MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Fall Semester, 2007

Assignment 11, Issued: Tuesday, Nov. 13

To do this week

...in Tuesday software lab

1. Start writing code and test cases for the numbered questions (1–6) in the software lab. Paste all your code, including your test cases, into the box provided in the "Software Lab" problem on the on-line Tutor. What you submit to the tutor will not be graded; what you hand in next Tuesday will.

...before the start of lab on Thursday

1. Read the lecture notes.
2. Do the Pre-Lab homework problems for week 11 that are due on Thursday (Questions 7-13).
3. Read through the entire description of Thursday's lab.

...in Thursday robot lab

1. Do the nanoquiz at the start of the lab. It will be based on the material in the lecture notes and the homework due on Thursday.
2. Answer the numbered questions (Questions 14–25) in the robot lab and demonstrate the appropriate ones to your LA.

...before the start of lecture next Tuesday

1. Do the lab writeup, providing written answers (including code and test cases) for **every** numbered question in this handout.

On Athena machines make sure you do:
`athrun 6.01 update`
so that you can get the `Desktop/6.01/lab11` directory which has the files mentioned in this handout.

- You need the file `search.py` for the software lab.

During software lab, if you are using your own laptop, download the file(s) from the course Web site (on the Calendar page).

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 (at first) 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. However, we will see later in this lab that the robot can in fact be in a similar situation if it fails to predict the effect of its actions accurately.

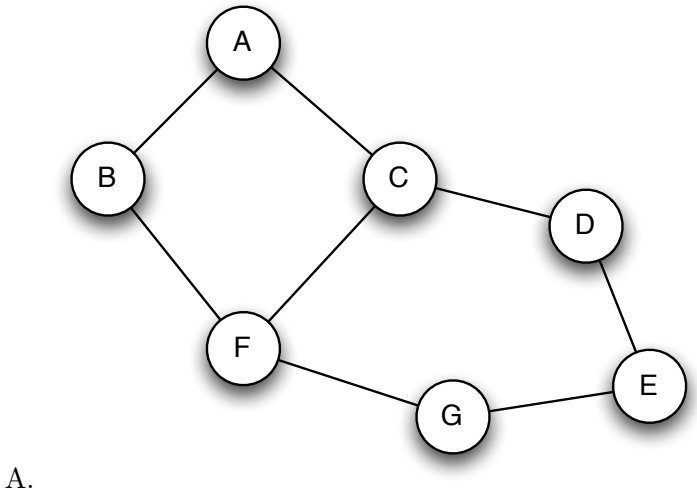
There are still a lot of details to be ironed out before we get this all to work, which we'll talk about later.

Tuesday Software Lab

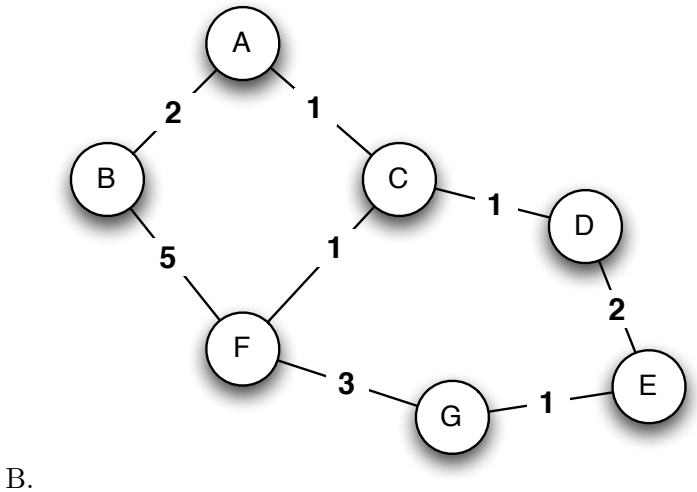
Please do the following programming problems.

Experimenting with 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) so you can experiment with it.



A.



B.

Figure 1: A. A simple map. B. A map with each road labeled by the time it takes to traverse it.

Question 1. Consider the graph in figure 1A. Imagine it's a road network between cities with short names. Formulate a successor function for this domain. Problems like this don't have a natural action space, because different states have different numbers of potential actions (roads that can be followed). If you name the roads (use numbers as the names), then the names of the roads emanating from a city can serve as the available actions there. Formulate a goal test function.

Use your successor and goal functions and the `search` function to find the path from A to D using each of our four search methods (breadth and depth search, with and without dynamic programming). Say something interesting about the results. (Your results will vary depending on the order in which you create the successors, if this example does not produce different paths for the different methods, pick a start and goal that do).

Question 2. How big is the search space for this problem, with and without using dynamic programming? That is, how big can the search trees possibly get?

Question 3. Now, look at the graph in figure 1B. It's the same road network, but now the arcs are labeled with an amount of time it takes to traverse them. We'd like to make plans to traverse this network, but with the goal of arriving at a particular node at a particular time, for example, reaching node F exactly at time 8, assuming we started at node A at time 0. What is an appropriate state space for this planning problem?

Question 4. Write down a successor function for this problem. Implement it. Run different kinds of searches on it. Report the results.

Question 5. How big is the search space for this problem, with an without using dynamic programming? That is, how big can the search trees possibly get?

Question 6. What happens when you ask for a goal that is impossible, for example, leaving node A at time 0 and arriving at node G at time 4? Is it possible to arrive at F (from A) exactly at time 3? How about exactly at time 4? How does the code deal with these cases?

Go to the on-line Tutor at `http://sicp.csail.mit.edu/6.01/fall07`, choose PS11, and paste the code that you wrote during lab, including your test cases, into the box provided in the "Software Lab" problem. Do this even if you have not finished everything. Your completed answers to these questions are to be handed in with the rest of your writeup for Tuesday, November 10.

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

Pre-Lab Homework problems - hand in on Thursday Nov. 15

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 represent the real world problem in our formal model. 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 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 (that is, that 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 is contained in the following files:

- `GridBrain.py`
- `GridWorld.py`
- `XYDriver.py`
- `Planner.py`
- `Sequence.py`
- `SM.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. You should answer all of the questions below and bring your solutions to lab on Thursday.

Basic structure

The `GridBrain` works roughly as follows:

- 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 it[1]), 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).

Note that the `GridBrain` 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.

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** (which specifies the "She World" for the simulator).
- Pick **Brain**, then `GridBrain.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 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.
- Click stop. Drag the robot somewhere else. Click start again. See what happens.

Question 7. The system is using breadth-first search, with dynamic programming. 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 typical branching factor? Approximately (just get the order of magnitude right) how many nodes will be visited to find a solution with dynamic programming turned on? (You can estimate it from the depth and branching factor or by having the search program print it out.) How about with dynamic programming 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.

[1] There 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.

SM.py

We have been modeling our worlds by describing the states of the world and the transition between these states induced by certain actions. This is precisely the State Machine abstraction that we looked at earlier in the term. We will be using a slightly extended version of the FSM class we saw earlier.

A primitive state machine, `PrimitiveSM`, is created by specifying:

- a name for the machine,
- a transition function that takes a state and an input (an action) and returns a new state,
- an output function which specifies the output for a state (this week, we assume that the output function simply returns the state),
- an initial state,
- a termination (goal test) function, and
- a list of the possible inputs (actions).

So, a `PrimitiveSM` instance has everything we need to plan in some domain; it is a **model** of the domain.

There's no need to use SoaR to do this question

Question 8. Create an instance of a `PrimitiveSM` that encodes the "city" problem in figure 1A. Use 0, 1, 2 as the names of the actions that take you from one city to another. If a city has fewer than 3 neighbors, have the extra actions return `None`. Leave the termination function unspecified (i.e. with value `None`). Show the machine going through the sequence of actions needed to go from state `'A'` to state `'G'` using calls to the machine's `step` method.

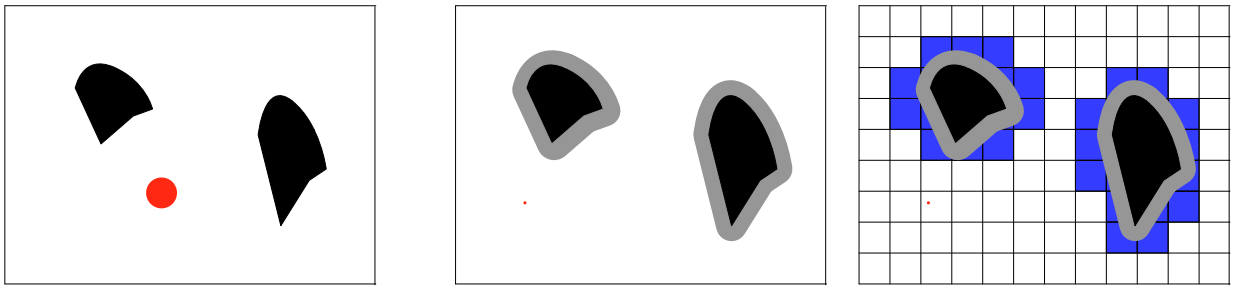
GridWorld.py

This file defines two classes. The `Map` class is used to specify the locations of obstacles in the world, and the `GridWorld` class specifies a state-machine model of a robot moving on a grid corresponding to a map.

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 `D2GridWorldFromMap` class is a subclass of `PrimitiveSM` (a primitive state machine) representing the space of legal poses of the robot and motions between them. A pose is described by the robot's x and y coordinates. A pose is free if the robot can be in that pose and not collide with any obstacle. 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. Convince yourself that this is true.)

We will make a two-dimensional 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



(a) Original obstacles with round robot

(b) Obstacles grown by robot's original radius with robot of zero radius.

(c) Grid with cells marked occupied if they contain any position for which the robot would collide with an obstacle.

Figure 2: Construction of a grid map in two dimensions for a round robot

say a grid cell is free only if every pose contained in that cell will not cause a collision with any obstacle.

Figure 2(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 2(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 2(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 occupied.

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.

```
class D2GridWorldFromMap (D2GridWorld):
    robotWidth = 0.2
    def __init__(self, theMap, xmin, xmax, ymin, ymax, gridN):
```

Here are the most important methods of the `GridWorld` 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 +10, and `gridN` were 40, then index 20 would be associated with `x` coordinate of 0.

```
    # specify the possible inputs to the robot in the grid;  these are
    # action specifications that move it west, north, east, south, or stay
    # put.
    inputs = [(-1, 0), (+1, 0), (0, +1), (0, -1), (0,0)]

    # Is the cell with indices i,j free?  Returns a boolean
    def free(self, (i,j)):

    # Specify the state transition function.  Takes a state which is a pair of
    # indices and an action, which is also a pair of coordinates, and returns
    # the resulting state, which is the pair of those coordinates if that
    # resulting state is free and returns None otherwise
```

```
def transitionFn((ix, iy), (dx, dy)):
    newix = ix + dx
    newiy = iy + dy
    if self.free((newix, newiy)):
        return (newix, newiy)
    else:
        return None

# Convert (x,y) world coordinates to (i,j) indices
def poseToIndices(self, (x,y)):
# Convert (i,j) indices to (x, y) world coordinates
def indicesToPose(self, (ix,iy)):
# Draw the grid map in a window, with occupied squares black and
# free squares white
def drawWorld(self, window):
# Draw the robot in a location on the grid map
def drawState(self, indices, window, color):
```

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

Planner.py

We have given you a file called `PlannerShell.py` for you to edit. When you fill it in, rename it to `Planner.py` to test. We have also given `Planner.pyc`, a compiled version of a complete `Planner.py`. If you overwrite it and want to get it back, there is a Pyc folder in the `lab11` folder that has the .pyc files for the original distribution.

A `Planner` object can make plans to move through the states of a state machine. We will use it to make plans for the robot to move through the grid map. It is initialized with a state machine, `sm`:

```
def __init__(self, sm):
    self.sm = sm
```

The primary method, `plan`, will make a plan. You can specify an initial state and a goal test function, as for the search methods we've seen. The `depthFirst` argument does not have to be specified: we have said that if no value is passed in for it, then it should be given the default value of `False`, which means we'll do breadth-first search. The only work that has to be done is to construct a successor function for the planner using the transition function of the state machine.

```
def plan(self, initialState, goalTest, depthFirst = False):
    def successors(s):
       # your code here
    result = searchDP(initialState, goalTest, successors, depthFirst)
    if result:
        print "Path:", result
        return result
    else:
        print "Couldn't get from ", initialState, " to goal"
        return None
```

Question 9. Create an instance of `Planner` using the state machine that represents the city graph in figure 1A. Use this instance to plan a path from state `'A'` to state `'G'`. Use the `Planner.pyc` file that we have given you, and test this in Idle, not SoaR.

Question 10. Recalling that `successors` takes a state as an argument and returns a list of `(action, state)` pairs, that `self.sm.inputs` is a list of the possible actions that can be generated as inputs to this state machine, and that `self.sm.transitionFunction` takes a state and an action as arguments and returns the resulting next state, fill in the definition of the `successors` function. It only needs a single line of code. Test this, first, on your city state machine, from Idle.

Question 11. Now, test your successor function by calling the planner on an instance of `D2GridWorldFromMap`, and be sure you're getting correct results. You can use `SheWorld` as the map; it is 4 x 4 meters square.

The simplest way to do this is to use SoaR, and define a brain that does all of its work (e.g., making an instance of `D2GridWorldFromMap`, making an instance of `Planner`, and then asking the planner to make a plan and display it) in the `setup` method. (The file `GridBrain.py` does this and more, but you can see examples of how to make the necessary calls there. Do not use `cheatpose` to specify the current pose, just type the indices for the initial grid in the file. You can try alternate initial grids by changing this entry in the file.).

Question 12. Change `GridWorld.py` to allow the robot to move to its diagonal neighbors. Find start and goal indices for the `SheWorld` that allow the planner to find a shorter plan using diagonal actions. Use SoaR.

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 to determine the robot's current pose; it will return $(\mathtt{x}, \mathtt{y}, \theta)$
- `motorOutput`: a function 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.

```
class XYDriver:
    forwardGain = 1.5
    rotationGain = 1.5
    angleEps = 0.05

    def __init__(self, goalXY, poseFun, motorOutput, distEps):
        self.goalXY = goalXY
        self.poseFun = poseFun
        self.motorOutput = motorOutput
        self.distEps = distEps
```

The behavior first rotates until it is aligned towards the goal location (within `angleEps`) and then move straight towards the goal coordinate. The behavior uses the gains to choose velocities as a function of the current distance from the goal. That is, it uses a simple feedback system of the type we have seen before.

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, then calls the `__init__` method of the parent class.

```
class XYDriverOA (XYDriver):
    def __init__(self, goalXY, poseFun, motorOutput, sonarDistanceFun, distEps):
        self.sonarDistanceFun = sonarDistanceFun
        self.blockedThreshold = 0.15
        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.

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

GridBrain.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 terminate either 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. You can see the details by looking at the `TBStream` class in `Sequence.py`.

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 `D2GridWorldFromMap` (be sure to use the set of boxes that agrees with the simulated world you're using) and a drawing window. Finally, we make an instance of the `Planner` class, which is set up to use this grid world.

```
def setup():
    (xmin, xmax, ymin, ymax) = (0.0, 4.0, 0.0, 4.0)
    gridN = 20
    gw = D2GridWorldFromMap(Map(sheBoxes), xmin, xmax, ymin, ymax, gridN)
    gwPlanner = Planner(gw)
    w = DrawingWindow(400, 400, xmin-0.5, xmax+0.5, ymin-0.5, ymax+0.5, "Plan")
```

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 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 there is no plan, or a plan exists but has length one, we just return the `Stop` behavior.

```
    def dynamicReplanner():
        # Get current robot pose
        currentPose = cheatPose()
        # Get grid indices for robot's x,y location
        currentIndices = gw.poseToIndices(currentPose[:-1])
        # Make a plan and draw it into the window
        plan = gwPlanner.plan(initialState = currentIndices,
                              goalTest = lambda x: x == (17, 3))

        if plan and len(plan) > 1:
            gwPlanner.drawPlan(plan, w)
            # Return a terminating behavior to take the first action
            # plan[1] is the first real step,  [0] is the action part
            motion = gw.indicesToPose(plan[1][0])
            return XYDriverOA(motion, cheatPose, motorOutput,
                              sonarDistances, (xmax - xmin) / (5 * gridN))
        else:
            return Stop(motorOutput)
```

(The expression `(xmax - xmin) / (5 * gridN)` specifies a distance tolerance of one fifth of a grid cell, which is used for deciding whether the robot is close enough to its local goal.)

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.

```
    robot.driver = TBStream(dynamicReplanner)
    robot.driver.start()
```

It's smooth sailing from here.

```
def step():
    robot.driver.step()
```

You will need to use SoaR to do this question

Question 13. Run the simulation again, this time picking a different (reasonable) goal for the robot, and change `GridBrain.py` to go there. (You need to change the goal test in the call to the planner.) Remember that whenever you change the code, you have to reload the brain.

What to turn in

Bring your written answers to the questions above and have them checked by a TA in class; then hand them in at the end of Lab.

Robot Lab for Thursday Nov. 15

Catching the bus!

This example requires a new version of SoaR that supports moving objects in simulator worlds; this will be propagated to our Lab laptops when they are started and Athena has an updated distribution. But, if you want to work on this on your own computer, you will need to re-install SoaR using the new distribution on our Web site.

Consider the problem of meeting up with another robot that is following a predetermined path with a known schedule: think of it as trying to catch a bus.

We want to construct a plan that will get us to be at some grid coordinates at the same time as the bus will be there. If we want to do that, we have to reformulate the state and action spaces that we have been using. Let's consider a new formulation of the state space, to include our robot's current grid coordinates as well as the current time.

Question 14. Do we need to change the actions to solve this problem? Why or why not? If yes, indicate what actions are needed.

Question 15. You can experiment with this model in the simulator. Go to SoaR and pick `busWorld` (in the `lab11` folder) as your simulated world. The red square is the bus. Now just click on the **joystick** and **start**. The bus will sit at its initial position for a bit, then drive to another corner of the world, wait a while, drive to the next corner, and so on. In our model, we can flag down the bus anywhere on its route, not just when it's stopped; but it would also be reasonable to consider the case when the bus can only be caught at a stop.

If you reload the brain when running a simulation with the moving bus, sometimes SoaR does not reset the bus successfully and the bus does not start in the lower-left corner and eventually drives off screen. You need the reload the `busWorld` in that case.

Go to the file `BusBrainShell.py`. The `busSchedule` function provided in that file will take a time as input and tell you where (in grid indices) the bus will be. An important thing to be aware of is that we are planning at a high level of abstraction as if every robot action takes the same amount of time, even though this isn't actually the case. We set `timePerRobotStep` to be 40. Use this value as the predicted amount of time that passes during each action.

Write a simple Python loop that prints the bus location at a series of times, incrementing time by `timePerRobotStep`. Recall that a robot action is a move between adjacent grid squares and is modeled as taking `timePerRobotStep` time to execute. If the robot has caught the bus at one location, and the bus moves forward first, can the robot catch up to the bus again at the next step? If not, describe the next possible opportunity to catch the bus.

Question 16. Go to the file `BusBrainShell.py`. You'll find that there are three places where you might want to put some code. Location `#1` is where you would put things you need to do only once, like define a function or create an instance of an object. Location `#2` is where you'd put things you need to do every time the robot needs to replan (ultimately a call to the search function needs to go here, either directly or via a call to the `plan` method of a `Planner` object). Location `#3` is where you take whatever plan structure you have and hand the sequence of robot locations, of bus locations, and of times to the `drawUsAndBus` routine, and then extract out the first action for execution. Show the additions you made the code and explain them.

Question 17. Explain the behavior that you see. Does it make sense?

Question 18. Watching the planner run and print out the starting state each time it is called, you should be able to see the time taken by actual robot moves. How long is the longest? The shortest?

Question 19. Try experimenting with values of `timePerRobotStep` that are both larger and smaller than the current setting. Explain what happens and why.

Checkpoint: 11:30 PM

- Demonstrate your planner to your LA, running in simulation, in this new search space.

Question 20. Try out your planner with the real robots. You can run them in the open areas at the ends of the lab. To run on the real robots you will have to make several changes to your code:

- On the real robot, we cannot know the true pose, as we can in the simulator using the `cheatPose` function, defined in `BusBrainShell.py`. Instead, you can call the `pose` function, which returns the robot's pose in a coordinate system defined when the robot is initially turned on; the robot is at the origin of that system and is pointing along the positive `x` axis. To make the situation be similar to the `busWorld` simulation, in which the robot starts out at pose $(3.0, 3.0, 0.0)$, you will need to add that initial pose to the value returned by `pose()`.
- The gains defined in `XYDriver.py` are set much too high for the real robot; try changing them to something smaller.
- In the `done` method of the `XYDriverOA` class, the robot decides that the behavior is done when it gets too close to an obstacle. For this to work even remotely reliably with real sonar sensors, you should increase the threshold for being too close to 0.3.

Think about where you need to start the robot to match the `busWorld`. Also, think about the value of the `timePerRobotStep` parameter and whether it needs to be changed. Describe your approach and the results you observe.

Question 21. What happens if you delay the robot by placing an obstacle in its way for a while and then removing it?

Checkpoint: 12:45 PM

- Demonstrate your planner to your LA, running on the real robot.

The following questions can be done with the simulator only. You can finish them at home or do them in the lab if you have time.

Depth first search

Question 22. Currently, the `GridBrain` asks for a plan using breadth-first search. Change it to use depth-first search. What happens? Why?

A more general goal condition

Question 23. Currently, `GridBrain` 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. 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.

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.

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.

Post-Lab Writeup and Exercises: Due before lecture on Nov. 20

All post-lab hand-ins should be written in clear English sentences and paragraphs. We expect at least a couple of sentences or a paragraph in answer to **all** of the numbered questions in this lab handout (including the software lab).

We also want the code you wrote. When you write up answers to programming problems in this course, *don't* just simply print out the code file and turn it in. Especially, don't turn in long sections of code that we've given you. Turn in your own code, with examples showing how it runs, and explanations of what you wrote and why.

We also expect at least a couple of sentences or a paragraph in answer to the reflection questions below. We're in interested in an explanation of your thinking, as well as the answer.

Question 24. The use of a single `timePerRobotStep` is clearly a limitation. How could we improve the model we are using to include actions with different times? What algorithmic extensions would we need to deal with the improved model?

Question 25. How could we deal with times that vary randomly? For example, if bus schedule times were modeled as an average time plus or minus a uniformly distributed random number that ranges from 0 to 10% of the average time.

The following section is not required. Explorations are things that you can work on if you're interested and/or have extra time in lab. If you hand in a page or two describing some work you did on this, you are eligible for up to an extra half a point on the lab.

Explorations

Include angles in the state space

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. 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.

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.

This will require changing the planner and the execution components of the system.

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 us 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 `GridWorld.py` that can be queried to get the cost of being in a state.