Project 1: Fast Trajectory Replanning

Vishnu Dhanasekaran, Swarnendu Roy, Nikhil Kulkarni
`netID: vd247, skr81, nsk62`

Rutgers University — July 13, 2020

Part 0: Setup your Environments

You will perform all your experiments in the same 50 grid-worlds of size 101 × 101. You first need to generate these environments appropriate.

To generate our 50 101x101 grids we ran the provided gird-world-generator.py file and read and stored them into a 2D list.

Part 1: Understanding the Methods

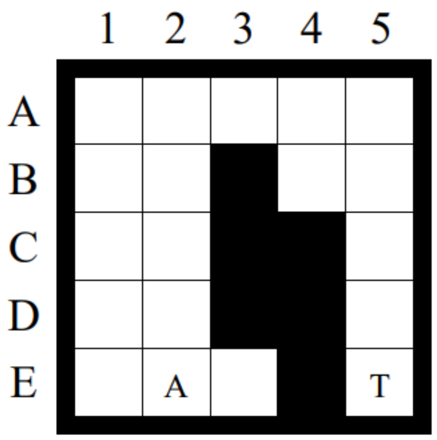


Figure 8: Second Example Search Problem

Explaining why the agent chooses to go east rather than north

The agent initially starts at cell E2, agent can now see all of the nodes that are neighboring the cell E2, these are (E3 f(3), E1 f(5), D2 f(5)) with g for all the nodes at 1. These nodes are added to the open list heap after their g(cost), h(heuristic) and f(g + h) values are updated. Once this is done, we pop the heap, which orders them in increasing order of f values, starting at E3. When the heap is popped the cell E3 will be popped before the other two and we will perform A* search on that E3. This is because according the heuristic E3 has a lower value while the cost of going to all the nodes stay the same, hence E3's f value is lower. Hence the agent will prefer to move east first.

This project argues that the agent is guaranteed to reach the target if it is not separated from it by blocked cells. Give a convincing argument that the agent in finite grid worlds indeed either reaches the target or discovers that this is impossible in finite time.

If the grid space is a finite space then A* search can find a solution in finite time through exhaustively searching the grid, assuming that there is no blocked cells. Since every step's cost increases by one and every step also generates a finite number of neighbors, we will eventually be able to find a solution. One can argue that we may expand the same nodes more than once and get stuck in a loop, to avoid this we have implemented a closed list that stores the nodes that have already been expanded avoiding any repeats.

Part 2: The Effect of Ties

Repeated Forward A* needs to break ties to decide which cell to expand next if several cells have the same smallest f-value. It can either break ties in favor of cells with smaller g-values or in favor of cells with larger g-values. Implement and compare both versions of Repeated Forward A* with respect to their run time or, equivalently, number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation.

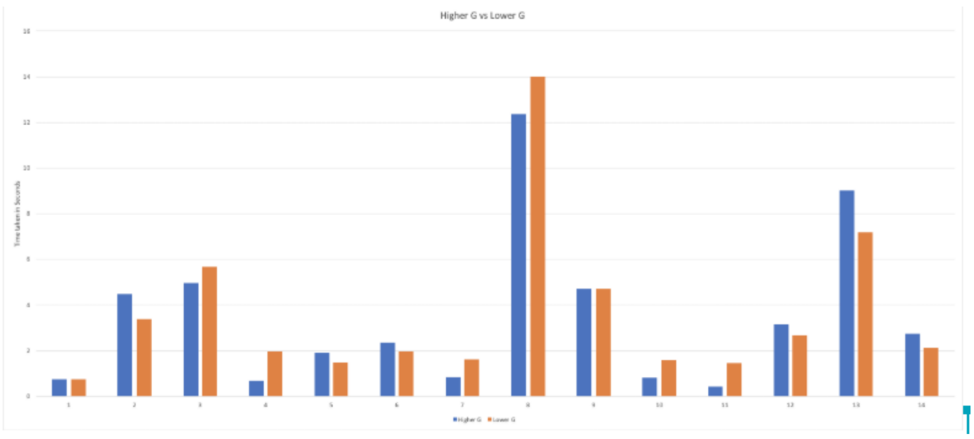
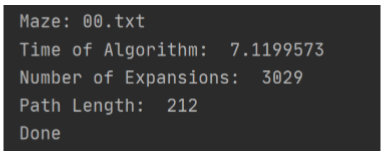


Figure 1: Avg run-times: higher G: 3.51s, lower G: 4.62s

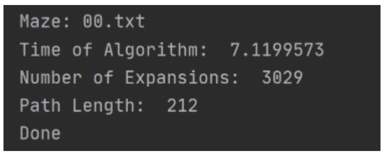
After running our experiments over 50 different grids, ignoring the grids where a path was not found we see that, tie-breaking with a lower g(x) value results in a much higher run-time than a higher g(x). This is can be seen in the graph above.
This can be attributed to the fact the lower g(x) value results in considerably more expansions than using higher g(x) which is visualized by the two screenshots provided. This happens because the lower g(x) algorithm expands more nodes near the start. The cases where the lower g(x) has a better run-time can be attributed to more blocked nodes being present near the start.
Both search algorithms were run on the same 101x101 grid. Lower g(x) gives more than 1000 expansions which then causes a much higher run time.

After a run with greater g(x) value tie-breaking the results are as follows:



```
Maze: 00.txt
Time of Algorithm:  7.1199573
Number of Expansions:  3029
Path Length:  212
Done
```

We can compare this to the lower g(x) value tie-breaking:



```
Maze: 00.txt
Time of Algorithm:  7.1199573
Number of Expansions:  3029
Path Length:  212
Done
```

Part 3: Forward vs Backward

Implement and compare Repeated Forward A* and Repeated Backward A* with respect to their run-time or, equivalently, number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both versions of Repeated A* should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly.

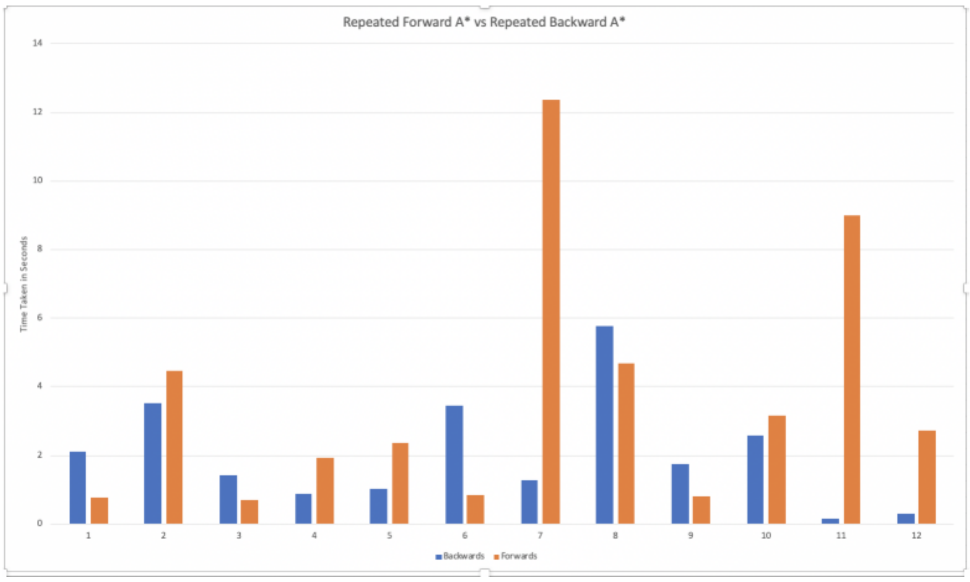


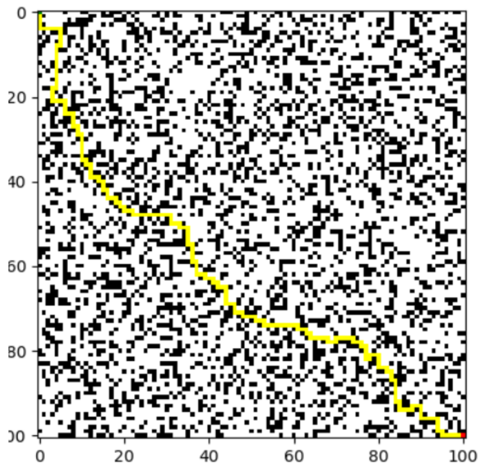
Figure 2: Avg run-times: Repeated Forward A*: 3.51s, Repeated Backward A*: 2.29s

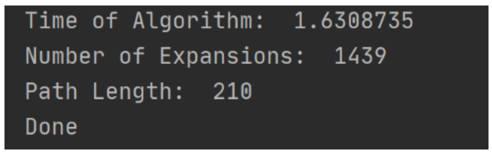
On running Repeated Forwards A and Repeated Backwards A, we find that for some cases Repeated Forwards A is faster than Repeated Backwards A but for certain cases, Repeated Backwards A runs quicker than Repeated Forwards A and hence the run-time is varying but on an average it favors backwards. The favoring of backwards is due to the nature of the formed grid. This can be attributed to the fact that there may be potential bias in the positioning of the start and goal. It can be seen that if there are more blocked nodes around the position of wherever the agent's starting position nodes, it leads to less expansion of nodes and a quicker run-time.

Given below we run both algorithms on two separate grids which gives us the different cases. We show pictures of the grids with the path, blocked and unblocked nodes, and also the run-times of the algorithms on each grid.

The start is marked in green and the goal is marked in red. We see near the start there are more blocked nodes than near the goal which causes the better run-time as shown below.

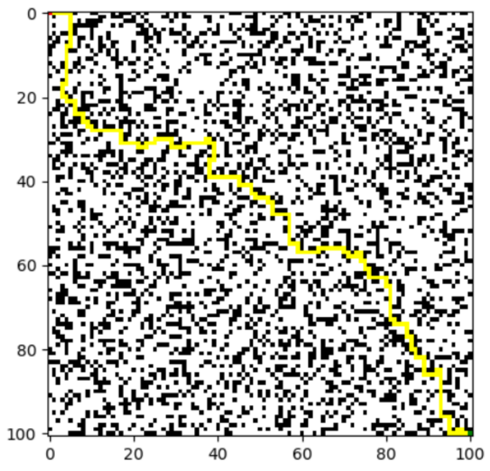
Here is an example of grid favoring Forwards A*:

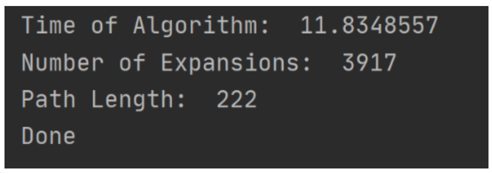




```
Time of Algorithm:  1.6308735
Number of Expansions:  1439
Path Length:  210
Done
```

Figure 1: Faster Forward A*

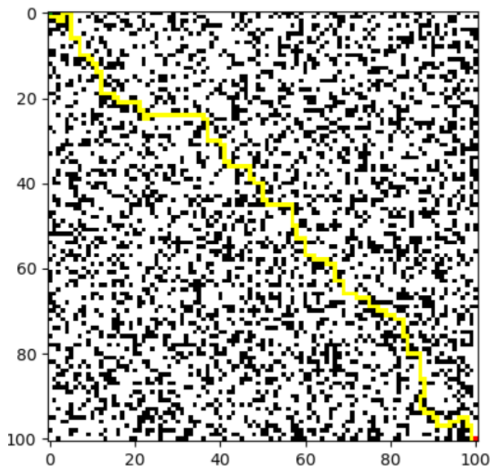


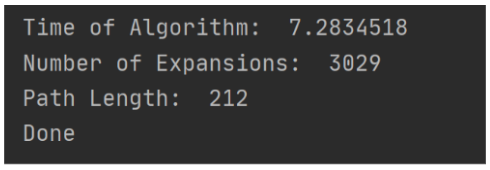


```
Time of Algorithm:  11.8348557
Number of Expansions:  3917
Path Length:  222
Done
```

Figure 2: Slower Backward A*

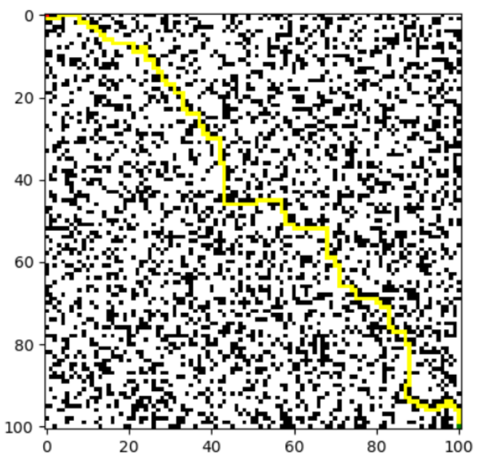
Example of a grid favoring Backwards A*:

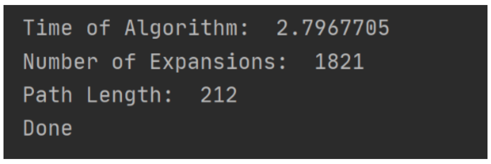




```
Time of Algorithm:  7.2834518
Number of Expansions:  3029
Path Length:  212
Done
```

Figure 3: Slower Forward A*





```
Time of Algorithm:  2.7967705
Number of Expansions:  1821
Path Length:  212
Done
```

Figure 4: Faster Backward A*

Part 4: Heuristics in Adaptive A*

The project argues that "the Manhattan distances are consistent in grid worlds in which the agent can move only in the four main compass directions." Prove that this is indeed the case.

The Manhattan Distance is a heuristic that calculates the distance between two nodes limiting the movement to strictly vertical and horizontal movements. Since we limit the agents movement to only vertical and horizontal moves, it means that The Manhattan Distance will be a perfect estimation on a non blocked grid, or an underestimation on a blocked grid, making this heuristic consistent. If the agent was allowed to move diagonally, then this heuristic would be an overestimation because it would take less moves for the agent to get to the goal.

Furthermore, it is argued that "The h-values $h_{new}(n)$)... are not only admissible but also consistent." Prove that Adaptive A* leaves initially consistent h-values consistent even if action costs can increase.

To prove this we will assume that $h(n)$ is the Manhattan Distance and $h_{new} = g(goal) - g(n)$
For a heuristic to be consistent, it will have to follow this inequality:

$$h(n) = c(n, a, n^{'}) + h(n^{'})$$

Now to prove that $h_{\text{new}} = c(n, a, n^{'}) + h_{\text{new}}(n^{'})$ We will have to substitute $h_{new} = g(goal) - g(n)$ into the above formula.

$$g(goal) - g(n) \leq c(n, a, n^{'}) + g(goal) - g(n^{'})$$

Simplify:

$$g(n) \geq g(n^{'}) + c(n, a, n^{'})$$

Since $c(n, a, n^{'})$ is our action cost and can only be equal to one, our formula is consistent. To prove that h(n) will continue to be consistent even while action cost increase we will substitute into the formula:

$$h_{new}(n) \leq h_{new}(n^{'}) + c(n, a, n^{'}) \leq h_{new}(n^{'}) + c(n, a, n^{'})^{'}$$

Proving that it is consistent even if the action costs can increase.

Part 5: Heuristics in Adaptive A*

Implement and compare Repeated Forward A* and Adaptive A* with respect to their run-time. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both search algorithms should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly.

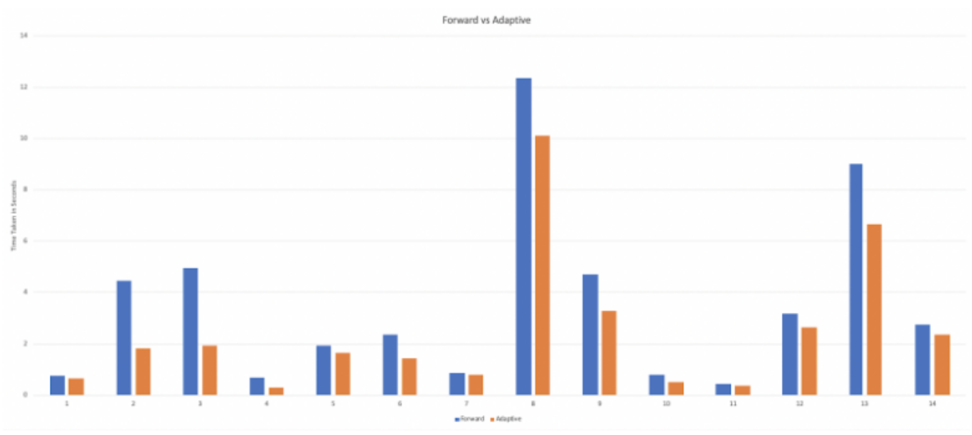
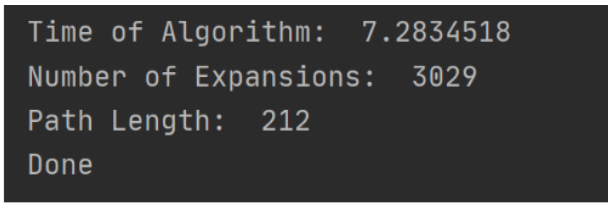


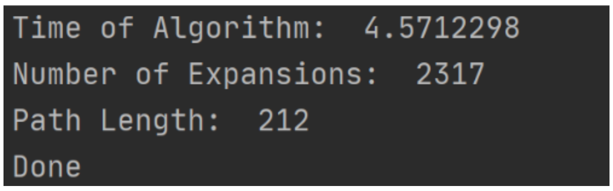
Figure 3: Avg run-times: Forward A*: 3.51s, Adaptive A*: 2.46s

After comparing Repeated Forward A* and Adaptive A*, we see that Adaptive A* runs much faster than Repeated Forward A* on an average aside from a few cases where the run-times are closer. This is because Adaptive A* results in a much lower number of expansions which changes the heuristic from the previous run of A*.



```
Time of Algorithm:  7.2834518
Number of Expansions:  3029
Path Length:  212
Done
```

Figure 1: Repeated A*



```
Time of Algorithm:  4.5712298
Number of Expansions:  2317
Path Length:  212
Done
```

Figure 2: Adaptive A*

Part 6: Memory Issues

Suggest additional ways to reduce the memory consumption of your implementations further.

For us to reduce the amount of memory consumed we could change the way we implement our class Node. Currently it can hold the following attributes: a nodes value (int), x-coordinate (int), y-coordinate (int), g-value (int), h-value (int), f-value (int), neighbors (list), previous (node), adjust (bool). We can reduce these attributes by holding external maps of each of the values g, f, and h. We could calculate the neighbors during each loop instead of storing them as a list. We could also use a pointer for previous instead of sending the reference to the node.Instead of using a grid like structure we could have implemented the structure using just x and y values stored in a tuple or list.

Then, calculate the amount of memory that they need to operate on grid-worlds of size 1001 × 1001 and the largest grid-world that they can operate on within a memory limit of 4 MBytes.

We use sys.getsizeof(cols[i][j]) to get the size of each cell, this came out to be 24 bytes.

To get the size for a 1001x1001 grid:

$$1001 * 1001 * 24 = 24.05\text{MB}$$

We are able to run a grid of size 408x408 when limited to 4MB.