Soar Workshop Tutorial

Created by Alex Turner, Eilam Morag, Aaron Mininger, Professor John Laird

Introduction

This is a guide for learning to create software agents in Soar. This is an interactive tutorial involving the use of Lego Mindstorms EV3 robots. It makes the assumption that you have completed the main Soar tutorial found here.

The goal of this document is to give you an interactive and fun Soar tutorial. It focuses entirely on the nuts and bolts of creating Soar agents. It does not focus on nor mention the history of Soar nor the theory behind Soar. If you would like to read about that, you should read either Chapter 3 of the Soar Manual or The Soar Cognitive Architecture.

This tutorial is split up into two parts, each of which has numerous subparts. The beginner tutorial has three parts and focuses on teaching you the basics of Soar in a fun environment. The advanced tutorial has four parts. It focuses on teaching you about semantic and episodic memory and how they relate to working memory. In addition, the code you will write relies on code we have provided for you. You will find this code in Common-Include/ and Common-Advanced/. This code handles a lot of trivial and complex interactions with the environment. For example, Common-Advanced/ contains the code which turns the robot in a specified direction upon reaching a junction. All of this code is well documented and can be considered an example of good-practice Soar code. You are free to read the code to understand how it works; however, we strongly caution against modifying this code before you have completed the tutorial.

Before you begin the tutorial, it is recommended that you assemble your robot according to the instructions found in “Assembling Your Lego Mindstorms Robot.” There, we walk you through how to create the robot that is used in this tutorial. You may create your own robot with equivalent functionality, but we make no guarantees that your unique robot will work as intended with our Soar code for this tutorial.

Assembling Your Lego Mindstorms Robot

TODO: write the instructions for v2 robot

Setting Up Your Lego Mindstorms Robot

Building the Lego Mindstorm Binaries from Source

Note: If you are doing the tutorial during the Soar Workshop, we provide the binaries. You do not need to compile any of these binaries yourself. Skip to “Connecting Your Robot To Your Computer.”

The source code for this project can be found at:
https://github.com/SoarGroup/Domains-LegoMindstorms
You can either clone the repo with git, or on the right-hand side of the page you can select ‘Download ZIP’.

Linux/Mac:

Under Robot-Code/ there are 3 makefiles which will build the 3 different programs. Before you build, you need to set the environment variables SOAR_HOME (See the Setting Up section).

```
make -f make_soar_client.mak
```

This will make the soar_client program which can run on an external laptop and connect remotely to the ev3_server program running on the lego robot.

```
make -f make_ev3_server.mak
```

This will make the ev3_server program which runs on the lego brick and connects to the soar_client over TCP. This has to be made with an arm cross-compiler. For linux, you can use the apt-get packages `gcc-arm-linux-gnueabi` and `g++-arm-linux-gnueabi.` Note that an already compiled version is included in the repo in the Robot-Code/native folder.

```
make -f make_ev3_standalone.mak
```

This will make the ev3_standalone program which runs on the lego brick by itself and includes a copy of soar. This also has to be made with an arm cross-compiler. Note than an already compiled version is included in the repo in the Robot-Code/native folder.

Windows

There are no automated build scripts for windows, but it is simple to get it working in Visual Studio. For the soar_client, here are the files you will need to import (don’t add all files under src because some are unix specific).

- `src/Constants.h`
- `src/comm/SoarCommunicator.h`
- `src/comm/CommStructs[.h .cpp]`
- `src/util/WMUtil[.h .cpp]`

- `src/util/CommUtil[.h .cpp]`
- `src/windows/[all files]`
- `src/soar/[all files]`

A couple of notes for creating a visual studio project:

- Add `/path/to/soar/bin/include` and `/path/to/lego/project/Robot-Code/src` to additional include directories (C/C++/General)
- Add `/path/to/soar/bin` to additional linker library directories (Linker/General)
- Add ws2_32.lib and Soar.lib to additional linker dependencies (Linker/Input)

Setting up

Add the environment variables `SOAR_HOME` and `PATH` (`LD_LIBRARY_PATH` for Linux, `DYLD_LIBRARY_PATH` for OS X) with values `.../soar/bin`, where `...` is the path to the directory where soar is stored on your computer. For example, a possible path on Windows would be `C:/Users/JohnDoe/SoarSuite_9.4.0-Windows_64bit/bin`. The variable `SOAR_HOME` is used to locate the Java Soar Debugger, and `PATH` is used to locate the Soar libraries.

Press the middle key on the robot to turn it on. Use this and the surrounding keys to navigate through the menus.

To ensure that the robot does not “go to sleep”, navigate to the settings menu on the robot (the wrench tab) and set “Sleep” to “never.”

Finally, in the same tab, set “Volume” to 0%.

Connecting Your Robot To Your Computer

Note: This section makes the assumption that you are using a WiFi dongle for connecting to the robot.

Follow these steps to connect the robot to your computer:

1. Connect your robot to WiFi
    a. Navigate to the Settings Menu (the wrench icon)

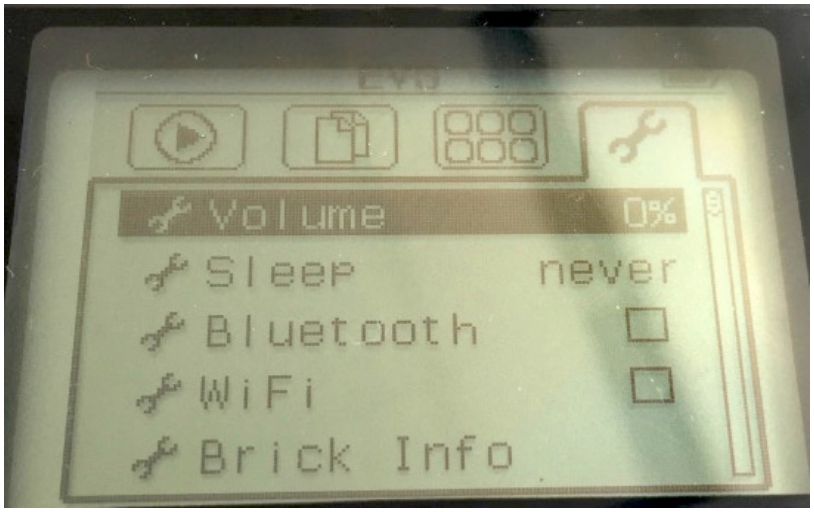

Figure 1: The Settings menu of a Lego Mindstorms EV3.

    b. Select WiFi

Figure 2: The settings menu with WiFi highlighted.

c. Enable WiFi by checking the “WiFi” box (there might be a slight loading time)

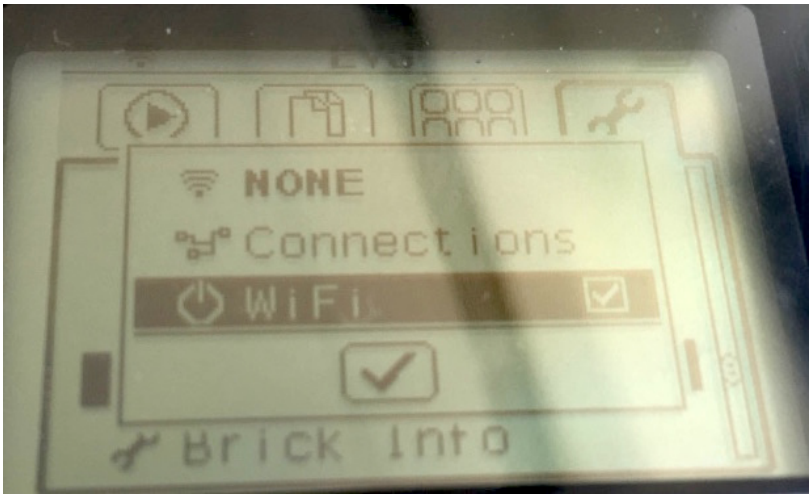

Figure 3: The WiFi menu of an EV3 robot.

d. In the same window, select “Connections” and search for your network (e.g. “Lego World 2GHz”)

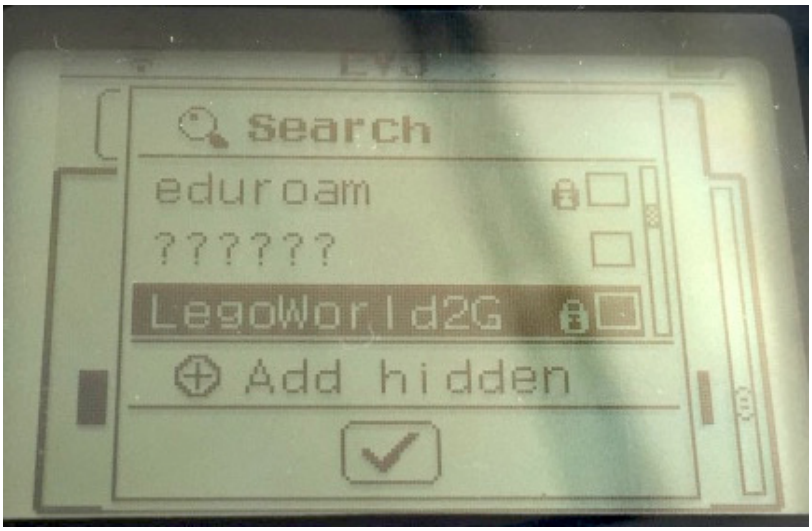

Figure 4: The network search menu of an EV3 robot.

e. Select and connect to your network.

f. Once your robot has connected, highlight your network's name and select it. It will display the IP address of the robot. Write this down, as it will be needed shortly.

Figure 5: The WiFi Info screen for the "LegoWorld2Ghz" network.

2. On a computer with the ability to use telnet and run soar_client, open two terminal windows. One will be for telnet and the other for the soar client.

   Windows users: telnet is built-into Linux and OS X. One way to use telnet on a Windows machine is to download "PuTTY".

3. Connecting to the robot using telnet for different operating systems. (Figure 6)

   OS X and Linux users: In the terminal for telnet, connect to your robot with the command "telnet [ip-address]" where [ip-address] is the IP of your robot.

   Windows users: Run "PuTTY" to open the putty window. Check the telnet box, then type in the IP address and connect.

   The username is "root" and there is no password.

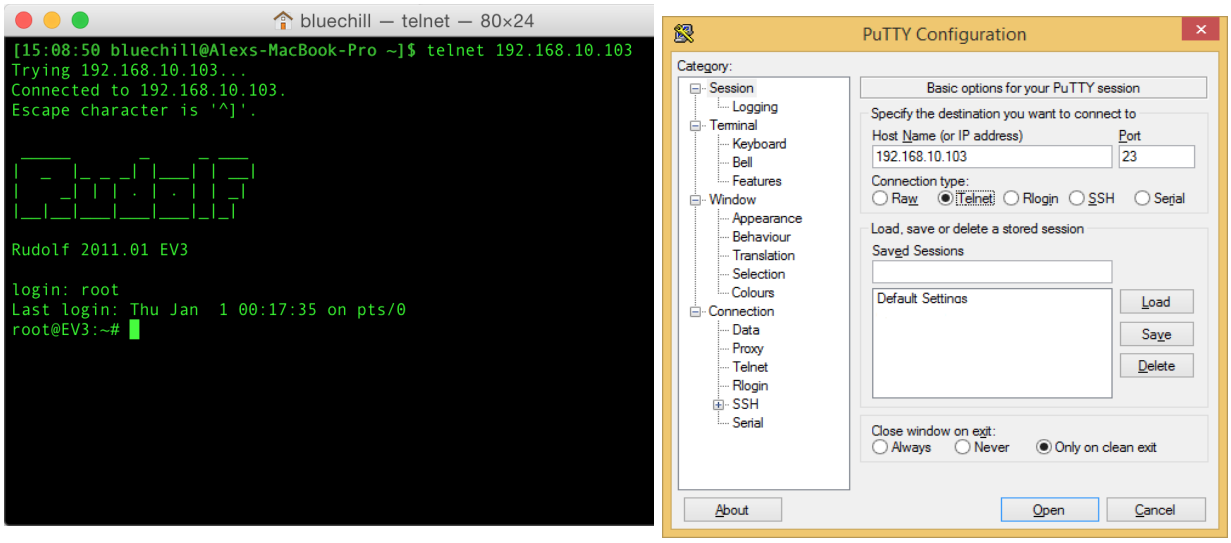

Figure 6: Connecting to the EV3 robot. For Linux and Mac Users on the left, for Windows users on the right (using PuTTY)

4. Navigate to "/media/card/". Then, type the following command
   `LD_LIBRARY_PATH=./lib ./ev3_server`
   This will launch the server on the robot and allow our Soar code to communicate with it.

5. In the terminal for the Soar client: navigate to the directory where your Soar client binary is. Execute it as follows
   `./soar_client [ip-address] [soar-code-path]`
   where `[ip-address]` is the robot's IP, and `[soar-code-path]` is the path to your Soar code will be stored. For simplification later on, store your Soar code in the directory that also contains the completed versions of the tutorial code (denoted with the suffix "`-completed`").

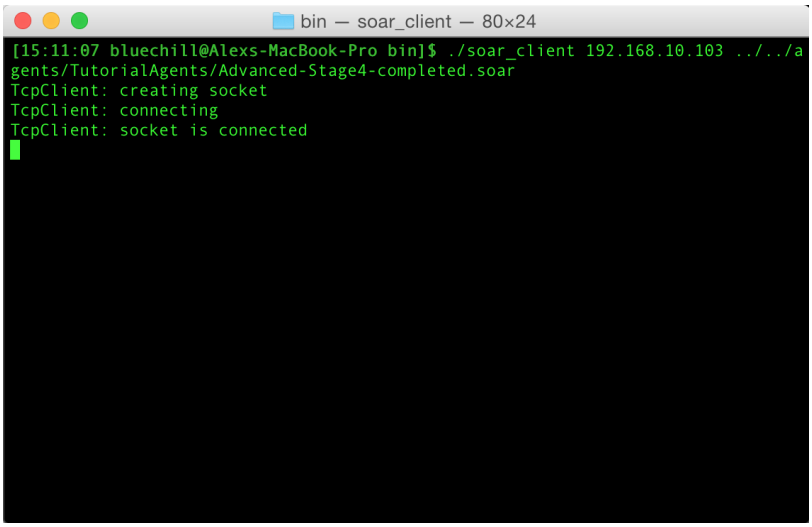

Figure 7: Connecting Soar to the EV3 robot using the soar_client command.

6. You should see in both windows that they are now connected to each other. Note that you can leave the ev3_server program running on the robot and just restart the client if you want to relaunch the agent.

Beginner Tutorial

In this tutorial, you create a simple line-following robot. In the first stage, you write a Soar program that allows the robot to follow a line. In the later stages, you modify this code to do reasoning in a sub-state via an operator no-change. Lastly, you modify the code to support turning around when the robot detects the red color.

Some helpful terminology:
"WME" stands for "working memory element"; "WM" stands for "working memory."

You can find more terminology used in the glossary at the end of the tutorial.

Stage 1: Simple Line Follower

For this stage, we provide starter code. You can find this code in the `Beginner-Stage1-starter.soar` file. We recommend you copy this to a new file, `Beginner-Stage1.soar`, so you can modify it freely and start over if necessary.

In this Stage the robot starts out on a black line surrounded by white. The robot has three color-sensors placed in a row on its underside. The center color-sensor of the robot should be aligned with the black line (Figure 8). The left and right color-sensors should be on either side of the line, detecting white. Since only the center color-sensor detects black, the robot should `move-forward`.

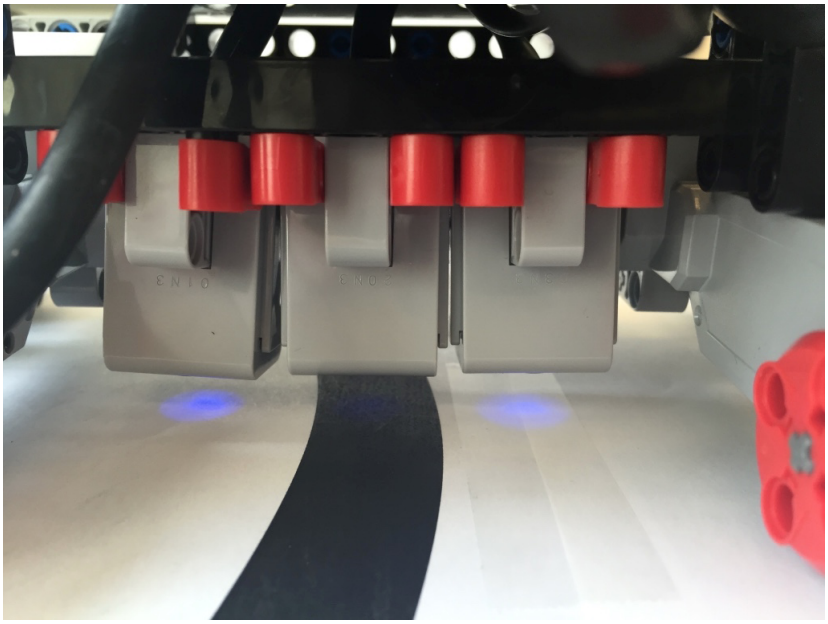

Figure 8: The center color sensor of an EV3 robot aligned to a black line.

When the robot travels along a curved path, one of the side color-sensors will detect black, while the remaining two sensors detect white. For example, in a path curving to the right, the left and center color-sensors will detect white and the right color-sensor will detect black (Figure 9). The robot will then turn to the right (`move-right`) to get back on the line and continue.

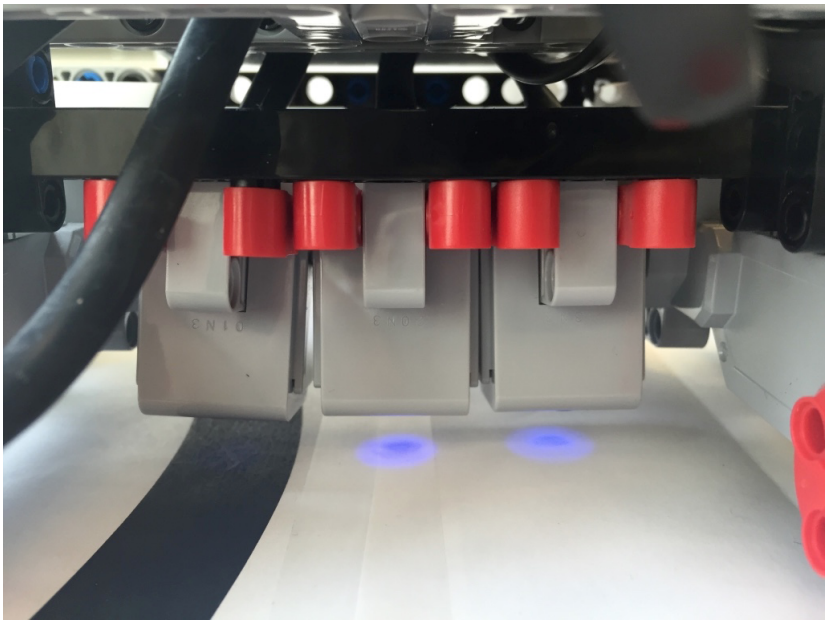

Figure 9: The right color sensor of an EV3 robot aligned to a black line, curving to the right.

The starter code contains 6 rules, three operators. In this stage, you write the proposals for the operators `move-forward`, `move-left`, and `move-right`. We have already completed the apply rules for you. Below are instructions on how to write the proposal for `move-forward`. The proposals for `move-left` and `move-right` are extremely similar to the proposal for `move-forward`. We have replicated the starter code for the `move-forward` proposal below for your convenience.

```
sp {Stage1*propose*move-forward
      # Conditions to test for:
      #     ^name
      #     ^mode
      #     ^color-forward
      #     ^color-turn
      #     ^color-sensor-values <color-sensor-values>
      #     <color-sensor-values> ^center <forward>
      #     <color-sensor-values> ^left …
      #     <color-sensor-values> ^right ...
-->
      # Create the operator here
}
```

This is the proposal for the `move-forward` operator. Currently, it does not do anything. You should test that the following conditions are true, on the top state:

- `^name` equal to `line-follower`

The `^name` test should be the first condition in your proposal; `line-follower` is the name of the top-state. This condition ensures this proposal will only fire for the top-state. It will look like: `state <s> ^name line-follower`

`<s>` will now contain the identifier of the top-state. You should use `<s>` for testing for the existence of the other WMEs on the top-state, shown below:

- `^mode` equal to `follow`

The `^mode` of the top-state signifies whether the robot should be stopped or following the line. If a button on the robot is pressed, the mode will change to `stopped` and the robot will stop following the line.

- `^color-forward`

The `^color-forward` WME stores the value that the center line is colored. In our case this is black. **Store its value** into a variable, `<forward>`

- ^color-turn

The ^color-turn WME stores the value that the center color sensor will see when the robot should turn to get back on the line. In our case this is white. **Store its value** into a variable, <turn>

- ^color-sensor-values

The ^color-sensor-values WME has an identifier for its value. This identifier contains information about the color-sensors' current values. There are three WMEs on this identifier, with attributes ^left, ^center, and ^right: these store the current color detected by the respective sensors. You should have one condition store ^color-sensor-values' value into a variable, <values>. You then should create another three conditions that, using the <values> variable, check that the ^left and ^right color-sensors detect the <turn> color and the ^center color-sensor detects the <forward> color.

The move-forward operator should be proposed with an acceptable preference, +. It should also be proposed with ^name move-forward.

Once you have completed the move-forward operator you should move on to the move-left and move-right operators. Left and right have slightly different conditions from move-forward, but are otherwise very similar to move-forward. The conditions are below:

Left: ^center <turn> ^left <forward> ^right <turn>

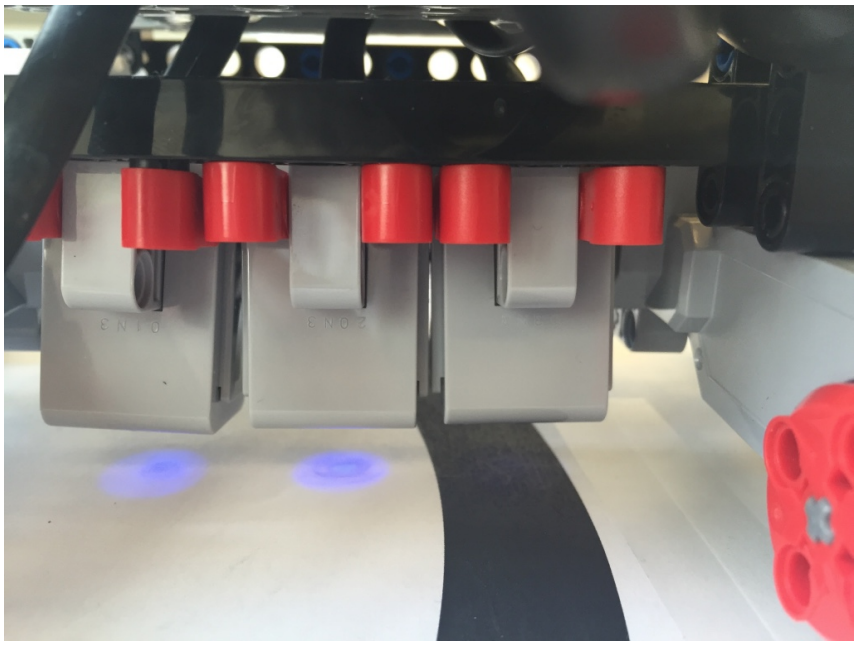

Figure 10: The left color sensor of an EV3 robot aligned to a black line.

Right: `^center <turn> ^left <turn> ^right <forward>`

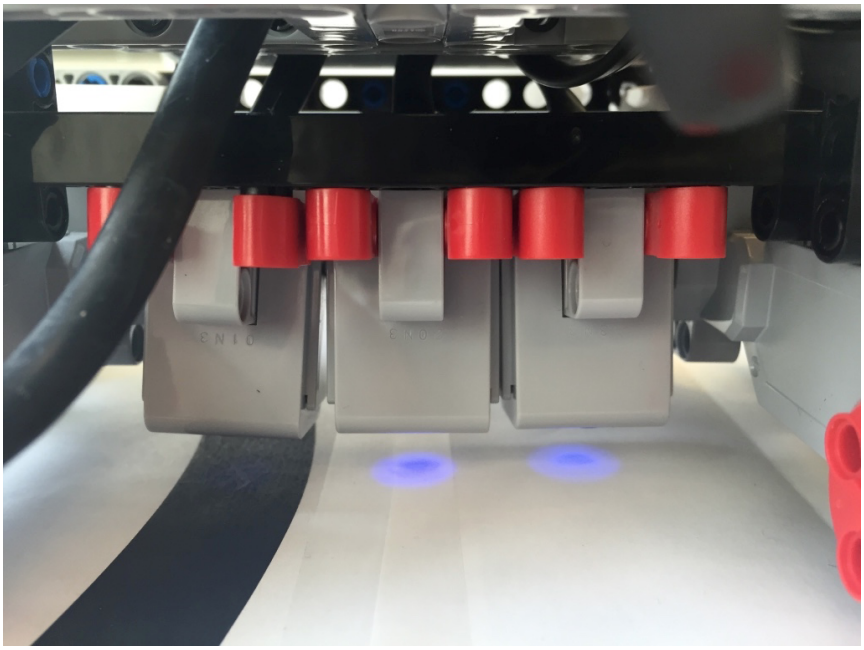

Figure 11: The right color sensor of an EV3 robot aligned to a black line.

After you have completed all three operators or if you get stuck, you should look at `Beginner-Stage1-completed.soar`. This file contains the completed code for this stage; it is implemented exactly how we asked you to complete it. We **strongly** urge you to attempt writing the code yourself and trying to solve it on your own before looking at the completed code.

Stage 2: Sub-state Line Follower

Congratulations on completing Stage 1! The goal of Stage 2 is to create a substate, `follow-line`, which will handle all the calculations and operations necessary to follow the black line. This will separate following the line from other actions. In Soar, it is common to use sub-states to denote high-level actions, and to perform the reasoning and intermediate steps needed to complete the action in the sub-state. In our case, the high-level action is `follow-line` and to do that the robot must `move-forward`, `move-left`, and `move-right` within the `follow-line` sub-state.

Before we begin, copy `Beginner-Stage1.soar` to `Beginner-Stage2.soar`. In Stage 2 you modify rules from the first stage and add two new rules, `Stage2*propose*follow-line` and `Stage2*elaboration*copy-sub-state-structures`. They create the `follow-line` sub-state and copy structures to the sub-state so that our movement operators will work,

respectively. **You also have to modify your proposals from Stage 1 to only be proposed within the "follow-line" sub-state,** instead of the top-state. In addition, like Stage 1, we have provided completed code which does what we describe below. As with Stage 1, you are **strongly** urged to write this code before looking at the completed code.

- `Stage2*propose*follow-line`

This `operator` will create the `follow-line` sub-state. It will use an `operator no-change` *impasse* to create the sub-state. An `operator no-change` is one type of impasse. Impasses occur when Soar needs to do more reasoning to complete the operator. A sub-state will be created to do this reasoning. An `operator no-change` impasse occurs when an `operator` is selected but nothing causes the proposal for the operator to be *retracted*. In our case, we will create this impasse by not creating apply rules for our `operator`. Our proposal will only be retracted when the `^mode` of the robot changes to `stopped` due to a button press. The `^mode` will change to `follow` when a button is pressed again.

To do this, our proposal should test for two things:

`^name` with a value equal to `line-follower`
`^mode` with a value equal to `follow`

If these conditions are met, our `operator` should be proposed with an acceptable preference (plus sign) and a `^name` of `follow-line`.

We have provided common code to initialize the sub-state for you. Since our operator is named `follow-line` it will create a WME, `^name`, with a value of `follow-line` on the sub-state. Your movement operators will test for a state named `follow-line` instead of `line-follower`.

- `Stage2*elaboration*copy-sub-state-structures`

In order for our movement operators to work, we need some values from the top-state. We can either test for them on the top-state directly using the link to the `^superstate` or we can copy the structures down from the superstate. This elaboration copies down the structures from the superstate to our sub-state.

Our elaboration will test for the following WMEs on the superstate:

```
state <s> ^superstate <ss>
<ss> ^name line-follower
<ss> ^io <io>
```

```
<ss> ^color-sensor-values <color-sensor-values>
<ss> ^left-motor-port <left-motor-port>
<ss> ^right-motor-port <right-motor-port>
<ss> ^color-turn <color-turn>
<ss> ^color-forward <color-forward>
```

Note: `<ss>` is the identifier of the superstate, set by the first line of WMEs above

The elaboration will then copy each WME to the `follow-line` sub-state. To do this, on the RHS of the rule you will create each WME with a parent of `<s>` like so:

```
sp {Stage2*elaboration*copy-sub-state-structures
      (state <s> ^name line-follower)
      ...
-->
      (<s> ^io <io>)
      ...
}
```

You should copy each WME tested except for the super-state's name WME, `^name line-follower`.

- `Stage1*propose*move-forward/left/right`

These rules, like in Stage 1, are for moving the robot. `move-forward` moves the robot forward, `move-left` rotates the robot left, and `move-right` rotates the robot right. Unlike Stage 1 though, in Stage 2 these rules will be proposed and selected within the `follow-line` sub-state.

To do this, you should modify each rule from Stage 1 such that:

`(state <s> ^name line-follower)` now reads `(state <s> ^name follow-line)`

In addition, you should remove the `^mode follow` WME test as that WME does not exist in our sub-state. Our elaboration copies the other WMEs our rules test for down to our sub-state allowing us to leave the rest of the rule as is.

Stage 3: Turning The Robot

This is the most complicated stage of the beginner tutorial. In this stage, the robot will function as in Stage 2 except that each time the center color sensor of the robot sees the color red, it will turn around and continue following the line.

You will be reusing your Soar code from Stage 2 without modification. To do that, you should create a file `Beginner-Stage3.soar` and at the top use the `source` command to source in your other Soar rules. If you are familiar with other programming languages such as `C++`, the `source` command is very similar to `#include`. The whole command will look like:

```
source Beginner-Stage2.soar
```

In this stage you will create 1 elaboration, 1 preference, and 4 operators. All of these rules will fire on the top-state, the `^name line-follower` state. The elaboration will simplify the proposals and apply rules for our operators by creating a copy of the current rotation value of the motors. The preference will ensure our Stage 3 rules are selected over the follow-line operator when red is detected.

The 4 operators will function as follows: 1) The `record-red` operator will be selected when we see red. It will record that we've seen the red color with our center color sensor. 2) The `turn-180` operator will be selected after we've recorded that we've seen the red color. It will output to the motors of the robot to turn around. Once the robot has turned around, another apply rule for the turn-180 operator will be selected to record that we have turned around and to stop the motors from continuing our turn. 3) Then, if necessary, the `move-forward` operator will move the robot off of the red square to allow us to continue following the line. 4) Finally, the `clear-flags` operator will be selected to clean up the top-state allowing us to turn around if we see the color red again.

- `elaborate*motor-sensor-values`

This elaboration will copy the amount each motor has rotated to the top-state. This rule is not necessary but it simplifies our operator code and makes our operator code easier to understand. Each motor can be found on the `^io.input-link` of the top-state. They each have the following structure:

```
(<io.input-link.motor> ^amount-rotated <rotation>
                       ^port <motor-port>)
```

The elaboration will test for each motor's `^amount-rotated` value and copy it to the top-state. There are two motors: a left motor and a right motor. To determine which motor is which, we have provided for you on the top-state two structures: `^left-motor-port` and `^right-motor-port`. These contain the ports of the left motor and the right motor respectively. Below is the Soar representation of the motors, and an example of how to reach the left and right motor ports and their `^amount-rotated` values.

```
(state <s>        ^io.input-link <input-link>)
(<input-link>     ^motor <left>)
```

```
(<left>          ^port <left-motor-port>
                 ^amount-rotated <left-motor-rotation>)
```

The right motor is identical except you should replace “left” with “right.” You should then copy the left motor’s amount rotated and right motor’s amount rotated to `^left-motor-sensor-value` and `^right-motor-sensor-value` on the top state, respectively.

- `prefer*180*operators*over*follow-line`

This preference ensures that our operators for turning around are selected over the follow-line sub-state operator if they’re proposed. As a reminder, to prefer one operator over another you should use the following syntax:

```
(<s> ^operator <o1> > <o2>)
```

This says that the operator with identifier `<o1>` should be selected over the operator with identifier `<o2>`. In our case, we should test that `<o2> ^name` is equal to `follow-line` and that `<o1>` is either `record-red`, `turn-180`, `move-forward`, or `clear-flags`. We can either create 4 different preference rules or use a *disjunction*. A disjunction is an OR statement in soar. It can only be used on the LHS of a rule and tests that something is equal to one of the items specified in the test. It looks like this:

```
(<id> ^<attribute> << item1 item2 item3 … itemN >>)
```

A disjunction does not allow variables, only constants. In our case, the disjunction will be:

```
<< record-red turn-180 move-forward clear-flags >>
```

The whole test for the first operator will look like:

```
<o1> ^name << record-red turn-180 move-forward clear-flags >>
```

You should test that `<o1>`’s name is any of our operator’s names (the test is immediately above using the disjunction) and that `<o2>`’s name is `follow-line`. Then, you should, on the RHS of the rule, create a preference for `<o1>` over `<o2>`.

- Operator: `record-red`

This operator will record that we have seen the red color with our center color-sensor.

It will be proposed when we see this color but have not yet recorded that we have, via a `^detected-red` WME on the top-state. It will then propose an operator with `^name`

`record-red` and `^action stop`. The `^action stop` will allow our common-code to fire in addition to your apply rule, halting the robot. It will do this by setting the power of the motors to 0.

To apply the operator, create a `^detected-red` WME on the top-state with a value of either `true` or `YES`. It does not matter which because in later rules you will check for the existence of `^detected-red`, not its value: the value is only for clarity.

- Operator: `turn-180`

This operator will fire after we have recorded that we have seen red. It will output to the motors to turn the robot around.

This operator will check for the existence of the `^detected-red` flag on the top-state. It should also check for the absence of `^turn-complete` on the top-state. Then it must propose an operator with `^name turn-180`.

There are three apply rules for our `turn-180` operator. The first, provided in its entirety below, outputs the commands to the motors to turn around.

```
# Start the rotation. Arbitrarily chose to start by
# rotating left. However, because we're rotating 180
# one could choose to rotate right.
sp {Stage3*apply*turn-180*motor-commands
      (state <s>  ^name line-follower
                  ^operator.name turn-180
                  ^io.output-link <out>
                  ^left-motor-port <left-port>
                  ^right-motor-port <right-port>)
-->
      (<out>      ^motor <left>
                  ^motor <right>)
      (<left>     ^port <left-port>
                  ^set-direction forward
                  ^set-power 40)
      (<right>    ^port <right-port>
                  ^set-direction backward
                  ^set-power 40)
}
```

In order for us to know when we have turned 180 degrees, we will use the `^amount-rotated` value from the right motor conveniently copied to the top-state by the elaboration we created earlier. We have determined ahead of time for you that the robot will

have turned 180 degrees when the amount the right motor has rotated is 1000 less than when rotation began. The application rule above commands the right motor to rotate backwards. This causes the `^amount-rotated` value for the right motor to decrease, which is why the right motor's final `^amount-rotated` value will be less than its starting `^amount-rotated` value.

The second apply rule calculates and records the value 1000 less than the right motor's current `^amount-rotated` value. It will store this value into a WME on the top-state, `^180-right-motor-sensor-value`. Our apply rule must fire when the `turn-180` operator is selected and there does not exist a `^180-right-motor-sensor-value`. It will create the `^180-right-motor-sensor-value` with a value of `^right-motor-sensor-value` minus 1000. If you recall, `^right-motor-sensor-value` is the `^amount-rotated` value of the right motor copied to the top-state by the elaboration we created earlier. To do this in Soar you may use the following Soar code:

```
(<s> ^180-right-motor-sensor-value (- <right-motor-sensor-value> 1000))
```

The third apply rule must test that the `^right-motor-sensor-value <=` to the `^180-right-motor-sensor-value`. If the conditions match, it must create the `^turn-complete` flag on the top-state with either `true` or `YES`, whichever you used for the `^detected-red` flag to be consistent.

- Operator: `move-forward`

This operator will move the robot off of the red square if, after completing the turn, the center color sees red. This will allow us to continue following the line without modification to the `follow-line` sub-state. This operator is meant to account for the (although unlikely) situation where the turn is nearly perfectly in-place.

If you recall, the apply rule for operator `move-forward` from Stage 1 and 2 only tests for `^operator.name move-forward`. It does not test that the state is `follow-line`. We will reuse the apply rule to tell the motors to move the robot forward. You should propose a move-forward operator if the following conditions are met:

1) `^turn-complete` exists on the top-state
2) `^color-sensor-values.center red`

- Operator: `clear-flags` (clean-up)

Once the robot has completed the turn and moved off the red square, if necessary, we need to clean up the state to allow the robot to turn around again. This operator, `clear-flags`, removes all the flags we created on the top-state with *o-support*. In Soar, there are two types

of support, *i-support*, and *o-support*. *i-support*, instantiation-support, means that the structure will exist in memory so long as the rule that created the structure has not been retracted (the rule is still instantiated). *o-support*, in contrast, allows a structure to persist in WM even after the rule that created it has been retracted. *o-support* is normally given via operators. Any structure created by a rule that tests an operator being selected is given *o-support*. In this Stage, the `^detected-red`, `^turn-completed`, and `^180-right-motor-sensor-value` flags have *o-support*. Since they will not be removed automatically, we need to remove them ourselves. To do that we will use our `clear-flags` operator.

This operator is proposed once the robot has completed its turn and the `^color-sensor-value.center <> red`. The apply rule will remove the following flags:

```
^detected-red
^180-right-motor-sensor-value
^turn-complete
```

To remove an *o-supported* WME in Soar, you must use the following syntax on the RHS:

```
(<s> ^<structure> <structure-value> -)
```

Where `<structure>` is the attribute of the WME you wish to remove and `<structure-value>` is the value of the WME tested for on the LHS. If you do not test for `<structure-value>`, your RHS action will not remove anything.

Congratulations! You have finished the beginner tutorial! You can now start on the Advanced Tutorial which will build on the concepts learned here, teach you more about working memory, and teach you about semantic and episodic memory.

Advanced Tutorial

In this tutorial you will build upon what you learned in the beginner tutorial. You will be building an exploring robot. In Stage 1, you will create a robot which, using code found in WM, chooses which direction to go upon seeing a junction. In Stage 2 you will switch from WM to SMem and recall where to go from SMem. In Stage 3 you will create a robot to explore the map and remember what each junction color means. In Stage 4 you will use episodic memory to recall what was the previous junction color.

For each of these stages you will be using the Lego Mindstorms EV3 Simulator written specifically for this tutorial. The simulator uses the same Soar code that you will use for the robot. If your code works in the simulator, it should also work on the robot. We will be using

the simulator to speed up development of your agents and make it easier for you to debug your Soar code.

Robot Simulator

The robot simulator is a java-based application which simulates all the necessary components of the robot. It does not simulate the robot in its entirety. For instance, it does not simulate the robot buttons. It does simulate motors and sensors. It is also discrete; it does not simulate time steps at a finer grain than one tile per step. Furthermore, it does not simulate sensor noise, it is a "perfect" information simulator. For our purposes, and for the purpose of this tutorial, it is good enough though.

To start, open up the robot simulator. You should see the following screen:

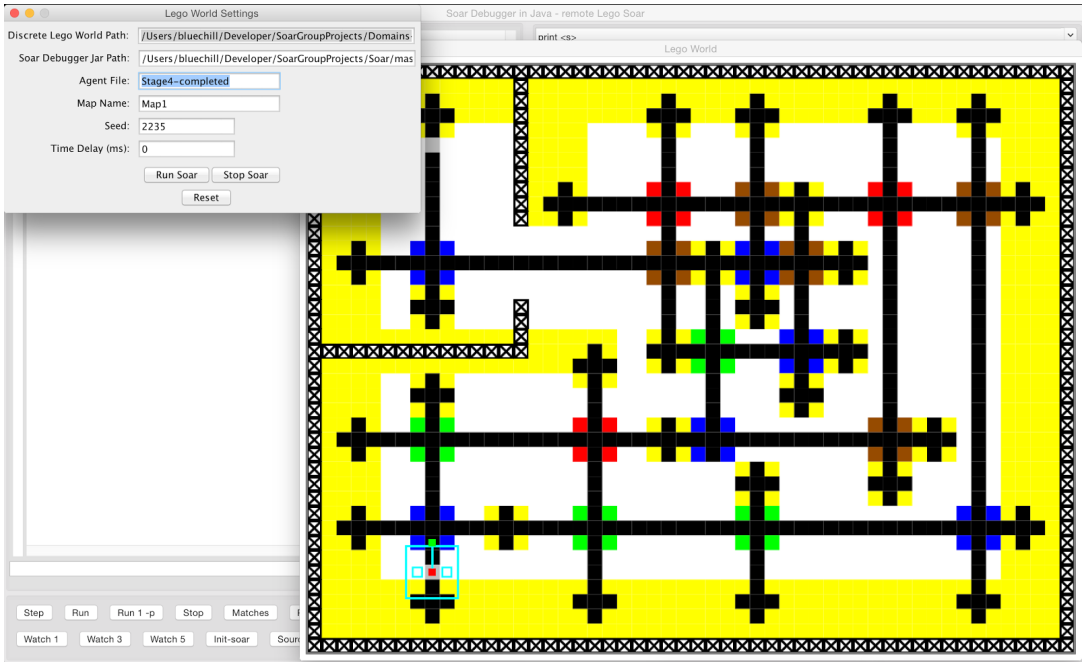

Figure 12: Starting up the robot simulator.

There are three windows. The first is the "Lego World Settings." In this window, you can change settings such as where the simulator code is located, where to find the Soar Java Debugger, which map you're using, the seed, and the time delay between steps. You can also run the currently loaded Soar agent and stop it. If you wish to change anything but the time delay you have to click the "reset" button for the settings to take effect. The reset button also resets the entire simulator. In addition, to change the time delay, you have stop and restart the robot via the run and stop buttons.

The other window is "Lego World" which is a display of the current map and where the robot is. Each color corresponds to the same color on the physical map. In addition, one unit (one

tile) is equal to 2 centimeter on the physical map. The robot is colored with cyan. A line extends outward from the center of the robot towards its front; this line shows the cardinal orientation of the robot (north, south, east, west). The three cyan squares in the middle of the robot are the color-sensors showing which color they currently perceive. At the tip of the orientation line of the robot is the touch sensor; when pressed, it will turn red, and when released, it will be green.

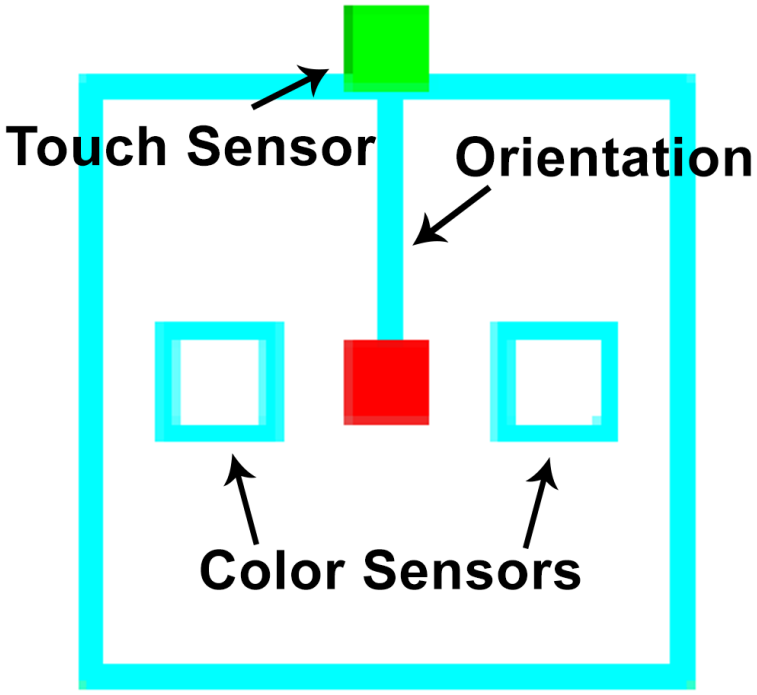

Figure 13: The robot simulator representation of the robot.

As with the real robot, the following map colors represent junctions: red, green, blue and brown. Yellow represents a special type of junction: an “end-of-path” junction. A junction consists of four colored tiles and a black “plus” in the middle.

Figure 14: A red junction.

Tiles marked by a black “X” are walls and off-limits (Figure 15). If your robot does not read the colors correctly, it will fail and drive off the path.

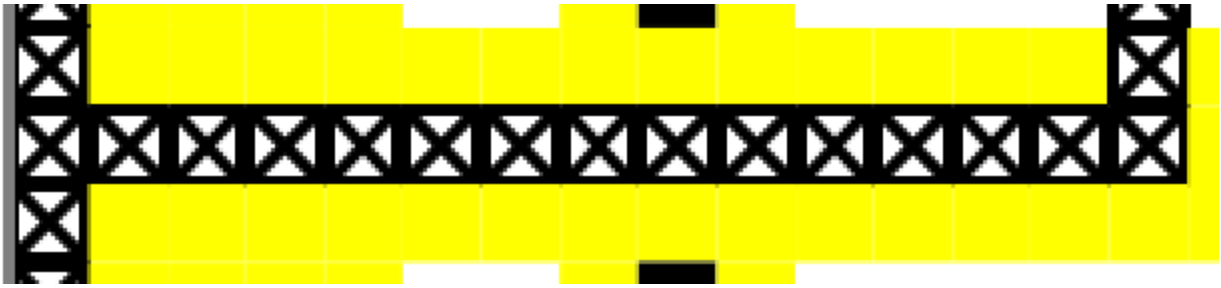

Figure 15: Tiles blocking off an area.

Before you begin on the actual tutorial, set the “Agent File” to be `Advanced-Stage3-completed.soar`, press reset, and run the simulation. You should see the robot explore the map and learn what each color means and eventually reach the finish tile.

Stage 1: WM Junction/Line Follower

As with the beginner tutorial, you should copy `Advanced-Stage1-starter.soar` to a new file, and name it `Advanced-Stage1.soar`. In this stage you will create one operator and one preference.

In this stage you will be creating a line following robot which, upon reaching a junction, will decide which direction to go in. Throughout the advanced tutorial, you will be writing the code to decide which direction to go in. Below is an outline of the order of operators that will be called upon reaching a junction. Your operators for all these stages will be selected at stage 3:

1. enter-junction
2. align-to-junction
3. select-direction and the other operators of your code
4. traverse-junction
5. exit-junction

In this stage, you will write the `select-direction` operator which you will reuse in stages 2, 3, and 4. The `select-direction` operator will select which direction that robot should go in upon reaching a specific colored junction. For example, if the robot reaches a blue junction, it should go north. The mapping of colors to directions for Stage 1 can be found in the `^color-map` on the top state on WM. The mapping is replicated below:

```
(<s> ^color-map <map>)
(<map>      ^red south
            ^green west
```

```
^blue north
^brown east)
```

Your `select-direction` operator will propose to go in each direction possible and then you will create a preference rule to prefer the correct direction given via the `^color-map`.

- Operator: `select-direction`

This operator will select which direction to go in upon reaching a junction. It will propose to go in all directions possible and rely on a preference operator to select the correct direction.

This operator should propose to go all directions found on the multi-valued top-state attribute `^direction`. This will create four proposals, one each for north, east, south, west. When it applies it should create a flag on the top-state `^selected-direction` with the direction from the operator.

- `prefer*right-color`

You should also create a preference for this operator. On the top state there is a `^color-map` which contains the colors and their direction. For instance, blue is `^color-map.blue north`. You should match on both the junction color and the direction for the color. This color can be found via the `^junction-color` WME on the top-state. The preference should mark the operator which correctly identifies the direction for the color with a best preference.

Stage 2: SMem Junction/Line Follower

In this stage you should source in `Advanced-Stage1.soar` and then `Common-Advanced-Stage2.soar`, **in that order**. You will be creating an operator and a preference for this operator.

As a **quick review**, Semantic Memory (SMem) is a declarative knowledge memory store. Soar can use it to retrieve and store facts and other knowledge.

In this stage, you will be using SMem to retrieve the correct direction to go in for `^junction-color`. You will query SMem for the mapping and then copy the mapping's direction to the `^color-map` in working memory. Only copy the direction for **the current color**, even though you may be able to copy all the color-direction mappings to the `^color-map`.

- Operator: `query-for-direction`

This operator should query SMem for the correct direction. Its proposal should check for the non-existence of `^color-map.<color>`, where `<color>` is the current junction color. We have provided the following map in SMem:

```
(<s>        ^color-map <map>)
(<map>      ^red south
            ^green east
            ^blue north
            ^brown west)
```

You should query SMem for `^<color> <direction>`, where `<color>` is the current junction color. In addition, this operator should also have an apply rule for handling creating the mapping. It should check for the result of the smem query, `^smem.result.retrieved.<color> <direction>` and add it to the color map in WM. Finally, you should have one last rule which cleans up SMem upon a successful query. It should remove the query from the smem `^command` link once the query has been processed.

This operator will no-change for one cycle until SMem returns the retrieval result and then continue as normal.

Your preference should prefer `query-for-direction` over `select-direction`, otherwise the preference rule for `select-direction` will not fire and you will randomly traverse the map.

Stage 3: WM Junction/Line Explorer

In this stage you create one operator and two preferences. You should create a new file and source in `Advanced-Stage1.soar` and `Common-Advanced-Stage3.soar`, **in that order**.

In this stage you create an agent that explores the map through trial and error and memorization. Upon reaching an unknown type of junction (i.e. `^junction-color` for that junction has not been seen before), the robot shall explore directions at random until it finds the correct one; upon reaching a known junction, the robot will remember the correct cardinal direction for that `^junction-color` and take it in the future. Each junction has four paths, but for an unknown junction there are only three possible paths the robot can take: one path is eliminated because it will take the robot back the way it just came. There are two directions per junction that lead to yellow junctions; yellow junctions indicate that the wrong direction was chosen for the previous junction, and that the robot should turn back. If the robot reaches a non-yellow junction, then the correct direction was chosen for the previous junction; this will be represented in Soar by marking all the incorrect directions as `blocked` for that `^previous-junction-color`.

- Operator: `mark-direction-blocked`

This operator is used to represent that the direction just taken by the robot is wrong by marking the direction as `blocked`. The operator should have three attributes:

1. `name`: `mark-direction-blocked`
2. `color`: the junction whose directions are being marked as `blocked`
3. `direction`: the direction to block

The operator is proposed in three different scenarios, and thus needs three proposal rules:

- First proposal:

The robot reaches a yellow junction. In this case, the direction taken was wrong and should be marked as `blocked`.

Propose the operator when the robot reaches a yellow junction from a non-yellow junction. It should block the current direction for the previous junction's color.

- Second proposal

The robot reaches a non-yellow junction that has a color not yet seen (unknown). In this case, the direction directly opposing the `^current-direction` should be marked as `blocked` for the current junction color to prevent the robot from going backwards.

Propose the operator when entering an unknown junction from another junction. It should block the opposite of the current direction to prevent you from going backwards. You can use the direction-map found on the top state to get the backwards direction.

- Third proposal

The robot reaches a non-yellow junction. In this case, all the directions not marked as `blocked`, excluding the correct one just taken, should be marked as `blocked` for the previous junction color.

Propose operator `mark-direction-blocked` when entering a non-yellow junction from a non-yellow junction. In other words, the proposal should fire if the robot travels from a red/blue/green/brown junction to another red/blue/green/brown junction. You can use `^junction-color` and `^previous-junction-color` to aid with this.

This proposal may fire multiple times. For example, assume a scenario in which the robot leaves a blue junction going north and enters a green junction. In this scenario, the north path was the only path the robot attempted from the blue junction: it happened to take the correct

path on the first try. That means three directions are not marked as `blocked` for blue junctions (one is blocked by the second proposal to avoid going backwards). One of the three is the correct path just taken (north) and should remain unblocked, which leaves two directions that are unblocked and should be blocked. Therefore, this proposal will fire twice.

- Apply Rule

This should take the color and direction found on the mark-blocked-direction operator and add the direction for the operator as a blocked direction for the color. In other words, you should add `^color-map.<color>.blocked-direction.<direction>` to the top-state from the operator.

- Preferences

Create a preference preferring `mark-direction-blocked` over select-direction, otherwise the robot will randomly traverse the map instead of learning the proper direction.

Create another preference to prevent you from randomly traversing the map. This preference should *reject* all `select-direction` operators proposing to go in a `blocked` direction. To reject an operator you should use the following syntax:

```
(<s> ^operator <o> -)
```

Note the minus. This will remove the acceptable preference for the operator (`<o>`) proposal causing the operator proposal to be retracted, preventing it from being selected.

Stage 4: EpMem Junction/Line Explorer

As a **quick review**, Episodic Memory (EpMem) is a long-term memory which stores the contents of WM at specific intervals for later retrieval. It allows an agent to recall previous data stored in WM.

In this stage you will add one operator and one preference to the `Advanced-Stage3.soar` code. First, you should create a new file `Advanced-Stage4.soar` and source in `Advanced-Stage3.soar` followed by `Common-Advanced-Stage4.soar`.

In this stage, `Common-Advanced-Stage4.soar` will remove the previous junction color upon leaving a junction, using an operator called `exit-junction`. You will use episodic memory to retrieve it. Episodic memory by default saves the top-state during the output phase of Soar. For this stage, we have changed that so it saves during the selection phase of Soar. In other words, EpMem saves the state of Soar immediately after an operator is selected. This allows us to query for the last time an operator was selected. In our case, we will be querying for the last time an `exit-junction` operator was selected. We will use this to grab the

previous ^junction-color, and store it as our current ^previous-junction-color on the top-state. This will allow our Stage 3 rules to be proposed and fired as normal.

You should also note that the retrieve-previous-junction-color operator and preference is very similar to Advanced-Stage2. In fact, ignoring the differences between EpMem and SMem command/result structures, it is the same format.

- Operator: retrieve-previous-junction-color

This operator proposes to retrieve the ^previous-junction-color when it is not present on the top-state. It fires before any of the other tutorial rules you have written fire allowing them to act as if nothing is different. Note though that it still fires after enter-junction as it requires a ^junction-color to be present! It proposes to retrieve the junction color and then causes an operator no-change followed by a state no-change followed by our wait operator until the episodic memory query returns a result. In practice this means it only causes an operator no-change before the second and third apply rules fire, creating the ^previous-junction-color WME and causing our proposal to retract. The first apply rule creates the query. The second creates the ^previous-junction-color WME once EpMem returns a result. The third cleans up EpMem allowing us to query again in the future for other junctions.

- First apply rule

This creates the query for EpMem. You should query for an operator with the name exit-junction. Your query should look like: ^query.operator.name exit-junction.

- Second apply rule:

This will add the ^previous-junction-color WME to the state. The previous junction color is the ^junction-color when the exit-junction operator was selected. In other words, it is ^junction-color in the result you retrieved: ^epmem.result.retrieved.junction-color <previous-junction-color>.

- Third apply rule:

This should remove the query from the EpMem command link. It should test that we have retrieved the junction color from EpMem: check that ^epmem.result.retrieved.junction-color exists.

- Preference

This, like the SMem operator in Stage 2, creates a better-than preference that prefers retrieving the previous junction color over selecting the direction. This prevents us from traversing randomly.

Congratulations on completing the advanced tutorial! Hopefully you found this tutorial both fun and engaging. In addition, we hope that this tutorial was instructive and that you now have a better understanding of how to actually create Soar agents.

Appendix

Common Include Rules & What They Do

EpMem Reference

SMem Reference

Top State Flags & Their Meaning

Glossary

EpMem: Episodic Memory, a long-term memory which stores the contents of WM at specific intervals for later retrieval. Allows an agent to recall previous data stored in WM.

Impasse: A special type of state in which there either are no acceptable operators suggested or there is insufficient preferences to distinguish among the acceptable operators. There are five impasse types: tie, conflict, constraint-failure, operator no-change, and state no-change. In our tutorial we only use the operator no-change impasse. An operator no-change impasse occurs when the conditions for the operator to be proposed have not changed in between decision cycles to cause the proposal for the operator to be retracted.

LHS: Left-Hand Side. This is the part of a rule between the name and the arrow. The arrow separates the LHS from the RHS.

Retract: When an item in WM is removed, usually a WME or preference.

RHS: Right-Hand Side. This is the part of a rule between the arrow and the last curly bracket. The arrow separates the LHS from the RHS.

SMem: Semantic Memory, a declarative knowledge memory store. Stores “facts” and other knowledge which Soar can add to or retrieve from.

WM: Working Memory, the main memory of Soar.

WME: Working Memory Element, the three-tuple used for representing knowledge in Soar.
`(<parent> ^attribute <value>).`