MASSACHVSETTS INSTITVTE 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.