16.413 Project Description

AUTONOMOUS LUNAR VEHICLE:
As your 16.413 term project you will develop an autonomous system that is capable of executing a lunar logistics mission. The mission is to control an autonomous lunar vehicle to transport cargo between certain sites, and conduct science experiements on the moon surface. The mission requires completing several such activities.

Traveling between sites may require a long drive on the lunar surface, which is full of craters and rocks that your vehicle can not traverse. So, for full autonomy your vehicle will have to plan paths around these obstacles to reach its goal destination.

A list of activities is uploaded from a ground station on earth as a goal expression in the PDDL format. During the day, your vehicle (one of many autonomous vehicles operating on the moon surface) has to autonomously complete all these activities.

Part 2: Building the path planner

Introduction

Your work in building an effective activity planner using a SAT solver was great success, and was well-appreciated in your company. Your hard work has paid off. Given a task including shipping cargo, crude navigation, or science experimentation, your planner figures out a correct sequence of actions that does the job. However, most of the time, the task involves navigating between the sites, for which your robot will need a path planner to actively avoid falling into the giant craters on the moon.

The second step to building your fully autonomous system is to design your own path planner. Your path planner will be given a map of the moon (this map is provided in real-time from a set of satellites orbiting the moon), the position of your robot as well as the goal region (i.e., the site that your robot is trying to reach) in this map. The map will show all the craters and rocks that your robot should avoid collision with.

The motion planning algorithm has to be executed online using the limited computational power on the robot. In fact, you were amazed when told by the experts in your company that the cpu clock rate will only be 50 Hz (orders of magnitude less than what is present on your laptop). It turns out this is the fastest cpu that can run reliablely under the environmental conditions that are present on the moon.

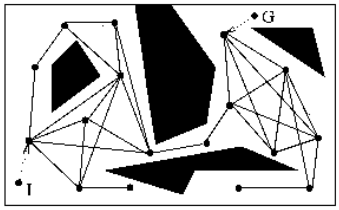


Figure 1: A Probabilistic Roadmap (figure by L. Kavraki)

You have quickly realized that limited computational resources and the need to do online real-time planning make the motion planning problem especially challenging. Doing some research, you find out that the Rapidly-exploring Random Tree (RRT) algorithm is well suited for such path planning problems. You decide to implement an RRT algorithm to do the path planning.

Background

Sampling-based methods in motion planning were pioneered by Kavraki et al. who have introduced the Probabilistic RoadMaps (PRM) algorithm [1]. The PRM algorithm has a very simple philosophy: take a set of N samples from the obstacle-free space, which form the vertex set of your graph. The PRM algorithm then tries to connect the samples with a straight line. Any of these straight lines that is collision-free, i.e., does not go over an obstacle, is added to the graph as an edge. A (random) graph formed in this way is called a roadmap (see Figure 1 for an example roadmap). After the roadmap is constructed, one can use it to check whether two given vertices are connected through a path in the roadmap. If so, there is a collision-free path connecting these two vertices. Usually, one of these vertices is the starting position of the robot, and the other one is the goal point to reach. Notice that once a roadmap is constructed, it can be used repeatedly to solve *multiple* motion planning queries that take place in the same environment.

Rapidly-exploring Random Tree (RRT) algorithm was introduced by LaValle and Kuffner in [2, 3] as an incremental version of the PRM algorithm (mostly to deal with systems that have differential constraints, which is not our focus in this project). The RRT algorithm maintains a tree (i.e., a graph that includes no cycles). Each sample is connected to the nearest neighbor in the tree and the tree is extended towards the sample by a straight line. If the straight line connecting the nearest neighbor and the sample is collision-free, then it is added to the tree as a new edge (see Figure 2 for an RRT resulting from planning for a simple spacecraft). The RRT algorithm was implemented on many experimental platforms and used in countless

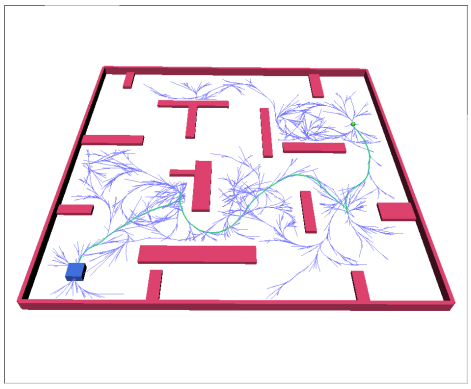


Figure 2: A Rapidly-exploring Random Tree (figure by S. LaValle)

practical applications. In Figure 3, MIT's Urban Challenge entry car running the RRT algorithm for motion planning is shown [4]; if you are curious, you can go take a look at the car, which is usually parked at the Aero-Astro hangar in building 33.

Today, sampling-based motion planning algorithms are a big focus of the cutting-edge motion planning research. Especially, the RRT algorithm has received increasing attention over the last decade. Some references that may give you some ideas on improving your path planner to generate better plans (e.g., those that minimize the path length etc.) are the following references.

Programming Architecture

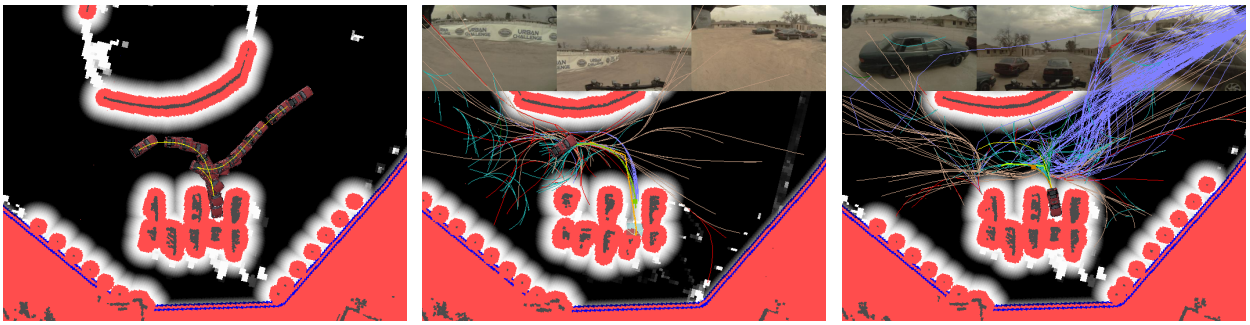
A sketch of the environment that your robot is working in is given in Figure 4. The craters that your robot can not cross are represented by disk-shaped obstacles. A precise representation of the environment is shown in Figure 5, which also shows the robot, the goal region, and the bounding box that the robot is working in. All these concepts are explained in the paragraph below.

Your robot will only navigate in an environment that has a rectangle shape centered at coordinates $(x_{\text{box}}, y_{\text{box}})$; the length and the width of this rectangle will be denoted as $(l_{\text{box}}, w_{\text{box}})$. For safety reasons, you will assume that your robot is a single disk (this disk bounds the actual robot). The radius of this disk will be denoted as R. The initial position of the robot in the map will be denoted as $(x_{\text{bot}}, y_{\text{bot}})$. The obstacles are given as a list. Each obstacle is denoted by its coordinates (x_i, y_i) and its radius r_i. Finally the goal region is a disk given with its coordinates $(x_{\text{goal}}, y_{\text{goal}})$ and its radius r_{goal}. The robot is considered to reach the goal, if the center of the robot is inside the goal region.

The input to the algorithm will be provided as four separate files described below.

- `box.txt` includes one line with four values separated by commas. The values are x_{box}, y_{box}, w_{box}, and l_{box}, respectively.
- `init.txt` includes one line with three values separated by commas. The values are x_{bot}, y_{bot}, and R, respectively.
- `obstacles.txt` includes several lines, one for each obstacles. Each line has three values separated by commas. The values are x_i, y_i, and r_i, respectively, representing the ith obstacle.
- `goal.txt` includes one line with three values separated by commas, The values are x_{goal}, y_{goal}, and r_{goal}, respectively.

Your code should be able to correctly parse these files. You are provided with a set of example files which you can experiment with. These examples are very simple ones. Note that you should generate your own files as stated in the requirements section.



(a) Overall trajectory of Talos.

(b) Going towards the parking spot.

(c) After parked, Talos reversing to its left and going to the exit.

Figure 3: MIT's Darpa Urban Challenge entry vehicle, Talos, executing a parking maneuver using an RRT [4].

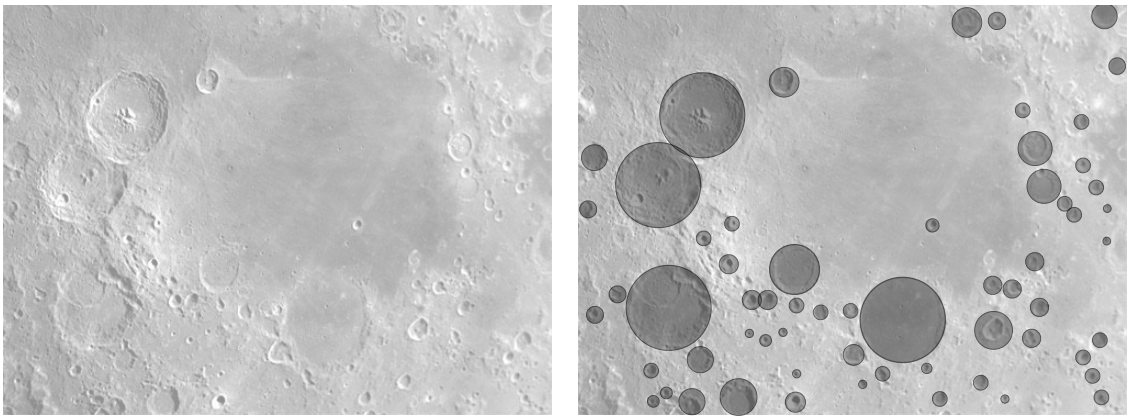


Figure 4: A satellite image of the moon surface is shown on the left. On the right, you can see the obstacles (shown as dark gray disks) detected by the vision-based obstacle detector, which is developed by the robotic perception department of the company. The detection algorithm is doing a reasonably good job in detecting many craters on the moon surface.

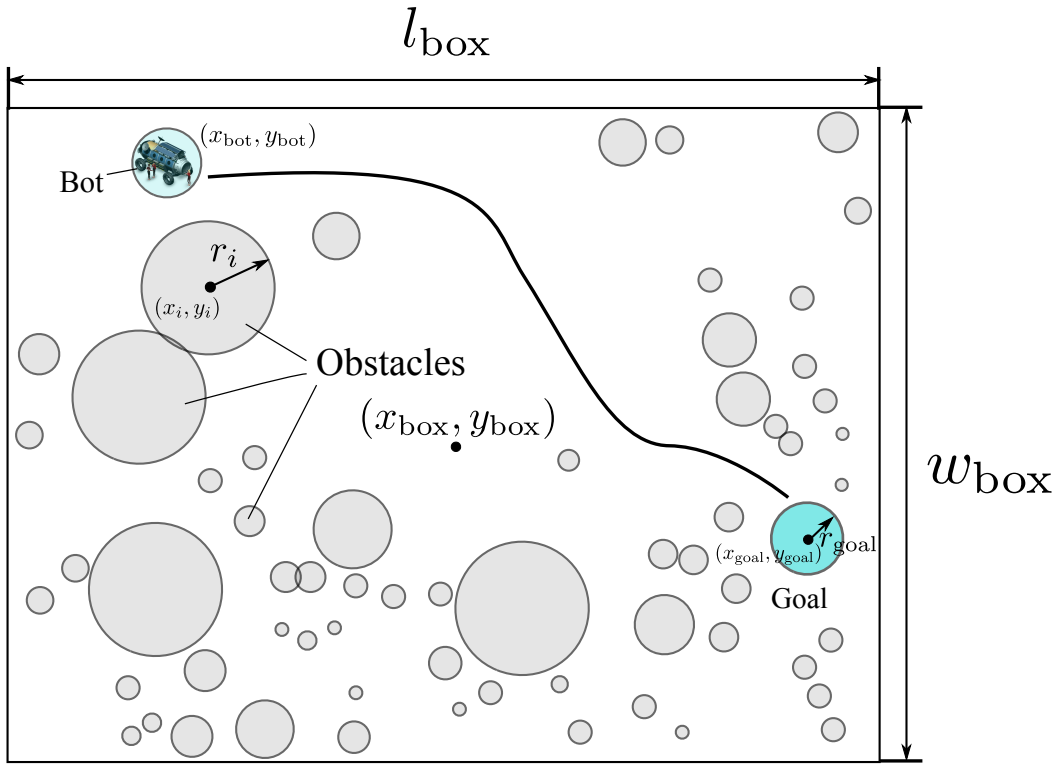


Figure 5: The bounding box, robot, obstacles, and the goal region are shown in one figure. The bounding box that the robot is working in is represented by the tuple $(x_{\text{box}}, y_{\text{box}}, l_{\text{box}}, w_{\text{box}})$, where $(x_{\text{box}}, y_{\text{box}})$ are the coordinates of the center of the bounding box, l_{box} is the length of the bounding box, and w_{box} is the width of the box (all of which are shown in the figure). The coordinate of the robot is shown in $(x_{\text{bot}}, y_{\text{bot}})$. The radius of the disk that encapsulates the robot is R (not shown in the figure). Each obstacle is represented by the triple $(x_i), y_i, r_i)$ where (x_i, y_i) is the coordinates of the center of the obstacle whereas r_i is its radius (shown for one of the obstacles in the figure). Finally, the goal region is represented by a triple $(x_{\text{goal}}, y_{\text{goal}}, r_{\text{goal}})$ as shown in the figure.

Collision Checking

There are many ways to check whether a straight line connecting two configurations of the robot is collision-free, i.e., the robot following this path does not collide with obstacles. You should come up with a collision checking algorithm that is computationally efficient. Note that in this part you can implement one of a variety of approaches that range from discretizing the path and checking collision with each obstacle to efficient maintenance of the obstacles in a kd-tree structure and using analytical techniques with the idea of the configuration space in mind to check collision of a path using a few vector additions/multiplications.

In your project, you should implement a collision checking method, describe this method in your report. You should describe why you have chosen that particular method and what assumptions have led you to this decision. Such assumptions may include, e.g., expecting no more than 100 obstacles (i.e., the number of obstacles is few) or more than a million obstacles (too many obstacles).

Nearest Neighbor Search

The original RRT algorithm attempts to connect each new sample to the nearest node in the tree. However, several researchers have proposed attempting to connect the new sample, for instance, to the node that has the lowest cost [5], or evaluating distance in a weighted Euclidean space [3].

You should design an effective nearest neighbor heuristic, and describe this heuristic in your report. This heuristic might very well be the original nearest neighbor heuristic that attempts to connect the new sample to the closest, or a different one such as those described above. In your report, you should describe the nearest neighbor heuristic that you have implemented, and you should justify your selection of this heuristic. If you have implemented a heuristic different than the usual nearest neighbor, then you should compare your heuristic with the usual one and present your experimental results.

Random Sampling

Most RRT algorithms uniformly sample the obstacle-free portion of the region to generate new samples. However, in most cases, the performance of the algorithm can be improved drastically by intelligent sampling strategies. One such strategy is to sample the goal region, e.g., once every hundred times to make the algorithm run faster. Another one is to set the sampling distribution so that samples are drawn from regions which you think would generate better paths to make the algorithm generate lower cost paths.

You should design a sampling heuristic (which may very well be uniform sampling) and describe your heuristic in your report. You should justify the selection of your heuristic. If you have used a heuristic other than uniform sampling, then you should compare your heuristic to uniform sampling in experiments.

Requirements

You should submit your code together with a maximum 8 page report (4-5 page recommended). Your report should describe the approach you have taken to implement collision checking, nearest neighbor heuristic, and your sampling strategy. You should evaluate your algorithm on three problem instances that you have created. One of these instances should include only a few obstacles, the second instance should include a moderate size of obstacles (such as 20 - 50), and the last one should include several obstacles (such as 1000). You can generate a random set of obstacles (each obstacle will be assigned a random coordinate and a random radius), e.g., using matlab to create the last problem instance. You should present examples of the path that your algorithm returns. You should also show the tree of collision-free paths maintained by your algorithm after it is finished[1].

[1] One way to plot the path and the tree is to generate a file that contains all the nodes and edges of your graph, and load these files to matlab along with the obstacles to generate a consistent plot. You are given a simple matlab .m file that does so. However, you are welcome to use your favorite visualization tool

Fast Execution *(extra credit)*

Ideally, your algorithm should find a solution fast, and your results should be repeatable. After building your planning algorithm, you should run it on two problem instances (that you have designed yourself) 100 times and plot a histogram of the time that the first solution is found to benchmark your algorithm. One of the problem instances should represent an environment with few obstacles, whereas the second one should represent an environment cluttered with obstacles.

Optimization *(extra credit)*

In most applications, it is important to generate not only a path that reaches the goal, but a path that does so while incurring minimum cost. Ideally, your algorithm should be able to generate paths that converge to an optimal path (e.g., the shortest path). You can use several heuristics to generate paths that are close to optimal (see, e.g., [5]) or you can use an algorithm that is provably optimal [6].

After building your algorithm you should run it on two problem instances (that you have designed yourself) 100 times and show the cost achieved by your algorithm after 5 seconds of planning time (for each run the algorithm is given 5 seconds of computation time on your laptop). You should show these results in a histogram. You should also show the average cost as a function of the computation time. You should generate the same histogram for a base-line RRT algorithm that does not use the heuristics or optimization methods that you have employed in this part. You should compare the two histograms, and comment on the improvement. One of the problem instances should represent an environment with few obstacles, whereas the second one should represent an environment cluttered with obstacles.

Branch and bound *(extra credit)*

When the objective is to generate optimal paths, once a feasible solution is found the cost of this solution can be used to prune paths that can not possibly be optimal using a technique usually called branch-and-bound.

The branch-and-bound technique resembles the A* algorithm and works as follows. Assume that you have an admissible heuristic function $h((x, y))$ that represents cost to go from a particular coordinate (x, y). For instance, the Euclidean distance from (x, y) to the nearest point in the goal region (ignoring the obstacles) would constitute such an heuristic. Given a node in your tree that is situated at the coordinate (x, y), let $f((x, y))$ denote the cost to get to this node along the edges in your tree. Say you have found a path that reaches the goal with cost c. Now you can delete all nodes in the tree that have cost larger than $f(x, y) + h(x, y)$.

You can implement this heuristic to make your algorithm more efficient. If you do so, you should compare the performance of your algorithm with the one that has no branch-and-bound heuristic in 100 runs. For each test you should show a histogram of the cost of the solution returned by your algorithm at the end of 5 seconds of computation. You should run the algorithm on two problem instances (that you have designed yourself). One of the problem instances should represent an environment with few obstacles, whereas the second one should represent an environment cluttered with obstacles.

References

[1] L. Kavraki and J. Latombe. Randomized preprocessing of configuration space for fast path planning. In *IEEE International Conference on Robotics and Automation*, 1994.

[2] J.J. Kuffner and S.M. LaValle. RRT-connect: An efficient approach to single-quert path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2000.

[3] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20(5):378–400, May 2001.

[4] Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J.P. How. Real-time motion planning with applications to autonomous urban driving. *IEEE Transactions on Control Systems*, 17(5):1105–1118, 2009.

[5] C. Urmson and R. Simmons. Approaches for heuristically biasing RRT growth. In *Proceedings of the IEEE/RSJ International Conference on Robotics and Systems (IROS)*, 2003.

[6] S. Karaman and E. Frazzoli. Incremental sampling-based algorithm for optimal motion planning. In *Robotics: Science and Systems (RSS)*, 2010.