

**The Engineer's Guide to Soar
Course 01: Soar Essentials**

Project 06: Substates

By Dr. Bryan Stearns, 2024



Problem

Our agent needs to sort suppliers with more complex reasoning.

- It will eventually need to account for:
 - Counts of required supplies satisfied by each supplier
 - Supplier attribute scores in descending order of configured attribute weights
- This can't easily all be done with a single decision cycle.

Solution

Use a *substate* to evaluate tied suppliers.

- A substate will allow us to use any number of decision cycles to derive the desired preferences.
- Substate processing will effectively "pause" the top-level (`select-supplier`) decision-making until the agent finds the solution that lets it select one (`select-supplier`) operator.

Project Goal

For this project, agent output should be the same as for Project 05,

- The agent should select suppliers in order of their total-score.

But the agent should use a *substate* to create its preferences.

- Since our output is the same as before, the preference logic will also be the same.

Our goal here is simply to reimplement our code using a substate.

- This will let us use more complex logic in future projects!

Lesson 06 – Outline

This lesson explains the following new concepts:

1. Hierarchical problem solving in Soar
2. Substates
3. Naming states
 - Using the "`^superstate`" WME
 - Using the "`run --elaboration`" command
4. Solving impasses in substate processing
 - Accessing tied operators from a substate
 - Returning results from a substate
 - Ensuring result persistence

Hierarchical Problem Solving

Using Impasses for Task Decomposition

Recall: Impasses

Recall from Lesson 03: An *impasse* occurs any time the agent is not able to continue its decision cycle.

```
INPUT WEIGHT: sustainability: 11
INPUT WEIGHT: total-cost: 11.010000
1: ==>S: S2 (operator tie)
```

This indicates a bug in our code **only if** we don't want the agent to engage in a deeper level of problem solving.

Whenever a Soar agent hits an impasse, Soar treats it as a **subproblem**

- It must be solved before the original decision making can continue.
 - The agent nests its subproblem solving in a new layer of decision making
- Impasses thus let Soar automatically organize *hierarchical problem solving*

```
1: Prob A: Step 1
2: Prob A: Step 2
3: Prob A: Step 3
   (impasse)
4: └─ Prob B: Step 1
5: └─ Prob B: Step 2
   (impasse solved)
6: Prob A: Step 4
```

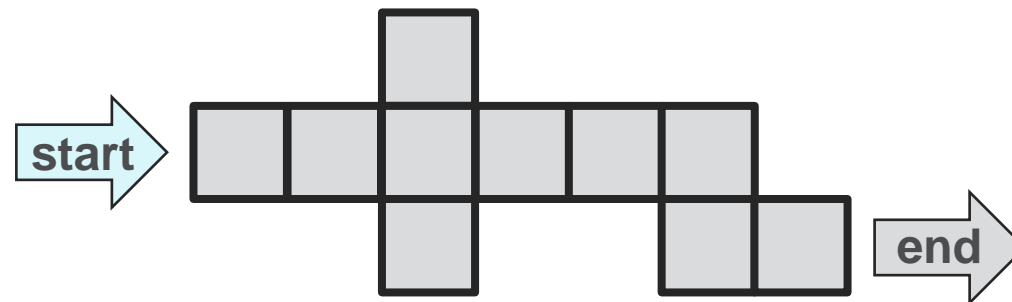
Maze Example – Hierarchical Problem Solving

What follows is an example of how a Soar developer can look at a problem hierarchically.

- (At a high level without getting into code yet)
 - (We'll look at what impasses and subproblems mean for code later on.)
-

Consider a Soar agent in a simple maze domain

- The goal is to move through the maze from the start to the end.



Maze Example – Main Problem

The main problem action is to move through the maze to the end.

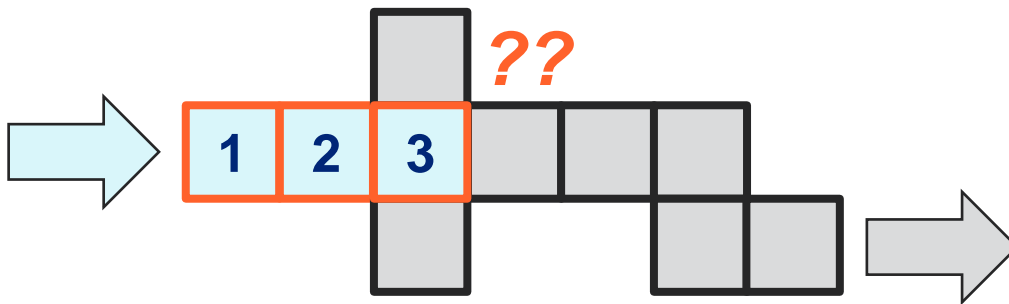
- Operators: Propose each currently possible move forward

The agent begins decisions like this:

- Step 1: Move to space 1. (the only choice)
- Step 2: Move to space 2. (the only choice)
- Step 3: Move to space 3. (the only choice)

But then there are 3 possible moves forward! (north, east, south)

- The agent does not know which operator to select.
- This leads to a *Tie Impasse!*



Decision Making

Problem: Navigate Maze



**The impasse entails a subproblem:
*Which path should the agent take?***

Maze Example – Subproblem 1

The subproblem action is to evaluate possible paths forward.

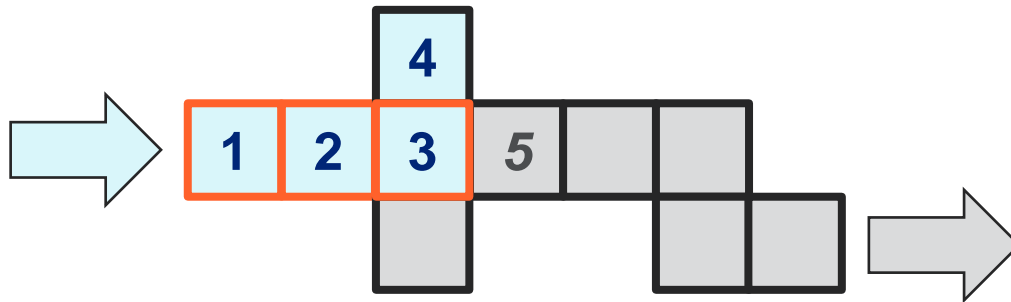
- Operators: Propose to evaluate each possible next path
 - Prefer them in clockwise order (arbitrarily)

Decisions proceed as follows:

- Step 4: Evaluate the north path. (dead end)
- Step 5: Evaluate the east path...

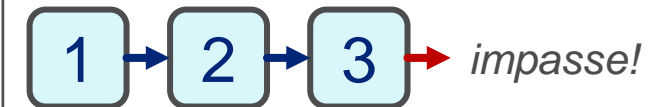
But let's say the agent can't immediately tell where the east path leads.

- So the operator cannot be applied.
- This leads to another impasse! (*operator no-change*)



Decision Making

Problem: Navigate Maze



Subproblem: Evaluate Paths



**The impasse entails a new subproblem:
*Where does the east path lead?***

Maze Example – Subproblem 2

The subproblem action is to search down the east path:

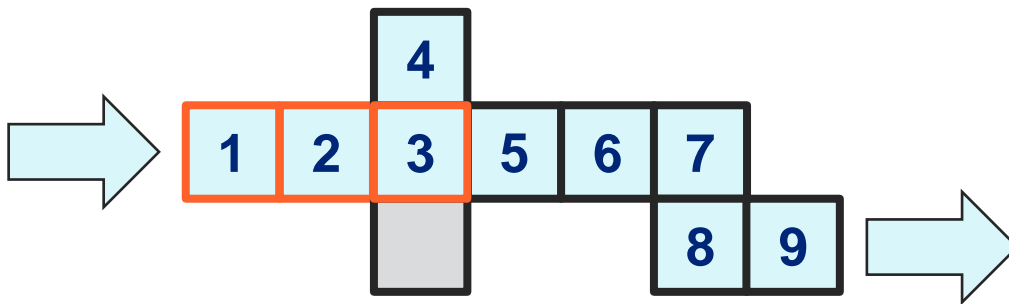
- Operators: Propose to visually search down the path step by step

Decisions proceed as follows:

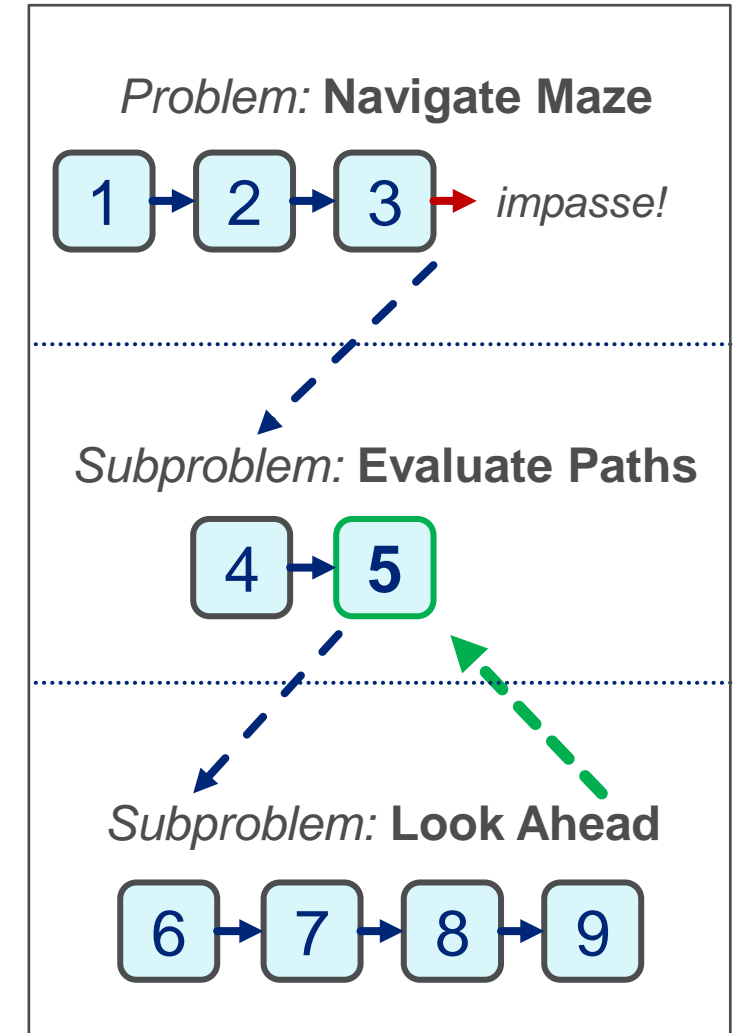
- Steps 6...8: Scan spaces 6-8
- Step 9: Scan space 9; see that it leads to the exit.

Step 9 provides the data needed to apply the operator from Step 5.

- This resolves the previous impasse!



Decision Making



Maze Example – Subproblem 1 Cont.

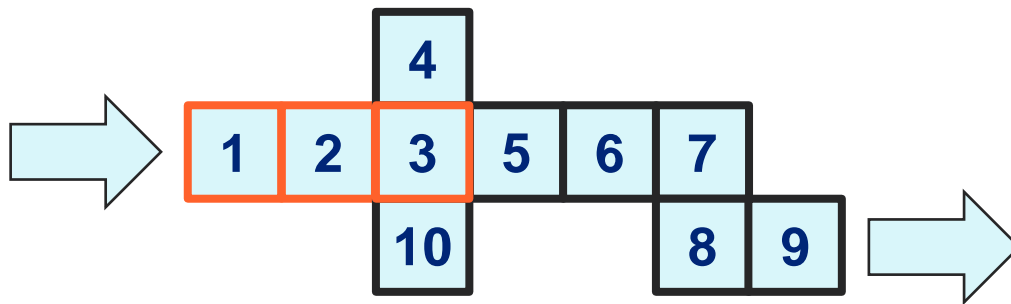
With information about the east path, the agent completes Step 5.

Decisions proceed as follows:

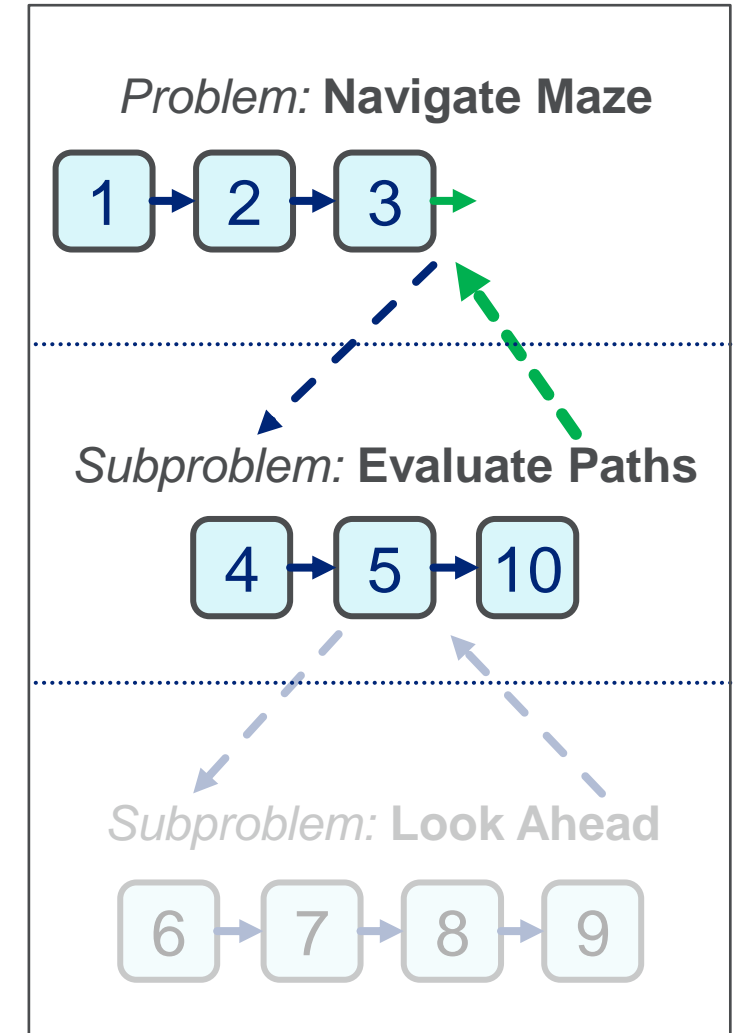
- Step 5: Evaluate the east path. (exit in 6 moves)
- Step 10: Evaluate the south path. (dead end)

After Step 10, the agent has evaluated all paths forward

- It can then prefer the quickest path to the exit
- This resolves the remaining impasse!



Decision Making

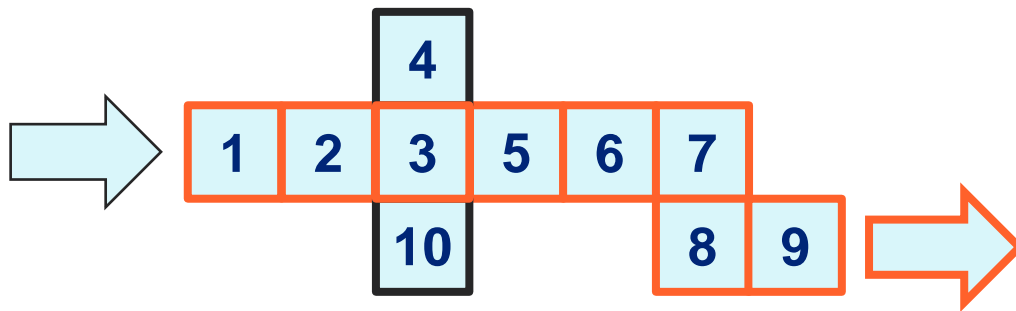


Maze Example – Main Problem Cont.

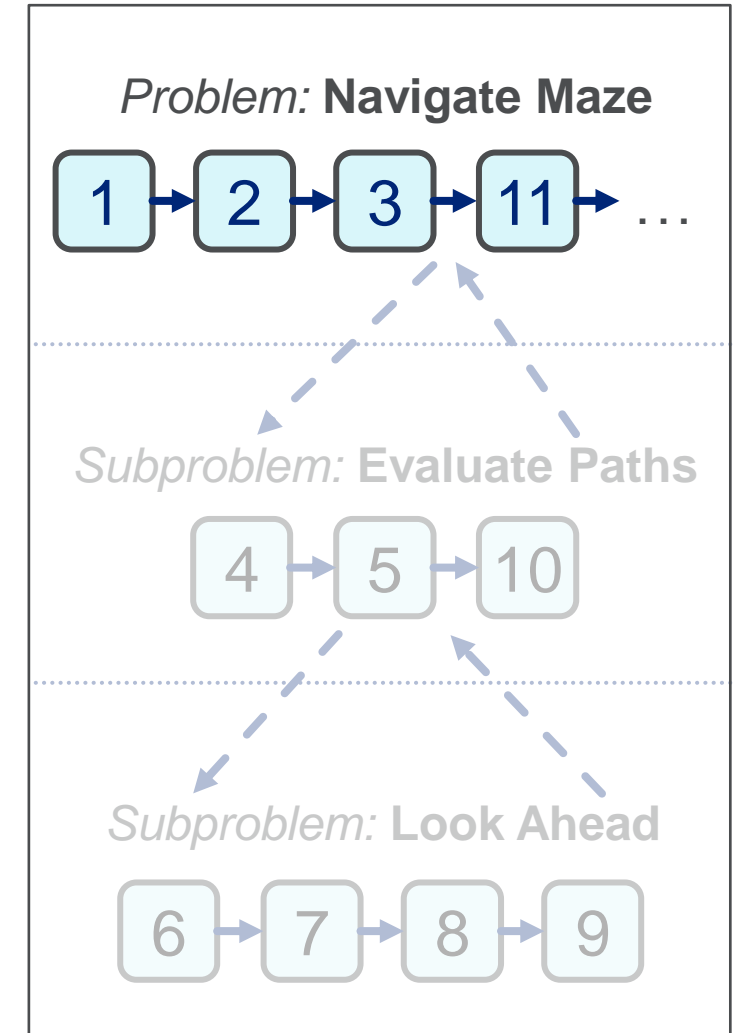
The agent proceeds solving the main problem from where it left off:

- Step 11: Move to space 5
- Step 12: Move to space 6
- ...
- Step 15: Move to space 9
- Step 16: Exit the maze

And the agent has solved the maze problem!



Decision Making



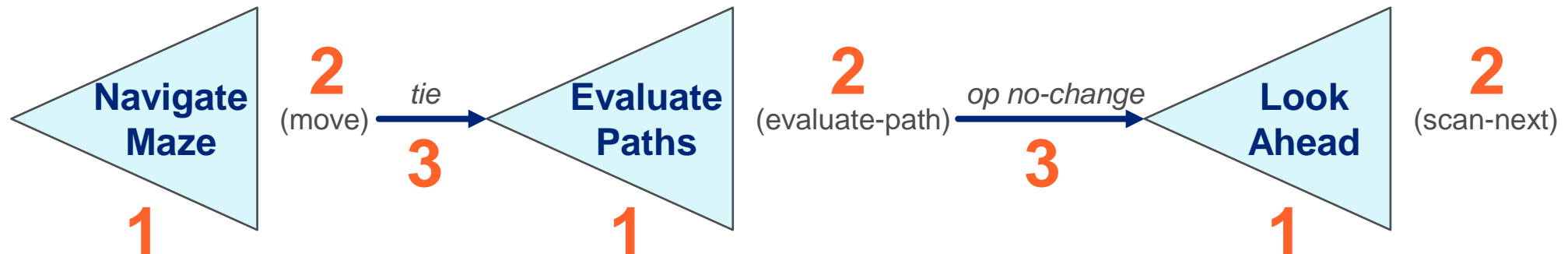
Soar Problem Space Diagramming

- 1 Problem
- 2 Operator(s) used across the problem
- 3 Impasse that leads to a subproblem

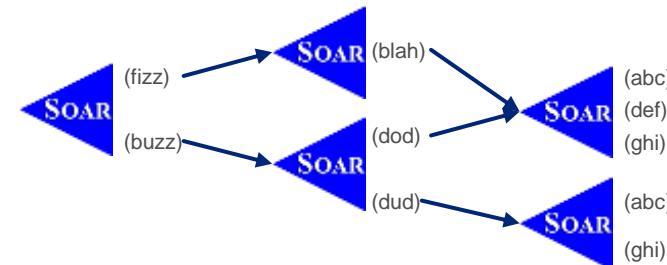
Legend

Hierarchical task composition helps us understand what we want a Soar agent to do.

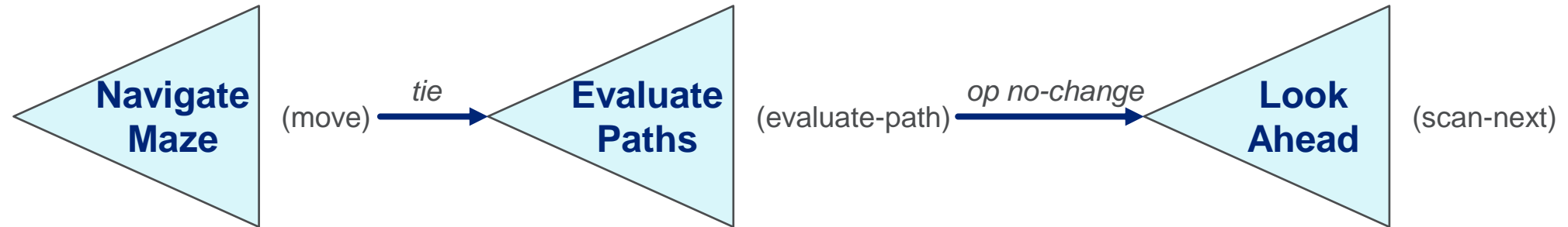
- We describe the task in terms of ***problems and subproblems***.
- We can diagram the maze task operators to help plan (as shown below).



- A diagram could show any number of subproblems per problem, as needed.



Soar States



But how do we translate the hierarchy of ***problems and subproblems*** into Soar code?

- We encode agent behavior in terms of ***states and substates***!

Substates

And How To Use Them

What is a State?

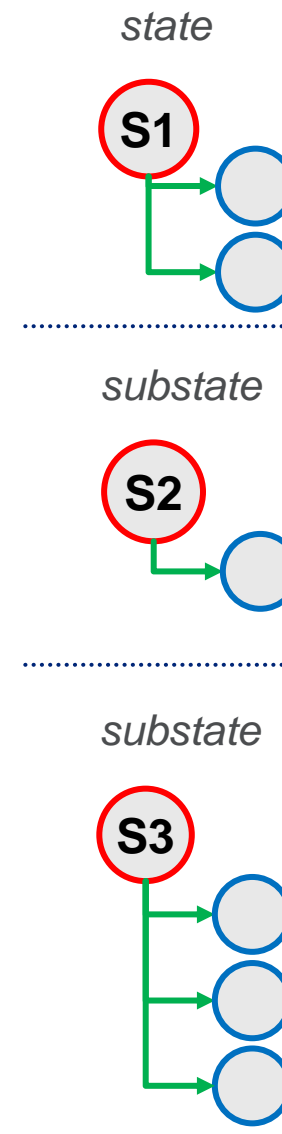
A Soar agent is called “stateful” because it can hold memory for its current state of problem solving.

- We learned in Lesson 01 that the “state” ID is the root of this memory graph structure

Soar keeps WM states for *each active layer of problem solving* at once

- Each subproblem gets its own partition of the WM graph
- Each partition has *its own root state ID*
- The agent works with that partition when solving that subproblem

Working Memory



Decision Making

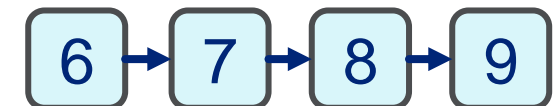
Problem: Navigate Maze



Subproblem: Evaluate Paths



Subproblem: Look Ahead



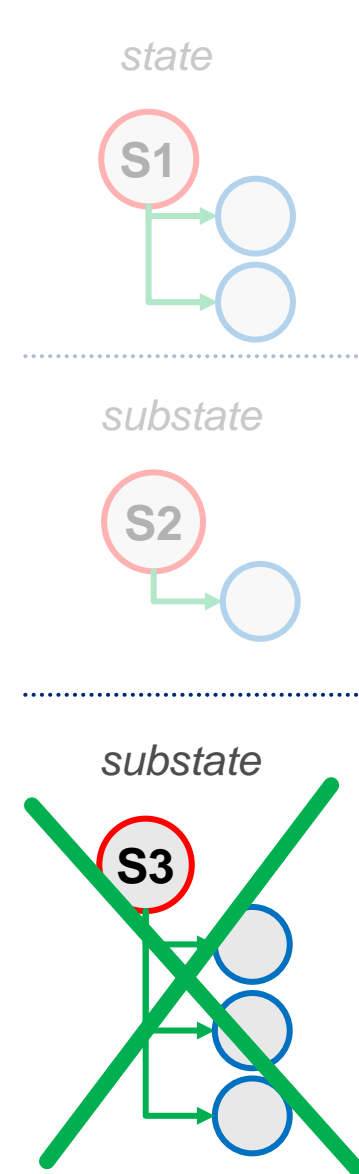
State Persistence

Soar tracks states in a *stack*

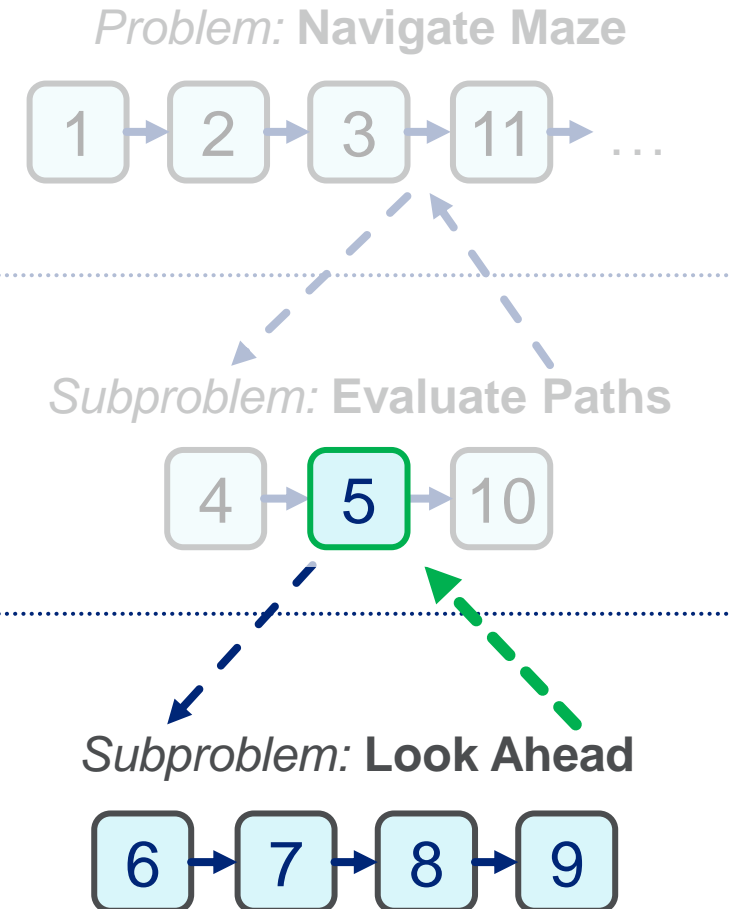
- When an impasse arises, Soar pushes a substate onto the stack
- You cannot have two substates of a single parent state at the same time.
- When an impasse is solved, Soar pops the substate off the stack
- All WMEs for the corresponding WM partition are **immediately removed** from WM

When processing returns to Step 5 from Step 9, Soar deletes S3.

Working Memory



Decision Making



State Terminology

“state”: Can (unfortunately) be used in multiple ways:

1. A single specific state ID in WM
2. A point in the theoretical space of possible steps toward solving a problem
3. The set of WMEs in a problem’s WM partition, which together represent the agent’s current point in the space of problem solving

“topstate”: The initial / default WM state

- The Soar CLI lets you use the keyword <ts> to quickly reference the topstate.

“substate”: Any WM state that is not the topstate.

- The Soar CLI lets you use the keyword <s> to quickly reference the latest substate ID.

“superstate”: The immediate parent state to a substate

- The Soar CLI lets you use the keyword <ss> to quickly reference the superstate of <s>.
- Similarly, use <sss> to quickly reference the superstate of <ss>.

**The topstate (<ts>)
(also <sss> in this case)**

```
==>S: S1
==>S: S2 (state no-change)
==>S: S3 (state no-change)
```

**S3’s superstate
(<ss>)**

**The latest
substate (<s>)**

Substate Working Memory

Soar initializes each substate in WM with WMEs like those provided to the topstate (S1).

- But only the topstate gets the `^io` WME.

```
(S3 ^epmem E3 ^reward-link R7 ^smem L3 ^superstate S2 ^type state
  ^attribute state ^impasse no-change ^choices none ^quiescence t)
(S2 ^epmem E2 ^reward-link R4 ^smem L2 ^superstate S1 ^type state
  ^attribute state ^impasse no-change ^choices none ^quiescence t)
(S1 ^epmem E1 ^io I1 ^reward-link R1 ^smem L1 ^superstate nil ^type state)
```

Soar also provides the `^superstate` WME, which links a state ID to its superstate ID.

- This lets rules for the subproblem access WM context from the parent problem

Soar also provides other WMEs that describe the impasse that led to the substate

- For instance, (`^attribute |state|`) and (`^impasse |no-change|`) tell us that the parent state processing reached a “*state no-change*” impasse
- (*See Soar Manual page 85 for more details about these WMEs and their meanings.*)

Rule Matching Across States

In a rule, <s>/<ss>/<ts> are *not* keywords like in the CLI.

But it is still good practice to only use those variable names to refer to the state/superstate/topstate.

All our rules begin with “state <s> ...”

- The state token indicates that <s> is a state ID.

When there are multiple states in WM, <s> could match *any one of them!*

```
sp {elaborate*fizz-buzz*all
    (state <s> ^type state)
  -->
  (<s> ^fizz buzz)}
```

<s> matches S1, S2, and S3

```
sp {elaborate*fizz-buzz*topstate
    (state <s> ^superstate nil)
  -->
  (<s> ^fizz buzz)}
```

<s> matches only S1

```
(S3 ^epmem E3 ^reward-link R7 ^smem L3 ^superstate S2 ^type state
  ^attribute state ^impassé no-change ^choices none ^quiescence t)
(S2 ^epmem E2 ^reward-link R4 ^smem L2 ^superstate S1 ^type state
  ^attribute state ^impassé no-change ^choices none ^quiescence t)
(S1 ^epmem E1 ^io I1 ^reward-link R1 ^smem L1 ^superstate nil ^type state)
```

Naming States

Why and How

Using Substates for Supplier Sort

The order in which the agent selects (`select-supplier`) operators determines how it sorts the suppliers.

When preference logic is simple, we can use a handful of preference rules to determine selection order.

- As we did in the previous project

But when preferences require step-by-step logic, we need a substate.

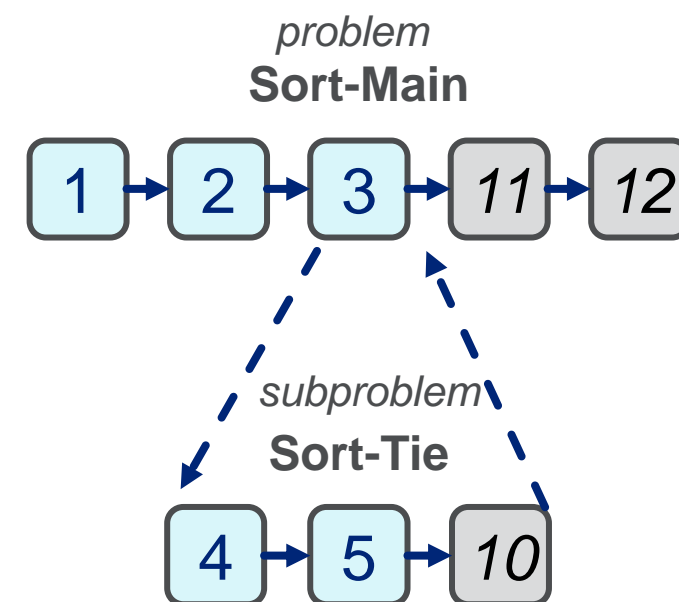
Eventually, we will need to support multiple kinds of step-by-step preference logic such as:

1. Iterating through different configuration weights
2. Summing and comparing sub-scores for particular input attributes
3. Sorting by some attributes only after first sorting by others

The (select-supplier) Subproblem

1. The subproblem begins any time the agent has a tie among 2 or more (select-supplier) operators.
 - This tie is an impasse in the decision cycle
 - Soar creates a substate for this impasse
2. The agent then does any number of processing steps in the substate to derive preferences among the tied operators.
3. The agent modifies the parent state's WM by attaching its derived preferences to the operators there
4. Once the agent has enough preferences to let it pick a single (select-supplier) operator, the subproblem is solved and it can proceed as before.

Decision Making



**Modifying the parent state is called
“returning a result” to that state.**

What's in a Name?

The first thing you want to do when using a substate is give each state a `^name` WME.

- This lets you easily control which state(s) your rules fire in.

```
sp {propose*for*nav-maze
  (state <s> ^name nav-maze
    ^open-dir <dir>)
  -->
  (<s> ^operator <o> +)
  (<o> ^name move
    ^direction <dir>))}
```

<s> matches S1 only

```
sp {propose*for*eval-paths
  (state <s> ^name eval-paths
    ^open-dir <dir>)
  -->
  (<s> ^operator <o> +)
  (<o> ^name evaluate-path
    ^path-dir <dir>))}
```

<s> matches S2 only

```
(S2 ^open-dir |north| ^open-dir |east| ^open-dir |south|
  ^name eval-paths ^superstate S1 ...)
(S1 ^open-dir |north| ^open-dir |east| ^open-dir |south|
  ^name nav-maze ^superstate nil ...)
```


State Names for Supplier Sort

For now, we want two states, each with their own name

- S1: Name = “**suppliersort-main**”
 - Proposes the (select-supplier) operator for each input supplier
- S2: Name = “**suppliersort-tie**”
 - Responds to a *tie impasse* in S1
 - Creates preferences for S1’s tied operators

One doesn’t *have* to always make a ^name WME for substates in Soar, but we will, because it is definitely best practice!

Let's Write Some Code!

1. Open your `agent_starter.soar` file and look at the first 2 rules.
 - They are propose and apply rules for the `(init)` operator.
 - We want to modify our `(init)` operator in two ways:
 - It should only be proposed in the topstate.
 - It should create `^name suppliersort-main` on the topstate.
2. Modify the rules accordingly, as instructed in comments.
3. Run the agent for 2 steps to test it.
 - After 2 steps, enter the command “`print S1`”. You should see the following:

```
step 2
1: 0: 01 (init)
2: ==>S: S2 (operator tie)
```

```
p S1
(S1 ^epmem E1 ^io I1 ^name suppliersort-main ^operator 06 + ^operator 02 +
  ^operator 08 + ^operator 07 + ^operator 04 + ^operator 05 +
  ^operator 03 + ^reward-link R1 ^smem L1 ^superstate nil
  ^supplier-list C9 ^type state)
```

4. Proceed to the next slide when you are done.

Let's Write Some More Code!

1. Open your `agent_starter.soar` file and uncomment the 3rd rule.
 - We'll use this elaboration rule to name the substate
 - Its LHS should detect that `suppliersort-main` had a *tie impasse*, and it should **only** fire for a substate that results from that *tie impasse*.
 - Its RHS names the resulting substate as `suppliersort-tie`.
2. Modify the rule so that it matches the following:

```
sp {elaborate*suppliersort-tie*name
  (state <s> ^superstate.name suppliersort-main
    ^impasse tie)
-->
(write |** (Figuring out how to sort the suppliers...)| (crLf))
(<s> ^name suppliersort-tie)}
```

Tests that the immediate superstate to <s> is the state we named "suppliersort-main".

Tests that the current substate <s> was created due to a tie impasse.

Let's Write Some More Code!

3. After running the agent for 2 steps, enter the command “print <s>”. You should see the following:

```
p <s>
(S2 ^attribute operator ^choices multiple ^epmem E2 ^impasse tie ^item 07
  ^item 06 ^item 04 ^item 03 ^item 02 ^item 05 ^item-count 6
  ^non-numeric 06 ^non-numeric 07 ^non-numeric 04 ^non-numeric 03
  ^non-numeric 02 ^non-numeric 05 ^non-numeric-count 6 ^quiescence t
  ^reward-link R4 ^smem L2 ^superstate S1 ^type state)
```


- The conditions are there, but where is the state name??
- No worries! Our rule hasn't fired yet after 2 steps. Soar only just finished making the substate.
 - Our tie impasse occurred at the end of a *Decision Phase*
 - Our rule won't fire until the first elaboration cycle of the *Apply Phase*



Let's Write Some More Code!

4. Enter the command “matches” to see what rules match and are now *about* to fire. You should see the following:

```
matches
0 Assertions:
I Assertions:
  elaborate*suppliersort-tie*name [S2]
```



- If there are no bugs, your rule name should appear
 - It shows which state the rule will fire in: [S2]
5. To run for only one rule-firing cycle (an elaboration cycle) without proceeding for a full decision cycle, enter the command “run --elaboration” (or just the command “e” for short).
- Your rule should then fire, so that you should see the following:

```
run --elaboration
** (Figuring out how to sort the suppliers...)
```

6. Proceed to the next slide when you are done.

Solving Impasses

In Substate Processing

Using Our Substate

We have a named substate for solving the *tie impasse*.

- We can now use it to propose and apply operators that solve the tie.

We need two more rules, which will fire in our substate to do the following:

1. **Propose** an operator to break the tie

- Propose if the state is named “suppliersort-tie”

2. **Apply** the operator to break the tie

- Create preferences among all tied items based on their total-score.

Let's Write Some Code!

1. In your `agent_starter.soar` file, find the commented instructions that are underneath the “`## OPERATOR: break-attr-tie`” comment header
2. Write a proposal rule here named **`propose*suppliersort-tie*break-attr-tie`** that does the following:
 - On its LHS: Tests that the state is named “suppliersort-tie”.
 - On its RHS: Proposes an operator named “break-attr-tie”.
 - Give it only *acceptable* (+) preference.
 - You can use the `(init)` proposal rule as a reference for overall rule structure.
3. Proceed to the next slide when you are ready to test your rule.

Let's Write Some Code!

4. Run your agent for 3 steps. You should see the following:

```
step 3
  1: 0: 01 (init)
  2: ==>S: S2 (operator tie)
** (Figuring out how to sort the suppliers...)
  3:      0: 09 (break-attr-tie)
```

5. If so, proceed to the next slide!
 - If you are stuck, compare with the `instructor_solution.soar` code

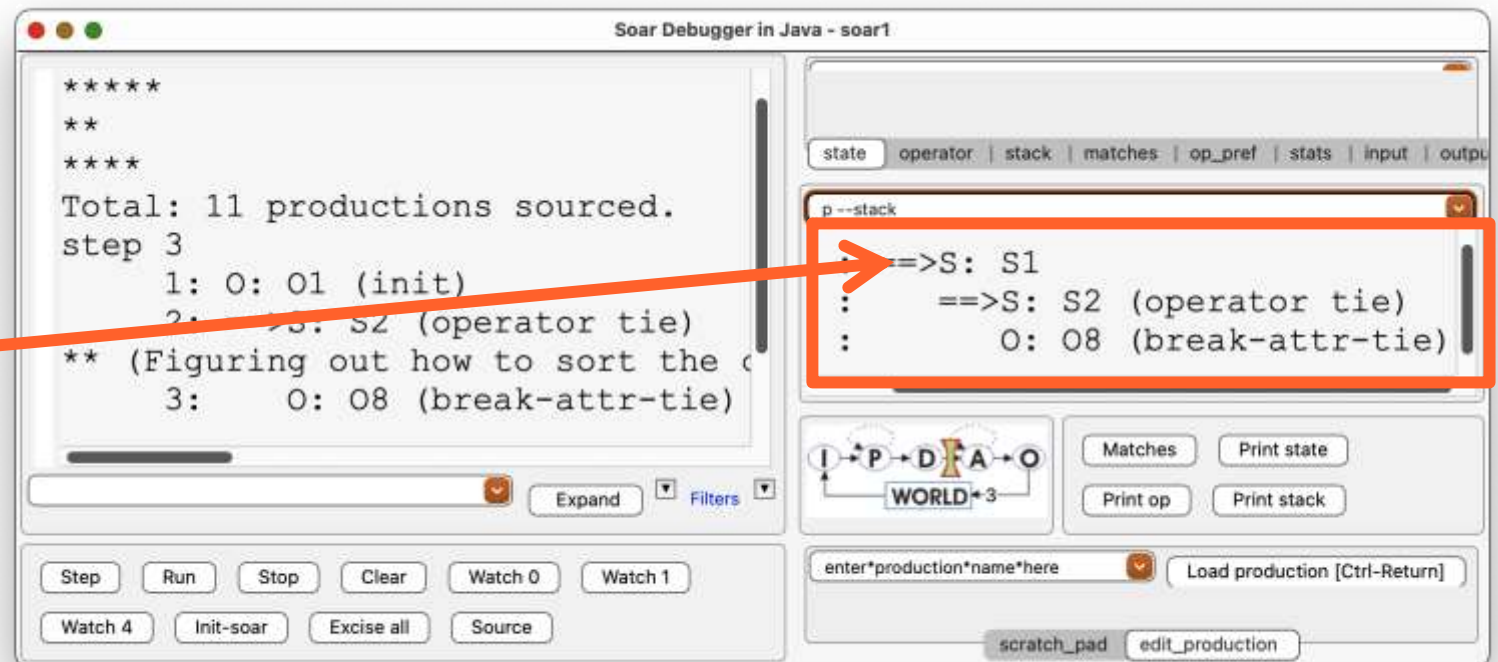
Inspecting the State Stack

You now have an operator selected in a substate (S2) after having used an operator in the topstate (S1). Well done!

At any time, you can use the command “print --stack” to see the stack of states and the operators currently selected in each.

- This command is automatically entered into a debugger pane for easy reference:

Notice there is no operator currently selected in S1. (init) is complete. The tie is for the agent's attempt to select the next operator for S1 after (init).



How To Solve the Impasse?

How do we use our substate operator to solve the impasse?

The solution involves multiple new concepts:

1. Referencing WMEs from the superstate that define the problem we need to solve
2. Returning the solution from the substate's WM to the superstate's WM
3. Ensuring that the solution results persist in the superstate once the substate is done

1. Referencing Tied Operators

For a tie impasse, Soar provides the substate with `^item` WMEs that reference the tied operators.

- Each `^item` edge points to a single `^operator` ID that is tied in the superstate.

S2:

```
p <s>
(S2 ^attribute operator ^choices multiple ^epmem E2 ^impasse tie ^item 07
  ^item 06 ^item 05 ^item 04 ^item 02 ^item 03 ^item-count 6
  ^name suppliersort-tie ^non-numeric 05 ^non-numeric 07 ^non-numeric 04
  ^non-numeric 03 ^non-numeric 02 ^non-numeric 06 ^non-numeric-count 6
  ^operator 09 ^operator 09 + ^quiescence t ^reward-link R4 ^smem L2
  ^superstate S1 ^type state)
```

These are the same IDs
that exist on the topstate!

S1:

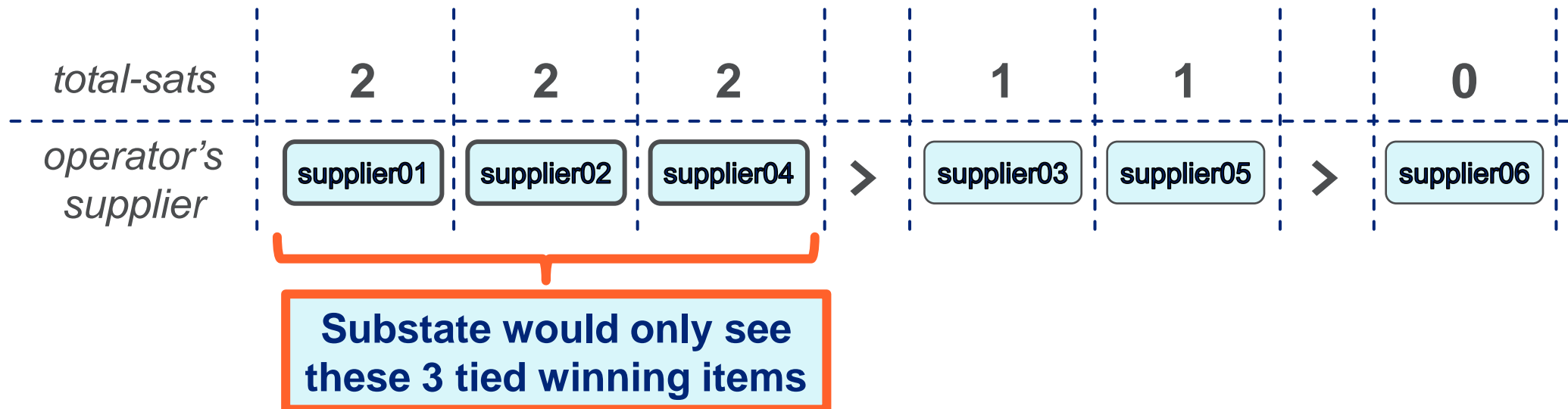
```
p <ts>
(S1 ^epmem E1 ^io I1 ^name suppliersort-main ^operator 06 + ^operator 02 +
  ^operator 08 + ^operator 07 + ^operator 04 + ^operator 05 +
  ^operator 03 + ^reward-link R1 ^smem L1 ^superstate nil
  ^supplier-list C9 ^type state)
```

Referencing *Only* Tied Operators

Any proposed superstate operators that were *not* tied will not be included in this set

This is very useful when the agent has *some* preferences, just not enough to avoid a tie

- Example: *If* the superstate had a preference rule to sort suppliers by $\wedge\text{total-sats}$, but 3 operators still tied for the most $\wedge\text{total-sats}$, then the substate would only see $\wedge\text{item}$ WMEs for those 3 operators.



Let's Write Some Code!

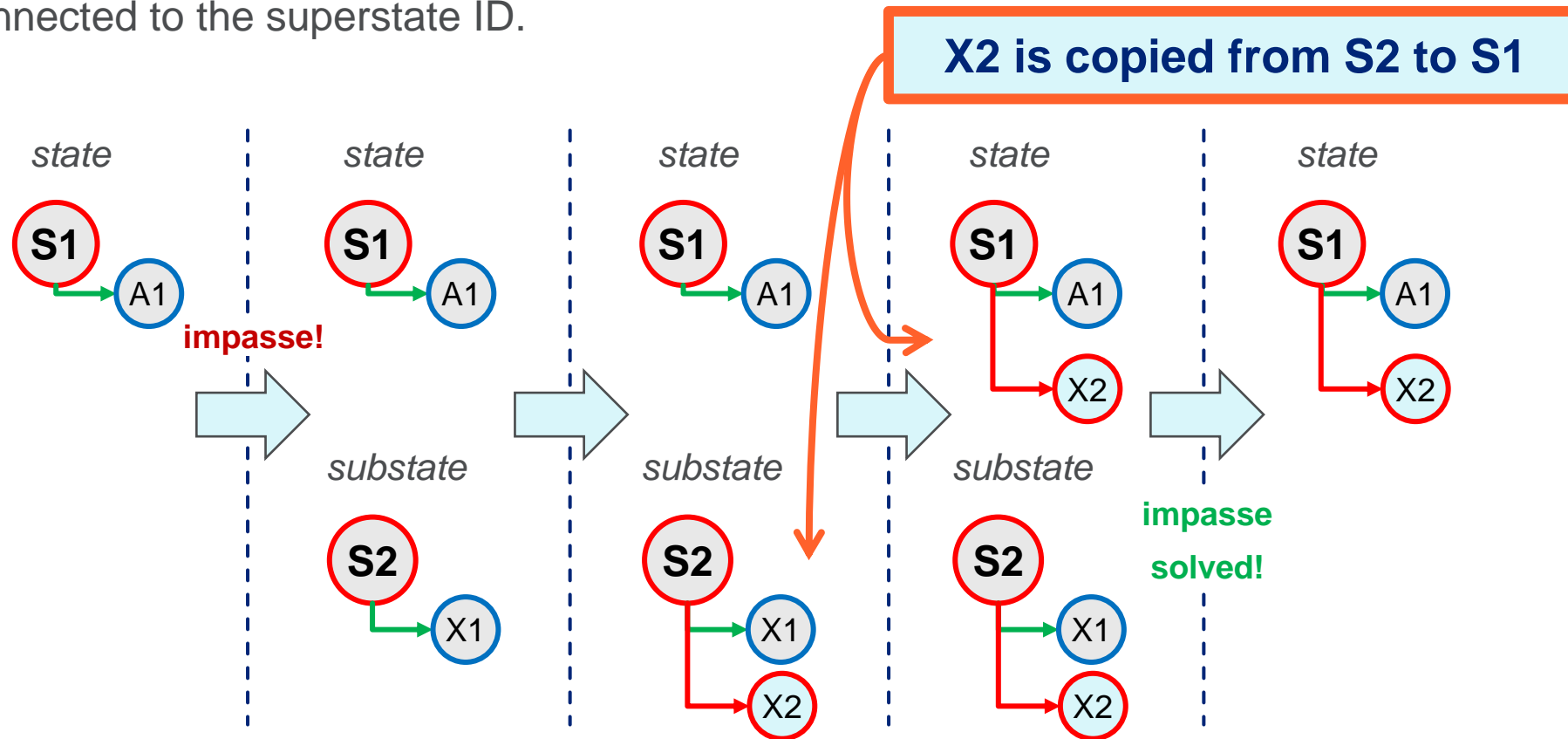
1. In your `agent_starter.soar` file, uncomment the final rule
 - Named **`apply*suppliersort-tie*break-attr-tie*prefer-total-score`**
2. Read the provided rule code to be sure you understand how the rule works.
 - It is conceptually the same as the preference rule we created in Project 03.
 - `prefer*suppliersort-main*select-supplier*total-score`
 - But now the logic is performed from a substate.
3. Follow the commented instructions to do the following:
 - On the LHS: Get a variable reference to the `suppliersort-tie` state's superstate.
 - Call it `<ss>`.
 - On the LHS: Get variable references to two tied operators.
 - Call them `<o1>` & `<o2>`.
 - *Note: If the operator name is unique, apply rules don't need to test the state name.*
4. Proceed to the next slide to learn some more details we'll need for the RHS.

2. Returning the Solution From the Substate

How do we solve a superstate's impasse from a substate?

- By modifying the superstate's WM

A rule modifies its state's superstate WM by adding/removing WMEs from any structure connected to the superstate ID.



2. Returning the Solution From the Substate

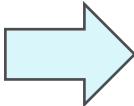
How do we solve a superstate's impasse from a substate?

- By modifying the superstate's WM

A rule modifies its state's superstate WM by adding/removing WMEs from any structure connected to the superstate ID.

```
sp {elaborate*fizz-buzz*returned
    (state <s> ^superstate <ss>
      ^drink-fizzy false)
  -->
  (<ss> ^fizz nil)}
```

**In this example,
^fizz nil is
created directly
on S1 from a rule
that fires in S2.**

<pre>(S2 ^superstate S1 ^drink-fizzy false) (S1 ^superstate nil)</pre>		<pre>(S2 ^superstate S1 ^drink-fizzy false) (S1 ^superstate nil ^fizz nil)</pre>
--	--	--

Remember: Each time the superstate is modified, this modification is called “returning a result” from the substate to the superstate

Let's Write Some Code!

1. Open your rule, `apply*suppliersort-tie*break-attr-tie*prefer-total-score`
2. Follow the commented instructions to fill out the RHS to do the following:
 - Modify the superstate `<ss>` (“return a result”) by creating a preference between the operator IDs there.
 - Mark item `<o1>` as better than item `<o2>`.
 - Note: You don't need to test for `(<ss> ^operator <o1>)` or `(... <o2>)`, because `(<s> ^item <o1> <o2>)` gives us the reference to the same IDs.
3. Proceed to the next slide when you are ready to test your rule.

Final Results

Hit “run”, and you should see the following:

- The (break-attr-tie) operator completes
- Our apply rule fires for each needed preference
- Topstate processing is then able to proceed to completion without further impasses, using all these results together
 - (Using rules imported from previous projects.)

If so, **congratulations!**

You have completed the agent for Project 06!

- But don't close the lesson yet – there is a little bit more to explore...
- Proceed to the next slide when you are ready.

```
run
  1: 0: 01 (init)
  2: ==>S: S2 (operator tie)
  ** (Figuring out how to sort the suppliers...)
  3: 0: 09 (break-attr-tie)
Total-score: Prefer supplier03 > supplier01
Total-score: Prefer supplier05 > supplier01
Total-score: Prefer supplier01 > supplier02
Total-score: Prefer supplier03 > supplier02
Total-score: Prefer supplier04 > supplier02
Total-score: Prefer supplier05 > supplier02
Total-score: Prefer supplier06 > supplier02
Total-score: Prefer supplier05 > supplier03
Total-score: Prefer supplier01 > supplier04
Total-score: Prefer supplier03 > supplier04
Total-score: Prefer supplier05 > supplier04
Total-score: Prefer supplier01 > supplier06
Total-score: Prefer supplier03 > supplier06
Total-score: Prefer supplier04 > supplier06
Total-score: Prefer supplier05 > supplier06
  4: 0: 06 (select-supplier)
  ** OUTPUT: First try supplier05
  5: 0: 04 (select-supplier)
  ** OUTPUT: Then try supplier03
  6: 0: 02 (select-supplier)
  ** OUTPUT: Then try supplier01
  7: 0: 05 (select-supplier)
  ** OUTPUT: Then try supplier04
  8: 0: 07 (select-supplier)
  ** OUTPUT: Then try supplier06
  9: 0: 03 (select-supplier)
  ** OUTPUT: Then try supplier02
 10: 0: 08 (output-supplier-list)
*** DONE ***
Interrupt received.
```

3. Result Persistence

Recall from Lesson 03 that the actions of non-apply rules (“i-supported” rules) only persist for as long as the rule instantiation that created them still matches.

- As soon as the instantiation no longer matches, the results go away.

Once our substate goes away, we see that the preferences that it created *remain*.

Is this because we used an apply rule to create them from the substate?

- Test the theory: On your own, replace your (break-attr-tie) rules with a single preference rule that fires from the substate.
 - All you need to do is copy the apply rule and replace the “^operator.name break-attr-tie” test with “^name suppliersort-tie”.
- Run the modified agent. What happens?

Substate Rule Type Doesn't Affect Result Persistence

The results *still persist* even if you don't use an apply rule to generate them!

- Why??

Result persistence is not from the substate, but from the *superstate*

- How??

Soar treats substate results as if they were generated by a rule that fired in the superstate

- What??

Soar generates a new *temporary rule* that acts as if it fired in the superstate to create the result

- This rule is called a “**justification**”
- Soar generates a justification for every superstate WM change made by a substate
- If the justification is an apply rule, the result persists like from any apply rule
- Otherwise, the result persists only for as long as the *justification's* LHS matches

Inspecting Justifications

Run the agent for 4 steps. Then enter this command:

- “print --full --justifications” (or “p -fj” for short)

Soar will show all justifications that it currently knows, such as the following:

```
p -fj
sp {justify29
  :justification ;# not reloadable
  (state S1 ^name suppliersort-main ^operator 07 + ^operator 06 +)
  (C7 ^name supplier05 ^total-score 14)
  (O6 ^supplier C7)
  (C8 ^name supplier06 ^total-score 10)
  (O7 ^supplier C8)
  -->
  (S1 ^operator 06 > O7)
}
```

You should see one justification for each result returned from the substate.

- Soar treats these as if they were the rules that created the results.

Justification Properties

It “fires” from the superstate’s context

It references specific IDs and numbers, not variables

```
p -fj
sp {justify29
  :justification ;# not reloadable
  (state S1 ^name suppliersort-main ^operator 07 + ^operator 06 +)
  (C7 ^name supplier05 ^total-score 14)
  (06 ^supplier C7)
  (C8 ^name supplier06 ^total-score 10)
  (07 ^supplier C8)
  -->
  (S1 ^operator 06 > 07)
}
```

It creates the same result as the substate

Justification Generation

How does Soar generate each justification?

Soar automatically tracks the WMEs that each of your rules tests in their LHSs.

IF any substate rule's LHS references a *superstate* WME in its LHS,
AND that substate rule was needed to produce the result,
THEN Soar will include that superstate WME in the justification's LHS.

The justification's RHS will simply be the returned result.

Most of this LHS comes directly from the apply rule for **break-attr-tie**, which also generated the result.

```
p -fj
sp {justify29
  :justification ;# not reloadable
  (state S1 ^name suppliersort-main ^operator 07 + ^operator 06 +)
  (C7 ^name supplier05 ^total-score 14)
  (06 ^supplier C7)
  (C8 ^name supplier06 ^total-score 10)
  (07 ^supplier C8)
  -->
  (S1 ^operator 06 > 07)
}
```

Backtracing

To determine if a rule was needed for a result, Soar uses a process called *backtracing*.

- It follows the chain(s) of dependance back from the rule that generated the result to any rule(s) that tested a parent state.

For most agents, you don't have to worry about how this works, but it's useful to know.

- Examine your code to see how results depend on (S1 ^name suppliersort-main).

1. **elaborate*suppliersort-tie*name**
tested (<ss> ^name suppliersort-main) ←
and created (<s> ^name suppliersort-tie) ←

2. **propose*suppliersort-tie*break-attr-tie**
tested (<s> ^name suppliersort-tie) ←
and created (<s> ^operator.name break-attr-tie) ←

3. **apply*suppliersort-tie*break-attr-tie*prefer-total-score**
tested (<s> ^operator.name break-attr-tie) ←
and created the result →

```
j
justify29
:justification :# not reloadable
(state S1 ^name suppliersort-main ^operator 07 + ^operator 06 +)
(C7 ^name supplier05 ^total-score 14)
(O6 ^supplier C7)
(C8 ^name supplier06 ^total-score 10)
(O7 ^supplier C8)
-->
(S1 ^operator 06 > 07)
```


Well Done!

Congratulations! You have finished Lesson 06!

- On your own: Play around with different ways of testing superstate WMEs and creating results from the substate.
- Question: How might you get a justification that acted like an apply rule?
 - *(Hint: You need a different kind of impasse than a tie impasse.)*

At this point, you have learned most of the concepts you need to program Soar!

- Future lessons will expand our understanding of rule coding syntax and techniques as we flesh out our agent's ability to complete its task