

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.081—Introduction to EECS I
Spring Semester, 2007
Work for Week 11

Issued: Tuesday, April 24

This handout contains:

- Software lab for Tuesday, April 24
- Pre-lab problems due April 26
- Robot lab for Thursday, April 26
- Post-lab due Tuesday, May 1

Planning

So far, our robots have always chosen actions based on a relatively short-term or “myopic” view of what was happening. They haven’t explicitly considered the long-term effects of their actions. One way to select actions is to mentally simulate their consequences: you might plan your errands for the day by thinking through what would happen if you went to the bank after the store rather than before, for example. Rather than trying out a complex course of action in the real world, we can think about it and, as Karl Popper said, “let our hypotheses die in our stead.” We can use state-space search as a formal model for planning courses of action, by considering different paths through state space until we find one that’s satisfactory.

This week, we’ll assume that the robot can know exactly where it is in the world, and plan how to get from there to a goal. Generally speaking, this is not a very good assumption, and we’ll spend the next two weeks trying to see how to get the robot to estimate its position using a map. But this is a fine place to start our study of robot planning.

We will do one thing this week that doesn’t seem strictly necessary, but will be an important part of our software structure as we move forward: we are going to design our software so that the robot might, in fact, change its idea of where it is in the world as it is executing its plan to get to the goal. This is very likely to happen if it is using a map to localize itself (you’ve probably all had the experience of deciding you weren’t where you thought you were as you were navigating through a strange city). This week, the only way it can happen is if, in the simulator, a malicious person drags the robot to another part of the world as it is driving around.

There are still a lot of details to be ironed out before we get this all to work, which we’ll talk about later.

Tuesday Software Lab

Please do the following programming problems.

Experimenting with breadth-first search

The code file `search.py` contains the code for the search algorithms and the `numberTest` domain discussed in lecture. Load this into Python (not SoaR, just ordinary Python, in Idle or Emacs) so you can experiment with it.

Try the code: Generate some paths that produce designated numbers by a sequence of doubling, adding 1, subtracting 1, squaring, or negating. For example, show how to generate 99, starting at 1. As you try various numbers, take note of the number of steps in the search and the length of the remaining agenda. Try the search both with and without using dynamic programming.

Robot on an infinite grid: Consider a robot on an infinite grid, with the squares labeled (i, j) for all integers i and j . The robot can move one square north, south, east, or west. Create a modified version of `numberTest` that will plan a path for the robot from an initial square to a designated goal square. This requires only a small change to `numberTest`. In fact, the *only* thing you need to change is the definition of `successors`. Try finding paths from $(0, 0)$ to (n, n) for various small values of n , both with and without using dynamic programming.

Forbidden squares: Modify your program to also take a list of “forbidden” squares that the robot cannot move into. Name your procedure `gridTestForbidden`, and have it take four arguments: an initial square, a goal square, a list of forbidden squares, and a boolean that says whether or not to use a visited list. For example,

```
gridTestForbidden((0,0), (4,4), ((1,0),(0,1)), True)
```

should generate a path from $(0, 0)$ to $(4, 4)$ that does not go through either $(1, 0)$ or $(0, 1)$.

Knight’s moves: According to the rules of chess, a knight on a chessboard can move two squares vertically and one square horizontally, or two squares horizontally and one square vertically. Modify your robot program so that it finds a path of knight’s moves from a given initial square to a given goal square on an 8×8 chessboard. Make sure to check that the knight remains on the board at each step. Use your program to find a path that a knight could take to get from the lower left corner of the chessboard $(0, 0)$ to the upper right $(7, 7)$.

What to turn in: For each of last three problems, include the new procedure you defined, as well as demonstrations on a set of test cases that you think demonstrates your code is entirely correct. You will be graded on your selection of test cases, as well as on the correctness of your code.

Pre-Lab problems for Thursday April 26

In this week’s lab we’re going to use search algorithms to plan paths for the robot. The biggest question, as always, is how to take our formal model and map it onto the real world. We need to define a search problem, by specifying the state space, successor function, goal test, and initial state. The choices of the state space and successor function typically have to be made jointly: we need to pick a discrete set of states of the world and an accompanying set of actions that can reasonably reliably move between those states.

Here is one candidate state-space formulation:

states: Let the states be a set of squares in the x,y coordinate space of the robot. In this abstraction, the planner won't care about the orientation of the robot; it will think of the robot as moving from grid square to grid square without worrying about its heading. When we're moving from grid square to grid square, we'll think of it as moving from the center of one square to the next; and we'll know the real underlying coordinates of the centers of the squares.

actions: The robot's actions will be to move North, South, East, or West from the current grid square, by one square, unless such a move would take it to a square that isn't free (could possibly cause the robot to collide with an obstacle). The successor function returns a list of states that result from all actions that would not cause a collision.

goal test: The goal test can be any Boolean function on the location grids. This means that we can specify that the robot end up in a particular grid square, or any of a set of squares (somewhere in the top row, for instance). We cannot ask the robot to move to a particular x,y coordinate at a finer granularity than our grid, to stop with a particular orientation, or to finish with a full battery charge.

initial state: The initial state can be any single grid square.

The planning part of this is relatively straightforward. The harder part of making this work is building up the framework for executing the plans once we have them.

Software organization

The code for our basic system (contained in `ps11-code.zip` on the calendar web page) is contained in the following files:

- `XYBrain.py`
- `GridMap.py`
- `XYDriver.py`
- `XYGridPlanner.py`
- `Sequence.py`
- `utilities.py`
- `search.py`

We'll go through the relevant code in each of these files. Be sure you understand what it is doing. Answer all of the questions and bring your solutions to lab on Thursday at 2.

Basic structure

The `XYBrain` works roughly as follows. Note that it really only pays attention to the robot's x,y location, and ignores the orientation part of the pose, except in the lowest-level driving routine, because the model we're using for planning only includes the robot's location.

- It finds its current location (either by cheating in the simulator or believing its odometry to be exactly true, in the robot).
- It converts the current world location to grid coordinates and plans a path to a goal grid location.
- It chooses the second location on the path as a "way point" and tries to drive directly there.

- Once it reaches the waypoint, it plans again (just in case someone has kidnapped it¹), and drives toward the next waypoint.
- Once it is near the goal location, it quits driving (though it keeps rechecking its location, again, just in case of kidnapping).

We will use the mechanism of terminating behaviors to manage the sequencing of the “macro” actions of driving from one square to the next.

Try it out!

Try out the planner and see how it works.

- Start `SoaR`
- Pick **Simulator**, then **she.py**.
- Pick **Brain**, then **XYBrain.py**
- You’ll see a new window, with a picture of the environment, drawn as a grid of squares, with the occupied ones drawn in black and the free ones in white. Furthermore, it shows the robot’s current plan. The green square is centered on the robot’s actual current pose. The gold square is the goal, and the dark blue squares show the rest of the steps in the plan.
- Click start. The robot will drive through the world, with the plan redrawing each time a subgoal is achieved, until it gets to the goal.
- Drag the robot somewhere else. See what happens.

Question 1. The system is using breadth-first search, with a visited list. Explain what the start and goal states are for the very first planning problem the system has to solve. What is the depth of the search tree when it finds a solution to that problem? What is the branching factor? Roughly (just get the order of magnitude right) how many nodes will be visited to find a solution with the visited list turned on? How about with it off? Explain why.

Now, we’ll go through all of the code, some of it only at the level of what procedures are available to you, and some of it in complete detail. The classes and methods you’ll need to concentrate most on are:

- `XYGridPlanner.py`
 - `makePlan(self, init, goal)`
- `XYDriver.py`
 - `XYDriver.step()`
 - `XYDriver.done()`
 - `XYDriverOA.done()`
- `XYBrain.py`
 - `setup()`

GridMap.py

We’re not going to discuss this code in much detail. We’ll just cover enough so you know what it does and what methods you have available to use in your own code. This file contains two classes, which are used to represent the obstacles in the world.

¹There is, believe it or not, a whole technical literature on the “kidnapped robot” problem, in which the the robot is moved to some completely unpredictable location from where it currently is. It will be even more interesting to deal with that problem two weeks from now.

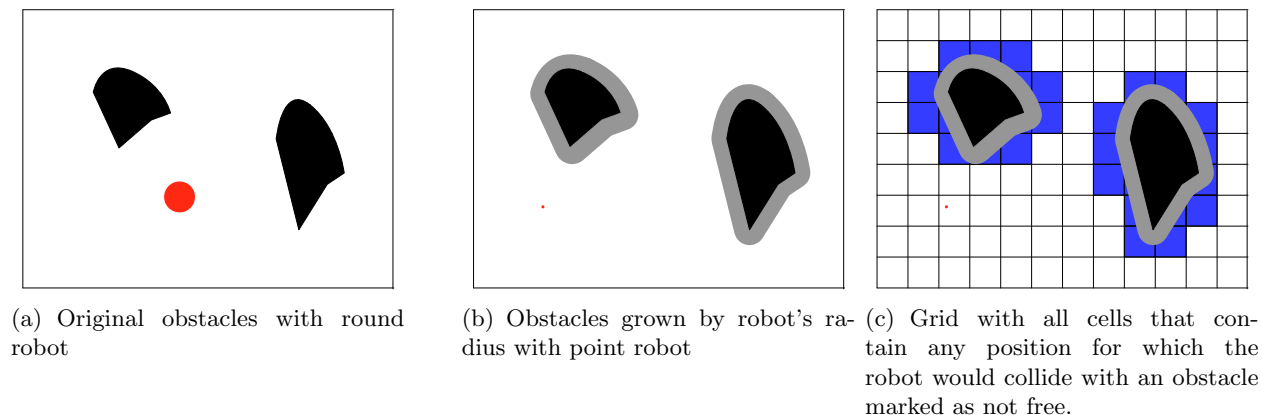


Figure 1: Construction of a grid map in two dimensions for a round robot

A **Map** object just stores a set of “boxes” that describe where the obstacles in the world are. An obstacle is represented by the coordinates of two diagonal corners.

The **GridMap** class represents the space of legal poses of the robot. A pose is described by the robot’s x , y , and θ (orientation) coordinates. A pose is legal if the robot can be in that pose and not collide with any of the obstacles. We will treat the robot as if it is round, with radius `robotWidth`, which simplifies matters, but makes our planning overly conservative. If your robot is round, a simple way of dealing with obstacles is to “grow” the obstacles by the radius of the robot, and then think of the robot as a point, moving through the space of fatter obstacles.

We will make a three-dimensional (the dimensions are x , y , and θ) grid of robot poses, and mark them as free (meaning the robot can be in that pose without running into anything) or occupied. We are conservative in this: we only say a grid cell is free if every pose contained in that cell will not cause a collision with any obstacle.

Figure 1(a) shows a simple world with two obstacles and a round robot. Growing the extent of each obstacle by the radius of the robot gives us the picture in figure 1(b), in which we can now think of the robot as a point, and any place in the x, y plane where it is not overlapping with the grown obstacles as being free. Finally, in figure 1(c) we show a grid superimposed on the world, and any cell that contains a position for which the robot would be in collision with an obstacle marked as not being free.

The initializer takes as input an object of the **Map** class, which specifies where the obstacles are, the minima and maxima of the x and y coordinates, and a granularity, `gridN`. The grid will have `gridN` cells in each dimension. There is no need to specify the range of θ , because it goes between 0 and 2π .

```
1 class GridMap:
2     robotWidth = 0.2
3     def __init__(self, theMap, xmin, xmax, ymin, ymax, gridN):
```

Here are the most important methods of the **GridMap** class. See the code file for the definitions of these, and a bunch of other useful methods, if you’re interested. Much of the work here is in translating between poses in real world coordinates, and those in “index” coordinates. Indices map into the stored array. So, for instance, if our x coordinate went from -10 to +9, and `gridN` were 40, then index 20 would be associated with x coordinate of 0.

```

4      # Is the cell with indices i,j free?
5      def free(self, (i,j)):
6      # Convert (x,y) world coordinates to (i,j) indices
7      def poseToIndices2(self, (x,y)):
8      # Convert (i,j) indices to (x, y) world coordinates
9      def indicesToPose2(self, (ix,iy)):
10     # Draw the grid map in a window, with occupied squares black and
11     # free squares white
12     def draw(self, window):

```

In addition, the file contains definitions of lists of boxes corresponding to the obstacles in many of the worlds in the simulator.

XYGridPlanner.py

This class is initialized with a `GridMap` and then can be used to make and visualize plans within that map.

```

1  class XYGridPlanner:
2      def __init__(self, gridMap):
3          self.gridMap = gridMap

```

The `makePlan` method that makes a new plan, given an initial state and a goal state, each of which is a length-two tuple of indices into the grid. It defines a successor function that moves to any of the four adjacent locations that are free. It either returns a list of grid locations, starting with `init` and ending with `goal`, or `None` if the goal is unreachable from the initial state.

```

4      # Given init (ix, iy) and goal (gx, gy) in XY grid coordinates
5      def makePlan(self, init, goal):
6          # Successor function
7          def s((i,j)):
8              # Robot can move north, south, east, or west, but only if that
9              # square is free
10             return [s for s in ((i-1,j),(i,j-1),(i,j+1), \
11                                 (i+1,j)) if self.gridMap.free(s)]
12
13     # Goal test
14     def g(state):
15         return state == goal
16     result = search.breadthFirstDP(init, g, s)
17     if result:
18         print "Path:", result
19         return result
20     else:
21         print "Couldn't get from ", init, " to ", goal
22         return None

```

Question 2. How would you change `makePlan` to allow the robot to make moves to its diagonal neighbors? You're not required to implement this, but you might go ahead and give it a try.

XYDriver.py

The `XYDriver` is in charge of making the robot drive from one waypoint to the next, assuming that the space between them is free. This is the trickiest part of the code, but it should be a fairly comprehensible application of our terminating behavior idea from week 4. The basic idea is that a behavior specifies three methods: `start`, `step` and `done`. Typically, a behavior will be initialized by calling its `start` method; then the `step` method will be called repeatedly until the `done` method returns `True`. You might want to go back and review that lab if this short review doesn't remind you sufficiently.

We start by defining a basic class `XYDriver`, which is a terminating behavior, in the sense that it supplies `start`, `step`, and `done` methods. This behavior is given a goal location in world coordinates, and will try to drive straight from its current location to that location.

It's important to pay attention to the arguments to the initializer; note that `poseFun` and `motorOutput` are functions that the behavior can call if it needs them.

- `goalXY`: a goal for the robot in *world* (x, y) coordinates (not indices)
- `poseFun`: a function that can be called to determine the robot's current pose; it will return (x, y, θ)
- `motorOutput`: a function that can be called to set the robot's motor velocities; takes two arguments, forward and rotational velocities
- `distEps`: a distance specifying how close the robot should come, in global coordinates, to the goal, before terminating.

```

1 class XYDriver:
2     forwardGain = 1.5
3     rotationGain = 1.5
4     angleEps = 0.05
5
6     def __init__(self, goalXY, poseFun, motorOutput, distEps):
7         self.goalXY = goalXY
8         self.poseFun = poseFun
9         self.motorOutput = motorOutput
10        self.distEps = distEps

```

There is no special work that needs to be done in the `start` method.

```

11     def start(self):
12         pass

```

Here is where the action is. If the robot is facing toward the goal, then it moves forward with a speed proportional to the distance to the goal. If it is not facing toward the goal, it rotates toward the goal heading with a rotational speed proportional to the angular distance to the goal heading. We make use of a number of procedures that do simple geometry on angles and distances, which are defined in `utilities.py`.

```

13     def step(self):
14         # These assignment statements just give convenient names to
15         # the components of the current pose and goal
16         (px, py, ptheta) = self.poseFun()
17         (gx, gy) = self.goalXY
18         # The direction from the robot's current location to the desired
19         # location

```

```

20     headingTheta = angleToPoint((px,py),(gx,gy))
21     # The difference between the desired and actual headings
22     headingDeltaTheta = fixAnglePlusMinusPi(headingTheta - ptheta)
23
24     if nearAngle(ptheta, headingTheta, self.angleEps):
25         # The heading is close to correct, so we should drive forward
26         r = math.sqrt((px - gx)**2 + (py - gy)**2)
27         # Move at a speed proportional to the distance to the goal
28         self.motorOutput(r * self.forwardGain, 0.0)
29     else:
30         # We need to adjust the heading before moving. Rotate at a
31         # speed proportional to the heading error
32         self.motorOutput(0.0, headingDeltaTheta * self.rotationGain)
33
34     # We don't care what our orientation is as long as our (x,y)
35     # coordinates are close
36     def done(self):
37         currentPose = self.poseFun()
38         return nearPose2(self.goalXY, currentPose[:-1], self.distEps)

```

Question 3. Given this XYDriver, why do we need the planner? What would happen if we just asked it to drive directly to the final goal location?

Question 4. What might the consequences be of having too large a value for distEps?

The basic XYDriver behavior has one major flaw: if something is in its way, the robot will run right into it. Here, we define a new class, XYDriverOA, (the “OA” stands for *obstacle avoidance*), which looks at the sonar readings and terminates the behavior if any reading is shorter than some threshold. It is a subclass of XYDriver, so it just overrides the methods it needs to change.

The `__init__` method takes an additional parameter, `sonarDistanceFun`, which, when called, returns a list of the current sonar readings. It stores that function, and then calls the `__init__` method of the parent class.

```

39 class XYDriverOA (XYDriver):
40     def __init__(self, goalXY, poseFun, motorOutput, sonarDistanceFun, distEps):
41         self.sonarDistanceFun = sonarDistanceFun
42         self.blockedThreshold = 0.15
43         XYDriver.__init__(self, goalXY, poseFun, motorOutput, distEps)

```

Finally, the `done` method returns `True` if either the parent `done` method returns `True`, or if one of the sonar readings is too short.

```

44     def done(self):
45         minDist = min(self.sonarDistanceFun())
46         return XYDriver.done(self) or minDist < self.blockedThreshold

```

XYBrain.py

We’ve defined a basic terminating behavior that drives to a location. Now, we need a way of using the planner to decide what locations to drive to. The control structure we want to have, at the top level, is that the planner is called to make a plan, it executes the first step (which is to use

XYDriver to drive toward the first waypoint in the plan, and either terminate because it arrives very near the waypoint or because an obstacle was encountered), and then the current location is updated and the planner is called again. This is kind of like a sequence of terminating behaviors, but the problem is that we don't know exactly what sequence of behaviors we want to execute in advance. So, we're going to define a new way of making a terminating behavior, which is defined as a *stream* of other terminating behaviors. A stream is different from a list, or other fixed sequence, because instead of having the whole thing determined in advance, we have a *generator* function that we can call to get the next element of the stream when we need it. The details of how this work are in the section on the **TBStream** class.

Let's look at the code for the brain, to see how it all fits together.

All the real work is here, in the setup method of the brain. We start by making a **GridMap** (be sure to use the set of boxes that agrees with the simulated world you're using) and a drawing window, and asking the map to draw itself into the window. We specify a goal pose for the robot (in world coordinates), and compute the (x, y) grid indices associated with that pose. We also set a persistent variable, **robot.lastDrawnPlan** that we can use to avoid drawing too much. Finally, we make an instance of the **XYGridPlanner** class, which is set up to use this map.

```

1 def setup():
2     map = GridMap(Map(sheBoxes), 0.0, 4.0, 0.0, 4.0, 30)
3     boxDim = 4.0/30.0
4     # Make a 300 x 300 pixel window, with coordinates ranging from
5     # -0.5 to 4.5 in each direction
6     w = DrawingWindow(300, 300, -0.5, 4.5, -0.5, 4.5, "SHE World")
7     map.draw(w)
8     goalPose = (3.5, 0.5, 0.0)
9     goalIndices = map.poseToIndices2(goalPose[:-1])
10    robot.lastDrawnPlan = None
11    planner = XYGridPlanner(map)

```

Now, we define the procedure that will generate a new terminating behavior each time it is called. It will be used as the generator for the stream of behaviors that the robot executes. Note that this procedure definition is still *inside* the **setup** procedure.

When it's time to replan, we start by getting the current actual pose of the robot, and converting it into grid indices. Now, we ask the planner to make a plan from the current indices to the goal indices. If it returns a plan and it's different from the last plan we drew, we draw the plan into the window. If it returns a plan of length greater than 1 (meaning that it actually has a step in it that needs to be done), then we convert the indices of the first waypoint into world coordinates (the conversion gives the world coordinates of the point at the center of the grid square), and then we make a new instance of the **XYDriverOA** behavior with the goal of driving to the waypoint. If either there is no plan, or it's of length one, we just return the **Stop** behavior.

```

12    def dynamicReplanner():
13        currentPose = cheatPose()
14        # Get grid indices for robot's x,y location
15        currentIndices = map.poseToIndices2(currentPose[:-1])
16        # Make a plan and draw it into the window
17        plan = planner.makePlan(currentIndices, goalIndices)
18        if plan and (plan != robot.lastDrawnPlan):
19            planner.drawPlan(plan, w)
20            robot.lastDrawnPlan = plan

```

```

21         if plan and len(plan) > 1:
22             subgoal = map.indicesToPose2(plan[1])
23             return XYDriver0A(subgoal, cheatPose, motorOutput, sonarDistances,
24                               boxDim/3.0)
25         else:
26             return Stop(motorOutput)

```

We're still in the `setup` procedure. So, now, having defined the generator function for our stream, we create the behavior stream, and ask it to start.

```

27     robot.driver = TBStream(dynamicReplanner)
28     robot.driver.start()

```

It's smooth sailing from here.

```

29 def step():
30     robot.driver.step()

```

Question 5. Run the simulation again, this time picking a different (reasonable) goal for the robot, and change `XYBrain.py` to go there. (You need to change the line that says `goalPose = (3.5, 0.5, 0.0)`.) Remember that whenever you change the code, you have to reload the brain.

Sequence.py

If this doesn't make detailed sense to you, don't worry too much about it.

To make everything work, we need to define a new class, `TBStream`. It is, itself, a terminating behavior. To initialize, we pass in `actionGenerator`, which is a function of no arguments that, when called, will produce the next terminating behavior to be executed.

```

1 class TBStream:
2     def __init__(self, actionGenerator):
3         self.actionGenerator = actionGenerator

```

To start the stream of behaviors, we call the `actionGenerator` to get a new terminating behavior, and then ask that behavior to start.

```

4     def start(self):
5         self.currentAction = self.actionGenerator()
6         self.currentAction.start()

```

To take a step of the stream of behaviors, we ask the currently executing behavior to take a step. Then, if that behavior is done, we call our own `start` method again, which will ask the generator for a new behavior and start it.

```

7     def step(self):
8         self.currentAction.step()
9         if self.currentAction.done():
10            self.start()

```

A stream is neverending.

```

11     def done(self):
12         return False

```

We'll also need this trivial little behavior, which makes the robot stop for one step. The initializer needs to be give a procedure that can be used to send commands to the motors.

```

13 class Stop:
14     def __init__(self, motorOutput):
15         self.motorOutput = motorOutput
16     def start(self):
17         pass
18     def step(self):
19         self.motorOutput(0.0, 0.0)
20     def done(self):
21         return True

```

What to turn in

Bring your written answers to the questions above to class.

Robot Lab for Thursday April 26

Depth first search

Currently, the `XYGridPlanner` uses breadth-first search. Change it to use depth-first search. What happens? Why?

Checkpoint: 2:20 PM

- Explain why depth-first search causes the robot to do what it does.

A more general goal condition

Currently, `XYBrain` is set up so that there's a single goal pose for the robot. Change the code so that it can accept a more general goal condition, such as a set of poses, or even a set of ranges on the x, y coordinates.

You might find it useful to use the Python `in` construct. The expression `x in S` returns `True` if `x` is a member of the tuple `S`. Experiment a bit with it to see how it works.

Checkpoint: 3:00 PM

- Show that you can change the goal from a single state to a set of states, so that the robot would be satisfied reaching any of the states in the set. Demonstrate it in simulation.

Finally, let's try running a planner brain on the real robot. If you use `XYRobotBrain.py`, it will ask you for the robot's initial pose when the brain starts. Then, when we call `cheatPose` in the code, it will use the robot's odometry to update the pose, assuming that it started where you said it did. This brain is set up to use a single goal; it might be best to download a fresh version of `XYGridPlanner.py` that deals with single goals when you run this.

We have set up some playpens that correspond to Shannon's world in the simulator. To make your planner work in that world, replace the reference to `sheBoxes` to `shannonBoxes`. You might want to try this in the simulator first, just to see what it looks like.

Run the robot around in this world.

Checkpoint: 3:30 PM

- Discuss ways in which behavior in the simulator differs from behavior in the real robot.

Changing formulation

In the current abstraction of the state space, the robot has no way to take its current heading or the time it spends rotating into account when it makes a path. If we want to do that, we have to reformulate the state and action spaces. Let's consider a new formulation of the state space, to include the current grid coordinates, as well as the robot's heading, but with the heading discretized into the 4 compass directions.

Question 6. If the x and y coordinates are discretized into 20 values each, and we have 4 possible headings, what is the size of the whole state space?

Along with changing the state space, we need to change the action space. In this view, we'll have three possible actions: **move**, which moves one square forward in the direction the robot is currently facing; **left**, which rotates the robot (approximately) 90 degrees to the left, and **right**, which rotates the robot (approximately) 90 degrees to the right. In fact, the **left** and **right** actions try to rotate the robot to line up to one of the compass directions, even if that requires rotating somewhat more or less than 90 degrees. Ultimately, these actions will be executed by `XYThDriver`, which can accept any new goal pose in world coordinates.

You can experiment with this model in the simulator. Load the brain `XYThBrain.py` in the simulator and try running it again from the default start position. How does it behave?

Go to the file `XYThGridPlannerX.py`. You'll find that the definition for the method `makePlan` is missing. It takes an initial state and a goal state, each of which are of the form (ix, iy, h) , where ix and iy are grid indices, and h is in $\{0, 1, 2, 3\}$, where 0 stands for **north**, 1 stands for **east**, 2 stands for **south**, and 3 stands for **west**. The class variable `self.gridMap` is an object that has a method `free`, which when applied to an (x, y) pair of indices, returns `True` or `False`. Write this procedure. It should contain a call to our old friend `search.breadthFirstDP`. The `XYThBrain` will take care of executing the plan you make. Now, save your file out as `XYThGridPlanner.py`, reload the brain, and try it out.

Compare the branching factor and depth of planning in this formulation to the previous one. Would you expect the planner to run faster or slower, in general? Which one do you think would give you better plans? What would be the result if we used a more fine-grained discretization of the heading? Can you think of another way to formulate this problem?

Checkpoint: 4:45 PM

- Demonstrate your planner in this new space.
- Discuss the advantages and disadvantages of this formulation over the previous one, in terms of depth and branching factor; and speculate about another formulation.

Optional Explorations**Move diagonally**

Change one of the planners so that the robot can move diagonally. Before you start, think about what you'd need to do for the first planner (where the states don't include orientation) and then for the second one. Then pick one and implement it.

Update the map

Change the system so that if you encounter an obstacle in a square that should be free, you change the map. Try starting with an empty map, and see if you can build a good map of the world.

Non-uniform costs

Now, what if we found that the floor was slippery in the upper part of the world? We might want to penalize paths that went through the upper locations relative to those that go through the lower ones.

Change the code to model that. You will have to:

- Add a new *uniform cost* search procedure to `search.py` that stores the path cost so far in a search node and always takes the cheapest node out of the agenda. (Ask lpk for help if you want to work on this).
- Add a way of specifying the costs of moving from state to state; if our costs are just because of the conditions of the states themselves, then it might be reasonable to add the cost information to the map, and put a procedure in `GridModule.py` that can be queried to get the cost of being in a state.

Post-Lab

There are some tutor problems to cement the ideas of how the different search algorithms work. Please do those, as well as writing up a clear descriptions of your approaches and solutions to problems posed during the lab. Include a description of code you wrote or changed, but please do not hand in long code files in which you've just changed a few characters.

Concepts covered in this assignment

Here are the important points covered in this assignment:

- The abstract notion of searching in a finite space can be applied to a real-world robot problem; the hardest part is the formulation.
- Different problem formulations can yield different running times and solution qualities.
- Practice with applying search algorithms.