24\} and \{3 8 15 24\} are related by way of applying action A, and, since action C is the reverse of action A, the knowledge also has been established that both states are related by way of applying action C. A relation between two states is based on two features; when states share two features it means they are directly related; when they share only one feature it means that they are indirectly related, i.e., they are connected through intermediate states.

5.2.5 Feedback

We now return to the general discussion of the Cohen experiment. An important element in the learning cycle is the feedback provided. In the original experiment, the feedback mechanism is of an explicit form. We do not talk about information about which state follows another state after a certain action has been applied, i.e. which features go on and which go off. We talk about information about whether a new state is closer to the goal state or not. This information, called feedback, is simply provided. Besides this explicitly presented form of feedback, an implicit form is also conceivable. The implicit feedback mechanism entails that the agents compare the detectable features of the new achieved state with those of the former state in order to establish whether progress has been made. By comparing features in this way, the agents can draw conclusions about the relative distance to the goal state. This works as follows. The number of corresponding features of the goal state and the current state can be viewed as an indication of distance. For instance, when only one feature of the current state corresponds with that of the goal state, one could say that the distance is 3 (recall the total number of features representing a state is 4). When the new state observed and the goal state, do not share any feature than the ‘distance’ has increased to four. Hence, the feedback is negative. Because explicit feedback has been applied in the original experiment, for alignment reasons we decided to use the same form of feedback even though the implicit form is preferable both from the point of view of cognition and realism.

In order to mimic the original experiment as closely as possible, we implemented the explicit feedback mechanism. Unfortunately, Cohen did not describe the explicit feedback mechanism in full detail and leaves room for different interpretations. It can either be based on the shortest-path rule or it can be based on the two dominant directions. The shortest-path principle is based on determining the shortest route in terms of number of state transitions needed to achieve the goal state. Of every state, the shortest route can be determined based on the birds’ eye view of figure 5.2. Actions that are in accordance with the shortest-path criteria elicit positive feedback; those that are not elicit negative feedback. The other mechanism is based on the two

---

3 Despite e-mail contact with the author, the problem of different interpretations has not been cleared up.
dominant directions required solving the task. Figure 5.2 indicates that reaching the
goal state in the upper right corner would only require moving “up” and “right” from
any starting point without the barriers. Positive feedback is elicited when either “up”
or “right” is applied and negative feedback when B “left” or C “down” is applied. The
difference between the two feedback mechanisms can be demonstrated as follows.
When an agent is placed in problem state {2 6 10 18} (see figure 5.2) and moves
upwards it gets negative feedback according to the shortest path principle but receives
positive feedback according to the dominant directions principle. The shortest path
principle has been judged as too rigid in providing feedback. Not only does it provide
information on the appropriate direction, but it also contains information about the
efficiency of the selected path. The principle based on the dominant directions has
been selected for implementation because it is less rigid. It simply says, “It is best to
travel North or West” without giving directions for the shortest path. This means that
the most efficient (i.e. shortest) route will not always be selected based on the second
mechanism. The functional difference between the two principles was judged to be
very small. The consequence of the Van den Broek approach is that the Soar agent
does not always takes the shortest route despite a complete, or complete enough model
of the environment. If the shortest path heuristic had been applied, the variation would
be even less and the performance slightly higher. The conclusion, however, is that if
the shortest path principle had been implemented, the outcome of the experiment
would not have been significantly different.

5.2.6 Summary and conclusion

This section introduced and described the simulation experiment that has been
mimicked. The Van den Broek experiment differs slightly concerning the explicit
feedback provided by the simulator. Within the Cohen experiment the feedback
mechanism, apparently, has been based on the shortest path principle. The feedback
mechanism within the replicated experiment has been based on the two dominant
directions. The conclusion is that if the shortest path principle had been implemented,
the outcome of the experiment would not have been significantly different.
Additionally, this section described the differences and the similarities between the
agent types contrasted.

5.3 The experimental design

The alignment study is based on comparing the results of two experimental models
each based on a different set of assumptions concerning the agents applied. The
original experiment is called the “Cohen experiment” and the replicated experiment is
called the “Van den Broek experiment”, see table 5.1. The data of the Cohen
experiment will be the baseline performance for contrasting and discussing the results

---

4 Because this situation actually occurs in one of the examples provided by Cohen (1992), it can be
deduced that the original feedback mechanism was indeed based on the shortest path principle.
Chapter 5: The Alignment Study

of the Van den Broek experiment. Each experiment has a single agent and a multiagent condition, also called the “cooperative work” condition and the “non-cooperative work” condition. The single agent condition consists of one agent performing a task. In the multiagent condition, two agents perform the same task in cooperation. The details of the cooperative work condition will be explained below. The Van den Broek experiment will produce performance data on both the individual level and the multiactor level. In combination with the existing performance data on both the individual level and the multiactor level of the Cohen experiment, this produces four different sets of data, which will be compared and discussed.

Table 5.1. The experimental design.

<table>
<thead>
<tr>
<th>Cohen Experiment</th>
<th>↔</th>
<th>Van den Broek Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent type</td>
<td>↔</td>
<td>Agent type</td>
</tr>
<tr>
<td>Task</td>
<td>↔</td>
<td>Task</td>
</tr>
<tr>
<td>Task environment</td>
<td>↔</td>
<td>Task environment</td>
</tr>
<tr>
<td>Two exp. conditions:</td>
<td>↔</td>
<td>Two exp. conditions:</td>
</tr>
<tr>
<td>1 Agent performance data</td>
<td>Compare (1)</td>
<td>1 Agent performance data</td>
</tr>
<tr>
<td></td>
<td>↔</td>
<td>Compare (2)</td>
</tr>
<tr>
<td>2 Agent performance data</td>
<td>Compare (3)</td>
<td>2 Agent performance data</td>
</tr>
</tbody>
</table>

In the single agent condition, the task performance depends entirely on the learning competencies of the agent. Within the cooperative work condition, the overall (multiagent) performance depends not only on the individual learning competencies of the agents but also on their ability to cooperate and to coordinate their actions, i.e. their multiagent performance. Additionally, because multiagent models are considered models of coordinated problem solving (see chapter 2) it is assumed that within the cooperative work condition the agents need some set of coordination rules.

The next section indicates how agent and multiagent performance will be assessed. After that, section 5.3.2 describes the two experimental conditions. Section 5.3.3 describes how the results of the two experiments will be compared and contrasted. Furthermore, it describes how the results found will be related to the issue of cognitive plausibility, i.e. to the architectural differences of both agent types. Section 5.5 presents an overview and the conclusions of this chapter.
5.3.1 The performance assessment

This section will indicate how the agent performance will be measured. Within the experiment, two ways are used to express agent performance:

1. The learning performance and
2. The task performance.

5.3.1.1 The learning performance

The learning performance will be expressed as the number of state transitions needed to attain a goal state. For instance, starting from the initial state (see figure 5.2) the shortest path to the goal state is nine state transitions. Finding the shortest path can only be achieved when the agent has a complete (or near-complete) model of the task environment. However, the agent starts with only some initial knowledge of the task environment. Therefore, the first attempts at reaching the goal state will require longer solution paths. Since the task environment knowledge will be augmented during each additional attempt at reaching the goal state, the knowledge about the task environment becomes more complete over time. Based on the more complete knowledge of the task environment the agent will become more effective in selecting the appropriate action for each encountered problem state, resulting in shorter solution paths. The efficiency of the solution paths increases because the environmental model is augmented. Hence, the solution path efficiency correlates with the completeness of the environmental model, that is, based on a complete task environment model, efficient solution paths may be expected. A second learning performance measurement, besides the completeness of the task environment model, is the learning rate of the agents, i.e. how fast the agents learn. This second measurement is only available for the Soar agents and will be expressed as learning curves in chapters 6 and 7.

At the start of each experimental run, or at any time the model reaches a goal state, the agent is reset to a random chosen state. A single run consists of 1000 periods of model operations. Within the replication process, a decision had to be made on what to count as “model operations”. On the level of the task, two options apply: the number of actions applied, or the actual number of state transitions. Based on the statement that “the best single agent performance would average 178.6 goal-state attainments per 1000 periods” (Cohen, 1992: 181) it has been assumed that “model operations” actually stands for “number of state transitions”. The average goal-state attainment (178.6) is the outcome of dividing 1000 state transitions with the average path length of 5.6. This means that in a period of 1000 state transitions, the theoretical optimum

---

5 The assumption is that the agent will indeed optimize its behavior in taking the shortest route, which is more an element of strategy than of knowledge of the environment.

6 The average path length is computed by adding the number of state transitions it minimally takes to reach the goal state from every single problem state (84). When this number is divided by the total number of states, minus the goal state (15) we have the mean path length (5.6).
number of goal-state attainments is 178.6, under the condition of random placement and replacement in case the goal state has been achieved. In the current implementation, the number of state transitions and the number of actions will not exactly align. This is because the Soar agent encounters states in which certain actions are not allowed or have no effect. This can only be learned by applying an action, i.e. within a certain state Y, action X will be applied and nothing happens in terms of state transitions. As a result, the number of actions applied will actually be higher than the number of state transitions. In the Cohen implementation, the difference between actions applied and state transitions is not an issue. In terms of changing the strength of associations, how the Cohen agent deals with actions that would cross barriers, i.e. actions that are not allowed, remains unclear.

5.3.1.2 The task performance

A second performance measurement will be the task performance. Within a single run, consisting of 1000 state transitions, the number of times a goal state as been attained will be counted. This is called the goal-state attainment number. In the case of a single-agent performance, the goal-state attainment number will correlate with the average path length. Remember that the average solution path efficiency depends on how fast an agent learns and the completeness of its environmental model. Hence, the more efficient the solution paths the higher the average number of goal attainments. This means that both measurements are different ways of expressing the same agent performance.

Within the cooperative work condition, the quality of the cooperation process is of course an additional variable, which can accelerate or hamper the overall task performance. It means that even when the individual learning performance of the agents is at its optimum, an inefficient coordination process could cause a drop in the overall goal-state attainment number. Hence, in the case of multiagent performance, the task performance of the multiagent system will depend on the individual learning performance and the quality of the cooperation process.

The data will be collected over 30 experimental runs, each consisting of 1000 state transitions. Each single experimental run must be viewed as a separate simulation experiment in which the agents were reloaded and set to the task based on only a priori knowledge. It means that the model of the task environment created within a run was erased, and therefore did not transfer to the next run. In this way, 30 goal-state attainment counts could be established, which resulted in an average goal-state attainment count. The same holds true for the average solution path.
5.3.2 The two experimental conditions

The basic experimental design of the original experiment⁷ consists of contrasting the task performance of a single agent with that of a team consisting of two agents. Above, the model task has been explained in single agent terms, this section addresses the cooperative work condition.

Within the cooperative work condition, the two employed agents take turns in controlling the organizational movements. This means that when one agent has brought about a state transition, the control over the actions will be passed over to the other agent, which in turn brings about a state transition. This “pass control” rule is in line with the original model. Within the Van den Broek experiment, however, a Soar agent stays in control until a state transition has been brought about. This means that when an agent has applied an action that would cross a “barrier” in terms of figure 5.2, this action will not bring about a state transition. This is the case for instance when an agent wants to go from state \{3 8 15 24\} to state \{2 8 14 20\}, which would cause no change within the task environment. In such a case, the same agent applies another action. If the agent now applies action “left”, this would cause a state transition and the action control shifts accordingly. Hence, in this case, one agent has applied two actions in order to bring about a state transition. In general, this means that within the replicated experiment the total number of actions applied, apart from the actions required for applying the “pass” rule, is more than the total number of state transitions. How the “barrier” issue was dealt with within the Cohen experiment has not been addressed. Within the Cohen model, the number of ‘model operations’ equals the number of state transitions. Whether this means that these actions simply die off in the reinforcement scheme of the action-feature associations, because nothing happens, has not been documented. However, I assume⁸ that this is the case and that therefore the only ‘real’ difference is that within the replicated experiment this aspect of ‘learning the environment’ has been made explicitly. The fact that this knowledge can be represented explicitly is a first indication of how much richer the representational ability of the Soar agent is.

Because of the default “pass” rule, one could view the cooperative work condition as based on the team as a contract principle (March, 1994). In such a team, the agents value the same outcome of group activity and will each attempt to contribute in whatever way they can to that global outcome. This assumes that the decision-making process is imagined to be divided into two stages. At the first stage, inconsistencies are removed concerning the “ends”, i.e. on what is viewed as the operational targets of the

---

⁷ In the alignment study, two additional experimental conditions reported by Cohen (1992) in which agents have limited action potentials, have been omitted.

⁸ The assumption is based on the representational capacity of the Cohen agent, which does not allow an explicit representation of a barrier. However, explaining this requires an extensive discussion of the representational differences between the two agent types, a discussion that lies at the very heart of the cognitive plausibility issue. Because this discussion takes place in chapter 6, we will address this issue there.
cooperation. These inconsistencies are removed through the various forms of bargaining, side payments, and agreements that define a contract binding the parties. At the second stage, the multiple agents operate as a team (see chapter 3). Part of the team “contract” is an agreement about the action control shift; hence, benevolence exists concerning cooperation. Both agents have a shared outcome preference in that they agree about the ends of their cooperation effort. In terms of Gilbert (1989), this means that the agents cooperate based on a joint commitment. The agents applied within the cooperative work condition are designed based upon the benevolence principle and will show veracity. A motivation for benevolence is having a common goal, that is, if the agents value the same outcome of group activity, they will each attempt to contribute in whatever way they can to that global outcome. Veracity is the assumption that an agent will not knowingly communicate false information. Even with the benevolence and veracity assumption, cooperation and coherent coordination are far from assured. Difficulties of timing and local perspectives can lead to uncooperative and uncoordinated activity as will become clear in chapter 7.

Because of the “pass” rule, the fifth action (E) comes into play, see figure 5.1. The benevolence for control shifts is based on a built-in rule, see Appendix A under label PROPOSE OPERATOR PASS. The initial knowledge held by the agents within the cooperative work condition is the same as applied within the non-cooperative work condition. This means that, except for the “pass” rule, the agents used in the cooperative work condition are exact copies of the model used within the non-cooperative work condition.

5.3.3 Comparing the results

Based on this experimental design, represented in table 5.1, the alignment study will compare and discuss the experimental results in four ways. The intention of this comparison is to not only contrast the performance data as such, but also to explicitly link the differences concerning the performance data to the architectural differences. Everything else being equal, the assumption is that differences found in experimental results, both at the individual level and the multiagent level, are somehow related with the agent architectures applied within each experiment. Therefore, the intellectual challenge is to reduce the performance differences found to functional differences concerning the agent architectures, and to explain the results in terms of the internal knowledge that is organized in a specific knowledge representation. The architecture incorporates the principles concerning the way internal knowledge will be represented and the way the agents learn. This means that the differences concerning the learning behavior and performance of the agent are to be related and reduced to architectural differences.

The first comparison is between the performance of a single Cohen agent and that of a single Soar agent. What are contrasted in fact are the learning characteristics and performances of both agent types performing the same non-cooperative task. Comparing the performance data and discussing the functional properties that account
for the differences found, takes place in chapter 6. Section 6.3 presents an overview of
the Soar agent performance data and section 6.4 discusses the differences concerning
the learning characteristics in terms of differences between the architectures. The
second comparison contrasts the single Soar agent performance with that of the multi
Soar agent condition. Hence, this entails a comparison of the non-cooperative and the
cooperative work condition within the Van den Broek experiment. The results will
indicate how the Soar agents cooperate and whether or not the cooperation condition is
faster than the single agent condition. Comparing and discussing the results is done in
chapter 7, section 7.4. The third comparison concerns the performance of the single
Cohen agent and that of the two Cohen agents in the cooperative work condition. The
performance data of the Cohen experiment are presented in table 5.2. The first row
indicates the performance of a single agent. The data indicates that the average number
of goal-state attainments over 30 experiments consisting of 1000 state transitions is
66.03. The average path length is 15.14. The second row indicates the performance of
the multiagent condition. The average number of goal-state attainments is 88.03. The
average path length is 11.36.

Table 5.2. Comparing the performance data of the Cohen experiment.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Mean</th>
<th>Standard Dev.</th>
<th>N</th>
<th>Average path length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Cohen agent</td>
<td>66.03</td>
<td>19.73</td>
<td>30</td>
<td>15.14</td>
</tr>
<tr>
<td>2 Cohen agents</td>
<td>88.03</td>
<td>14.22</td>
<td>30</td>
<td>11.36</td>
</tr>
</tbody>
</table>

Table 5.2 clearly indicates that the learning achieved by multiple agents within the
cooperative work condition outperforms the single agent learning performance of the
non-cooperative work condition. The data table 5.2 presents the baseline performance
for contrasting the results of the Van den Broek experiment. An extensive explanation
and discussion about the reason why the multiagent condition outperforms the single
agent condition takes place in chapter 7, section 7.5. The fourth comparison is that
between the cooperative work conditions of both experiments. The differences found
between the multiagent conditions will be discussed in full in chapter 7, section 7.6.

5.3.4 Summary

This section described the experimental design of the alignment study. It addressed the
way in which the agent and multiagent performance will be assessed. In addition, it
discussed the two experimental conditions and how this leads to comparing the results
of the alignment experiment in four different ways. This section discussed a possible
difference in the way both experiments deal with “barriers”. In the replicated
experiment, the agents have no a priori knowledge and have to learn to situate them.
This means that the number of actions applied is actually higher than the number of
Chapter 5: The Alignment Study

state transitions. How the Cohen experiment deals with the barriers has not been documented. However, it has been assumed that actions that would cross barriers simply die off in the reinforcement scheme of the action-feature associations, because nothing happens. The conclusion is that the only ‘real’ difference is that within the replicated experiment this aspect of ‘learning the environment’ has been made explicitly, which in fact is a first indication of the richer representational ability of the Soar agent. Furthermore, this section explained how the alignment experiment results are related to the issue of cognitive plausibility by reducing them to the functional and representational level.

5.4 The simulation environment

The simulation environment was implemented initially using Soar7.0, the first Soar distribution\(^9\) to incorporate Tcl\(^10\). Tcl is pronounced “tickle” and stands for “tool command language”. It is a simple scripting language that allows a Soar user to write extensions to Soar without having to compile Soar itself. Tcl allows the addition of Tk, pronounced “teekay”, a toolkit for writing graphical interfaces. Tcl and the Tk toolkit (Ousterhout, 1994) were initially developed for machines running Unix and X windows. Since the Soar7.2 release, Soar has been fully integrated with Tcl7.6/Tk4.2 and is available for Apple computers and the Intel platform\(^11\). The simulation experiment was implemented using the Soar7.2 distribution running on an Intel/Windows98 platform. Within the design philosophy of the Soar7.2 distribution, Soar is no longer a stand-alone process. Instead, it is implemented as a dynamically loadable Tcl extension (library) and is loaded into a Tcl interpreter at run time. More specifically, Soar is a Tcl package that can be made available in an interpreter using the “package require Soar” command. When Soar is loaded into the Tcl interpreter, a single Soar agent is created and assigned to the interpreter. The resulting integrated computational environment, into which Soar fits as a package, is called the Soar environment. Figure 5.3 depicts the Soar environment.

\(^9\) Soar distributions are available as public domain software and can be downloaded from: http://bigfoot.eecs.umich.edu/~soar/software.html. Other links concerning Soar are: http://bigfoot.eecs.umich.edu/~soar/main.html; http://www.ccc.nottingham.ac.uk/pub/soar/soar-web-page.html; http://www.nottingham.ac.uk/pub/soar/nottingham/soar-faq.html.

\(^10\) The initial simulation environment was implemented using Soar 7.0.0.beta in combination with Tcl7.4/Tk4.0 on an IBM rs6000 workstation. The simulation environment was ported in 1999 to Soar7.2 using Tcl7.6/Tk4.2 running on an Intel PC.

\(^11\) In fact, Soar7.1 was already a multiple platform (Unix, Macintosh, and Windows) distribution, but did not prove sufficient in supporting interaction among multiple agent interpreters.
The two main components of the Soar environment are the Soar architecture (Soar kernel) and the agent interpreters. The agent interpreters connect the Soar kernel and non-agent interpreters, like graphical user interfaces and other programs. Within the Soar environment, more than one TCL agent interpreter can be created to run in parallel. Each TCL agent interpreter holds a single Soar process. Each Soar process has its own set of decision cycles, a separate chunking mechanism as well as a separate working memory, production memory, and preference memory. Thus on the implementation level, consists each agent of a Soar process and an agent interpreter that hooks the agent up to other non-agent processes. The idea is that you are running the thus implemented agents all together, so that each does its own reasoning on a problem task in sync with the other. This means that when you do not want to run the
agents all together or you are in need of agent interaction, you need to write a multiaagent scheduler in Tcl, allowing you to implement whatever scheduling criteria you want. Inter-agent communication can be established through Tcl. This is possible with an RHS function call that can execute a Tcl script. The script can do something as simple as send a message to the simulator (also written in Tcl). The simulator can then send the message to a particular recipient agent or to all recipient agents. The Soar cognitive architecture will be invoked within the TCL environment as a Soar package. The Tcl/Tk Soar Interface (TSI) is the other component that is part of every distribution of Soar. TSI is a GUI-driven interface between Tcl and Soar by means of which Soar processes can be controlled and monitored.

For the simulation effort described below, a special purpose simulation environment was created—with which the extendibility of the Soar environment has been demonstrated—consisting of the following elements:

1. A simulator program (programmed in Tcl/Tk) consisting of an interface with control buttons and the topological map of the task environment. The code of the simulator program has been listed in Appendix B.

2. An interface representing the problem-solving view of the agents holding the goal state, current state, action applied, and feedback obtained. The code of the interface presenting the problem-solving view of the agents has been listed in Appendix C.

3. The Soar package, including different agent interpreters. These elements of the simulation computational environment are depicted in figure 5.4.
The simulator program implements the task environment and operates according to the model depicted in figure 5.2. Furthermore, the simulator program handles all the I/O routines with the agent interpreters and updates the interfaces. The simulator’s front end is a graphical user interface that contains the topological map and some control buttons. The interfaces that are part of the simulator environment are depicted in figure 5.5.

It is important to bear in mind that the agents performing the task do not know or use the topological map that is part of the control interface. The information presented through the problem-solving view interface is all the information that the Soar agents use within the problem-solving process. Hence, the simulator program presents the
Soar agents the current state, receives action requests from the agents and responds respectively with a new current state and feedback, or a ‘no state’ transition response. The interface containing the problem-solving state was built for observer convenience and reveals which agent makes which decision and to what effect. The topological map, therefore, must be viewed as the bird’s eye perspective of the theorist/observer who understands the relations between the model states. Both the interfaces of the problem-solving state and the topological map are needed for human observers to be able to follow and interpret the agents’ decision-making behavior and learning progress.

Figure 5.5. The visible elements of the simulator environment.

The control interface contains a number of buttons to control the experiment. The “create agents” button creates either one or two agents depending on the experimental condition. This will be determined by selecting the appropriate radio button in the “select number of agents” area. The process of “creating” agents means that an agent interpreter is created that runs decision cycles, has a production memory, working memory, and preference memory, etc. These interpreters are of type “agent”. Within Soar, different agents can be created running simultaneously as self-contained
processes using the \texttt{INTERPRETER CREATE} command. Figure 5.4 depicts two agent interpreters, which are labeled “Alpha” and “Beta” as default within the simulation experiment. Because each Soar agent is an agent interpreter, it holds that each agent has its own “name” for identification, its own I/O routines with the simulator, its own production memory, working memory, etc. The Soar/Tcl output window contains information regarding the internal processing of the Soar agents. The level of detail can be different depending on the variables set. For instance, it is possible to print the default rules that fire, or the user-defined rules that fire. Furthermore, it is possible to list the working memory elements (wmes) that are active, etc. The Tcl simulator program prints the number of goal-attainments, and the number of transitions that have taken place.

In chapter 1, we indicated that we selected the Soar cognitive architecture because of its multiagent functionality and extendibility. At this point, it must be made clear that Soar’s multiagent functionality does not constitute a complete multiagent simulation model. Instead, the multiagent functionality entails the fact that when you create a new Soar interpreter (by creating a new interpreter and then loading in the Soar package), that interpreter registers itself in a global list of Soar agents. Then, when you do a “run” command in any one of the Soar agent interpreters, it starts activating every Soar agent interpreter, no matter where those Soar agents appear in Tcl’s interpreter hierarchy. It means that you are running these agents all together, each doing their own reasoning on problem tasks in synch with each other without any interaction and without the agents being part of the same simulated environment. So, when you do not want to run the agents all together, and you require agent interaction, and you wish the agents to share the same simulated environment, you need to write a simulator program that includes a multi-agent scheduler. This is exactly what the simulator created for this experiment does. First, it provides the agents their shared task environment, which provides them feedback and upon which they act. Secondly, the simulator schedules the agents in executing their control over actions applied. Thirdly, the simulator provides the shared information space through which the agents interact. This extendibility of the Soar environment, thanks to the full integration within Tcl, with simulators and interfaces is one of the fine features of the Soar environment enabling inter-agent communication.

The inter-agent communication is established through Tcl. This is possible with a new RHS function (called “tcl”) that can execute a Tcl script. The script can do something as simple as sending a message to the simulator (also written in Tcl). The simulator can then send the message to the desired recipients. The Soar agents communicate with the simulator and vice versa through the Tcl interpreter. A Soar production rule is shown that applies the selected action. The last line is a Tcl interpreter call using the tcl RHS function to inform the simulator that the action selected is the contents of variable \texttt{<ACTION>}.  

\begin{verbatim}
sp {agent*apply*operator*action
  (state <s> ^problem-space.name model-task-ps
\end{verbatim}
The Tcl interpreters communicate with one another using the existing Tcl interpreter communication primitive \texttt{INTERP EVAL}. For instance, the following is part of the simulator reply to an agent: \texttt{$AGENT EVAL "$\texttt{ADD-WME $INPUT\_ROOT APPLIED\_ACTION DOWN}$"}. It means that a working memory element is added stating that the requested action has been effected.

After the agent interpreter has been created, the “create agent” button loads the production memory within the individual agent interpreter, a process called “sourcing” after the \texttt{SOURCE \texttt{<NAME>}} command. A production memory contains production rules that enable the agent to communicate with the simulator and contains production rules necessary for coping with the task environment. The production rules for coping with the task environment represents the following elements in general terms:

- Rules for goal detection
- Rules containing the current state of the problem-solving process
- Rules that effectuate the actions, which can be applied externally, as operators
- Search control rules fit for selecting among the multiple actions
- Rules providing structures for holding the experience
- Rules for interpreting the feedback

\subsection*{5.4.1 Summary}

This section explained how the alignment experiment was set up and implemented using the Soar architecture. Furthermore, it identified and explained the computational elements within the simulation environment as well as the issue of multiagent functionality and the extendibility of the Soar computational environment.

\subsection*{5.5 Summary and conclusion}

This chapter addressed the fourth research question: How has the alignment study been setup? It described the original experiment, based on which the alignment study was implemented and conducted. In addition to introducing the original experiment and its results, the chapter indicated some differences between the two experimental versions. One difference concerns the way the explicit feedback mechanism has been implemented. The conclusion is that if the shortest path principle had been implemented, the outcome of the experiment would not have been significantly different. The second (possible) difference concerns the way in which both experiments deal with the “barriers”. Within the Van den Broek experiment, the Soar
agents explicitly have to model these “barriers”, that is, the placement of the barriers is not a priori knowledge. How the Cohen experiments deal with the “barriers” has not been documented. The conclusion is that the possibility of explicating this aspect of ‘learning the environment’ is in fact a first indication of the richer representational ability of the Soar agent. This chapter, furthermore, made clear that the only experimental variable concerns the internal learning mechanism of both agents. The differences between the agent performances are compared and discussed in four different ways. First, comparing the single agent performance of both experiments. Second, comparing the multiagent performance of both experiments. The third and fourth comparisons are that between the single and multiagent performance within each experiment. The alignment experiment results will be related to the issue of cognitive plausibility by reducing them to the functional and representational level. Additionally, this chapter described how the alignment experiment was set up and implemented using the Soar computational environment, and identified the computational elements within the simulation environment.

Instead of a correlation between the four actions established a priori, a cognitive modeling approach is conceivable in which the correlation between the actions is one of the detectable variances within the model. Consider the following example from the perspective of a problem-solving agent. When the problem solving state is the initial state (4 8 16 24) and the agent applies action A, the result is in a state change to (3 8 15 24). The observation is that two features changed (4 and 16 into 3 and 15) and two features did not change (8 and 24). The deduction is that action A has only an effect on the first and the third feature.

Suppose the next action the agent applies is C. The effect is that the organization returns to state (4 8 16 24). The deduction is that action C is the inverse of action A, and vice versa. From the perspective of the agent, this means that by applying C the effect of action A gets undone and vice versa. Action A and C are antagonistic because they operate on the same “dimension” represented by the first and third feature (8 and 24).

When in the state (4 8 16 24) action B is applied the state changes to (4 7 16 23). The observation now is that the two features that did change are (8 and 24 into 7 and 23). The deduction is that action B has only an effect on the second and the fourth feature. When action D is applied again the state will set back, leading to the conclusion that action B and D are antagonistic as well on some second “dimension” represented by the second and the fourth feature. This results in knowledge of two sets of pair actions (in the example A and C versus B and D) each operating on different dimensions. Within the simulator the effects the different action have within the task environment can be changed in a randomly fashion each time the problem solving sequence begins. In practice, it can take a considerable “action taking effort” before the action pairs can be determined in combination with the subsequent “dimensions”.

Within the implementation described herein, the assumption is that the learning sequence necessary for detecting the action pairs and the subsequent dimension on which they operate has been concluded. In other words, we started the simulation with an agent holding this knowledge a priori. The assumption has been made because the exercise of cognitive modeling this form of concept formation falls beyond the scope of this research. Furthermore, in this way it is more in line with the original experiment in which the action relation is fixed and pre-determined.