Design Lab 13: I'm the Map!

You can use any computer that runs soar.

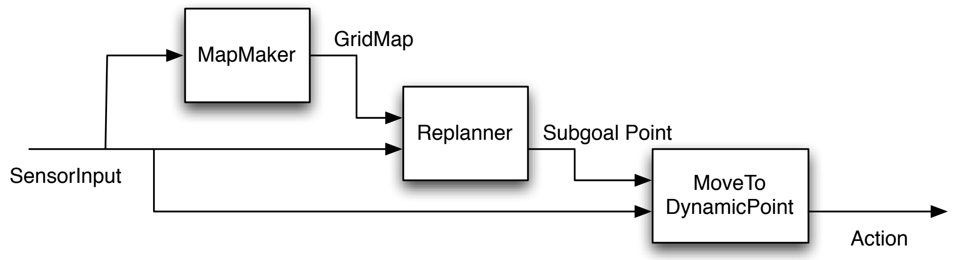
- **Athena machine:** Do `athrun 6.01 update` and `add -f 6.01`.
- **Lab laptop:** Do `athrun 6.01 update`.
- **Personal laptop:** Download design lab 13 zip file from course web page.

Checkoff 1. **From software lab. If the staff is busy at the beginning of lab, go ahead and start working on the next checkoffs.** Demonstrate your search running in `bigPlanWorld.py` with the heuristic function. Compare its behavior to the search without a heuristic function.

1 Introduction

In this lab, we will connect the planner from Software Lab 13 with a state machine that dynamically builds a map as the robot moves through the world. The robot will, optimistically, start out by assuming that all of the locations it does not know about are free of obstacles, it will make a plan on that basis, and then begin executing the plan. But, as it moves, it will see obstacles with its sonars, and add them to its map. If it comes to believe that its current plan is no longer achievable, it will plan again. Thus, starting with no knowledge of the environment, the robot will be able to build a map. We'll start by building a simple map maker, then see what happens as the sensor data noise increases, then adapt the map maker to handle noise.

Here is a diagram of the architecture of the system we will build.



Our architecture has three modules. We will give you our implementations of the replanner and the module that moves to a given point; you will concentrate on the mapmaker.

The `move.MoveToDynamicPoint` class of state machines takes instances of `util.Point` as input, and generates instances of `io.Action` as output. That means that the point that the robot is 'aiming' at can be changed dynamically over time.

The `replanner.ReplannerWithDynamicMap` state machine takes a `goalPoint` as a parameter at initialization time. The `goalPoint` is a `util.Point`, specifying a goal for the robot's location in the world, which will remain fixed. The robot's sensor input (which contains information about the robot's current location) as well as the `DynamicGridMap` instance that is output by the `MapMaker` will be the inputs to this machine. The replanner makes a new plan on the first step, draws it into the map, and outputs the first 'subgoal' (that is, the center of the grid square that the robot is supposed to move to next), which is input to the driving state machine. On subsequent steps the replanner does two things:

1. It checks to see if the first or second subgoal locations on the current plan are blocked in the world map. If so, it calls the planner to make a new plan.
2. It checks to see if it has reached its current subgoal; if so, it removes that subgoal from the front of its stored plan and starts outputting the next subgoal in the list.

2 Mapmaker, mapmaker, make me a map

Your job is to write a state-machine class, `MapMaker`, in the file `mapMakerSkeleton.py`. It will take as input an instance of `io.SensorInput`. Its state will be the map we are creating, which can be represented using an instance of `dynamicGridMap.DynamicGridMap`, which is like `basicGridMap.BasicGridMap`, but instead of creating the map from a file, it allows the map to be constructed dynamically. The grid map will be both the state and the output of this machine. The starting state of the mapmaker can just be the initial `dynamicGridMap.DynamicGridMap` instance.

For efficiency reasons, we are going to violate our state machine protocol and say that your `getNextValues` method should return *the same instance* of `dynamicGridMap.DynamicGridMap` that was passed in as the old state, as the next state and the output. It should make changes to that map using the `setCell` and `clearCell` methods. If we were to copy it every time, the program would be painfully slow.

What should the mapmaker do? The most fundamental thing it knows about the world is that the grid cell at the very end of a sonar ray is occupied by an obstacle. So, on each step, for each sensor, if its value is less than `sonarDist.sonarMax`, you can set the grid cell containing the point at the end of the sonar ray in the map as containing an obstacle. The sonar hit procedure you wrote in the tutor problem is available as

```
sonarDist.sonarHit(dist, sonarPose, robotPose)
```

A list of the poses of all the sonar sensors are available in `sonarDist.sonarPoses`.

For this check-off, you can build on a class we have constructed, called `dynamicGridMap.DynamicGridMap`. It supports all of the methods and attributes of `basicGridMap.BasicGridMap` (which you used in software lab 13), but instead of constructing itself at initialization from a soar world definition, it allows incremental changing of the map values. It provides these methods:

- `__init__(self, xMin, xMax, yMin, yMax, gridSquareSize)`: initializes a grid with minimum and maximum real-world coordinate ranges as specified by the parameters, and with grid square size as specified. The grid is stored in the attribute `grid`. Initially, all values are set to `False`, indicating that they are **not** occupied.
- `setCell(self, (ix, iy))`: sets the grid cell with indices `(ix, iy)` to be occupied (that is, sets the value stored in the cell to be `True`).
- `clearCell(self, (ix, iy))`: sets the grid cell with indices `(ix, iy)` to be **not** occupied (that is, sets the value stored in the cell to be `False`).
- `occupied(self, (ix, iy))`: returns `True` if the cell with indices `(ix, iy)` is occupied by an obstacle.
- `robotCanOccupy(self, (ix, iy))`: returns `True` if it is safe for the robot to have its center point anywhere in this cell.
- `squareColor(self, indices)`: returns the color that the grid cell at `indices` should be drawn in; in this case, it draws a square in black if it is marked occupied by an obstacle.

The file `mapAndReplanBrain.py` contains the necessary state machine combinations to connect all the parts of system together into a runnable soar brain. You can work in any of the worlds described in the top of the brain file; select the appropriate simulated world in soar, and then be sure that you have a line like `useWorld(frustrationWorld)` that selects the appropriate dimensions for the world you're working in. Be sure to use the simulated world corresponding to the world file you have selected, when you test your code.

Checkoff 2. Show the map that your mapmaker builds for to a staff member. If it does anything surprising, explain why. How does the dynamically updated map interact with the planning and replanning process? Is the total path that the robot takes optimal with respect to the true map?

3 A noisy noise annoys an oyster

By default, the sonar readings in soar are perfect. But the sonar readings in a real robot are nothing like perfect. Find the line in `mapAndReplanBrain.py` that says:

```
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: noNoise
```

and change it to

```
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: mediumNoise
```

This increases the default variance (width of gaussian distribution) for the sonar noise model to a non-zero value.

Check Yourself 1. Run the brain again in this noisier world. What happens? Why?

In fact, we get more information from the sonar sensors than just the fact that end of the ray is occupied. We also know that the grid cells along the sonar ray, between the sensor and the very last cell, are clear. Even when the sonar reading is greater than the maximum good value, you might consider marking the cells along the first part of the ray clear.

Improve your `MapMaker` class to take advantage of this information. You will probably find the procedure `util.lineIndices(start, end)` useful: `start` and `end` should each be a pair of (x, y) integer grid cell indices; the return value is a list of (x, y) integer grid cell index pairs that consititute the line segment between `start` and `end`. You can think of these cells as the set of grid locations that could reasonably be marked as being clear, based on a sonar measurement. *Be sure not to clear the very last point, which is the one that you are already marking as occupied; although it might work now, if you clear and then mark that cell each time, it will cause problems later on.*

Checkoff 3. Show your new map maker running, first with no noise and then with medium noise. What happens? Why? Are there systematic faults in the resulting map?

4 Bayes Map

This is a preview of the next step. Please stay and work on it through the end of this lab period; you can finish it during software lab 14.

One way to make the mapping more reliable in the presence of noise is to treat the problem as one of *state estimation*: we have unreliable observations of the underlying state of the grid squares, and we can aggregate that information over time.

Our space of possible hypotheses for state estimation should be the space of all possible maps. But if our map is a 20 by 20 grid, then the number of possible map-grids is 2^{400} (each cell can either be occupied or not, and so this is like the number of 400-digit binary numbers), which is*much* too large a space to do estimation in. In order to make the problem computationally tractable, we will make a very strong **independence assumption: the state of each square of the map is independent of the states of the other squares.** If we do this, then, instead of having one state estimation problem with 2^{400} states, we have 400 state estimation problems, each of which has 2 states (the grid cell can either be occupied or not).

Luckily, we have already built a nice state-machine class for state estimation, and we can use it to build a new subclass of `dynamicGridMap.DynamicGridMap`, where each cell in the grid contains an instance of `seFast.StateEstimator` (which you implemented in **Wk.11.2.1**).

Your job is to write a the definition for the `BayesGridMap` class, in the file `bayesMapSkeleton.py`. Before doing this, you'll need to think through how to use state estimators as elements of the grid.

Recall that the argument to the `__init__` method of `StateEstimator` is an instance of `ssm.StochasticStateMachine`, which specifies the dynamics of the environment. Here are some points to think about when specifying the world dynamics of a single map grid cell:

- There are two possible states of the cell: occupied or not.
- There are two possible observations we may make of this cell: it is free, or it was the location of a sonar hit.
- You can assume that the environment is completely static: that is, that the actual state of a grid cell never changes, even though your belief about it changes as you gather observations.

Check Yourself 2. Remember that the sonar beams can sometimes bounce off of obstacles; and that when we say a square is clear, we say that it has nothing anywhere in it. What do you think is more likely: that we observe a cell to be free when it is really occupied, or that we observe it as a hit when it is really not occupied? What should the prior (starting) probabilities be that any particular cell is occupied?
If you are having trouble formulating the starting distribution, observation and transition models for the state estimator, talk to a staff member.

You will have to manage the initialization and state update of the state estimator machines yourself. You should be sure to call the `start` method on each of the state-estimator state machines just after you create this grid. You will also, whenever you get evidence about the state of a cell, have to call the `step` method of the estimator, with the input `(o, a)`, where `o` is an observation and `a` is an action; we will be, effectively, ignoring the action parameter in this model, so you can simply pass in `None` for `a`.

You can remind yourself of the appropriate methods for creating a state estimator and for starting and stepping a state machine by looking at the online software documentation.

Your `BayesGridMap` will be a subclass of `DynamicGridMap` and can be modeled directly on the following aspects of `DynamicGridMap.py`:

```
class DynamicGridMap(gridMap.GridMap):
    def makeStartingGrid(self):
        return util.make2DArray(self.xN, self.yN, False)
    def squareColor(self, indices):
        if self.occupied(indices): return 'black'
        else: return 'white'
    def setCell(self, (xIndex, yIndex)):
        self.grid[xIndex][yIndex] = True
        self.drawSquare((xIndex, yIndex))
    def clearCell(self, (xIndex, yIndex)):
        self.grid[xIndex][yIndex] = False
        self.drawSquare((xIndex, yIndex))
    def occupied(self, (xIndex, yIndex)):
        return self.grid[xIndex][yIndex]
```

Here is some further description of the methods you'll need to write.

- `makeStartingGrid(self)`: Construct and return two-dimensional array (list of lists) of instances of `se.StateEstimator`. You can use the attributes `xN` and `yN` of `self` to know how big to make the array.
- `setCell(self, (xIndex, yIndex))`: This method should do a state-machine update on the state machine in this cell, for the observation that there is a sonar hit in this cell. And it should redraw the square in the map in case its color has changed.
- `clearCell(self, (xIndex, yIndex))`: This method should do a state-machine update on the state machine in this cell, for the observation that this cell is free. And it should redraw the square in the map in case its color has changed.
- `occupied(self, (xIndex, yIndex))`: This method returns `True` if the cell should be considered to be occupied for the purposes of planning and `False` if not. You may have to experiment with this a bit, in order to find a good threshold on the probability that the square is occupied.
- `squareColor(self, (xIndex, yIndex))`: This method should return the color we use to draw this square. It will be called by the `drawSquare` method. It should be of the form

  ```
  return colors.probToMapColor(p)
  ```

  where `p` is the probability that the cell is occupied.

Next Checkoff

To be completed during software lab 14 if not finished in lab today. Demonstrate your mapper in `mapAndReplanBrain` using your `BayesMap` module with medium and high noise. If it doesn't work with high noise, explain what the issues are.