MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.099—Introduction to EECS I
Fall Semester, 2006

Assignment for Week 13

- Issued: Tuesday, Nov 28
- Includes homework and preparation due before class on Thursday, Nov 30
- Post-lab: Due December 5

Final exam: The final exam for 6.081 will be held on Tuesday, December 19. It is a 6-hour *ex camera* (take-home) exam, starting at 1:30PM. It will consist primarily of conceptual and technical questions motivated by the lab on December 5 and 7, which will, we hope, integrate all the material we have covered in the course.

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.

There are two big gaps between state-space search as we have studied it so far, and the reality of the robot.

First, the basic search model assumes we know the initial state of the system. We don't know the robot's pose exactly—we spent much of last lab just trying to figure out where the robot is, and ending up with a distribution over its pose. We will make a big assumption for the purposes of planning, that the robot is actually in the pose that is currently the most likely, according to the belief state.

Second, the basic search model assumes that the "dynamics" of the world are deterministic. That is, among the set of possible successors of a state, we can reliably choose which one will occur. This is like assuming there is no noise in the transition model, and each action has exactly one outcome state, with probability one. We will make this assumption for the purposes of planning.

This is strategy might seem pretty misguided. We don't know where the robot is, and we don't know what the outcomes of its actions are going to be, but we're going to close our eyes and generate plans anyway. This would be completely ridiculous if we were then going to execute the entire plan without looking at the world again. But we will gain a large measure of robustness by pursuing an approach called *continuous replanning.* This means that after the robot takes the first step of a plan, we will re-estimate the pose, take the most likely one to be our initial state, and re-plan. Thus, errors in localization and execution can be corrected after a single step.

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

1. Homework in preparation for lab on Nov. 30

There are problems to work with the online tutor, some of which review material from earlier in the semester. You can also use the tutor to check your answers to the following programming problem.

Experimenting with breadth-first search

The code file `search.py` contains the `breadthfirst` and `numberTest` code discussed in lecture. Load this into Python (not SoaR, just ordinary Python) 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, or squaring. 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 an expanded list.

Robot on an infinite grid: Consider a robot is 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 an expanded list.

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 square, and a boolean that says whether or not to use an expanded 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 $(1, 1)$ to the upper right $(8, 8)$.

You can use the tutor to check your answers as you work through this part of the assignment.

2. In lab on Nov. 30

Now 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've already discussed an approach for dealing with the uncertainty in state estimation and transitions by pretending they're deterministic and using continuous replanning.

Now 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 abstract states of the world and an accompanying set of actions that can reasonably reliably move between those states. What is an abstract state? You can think of it as a set of states in the underlying state space.

Here is one candidate abstraction:

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 the results of 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. It is important that the state estimator use a representation that is at least as fine-grained as the state space of the planner, so that we know what initial state is most likely.

So, the planning part of this is relatively straightforward. Here's the code we use to invoke the search procedure, which should feel very familiar to you.

```
def plan(init, goal, gridMap):
    def s((i,j)):
        return [s for s in ((i-1,j),(i,j-1),(i,j+1),i+1,j)) if gridMap.free(s)]
    def g(node):
        return node.state == goal
    result = search.breadthFirst(init, g, s, True)
    return result
```

The hard part of making this work is building up the framework for executing the plans once we have them. We can think of the overall system now as being composed of *high-level steps* and *low-level steps*. (In the code, these are called *big step* and *little step*).

In a high-level step, the robot

- Updates its belief about the current pose and returns the most likely state
- Calls the planner to get a path to the goal from the most likely state
- Executes the first step of the plan

Execution of a plan step requires many low-level steps of velocity adjustment, until the "waypoint" (in this case, the center of the grid square that's the first step of the path) is reached.

In a low-level step, the robot computes new forward and rotational velocities based on the current sonar and odometry readings. (In fact, in the code we're giving you, the robot ignores the sonar in the low-level driving, which is not a good idea, but we didn't have time to fix it. We'd love it if one of you would like to!) It checks to see if we are close enough to the low-level goal, or waypoint, and if so, tells the high-level control loop about it.

A Simple Example

Let's work through an example, in terms of what happens on successive low-level steps of the brain. Imagine that the robot is in square (1,1), and also believes it is in square (1,1), and wants to reach square (2,2).

Step 1:

1. In the first big step, we update the state estimate and find that our most likely location is (1,1).
2. Then we call the planner, and receive the plan ((1,1),(1,2),(2,2)).
3. Now we need to execute the high-level action of driving from (1,1) to (1,2), which is the first step of our plan. We store the local goal (1,2).
4. We ask the `DriveModule` for the best wheel velocities for driving toward (1,2), and set those as our motor output.

Step 2:

1. We test to see if we're close enough to (1,2) to terminate the high-level action. We aren't.
2. We get velocities from the `DriveModule` and execute them.

Step k:

1. We test to see if we're close enough to (1,2) to terminate the action. We are. We change the status of the driver to be `finished` and don't move.

Step k+1: We notice that the low-level action terminated and that it's time for a new big step!

1. We update the state estimate and find that our most likely location is (1,2).
2. Then we call the planner, and receive the plan ((1,2),(2,2)).
3. Now we need to execute the high-level action of driving from (1,2) to (2,2), which is the first step of our plan. We store the local goal (2,2).
4. We ask the `DriveModule` for the best wheel velocities for driving toward (2,2), and set those as our motor output.
5. Return to Step 2, with new local goal of (2,2).

Of course, it's entirely possible that when we do the state estimation on step $k+1$, we'll find that the robot seems to be in another location altogether. That's okay. We'll just make a plan from there to the goal and execute the first step, and see where we wind up!

How the driver really works

Now we're really getting to the nitty gritty. What does it mean for the robot to go from (1,1) to (1,2)? We have to be able to set velocities that take the robot closer to (1,2) and to know when it has gotten there.

One way of deciding whether it has arrived at the goal would be to do pose estimation after every small step, and to stop when the most likely pose is (1,2). There are a number of reasons why this is not such a good idea.

Question: What are they?

Instead, we are going to use the robot's local odometry to drive from one square to another. We know we shouldn't trust the odometry over a very long distance, but for local moves it isn't too bad. And, if we occasionally really mess up, state estimation and the high-level planner will take care of the problem.

Now we have to do some playing around with coordinate frames. First of all, (1,2) are indices into the grid; we need to convert those grid indices into real-valued coordinates in the global frame, which is the frame we are using for state estimation. We can use those coordinates to obtain the vector (dx_g, dy_g) in the global frame that will move the robot from the current pose to the destination pose. Now, the odometry is reported in a completely different reference frame that generally has no connection to this global frame. We need to put our desired displacement vector into the odometry frame.

You know the robot's pose in the odometry frame (x_o, y_o, θ_o), and your best guess at its pose in the current global frame (x_g, y_g, θ_g). You also have some goal, (x'_g, y'_g), in the global frame. The key is that regardless of what frame you're in, you could move the robot from the current location to the desired location by turning by some angle $d\theta$, and moving in that direction some distance D.

The desired change in position in the global frame is $dpos_g = (dx_g, dy_g)$. The angle $d\theta$ is the difference between the heading that would move the robot toward the goal and the current heading: $d\theta = \tan 2(dy_g, dx_g) - \theta_g$. The distance is the magnitude of the change in position: $D = |dpos_g|$. Because the change in heading $(d\theta)$ is the same in global or odometry frame, the desired heading in odometry frame is $\theta'_o = \theta_o + d\theta$, and the new desired position in odometry frame (obtained by moving a distance D in the direction θ'_o is:

$$\begin{aligned} x'_o &= D * \cos(\theta'_o) + x_o \\ y'_o &= D * \sin(\theta'_o) + y_o \quad . \end{aligned}$$

So now our job is to drive there.

The driver calculates the angle from the robot's current location to the desired location; then it uses two feedback loops, like this:

- If the robot is pointed along the desired heading (toward the destination), then it moves forward with a velocity proportional to the distance to the goal.
- If the robot is not pointed along the desired deading, it rotates with a velocity proportional to the error in heading.

So, the robot turns to face where it's going, and then drives there. If, as it's going along, it finds that the angular error has gotten too big, it stops and rotates, and then goes again.

Checkpoint: 1:45 PM

- Discuss the relative merits of doing the low-level driving based on local odometry versus continuously re-localizing in the map.

1. Using the planner

You can find the following files in `SoaR/brains/ps13code`:

- `CheatPlan.py`
- `CheatPlanAngle.py`
- `DriveModule.py`
- `DriveModuleAngle.py`
- `GridModule.py`
- `GridPlan.py`
- `PlannerModule.py`
- `PlannerAngleModule.pyc`
- `PlannerAngleModuleX.py`
- `search.py`
- `KJ.py`
- `KJ.dat`
- `utilities.py`

Now, start up SoaR simulator, using the `KJ` world, and use `CheatPlan.py` as the brain. This brain "cheats," and instead of using the state estimator to figure out the current pose, it just reads it straight out of the simulator. We can study the planner's behavior in this mode first, without having to worry about the localization.

When it starts up, you'll see two windows. The first is a familiar grid belief state window. It's useful because it shows you the obstacles, but it will never be updated. The second window is blank to start with, but once the brain starts to run, it shows the current plan. The goal is currently wired into the `CheatPlan.py` program to be grid square (7,3). You can, of course, change it.

If you hit **step**, then the system will start a high-level step (and execute just one low-level one). It makes a plan, which is shown in the plan window. The green square is centered on the robot's actual current pose. The magenta square shows the grid square that it maps on to (the first time you start up the robot, it will be (5,17)). The dark red square is the first waypoint in the plan, and will be the goal of the first low-level action. The gold square is the goal, and the dark blue squares show the rest of the steps in the plan.

Question: What is the depth of the search tree when it finds a solution? What is the branching factor? Roughly how many nodes will be expanded to find a solution with the expanded list turned on? How about with it off?

Now, let the program run. You can see the robot turning toward its next destination, moving until gets there, and stopping. Then the planner is called again and a new plan is drawn in the plan window and a new high-level action is started. Experiment with moving the robot around in the simulator and seeing what happens.

You may sometimes see the robot turn around and go backwards. Think about why that might be happening.

Question: From the default robot start position and start goal, what plan does it make? Do you think it's the best one?

Gridplan

Now, run the robot in the simulator again, but using the `GridPlan.py` brain. Does the localization work the way you expect it to in this world? Have the robot drive from the default start to the goal. How many steps does it actually take? Does it get confused along the way? If the robot gets stuck, you may have to rescue it by moving it slightly until it can move forward again.

Question: Change the goal so that the robot is happy to be in any one of the four corners of the maze. This will require changing the code around a bit in a number of places.

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.

Finally, let's try it on the real robot. Use GridPlan but go and run it in the real world. Does the behavior differ from the simulator?

Question: Try starting the robot out in the same pose as you did in the simulator and counting (either in your head, or by modifying the program) how many high-level steps it takes to get to the goal. Compare that to what happens in simulation. If they're different, explain why.

Checkpoint: 3:30 PM

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

2. Changing the state and action spaces

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: 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 should 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.

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

Question: Go to the file `PlannerAngleModuleX.py`. You'll find that the definition for the procedure `plan` 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 argument `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.breadthFirst`.

Question: 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, and speculate about another formulation.

3. Optional Exploration: Actions with different costs

Now, what if 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.
- 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 Due December 5

Hand in answers to all of the questions raised in this lab assignment, written in coherent English sentences.

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.