Motion Planning

Carter J. Bastian

January 29, 2016

Introduction

In this assignment, a series of probablistic algorithms were implemented so as to enable a robot to navigate between two points in a pseudo-continuous, obstacle-filled environment. The first robot, which navigates with using a Probabilistic Road Map, is an arm with four linked segments. The second robot, which navigates using a Rapidly Exploring Random Tree, is a planar robot capable of moving differentially (or in one case, omnidirectionally).

Probabilistic Road Map (PRM)

The implementation of this algorithm involves iteratively selecting a random free (AKA valid) configuration, adding it to a "road map" of known configurations, and connecting it to each of the K-closest configurations that were known before it.

To this effect, the only necessary data structure was a HashMap, which had a set of configurations as its keys and an inner HashMap as its values. The inner HashMap maps a configuration's set of closest known neighboring configurations to the cost of moving from the configuration to any given neighbor.

To illustrate how this HashMap was updated, below is partial listing of the `addNode`method, which adds and connects a free node to the rest of the graph:

```
protected void addNode(Vector n, List<Vector> knownConfigs) {
      List<Vector>closestNeighbors = nearestKNeighbors(configs, n, kValue());
      HashMap<Vector, Double> validNeighbors = new HashMap<Vector, Double>();
      ListIterator<Vector> it = closestNeighbors.listIterator(0);
      Vector curr;

      while (it.hasNext()) {
        curr = it.next();
        if (getEnvironment().isSteerable(getRobot(), n, curr, RESOLUTION)) {
          validNeighbors.put(curr, getRobot().getMetric(n, curr));

          // Update the other direction of the graph
          if (getEnvironment().isSteerable(getRobot(), curr, n, RESOLUTION)) {
            if (roadMap.containsKey(curr))
              roadMap.get(curr).put(n, getRobot().getMetric(curr, n));
          }
        }
        roadMap.put(n, validNeighbors);
      }
}
```

After the graph is created, A* search is used to find a path the starting configuration and the goal. The full implementation of RRT is in the source file `/assignment_motion_planning/PRMPlanner.java`.

Figure 1: A robot arm using PRM to get from the lower right corner of the screen to the upper left corner of the screen with obstacles.

Rapidly Exploring Random Tree (RRT)

The implementation of the RRT algorithm involves iteratively selecting a series of configurations, finding the closest known-configuration to that point, and expanding said known-configuration in a random direction for some duration of time. This process creates a tree from the Start node out across the map, ass can be seen in Figures 2 and 3.

In order to implement this, two data structures were used. The first and most important is a map, mapping each configuration to the edge on the graph by which the configuration was reached. Notice that `Edge`is a subclass defined containing a reference to the parent node, selected control, and duration from which a child vertex derives. The second data structure used was a linked list of all the configurations currently on the tree. This was necessary only for the sake of convenience when selecting a random configuration's nearest neighbor.

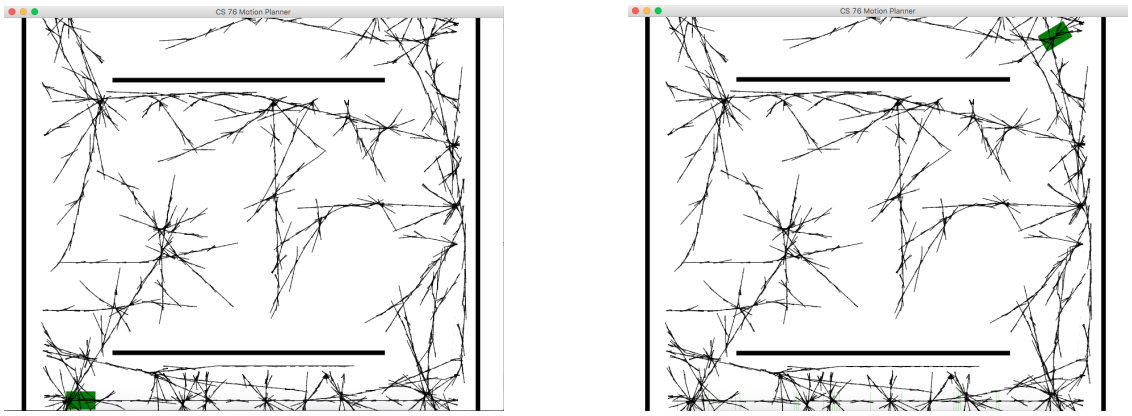

Figure 2: A planar robot (differential drive) using RRT to navigate from the bottom left of the screen to the top right ($k = 20,000$).

To illustrate the difference in how a new configuration is added in RRT (compared to PRM), a listing of the newConf method follows. This method randomly expands the tree by one configuration:

```
private boolean newConf(Vector qnear, double duration) {
    Vector qnew;
    Vector control = getRobot().getRandomControl(random);

    if (getEnvironment().isValidMotion(getRobot(),
                                       qnear,
                                       new Trajectory(control, duration),
                                       RESOLUTION)) {
      // Apply the valid motion to get the next configuration
      qnew = getRobot().move(qnear, control, duration);

      // Add the new vector to the list and the map
      if (! parents.containsKey(qnew)) {
        configs.add(qnew);
        parents.put(qnew, new Edge(qnear, qnew, control, duration));
        return true;
      } else {
        return false;
      }
    } else {
      return false;
    }
  }
```

The tree is explored to a size of K configurations using this `newConf`function by the following for-loop inside the `growMap`function:

```
    for (int i = 0; i < K; i++) {
      qrand = getRobot().getRandomConfiguration(getEnvironment(), random);
      qnear = nearestNeighbor(configs, qrand);

      while (! newConf(qnear, DEFAULT_DELTA) && ++tries <= maxTries)
        ;
      tries = 0;
    }
```

After the tree is made entirely, a trajectory is formed by back-chaining from the nearest known configuration on the RRT back to the root node as follows:

```
protected Trajectory findPath() {
    LinkedList<Edge> chain;         // The Backchain
    Vector curr;                    // Current configuration
    Trajectory result;              // The forward-directional trajectory object

    curr = nearestNeighbor(configs, getGoal());
    chain = new LinkedList<Edge>();

    if (parents.get(curr) == null)
        System.out.println("Curr is null: " + curr + ", " + parents.get(curr));

    while(parents.get(curr) != null) {
```

```
            chain.addFirst(parents.get(curr));
            curr = parents.get(curr).parent;
        }

        result = new Trajectory();
        for (Edge e : chain)
            result.addControl(e.control, e.duration);

        return result;
}
```

A full implementation of the `edge`class and the RRT algorithm is available in `/assignment_motion_planning/RRTPlanner.java`.

Results and Analysis of PRM and RRT

Summary of Performance Results

Both methods performed well on the standard tests. PRM was able to solve the basic Robot Arm problem in about 42 seconds using 1000 total nodes (and generating about 23.8 nodes per second). RRT was able to solve the standard Planar Robot Problem in about 48.5 seconds using 20,000 nodes with an error of only 0.1.

Furthermore, given slightly more time, the RRT implementation was able to perform quite well on much more difficult problems. For example, as can be seen in Figure 3, RRT enabled a differential planar robot to navigate a maze-like environment with 100,000 nodes to an error of only about 0.18.

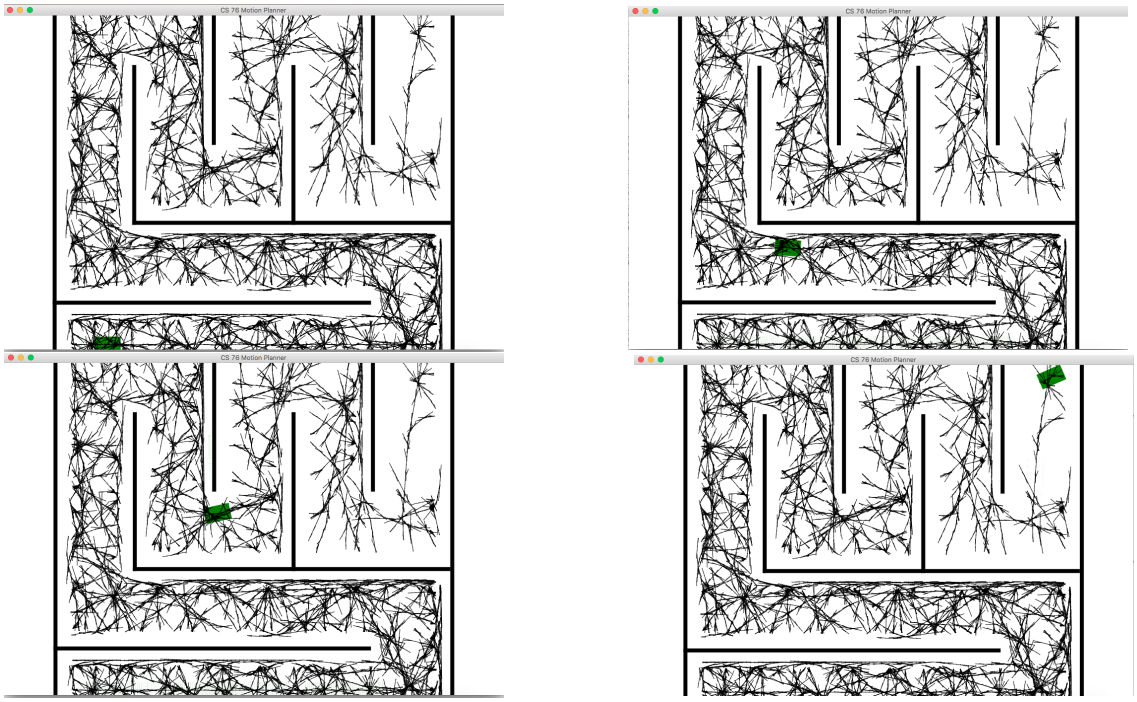

Figure 3: A planar robot (differential drive) using RRT to navigate through a maze ($k = 100,000$).

Further, even in a real world environment, the RRT-enabled robot was able to navigate from outside the

city limits of Hanover to the middle of the Green after generative 100,000 nodes with an error of only about 0.2 (see Figure 4).

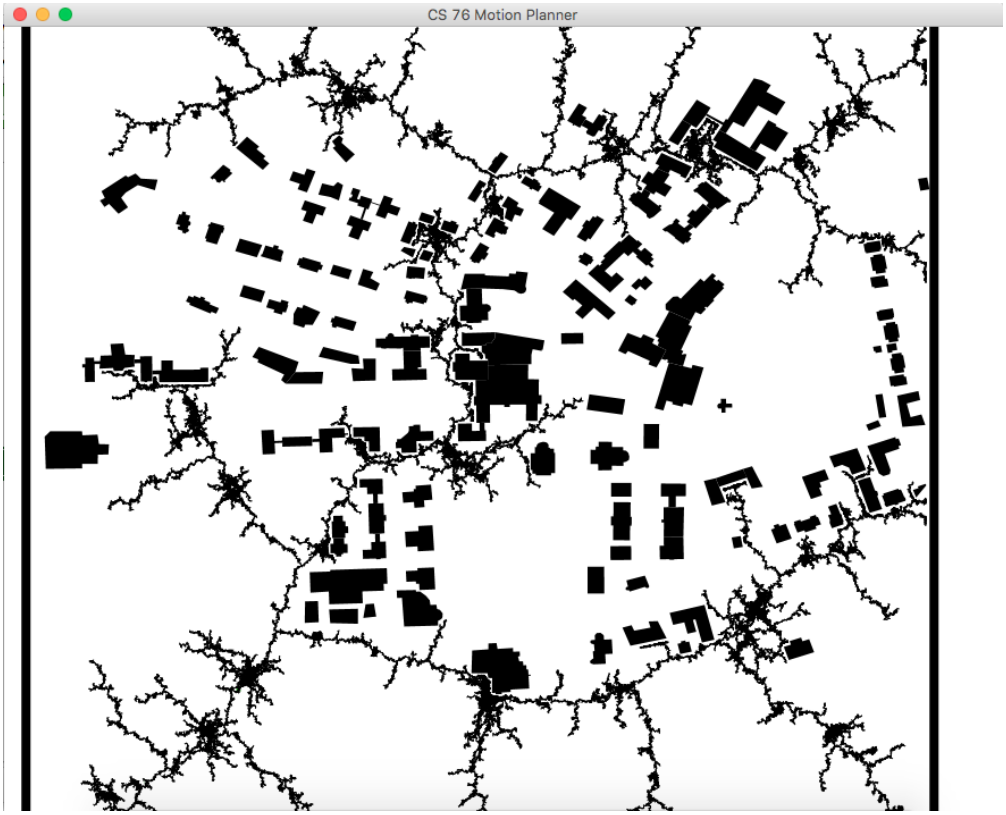

Figure 4: A planar robot (omnidirectional drive) using RRT to navigate through the city of Hanover to the middle of the Green ($k = 100,000$).

The full outputs of multiple runs of the programs can be found in the file `/assignment_motion_planning/results.txt`.

Bottlenecks and Possible Improvements

After using both the `Benchmark.java`file and the VisualVM profiler, I realized that the bottlenecks in the code come largely from the computational expenses of checking each possible path for collision.

What's interesting is that, for environments with relatively large obstacles, increasing the resolution may have cause a drastic improvement in computational resources used while also having little effect on the final trajectory of the robot.

For example, by increasing the resolution in `MotionPlanner.java`from 0.1 to 0.3, the PRM algorithm consistently found a valid solution to the basic Robot Arm problem in less than 8.5 seconds (as compared to about 42 seconds) otherwise.

The full benchmark results of my experiments are in the source directory entitled `planar_benchmark.results`and `arm_benchmark.results`.

Further, while both can be effective, the most important difference is that PRM can be iteratively extended for a long period of time without repeating, whereas RRT can be cross-applied to multiple goals after a single run. These pros and cons are extremely important in terms of when you would use which algorithm.

A Low-Memory Alternative to PRM

Description

By iteratively adding nodes to the PRM graph, one is making a graph that, while quick, is also poorly connected. My idea was to remove this inefficiency by first selecting all of the K free nodes (which will ever be added to the graph), and then connecting the nodes together. In this way, no node is considered to be "nearest" another node until every node that could be considered such is available for comparison.

Implementation

The only modification I made to the PRM algorithm was to replace the `addNode`and `growMap`functions with a single `growMap`function that is fully listed below:

```
protected void growMap(int K) {
        // Data structures to help with the algorithm
        List<Vector> configs = new LinkedList<Vector>();
        List<Vector> closestNeighbors;
        List<Vector> validNeighbors;
        ListIterator<Vector> it;
        Vector curr;

        int i = 0;

        // Add the start configuration and end configuration
        configs.add(getStart());
        configs.add(getGoal());

        // Get K free points into a list
        while(i < K) {
          curr = generateFreeConfiguration();
          if (curr != null) {
            configs.add(curr);
            i++;
          } // CONSIDER: putting a checker to ensure we don't loop forever
        }

        // For each free point in the list, get it's closest neighbors
        for (Vector v : configs) {
          closestNeighbors = nearestKNeighbors(configs, v, kValue());
          validNeighbors = new LinkedList<Vector>();
          it = configs.listIterator(0);

          // Remove any non-steerable nearest neighbors
          while (it.hasNext()) {
            curr = it.next();
            if (getEnvironment().isSteerable(getRobot(), v, curr, RESOLUTION)) {
              validNeighbors.add(curr);
            }
          }

          // Add resulting free point and nearest neighbors list into hashmap
          if (closestNeighbors.size() > 0) // But only if it's connected
```

```
            roadMap.put(v, validNeighbors);
        }

        // Some assertions to make sure nothing broke
        assert(roadMap.containsKey(getStart()));
        assert(roadMap.containsKey(getGoal()));

        // roadMap is successfully created.
}
```

A full implementation of this modified algorithm is available in the file `/assignment_motion_planning/MPRMPlanning.java`.

Results

This variation on the PRM algorithm was able to solve the basic Robot Arm problem using 10 times less memory (a graph of 100 configurations vs. a graph of 1000 configurations) and in about one fifth of the time. This proves my concept and shows that this is a viable alternative worth exploring.

However, it is worth noticing this precludes one from extending the search iteratively over time. One must be fairly confident of the number of nodes it will take to solve the problem, as re-running the algorithm entails starting over from scratch.