Part V: Planning and Learning

This part of the tutorial will teach you to build Soar programs that use subgoals to plan and learn. The type of planning that you will learn about in this section is simple look-ahead planning. Other types of planning are possible in Soar, but this is the most natural. In fact, you will need to make only minor changes and a few additions to use planning in the Water Jug and Missionaries and Cannibals problems. This type of learning is called chunking, which is a form of explanation-based generalization. Within the examples in this section, chunking speeds up problem solving by creating search-control rules. In other types of problem solving, chunking creates other types of rules including elaborations, operator proposals and operator applications.

In simple look-ahead planning, your program will try out operators internally to evaluate them before it commits to applying them in the actual task. In the programs that you developed in Part I and IV, all of the problem solving was internal, so it didn't really matter if a mistake was made. For example, in the Missionaries and Cannibals problem, you added rules to backup when a failure state was reached. However, in problems that involve interaction with a world where you can't always undo an action, it is often critical to try out actions internally before executing them in the world. In addition, planning provides a way of comparing alternative operators based on the states they produce. Chunking builds rules that summarize the comparisons and evaluations that occur in the look-ahead search so that in the future the rules fire, making look-ahead search unnecessary – converting deliberation into reaction.

Many planning systems have a two-stage cycle of planning and execution. They always plan when given a new problem and then execute the plan step by step. Our approach in Soar is different. In Soar, each decision is made with whatever knowledge is available and planning is one more source of knowledge that can be used whenever the knowledge encoded directly as rules is insufficient to make a decision. Thus, planning is used on an as-needed basis. As we will see later, planning in Soar does not create a step-by-step sequence of operators to apply blindly, but instead learns a set of context-dependent rules that prefer or avoid specific operators for states in the space. As a result, Soar's plans are more flexible so that plans learned for one problem may transfer to part of a second problem. It also makes it so that Soar only plans when it needs to.

Our study of planning will parallel the study of problem solving. You will first modify and extend the rules you wrote for the Water Jug problem and then the Missionaries and Cannibals problem.

1. The Water Jug Problem

Planning in Soar arises out of its impasse detection and substate creation mechanism. Remember in TankSoar how subgoals were created when an operator was selected but there were no rules that could directly apply the operator. Soar has other impasses that arise when progress cannot be made for other reasons (see the manual for a complete description of impasses and substates). Soar automatically creates a subgoal whenever the preferences are insufficient for the decision procedure to select an operator. This is a signal that there is insufficient knowledge to make a decision and further problem solving is required. The decision procedure can choose an operator if there is either one clear winner (one operator that dominates the others based on the available preferences), or when there are multiple operators that are all indifferent. In the rules you've written so far, there have always been indifferent preferences for every operator, so that the decision procedure always selected randomly among the preferred operators. However, indifferent preference should be used only when it is known that one operator is no better than another, instead of when it is not known which operator is best. Therefore, the first step to allow planning to be invoked is to eliminate the indifferent preference from the operator proposal rules.

Below is the operator proposal for the fill operator in the Water Jug without the indifferent preference.

```
sp {water-jug*propose*fill
    (state <s> ^name water-jug
               ^jug <j>)
    (<j> ^empty > 0)
    -->
    (<s> ^operator <o> +)
    (<o> ^name fill
         ^jug <j>)}
```

Once that indifferent preference is removed, rerun your program and step through the first decision. You will discover that Soar automatically generates a new substate in situations where the operator preferences are insufficient to pick a single operator. This is called a *tie impasse.*

When a tie impasse arises, a new substate is created that is very similar to the substate created in response to an operator no-change. Print out the substate and examine the augmentations. The key differences are that an operator tie has `^choices multiple`, `^impasse tie`, and `^item` augmentations for each of the tied operators.

For the operator no-change impasses you used in TankSoar, the goal was to apply the selected operator to its state. This was achieved by selecting and applying operators in the substate that modify the superstate either directly (by adding or removing augmentations) or indirectly (by performing actions in the world that led to changes in the sensors that are part of the state). For a tie impasse, the goal is determine which task operator is the best to apply to the current state. This will be achieved by selecting and applying *evaluation* operators in the substate, one for each of the task operators. The purpose of these operators is to create an evaluation of the task operators, such as failure, success or a numeric evaluation of the likelihood of success. These evaluations will then be converted in preferences for the task operators. Once sufficient preferences have been created to select a task operator, it will be selected and the substate will automatically be removed from working memory. It is possible that after evaluating only a subset of the tied task operators, sufficient evaluations will be created so that preferences are generated that break the tie. For example, if an operator is evaluated and it is determined that it leads directly to the goal, a best preference can be created. This might be sufficient to break the tie, select an operator, and resolve the impasse.

One way in which task operators can be evaluated is by trying them out on internal copies of the states in which the tie occurred, and then evaluating the states they create. The evaluation of states can be based on whether they are at desired state, a failure state (as in Missionaries in Cannibals), or the initial state. Many problems also lend themselves to an evaluation function that can be applied to a state that estimates how close the state is to the goal. For example, in Chess, numbers are assigned to the different pieces and you compare the value of your pieces to your opponents.

We will call the problem of resolving a tie impasse, the *Selection* problem throughout the rest of the tutorial. We will treat the development of rules and operators for the selection problem just as we would treat the development of a program to solve other problems, such as the Water Jug or Missionaries and Cannibals, and go through the same steps we went through for those problems. Although coming up with good state evaluation functions for a specific problem can be challenging, the overall approach is very general and requires only a few bits of domain-specific information. In fact, we have created a set of generic Soar operators/rules that carry out the evaluation and comparison of operators using this approach. This set of generic rules can be used in a wide variety of problems for simple planning and they are available as part of the Soar release. These rules are part of the default rules that come with the Soar release in the file demos/default/selection.soar. You should load these rules when you load in the Water-jug rules. You can do this by adding the following commands to your files (assuming that your program is in a subdirectory of the Soar 8 release):

```
pushd ../default
source selection.soar
popd
```

By going through the rest of this section you will learn how these rules work and what you need to provide for them to be used on other problems.

From our previous work on solving problems in Soar, here are the types of knowledge we need to consider for the selection problem:

1. The state representation. This includes the representation of evaluation objects that link evaluations with operators.
2. The initial state creation rule. Since this problem arises as a substate, an initial state is generated automatically.
3. The operator proposal rules. The only operator is evaluate-operator and it must be proposed for every tied operator.
4. The operator application rules. Evaluate-operator is implemented hierarchically, like the abstract operators in TankSoar. Therefore, operator application involves a substate and substate operators.
5. The operator and state monitoring rules. There are no special monitoring rules for Selection.
6. The desired state recognition rule. Desired state recognition is automatic. When there are sufficient preferences, the decision procedure selects an operator, the impasse is resolved, and the substate is removed from working memory.
7. The failure recognition rule. There are no failure states for this problem.
8. The search control rules. These rules help guide which operator should be evaluated first.

1.1 Selection State Representation

For the selection problem, the operators are going to create and compare evaluations. Eventually, preferences for the superstate will be created, but those do not have to be represented in the Selection problem.

If the evaluations had only a single value, namely the evaluation, then they could be represented as simple augmentations of the state, such as: `^evaluation success`. However, the evaluations must also include the task operator that the evaluation refers to. Moreover, we will find it useful to have different types of evaluations that can be compared in different ways, such as symbolic evaluations (success, failure) or numeric evaluations (numbers over some ordered range). We will also find it useful to create the evaluation structure before the evaluation is known, so that it will be useful to have an augmentation of the evaluation object that signifies that a value has been created. Therefore, the state should consist of a set of evaluation augmentations, with each evaluation object having the following structure:

- `operator <o>` the identifier of the task operator being evaluated
- `symbolic-value success/partial-success/partial-failure/failure/indifferent`
- `numeric-value [number]`
- `value true` indicates that there is either a symbolic or numeric value.
- `desired <d>` the identifier of a desired state to use for evaluation if there is one

Another alternative to creating evaluations on the state is to create evaluations on the operators themselves. The advantage of creating evaluations on the state is that they are automatically removed when the impasse is resolved.

1.2 Selection Initial State Creation

In the previous problems in this part, you have had to create a rule that generates the initial state. However, for the selection problem, the state is automatically created in response to an impasse. Furthermore, the evaluation objects will be created by operator applications. The one initial state elaboration rule that is necessary is one that names the state selection. This rule really isn't necessary, but will allow us to test for the name of the state instead of the impasse type in all of the remaining rules.

```
sp {default*selection*elaborate*name
   :default
   (state <s> ^type tie)
   -->
   (<s> ^name selection)}
```

This rule uses a new bit of syntax, the `:default`, which tells Soar that this is a default rule. Default rules do not behave any differently than other rules; however, Soar keeps separate statistics on default rules and allows you to remove all of the non-default rules easily. This rule and all others labeled default are included in the selection.soar file.

1.3 Selection Operator Proposal

The only operator in the Selection problem is evaluate-operator. This should be proposed in the Selection problem if there is an item that does not have an evaluation with a value. The operator will first create an evaluation and later compute the value.

Selection*propose*evaluate-operator
If the state is named selection and there is an item that does not have an evaluation with a value, then propose the evaluate-operator for that item.

The tricky part of translating this into a rule is the test that there is an item without an evaluation with a value. The item can be matched on the state as: `^item <i>`. The test that no evaluation exists is going to be a *negation* of an evaluation object `(<s> ^evaluation <e>)`. The evaluation object must also include a reference to the item, which is encoded as `(<e> ^operator <i>)`. An additional test `(<e> ^value true)` must be included in the negation so that the rule will match only if there does not exist an evaluate with a value, giving:

```
      sp {selection*propose*evaluate-operator
         :default
         (state <s> ^name selection
                    ^item <i>)
       -{(state <s> ^evaluation <e>)
         (<e> ^operator <i>
              ^value true)}
         -->
         (<s> ^operator <o> +, =)
         (<o> ^name evaluate-operator
              ^operator <i>)}
```

Given these conditions, once an evaluate-operator operator is selected, it will stay selected until an appropriate evaluation is created with a value. Therefore, one and only one evaluate-operator operator will be created and applied for each of the tied task operators.

Unlike the Water Jug and Missionaries and Cannibals operators, these operators can be made indifferent. It is unlikely that it would be useful to do problem solving to decide which operator should be evaluated first. However, there might be some cases where some knowledge about the task can be used to order the evaluation of operators. For example, many Chess playing programs use iterative deepening to order the evaluation. Moreover, in a later section, we will see an example where it is useful to add some search control knowledge for evaluate-operator selection.

1.4 Selection Operator Application

The application of the evaluate-operator operators has two parts. The first part is the creation of the evaluation data structure, but without any value. In parallel with the creation of the evaluation structure, the operator is elaborated with additional structures to make application simpler. The end result is the following:

```
(<s> ^evaluation <e>
     ^operator <o>)
(<e> ^superoperator <so>
     ^desired <d>)
(<o> ^name evaluate-operator
     ^superoperator <so>
     ^evaluation <e>
     ^superstate <ss>
     ^superproblem-space <sp>)
```

The desired augmentation is an object that describes the desired state for the original task. In Water Jug, it was an object that described the three-gallon jug having a single gallon of water. In the Missionaries and Cannibals, it was an object that described having all of the missionaries and cannibals on the right side of the river. The desired structure is included so that the evaluation of an operator can be based on how well it helps achieve the desired state. This might be by achieving the desired state, or it might be based on a heuristic calculation of distance from the desired state to the state created by applying the operator being evaluated.

The superproblem-space augmentation is the problem-space object of the superstate. What is a problem-space object? It is an explicit representation on the state of properties of the current problem space and can be used to cue the firing of rules relevant to the current problem space. The name augmentation of the state has served this purpose in the previous systems; however, it is insufficient in general because it contains only a single symbol, the name. By having an object, instead of just a name, additional properties of the problem space can be included. This is important in planning because the look-ahead search must use the same problem space and must copy the current state, which requires information about the structure of the problem space states..

The second part is the calculation of the evaluation, which cannot be done directly with rules but requires its own substate. This substate is called to the *evaluation* problem. The original task state is copied to the evaluation substate. Then, the task operator being evaluated is applied to that state. If that new state can be evaluated, then the evaluation is returned as a result and added to the appropriate evaluation object. If no evaluation is possible, then problem solving in the *task* (such as the Water Jug) continues until an evaluation can be made. That may lead to more tie impasses, more substates, … The key computations that have to be performed are:
1. Creating the initial state
2. Selecting the operator being evaluated
3. Applying the selected operator
4. Evaluating the result

1.4.1 Creating the Initial State

The initial state must be a copy of the state in which the tie impasse arose. Copying down all of the appropriate augmentations can be done by a few Water Jug specific rules; however, that means that with each new task, new state copy rules must also be written. To avoid this, we have written a general set of rules that can copy down the augmentations. These rules match against augmentations of the problem space to determine with state augmentations to copy. Below are the legal augmentations dealing with state copying and their meaning:

- default-state-copy no: Do not copy any augmentations automatically.
- one-level-attributes: copies augmentations of the state and preserves their value.
  Example:
  ```
  (p1 ^one-level-attributes color) (s1 ^color c1) (c1 ^hue green) ->
  (s2 ^color c1)
  ```
- two-level-attributes: copies augmentations of the state and creates new identifiers for values. Shared identifiers replaced with same new identifier.
  Example:
  ```
  (p1 ^two-level-attributes color) (s1 ^color c1) (c1 ^hue green) ->
  (s2 ^color c5) (c5 ^hue green)
  ```
- all-attributes-at-level one: copies all attributes of state as one-level-attributes (except dont-copy ones and Soar created ones such as impasse, operator, superstate)
  Example:
  ```
  (p1 ^all-attributes-at-level one) (s1 ^color c1) (s1 ^size big) ->
  (s2 ^color c1) (s2 ^size big)
  ```
- all-attributes-at-level two: copies all attributes of state as two-level-attributes (except dont-copy ones and Soar created ones such as impasse, operator, superstate)
- dont-copy: will not copy that attribute.
  Example:
  ```
  (p1 ^dont-copy size)
  ```
- don't-copy-anything: will not copy any attributes
  ```
  (p1 ^dont-copy-anything yes)
  ```

If no augmentations relative to copying are included, the default is to do all-attributes-at-level one. The desired state is also copied over, based on the copy commands for the state.

These rules support two levels of copying. How should you decide what level of augmentations need to be copied? In the water jug, the state has two levels of structure: the jugs and their contents, volume, and empty:

```
      (s1 ^jug j1 j2)
      (j1 ^volume 3
          ^contents 0
          ^empty 3)
      (j2 ^volume 5
          ^contents 0
          ^empty 5)
```

Is it sufficient to copy the augmentations of the state, or do the subobjects need to be copied as well? To answer this question, you need to look closely at how the operators modify the state. In the formulation of the problem we've been using, the operators change the augmentations of the jugs. For example, the application rule for fill is:

```
sp {water-jug*apply*fill
    (state <s> ^operator <o>
               ^jug <j>)
    (<o> ^name fill
         ^jug <j>)
    (<j> ^volume <volume>
         ^contents 0
         ^empty <volume>)
    -->
    (<j> ^contents <volume>
                   0 -
         ^empty 0
               <volume> - )}
```

This rule modifies the contents and empty augmentations. So if this is used to apply the fill operator to the three-gallon jug using the working memory elements above, `(j1 ^contents 0)` and `(j1 ^empty 3)` will be removed from working memory and `(j1 ^contents 3)` and `(j1 ^empty 0)` will be added to working memory. The result will be that not only is the "copy" modified, but also the original state. Thus, this is unacceptable because it does not make it possible to apply operators to substate without modifying the original.

There are two possible solutions. The first is to copy two levels of attributes. This can be achieved by adding the following to the actions of original state initialization rule for Water Jug:

```
sp {water-jug*elaborate*problem-space
    (state <s> ^name water-jug)
    -->
    (<s> ^problem-space <p>)
    (<p> ^name water-jug
         ^default-state-copy yes
         ^two-level-attributes jug)}
```

You could use `^all-attributes-at-level two` instead, but it is best to list exactly the attributes you need to have copied. After using this, the substate, `s3`, would have the following structure:

```
(s3 ^jug j3 j4)
(j3 ^volume 3
    ^contents 0
    ^empty 3)
(j4 ^volume 5
    ^contents 0
    ^empty 5)
```

The second approach is to change the operator applications so that they do not modify the augmentations of the jug objects but instead create completely new jug objects. For example, a modified version of fill would be:

```
sp {water-jug*apply*fill
   (state <s> ^operator <o>
              ^jug <i>)
   (<o> ^name fill
        ^jug <i>)
   (<i> ^volume <volume>
        ^contents 0
        ^empty <volume>)
   -->
   (<s> ^jug <i> -
        ^jug <ni>)
   (<ni> ^volume <volume>
         ^contents <volume>
         ^empty 0)}
```

The disadvantage of this approach is that it requires more changes to working memory when an operator is applied, although it requires fewer working memory elements to be copied during the creation of the initial state. This approach also is less natural in that it makes it implies that a new jug is created as opposed to just modifying the contents of the existing jug. For these two reasons, the first approach (two-level-attribute copying) is preferred.

1.4.2 Selecting the operator being evaluated

Once the initial state of the evaluate-operator operator is created, a copy of the operator being evaluated needs to be selected. The reason a copy is necessary is two fold. First, the original operator may have augmentations that refer to objects in the original (non-copied) state. Therefore, a copy of the operator must be made with those new objects. In the Water Jug, the operator for fill has an augmentation for the jug being filled: `(<o> ^name fill ^jug j1)`. The second reason is that in course of applying some operators, augmentations are added to the operator, which in this case would modify the original operator before it is applied. For these reasons, a duplicate of the operator is automatically made (unless `^default-operator-copy no` is an augmentation on the problem space). The copying replaces any identifiers in the operator structure that were changed in the state copying process (in this case `j1` would be replaced by `j3`).

Once a copy is made, additional rules reject all of the other proposed operators so that the copy will be selected.

1.4.3 Applying the selected operator

Once the operator is selected, the rules for applying it will fire and modify the copied state. You do not need to add any addition rules or modify existing rules. For tasks where the original operators involve action in an external world, such as in Eaters or TankSoar, you would have to write rules to simulate the effects of the selected operator on the state. For example, in Eaters, rules would have to be written to simulate the movement of the eater to a new square. The simulation can be only approximate because the eater does not have access to the complete world map so it cannot update sensors for squares it has not seen.

1.4.4 Evaluating the result

Once the new state is created, an evaluation can be made. An augmentation is added to the state in parallel to the operator being applied: `^tried-tied-operator <o>.` This augmentation can be tested by rules to ensure that they are evaluating the result of applying the operator as opposed to the copy of the original state – although this will not work for operators that apply as a sequence of rules.

The simplest evaluations to compute are success and failure. Success is when the goal is achieved. You have already written a rule that detects success for the Water Jug; however, this just halts the problem solving. This rule can be modified to create a working memory element that is used as an evaluation. The selection rules will automatically process an evaluation on a state in an evaluation state. The evaluation has two parts. The attribute is the type of evaluation and the value must be the identifier of the desired state. This ensures that the evaluation is done relative to the correct desired state. Therefore, the rule for detecting success should be modified to be as follows:

```
sp {water-jug*evaluate*state*success
   (state <s> ^desired <d>
              ^problem-space.name water-jug
              ^jug <j>)
   (<d> ^jug <dj>)
   (<dj> ^volume <v> ^contents <c>)
   (<j> ^volume <v> ^contents <c>)
   -->
   (<s> ^success <d>)}
```

In addition to success and failure, the selection rules can process other symbolic evaluations as well as numeric evaluations. The symbolic preferences that are processed are as follows:

- success: This state is the desired state. This is translated into a best preference. It is also translated into a better preference if another operator has a result state with an evaluation of partial-success.
- partial-success: This state is on the path to success. This is translated into a best preference.
- indifferent: This state is known to be neither success of failure. This is translated into an indifference preference.
- failure: The desired state cannot be achieved from this state. This is translated into a reject preference.
- partial-failure: All paths from this state lead to failure. This is translated into a worst preference.

For numeric evaluations, an augmentation named `^numeric-value` should be created for the evaluation object for an operator. We will discuss numeric evaluations in more detail in a future section.

If you include your original rules, the selection rules, and the two new rules described above (`water-jug*elaborate*problem-space, water-jug*evaluate*state*success`), your system will start doing a look-ahead search. You will notice that at each operator selection there is a tie and a recursive stack of substates is created. If the solution is ever found, it is only used locally to make select the operator for the most recent tie impasse. Another problem is that the system will often revisit the same state and generate an identical tie impasse deeper in the stack of states. The resolution of the first problem will wait until we introduce chunking.

You can resolve the second problem by evaluating states that are already in the stack as failure. That means that if an operator is being evaluated, then if it leads back to a state that has already been considered, it leads to failure. Not only will that lead to avoiding long loops but it will also mean that the operator that undoes the most recent operator will be avoided. The good news is that we can eliminate all of the rules for remembering the last operator.

What does this rule need to have as its conditions? It needs to test the following:

1. The desired state, which is used in the action: `(<s2> ^desired <d>)`
2. The contents of the state.
3. That state exists *after* the operator has been applied in an evaluation. Remember from earlier in this section that the correct test for this is: `(<s2> ^tried-tied-operator)`
4. That there is a duplicate of that state that is not the same state.

With only those tests, the rule could just as well evaluate the earlier state as failure. You need to add a condition that tests that makes sure that the state that will be labeled as failure is the most recent state.

5. That state is the only state for which there is not a substate with a superstate augmentation for it.

Thus, the rule is as follows:

```
sp {water-jug*evaluate*state*failure*duplicate
   (state <s1> ^desired <d>
               ^jug <j1>
               ^jug <j2>
               ^tried-tied-operator)
   (<j1> ^volume 5 ^contents <c1>)
   (<j2> ^volume 3 ^contents <c2>)
   (state { <> <s1> <s2> }
               ^jug <i1>
               ^jug <i2>)
   (<i1> ^volume 5 ^contents <c1>)
   (<i2> ^volume 3 ^contents <c2>)
  -(state <s3> ^superstate <s1>)
   -->
   (<s1> ^failure <d>)}
```

With this rule added, the problem solving will be more directed, but it will take an excessive number of decisions to solve the problem. The problem is that as soon as a result is produced, such as that in a given state a specific operator leads to failure, it is forgotten so that the next time that state is encountered, the evaluation must be repeated. What is needed is some memory that relates a state and operator with a result, which is essentially a rule. Conceptually, the conditions that should be included in the rule are templates of the working memory elements that were tested in the processing that produced the result. This is exactly what chunking does.

1.4.5 Chunking

Chunking is invoked when a result is produced in a substate. The result will be the action of a new rule. Chunking examines the working memory elements that were matched by the rule that created the result. If any of those working memory elements are linked to superstates, they become conditions in the new rule. For each of the remaining working memory elements, chunking finds the rule and associated working memory elements that were responsible for generating it and recursively *backtraces* until all of the conditions are collected. Some of the working memory elements tested in conditions require special processing. For example, if one of the tied operators is tested via an item augmentation of a state is tested in a rule that leads to the inclusion of a test for an acceptable preference for that operator in the superstate.

This process requires that Soar maintain a copy of every instantiated rule firing in a substate. Once all of the conditions have been collected, Soar converts all identifiers that were in the working memory elements into variables and adds the rule to rule memory. The rule is immediately available for matching.

Because chunking is based on the problem solving in a subgoal, the generality of any rule is determined by what was tested by the rules that participated in producing the result. If the original rules tested working memory elements that were not necessary to produce the result, then the chunk will also test for them and the new rule will be overly specific.

To invoke chunking, type in: `learn –on` at the prompt of the interaction window or add `learn –on` as a line in the file containing your rules. You will also want to see when chunks are created and fire, which is enabled using: `watch –chunks –print.` Before rerunning your program, try to predict what types of rules will be learned. There should be two different types of rules because there are two different types of results: preferences created by comparing evaluations in the selection problem, and evaluations created in the evaluation substate. Rules for the first type will test that there is an acceptable preference for the operator, as well as features of the operator and state that led to the creation of the preference. Rules of the second type will test that there is an evaluate-operator operator selected and features of the operator and state being evaluated.

Now run and examine the trace. Print out the chunks.

2. The Missionaries and Cannibals Problem

Now that you have done the conversion of the Water Jug to a simple planner, it should be relatively straightforward to convert the missionaries and cannibals. In this section we go through the conversion quickly, highlighting a problem, and then exploring the use of numeric evaluations in addition to symbolic evaluations.

2.1 Conversion to Planning and Learning

You should try to do the conversion yourself. There is one very subtle problem that will arise when you start to use learning. When you run into trouble, try to figure it out, and then read the rest of this section.

2.1.1 Add selection rules

The first step is to add the following to your program so that the selection rules are added in:

```
pushd ../default
source selection.soar
popd
```

2.1.2 Add problem space and state copying information

The second step is to add the rule that defines the problem space and specifies how the state copying should be done for evaluation. If you look back at your implementation of Missionaries and Cannibals, you will see that the operator application rules modify augmentations of the left-bank and right-bank structures, so that a two-level attribute copy is necessary.

```
      sp {mac*elaborate*problem-space
         (state <s> ^name mac)
         -->
         (<s> ^problem-space <p>)
         (<p> ^name missionaries-and-cannibals
              ^default-state-copy yes
              ^two-level-attributes right-bank left-bank)}
```

2.1.3 Modify goal detection

The third step is to modify the goal detection rule so that it creates a symbolic evaluation of success instead of printing a message and halting.

```
      sp {mac*detect*state*success
         (state <s> ^desired <d>
                    ^<bank> <ls>)
         (<ls> ^missionaries <m>
               ^cannibals <c>)
         (<d> ^{ << right-bank left-bank >> <bank> } <dls>)
         (<dls> ^missionaries <m>
                ^cannibals <c>)
         -->
         (<s> ^success <d>)}
```

2.1.4 Modify failure detection

Although there were no failure states in the Water Jug, there are in the Missionaries and Cannibals problem. The action of the rule that detects failure must be modified to create a symbolic value of failure.

```
sp {mac*evaluate*state*failure*more*cannibals
   (state <s> ^desired <d>
              ^<< right-bank left-bank >> <bank>)
   (<bank> ^missionaries { <n> > 0 }
           ^cannibals > <n>)
   -->
   (<s> ^failure <d>)}
```

2.1.5 Add duplicate state detection rule

The fifth step is to add a rule that detects when there are duplicate states in the state stack and evaluates the most recent one as a failure.

```
sp {mac*evaluate*state*failure*duplicate
   (state <s1> ^desired <d>
               ^right-bank <rb>
               ^left-bank <lb>)
   (<rb> ^missionaries <rbm> ^cannibals <rbc> ^boat <rbb>)
   (<lb> ^missionaries <lbm> ^cannibals <lbc> ^boat <lbb>)
   (state { <> <s1> <s2> }
          ^right-bank <rb2>
          ^left-bank <lb2>
          ^tried-tied-operator)
   (<rb2> ^missionaries <rbm> ^cannibals <rbc> ^boat <rbb>)
   (<lb2> ^missionaries <lbm> ^cannibals <lbc> ^boat <lbb>)
  -(state <s3> ^superstate <s2>)
   -->
   (<s2> ^failure <d>)}
```

2.1.6 Remove last-operator rules

The last-operator rules are no longer necessary because the planning and duplicate state detect replace (and improve) the processing they were designed for.

After making these six changes, your program should be able to solve the Missionaries and Cannibals problem with planning. However, just as with the Water Jugs problem, it may take a very long time. The obvious solution is to use chunking, Soar’s learning mechanism. The next section describes a subtle problem that arises when chunking is used.

2.2 Subtle chunking problem

When you run your program with chunking enabled, it might stop in the middle of solving the problem. The reason for this is that with the current rules for operator implementation, chunking can learn some control rules that are overgeneral, rejecting an operator that is on the path to the goal. Specifically, chunking can learn a rule that states:

```
If the current state has the boat on the right bank and two missionaries and
two cannibals, reject an operator that moves one missionary and the boat to
the left.
```

This rule is learned when the operator to move one missionary to the left is evaluated in a evaluation subgoal and discovered to lead to a failure state – where there is one missionary and two cannibals on the right bank. The problem with this rule is that it doesn't include a test that no cannibals are also moved. If the operator moves a cannibal at the same time it moves a missionary, it does not produce a failure state. However, with the way the application rule is written, chunking is unable to pick up any test that no cannibals are being moved. A satisfactory fix to this problem is to add a test to the operator application rules for the number of types being modified by the operator (`^types`).

```
      sp {mac*apply*move-mac-boat
         (state <s> ^operator <o>)
         (<o> ^name move-mac-boat
              ^{ << missionaries cannibals boat >> <type> } <num>
              ^bank <bank>
              ^types <types>)
         (<bank> ^<type> <bank-num>
                 ^other-bank <obank>)
         (<obank> ^<type> <obank-num>)
         -->
         (<bank> ^<type> <bank-num> -
                         (- <bank-num> <num>))
         (<obank> ^<type> <obank-num> -
                          (+ <obank-num> <num>))}
```

Once this is included, the learned chunked changes to be:

If the current state has the boat on the right bank and two missionaries and two cannibals, reject an operator that moves *only* one missionary and the boat to the left.

Now the system should successfully solve the problem, and after chunking, solving it in the minimal number of steps.

2.3 Numeric Evaluations

Without chunking, Soar takes a very long time to solve the problem because it does not get any evaluation until the problem is solved, and then must resolve the problem many times. One approach to avoid the long search is to add in some evaluations of the states that don't require the final state to be achieved. All that is needed is a good evaluation function. A very simple evaluation function is to prefer states that have more missionaries and cannibals on the desired bank of the river. Thus, the evaluation is just the sum of the number of missionaries and cannibals on the desired bank of the river, with higher numbers preferred.

There are selection rules that do all of the domain-independent processing, such as comparing evaluations and creating the appropriate preferences. All you need to do is add a rule to compute the evaluation, and augment the desired state such that higher numbers are better. For example, the second of these changes leads to the following initialization rule:

```
sp {mac*elaborate*initial-state
   (state <s> ^name mac
              ^superstate nil)
   -->
   (<s> ^right-bank <r>
        ^left-bank <l>
        ^desired <d>)
   (<r> ^missionaries 0
        ^cannibals 0
        ^boat 0
        ^other-bank <l>)
   (<l> ^missionaries 3
        ^cannibals 3
        ^boat 1
        ^other-bank <r>)
   (<d> ^right-bank <dl>
        ^better higher)
   (<dl> ^missionaries 3
         ^cannibals 3
         ^boat 1)}
```

New augmentation

The rule that computes the evaluation must test the desired state to determine which bank is the desired one. It must also match the number of missionaries and cannibals on that bank, as well as test that the operator being evaluated has applied (`^tried-tied-operator`). The action of the operator is to create an augmentation on the state with the computed evaluation.

```
sp {mac*evaluate*state*number
   (state <s> ^desired <d>
              ^tried-tied-operator
              ^<bank> <ls>)
   (<ls> ^missionaries <m>
         ^cannibals <c>)
   (<d> ^{ << right-bank left-bank >> <bank> } <dls>)
   -->
   (<s> ^numeric-value (+ <m> <c>))}
```

Now run your system. Without chunking, the solution is found much faster with this rule. However, once chunking is used, an interesting thing happens. This rule actually hurts performance. The reason is that the evaluation is only a heuristic and is sometimes wrong. Chunking captures both the good and the bad aspects of the evaluation and uses them for selecting the operators. When this evaluation is not used, all of the learning is based on absolute features of the task: success and failure. There are no heuristics that are sometimes wrong, so the rules are always correct. The lesson is that chunking is only as good as the problem solving and that sometimes it is not worthwhile to put in partly correct information.