Introduction to Intelligent, Knowledge-Based, and Cognitive Systems

89-674

Mor Vered
http://u.cs.biu.ac.il/~veredm/89-674/
mvered89230@gmail.com
Sunday 10-11, Building 202, Room 119
Eaters Game

- No inside corners
- No dead-ends
- Random initial location
- Blue circle – 5 pt
- Red square – 10 pt
- Up, down, left, right
- When 2 eaters collide, their scores are averaged and they are transported to new random location.
- Can jump two squares – cost 5 pt
- Can sense up to 2 squares in every direction
Create an Eater

- Open ...\SoarTutorial_9.4.0-Windows_64bit\Eaters.bat
Create an Eater

• Example: Agents\Eaters\Tutorial\move-to-food.Soar
• What happens?

Press this button to change the production rules that are loaded when an Eater is created.

This label shows which productions will be loaded when the Eater is created.

After productions are selected, the “Create Agent” button will be enabled.
Create an Eater

• When letting the eater run for a long time, it will eventually get to a place where there is no food directly next to it. The move-to-food eater does not have any rules to respond to this situation.

• When an eater does not know how to respond to a situation, it will start generating new states – indicating the agent doesn’t know what to do.
Create Several Eaters

- Can create several eaters. New and upload new rules.
- To make a new agent with the same set of rules as an existing agent, you can use the “Clone” button.
Building A Simple Eater Using Rules

• This time an Eater will start with information about its situation coming in through a structure called the input-link.

• The input and output links in working memory are substructures of the io link.

• A simple eater – create only operator (proposal and application rules) that moves an eater north one step.
How to Move an Eater

• Create working memory elements that are augmentations of the output-link.

• Create a “move” augmentation on the output-link object, which in turn has an augmentation called “direction” with a value of the direction to move: “north”, “south”, “east”, or “west”.

• For each task in Soar, a set of output commands is defined, and in Eaters there are two commands: move and jump.
Building A Simple Eater Using Rules

Move-north operator

Propose*move-north:
If I exist, then propose the move-north operator.

Apply*move-north:
If the move-north operator is selected, then generate an output command to move north.
Propose*move*north

sp {propose*move-north
(state <s> ^type state)
→
(<s> ^operator <o> +)
(<o> ^name move-north)}
Apply*move*north

• The action is to add a move command to the output-link. This will cause the eater to move one cell to the north.

```lisp
(sp {apply*move-north
    (state <s> ^operator <o>
      ^io <io>)
    (<io> ^output-link <out>)
    (<o> ^name move-north)
    -->
    (<out> ^move <move>)
    (<move> ^direction north})
```

^io must be matched to get to the output-link

^output-link gives path to identifier for action

Add action to output-link to move north
Move North

• What happened?

• After selecting move-north and taking one step, the eater had nothing more to do and a new state is created. Unfortunately, the rules you wrote are not sufficient to get the eater to continually move north.
Operator Instance

• Each action in the world, such as moving an eater, should be performed by a separate operator instance.

• An operator instance is a separate operator in working memory element created by the firing of an operator proposal rule.

• Thus, each action should test for the creation of a new operator object in working memory.

• Different instances of the same operator will have the same name, and sometimes will have the same augmentations, but a given instance will be used for only one move.

• Some operator instances may include many actions, but they will be selected and applied only once.

• Therefore, new instances of the move-north operator should be created in working memory for each new move.
Operator Instance

- You should not attempt to have the move-north operator selected once, and have it move an eater multiple times.
- Instead, you should design you eater so that a new instance of the move-north operator is created for each move.
- You can do this by having the operator proposal rule fire each time the eater is to move.
- As the rule is currently written, it will fire only once because it only tests type state, which stays in working memory forever.
- Your rule needs to test working memory elements that change each time the eater moves – those working memory elements that correspond to the eater’s senses on the input-link.
• The input-link object, I2, has two augmentations:
  • 1) ^eater, has information about the eaters’ direction, name, score, and x, y coordinates.
  • 2) ^my-location, includes the eater’s sensing of nearby cells.
• Some of the ^eater information changes during the game: The x location will change when the eater moves east or west, y location will change when the eater moves north or south.

(I2 ^eater I4 ^my-location I5)
(I4 ^direction south ^name red ^score 25 ^x 1 ^y 10)
Propose*move*north

• We can modify the conditions of our proposal rule to test both of these working memory elements, and eliminate the test for ^type state.

```
sp {propose*move-north
 (state <s> ^io.input-link.eater <e>)
 (<e> ^x <x> ^y <y>)
 -->
 (s> ^operator <o> +)
 (<o> ^name move-north})
```

• When the original x,y WME’s are removed from working memory the original move-north operator will be retracted because the rule instantiation that created it no longer matches.

• A new instance of the move-north operator will be created because propose*move-north will match the new values of x and y and fire.
Propose*move*north

• When debugging we will see “=>” meaning an addition to working memory but we will not see “<=“ a removal from working memory.
• The old move command is not removed because it is the action of an operator application rule. Such actions are not automatically removed when the rule no longer matches.
• The output-link will gradually accumulate stale commands.
• Why?
• Because the rule that created the command is part of the application of an operator, and operator applications create persistent working memory elements.
Persistence

• Persistence is necessary for creating memories of prior events, such as the memory of something sensed in the environment.

• Example: to remember the location of uneaten food the eater would have to create a persistent structure in its working memory. Otherwise, as soon as it stopped sensing the food, it would forget about it.

• A rule is an operator application rule if it tests a selected operator and modifies the state.

• In Soar, all working memory elements created by an operator application rule are persistent.

• Persistent working memory elements are said to be operator-supported, or o-supported, because they are created by operators.
Persistence

• apply*move-north is an operator application rule and creates o-supported working memory elements

```
sp {apply*move-north
  (state <s> ^operator.name move-north
     ^io.output-link <out>))
  -->
  (<out> ^move.direction north)}
```

Tests selected operator

Actions are o-supported
Persistence

• propose*move-north is not an operator application rule because it does not test a selected operator.

• All of its actions are i-supported and they will be removed from working memory when propose*move-north no longer matches.

```
sp {propose*move-north
 (state <s> ^io.input-link.eater <e>)
 (<e> ^x <x> ^y <y>)
 -->
 (<s> ^operator <o> +)
 (<o> ^name move-north})
```
Persistence

• We need to add a rule that removes old move commands from the output-link after the move is finished.

• In Eaters, the output system creates an augmentation on the move object after the action has executed: ^status complete.

# Apply*move-north*remove-move:
# If the move-north operator is selected,
# and there is a completed move command on the output link,
# then remove that command.
Persistence

• The action of this rule removes the working memory element specified by \( <\text{out}> ^\text{move} <\text{move}> \)

• Rules fire (and retract) in parallel as a wave.

• During the propose phase, only rules that have i-supported actions will fire. During the apply phase, both o-supported and i-supported rules fire.
Soar Cycle

• The default will stop each step before the application phase, right after Soar has made an operator selection (decision phase). It’s possible to change the location where Soar will stop when stepping, using the Soar Cycle widget in the Debugger.

Using your mouse, pick up the hourglass shaped red icon and drag it to the location between World and Input.
Move To Food

• Create an eater that greedily moves to any food it senses
  • A rule to propose the operator when there is normal-food in a neighboring cell.
  • A second rule to propose the operator when bonus-food is in a neighboring cell.
  • A third rule to apply the operator and move the eater in the correct direction.
  • A fourth rule to remove the move command from the output-link.
Move To Food Operator

# Move-to-food operator

# Propose move-to-food*normalfood
# If there is normalfood in an adjacent cell,
#   propose move-to-food in the direction of that cell
#   and indicate that this operator can be selected randomly.
#
# Propose move-to-food*bonusfood
# If there is bonusfood in an adjacent cell,
#   propose move-to-food in the direction of that cell
#   and indicate that this operator can be selected randomly.
#
# Apply move-to-food
# If the move-to-food operator for a direction is selected,
#   generate an output command to move in that direction.
#
# Apply move-to-food*remove-move:
# If the move-to-food operator is selected,
#   and there is a completed move command on the output link,
#   then remove that command.
Eaters World Augmentations

• Adjacent cells are augmentations labeled north, east, south, and west.

• Every cell also has a ^content augmentation, whose value can be wall, empty, eater, normalfood, or bonusfood.

• If the cell has an eater in it, there will be additional augmentations for the color of the eater (^eater-color), and the eater’s current score (^eater-score).
Move To Food Proposal Rule

• If there was normalfood to the south and west, two operators would be created, one with ^direction south and one with ^direction west.
Move To Food Apply Rule

```plaintext
sp {apply*move-to-food
  (state <s> ^io.output-link <out>
    ^operator <o>)
  (<o> ^name move-to-food
    ^direction <dir>)
  -->
  (<out> ^move.direction <dir>)}
```
Move To Food Removal Rule

```plaintext
sp {apply*move-to-food*remove-move
  (state <s> ^io.output-link <out>
    ^operator.name move-to-food)
  (<out> ^move <move>)
  (<move> ^status complete)
  -->
  (<out> ^move <move> -)
}
```
Generalized Move Operator

• The problem is the former move would get stuck when there is no food near by.

# Propose*move*1a:
# If there is normalfood, bonusfood, eater, or empty in an adjacent cell,
# propose move in the direction of that cell, with the cell’s content,
# and indicate that this operator can be selected randomly.
Instead of writing an individual rule for each value, it is possible to write a single rule that tests for any one of a set of alternative values. The alternative values are written in the same position as a single value, but are surrounded by double angle brackets: << normalfood bonusfood >>.
Generalized Move Proposal

```plaintext
sp {propose*move*2a
  (state <s> ^io.input-link.my-location.<dir> .content
   { <content> <> wall })

  -->
  (<s> ^operator <o> + =)
  (<o> ^name move
   ^direction <dir>
   ^content <content>))
```

Will match any value except wall.
Generalized Move Apply/Remove

# Apply*move
# If the move operator for a direction is selected,
#   generate an output command to move in that direction.

    sp {apply*move
      (state <s> ^io.output-link <out>
       ^operator <o>)
      (<o> ^name move
       ^direction <dir>)
      -->
      (<out> ^move.direction <dir>))

# Apply*move*remove-move:
# If the move operator is selected,
#   and there is a completed move command on the output link,
#   then remove that command.

    sp {apply*move*remove-move
      (state <s> ^io.output-link <out>
       ^operator.name move)
      (<out> ^move <direction>)
      (<direction> ^status complete)
      -->
      (<out> ^move <direction> -)})
Adding Preferences

• To improve the performance of the eater, we can add rules that prefer moving to bonusfood over normalfood or an empty cell and prefer moving to normalfood over moving into an empty cell or a cell with another eater.

• To get started, you need to create a rule to prefer bonusfood to normalfood or empty or an eater. The condition part of the rule must match against operator proposals, while the action part must prefer the operator that moves to the bonusfood.
Adding Preferences

• An operator can be preferred by creating a *better than* preference
• The better preference is the greater than sign: “>”
Adding Preferences

If there are adjacent cells with both bonusfood and normalfood, this rule will fire right after propose*move creates acceptable preferences, but during the same proposal phase so that it will influence the next operator selection.

It will fire multiple times if there are multiple cells with bonusfood or normalfood.
HomeWork

• Read tutorial 3, we will continue next week.
• Implement your own eater with smart jump and move operators to be handed in on the 9.5.
• On the 16.5 we will run all your eaters in class – the winning eater will get a surprise 😊
Worst Preference

# Select*move*avoid-empty-eater
# If there is a proposed operator to move to an empty cell or a cell with
# another eater, then avoid that operator.
worst preference is written as a less than sign: “<”.

```
sp {select*move*avoid-empty-eater
  (state <s> ^operator <o> +)
  (<o> ^name move
    ^content << empty eater >>)
  -->
  (<s> ^operator <o> <)}
```

Tutorial 2 page 38, summary of preferences
Smart Agent – Avoid Moving Back

• The eater should be to a cell different from the one it just came from.
• The eater must remember which direction it moved to get to the current cell.
• What is needed is a persistent augmentation of the state that records the direction of the last operator.
• Need to determine what is an opposite direction.
Opposite Directions

• Create a structure in working memory that contains each direction, and each direction has an attribute that is its opposite direction.

• Rules such as initialize*state*directions are quite common in Soar program because they create fixed working memory structures.

```plaintext
sp {initialize*state*directions
  (state <ss> ^type state)
  -->
  (<ss> ^directions <n> <e> <s> <w>)
  (<n> ^value north ^opposite south)
  (<e> ^value east ^opposite west)
  (<s> ^value south ^opposite north)
  (<w> ^value west ^opposite east})
```

• This rule will fire during the first cycle when (s1 ^type state) is added to working memory.
• The structures it creates will have i-support because the rule conditions do not test an operator.
• However, the structures will never be removed because the conditions match throughout the life of the eater.
Maintain Memory Of the Last Direction Moved

```
# Apply*move*create*last-direction
# If the move operator for a direction is selected,
# create an augmentation called last-direction with that direction.

sp {apply*move*create*last-direction
  (state <s> ^operator <o>)
  (<o> ^name move
    ^direction <direction>)
  -->
  (<s> ^last-direction <direction>))

# Apply*move*remove*last-direction
# If the move operator for a direction is selected,
# and the last-direction is not equal to that direction,
# then remove the last-direction.

sp {apply*move*remove*last-direction
  (state <s> ^operator <o>
    ^last-direction <direction>)
  (<o> ^direction <> <direction>
    ^name move)
  -->
  (<s> ^last-direction <direction> -))
```
Proposal

• Modify the proposal and selection rules so that the operator for moving back is never proposed

• A condition can easily be added that tests that the direction of the adjacent cell (<d>) is not equal to the opposite of the direction of the last move.

• Problem ?
Proposal

• Modify the proposal and selection rules so that the operator for moving back is never proposed
• A condition can easily be added that tests that the direction of the adjacent cell (<d>) is not equal to the opposite of the direction of the last move.
• Problem?
• This will not work for the first move when there is no direction for the last move because no operator has applied to create the memory of a previous move.
Proposal

• First test that there does not exist an opposite of last direction that is equal to the direction of the adjacent cell.

# Propose*move*no-backward:
# If there is normalfood, bonusfood, eater, or empty in an adjacent cell, and there is no last direction equal to the opposite direction for that cell, propose move in the direction of that cell, with the cell’s content, and indicate that this operator can be selected randomly.
Proposal

- means that this rule will match only if no working memory elements match this attribute value.

<dir> matches the direction of the adjacent cell that is not a wall.

&o-dir> matches the opposite of direction of the adjacent cell that is not a wall.
HomeWork

• Read tutorial 3 implement Jump on your own.
• Implement your own eater with smart jump and move operators to be handed in on the 9.5 .
• On the 16.5 we will run all your eaters in class (on random map) – the winning eater will get a surprise 😊