Part III: Problem Solving

This part of the tutorial will teach you how to build Soar programs that solve problems through internal search. You will build Soar programs for two classic AI problems: Water Jug and Missionaries and Cannibals. Other classic AI problems, including Blocks World, the Eight Puzzle, and Towers of Hanoi are included in the set of demonstration programs that comes with the Soar release. The Soar programs you develop will solve these problems through search and the manipulation of internal data structures. This will teach you more about internal representations of states and creating operators that manipulate internal data structures. You will start by building the operators, state descriptions, and goal tests that are required to define each problem. You will also be introduced to more of the theory of problem solving based on search in problem spaces. In Part IV, you will learn how to modify and extend to your programs so that they use planning and learning to solve problems.

These problems are challenging in a different way than playing games such as Eaters and TankSoar. Eaters and TankSoar are competitive games and they require fast intelligent responses to the current situation, which can change quickly. The problems covered here do not have dynamic environments. However, solving these problems requires selecting the one appropriate operator from a set of many at each decision point. Selecting the correct operator is not easy given the knowledge available from the problem description. The problems can be solved only through trial and error, which involves searching through the space of possible states.

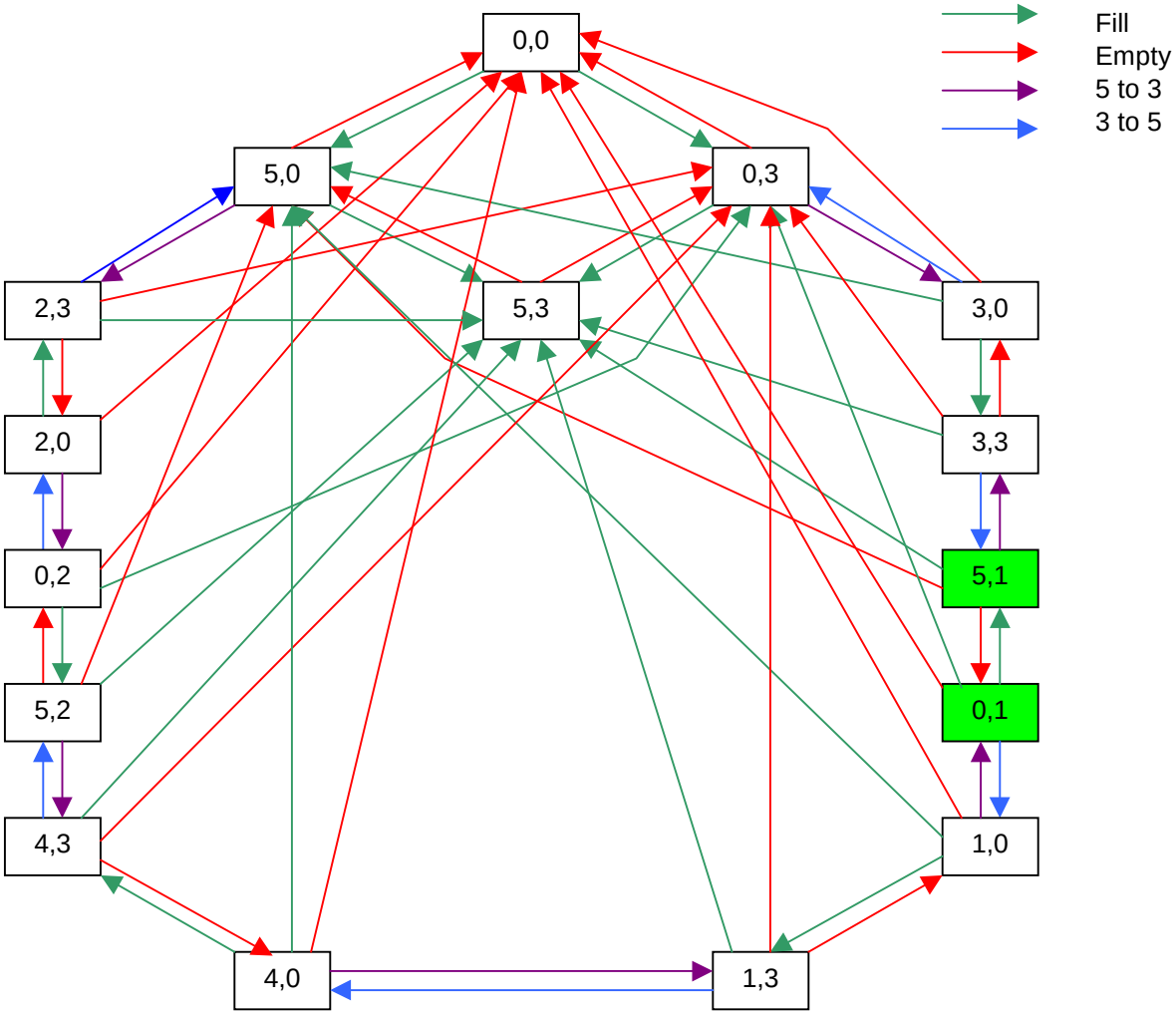
1. Water Jug

Problem Statement:

> *You are given two empty jugs. One holds five gallons of water and the other holds three gallons. There is a well that has unlimited water that you can use to completely fill the jugs. You can also empty a jug or pour water from one jug to another. There are no marks for intermediate levels on the jugs. The goal is to fill the three-gallon jug with one gallon of water.*

The first step in formulating a problem for solution is determining the space of possible states that the problem solver can be in. The space of states, or *problem space,* is determined by the objects that are available to be manipulated (the two jugs) and their possible values (0-5 gallons full), which we call the state representation, and the set of transformations that can be applied to them (fill, empty, pour), which we call the operators. A problem is defined as an initial state that the problem solver starts at, and of a set of desired states, any one of which can be achieved to solve the problem.

The diagram below shows all the states and operator transitions in the Water Jug problem space. Each state contains the contents of the five-gallon and three-gallon jugs in order. The operator transitions are colored coded. The goal states are shared in light green. In this problem, there are two distinct paths to the solution, one path much shorter (4 steps) than the other path (10 steps). The difficulty of the problem stems from the fact that usually four operators can apply to each state but only one leads toward the solution. The remaining operators lead back toward one of four states, all of which are at least three steps from the solution. Therefore it is unlikely that this problem will be solved quickly via a random search.



A problem is solved by starting from the initial state and then searching for a desired state by selecting and applying operators. Although the states and the operators form a graph such as in the prior diagram, you shouldn't think of problem solving as examining and searching an explicit graph data structure. In problem solving, the complete graph can be infinite is rarely available for the problem solver to peruse. You should instead consider the problem solver as being at a specific state and being faced with the selection of an operator to apply to the state to move to a new state. Often AI planners reformulate problems so that they are searching through a space of plans (and not states of the problem). In those cases, the states are partial plans and the operators modify the plan by adding, removing, or reordering plan steps. For the problems we consider in this tutorial, the problem solver will always be in the space of states of the problem (task states) and not in a planning space.

Given a specific problem spaces, many different problems can be defined by selecting different the initial and desired states. For example, you could define the problem of starting with both jugs full and getting four gallons of water into the five-gallon jug. Should that problem be easier or harder than the original problem?

Just as the difficulty of solving a problem can vary with the definition of the problem, so can the difficulty vary with the definition of the problem space. The original problem becomes easy if we expand the problem space to include a jug that holds four gallons of water (changing the objects in the space), and it becomes impossible if there is no pour operator.

Soar attempts to organize knowledge in terms of the concepts described above: states, operators, and so on. To formulate a problem for Soar means defining the components of the problem space: the state representation and the operators (proposal and applications); and the components of the problem: the initial state, and a description of the desired states. We will also add rules to monitor the problem solving, and some general heuristics in the form of search-control rules to provide some direction to the search. All of these aspects of the problem space and problem are presented in the remainder of this section:

1. The state representation. These are the attributes and values that are used to describe the different states of the problem. For this problem the state must represent how much water is in each jug.
2. The initial state creation rule. A rule must generate the state where the problem solving starts. In this problem, the initial state has both jugs empty.
3. The operator proposal rules. These are the rules that propose the legal state transformations that can be made toward solving the problem. For this problem there are three classes of operators:
   a) Pour water from the well into a jug.
   b) Pour water from a jug to a jug.
   c) Pour water from a jug to the well.
4. The operator application rules. These are the rules that transform the state when an operator is selected.
5. The operator and state monitoring rules. These are optional rules that print out the operator as it applies and prints out the current state.
6. The desired state recognition rule. This is a rule that notices when one of the desired states is achieved. In this problem, the desired states have one gallon of water in the three-gallon jug.
7. The search control rules. These are optional rules that prefer the selection of one operator over another. Their purpose is to avoid useless operators and/or direct the search toward the desired state. Theoretically you could encode rules that select the correct operator for each state. However, you would have had to already solved the problem yourself to come up with those rules. Our goal is to have the program solve the problem, using only knowledge available from the problem statement and possibly some general knowledge about problem solving. Therefore, search control will be restricted to general problem solving heuristics.

1.1 State Representation

For Eaters and TankSoar, the state structure was defined by the structure of the input-link. For this problem, Soar will solve the problem through the manipulation of internal data structures. We could create a simple world for your program to work in, with a well, water, and two jugs. This world could possibly simulate pouring water, or just writing the contents of the two jugs on a piece of paper; however it is possible to solve the problem internally to Soar. This will give you practice designing Soar programs that manipulate internal data structures. Moreover, this will prepare you for planning, where your program will need to create an internal representation of the problem even if there is an external world.

What are the parts of the problem that must be represented on the state? As mentioned above, the only information that needs to be represented in the state is the amount of water in each of the two jugs. So an initial state representation might include:

- The amount of water each jug currently holds.

You also need a way to distinguish the two jugs and to represent that one jug holds three gallons and the other holds five gallons. One way to do this is to include the volume of the jug in the attribute. Under this scheme, the initial state would look something like:

```
(state <s> ^five-gallon-jug-contents 0
           ^three-gallon-jug-contents 0)
```

This is a very compact representation; however you will have to write very specific rules for proposing and applying operators. You will need one set of rules to propose the operators for the five-gallon jug and one for the three-gallon jug. A more general approach would be to represent each jug as an object that had two attributes:

- The amount of water each jug currently holds (`^contents`).
- The amount of water each jug can hold (`^volume`).

In this scheme, the initial state would look something like this.

```
(state <s> ^jug <j1>
           ^jug <j2>)
(<j1> ^volume 5
      ^contents 0)
(<j2> ^volume 3
      ^contents 0)
```

Using this representation, you can write one set of rules for both jugs. Although this representation is sufficient for defining and solving the problem, the rules can be simplified if you add an attribute for the amount of water that can be added to a jug to fill it (`^free`). Therefore the state should include the following:

- An object for each jug (`^jug`).
- The amount of water each jug can hold (`^volume`).
- The amount of water each jug currently holds (`^contents`).
- The amount of free space available in each jug (`^free`).

In this scheme, the initial state looks something like the following.

```
(state <s> ^jug <j1> <j2>)
(<j1> ^volume 3
      ^contents 0
      ^free 3)
(<j2> ^volume 5
      ^contents 0
      ^free 5)
```

1.2 Initial State Creation

Given this state representation, it is straightforward to write a single rule that creates the initial state. This rule should also augment the state with a name (waterjug) for the problem. This name can then be matched by the other waterjug rules.

```
sp {waterjug*elaborate*state*initial
   (state <s> ^superstate nil)
   -->
   (<s> ^name waterjug
        ^jug <i> <j>)
   (<i> ^volume 3
        ^contents 0
        ^free 3)
   (<j> ^volume 5
        ^contents 0
        ^free 5)}
```

1.3 Operator Proposal

The operators for this task can be broken into three categories. For each category, tests are listed that must be true for the operator to have any effect.

- *Fill* a jug with water from the well
  Test that the jug is not full.
- *Empty* the water from a jug to the well
  Test that there is water in the jug.
- *Pour* water from one jug to another jug
  Test that there is water in the source jug and that the destination jug is not full.

Try to write English descriptions of the proposal for first operator. It should include a test for the problem, which is encoded in the name of the state, and it should test that the jug is not full.

```
waterjug*propose*fill
If the name of the state is waterjug and there is a jug that is not full,
then propose filling that jug.
```

The other operator proposals are very similar:

```
waterjug*propose*empty
If the name of the state is waterjug and there is a jug with water in it then
propose emptying that jug.

waterjug*propose*pour
If the name of the state is waterjug and there is a jug that is not full and
the other jug has water in it, then propose pouring water from the second jug
into the first jug.
```

Before translating this into Soar, consider what the structure of the proposed operator will be in working memory. Unlike most of the operators in Eaters or TankSoar, this operator will not have actions performed via the output-link in some external world that are later sensed via the input-link. You want a representation of the operator so that a few general rules can make the appropriate changes to the state. The trick to being able to use general rules is having an operator representation that has *parameters*. Parameters are augmentations of the operator that can differ for different ways of applying the operator. For this task, the operator parameters that make sense are:

- The name of the operator: `fill/empty/pour`.
- The jug that is being filled by the fill operator: `^jug`.
- The jug that is being emptied by the empty operator: `^jug`.
- The jug that is being poured out of by pour: `^jug`.
- The jug being poured into by pour: `^into`.

The operator representation for pouring from jug <j1> to <j2> would be:

```
      (<o> ^name pour
           ^jug <j1>
           ^into <j2>)
```

Now try to write the first proposal as a Soar rule for fill. The only test besides the name of the state is that the jug is not full.

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

Now try to write the second proposal for empty. This requires only minimal changes to the first so that the test is that the jug is not empty.

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

Now try to write the third proposal, which is for pour. This must test that one jug is not empty and the other jug is not full. You can combine the tests you used in the two previous rules.

```
sp {waterjug*propose*pour
    (state <s> ^name waterjug
               ^jug <i> { <> <i> <j> })
    (<i> ^contents > 0 )
    (<j> ^free > 0)
    -->
    (<s> ^operator <o> + =)
    (<o> ^name pour
         ^jug <i>
         ^into <j>)}
```

1.4 Operator Application

The next step is to write the operator application rules that will fire once an operator is selected. In Eaters and TankSoar, the operators were applied by sending output commands to the games. However, in this problem, your program must apply the operator directly and change the state by adding and removing working memory elements to reflect the pouring of the water.

What keeps the operator application productions from applying over and over again? Once all of the operator application rules have fired and made changes to the state, the acceptable preference for the selected operator will be retracted because the working memory elements that were matched by the proposal rules will have changed. The changes to working memory will cause new matches of proposal rules, which will be followed by the selection of a new operator, its application, and so on.

In this problem, there are two different ways the operators can change the state representation. The most straightforward is to add and delete the `contents` and `free` attributes of the jug objects. However, the operators could also remove a jug and create a completely new jug object with the correct attributes. Although this is less efficient, it has some advantages for planning and we will return to this approach in Part IV. For now you should write operator application rules that modify the `contents` and `free` attributes of the jug objects.

In the Water Jug, there are three separate operators, and each ones requires its own operator application rules. We will examine them one at a time. The first is fill. Try to write an English version of its operator application rule. One thing different about operator application rules is that because they test that the operator is selected, it is not necessary for them to test that the name of the state is waterjug. By separating out the proposal from the application, you can write application rules that apply whenever the operator is selected, possibly for a completely different problem. This makes it possible to share operators across problems, but it means that you must be careful to use distinct names for all of the operators you are going to use on problems and subproblems. Because the name of the state is not tested in an operator application rule, the name of the state is not included in the name of the rule.

```
apply*fill
If the fill operator is selected for a given jug, then set that jug's
contents to be its volume and set its free to be 0.
```

Converting this to Soar rule is not very difficult. Try to write this rule yourself.

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

The rule to apply the empty operator is essentially the inverse of fill.

```
apply*empty
If the empty operator is selected for a given jug, then set that jug's
contents to be 0 and its free to be its volume

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

The pour operator requires two rules, one for the case where there will be some water left over in the jug that is being poured from, and the second for the case when there isn't.

```
Apply*pour*not-empty-source
If the pour operator is selected, and the contents of the jug being poured
from is less than or equal to the free amount of the into jug,
then set the contents of the source jug to 0, its free to its volume;
     set the contents of the into jug to the sum of the two jugs and its free
     to its free minus the original contents of the source jug.

      sp {apply*pour*not-empty-source
          (state <s> ^operator <o>)
          (<o> ^name pour
               ^jug <i>
               ^into <j>)
          (<j> ^volume <jvol>
               ^contents <jcon>
               ^free <jfree>)
          (<i> ^volume <ivol>
               ^contents <icon> <= <jfree>
               ^free <ifree>)
          -->
          (<i> ^contents 0 <icon> -
               ^free <ivol> <ifree> - )
          (<j> ^contents (+ <jcon> <icon>) <jcon> -
               ^free (- <jfree> <icon>) <jfree> - )}
```

```
Apply*pour*empty-source
If the pour operator is selected, and the contents of the jug being poured
from is greater than the free amount of the into jug,
then set the contents of the source jug to its original contents minus the
     free of the destination jug, its free to its original free plus the free
     of the destination jug;
     set the contents of the into jug to its volume, and set its free to 0.

      sp {apply*pour*empty-source
          (state <s> ^operator <o>)
          (<o> ^name pour
               ^jug <i>
               ^into <j>)
          (<i> ^volume <ivol>
               ^contents <icon> > <jfree>
               ^free <ifree>)
          (<j> ^volume <jvol>
               ^contents <jcon>
               ^free <jfree>)
          -->
          (<i> ^contents (- <icon> <jfree>) <icon> -
               ^free (+ <ifree> <jfree>) <ifree> - )
          (<j> ^contents <jvol> <jcon> -
               ^free 0 <jfree> - )}
```

Now that you have written both operator proposals and application rules, you can load them into Soar and try them out. It may be difficult to follow the problem solving, so before running your program you should probably add the monitoring rules described in the next section.

1.5 State and Operator Monitoring

Monitoring rules are useful for printing out the details of the operator being applied and the contents of each state. Below are four rules that monitor the selected operator and the state (one rule for each operator and one for the state).

```
sp {waterjug*monitor*state
    (state <s> ^jug <i> <j>)
    (<i> ^volume 3 ^contents <icon>)
    (<j> ^volume 5 ^contents <jcon>)
    -->
    (write (crlf) | 3:| <icon> | 5:| <jcon> )}

sp {waterjug*monitor*operator-application*empty
    (state <s> ^operator <o>)
    (<o> ^name empty
       ^jug.volume <volume>)
    -->
    (write | EMPTY(| <volume> |)|)}

sp {waterjug*monitor*operator-application*fill
    (state <s> ^operator <o>)
    (<o> ^name fill
       ^jug.volume <volume>)
    -->
    (write | FILL(| <volume> |)|)}

sp {waterjug*monitor*operator-application*pour
    (state <s> ^operator <o>)
    (<o> ^name pour
       ^jug <i>
       ^into <j>)
    (<i> ^volume <ivol> ^contents <icon>)
    (<j> ^volume <jvol> ^contents <jcon>)
    -->
    (write | POUR(| <ivol> |:| <icon> |,| <jvol> |:| <jcon> |)|)}
```

With these rules, your program will apply operators and pour water. At some point it might even reach the desired state; however, it will not recognize that state and it will just keep going.

1.6 Desired State Recognition

The final step in creating a program is generating a rule that recognizes when one of the desired states has been achieved. You need to write a rule that recognizes when the three-gallon jug has one gallon in it. The action of the rule should be to print out a message that the problem has been solved and halt.

Write an English version of this rule.

```
waterjug*detect*goal*achieved
If the name of the state is waterjug and there is a jug with volume three and
contents one, write that the problem has been solved and halt.
```

Translating this into Soar is relatively straightforward. Try to write your own before looking below.

```
sp {waterjug*detect*goal*achieved
   (state <s> ^name waterjug
              ^jug <j>)
   (<j> ^volume 3
        ^contents 1)
-->
   (write (crlf) |The problem has been solved.|)
   (halt)}
```

Another approach that is often used is to create a representation of the desired state in working memory and then create a rule that compares that desired state to the current state. There are two advantages to this approach. First, a new problem can be created by modifying working memory (possibly by getting input from a sensor) instead of changing a rule. Second, in many problems, the description of the desired state can be used to guide the search using a technique called *means-ends analysis*. Using means-ends analysis in Soar will be included in a future tutorial.

Below is an example rule that creates the desired state, which in this case is the three-gallon jug containing 1 gallon.

```
sp {waterjug*elaborate*state*desired-state
    (state <s> ^superstate nil
               ^name waterjug)
    -->
    (<s> ^desired.jug <k>)
    (<k> ^volume 3
         ^contents 1)}
```

Below is the modified goal recognition rule. It compares the contents and volume of the desired jug to one in the current state. This rule is correct only for desired states that describe a single jug. If the desired state required specific amounts in each of the two jugs, a more elaborate rule would be required.

```
sp {waterjug*detect*goal*achieved
    (state <s> ^name waterjug
               ^desired.jug <d>
               ^jug <j>)
    (<d> ^volume <v> ^contents <c>)
    (<j> ^volume <v> ^contents <c>)
    -->
   (write (crlf) |The problem has been solved.|)
   (halt)}
```

If you run this with the earlier rules, the program should halt at some point; however, it can take a long time. When I ran it ten times, it took an average of 168 decisions to solve the problem, with a range of 4 (the optimal!) to 428.

1.7 Search Control

In order to make the search more efficient; you need to add rules that prefer operators that have the best chance of leading to one of the desired states. There are few general heuristics that you can use in the water jug. The total number of possible states is only 16; however, they are highly connected, and it is difficult to avoid revisiting the same state over and over again unless a list of visited states is maintained. Maintaining such a list is possible in Soar, but it is not easy – it would require creating a copy of every state after an operator has applied and then comparing a new state to the states in the list. In the section on planning you will learn about an alternative approach that is a more natural way for Soar programs to avoid repeated visits to the same state. For now, we will concentrate avoiding undoing the last operator that was applied, such as emptying a jug right after it has been filled, filling a jug after it has been emptied, or pouring water from one jug into the other right after the opposite pouring has been done. These heuristics will be even more effective when we look at the Missionaries and Cannibals problem and planning, but for now they give us a chance to look at how to maintain a history of the last operator application.

In order to avoid undoing the last operator, the program must remember in the state each operator after it applies. In Soar, this memory is not automatic. The operator is retracted as soon as it applies. In order to have a record of the previous operator; you must add some rules that deliberately record the operator each time an operator is applied. The rules will be part of the operator application because they will test the operator in order to record it, which in turn will make the record persistent (which is what you want).

Recording an operator has two parts. The first is creating a structure on the state that is the memory of the most recent operator. The second is to remove any record of an older operator. Given the representation of the water jug operators in working memory, you will have to write one rule to record the last operator. The action of this rule should be to create an augmentation of the state with information on the operator that is being applied. It should not create a link to the operator because all of the substructure of the operator will be removed as soon as the rule that created operator retracts. In this problem it is sufficient to record the name of the operator and the jug augmentation. This is sufficient even for the pour operator since there are only two jugs Try to write English versions of these rules.

```
waterjug*apply*operator*record*last-operator
If an operator is selected, then create an augmentation of the state (last-
operator) with the name of the operator and a copy of the jug augmentation.
```

This can then be converted into Soar rules:

```
sp {waterjug*apply*operator*record*last-operator
    (state <s> ^operator <o>)
    (<o> ^name <name>
         ^jug <j>)
    -->
    (<s> ^last-operator <o1>)
    (<o1> ^name <name>
          ^jug <j>)}
```

The rule to remove old records has to test only if the name of the current operator and the jug augmentation are different because it is not possible in this task to apply an operator twice in a row.

`waterjug*apply*move-waterjug-boat*remove*old*last-operator`
If an operator is selected and last-operator has a different name and jug, then remove the last-operator.

```
sp {waterjug*apply*operator*remove*last-operator
    (state <s> ^operator <o>
               ^last-operator <lo>)
    (<o> ^name <name>
         ^jug <j>)
   -(<lo> ^name <name>
          ^jug <j>)
    -->
    (<s> ^last-operator <lo> -)}
```

Once you add these rules, add rules that avoid applying an operator that undoes the previous operator.

```
waterjug*select*operator*avoid*inverse*fill
If the last operator is empty then avoid a fill.

waterjug*select*operator*avoid*inverse*empty
If the last operator is fill then avoid an empty.

waterjug*select*operator*avoid*inverse*pour
If the last operator is pour from one jug then avoid a pour the opposite way.
```

```
sp {waterjug*select*operator*avoid*inverse*fill
    (state <s> ^name waterjug
               ^operator <o> +
               ^last-operator <lo>)
    (<o> ^name fill ^jug <i>)
    (<lo> ^name empty ^jug <i>)
    -->
    (<s> ^operator <o> <)}

sp {waterjug*select*operator*avoid*inverse*empty
    (state <s> ^name waterjug
               ^operator <o> +
               ^last-operator <lo>)
    (<o> ^name empty ^jug <i>)
    (<lo> ^name fill ^jug <i>)
    -->
    (<s> ^operator <o> <)}

sp {waterjug*select*avoid*inverse*pour
    (state <s> ^operator <o> +
               ^last-operator <lo>)
    (<o> ^name pour ^into <j>)
    (<lo> ^name pour ^jug <j>)
    -->
    (<s> ^operator <o> <)}
```

After adding these rules, your search should be faster, on average. My ten-run average went down from 168 to 108.

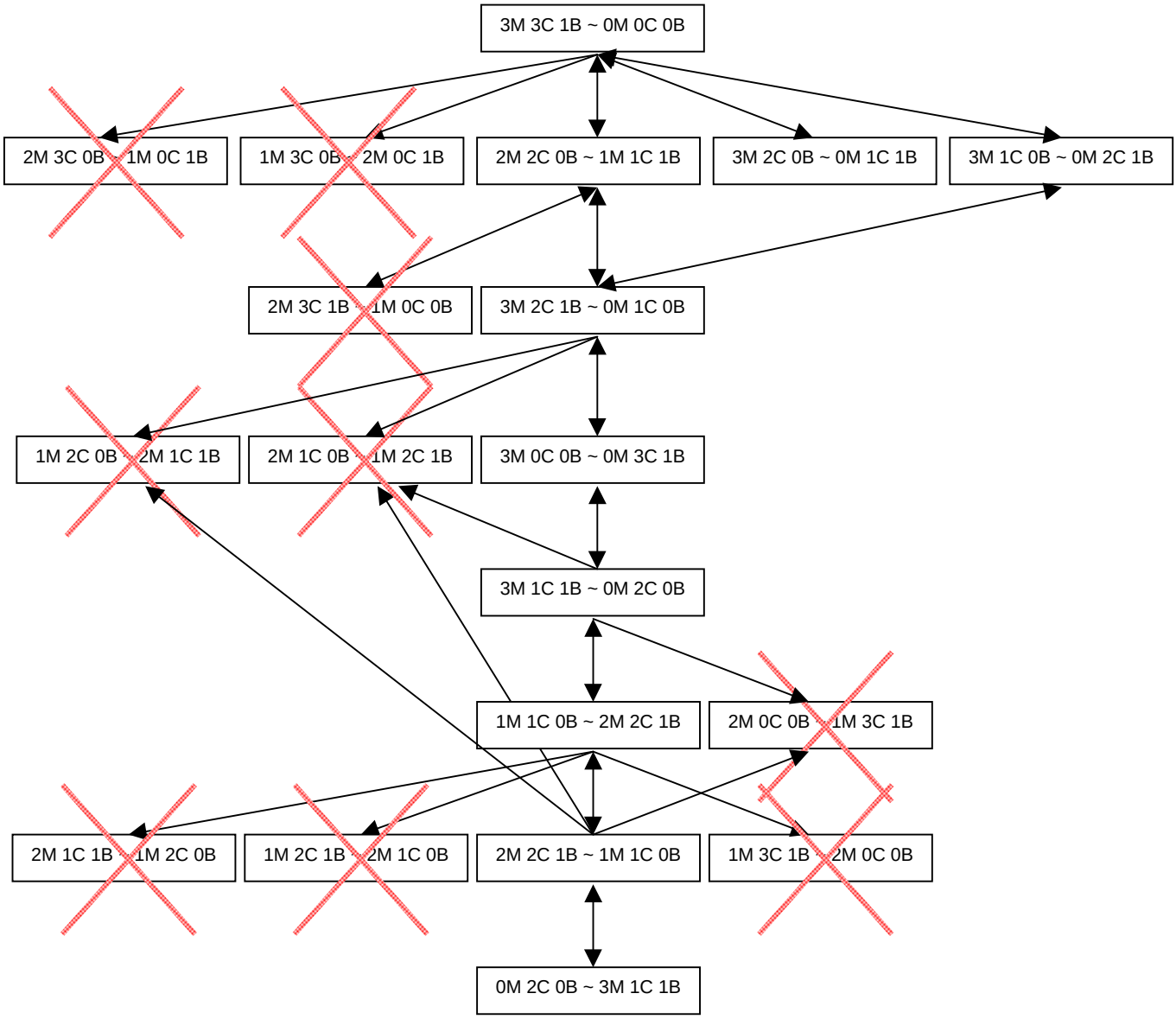
2. Missionaries and Cannibals

The second problem you are going to explore in this part is the Missionaries and Cannibals problem. This is another simple puzzle, but it has some additional complexities that make it worth covering.

Problem Statement:

> *Three missionaries and three cannibals come to a river. There is a boat on their bank of the river that can be used by either one or two persons at a time. This boat must be used to cross the river in such a way that cannibals never outnumber missionaries on either bank of the river. How do the missionaries and cannibals successfully cross the river?*

Once again, the first step to creating a Soar program to solve this problem is to decompose it into the problem space (state representation and operators) and the problem (initial state and desired state). One interesting aspect of this problem is that it also includes failure states. If the cannibals ever outnumber the missionaries, then you have failed. Below is a partial graph of the problem space, which shows that there are many illegal states that need to be avoided along the way to a solution.



Below is a list of the aspects of the problem space and problem that you will define for this problem:

1. The state representation. For this problem this will include the positions of the missionaries, cannibals, and boat, relative to the river.
2. The initial state creation rule. In this problem, all the missionaries, cannibals and the boat are on one bank of the river.
3. The operator proposal rules. For this problem the operators move up to two of the missionaries and/or cannibals across the river with the boat.
4. The operator application rules.
5. The operator and state monitoring rules.
6. The goal recognition rule. In this problem, the desired state is achieved when all missionaries and cannibals have crossed the river.
7. The failure recognition rule. These are rules that detect when a state is create in which the goal cannot be achieved. In this problem, the failure states are whenever the cannibals outnumber the missionaries on one bank of the river.
8. The search control rules.

It may be tempting to try to incorporate the avoidance of failure states into the operators, so that operators are never proposed that lead to failure states. However that is moving an aspect of the problem into the problem space and requires some problem solving to determine what the conditions of the proposal should be. We will see in Part IV how Soar can learn to rules that avoid proposing operators when they will lead to failure.

As in the Water Jug problem, this program does not create a plan to solve the problem. Instead, when the program finishes, all of the missionaries and cannibals will have been moved across the river.

2.1 State Representation

What are the parts of the problem that must be represented on the state? Everything in the problem description is important (there are no irrelevant objects are characteristics of the objects), so an initial list of objects includes three missionaries, three cannibals, a river, and the boat.

At any point in solving the problem, it is necessary to represent which bank of the river the boat is and the banks that the missionaries and cannibals are on. One important observation is that it is not necessary to keep track of each missionary and cannibal individually. All that is important is the *number* of missionaries and cannibals on each bank of the river. For the purposes of this problem all cannibals are the same and all missionaries are the same. Therefore, it is not necessary to have a separate representation of each missionary or cannibal and its current position. Another observation is that you never have to represent a state where the boat has missionaries and cannibals in it – that happens only during the application of an operator. The only states that need to be represented are those with the boat on one bank of the river or the other. Therefore the important aspects of the states that need to be represented are:

- The number of missionaries on each bank of the river.
- The number of cannibals on each bank of the river.
- The bank of the river that the boat is on.

There are many possible ways to represent this information using Soar's attributes and values. Try to come up with one on your own before looking at the representations listed below.

In creating the representations below, the two banks of the river are named left and right, with left being the bank of the river the missionaries and cannibals start out on.

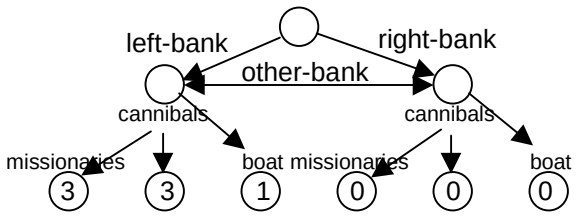
Here is one possible representation:

```
(state <s> ^right-bank-missionaries 0-3
           ^left-bank-missionaries 0-3
           ^right-bank-cannibals 0-3
           ^left-bank-cannibals 0-3
           ^right-bank-boat 0/1
           ^left-bank-boat 0/1)
```

Although this representation is adequate for solving the problem, it doesn't allow you to write general rules for proposing and applying operators. Using this representation, you would have to write separate proposal and application rules for when the boat is on each bank of the river. You would also have to write separate rules for moving cannibals and missionaries. When you have an attribute like right-bank-boat, the rules in Soar cannot match the different substructures, such as right-bank and boat. By representing each aspect separately using structured objects, you will find that it is possible to write very general operator proposal and application rules.

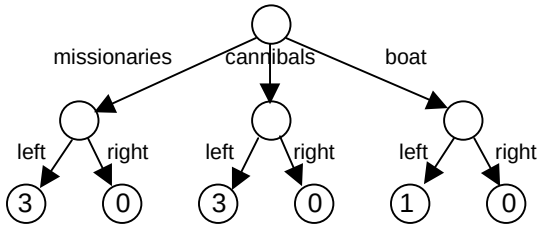
There are two obvious structured representations to choose from. One state representation has objects on the state for the two banks of the river, with subobjects for the missionaries, cannibals, and boat on that bank. To simplify later processing, an additional attribute (other-bank) can be added to the subobjects so that opposite bank can be matched easily. Below is a representation for the initial state:

```
(state <s> ^left-bank <l>
           ^right-bank <r>)
(<l> ^missionaries 3
     ^cannibals 3
     ^boat 1
     ^other-bank <l>)
(<r> ^missionaries 0
     ^cannibals 0
     ^boat 0
     ^other-bank <r>)
```



An alternative is to make the missionaries, cannibals, and boat the primary way of structuring the data, with the number of entities on each bank of the river as secondary.

```
(state <s> ^missionaries <m>
           ^cannibals <c>
           ^boat <b>)
(<m> ^left 3
     ^right 0)
(<c> ^left 3
     ^right 0)
(<b> ^left 1
     ^right 0)
```



For this problem, both of these representations are sufficient and they are similar in terms of the ease of writing the operators and goal tests. Soar programs for both are included as demonstration programs with the Soar release. For the remainder of this section, the first representation will be used because it is closer to the physical structure of the problem.

2.2 Initial State Creation

Just as in the Water Jug problem, you should write a rule to name the state, generate the initial state and generate the desired state. To simplify typing, name the state “mac”.

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

2.3 Operator Proposal

The operators for this task move 1 to 2 individuals (missionaries or cannibals) across the river. In writing the proposal rules, it is easiest to break the operators into three classes:

- move one missionary or cannibal to the other bank
  - test that there is at least one of the given type on the bank with the boat
- move two missionaries or two cannibals
  - test that there is at least two of the given type on the bank with the boat
- move one missionary and one cannibal together
  - test that there is at least one of each type on the bank with the boat

Try to write an English description of the proposal for first operator.

```
mac*propose*move-mac-boat*1
If the name of the state is mac and there is one or more cannibals or
missionary on the same bank of the river as the boat, then propose moving
that cannibal or missionary across the river.
```

The other operator proposals are very similar:

```
mac*propose*move-mac-boat*2
If the name of the state is mac and there are two or more cannibals or
missionaries on the same bank of the river as the boat, then propose moving
two of that type across the river.

mac*propose*move-mac-boat*1
If the name of the state is mac and there are one or more cannibals and one
or more missionaries on the same bank of the river as the boat, then propose
moving one cannibal and one missionary across the river.
```

As in the Water Jug problem, you need to decide on a representation of the operator and its parameters. For this task, the operator parameters that make sense are:

- The name of the operator: move-mac-boat.
- The type of entities being moved: cannibal or missionary.
- The number of each type of entity being moved: 1 or 2.

The second two can be combined as a single attribute-value pair, with the type of entity being the attribute and the number being the value. This makes it easy to represent moving one or two entities of the same type as well as moving one missionary and one cannibal. To simplify later matching, you can also include the bank of the river that the boat is on. Also, for some of the reasoning, it will be useful to also represent how many different types of people are being moved, usually just one, but two when moving one missionaries and one cannibal. It is not necessary to include the bank of the river that the boat is on because it is represented in the current state.

The operator representation for moving one cannibal (with the boat) from the bank with object `l3` would be:

```
      (<o> ^name move-mac-boat
           ^cannibals 1
           ^boat 1
           ^bank l3
           ^types 1)
```

Now try to write the first proposal as a Soar rule. To make it easier, a good approach is to initially write a very specific rule for one type of operator, and then attempt to generalize it by adding variables. To get started, you can try writing a proposal rule for just cannibals on the left bank of the river. That would be:

```
sp {mac*propose*move-mac-boat1
   (state <s> ^name mac
              ^left-bank <bank>)
   (<bank> ^cannibals > 0
        ^boat 1)
-->
   (<s> ^operator <o> + =)
   (<o> ^name move-mac-boat
        ^bank <bank>
        ^boat 1
        ^cannibals 1
        ^types 1)}
```

The operator is created with both an acceptable and a parallel preference. In a later section you will explore adding search control.

You can generalize this rule by using a variable for the bank of the river; making it so that the proposal applies no matter which bank the boat is on. To be safe, this requires introducing a disjunctive (`<< left-bank right-bank >>`) test for the attribute.

```
sp {mac*propose*move-mac-boat*1
   (state <s> ^name mac
              ^<< right-bank left-bank >> <bank>)
   (<bank> ^cannibals > 0
           ^boat 1)
-->
   (<s> ^operator <o> + =)
   (<o> ^name move-mac-boat
        ^bank <bank>
        ^boat 1
        ^cannibals 1
        ^types 1)}
```

You can then further generalize the rule so that it can match against both cannibals and missionaries (but not the boat). This requires introducing a disjunctive (`<< cannibals missionaries >>`) test for the attribute of the bank object, and also a surrounding conjunctive test (`{ << cannibals missionaries >> <type> }`) to match the entity type to the variable `<type>`, which can then be used in the action. The final rule is:

```
sp {mac*propose*operator*move-mac-boat1
   (state <s> ^name mac
              ^<< right-bank left-bank >> <bank>)
   (<bank> ^{ << cannibals missionaries >> <type> } > 0
           ^boat 1)
   -->
   (<s> ^operator <o>)
   (<o> ^name move-mac-boat
        ^bank <bank>
        ^<type> 1
        ^types 1)}
```

Now try to write the second proposal that moves two missionaries or two cannibals as a Soar rule. This requires only minimal changes to the first. The only changes are to test for more than one missionary or cannibal, and to increase the number being moved to 2.

```
sp {mac*propose*operator*move-mac-boat2
   (state <s> ^name mac
              ^ << right-bank left-bank >> <bank>)
   (<bank> ^{ << cannibals missionaries >> <type> } > 1
           ^boat 1)
   -->
   (<s> ^operator <o>)
   (<o> ^name move-mac-boat
        ^bank <bank>
        ^<type> 2
        ^types 1)}
```

Now try to write the third proposal for moving one missionary and one cannibal.

```
sp {mac*propose*operator*move-mac-boat11
   (state <s> ^name mac
              ^ << right-bank left-bank >> <bank>)
   (<bank> ^missionaries > 0
           ^cannibals > 0
           ^boat 1)
   -->
   (<s> ^operator <o>)
   (<o> ^name move-mac-boat
        ^bank <bank>
        ^missionaries 1
        ^cannibals 1
        ^types 2)}
```

2.4 Operator Application

The operator application rules must change the state to reflect the movement of the boat and the missionaries and cannibals that cross the river. As part of applying the operators, it is not necessary to represent that the missionaries and cannibals are in the boat, only that they change banks of the river. The changes that need to be made to the state are to decrease the number of missionaries and cannibals that are moving from the bank of the river that the boat is on, and increase the number on the bank the boat is moving to. Similarly, the count of the boat (0 or 1) must be changed. You might try to come up with a set of rules to do this, but because of the operator representation, a single rule can make changes for moving cannibals, missionaries and the boat from either bank of the river to the other. The rule must test for an augmentation of the operator, such as `^boat`, `^cannibals`, or `^missionaries`, and then change the corresponding subobject on the state. The rule will fire in parallel for all entities being moved, including the boat.

Below is an English version of the required rule.

```
mac*apply*move-mac-boat
If there is a move-mac-boat operator selected for a type and number, then
subtract the values of that type on the current bank and add those values to
the other bank.
```

Converting this to a Soar rule is a bit tricky because of all of the variables. To simplify the conversion, we will start with a rule that applies the operator for moving one cannibal. Try to write this rule yourself.

```
mac*apply*move-mac-boat*one*cannibal
If there is a move-mac-boat operator selected for one cannibal, then subtract
one from cannibal object on the left bank and add one to the cannibal object
on the other bank.
```

```
sp {apply*move-mac-boat*one*cannibal
   (state <s> ^operator <o>)
   (<o> ^name move-mac-boat
        ^cannibals 1
        ^bank <bank>)
   (<bank> ^cannibals <bank-num>
           ^other-bank <obank>)
   (<obank> ^cannibals <obank-num>)
   -->
   (<bank> ^cannibals <bank-num> -
                    (- <bank-num> 1))
   (<obank> ^cannibals <obank-num> -
                     (+ <obank-num> 1))}
```

The above rule tests the operator to ensure that one cannibal is being moved (`<o> ^cannibals 1`) and to detect the bank of the operator. It then matches the number of cannibals on that bank, matching `<bank-num>`, via the other-bank attribute, matching `<obank-num>`. The actions of the rule modify the number of cannibals on the left bank by rejecting the current value (`^cannibals <bank-num> -`), and by asserting the new value which is the original value minus one (`^cannibals (- <bank-num> 1)`). Arithmetic operations such as addition, subtraction, and multiplication are done in Soar using prefix notation where the operation is given first followed by the operands.

One concern you might have about the above rule is that it will apply multiple times if there is more than one cannibal on the left bank of the river, moving each cannibal, one by one to the other bank. However, that will not happen because immediately after this rule fires (multiple times in parallel for each entity

being moved) the rule that proposed the operator will no longer match, causing the operator to be removed. The operator proposal rule will no longer match because it tested the number of cannibals on the left bank of the river, which is changed by the rule. In addition, the boat will move from one bank to another by at the same time, providing a second reason for the proposal rule not to match. Thus, the operator will terminate immediately after the above rule fires.

The next step is to generalize this rule so it can apply to moving 1 or 2 cannibals. This requires replacing the test for `^cannibals 1` on the operator to `^cannibals <number>` and then using <number> in the actions to subtract from the current

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

1 is replaced by <number> so that rule applies to moving anv number of cannibals

The final generalization is to replace the test for the `^cannibals` attribute of the operator with a more general test that matches cannibals, missionaries, or the boat to a variable `<type>`. That variable is used to match the appropriate object on the state. This rule will now fire multiple times to move the boat as well as any cannibals or missionaries that are moving.

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

<type> matches the type of object being moved

2.5 State and Operator Monitoring

Below are three rules that monitor the selected operator and the state (one rule for each bank that the boat is on).

```
sp {monitor*move-mac-boat
   (state <s> ^operator <o>)
   (<o> ^name move-mac-boat
        ^{ << cannibals missionaries >>  <type> } <number>)
   -->
   (write | Move | <number> | | <type>)}

sp {monitor*state*left
   (state <s> ^name mac
              ^left-bank <l>
              ^right-bank <r>)
   (<l> ^missionaries <ml>
        ^cannibals <cl>
        ^boat 1)
   (<r> ^missionaries <mr>
        ^cannibals <cr>
        ^boat 0)
   -->
   (write (crlf) | M: | <ml> |, C: | <cl> | B ~~~   |
                 | M: | <mr> |, C: | <cr> |  |)}

sp {monitor*state*right
   (state <s> ^name mac
              ^left-bank <l>
              ^right-bank <r>)
   (<l> ^missionaries <ml>
        ^cannibals <cl>
        ^boat 0)
   (<r> ^missionaries <mr>
        ^cannibals <cr>
        ^boat 1)
   -->
   (write (crlf) | M: | <ml> |, C: | <cl> |   ~~~ B |
                 | M: | <mr> |, C: | <cr> |  |)}
```

When you run your program, you will observe that your program runs forever and also sometimes visits states that are illegal according to the problem statement.

2.6 Desired State Recognition

The next step in creating a program to solve missionaries and cannibals is creating a rule that recognizes when a desired state has been achieved. Although a rule specific to the given problem can easily be written, it might be better to write one that is more general. For example, you might assume that the desired state will always have some number of missionaries, cannibals, on one bank of the river. The action of the rule should be to print out a message that the problem has been solved and halt.

Write an English version of this rule.

```
mac*detect*goal*achieved
If the name of the state is mac and the number of missionaries and cannibals
on one bank of the river in the desired state matches the number of
missionaries and cannibals on the same bank in the current state, write that
the problem has been solved and halt.
```

Translating this into Soar is relatively straightforward. Try to write your own before looking below.

```
sp {mac*detect*state*success
   (state <s> ^desired <d>
              ^<side> <ls>)
   (<ls> ^missionaries <m>
         ^cannibals <c>)
   (<d> ^{ << right-bank left-bank >> <side> } <dls>)
   (<dls> ^missionaries <m>
          ^cannibals <c>)
   -->
      (write (crlf) |The problem has been solved.|)
   (halt)}
```

If you run this with the earlier rules, the program should halt at some point; however, it is likely that it will visit a failure state and thus solved the problem incorrectly.

2.7 State Failure Detection

The next step is creating a rule that recognizes when a failure state has been encountered. According to the problem statement, a failure state is one where the cannibals outnumber the missionaries on one bank of the river. One condition that is often forgotten is to test that the number of missionaries is greater than zero. The action for this rule is to print out a message that the problem has failed to be solved, and then halt. Write an English version of this rule.

```
mac*detect*goal*failure
If the name of the state is mac and there are more cannibals than
missionaries, and there is at least one missionary, on one bank of the river,
then write that the problem has failed to be solved, and halt.
```

Translating this into Soar is relatively straightforward. Try to write your own before looking below.

```
sp {mac*evaluate*state*failure*more*cannibals
   (state <s> ^desired <d>
              ^<< right-bank left-bank >> <bank>)
   (<bank> ^missionaries { <n> > 0 }
           ^cannibals > <n>)
   -->
  (write (crlf) |The problem has failed.|)
  (halt)}
```

Try running your complete program. Invariably the program will halt with failure because of the high likelihood of encountering a failure state.

2.8 Search Control: Undoing the Last Operator

In the current problem, when a failure is reached, the program halts. One possibility is to have the program start over again from the initial state. But if you were working on the problem, you would probably notice that you reached an illegal state, and you would go back one step by undoing the last operator and try to find another path. In order to undo the last operator, you must remember what it was. You can use the same of the work you did on the Water Jug where you created a memory to *avoid* undoing the last operator to *prefer* to undo an operator when a failure state is achieved.

Given the representation of the move-mac-boat operator in working memory, you will have to write two rules to record the last operator, one that handles instances of the operator that move a single type of entity, and a second that handles instances of the operator that move one missionary and one cannibal. The action of these rules should create an augmentation of the state with information on the operator that is being applied. Try to write English versions of these rules.

```
mac*apply*move-mac-boat*record*last-operator*types*1
If an operator is selected to move one type of entity, then create an
augmentation of the state (last-operator) with the bank of the boat, the type
of entity being moved, the number, and that there is one type being moved.

mac*apply*move-mac-boat*record*last-operator*types*2
If an operator is selected to move two types of entity, then create an
augmentation of the state (last-operator) with the bank of the boat and that
there is two types being moved.
```

These can then be converted into Soar rules:

```
sp {mac*apply*move-mac-boat*record*last-operator*types*1
   (state <s> ^name mac
              ^operator <o>)
   (<o> ^name move-mac-boat
        ^bank <bank>
        ^{ << missionaries cannibals >> <type> } <n>
        ^types 1)
   -->
   (<s> ^last-operator <o1>)
   (<o1> ^types 1
         ^bank <bank>
         ^type <type>
         ^number <n>)}

sp {mac*apply*move-mac-boat*record*last-operator*types*2
   (state <s> ^name mac
              ^operator <o>)
   (<o> ^name move-mac-boat
        ^boat <bank>
        ^types 2)
   -->
   (<s> ^last-operator <o1>)
   (<o1> ^types 2
         ^bank <bank>)}
```

The rule to remove old records only has to test if the bank of the boat in the record of the last operator does not match the current bank that the boat is on because each time an operator is applied the boat changes banks.

mac*apply*move-mac-boat*remove*old*last-operator
If the move-mac-boat operator is selected and the boat in the last-operator is not equal to the bank of the current boat, remove the last-operator structure.

This can then be converted into Soar a rule:

```
sp {mac*apply*move-mac-boat*remove*old*last-operator
   (state <s> ^name mac
              ^operator <o>
              ^<lr-bank>.other-bank <o-bank>
              ^last-operator <lo>)
   (<lo> ^bank <obank>)
   -->
   (<s> ^last-operator <lo> -)}
```

Once you add these rules, you can now add rules that undo an operator whenever one leads to a failure state. However, you must first modify the rule that detects failure so that it doesn't halt the program, but just augments the state with failure:

mac*detect*goal*failure
If the name of the state is mac and there are more cannibals than missionaries, and there is at least one missionary, on one bank of the river, then augment the state with failure true.

Translating this into Soar is relatively straightforward.

```
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>)}
```

Note that this rule only fires when there is an illegal state and it is not part of the application of an operator. Thus, it will retract and remove the failure augmentation automatically if the state changes and there is no longer an illegal state.

Now you can write rules that prefer operators that undo the last operator when there is failure. Just as before, this will require two rules, one for moving a single type of entity, and one that moves one missionary and one cannibal. Below is a general English version for both rules.

mac*select*operator*prefer*inverse*failure
If the name of the state is mac and the current state is a failure state and the last operator is the inverse of a proposed operator, then prefer that operator.

```
sp {mac*select*operator*prefer*inverse*failure*types*1
   (state <s> ^name mac
              ^operator <o> +
              ^failure <d>
              ^last-operator <lo>)
   (<o> ^name move-mac-boat
        ^<type> <number>
        ^types 1)
   (<lo> ^types 1
         ^type <type>
         ^number <number>)
   -->
   (<s> ^operator <o> >)}

sp {mac*select*operator*prefer*inverse*failure*types*1
   (state <s> ^name mac
              ^operator <o> +
              ^failure true
              ^last-operator.types 2)
   (<o> ^types 2)
   -->
   (<s> ^operator <o> >)}
```

After you have added these rules, your program will be able to solve the problem; however it will probably take a very indirect path to the solution. One reason is that after an operator has been successfully applied and generated a valid state, the inverse of that operator will often be selected, undoing the operator and wasting both operator applications. To avoid this, you can add two more rules that *avoid* undoing the last operator when the state is not a failure state.

```
mac*select*operator*avoid*inverse*not*failure
If the name of the state is mac and the current state is not a failure state
and the last operator is the inverse of a proposed operator, then avoid that
operator.

sp {mac*select*operator*avoid*inverse*not*failure*1
   (state <s> ^name mac
              ^operator <o> +
             -^failure true
              ^last-operator <lo>)
   (<o> ^types 1
        ^<type> <number>)
   (<lo> ^types 1
         ^type <type>
         ^number <number>)
   -->
   (<s> ^operator <o> < )}

sp {mac*select*operator*avoid*inverse*not*failure*2
   (state <s> ^name mac
              ^operator <o> +
             -^failure true
              ^last-operator <lo>)
   (<o> ^types 2)
   (<lo> ^types 2)
   -->
    (<s> ^operator <o> < )}
```

After you have added these final rules, your program should solve the problem much quicker. However, you will notice that there is still some inefficiency. For example, at the initial state, it is possible for the program to attempt to apply the same operator to the state after it has failed with moving one or both the other missionaries. The figure below shows the initial state and the three successive states that it can cycle among. The problem is that only the most recent operator for a current state is recorded. It is not possible to associate all prior operators that applied to a state because the state is continually changing. In the next Part of the tutorial, you will learn how to use impasses and substates so that your programs can use look-ahead planning and solve this type of problem more directly.