16x62: Lab4

Due: Tuesday, Week 7

Introduction:

Now that you have developed the tools you need for navigating safely and reliably in the node-based world, things are going to get really interesting. This week you are going to use `GTNN`, `TurnTo`, and `WhatDoISee` to execute some plans. The robot won't have to do any planning, but *you* will. This lab should give you a feel for the complexity of planning and make you thankful that the robot will be doing the planning in later labs.

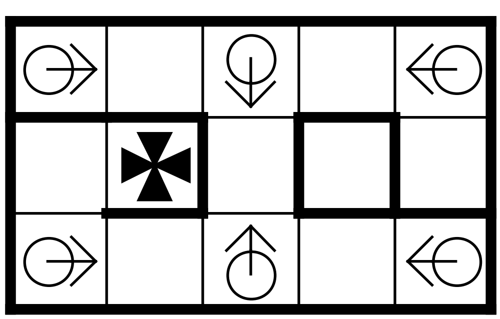
In the first assignment, you will develop a Conditional Plan to guide the robot from a limited set of possible starting positions. Then, you will write a Universal Plan which can achieve the goal from any starting position. And finally, you will write a State-Set Tracking Automata (which is almost as complicated as it sounds).

States and State-sets

We need a standardized way to describe the robot's state in the world. We represent **state** as a list of the form `(node direction)`, where `node` is the cell in the environment and `direction` is `u`, `r`, `d`, or `l` (corresponding to up, right, down, and left, respectively). For example, `((0,0), r)` means that the robot is in the lower left node facing right. A **state-set** is simply a list of 0 or more **states**. For example, `(((0,0),r), ((0,0),l))` is a valid state set depicting that we know we're in the lower left cell but we may be facing either left or right.

Assignment 4.0: Conditional Plans

Write the function: `GoHome()` When called in the environment below, the robot always reaches the goal (1,1) by executing a conditional plan, provided that it starts in any of the six specified starting positions and orientations. You may choose any implementation of conditional control flow: `if, case, etc.`.The final orientation at location (1,1) does not matter.

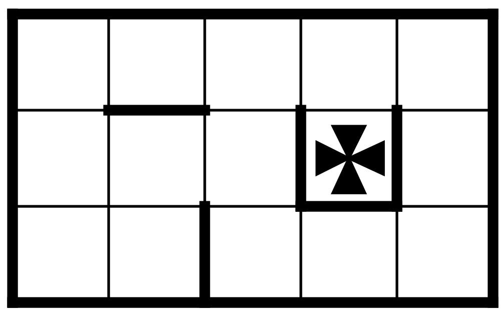


Environment for Assignment 4.0

We will test your solution by executing your program with one of the initial states depicted above. For this assignment (and from now on), the robot's starting position will always be in the center of a node and rotationally aligned with the world. **NOTE**: *this assignment should take you about 5 minutes!*

Assignment 4.1: Universal Plans

Write the function `UniPlan()` which solves the following problem: The robot will be placed in an unknown position and orientation in the environment pictured below. But it will be aligned and in the center of a cell. The robot must reach the goal (3,1) and then terminate (stop moving, not blow up).



Environment for Assignment 4.1

In this assignment, you will solve a navigation problem using a percept-activated universal plan. Your universal plan chooses one of three high levels outputs (i.e. `GTNN(1)`, `TurnTo(900), TurnTo(-900)`) based **solely** on the result of the high-level percepts (i.e. `WhatDoISee()`). This contrasts to assignment 4.0, where you create a conditional plan executor that uses internal state (a program counter) during execution. Like early reactive programs such as `SmartWander()`, this universal plan executor is totally functional, basing action decisions exclusively on current inputs. The difference between this and `SmartWander` is simply that the inputs and outputs for this assignment are higher level functions that you've built on top of `Sensors` and `RMove()`.

In gory detail, you will choose one of the three actions:

```
GTNN(1)
TurnTo(900)
TurnTo(-900)
```

You will do so each time the last action completed, and you may only use the current value of WhatDoISee() to determine which of these three actions you're choosing. This is a Universal Plan!

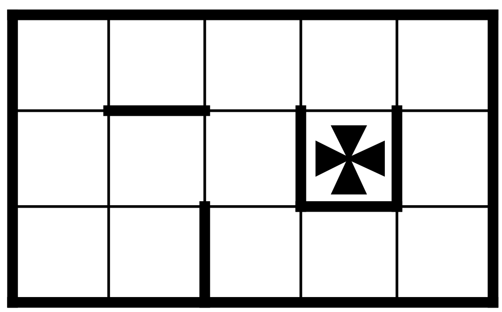
Obviously, one further action you will need will be to terminate execution when you get to the goal. We allow you this one departure from the rules above... (You could sing a song like "Daisy Daisy" or something, for instance.)

We will test your universal plan executor by placing the robot in several different initial starting positions and orientations and running your program until it terminates, or until we're bored.

Assignment 4.2: State-set Tracking Automata

The universal plan in the assignment above is not too smart because the plan is purely functional: the robot forgets all past experience with each new move. The robot can do better if it retains a history of what its seen. In fact, we can use this history to generate **optimal** behavior for the robot.

Write the function `SST()` which solves the following problem: The robot will be placed in an unknown position and orientation in the environment pictured below. The robot must use a state-set tracking universal plan to reach the goal (3,1) and terminate.



Environment for Assignment 4.2

The Algorithm

The program you write must be based on the following algorithm:

1 `state-set = {all possible positions and orientations}`
   Begin with a map of the world, but no information about the robot's location. Initialize the state-set to all possible states in the world.

2 `state-set = sees(WhatDoISee(), state-set)`
   Look at the current node and compare the percepts with those that the robot would see if it were in each state in the state-set. Filter out impossible states.

3 `action = GetAction(state-set)`
   Determine the robot's next action based on the current state-set. The action must be one of the three actions as in Assignment 4.1.

4 `do action`
   Execute the action.

5 `state-set = results(action, state-set)`
   Replace the state-set with the states the robot could be in after executing the above action from each state in the state-set.

6 `if (AtGoal(state-set)) terminate; else goto step 2`

If the robot is at the goal, exit with success; otherwise, go to step 2.

Most of this algorithm is doing state-set tracking: what are my possible states based on my current perceptual input, and what are my states based on the action I just took?

Sees

Given a set of states that the robot might be in, what states are consistent with the robots current percept?

The `sees` function filters the current state-set by removing states that are incompatible with the current percept. The resulting state-set will be a subset (not necessarily a proper subset!) of the original one. For each state in the state-set, you will need to compare the current percept with the expected percept for that state. For example, when you start (complete uncertainty), all the possible states are in your state set. Suppose that your robots immediately sees, as a result of WhatDoISee(), three walls around itself. Well then, `sees` will filter away all possible states except for one particular goal state; you're done in this case!

Results

Given an action and a set of states that the robot might be in, what is the result of applying the action to the state-set?

`Results` is a transformation function. For each state in the original state-set, you will need to compute the resulting state based on the specified action. Notice that it is possible for multiple states to map onto one, so you will need to make sure that you don't duplicate states in your new state-set. For example, in the five-node world, if the state-set were `(((0,0), r), ((1,0), r))` and the action were `(gtnn 1)`, the resulting state set would be `(((1,0), r))`.

GetAction

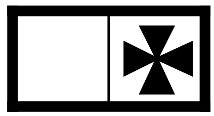
Given a set of states that the robot might be in, what is the proper action to take? Well, the answer seems quite easy in cases where the set of states contains precisely one state (a singleton set), but what if there are many states in the set?

This is not an easy question to answer, so we're giving you hints later in this handout.

We will test your state-set tracking automata by placing the robot in several different initial starting positions and orientations and running your program.

Appendix A: The Making of `GetAction`

First, let's consider the problem of representing `GetAction`'s decisions. We could create a big lookup table similar to the one in assignment 4.1 except much bigger than 16, with the first column listing each possible state set and the corresponding entry in the second column listing the correct action to perform.



The two-node world

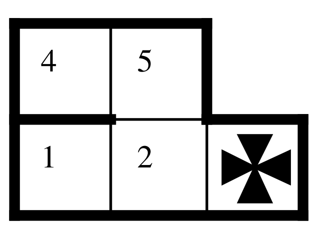
For example, given a simple two-node world shown above, the table would be:

state-set	action
(((0,0) u) ((0,0) r) ((0,0) d) ((0,0) l) ((1,0) u) ((1,0) r) ((1,0) d) ((1,0) l))	<action>
(((0,0) u) ((0,0) r) ((0,0) d) ((0,0) l) ((1,0) u) ((1,0) r) ((1,0) d))	<action>
(((0,0) u) ((0,0) r) ((0,0) d) ((0,0) l) ((1,0) u) ((1,0) r) ((1,0) l))	<action>
(((0,0) u) ((0,0) r) ((0,0) d) ((0,0) l) ((1,0) u) ((1,0) r))	<action>
... etc., etc., etc. ...	
(((0,0) r))	(gtnn 1)
(((0,0) d))	(turn-to 900)
(((0,0) l))	(turn-to 1800)

Unfortunately, for any reasonably large problem, this table can be a handful. For this assignment, the total number of entries would be $1.152921505 \times 10^{18}$. This is bad.

But don't give up! We can shrink the size of the table considerably by only including state-sets that can be reached from the initial state-set containing all possible states. For example, in the five-node world pictured below, there are only 12 possible state-sets after the first call to `sees` (once you filter based on the return value of `WhatDoISee()).`:

```
 1. (((0,0) r) ((0,1) r) ((2,0) l))
 2. (((0,0) u) ((0,1) u) ((2,0) d))
 3. (((0,0) d) ((0,1) d) ((2,0) u))
 4. (((0,0) l) ((0,1) l) ((2,0) r))
 5. (((1,0) u))
 6. (((1,0) r))
 7. (((1,0) d))
 8. (((1,0) l))
 9. (((1,1) u))
10. (((1,1) r))
11. (((1,1) d))
12. (((1,1) l))
```



The five-node world

Now lets think about how to choose the correct action for a particular state-set. If the state-set contains only one state, the robot knows its location exactly. The choice of actions is simple move the robot toward the goal node. If there are multiple states in the state-set, then the idea is to choose an action which will allow the robot to eliminate some of the states the next time it calls `sees`. Eventually, all states but one will be eliminated.

Now, you *could* write an optimal get-action, but we aren't that sadistic. Just write a decent one. Once again: your get-action does *NOT* need to be optimal for this assignment. That may take you too long.

Appendix B: Theoretical Aside for the Incurably Curious

There is a side-effect of only including reachable state-sets in the table of actions: what happens if the robot somehow believes that its state-set is one of those not in the table? This could happen if we were to provide your robot with extra information. For instance, if we specify that the robot starts out facing north, the initial state set is not in the simplified table of reachable state-sets. Another way this can happen is if your `what-do-i-see` errs, resulting in an incorrect state set.

Handling these situations in the general case is hard. To be optimal for all possible state-sets, you would have to think through the right actions for each and every such state set. However, note that you can simply choose a table entry that corresponds to a superset of the actual state set. The action prescribed by this entry will certainly work, since it works on a superset (i.e. more uncertain case).

Of course, if the impossible state set was the result of a `what-do-i-see` error, then it's actually wrong! In this case, the robot would have to actually modify the state-set, growing it to encompass the real-world state. In effect, when faced with such an impossibility, the robot forgets some of the information that it has previously acquired. Another solution would be to just completely start over, setting the state-set to all possible states again. This is especially useful if the state-set becomes empty after a call to sees!

Hint: vectors in Java are cool things! We described them in the intro to Java handout given out at the beginning of class.