Spring 2018

Jet Propulsion Laboratory: Martian Autonomous Routing System Research Document

M.A.R.S Team
University of Colorado at Boulder

Jet Propulsion Laboratory:

Martian Autonomous Routing System Research Document

by

Ross Blassingame

Robert Ballard

Shawn Polson

Chandler Samuels

Dean Moser

Josh Jenkins

B.S., University of Colorado at Boulder, 2018

B.A., University of Colorado at Boulder, 2018

A research paper designed to compliment

the Martian Autonomous Routing System.

Designed for NASA Jet Propulsion Laboratory

during the Senior Capstone Projects at the

University of Colorado at Boulder 2018

Contents

1. Introduction

The Martian Autonomous Routing System (M.A.R.S.) is a suite of software written by Ross Blassingame, Shawn Polson, Robert Ballard, Josh Jenkins, Chandler Samuels, and Dean Moser at the University of Colorado Boulder as a part of their Senior Capstone Software Design Project. The Senior Capstone is a year-long course where students form teams of five or six, pair with a sponsor, and build software based on the sponsor's initial proposal. Our team paired with NASA's JPL, specifically Marcel Llopis, to build the pathfinding software. In the initial project proposal, it was required that the software be able to ingest:

- An elevation map of a section of the Martian terrain
- The rover start coordinates
- The rover end coordinates

It was also required that the software be able to find, if they exist:

1. A path from the start coordinates to the end coordinates, assuming that (1) the rover cannot climb hard climbs, (2) the rover cannot descend steep descents, and (3) that the rover cannot see too far ahead—the rover is essentially discovering the terrain as it moves.
2. A path from the start coordinates to the end coordinates, assuming that (1) the rover cannot climb hard climbs, (2) the rover cannot descend steep descents, and (3) that the rover is aware of all of the terrain—we can assume we have preloaded the entire elevation map into the rover's internal memory.

The first option is an exploration in which there could be backtracking involved because the rover might try paths and end up in places where it cannot continue, whereas the second option involves the calculation of the ideal path upfront. In these initial project specifications, paths could be output as an ordered list of coordinates.

The central ability of M.A.R.S., then, is to discover constrained paths in elevation maps. Elevation maps in our context are specifically GeoTIFFs. M.A.R.S accomplishes these tasks by leveraging the GeoTools toolkit (http://docs.geotools.org/stable/userguide/extension/brewer/index.html), which is an open source Java library that provides tools for geospatial data, in conjunction with other Java code written from scratch by us. Using elevation data stored in the elevation maps, M.A.R.S. is able to find paths by employing pathfinding algorithms. Ultimately, the purpose of this is to aid in the design and planning of future rover missions by providing proposed paths, and to simulate a hypothetical future rover which is capable of autonomously maneuvering itself across the Martian surface. This paper's primary purpose, aside from describing M.A.R.S., is to competitively analyze the pathfinding algorithms employed in our software with the purpose of drawing conclusions about their respective advantages and disadvantages in guiding a rover.

Upon completion of the initial goals above, several other features were implemented, such as a partial GUI, an expanded user-interface, and improved output options. One of the goals that was simply deemed as "nice to have" in the initial project proposal was to output the total elevation gain and distance covered along with the route. We were unable to complete this stretch goal, but were able to implement all other goals and stretch goals defined in the proposal. Finally, a comprehensive test suite was built in addition to the main project, which was maintained at 70% minimum coverage in order to minimize bugs and other problems within the software.

2. Summarized Packages

Before discussing individual algorithms and in order to better describe what M.A.R.S. is, it is pertinent to describe in some depth the software packages that comprise the system. The source code of M.A.R.S. is separated into seven main packages: *rover, algorithm, coordinate, map, out, ui, and views.*

2.1. Rover

The *rover* package contains classes related to planetary rovers. We sought to create a simplistic representation of a rover, so in this context, a rover is an object that maintains just five things: a current location, a starting location, a goal location, a field of view, and a maximum allowable slope. These parameters are used as rules in whichever algorithm drives the rover. Algorithms search for paths from a rover’s start location to its goal location, they will only consider paths that are within the rover’s field of view, and they will never drive the rover over terrain that is too steep (terrain which surpases the rover’s maximum allowable slope).

This package holds a parent class named “Rover” from which all rovers inherit. This abstraction allows for rovers to be interchangeable; any number of rovers with different specifications could be created and used without having to modify the surrounding code base. At present, however, only one subclass exists and it is named “MarsRover.” Beyond what “MarsRover” inherits from “Rover,” it stores an extra method that calculates the angle between two coordinates and another method that determines whether or not it can traverse from its current location to a new one; this information is fed into its driving algorithm during runtime.

2.2. Algorithm

The *algorithm* package contains classes related to pathfinding algorithms. The algorithms themselves are self-contained classes that perform all the logic of an algorithm, and all algorithm classes are subclasses of a parent “Algorithm” class. This abstraction allows the system to mix-and-match algorithms at runtime without having to modify any of the existing code base. Six distinct algorithms are implemented by the classes in this package: A* Search, Best First Search, Breadth First Search, Dijkstra’s Shortest Path Algorithm, Greedy Search, and IDA* Search. Furthermore, each of these six algorithms have a “limited” and an “unlimited” implementation, with the difference between the two being the rover’s field of view.

In a “limited” implementation, the algorithm’s rover can only “see” portions of the map that are within a given radius from it. From a conceptual standpoint, this means that to find a path to its destination, the rover must physically explore the map, potentially backtracking if it ends up in a bad spot. In an “unlimited” implementation, on the other hand, the rover’s field of view is unlimited. Conceptually, it stores the entire elevation map in its memory so that it can discover an optimum path to its destination before ever moving.

2.3. Coordinate

The *coordinate* package contains classes related to coordinates—locations in 2D space. The parent class “Coordinate” is simply an object that wraps an (X, Y) ordered pair. The real utility of this abstraction comes from the subclasses of “Coordinate;” four M.A.R.S. algorithms make use of specialized Coordinate subclasses to handle heuristics and other calculations required by the algorithms. These subclasses are “AStarCoordinate,” “BestFirstCoordinate,” “DijkstraNode,” and “GreedyCoordinate.” It is a memory optimization to put special data/logic in coordinate subclasses, because elevation maps are 2D grids of discrete locations and algorithms generate new “Coordinate” objects for each grid location they visit. Without these subclasses, algorithms would require a separate data structure—in addition to the generated Coordinate objects—in which to store information produced by the algorithms. So the subclasses eliminate the need for such extra data structures.

An AStarCoordinate stores the cost to arrive at it from the starting location, the (Euclidean) distance from it to the goal location, its “parent” coordinate (the coordinate that was visited directly before it), and special logic used to sort lists of AStarCoordinates according to these heuristics. A BestFirstCoordinate stores its parent coordinate but no other special information beyond that. A DijkstraNode stores the cost to arrive at it, its parent coordinate, and whether or not it is in a “visited set,” which is necessary information for the Dijkstra algorithm (and Greedy Search for that matter). Lastly, a

GreedyCoordinate stores whether or not it is in a visited set and what cardinal direction the rover was facing when it arrived at it.

2.4. Map

The *map* package contains classes related elevation maps. Elevation maps in M.A.R.S. are specifically GeoTIFFs, which is a map image format that represents terrain as a 2D grid of discrete locations with an elevation stored at each location (i.e., pixel). The parent class in this package is "TerrainMap," but similar to the *rover* package, only one subclass exists and that subclass is "GeoTIFF." Having this abstraction allows for the possibility of adding support for additional map types in the future. M.A.R.S. elevation maps are required to contain methods for providing information about themselves, including their height and width (in pixels) and the elevations at specific pixels. Coordinates in the map can be represented either as an (X, Y) ordered pair of pixels or as a pair of latitude/longitude values, and maps hold the methods for converting between these two representations (because the conversions are map-specific). It is important to note that the discretization of the elevation values makes it so that these maps only approximate the terrain. They are only as accurate as the pixel density of the image so it is not a perfect representation.

2.5. Out

The *out* package contains classes related to the system's output. Each time a user runs M.A.R.S., they will discover exactly one path for one rover from its start to goal coordinates, using one specified algorithm, and the output will be the discovered path. The parent class in this package is "Output" and its only requirement for subclasses is that they be given a list of "Coordinates" to output. Three subclasses of "Output" currently exist: "TerminalOutput," "FileOutput," and "MapImageOutput." "TerminalOutput" is the simplest output type and just prints a discovered path to the terminal as a list of (X, Y) ordered pairs. "FileOutput" outputs essentially the same information as "TerminalOutput" except that the ordered pairs are put into a CSV file instead of being printed to the terminal. "MapImageOutput" is unique because it actually displays an image of the map in a new window and the discovered path is drawn on the map as a red line. Users may request any combination of these output types at runtime.

2.6. UI

The *ui* package contains classes related to the system's user interface. At a high level, the purpose of the M.A.R.S. user interface is to allow the user to specify which map they want to traverse, the parameters for their rover—start coordinate, goal coordinate, and maximum allowable slope—, the algorithm that will guide the rover, and their desired output type(s). At the time of this writing, M.A.R.S. is primarily a command-line application, so the user sees a command-line interface in which they are prompted one question at a time until the system has all needed information and can discover their path (if one exists). Users are given the option to specify some or all answers to the prompts upon first calling M.A.R.S. by passing in command-line arguments. If the user provides any such arguments, they will be prompted for any information that was not provided as an argument. This setup is useful because the prompts allow inexperienced users to understand the system while the optional command-line arguments allow for the possibility of incorporating M.A.R.S. into a broader workflow, such as a bash script.

2.7. Views

The *views* package contains all classes related to graphical user interface (GUI) components. At the time of this writing, the only GUI component that exists in M.A.R.S. is the map image output type. All components were built with Java Swing (https://docs.oracle.com/javase/8/docs/technotes/guides/swing/), which is a GUI widget toolkit for Java. Having this *views* package was a design decision that provides separation between the front-end and back-end components of the system, so that they do not interfere with one another.

3. Caveats

M.A.R.S. is as much a scientific/engineering project as it is a software engineering one, and because of this we should be upfront about the caveats of our software. There are many caveats, and we detail them here.

To begin, GeoTIFFs do not perfectly model the elevation of a planet. M.A.R.S's slope calculation, a crucial part of M.A.R.S, compares the elevation data of adjacent pixels and performs some basic (and some not-so-basic) trigonometry to estimate the terrain's slope within the area. While this gives a good estimate of the slope between those two points, it cannot guarantee complete accuracy due to the possibility of dramatic slope changes within an individual pixel in the map (which often represents hundreds of square meters). Pixels in different maps therefore represent different sized areas depending on the resolution of the file being used. GeoTIFFs that have large areas represented by a single pixel are more likely to experience inaccuracies in slope calculation.

Another caveat is that there are probable inaccuracies when converting between pixel and latitude and longitude coordinates (a unit conversion which happens often in M.A.R.S). Converting from latitude/longitude to a pixel is straightforward; since pixels contain many latitude/longitude points, we just return the pixel that contains the given latitude/longitude point (assuming it exists). When converting from pixels to latitude/longitude, however, we always return the latitude/longitude point at the exact center of the pixel. This means that one can start with a latitude/longitude coordinate, convert it into a pixel, and then convert that pixel back to latitude/longitude and end up with different coordinates than what they started with. In fact, the only case where this will not happen is if the starting latitude/longitude point happens to rest exactly in the center of a pixel.

A significant portion of this project was building the software. The other side of it was to performance test the algorithms employed by the software, and there are limitations in our approach to doing that. An in-depth explanation of our methods for this paper can be found in the "Experiment" section below, but we will touch on the caveats here. The first to note is that when using M.A.R.S. to performance test algorithms, all run times will differ based on the specifications of the device running M.A.R.S. The distance of the discovered paths will be consistent across all devices, however. This is an intuitive fact but it is worth mentioning; a supercomputer and a normal laptop would both produce the same path because the algorithm would be the same, but the supercomputer would crunch the numbers much faster than a laptop. Another limitation of our performance testing is that we only performance tested with one route. We did our best to choose an interesting and informative set of start and end coordinates, but it is important to note that we are drawing conclusions from a small sample size. A more robust performance test would gather more data.

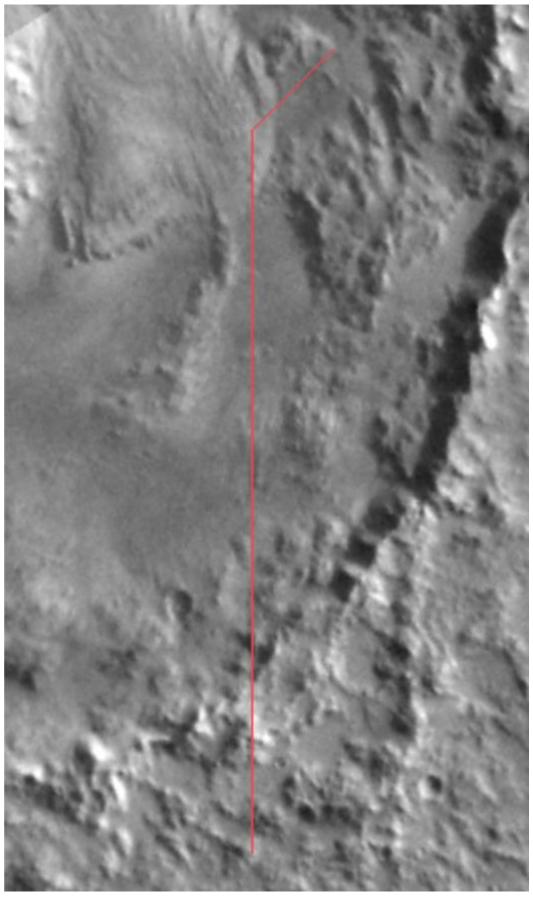
A final caveat is that M.A.R.S. does not support GeoTIFFs whose rasters take up more memory locations than Java's `Integer.MAX_VALUE` (roughly 2^{31}) limit. This is noteworthy because it turns out that most GeoTIFFs representing an entire planet do exceed this limit, meaning that in practice, our software only supports maps of *sections* of planets. The limit is imposed by the GeoTools library, which handles all the parsing of our maps, so fixing that problem was beyond the scope of this project.

4. Experiment

As stated above, aside from generally describing M.A.R.S., this paper is a competitive analysis of the pathfinding algorithms it employs. We have described M.A.R.S. in sufficient depth, so we will now move on to the analysis. We ran one main experiment to compare our algorithms, and our methods are detailed here.

The objective of this experiment is to determine differences in how a variety of algorithms perform against the problem of pathfinding on a elevation map. In order to accomplish this, we need to determine by which conditions the algorithms should be tested against, so we can compare the metrics from those tests across the algorithms' results. The conditions that need to be set for this comparison include the elevation map used, the starting and ending coordinates on that map, the possible slopes that the rover could traverse, and the rover's field of view ("FOV" hereafter).

For the map, we chose a segment of the Mars Global Digital Image Mosaic 2.1 map (MDIM) (https://astrogeology.usgs.gov/maps/mdim-2-1). Our section was generated from a combination of Viking probe and other mission observations, and it contains the Aeolus region of Mars. We chose this map because of the visual quality, as well as the noteworthiness of the region, which contains the landing sites of Spirit and Curiosity. We also opted to only use this region instead of the full global map due to the memory usage limitations mentioned above. MDIM2.1 has a pixel size of a 231m^2 close to the equator, and the Aeolus region is indeed close to the equator of Mars, so that is the pixel size for this map. For starting and ending coordinates, we chose a route near the landing site of Curiosity where it escapes an edge of a large crater. The coordinates are [138.73, -5.19] to [138.57, -6.75], which are separated by 92.9km in straight-line distance. The route can be seen in the image to the right, connected by the red line (which, as an aside, was generated using A* with a 90º slope). We chose this route both for its proximity to meaningful features, as well as all algorithms being able to navigate it, and most algorithms doing so in a reasonable amount of time. The slopes chosen for testing were a range from 8º to 32º, in increments of 2º. 8º was chosen as it was the most difficult slope the algorithms could use on this route, and 32º was chosen based on a combination of correspondence with our sponsor, Marcel Llopis, as well as observations of routes becoming trivial beyond that slope. Finally, a range of FOVs from 2.31km (or 10 pixels) to 7.16km (31 pixels), in increments of 1.61km were chosen. These were



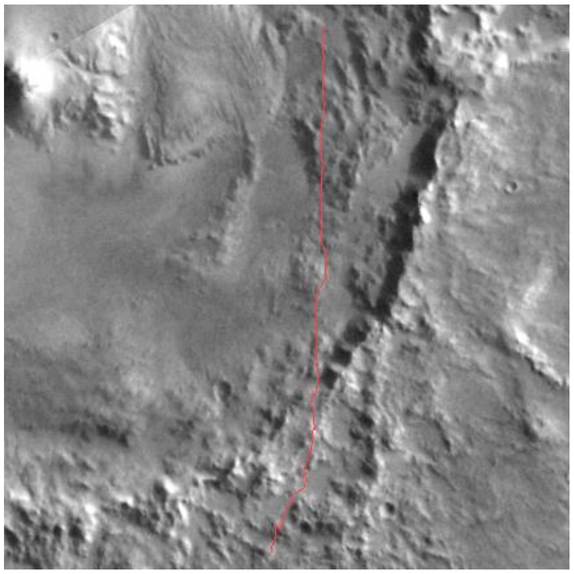
chosen primarily due to algorithm performance, as FOVs below 2.31km caused computation time to take too long to reliably test on our machine, and FOVs above 7.16km became increasingly trivial.

The metrics we decided to track were the length of a generated route for a given rover configuration, and the time taken for that route to be generated. These metrics were tested across the different slopes and FOVs. To ensure a more trustworthy value for the time taken (as well as to confirm the route decided upon by each algorithm), each trial, defined by what algorithm being tested against what slope and FOV, was tested 10 times, and the resultant time taken was the average of those trials. For limited algorithms, whose routes would vary by FOV, the overall route length for one given slope would be defined as the average of the four FOVs tested: 2.31km, 3.93km, 5.58km, and 7.16km. In this way, for each rover configuration, we could know both which algorithms were faster or slower in certain situations, as well as which algorithms did the best job in each situation.

Data was obtained using a combination of a custom testbed generating rovers and running tests, combined with Java's internal timing library, as well as traditional spreadsheet software like Microsoft Excel. The above specifications were set, and the testbed code repeatedly ran each algorithm against each possible combination of slope and FOV of those we chose to test. At the end of each trial, the average time and route length were recorded. The full results were output in CSV format, and the results in that file were analyzed using spreadsheet software. All the trials were run on the same machine, using a FX-6350 processor and 8GB of RAM, to ensure that there were no discrepancies due to variance in hardware used.

Finally, to clarify the terminology used below, in the "unlimited" implementation of an algorithm, the algorithm is not constrained by any FOV and will therefore scan the entire elevation map to find a path from the start coordinate to the goal before ever moving the rover. As a result, these discovered paths tend to be more optimal than those of the "limited" implementations. In the "limited" implementation of an algorithm, the algorithm is constrained by the rover's FOV. They therefore alternate between phases of scanning the map and moving the rover until the goal is found.

5. Algorithm Analysis



5.1. Greedy Algorithm

The Greedy algorithm is an aggressive algorithm that relies entirely on its heuristic in order to find a route to the finish. Unlike the other algorithms, it only takes into account the direct neighbors, and attempts to choose the one closest to the finish. This means that the algorithm is very quick, since there are very few calculations actually performed before taking a movement, with the most challenging routes taking only roughly 33ms total to calculate, and the shortest taking a mere 2ms. It also means that the algorithm is relatively memory-efficient, since it only needs to track what nodes it has visited, with a worst case of storing a copy of every possible node in a map. As such, its Big-O complexity is `O(n)`. The downside to this approach is that the routes generated are generally much less efficient than those generated by other algorithms. On average, compared to the 92.9km that separate the start and end coordinates, the Greedy algorithm produced routes that took an additional 2.31–4.62km with easier routes, and 23.1-69.3km with difficult routes (e.g. the algorithm got stuck and had difficulty finding a way out). There generally seems to be a point where the slope becomes difficult enough that the Greedy approach of not accounting for further surroundings causes it to have increasingly challenging problems, and the quality of the routes the algorithm produces degrade severely. The Greedy algorithm is also unique in that it is not impacted by FOV. Since it will always account for direct neighbors, regardless of FOV, it will always functionally work at a FOV of 2.31km (i.e., one pixel).

5.2 A* Algorithm

The A* algorithm is an efficient path-finding algorithm which considers many possible paths to the goal location, and ultimately chooses the one which incurs the smallest cost. *Cost* could mean many different things, but in our case, it means least distance travelled. At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes

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

where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic—Euclidean distance, in our case—that estimates the cost of the cheapest path from n to the goal. The Big-O complexity of A* is $O(b^d)$ in the worst case, where b is the branching factor and d is the distance of the shortest path. In practice, however, A* will perform much better than this because of its heuristic.

5.2.1 Unlimited

A* generally finds shorter paths as its allowable slope goes up, owing to less obstacle avoidance. When constrained by an 8º slope, it found a 104.64km to find a path from the start to the goal. When that allowable slope increased to 32º, it took 94.25km, which is a 9.9% decrease in distance. On average across all allowable slopes, unlimited A* took 96.44km to get from start to finish. With respect to time, it generally takes less time to find a path as its allowable slope goes up. At an 8º slope, it took 3.56 minutes to find a path. At a 32º slope, on the other hand, it took 9.06 seconds, which is a 95.8% decrease. On average across all slopes, it took 49.1 seconds to find a path. From this data, we conclude that when using A* with an unlimited FOV to guide a rover, increasing that rover's maximum allowable slope will greatly improve its performance.

5.2.2 Limited

In the "limited" implementation of A*, the algorithm is constrained by the rover's FOV. It therefore alternates between phases of scanning the map and moving the rover. As we've written it, it first starts by scanning the area within the rover's FOV for the goal. Assuming the goal is not found in this first scan, it then moves the rover to the coordinate in that area with the least cost (often the spot closest to the goal). Having arrived at the best spot within the rover's initial FOV, the algorithm runs itself again, finding another coordinate within its FOV with the least cost, and this process repeats until the goal is found and a path to it is constructed. As a result, its discovered paths tend to be noticeably less optimal than those of the "unlimited" implementation—which is to be expected.

Much like its unlimited counterpart, it generally finds shorter paths as its allowable slope goes up, again owing to less obstacle avoidance. When constrained by an 8º slope, it found a 202.64km path from the start to the goal. When that allowable slope increased to 32º, it took 95.34km, which is a 53% decrease. On average across all allowable slopes, limited A* took 117.49km to get from start to finish. With respect to time, limited A* actually tended to spend more time finding its paths as the allowable slope went up. This is because increasing the allowable slope gave the algorithm more potential paths to consider. At an 8º slope, it took 22.15 seconds to find a path. At a 32º slope, it took 38.473 seconds, which is a 73.7% increase in time spent. On average across all slopes, it took 32.328 seconds to formulate the path.

When considering the rover's FOV, the algorithm found shorter paths as the FOV increased. With a FOV of 2.31km, it found a path that was 140.79km long. When the FOV increased to 7.161km, that number went down to 104.11km, which is a 26.1% decrease in distance. Adjusting the FOV also affected the time taken to calculate these paths. When the FOV was 2.31km, it took 21.22 seconds. That number increased to 47.7 seconds with a FOV of 7.161km, which is a 124.8% increase. Again, this appears to be the result of the algorithm having more possible paths to consider. On average across all FOVs it took limited A* 32.328.2 seconds to find the path. From this data, we conclude that when using A* with a limited FOV to guide a rover, extending its FOV will result in overall better performance—at the cost of longer calculation times.

5.2.3 Recursive

The data above came from a non-recursive implementation of A*. We also did write a recursive implementation, and it ran faster than the non-recursive version (88.7% faster on average, in fact). But because the recursive version quickly fills up its machine's stack (a problem we couldn't get around because M.A.R.S. is written purely in Java), the algorithm could only find relatively short paths. If the start and end coordinates were sufficiently far away, then, the program would break with a stack overflow error. This was the case for the coordinates used in this experiment, so we will not include results here.

5.2.4 Unlimited vs. Limited

The longest path taken by unlimited A* was 104.64km, whereas the longest path taken by limited A* was 202.64km. This means that in the worst case, limited A* discovers paths that are about 93.7% longer than those of its unlimited counterpart. At best, the shortest path found by unlimited A* was 94.25km and that of limited A* was 93.32km. That is only a 1% difference, so under optimal conditions, the two versions do produce comparable paths. With regard to the time taken to calculate those paths, however, unlimited A* took 49.106 seconds on average to calculate a path, but limited A* took 32.328

seconds on average, which is actually a 34.2% improvement in average calculation time. Given all this, guiding a rover with A* under a limited FOV can be a reasonable choice for rover designers operating under significant memory constraints, provided that its FOV is close to 7.161km. Removing the FOV constraint from the algorithm does yield better results overall, however.

5.3 Breadth-First Search

Breadth-First-Search is a traversing algorithm that explores every node layerwise. This means that every time the algorithm visits a node, it scans for possible neighbors to traverse. When it finds a neighbor that can be visited, it stores the neighbor node into a queue data structure. After all neighbor nodes are visited, the algorithm will then dequeue the next neighbor node from the queue and scan its neighbors. This process repeats until it reaches its destination. Since the algorithm does not rely on a heuristic, the algorithm is remarkably slower than more advanced algorithms like A*. Its Big-O complexity is $O(n^2)$. In our analysis, we found that the limited and unlimited versions of BFS vary dramatically. Using the path defined in our experiment section, we found that the unlimited version, with a slope allowance of 8 degrees, took approximately 3 hours to complete. As a result, it was difficult to gain information on performance time for the unlimited version of Breadth-First search. Unlike the unlimited version, the limited version was much faster. A standard run of the limited version of Breadth-first search, with a slope allowance of 8ºs and with a FOV of 18.5km, took only 109.806 seconds to complete. This evidence suggests that the limited version of this algorithm is approximately 98 times faster than its unlimited counterpart. This difference is consistent throughout all of our tests between unlimited and limited versions of Breadth-First search, regardless of slope allowance and FOV range.

Since the unlimited version of Breadth-First search took an extensive amount of time to run, we primarily focused on how the limited version of Breadth-First Search compares to other algorithms. Our first iteration, using a slope of 8º and a FOV of 18.5km, we calculated that it takes 109.806 seconds to complete. When comparing these numbers to our other algorithms, the limited version of Breadth-First

Search ranked second to last in total amount of time taken, behind Best-First Search, with a total time of 4.867 seconds. This evidence shows that Breadth-First search with a relatively high FOV takes approximately 27 times longer than the next fastest algorithm, Best-First search. Data from this run also shows that the limited version of Breadth-First Search yielded a relatively accurate route compared to the other algorithms.

Using the same path but now with a FOV of 4.62km and a slope allowance of 8º, our data shows that the algorithm gets faster and more accurate. When running the test with the specified metrics, the Breadth-First search ran in just 7.287 seconds and found a path 175km long. This is approximately 15 times faster and only 10km farther than when the field of view was 18.5km.

Our findings conclude that the unlimited version of Breadth-First search is very inefficient in both time taken and total distance traveled. Therefore, we suggest that the unlimited Breadth-First Search algorithm should not be used to navigate a theoretical rover. Instead, we would suggest using a limited version of the algorithm. The limited version of Breadth-First Search proved to be a far superior algorithm when we ran it with a smaller FOV.

5.4 Iteratively Deepening A* (IDA*) Algorithm

Following the same methodology as the A* search algorithm, IDA* uses a heuristic function to determine the cost to reach the destination within a weighted graph. In form and function, our implementation of IDA* looks very similar to A* but with one major difference: Iteratively Deepening A* does not store visited nodes into memory for consideration later on. This optimization was considered with the intent to lessen the amount of required memory in order for our algorithm to successfully determine an optimal path to the destination, with the trade-off of having a more complex computational stack. Its Big-O complexity is $O(n^b)$ where b is the branching factor.

As it turned out, IDA* ended up being unable to complete even quite simple path requirements, because the stack would fill up and overflow dramatically quickly. After analysis, we came to realize that the algorithm attempted to consider hundreds of times more path considerations than the other algorithms, given that it wouldn’t rule out paths that visited the same nodes which lead to dead ends, and so the depth of the loops of iteration for path determination ended up being tremendously more complex and would quickly overflow the stack even for start-finish coordinates that the other algorithms could determine a path for within seconds.

It seems that there is an inherent incompatibility with this solution’s approach to our given problem. Because we are treating our elevation map as a very large graph of nodes with weights that determine if the rover may traverse to another adjacent node or not within said graph, there are a vast number of nodes that the rover could visit at each iteration. It seems that IDA* would be better suited for a graph-traversal scenario that resembles something more like a maze, where at most locations, there is one-two directions for which the search algorithm would have to decide between heuristically, and in the event that one path fails, the computation stack would bounce back to the recent junction where there was more than one direction to try, and then continue. One could imagine, however, that because the algorithm does not keep track of places it has been as visited within memory, if the maze were to have a loop within its pathways, IDA* would not believe any direction to be invalid and would then again try an unsuccessful route before continuing. If one were apply this logic to the graph which is commonly generated using our program on a typical elevation map of a planetary surface, it becomes quite clear that, because there are much fewer obstructions and so many more paths to get from point A to point B (acting as maze loops) even within pixels of each other, IDA* tends to build up a tremendous computational feat which becomes unachievable due to stack limitations. We recognize a problematic trend in scientific literature that negative results tend not to get published, so we wanted to include this algorithm as an important example of one that does not work within this problem space.

5.5 Best-First Search

The Best-First Search algorithm is a search algorithm that, similar to the Greedy algorithm, compares all neighboring nodes that are reachable from the current node and expands on the node that is determined to be the closest node to the goal. Unlike the Greedy algorithm, Best-First stores both a list of closed nodes (visited) and a list of open nodes (nodes to visit) causing it to use more memory than Greedy. The Best-First algorithm avoids retracing its steps by checking if nodes are already in the closed list and invalidating nodes that are. The limited version of Best-First is similar to the unlimited, with the key difference being that the goal node in the limited version is the closest node to the actual goal that is within the FOV of the rover. The Big-O complexity of Best-First Search is $O(b^d)$ in the worst case, where b is the branching factor and d is the distance of the shortest path. In practice, however, it will perform much better than this because of its heuristic.

5.5.1 Unlimited

When testing the algorithms with different slope acceptances, the unlimited version of Best-First performs slightly better as it gets a higher slope acceptance. At 8º, it took unlimited Best-First 112.49km to find a path from start to goal. When the slope acceptance is increased to 32º, it does it in 97.02km. On average across all allowable slopes, the unlimited Best-First found paths 104.87km in length.

With respect to time, the unlimited Best-First performs extremely well, Greedy being the only algorithm that is faster. With an 8º slope acceptance, unlimited Best-First took just 45.75ms. Increasing the slope acceptance to 32º allowed the algorithm to finish in just 15ms. On average across all slope acceptances, unlimited Best-First completed in 23.5ms.

5.5.2. Limited

The limited Best-First Search is much less consistent. With an 8º slope acceptance, the limited version took 237.24km to get from start to goal, which is the highest number of steps taken by any algorithm. With a 32º slope acceptance, limited Best-First does a much better job, taking only 101.87km. On average across all allowable slopes, limited Best-First took 132.13km. The limited version is again much less reliable and can take much longer. At 8º slope acceptance it took the limited best first 30.1 seconds, which is over a 700% increase from its unlimited counterpart. A 32º slope acceptance does not produce much better of a result, with the limited Best-First taking 28.54 seconds. This slower performance is likely due to the limited FOV causing the algorithm to more often try paths that lead to a dead end.

When considering the rover's FOV, the limited algorithm performs much better as the FOV is increased. With a FOV of 2.31km, the algorithm found a path that was 158km long. When the FOV is increased to 7.161km, the algorithm found one 118.04km long. This is likely due to the limited "goal node" getting closer to the actual goal as FOV increases.

5.6. Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm used to find the shortest path between start and goal nodes. Unlike the A* algorithm, Dijkstra's algorithm doesn't use any sort of heuristic to determine the node to check next, resulting in decreased efficiency when compared to A* and many of the other algorithms. Its Big-O complexity is `O(n)`.

Dijkstra's algorithm begins by looking at the start node. It then looks at all of the start node's neighbors, calculating the distance from the start node to the neighbor node. Once all neighbor nodes have been checked, the start node is marked as "visited", and the process is repeated for all of the start node's neighbors.

One problem we faced when implementing Dijkstra's algorithm in this software was the size of the GeoTIFFs. For a true implementation of Dijkstra's algorithm, every node in the graph must be marked as unvisited, and put into an unvisited set. Doing this caused the algorithm run for an unreasonable amount of time due to the vast number of nodes we had to consider in our GeoTIFF. In order to implement the algorithm, we only considered a subset of the each GeoTIFF; a box around the start and end coordinates. We assumed that the path was most likely going to be contained within the box, but we expanded the box in order to give the algorithm some buffer space. One limitation of our Dijkstra implementation is the fact that we failed to implement any sort of dynamic buffer increasing if the path strays outside of the predefined box. Due to these challenges, only a limited version of Dijkstra's algorithm was implemented.

When comparing our implementation of Dijkstra's algorithm to the other algorithms we implemented in terms of the calculated path's distance, we see that Dijkstra was one of the worst performing algorithms. When the slope was set at 8º, it was the second-worst algorithm in terms of the calculated path's distance. As the slope was increased, the distance decreased as expected. However, the rate of decrease was much slower than the other algorithms, resulting in Dijkstra being the worst-performing algorithm in regards to the calculated path's distance at a slope allowance of 32º.

When we look at the total time taken to run, Dijkstra performs better than several other algorithms. At a slope of 8º, the total time taken to run was 8.17 seconds. The slowest algorithm was the Unlimited A* algorithm, which took a total time of 3.56 minutes. The time taken by Dijkstra was roughly 4% that of the time taken by the Unlimited A* Non-recursive algorithm. The time taken to run decreased as expected as the slope increased, with the time taken by Dijkstra's algorithm staying in the same relative position when compared to the other algorithms.

6. Algorithm Comparisons

While the route used in testing was chosen so that all the algorithms, minus IDA*, could complete it, not all the algorithms handled the problem in the same way, or as well. The two metrics used, length of the route and time taken to generate it, gave us the following takeaways: first, as expected, a higher slope tolerance will always result in a better (or equal) route for the unlimited algorithms, and they will take less time to generate. Secondly, based on the results of unlimited Breadth-First Search and

Dijkstra’s algorithm versus other algorithms, we determined that usage of a heuristic is essential to generating routes quickly. Finally, a changing FOV has varying impact on algorithms, based on their design and their time complexity.

Logically speaking, it is no surprise that an algorithm would have an easier time with an easier problem, which a higher slope tolerance supplies. First, in terms of length, limited algorithms benefited most from a high slope tolerance: their routes for the most challenging slope, 8º, were on average 100km longer than the easiest slope, 32º. Unlimited algorithms benefitted from full information, and the 8º route was only 11% longer than the Euclidean distance between the start and end points. In terms of speed, all but one algorithm improved (sometimes massively). In this case, the unlimited variants benefitted more: the 8º slope routes were, on average, a full 2633% slower to calculate than the 32º slope routes. For context, the average unlimited 8º route calculation took ~81 seconds per run, versus a 32º route taking only ~3 seconds. Limited algorithms also benefited, but less extremely, with the more difficult routes being 511% slower. It is worth noting that Limited A* actually became slower as slope improved--owing to the fact that it explored more possible paths. Also note that the Greedy algorithm was only used in Limited calculations, because of how it works. Unlimited Breadth-First Search and Dijkstra were also not included in these calculations for reasons elaborated in the following paragraph.

The problem faced by these algorithms is essentially one of pathfinding on a particular sort of graph, where it is the complete elevation map as a grid, with nodes at each pixel and edges to each neighboring pixel wherein the slope is within the specified slope tolerance. This graph works out to having a massive number of nodes with, generally, an even greater number of edges connecting them. Because of this, a heuristic is essential to avoid getting bogged down by the size of the grid. We can see this in comparing the time taken by the unlimited variants of Breadth-First Search and Dijkstra versus the other algorithms. These two are the only variants where a heuristic is not used, and their average time to compute any route is in the range of 3–4 hours. For comparison, the average time taken for all other algorithms, limited and unlimited, for the most difficult slope, 8º, is ~35 seconds. As such, a major takeaway of algorithm development for this sort of problem is that usage of a heuristic is critical.

Generally speaking, a greater FOV will reduce the length (or in other words, improve the quality) of a route, but will cause the algorithm to take longer to complete. On average, paths were 26% longer with the lowest FOV compared to the highest. In contrast, the highest FOV took just over twice as long as the lowest FOV. The exception to this is Limited Breadth-First Search. It appears to have a “sweet spot” for its FOV, around 5.54 km, because all other FOVs tested consistently performed worse than that FOV, both in terms of time taken and route length.

From the data taken, we can conclude which algorithms performed best and worst. There are four categories in which ranking can be made—time taken and route length, with respect to limited FOV and unlimited FOV. The algorithm that produced the best routes was Breadth-First Search, although A* produced similarly good results much more quickly. The fastest algorithm was Greedy. In the figure below are the exact results in these rankings. The rankings are based on median time taken with a precision of 1ms, and best 8º slope route, averaging based on results from varying FOVs with a precision of 1 pixel, or 231m.

FOV Scope	Data	1st	2nd	3rd	4th	5th
Unlimited	**Length**	BFS (97.944km)	A* (99.561km)	Dijkstra (104.643km)	Best First (112.497km)	Greedy (209.517km)
Unlimited	**Time**	Greedy (0.005s)	Best First (0.016s)	A* (0.613s)	Dijkstra (hours)	BFS (hours)
Limited	**Length**	BFS (176.311km)	A* (202.645km)	Greedy (209.517km)	Dijkstra (221.414km)	Best First (236.295km)
Limited	**Time**	Greedy (0.003s)	Dijkstra (3.433s)	BFS (11.540s)	Best First (25.928s)	A* (32.081s)

7. Future Improvements

The easiest mode of expansion for the work we have completed would be to continue implementing various search algorithms for optimal paths because of how these algorithms are classes and the fact that the user interface uses a factory to generate the listed search options.

A large amount of future work resides in the realm of adding simulation-based functionality to our program. M.A.R.S. is designed as an optimal pathfinding application for a future hypothetical rover, so it would be in the rover operator's best interest to be able to take into account as many physical limitation variables as possible before deciding on a route. There are a few scenarios which we would consider within this realm of future work.

1. Soil type, in this manner, could be used to determine a dynamically changing slope acceptance of the rover, given the physical possibility of slipping more easily on certain soil types than others. One could theoretically take a map corresponding to the same coordinates as the elevation map that contains colors corresponding to different soil types and then integrate an additional layer of logic when determining if the rover could traverse such a slope, using the soil information.

2. Battery life would also be an option that could improve overall optimal pathfinding, as a more indirect path may be considered more helpful to the rover if the end of a more direct path places the rover inside the shadow of a large landmass or crater. Implementing battery life into the algorithms could work well if the rover was given a battery life parameter and overall trip time and power were simulated along with the calculation of possible routes and effectively used as an additional heuristic.
3. The treads on the rover could potentially wear more when the rover attempts to traverse more dangerous slopes and soil type combinations. This consideration could be used alongside the proposed simulated terrain types in order to avoid the wear of the rover treads. This way, the lifespan and usefulness of the rover does not decrease due to it always taking the most aggressive possible path just for the sake of following a raw pathfinding algorithm.
4. Our software's current state only supports the use of GeoTIFF elevation file formats, so it may become useful to add various map types in order to have better data compatibility.

8. Conclusions

The primary function of M.A.R.S. is to find a path between two points when given GeoTIFF elevation maps. We implemented several different pathfinding algorithms, and for each algorithm, we implemented what we call a *limited* and an *unlimited* version of the algorithm. *Limited* means that the rover can only 'see' a certain distance away from itself, and must calculate the route dynamically as the rover moves. *Unlimited* means that a GeoTIFF of the entire planet/area is pre-loaded into the rover and the ideal path is calculated upfront before the rover makes its first move. Additionally, we performed an aggressive algorithm comparison where the algorithms were given the same start and end coordinates, and we tracked both the distance taken and the total time taken to run.

Among the algorithms, Greedy was the clear winner when judging by completion time, with both the unlimited and limited versions completing in just a few milliseconds with most slope acceptances.

However, because NASA JPL is more concerned with the safety and efficiency of the rover, time is not the primary variable we judged the algorithms with. While A* took a little longer than most of the other algorithms, it consistently produced shorter paths which would reduce any wear and tear on our theoretical rover. Greedy and the unlimited version of Best-First represent options that both produce shorter paths, though not usually shorter than A*, and finish quickly, so they would be ideal algorithms if completion time is also being considered.

Upon completion of these initial goals, we attempted to implement several other features:

- A full GUI
- Expanded user-interface
- Additional output options
- Additional models such as weather, battery life, and soil type

The only additional feature we were able to implement was a simple user-interface which allows the user to click the start and end coordinates on a map instead of manually entering the coordinates. The unimplemented features above are written about in depth in the “Future Improvement Suggestions” section of this paper. This kind of software is only going to become more prevalent as space science progresses and rovers become more autonomous. As humans venture further into space, autonomy will become imperative due to the infeasibility of manual control at long distances. This work helps pave the way for these kinds of breakthroughs in rover autonomy.