Report of Capstone project

Land rover on Mars

Introduction to Artificial Intelligence

Author: Group 06

Tran Ngoc Khanh - 20200326

Nguyen The Minh Duc - 20204904

Nguyen Ngoc Toan - 20200544

Cu Duy Hiep - 20200212

Nguyen Hoang Tien - 20204927

Advisor: Prof. Muriel Visani

Co-advisor: TA. Nguyen Minh Chau

Academic year: 2021-2022

1. Project Presentation

Given a grayscale topographic map of a part of Mars and some features of the land rover.

Assume that:

- The land rover knows the starting point and the ending point, it has also an elevation map in which the height is represented by 254 pixels (2 for the deepest position and 255 for the highest position) and the sensors to know where it is. (1)
- The land rover can move in only 4 directions: forward, backward, left and right, it can go up or down with an inclination of $\theta \leq 10°$ (this value can be computed by the formula: $tan(\theta) = \frac{|h_1 - h_2|}{dist(P_1, P_2)}$ which is the ratio between the altitude difference of P_1 and P_2 and their real distance (P_1 and P_2 are 2 consecutive positions, therefore $dist(P_1, P_2) = const$). (2)

We need to find the shortest path (if it exists) for the land rover to travel between 2 points from the starting point A and the ending point B. The goal is the destination point B.

This is Mars Orbiter Laser Altimeter **(MOLA)** map of Mars. According to NASA, MOLA is an instrument on the Mars Global Surveyor (MGS), a spacecraft that was launched on November 7, 1996. The mission of MGS was to orbit Mars, and map it over the course of approximately 3 years, which it did sucessfully, completing 4 1/2 years of mapping.

As we can deduce from the image, the darker the region depicts, the deeper the region is. Conversely, the lighter areas represent the regions with higher altitudes.

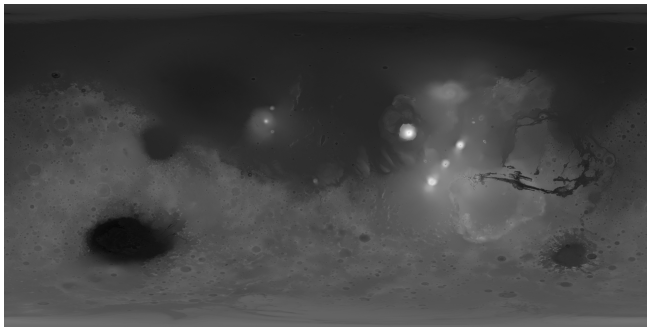

Figure 1: MOLA Mars Map 12K

Mars information

According to NASA, the maximum altitude is 21229 m (peak of Olympus Mons) and the minimum altitude is -8200m (Hellas Impact Crater). The pixel resolution of this map is 1736.25 meters per pixel (m).
So $dist(P_1, P_2) = const = 1736.25(m)$ (P_1 and P_2 are 2 consecutive positions).

Topography map information

We need to note that for a grayscale image, the pixel value is a single number that represents the brightness of the pixel. The most common pixel format is the byte image, where this number is stored as an 8-bit integer giving a range of possible values from 0 to 255. Typically, zero is taken to be black, and 255 is taken to be white at most.

For this particular MOLA Mars map image above, we found that the maximum height (which is equivalent to the peak of Olympus Mons) is represented by a pixel value of 255 and the minimum height (which is equivalent to the Hellas Impact Crater) is represented by a pixel value of 2 (this is the reason why in part (1) of the problem description, we assume that 2 is the pixel value representing for the deepest position and 255 for the highest position)

Therefore, each value pixel represents for: $\frac{1}{255-2+1} \times (max_{height} - min_{height}) = \frac{1}{254} \times (21229 + 8200) = 115.86$ meters of height.

We need to find the shortest path (if it exists) for the land rover to travel between 2 points from the starting point A and the ending point B, but our path needs to satisfy (2). Let's take an example of 2 consecutive points that our land rover can't move between them.

Take 2 points P_1 and P_2 next to each other:
- P1: img_array[2000,5085] = 75
- P2: img_array[2000,5086] = 78

Since the difference between the pixel value of these 2 points is 3, so in reality, the difference between the elevation of these 2 points is $3 \times 115.86 = 347.58(m)$. Others, because $dist(P_1, P_2) = const = 1736.25(m)$. Therefore $\theta = arctan\left(\frac{347.58}{1736.25}\right) = 11.32° > 10°$. In conclusion, the land rover can't go from P_1 to P_2 or vice versa.

2. Research Methodology

2.1. Algorithms

The goal of this project is to find the shortest path (if it exists) between 2 random points on a section of the map of Mars. To achieve this, we implement 3 different searching algorithms:
- Uniform Cost Search
- Greedy Best First Search
- A-star

We consider some factors that can have an effect on the performance of these searching algorithms:
- These algorithms need to sort the frontier while running, so we consider 2 data structures with different sorting algorithms:
  - Priority queue (with Heap sort).
  - List (with Timsort).
- 2 approaches for finding the neighbors of each point:
  - Create the list of neighborhood for each node first and put it in an array before running the algorithm.
  - Check the available neighbors for each node while running the algorithm.
- 5 types of heuristics for informed searching algorithms:
  - Manhattan distance
  - Tie-Breaking Low g-cost
  - Tie-Breaking High g-cost
  - Variance of Tie-Breaking Low g-cost
  - Variance of Tie-Breaking High g-cost

We also hypothesize that the performance of each algorithm depends on the standard devia-

tion of pixel values of the image, so we consider different bins of standard deviation.

2.2. Collecting computational results for analysis and discussion

Since the size of the original image is very big (6144 x 12288), we first consider small image sizes (15 x 15, 20x20, 25x25, 30x30) and then (50x50, 75x75, 100x100), these images are taken from 16 different bins of standard deviation (for example, there exists a bin containing all the images whose the standard deviation is in the range (0.0, 0.5)). Then, for each image, we run the algorithms with 100 random pairs of points (starting point and ending point).

We calculate the average running time and average steps count for each case to compare these algorithms, and discuss some insights about this project.

3. Project idea and insights

In this part, we'll discuss more about our thought process and research idea for this project.

After part 3 of the course, we have some insights about the searching algorithm to solve problems. Then we found lots of applications for these algorithms in real life such as Google Maps, logistics, optimization, etc. However, we want to find solutions to something really unique and distinct from other problems like maze solver or games. After spending a huge amount of time, we come up with the idea of finding a path on an image that reflects all the point's altitudes - which is called a topographic map.

Initially, we find some images for this project. It costs a fortune of time to find on Google an appropriate image that meets our expectations and requirements. Eventually, we found an image that fully comprehends the scene of Mars. We implemented algorithms and tests on that picture immediately and an issue came up: very long running time. That's the reason why we decided to divide it into small pieces for testing the algorithms. We had to spend a week to complete all the main parts of the algorithm and then nearly 2 months to test the algorithm on many instances.

The most difficult part is comparing and improving our algorithms to be more precise and run faster. We found that many aspects influence our results even the data structure we choose to implement the frontier or the base of log function in time complexity. However, the image is too large so we couldn't test many instances because it took a huge amount of time to wait for the land rover to find a path on that image so we decide to divide that image into many small pieces and test individually on these. In order to reduce bias, we randomly pick an image as well as the starting and ending points. We also classify images based on the standard deviation of their altitude at each point to figure out how the algorithms work on various types of topography whether it is flat or rough.

After collecting the data, we get into the parts of analysis and visualization. This is not as difficult as the testing part when we already have everything we need before. Thus, we spent only a week completing this part.

Finally, we want to announce that our idea and applied algorithms are only the very beginning of future masterpieces. If we have data that is more general and comprehensive, we will improve this initiative to be more functional in certain situations (perhaps it could be accustomed to exploring Mars over random insurmountable obstacles to search for micro-organism or extra-terrestrial life; or to study the structure, composition, variability, and dynamics of Mars' atmosphere to realize a dream of creating habitats for humanity on this planet, etc).

4. Result Discussion

4.1. Comparison of frontier definitions

4.1.1. Description

In order to implement the frontier, we have 2 ideas:

- First, use priority queue and its corresponding sorting algorithm (heap sort) to order the key-value pair of all the elements in the queue.
- Second, use list and timsort for sorting the f-cost of each node.

Before running the whole analysis, we need to

define which is the good data structure for our frontier. Below are our results after testing these 2 data structures for configuring the frontier.

4.1.2. Parameters

- 7 image sizes: 15, 20, 25, 30, 50, 75, 100.
- Neighbors finding method: Find while running.
- Heuristic type: Manhattan distance.
- Number of pair of random points (both starting point and ending point): 100.
- Number of images per bin: 100.
- Number of bins: 16.

$\Rightarrow$ Total number of tests for each type of frontier: $100 \times 100 \times 16 = 160,000$ tests.

4.1.3. Results

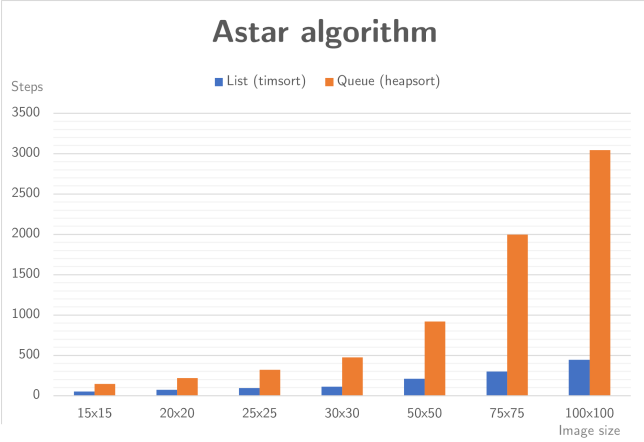

Figure 2: Steps comparison for different frontier data structure - Astar

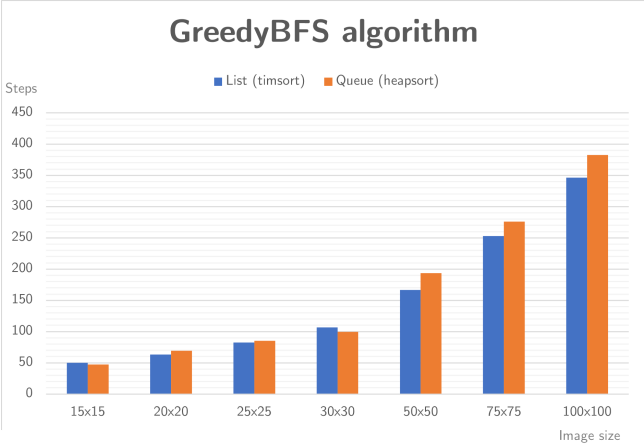

Figure 3: Steps comparison for different frontier data structure - GreedyBFS

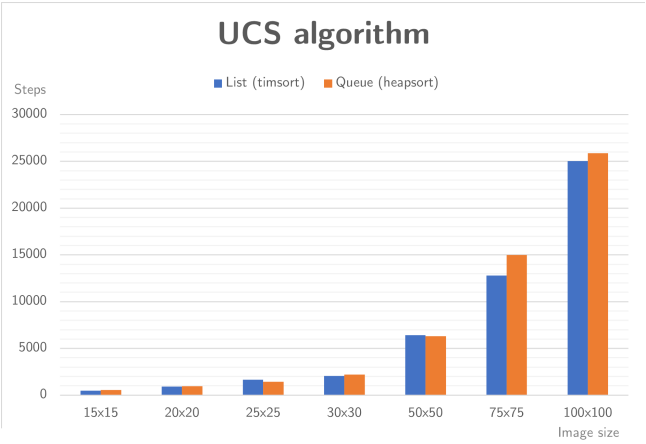

Figure 4: Steps comparison for different frontier data structure - UCS

As we can see on the charts, for the algorithm using priority queue as the frontier's data structure, the number of steps are slightly more than that using list as its frontier's data structure in UCS and Greedy BFS. But, in Astar, it's considerably bigger than that using list as its frontier's data structure. The reason may belong to the stability of the sorting algorithms, heapsort is an unstable sorting algorithm while timsort (the combination of merge sort and insertion sort) is a stable sorting algorithm.

Therefore, in all the tests below, we choose list (with timsort) as the frontier's data structure.

4.2. Comparison of different algorithms

4.2.1. Description

- UCS is an algorithm that always finds the optimal solution, but the running time of the algorithm is very long (because it has to consider all the points in the queue until it reaches the destination). The algorithm is suitable for use in cases to find an optimal solution without regard to processing time. For example, in our problem, images with size 100*100 make the UCS algorithm run very long and it is almost impossible to test too many cases on this algorithm.
- Greedy best-first search is an algorithm with a strong greedy smell, so sometimes we find paths much worse than the optimal solution. But in return, the running time of BFS algorithm is very fast, which can be applied in real-time problems. The per-

forming tests, which run with BFS, give the results very fast: solved or cannot be solved (stuck in an infinite loop).
- A* is a complete algorithm in that case because there are finite states and only non-negative edge weights. It has the advantages of both UCS and BFS (obtains a good solution from UCS, very fast running time of BFS). This is a very comprehensive algorithm in graph search and has many practical applications.

This is a pseudo-code of A*:

Algorithm 1 Astar search

```
steps ← 0
visitedNodes ← //
frontier ← Priority-Queue()
frontier.push(startNode)
while frontier is not empty do
    actualNode ← frontier.pop()
    steps ← steps + 1
    if actuaNode is goalNode then
        return Found
    end if
    visitedNodes.add(actualNode)
    for nextNode in actualNode.neighbors do
        steps ← steps + 1
        if nextNode not in visitedNodes then
            g ← actualNode.g + 1
            h ← heuristics(nextNode, goalNode)
            nextNode.g ← g
            nextNode.h ← h
            nextNode.f ← g + h
            if nextNode already in frontier then
                if old nextNode.f of frontier < nextNode.f then
                    replace old nextNode.f by nextNode.f
                end if
            else
                frontier.push(nextNode)
            end if
        end if
    end for
    Final instructions
end while
```

Greedy Best-first search and Uniform cost search can be easily deduced from A* by just changing the h-cost and f-cost.

4.2.2. Parameters

- 7 image sizes: 15, 20, 25, 30, 50, 75, 100.
- Type of frontier: List (using timsort to sort).
- Neighbors finding method: Find while running.
- Heuristic function: Manhattan distance.
- Number of pairs of random points (both starting point and ending point): 100.
- Number of images per bin: 100.
- Number of bins: 16.

⇒ Total number of tests for each algorithm: $100 \times 100 \times 16 = 160,000$ tests.

4.2.3. Results

Size	UCS	Astar	GreedyBFS
15	496.19	52.06	49.88
20	922.69	73.94	63.44
25	1641.63	93.63	82.69
30	2050.56	110.56	106.94
50	6412.50	208.13	166.87
75	12798.10	297.88	253.06
100	25046.38	443.06	346.31

Table 1: Algorithms comparison (steps count)

From the table above, we can deduce that, overall, GreedyBFS gives us the fastest solution, while UCS needs the largest number of steps to solve the search problem. Since Astar is the combination of real cost function and heuristic function, it always returns the optimal solution and the running time of this algorithm is a little bit more than that of Greedy BFS.

In order to keep the optimal solution for path searching and have a good running time, we choose Astar for all the analysis below.

4.3. Influence of standard deviation to the running time

4.3.1. Description

As we mentioned earlier, we divided the images of different sizes into each particular bin of standard deviation. We hypothesize that the larger the standard deviation is, the faster the algorithm finds the solutions (the reason we think is that for an image of immense standard deviation, which means that the region represented by this figure is less flat, the algorithm is more

likely to be stuck in dead-points. Therefore, it can't find the solution in an easy way). But, the test results are different from what we thought.

4.3.2. Parameters

- 7 image sizes: 15, 20, 25, 30, 50, 75, 100.
- Type of frontier: List (using timsort to sort).
- Neighbors finding method: Find while running.
- Heuristic function: Manhattan distance.
- Number of pair of random points (both starting point and ending point): 100.
- Number of images per bin: 100.
- Number of bins: 16.

4.3.3. Results

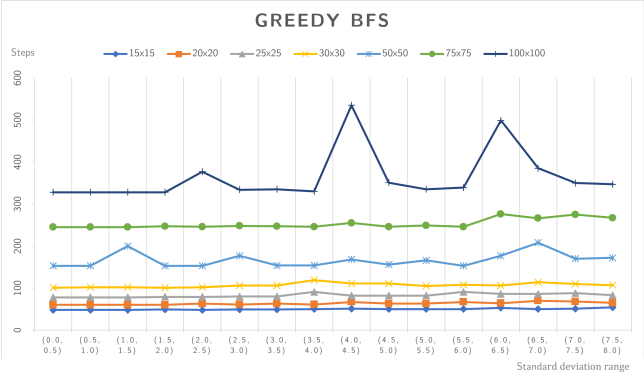

Figure 5: Influence of standard deviation to the running time - GreedyBFS

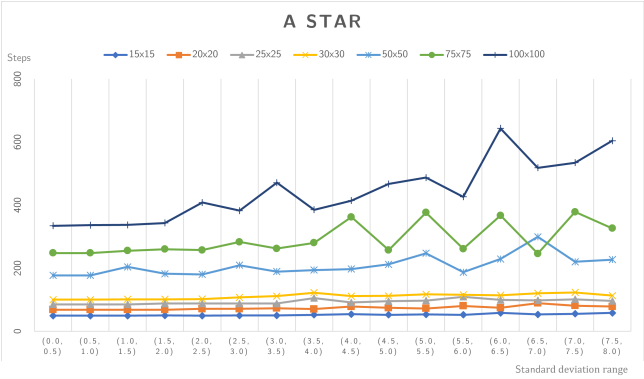

Figure 6: Influence of standard deviation to the running time - Astar

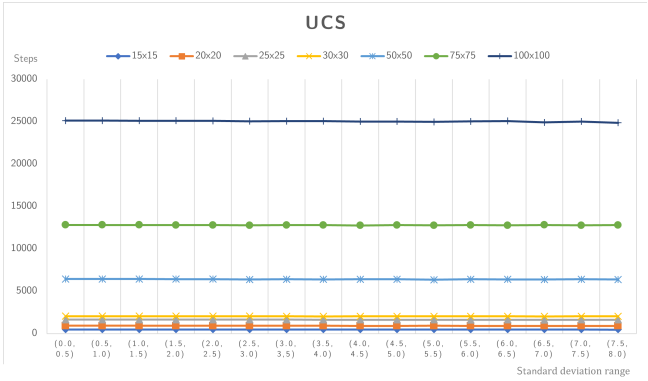

Figure 7: Influence of standard deviation to the running time - UCS

In this part, we can see that:

- GreedyBFS and Astar seem to have a local peak at two bin, one in range (1.5, 2.0) and one in range (6.0, 7.0) for all kinds of images. They also seem to increase a little bit from the smallest bin to the larger one.
- UCS seems to have no peak and its graph is flat, the reason is that UCS does not use any heuristic function and it considers every node from the beginning for all categories of images.

4.4. Heuristic functions

4.4.1. Description

It's not easy to choose a good heuristic function. That's what we got after doing experiments. Actually we have 5 different heuristic functions (including 2 variances that we created).

First of all, we use Manhattan distance. It is the distance between two points measured along axes at right angles. For example, we consider two points: $A(x_A, y_A)$ and $B(x_B, y_B)$, so its Manhattan distance is $|x_A - x_B| + |y_A - y_B|$. We choose this kind of distance and don't consider others distances like Euclidean distance or Octile distance because our agent can only move horizontally and vertically, not diagonally.

But, from the results of what we have done, after doing experiments and printing the frontier while running, there are many different paths having the same length. That is why we think about including also a parameter coefficient δ which is very small for the heuristic result. The reason is simple: change a little bit the f-cost for each path. And the algorithm may explore less

than usual.

There are 2 strategies that may be applied to this kind of heuristic (which is called Tie-breaking): Tie-breaking Low g-cost (TLg) and Tie-breaking High g-cost (THg). The first strategy tends to reduce the g-cost a little bit, which means that we prefer to value less the true cost g (Low g-cost) until the agent arrive to the goal node. This heuristic function can be simply explain by this formula: $h = Manhattan(startNode, endNode) * (1 - \delta)$, note that $\delta > 0$. But that's not what we want, we don't want to reduce our heuristic cost. So we continue to consider the remaining type of tie-breaking. The formula for Tie-breaking High g-cost is as follows: $h = Manhattan(startNode, endNode) * (1 + \delta)$, note that $\delta > 0$. It may happen problems with this heuristic that we can break the admissibility a little bit but this doesn't matter if δ is small enough.

But δ is fixed, why can't we increase or decrease δ by multiplying it with a coefficient which can vary corresponding to where the agent (rover) is? That is why we decided to create 2 variances of Tie-breaking: Variance of Tie-breaking Low g-cost (v-THg) and Variance of Tie-breaking High g-cost.

Algorithm 2 var-tie-breaking-low-g-cost

1: $\text{rate}_1 = \text{manhattan(actualNode, goalNode)}$
2: $\text{rate}_2 = \text{manhattan(startNode, goalNode)}$
3: **return** $\text{rate1} * (1 + \text{delta} * (1.5 - \text{rate}_1/\text{rate}_2))$

The idea of 1.5 at line 3 is because $\text{rate}_1/\text{rate}_2$ usually $\in (0, 1)$ so new δ varies between $0.5 \times \delta$ and $1.5 \times \delta$ and its center is $1 \times \delta$, for which we can compare this variance with the original heuristic function.

Algorithm 3 var-tie-breaking-high-g-cost

1: $\text{rate}_1 = \text{manhattan(actualNode, goalNode)}$
2: $\text{rate}_2 = \text{manhattan(startNode, goalNode)}$
3: **return** $\text{rate1} * (1 + \text{delta} * (0.5 + \text{rate}_1/\text{rate}_2))$

The idea of 0.5 at line 3 is because $\text{rate}_1/\text{rate}_2$ usually $\in (0, 1)$ so new δ varies between $0.5 \times \delta$ and $1.5 \times \delta$ and its center is $1 \times \delta$, for which we can compare this variance with the original heuristic function.

4.4.2. Parameters

- 1 image size: 50x50 (but we use also 25x25 image for experiment with $\delta = 0.001$).
- Algorithm: Astar search (we compare the influence of different heuristic functions in f-cost (which is equal to the sum of g-cost and h-cost) and Astar uses both of these 2 types of cost).
- Type of frontier: List (using timsort to sort).
- Neighbors finding method: Find while running.
- Heuristic function: Manhattan distance.
- Number of pair of random points (both starting point and ending point): 100.
- Number of images per bin: 100.
- Number of bins: 16.

4.4.3. Results

Let's see the experimenting results for $\delta = 0.001$.

Bin	MHT	THg	v-THg	TLg	v-TLg
0.0-0.5	85	85	85	476	131
0.5-1.0	86	86	86	475	131
1.0-1.5	86	86	86	475	131
1.5-2.0	87	87	87	474	132
2.0-2.5	86	86	86	475	131
2.5-3.0	89	89	89	474	133
3.0-3.5	91	90	90	475	135
3.5-4.0	91	91	91	476	135
4.0-4.5	99	99	99	480	143
4.5-5.0	93	93	93	472	137
5.0-5.5	98	98	98	477	142
5.5-6.0	98	97	97	475	142
6.0-6.5	98	97	97	474	141
6.5-7.0	100	100	100	474	143
7.0-7.5	103	102	102	473	145
7.5-8.0	94	93	93	474	138

Table 2: Image of size 25 - Astar

Bin	MHT	THg	v-THg	TLg	v-TLg
0.0-0.5	162	162	162	162	162
0.5-1.0	162	162	162	162	162
1.0-1.5	162	162	162	162	162
1.5-2.0	163	163	163	163	163
2.0-2.5	166	166	166	166	166
2.5-3.0	170	170	170	170	170
3.0-3.5	171	171	171	171	171
3.5-4.0	176	176	176	176	176
4.0-4.5	174	174	174	174	174
4.5-5.0	177	177	177	177	177
5.0-5.5	182	182	182	182	182
5.5-6.0	168	168	168	168	168
6.0-6.5	174	174	174	174	174
6.5-7.0	171	171	171	171	171
7.0-7.5	184	184	184	184	184
7.5-8.0	189	189	189	189	189

Table 3: Image of size 50 - Astar

The steps are not considerably different but we can see that for some bin A* used with the Tie-breaking method gives us one step less than the original approach (for Table 2 with image size 25) with only Manhattan distance, but no changes in steps in Table 3 (with image size 50). The question is why there is only a very little difference here? Because δ is too small. And let us increase δ a little bit.

The tables below show the result of another analysis. These are the number of steps according to different values of delta and to different types of heuristic functions, and we evaluated also optimality the score of the path obtained from the algorithm (We evaluated this score by comparing the cost returned by the algorithm with that returned by A* search using Manhattan distance - an admissible heuristic function)

Bin	$\delta = 0.01$	$\delta = 0.1$	$\delta = 0.5$	$\delta = 1$
0.0-0.5	148(1.0)	148(0.9999)	148(0.9994)	148(0.9994)
0.5-1.0	148(1.0)	148(0.9998)	148(0.9986)	148(0.9984)
1.0-1.5	151(1.0)	151(0.9992)	150(0.9939)	150(0.9920)
1.5-2.0	152(1.0)	152(0.9984)	151(0.9936)	151(0.9919)
2.0-2.5	154(1.0)	153(0.9981)	152(0.9903)	152(0.9889)
2.5-3.0	179(1.0)	178(0.9971)	173(0.9826)	171(0.9779)
3.0-3.5	164(1.0)	162(0.9963)	157(0.9809)	155(0.9759)
3.5-4.0	167(1.0)	165(0.9967)	159(0.9734)	156(0.9676)
4.0-4.5	169(1.0)	166(0.9952)	159(0.9711)	157(0.9644)
4.5-5.0	161(1.0)	160(0.9961)	155(0.9753)	153(0.9675)
5.0-5.5	157(1.0)	156(0.9975)	153(0.9831)	152(0.9804)
5.5-6.0	234(1.0)	230(0.9955)	221(0.9712)	218(0.9638)
6.0-6.5	187(1.0)	183(0.9956)	174(0.9673)	169(0.9585)
6.5-7.0	187(1.0)	185(0.9940)	177(0.9687)	174(0.9610)
7.0-7.5	192(1.0)	189(0.9964)	181(0.9714)	178(0.9633)
7.5-8.0	184(1.0)	181(0.9955)	173(0.9689)	171(0.9590)

Table 4: Tie-breaking High g-cost

Bin	$\delta = 0.01$	$\delta = 0.1$	$\delta = 0.5$	$\delta = 1$
0.0-0.5	148(1.0)	148(0.9998)	148(0.9994)	148(0.9994)
0.5-1.0	148(1.0)	148(0.9996)	148(0.9986)	148(0.9985)
1.0-1.5	151(1.0)	150(0.9982)	150(0.9931)	150(0.9919)
1.5-2.0	152(1.0)	152(0.9980)	151(0.9933)	151(0.9919)
2.0-2.5	154(1.0)	153(0.9959)	152(0.9895)	152(0.9886)
2.5-3.0	179(1.0)	176(0.9943)	171(0.9805)	169(0.9766)
3.0-3.5	164(1.0)	160(0.9926)	156(0.9780)	155(0.9747)
3.5-4.0	167(1.0)	164(0.9909)	157(0.9710)	156(0.9659)
4.0-4.5	169(1.0)	164(0.9888)	157(0.9678)	156(0.9625)
4.5-5.0	161(1.0)	159(0.9916)	154(0.9715)	153(0.9669)
5.0-5.5	157(1.0)	155(0.9931)	152(0.9822)	152(0.9795)
5.5-6.0	234(1.0)	228(0.9903)	218(0.9668)	215(0.9605)
6.0-6.5	187(1.0)	181(0.9889)	171(0.9620)	166(0.9549)
6.5-7.0	187(1.0)	182(0.9888)	175(0.9644)	172(0.9581)
7.0-7.5	192(1.0)	187(0.9893)	179(0.9671)	178(0.9614)
7.5-8.0	184(1.0)	179(0.9899)	172(0.9630)	170(0.9575)

Table 5: Variance of Tie-breaking High g-cost

Fewer steps means lower accuracy, the optimality of path returned by our searching algorithm decreases while delta increases. The other insight that we can get from the table is: the more the standard deviation is, the less the optimality of path returned. But for a very small value of δ, we still get the optimality score of 1.0.

Let's take a look at the analysis results of the two remaining heuristic functions.

Bin	$\delta = 0.01$	$\delta = 0.1$	$\delta = 0.5$	$\delta = 1$
0.0-0.5	1179(1.0)	1332(1.0)	2785(1.0)	5423(1.0)
0.5-1.0	1178(1.0)	1332(1.0)	2784(1.0)	5421(1.0)
1.0-1.5	1176(1.0)	1330(1.0)	2782(1.0)	5417(1.0)
1.5-2.0	1179(1.0)	1332(1.0)	2783(1.0)	5418(1.0)
2.0-2.5	1177(1.0)	1331(1.0)	2784(1.0)	5416(1.0)
2.5-3.0	1192(1.0)	1346(1.0)	2797(1.0)	5415(1.0)
3.0-3.5	1172(1.0)	1326(1.0)	2775(1.0)	5398(1.0)
3.5-4.0	1172(1.0)	1327(1.0)	2778(1.0)	5405(1.0)
4.0-4.5	1171(1.0)	1326(1.0)	2780(1.0)	5403(1.0)
4.5-5.0	1168(1.0)	1324(1.0)	2776(1.0)	5404(1.0)
5.0-5.5	1172(1.0)	1326(1.0)	2778(1.0)	5408(1.0)
5.5-6.0	1222(1.0)	1377(1.0)	2810(1.0)	5390(1.0)
6.0-6.5	1174(1.0)	1331(1.0)	2780(1.0)	5389(1.0)
6.5-7.0	1179(1.0)	1334(1.0)	2782(1.0)	5388(1.0)
7.0-7.5	1186(1.0)	1341(1.0)	2782(1.0)	5384(1.0)
7.5-8.0	1182(1.0)	1337(1.0)	2784(1.0)	5397(1.0)

Table 6: Tie-breaking Low g-cost

For Tie-breaking Low g-cost, we can see that the optimality always remains 1.0 but as */delta* increases, so does the number of steps increase sharply! We know that f-cost is the sum of g-cost and h-cost, for Tie-breaking Low g-cost, the h-cost, which is already admissible, is now even more admissible because the coefficient of h is always less than 1. That's the reason why the optimality is always guaranteed although the num-

ber of steps is very large.

Bin	$\delta = 0.01$	$\delta = 0.1$	$\delta = 0.5$	$\delta = 1$
0.0-0.5	284(1.0)	284(0.9999)	286(0.9996)	300(0.9995)
0.5-1.0	284(1.0)	284(1.0)	285(0.9989)	299(0.9990)
1.0-1.5	286(1.0)	286(0.9999)	288(0.9968)	302(0.9954)
1.5-2.0	287(1.0)	288(0.9996)	288(0.9951)	303(0.9946)
2.0-2.5	289(1.0)	289(0.9994)	289(0.9929)	304(0.9909)
2.5-3.0	313(1.0)	313(0.9996)	320(0.9903)	342(0.9869)
3.0-3.5	296(1.0)	295(0.9989)	294(0.9880)	313(0.9835)
3.5-4.0	299(1.0)	298(0.9991)	299(0.9866)	319(0.9794)
4.0-4.5	301(1.0)	300(0.9992)	298(0.9820)	320(0.9770)
4.5-5.0	293(1.0)	293(0.9992)	293(0.9876)	312(0.9812)
5.0-5.5	290(1.0)	290(0.9996)	290(0.9906)	305(0.9869)
5.5-6.0	365(1.0)	364(0.9989)	367(0.9834)	391(0.9777)
6.0-6.5	317(1.0)	316(0.9993)	314(0.9832)	341(0.9727)
6.5-7.0	318(1.0)	317(0.9986)	320(0.9805)	343(0.9728)
7.0-7.5	322(1.0)	321(0.9991)	318(0.9833)	336(0.9749)
7.5-8.0	315(1.0)	314(0.9991)	312(0.9830)	336(0.9744)

Table 7: Variance of Tie-breaking Low g-cost

The variance of Tie-breaking Low g-cost will improve its disadvantage because its heuristic value is not always less than 1, so it's a better choice than the original Tie-breaking Low g-cost.

4.5. Find neighbors

4.5.1. Description

There are 2 methods to find the neighbors of each point. Return to the problem description at (2), we have that: the land rover can move in only 4 directions: forward, backward, left and right, it can go up or down with an inclination of $\theta \leq 10°$ (this value can be computed by the formula: $tan(\theta) = \dfrac{|h_1 - h_2|}{dist(P_1, P_2)}$ which is the ratio between the altitude difference of P_1 and P_2 and their real distance (P_1 and P_2 are 2 consecutive positions, therefore $dist(P_1, P_2) = const$). So the maximum pixel value of the two points consecutive is 2, and the algorithm must decide if the next point satisfying the condition or not.

We have 2 approaches:

- First, we take an array and save all the neighbors of each point in the map and then run the search function.
- Second, we find the neighbors inside the search function.

4.5.2. Parameters

- 2 image sizes: 25x25 and 50x50.
- 2 algorithms: Astar.
- Type of frontier: List (using timsort to sort).
- Heuristic function: Manhattan distance.
- Number of pair of random points (both starting point and ending point): 100.
- Number of images per bin: 100.
- Number of bins: 16.

4.5.3. Results

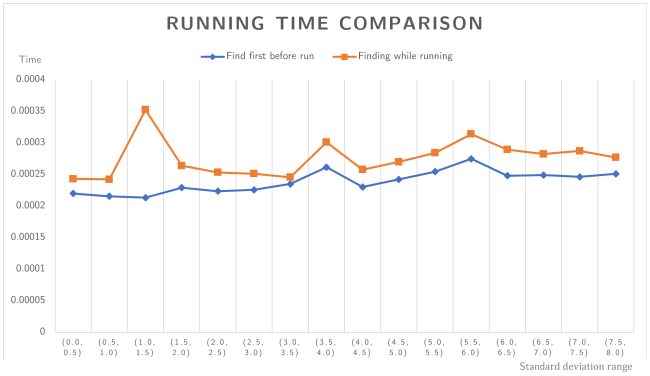

Figure 8: Running time comparison - image size 25

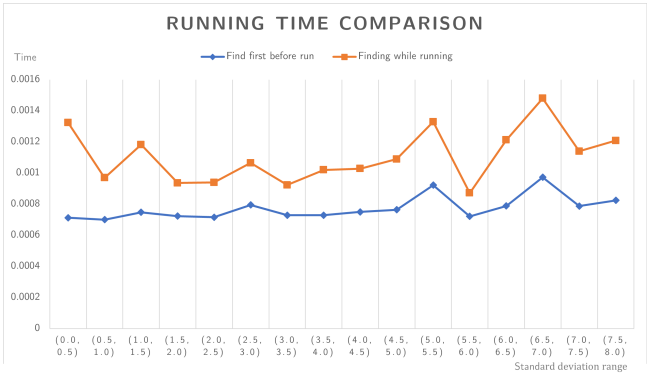

Figure 9: Running time comparison - image size 50

As we can see in the graph, if we find the neighbors first and run the search algorithm later, the running time is less than the case where we find the neighbors while running the search algorithm. Others, after experimenting, both methods produce the same costs in test cases.

It's reasonable since if we spend more memory-saving data, we have less time for running the algorithm and the algorithm is always the same, so the returned cost does not change.

5. Conclusions

In the process of problem-solving, we found that there are 4 main factors that directly affect the

results.

- Heuristic function
- How to define the frontier
- How to determine neighbor points
- What is the standard deviation of the image

Thereby, we have let the algorithm perform by keeping constant some typical factors and altering the value of considering factors. We crop the big one and use the small-size images in order to be able to test on as many cases as possible (taken by random selection). That work is to avoid bias and to generalize the results for all the images belonging to each specific bin of standard deviation.

We have done a lot of tests to compare the effects of alternative factors on the performance of the algorithm.

First, we compare the running time between 2 different data structures for the frontier (list with timsort and priority queue with heapsort). In this part, we've decided to choose list as the main data structure for our frontier.

Second, we compare the running time between our 3 different searching algorithms. Greedy BFS is indeed the fastest algorithm and UCS is the lowest. A* is like a combination between Greedy BFS and UCS.

Third, we consider the influence of standard deviation of each image on the running time.

Fourth, we compare 5 heuristic functions:

- Manhattan distance is an admissible heuristic which gives us a good result.
- Tie-breaking high g-cost and its variance give a slightly better result then Manhattan distance but if we ignored the optimality a little bit, A* search would run much more faster.
- Tie-breaking low g-cost is the worst heuristic function as it leads us to a very bad set of solutions in which the number of steps are enormous.
- Variance of Tie-breaking Low g-cost improves the original heuristic function significantly.

Fifth, we compare 2 methods finding the neighbor: Finding while running, Find first before run:

- Both methods produce relatively similar costs in test cases. No cost difference.
- But Find first before run has a faster runtime.

6. Further research directions

We think that there remains lots of interesting insights that can be inferred from the analysis results.

Firstly, let's take a look at figure (2), as we can see, the path length of A* algorithm using priority queue as its frontier's data structure is considerably larger than that using list as its frontier's data structure. So the question is that: Does the order of path cost of frontier node saved have a big impact in the result path length? How does the type of sort affect the result? And whether for every considering node which has the same cost, does the node appearing first in the frontier should be chosen first?,... There are a lot of stories to tell after discovering this problem.

Secondly, about the part of heuristic functions, δ there remains constant. But how can we choose δ automatically and do not need to care about its constant value? Are there any formulas to define δ so that we just need to provide the optimality acceptable (for example, greater than 0.95)? The trade-off between the optimality and the running time of the search algorithms can be studied more deeply.

7. List of tasks

7.1. Idea forming and data preparation

Idea forming: all members

Problem description: Cu Duy Hiep

Image searching: Nguyen Hoang Tien

Data generating: Nguyen Ngoc Toan

7.2. Programming part

7.2.1. Find first before run method

- UCS implementation: Nguyen The Minh Duc
- Greedy BFS implementation: Nguyen The Minh Duc

- Astar implementation: Nguyen The Minh Duc
- Cleaning code: Nguyen Hoang Tien

7.2.2. Find while running method

- Astar implementation: Tran Ngoc Khanh
- UCS implementation: Nguyen Hoang Tien
- Greedy BFS implementation: Nguyen Ngoc Toan
- Cleaning code: Nguyen Hoang Tien

7.3. Analysis part

- Planning analysis process: Tran Ngoc Khanh
- Analysis implementation: Nguyen Hoang Tien
- Visualization: Nguyen The Minh Duc
- Analysis: Tran Ngoc Khanh, Nguyen Hoang Tien
- Writing report: all members
- Writing presentation: Nguyen The Minh Duc, Nguyen Ngoc Toan, Cu Duy Hiep

References

[1] Stuart J. Russell and Peter Norvi. Artificial Intelligence - A Modern Approach Third Edition

[2] Wikipedia. A* search algorithm.

[3] Geeksforgeeks. A* Search Algorithm

[4] Towards Data Science. A-Star (A*) Search Algorithm

[5] My Great Learning. Best First Search Algorithm in AI | Concept, Implementation, Advantages, Disadvantages

[6] Educative.io. What is uniform-cost search?

[7] Stanford. Heuristics