Software Lab 14 *6.01 – Fall 2011*

Plan From Outer Space

Goals: In this week's software and design labs, you will build a system that can run on the robot, allowing it to make a 2D world map of the obstacles around it and plan a path to a desired destination.
This software lab focuses on geting the basic planning infrastructure up and running. You will:

- Model the world in which the robot moves as a discrete 2D grid
- Write a state machine representing the robot's dynamics in the grid world
- Apply **A* search** and **uniform cost search** to build a planner which finds optimal paths for the robot to move among states in the discrete map of the world

Resources: This lab should be done **individually**.
Do `athrun 6.01 getFiles`. The relevant files in the distribution are:

- `plannerStandaloneSkeleton.py`: file to write your code in
- `mapTestWorld.py, bigPlanWorld.py`: files describing world configurations that you can read in as specifications of planning problems.

Read Section 8.5 of the readings, if you haven't already.

Our goal in this lab is to build a *planner*, which provides a proposed path for a robot to navigate to a goal, given a map of obstacles for the robot's motion. We begin by formulating the basic planning problem as a search in a two-dimensional grid of states, build a machine that makes a map using sonar data, and make the robot dynamically replan new paths through the world as its map changes.

1 Grid Map Representation

Consider a **circular robot**[1]. We will be making plans for this robot moving around a simulated world containing obstacles. The state of the robot can be described by its **pose**: x in meters, y in meters, and θ in radians: (x, y) is the location of the robot's center and θ is the angle that the front of the robot makes relative to the x axis. We will use instances of **the `util.Pose` class** to represent the robot's poses in the real world. We can obtain the `util.Point` representing the x, y position of a pose by `pose.point()`.

Planning paths as continuous curves in x, y, θ is very hard, so we will instead model the robot's state space somewhat more coarsely. We will ignore orientation altogether and we will discretize

[1] Not unlike a spherical cow. `http://en.wikipedia.org/wiki/Spherical_cow`

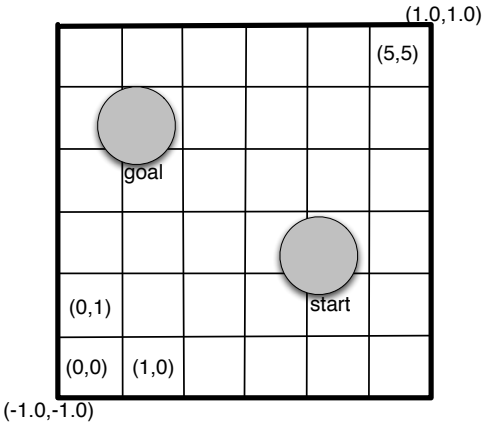
Some of the software and design labs contain the command athrun 6.01 getFiles. Please disregard this instruction; the same files are available on the 6.01 OCW Scholar site as a .zip file, labeled Code for [Design or Software Lab number].

the x, y positions of the robot into a grid, with indices ranging from `0` to `xN - 1` for x and from `0` to `yN - 1` for y.

So, a state for the robot, in the space that we will search for plans, is described by the coordinates of a grid cell – a tuple of two indices, `(ix, iy)`. It is important to be clear about when you are working with:

- a real world pose (an instance of `util.Pose` with coordinates in meters and radians), or
- a real world point (an instance of `util.Point` with coordinates in meters), or
- indices describing a grid cell (a tuple of two indices in a grid).

The figure below shows the robot in a world where the x and y coordinates vary from -1m to $+1$m, which is discretized into 6 intervals in each dimension, resulting in 36 grid cells. The lower-left cell always has indices `(0, 0)`, with x increasing to the right and y increasing going up.



We will use **the `BasicGridMap` class** (defined in `lib601.basicGridMap.py`) to represent the grid and indicate which grid cells contain obstacles. We can create a grid with obstacles from a world file of the kind that we use in soar. These world files specify the min and max values for x and y as well as boundary lines for the walls and obstacles.

These are the most important methods and attributes of the class `BasicGridMap` (more detailed documentation is available in the **software documentation** :

- `pointToIndices(self, point)`: takes a `util.Point` representing coordinates of the robot in the real world map and returns a tuple of integer indices, representing the grid cell of the robot.
- `indicesToPoint(self, indices)`: takes a tuple of integer indices, representing the grid cell of the robot and returns a `util.Point` representing coordinates of of the center of that cell in the real world map.
- `xN, yN`: number of cells in the x and y dimensions
- `robotCanOccupy(self, indices)`: returns `True` if the robot can be positioned with its center in this cell and not cause a collision with an obstacle in the world, and `False` otherwise.
- `xStep`: attribute representing the length, in meters, of a side of a grid cell; we assume the cells are square (so `yStep` equals `xStep`).

Step 1.

Wk.14.1.1 Solve this tutor problem to develop an understanding of the grid map representation.

2 Grid Dynamics

Now we will think about how to design **a state machine class** that represents the dynamics of the robot on a grid map. An instance of the `GridDynamics` class (which you will write) will be a state machine, whose inputs are actions the robot can take to move on the grid, and whose states are pairs of grid indices indicating the robot's position.

We will use *uniform cost search* **(UCS)** to find shortest paths through the world defined by the grid dynamics: UCS requires each action to be annotated by its cost, so we will use the output of the state machine to encode the cost of each action.

The `GridDynamics` class needs to provide a `legalInputs` attribute and a `getNextValues` method. It does not need to supply a `done` method or a starting state: we will want to specify the starting state and the goal when we call the search procedure.

The state should be a pair `(ix, iy)` of indices representing the robot's position in the grid.

The state machine should allow 8 possible actions: moving to the four directly adjacent and the four diagonally adjacent grid cells. The elements of `legalInputs` will be the names of each of these actions. It doesn't matter what names you give the actions; in fact, the names can be tuples that describe the actions.

Remember that the input of the state machine will be one of the elements of the list of legal inputs; and the output of the `getNextValues` method should be a pair `(nextState, cost)`, where `nextState` is an `(ix, iy)` pair, and `cost` is a positive number representing the cost of taking that action. The cost of each move should be the distance the robot will travel, measured in meters (the length of a grid-cell side, in meters, is stored in the `xStep` attribute of instances of `basicGridMap.BasicGridMap`). Remember that a diagonal motion is longer than a horizontal or vertical one.

The `__init__` method of your `GridDynamics` class should take as input an instance of the `BasicGridMap` class, as described in section 1.

When implementing `getNextValues` be sure to consider the following:

- If the robot attempts to move into a square that it cannot occupy, it should stay where it was, but the cost should be the same as if the move had been legal.
- You do not need to worry about moving off the boundary of the map, because the boundary squares will already be marked as not occupiable.
- You do, however, have to be extra careful about moving diagonally: when your current and target squares are free, but one of the other two squares that are adjacent to both the current and target squares is occupied, it is possible that the robot will have a collision. Such a move should be treated in the same way as attempting to move into a square that is occupied.
- *When we connect up with the map maker it may occasionally happen that the grid cell the robot is currently in is suddenly marked as not occupiable; your dynamics should allow the robot to move* ***out*** *of a cell that is not occupiable, as long as the cell it is moving into is occupiable.*

Step 2. Implement the `GridDynamics` class in the file `plannerStandaloneSkeleton.py`.

Check Yourself 1. This procedure is defined in `plannerStandaloneSkeleton.py`:

```
def testGridDynamics():
    gm = TestGridMap(0.15)
    print 'For TestGridMap(0.15):'
    r = GridDynamics(gm)
    print 'legalInputs', util.prettyString(r.legalInputs)
    ans1 = [r.getNextValues((1,1), a) for a in r.legalInputs]
    print 'starting from (1,1)', util.prettyString(ans1)
    ans2 = [r.getNextValues((2,3), a) for a in r.legalInputs]
    print 'starting from (2,3)', util.prettyString(ans2)
    ans3 = [r.getNextValues((3, 2), a) for a in r.legalInputs]
    print 'starting from (3,2)', util.prettyString(ans3)

    gm2 = TestGridMap(0.4)
    print 'For TestGridMap(0.4):'
    r2 = GridDynamics(gm2)
    ans4 = [r2.getNextValues((2,3), a) for a in r2.legalInputs]
    print 'starting from (2,3)', util.prettyString(ans4)
```

It creates two different instances of `GridDynamics`, tests them, and prints out the results. Be sure you understand what the results should be. Note that the `TestGridMap` class is a simplified version of the `BasicGridMap` class with a small fixed grid; it's only useful only for testing your dynamics. Each time we create an instance of `TestGridMap`, a window will pop up showing the map (it's basically the same in both cases, except for the world is bigger in the second (the same number of cells, but they are larger)).

Step 3. Test your code by running `testGridDynamics()` or other test cases you find helpful, and be sure it is correct.

Step 4. Note that three of the test cases in the tutor problem are the same as the first three test cases in our `testGridDynamics`; but the answers may look different because we are iterating over your `legalInputs` attribute in one case, and our `legalInputs` attribute in the other, and they may be in a different order. The tutor takes that into account when checking.

Wk.14.1.2 Paste your `GridDynamics` class definition and any helper procedures it needs into this tutor problem; check it and submit.

3 Making a plan and sticking to it

Now that we have a state machine that represents the dynamics of our domain, we can run search algorithms on that state machine to find good paths through the space.

We want to construct a procedure

```
planner(initialPose, goalPoint, worldPath, gridSquareSize)
```

that plans a path using **ucSearch.smSearch** on the grid dynamics of a grid map. It should draw the resulting path in the map, and return it.

Step 5. Implement the `planner` procedure in `plannerStandaloneSkeleton.py`. It should return the plan found by the search.

You need to pass a grid map as an argument to your `GridDynamics` class initializer. You can create a grid map corresponding to a soar world with

```
basicGridMap.BasicGridMap(worldPath, gridSquareSize)
```

where `worldPath` is a string representing the name of a file containing a soar world definition, and `gridSquareSize` is the size, in meters, of a side of each grid cell. Don't worry about what `worldPath` needs to be; just take the argument you're given and pass it through to initialize the `BasicGridMap` instance.

Then, you use `ucSearch.smSearch` (see **the documentation** on the class website) to search that machine for a path from the initial pose to the goal point. You will need to convert both the initial pose and the goal point into grid indices for planning.

When a `BasicGridMap` is created, it will create a new window, displaying the obstacles in the world. Your planner should draw the path it finds in that window. **The `BasicGridMap` class** provides the method `drawPath(self, listOfIndices)`, where `listOfIndices` is of the form `[(ix1, iy1), (ix2, iy2), ...]`, specifying a list of grid-index pairs. It draws the starting cell in purple, the ending cell in green, and the rest in blue. Remember that the plan returned by the search is a list of `(action, state)` tuples, so you cannot pass that in directly.

To get the algorithm to display the states it is visiting, define your goal test function (inside the planner procedure) to have this form:

```
def g(s):
    gm.drawSquare(s, 'gray')
    return yourGoalTestHere
```

where `gm` is the name of your instance of `BasicGridMap`, `s` is a pair of grid indices, and `yourGoalTestHere` is the actual expression that you are testing to see whether `s` is a goal state.

Step 6. At the top of `plannerStandaloneSkeleton.py`, there are definitions of two worlds you can test in, each with a reasonable start and goal point and grid square size specified. Test your procedure in a world by running `plannerStandaloneSkeleton.py` in Idle (be sure to start Idle with -n, so you can see the graphics), and then evaluating, for example,

```
testPlanner(mapTestWorld)
```

You should see a map window pop up, first showing the obstacles, the states as they are visited, and then, when the planner has completed, the path you drew.

Check Yourself 2. In `mapTestWorld.py` with the discretization, start, and goal as defined in `plannerStandaloneSkeleton.py`, you should find a solution with cost about 6.7 (that is, a path about 6.7 m long.) The search should visit about 800 nodes and expand about 270 states. Be sure you understand why some squares are being colored gray. Does this search seem efficient?

Step 7. Read Section 8.6 of the readings, if you haven't already.

Let us now try to improve the speed of the search algorithm, by employing a heuristic function to prune the search tree. As we saw in lecture and is described in the lecture notes, the use of heuristics with suitable properties – that is *admissible* heuristics, can lead to significant performance gains while still guaranteeing that a shortest path will be found. We will use the popular and well known A* algorithm, which is nothing more than the uniform cost search algorithm with an added heuristic cost function.

Think of a suitable admissible heuristic, implement it, and see how it affects the planning process. Be sure that your heuristic is expressed in the same units as the cost function.

Check Yourself 3. Test your heuristic search, first, in `mapTestWorld`. You should find that it doesn't make any difference in the length of the solution, though it may choose a slightly different path. You should find, however, that the number of nodes visited goes down to about 500 and the number of states expanded down to about 150.

Step 8. Now, test in `bigPlanWorld` with and without the heuristic. You should see a very noticable difference in the number of nodes expanded. Keep screenshots showing the paths and visited grid cells with and without the heuristic.

Checkoff 1. **Wk.14.1.3**: Demonstrate your search running in `bigPlanWorld.py` with the heuristic function. Compare its behavior to the search without a heuristic function. Explain the difference in visited states.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011