Technical Design

Tavish Mankash

Abstract

This project will create a navigation algorithm to traverse complex 3D terrain whose 3D maps are known, which could be pratically used in robotics. We will use complex, rugged, real-world terrain data to test the algorithm's effectiveness and efficiency. As a model, we will use USGS' high resolution scan of the area surrounding MDRS (Mars Development Research Station) in Utah, USA. The algorithm will be designed to find optimal paths for a rover to navigate from one point to another while avoiding obstacles and minimizing travel time or energy consumption. We will also try to optimise for minimum turbulance between two similar paths, as robots are more likely to be damaged, if traversing such paths. The project will involve data preprocessing, algorithm development, and performance evaluation.

1 Data

We use a model 3D map around MDRS to build our algorithm upon. This gives us a complex, real-world terrain to test our navigation algorithm. The data is sourced from USGS' high-resolution LIDAR scans, which provide detailed elevation information. The dataset includes various terrain features such as hills, valleys, and rocky outcrops, making it ideal for testing the robustness of our navigation algorithm.

- USGS LiDAR Explorer: `http://tiny.cc/4iou001`
- This scan contains the MDRS terrain (and many others across Utah): UT Southern QL1 2018
- 3D scans (LAZ folder): `http://tiny.cc/2iou001`
- The entire dataset is 90–220 GB, as it also contains many other scans accross Utah. Instead of downloading everything, we use metadata files to download only files within a 6–7 km range around MDRS.

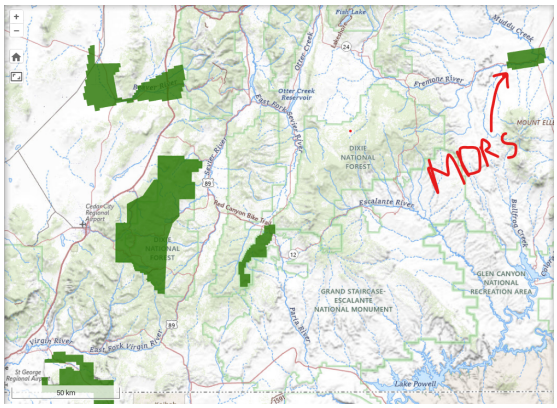

Figure 1: UT Southern QL1 2018 LIDAR scan (MDRS area is marked).

2 Algorithm

2.1 Preprocessing

- USGS gives a lot of seperate point clouds in LAZ format. Each scans' location in lat, long is given in its metadata file.
- We first voxelise the point-cloud, and then stitch them all together to create a single 3D grid of voxels representing the terrain.
- This file is stored.

2.2 Path-Planning

A-star is a graph search algorithm that finds the shortest path between two points by intelligently exploring the most promising routes first. It works like a hiker who not only considers the distance already traveled, but also estimates how far they still need to go to reach the destination. At each step, the algorithm picks the next location that minimizes the sum of the actual cost to reach it plus the estimated cost to the goal. This combination of past cost and future estimate makes A-star both optimal (guaranteed to find the shortest path) and efficient (explores fewer nodes than simpler algorithms like Dijkstra).

The cost is mathematically defined as:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$ is the cost to reach the current node n from the start node.
- $h(n)$ is the estimated cost to reach the goal from the current node n. h must be admissible, meaning it never overestimates the true cost to reach the goal.

2.3 Cost Function

For the h cost, we can use the Euclidean distance between the current node and the goal node:

$$h(n) = \sqrt{(x_{goal} - x_n)^2 + (y_{goal} - y_n)^2 + (z_{goal} - z_n)^2}$$

However, the g cost is where the majority of our customisation will go. Let's come up with a series of formulae intuitively before we explore them in practice. Just for fun.

Definitions:

Say, our robot is at position (x_1, y_1, z_1) and wants to move to position (x_2, y_2, z_2). Let the vector Δs be a unit vector defined as:

$$\Delta s = (x_2 - x_1, y_2 - y_1, z_2 - z_1) / \| (x_2 - x_1, y_2 - y_1, z_2 - z_1) \|$$

Δr is the projection of Δs onto the XY plane:

$$\Delta r = \Delta s.(1, 1, 0)$$

Δz is the projection of Δs onto the Z axis:

$$\Delta z = \Delta s.(0, 0, 1)$$

Δr_p is perpendicular to Δr in the XY plane:

$$\Delta r_p = \Delta r \times (0, 0, 1)$$

Approach 1:

We would like the rover/robot to not climb steep slopes, and in general, not like climbing or descending. Therefore, we can add a penalty for elevation change:

$$h(n) = \sum_{i=start}^{n} (\frac{\Delta z_i}{\Delta r_i})^n$$

Where n is emperically determined. Higher values of n penalise steep slopes more. It is robot dependent.

Approach 2:

The above formula does not care about moving tangentially to a sloping terrain. A robot would find this more challenging than moving on a flat terrain. Therefore, we can add a penalty for moving tangentially to a slope:

$$h(n) = \sum_{i=start}^{n} (\frac{\Delta z_i}{\Delta r_i})^n 1 + (\frac{\Delta z_i}{\Delta r_{pi}})^n 2$$

Where n_1 and n_2 are emperically determined and robot dependent.

Approach 3:

In trigonometry, we know that the perpendicular side over the base side is equal to the tangent of the angle. Δz is the perpendicular side, and Δr is the base side. Therefore, we get the tan angle. This means that for angles less than 45 degrees, the penalty is further reduced by exponentiation. For angles greater than 45 degrees, the penalty is increased by exponentiation. This is likely not ideal behaviour since it's relatively rare to have slopes greater than 45 degrees in nature. We can correct for this with:

$$h(n) = \sum_{i=start}^{n} (\alpha_1 * \frac{\Delta z_i}{\Delta r_i} + \beta_1)^n 1 + (\alpha_2 * \frac{\Delta z_i}{\Delta r_{pi}} + \beta_2)^n 2$$

Approach 4:

We can also think of an alternate approach.

$$h(n) = \sum_{i=start}^{n} \Delta r * (\Delta z)^n 1 + \alpha * \Delta r_p * (\Delta z)^n 2$$

Throughout this project, we'll come up many new ideas/formulas. We will test out each of these intuitively derived formulas, implement work other people have done in this area, and see which works best for our use-case.