**Part 1 - Understanding the Methods**

A. *Explain why the first move of the agent for the example search problem from figure 8 is to the east rather than the north given that the agent does not know initially what cells are blocked.*

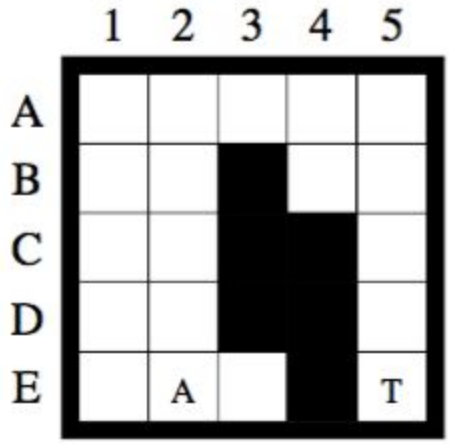



Figure 8: Second Example Search Problem

a. The agent starts in cell E2, as such they are only able to "see" cells {E1, D2, E3}. All of these cells are added to the open list. For each cell g ( the move cost), h(heuristic distance to goal state), and f = g + h have to be calculated. In this problem the heuristic is manhattan distance and move cost between cells is constant at 1 since we cannot move diagonally. Therefore, we have {E1 - f(5), D2 - f(5), E3 - f(3)} A* is greedy in the sense it will move to the cell with the lowest f cost. Since E3 has the lowest f cost due to the lowest heuristic distance to the goal it will explore that cell first.

B. *This project argues that the agent is guaranteed to reach the target is 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. Prove that the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of unblocked cells squared.*

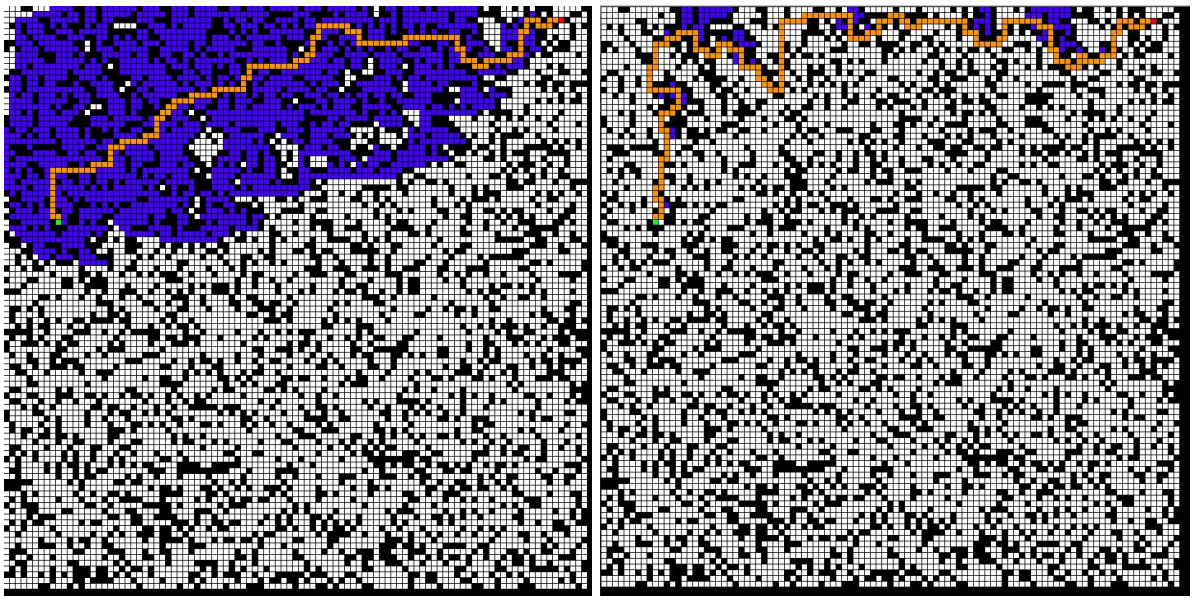
a. If the grid-world is finite then it can be searched exhaustively in finite time using A*. Given that every step has $cost > \varepsilon \; for \; some \; \varepsilon > 0$ and the branching factor is also finite then there is a finite number of expansions required. Since the number of expansions is finite we must be able to complete them in finite time.

b. As previously stated the A* algorithm terminates once the open list is null or the closed set = $|number \; of \; unblocked \; cells|$ . Given an $n \, x \, n$ grid-world we have $n^2$ total cells, let $b$ of those cells be blocked. Then we have $n^2 - b$ unblocked cells that we must explore (add to the closed list). Let $q = n^2 - b$ . The maximum distance our agent can move from one cell to any other cell in the grid-world is $q$ steps (the agent goes through every unblocked cell to get there). This can happen a maximum of $q$ times before every cell has been visited . Thus our agent can move at most $q^2$ times before every cell has been removed from the open list and added to the closed list. At this point the agent will return that there is no path.

**Part 2 - The Effects of Ties**

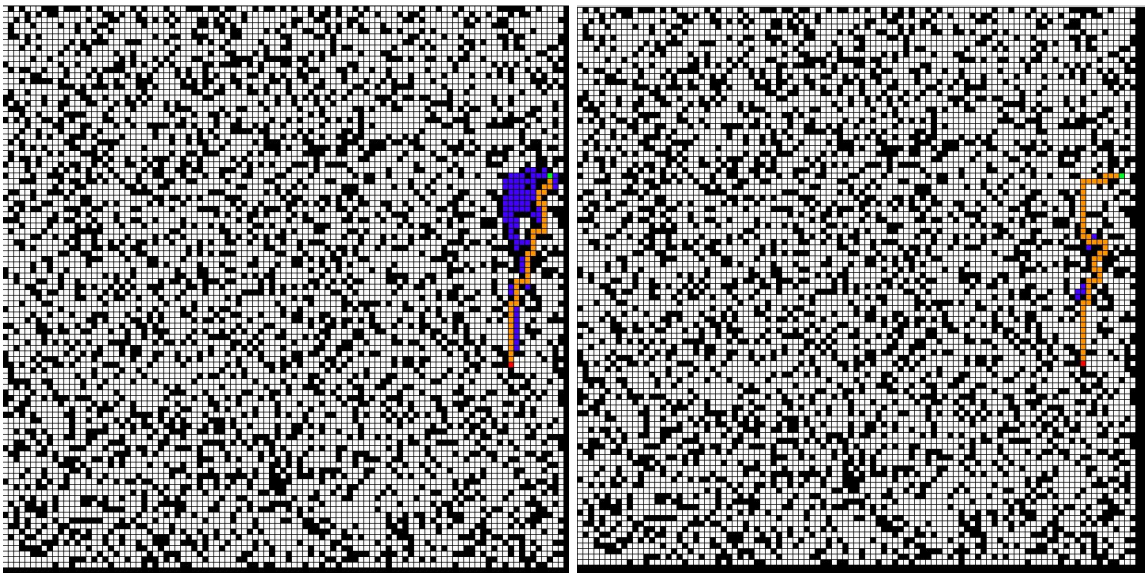
My observations lead me to notice that the number of explored cells greatly decreased with the use of the higher g score. Below are some examples of mazes solved with both tie breaking favoring lower g scores and higher g scores. The explored cells are colored blue, the shortest path is colored orange, start and goal are green and red.

**Lower G Score** **Higher G Score**

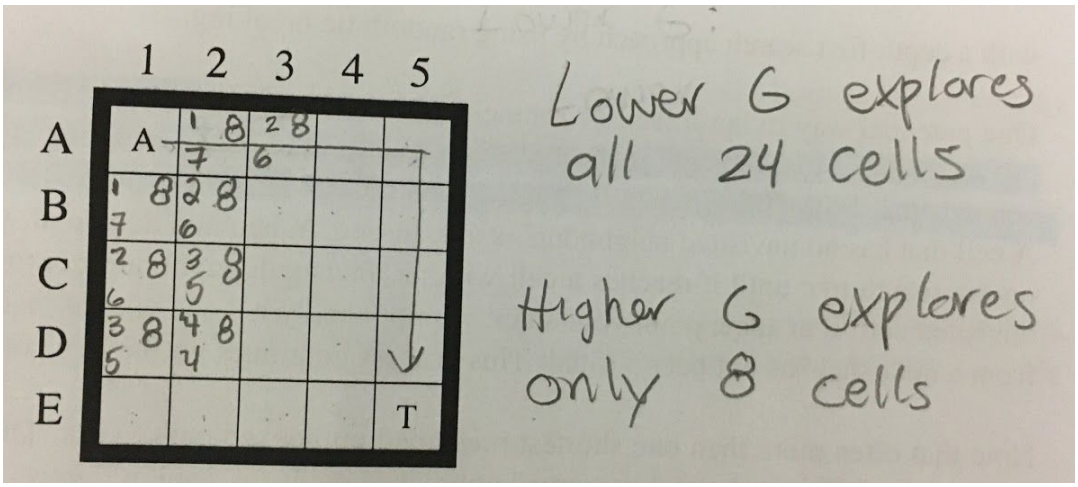


The algorithm favoring cells with a high g score explored only 293 cells compared to the 1963 explored by lower g tie breaking. Lower g explored over 6 times as many cells as higher g. The shortest path found the the higher g score has length of 200 cells. This is about 40% longer than the path found by lower g of only 142 cells. It appears that higher g score tie breaking can cause a drastic reduction in the number of cells explored. This leads to getting a solution faster and more realistically simulates an actual agent. However, this doesn't come without a price. The path returned when favoring high g scores can be significantly longer than the optimal solution. It is a classical problem where we are favoring speed over accuracy. Below is another example.

**Lower G Score** **Higher G Score**



Again the lower g score version opened more cells and found a shorter path. However, in this example the difference was much smaller. 119 cells were explored by lower g only about twice as many as higher g (57 cells). Path length was 43 for lower g vs 51 for higher g, which is almost 19% longer. Based on these results and some other mazes I looked at it appears that as the distance from start to the goal position increases so does the percentage of error in the shortest path returned by higher g. Let's take a look at an example to figure out exactly what is happening.

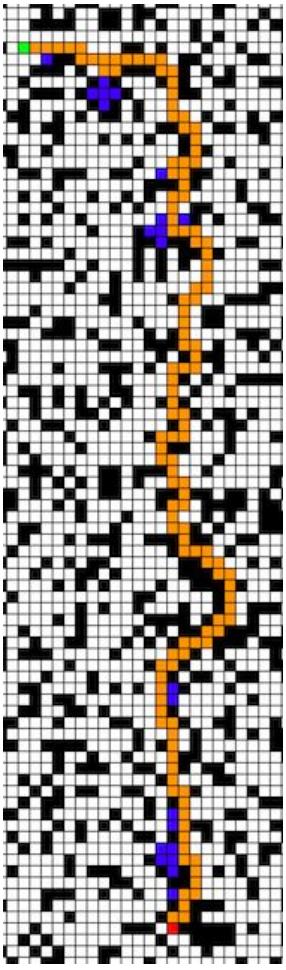



Given the graph above lower g will explore (add to the closed list) all 24 cells before returning the shortest path, lower g A* will explore all cells with a cost less than the cost of the optimal path. Higher g will only explore (add to the closed list) 8 cells and also returns an optimum path in this case. The starting cell is (0, 0) and the first move by lower g is to calculate f scores for (1, 0) and (0, 1). Both cells have the same h, g, and f cost so we have our first tie right away. Lower g with explore the cell that was added to the heap last (lets say its (0,1), generate its neighboring cells and notice they all also have an f score of 8. At this point lower g will pop the cell with the lowest g score which will be (1, 0). Lower g short of fans out evenly in all directions that are towards the goal. So it will continue to explore every cell based on its g cost until all cells with a cost less than the optimal path have been added to the closed list (explored).

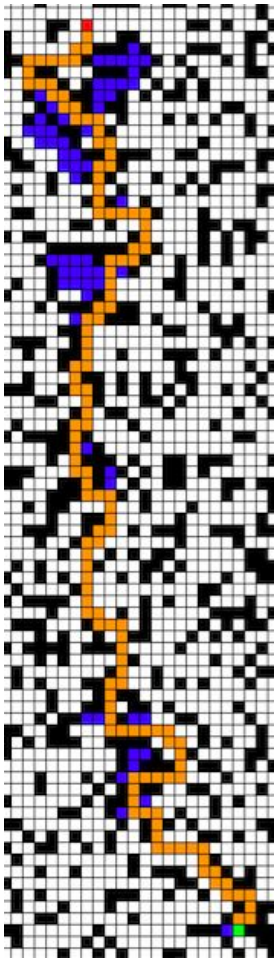
Higher g will start out exactly the same way as lower g, it will calculate f scores for (1, 0) and (0, 1). Let's assume the last one added to the heap will be explored (added to the closed list) let's also assume higher g explores (0, 1) first. The f score for all of its neighbors will be calculated, again we have a tie all the cells have an f score of 8. However, higher g will not move back and explore (1, 0) because that cell has an g cost of 1. It will explore either (2, 0) or (0, 2) since they have an g cost of 2. Higher g will continue to move "forward" towards the goal in more narrow fashion than lower g does.

**Part 3 - Forward vs Backward**

*Forward Higher G - 1*

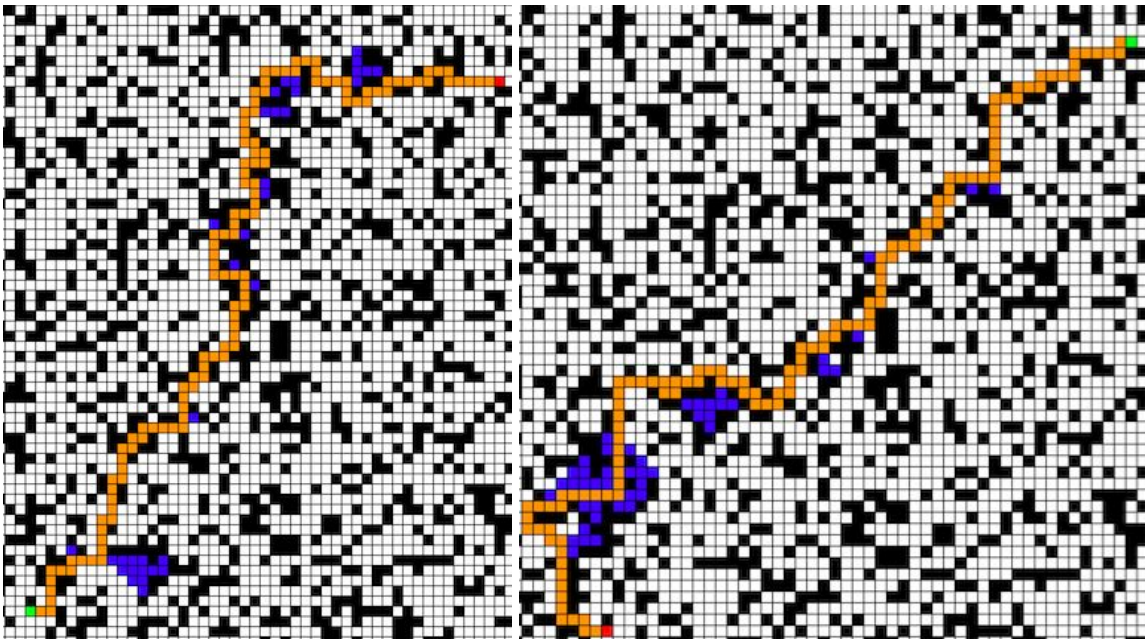


*Backward Higher G - 1*

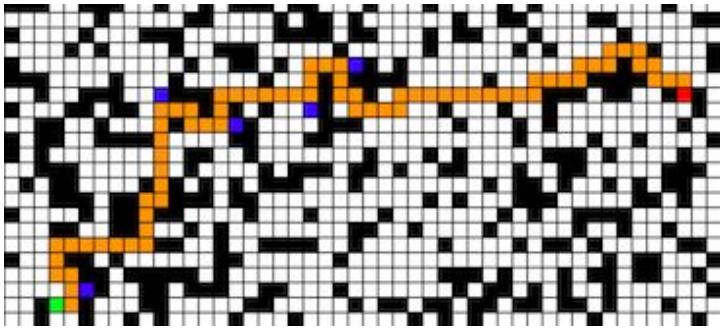


*Forward Higher G - 2*

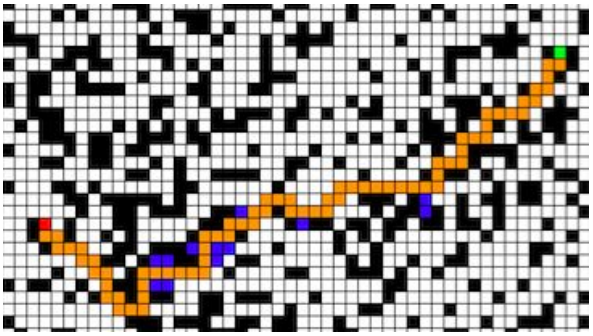
*Backward Higher G - 2*



*Forward Higher G - 3*



*Backward Higher G - 3*

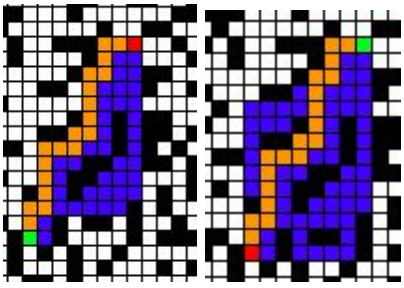


| Maze Number | Forward Path Length | Forward Cells Explored | Backward Path Length | Backward Cells Explored |
|---|---|---|---|---|
| 1 | 128 cells | 150 cells | 132 cells | 193 cells |
| 2 | 120 cells | 154 cells | 120 cels | 166 cells |
| 3 | 72 cells | 77 cells | 72 cells | 84 cells |

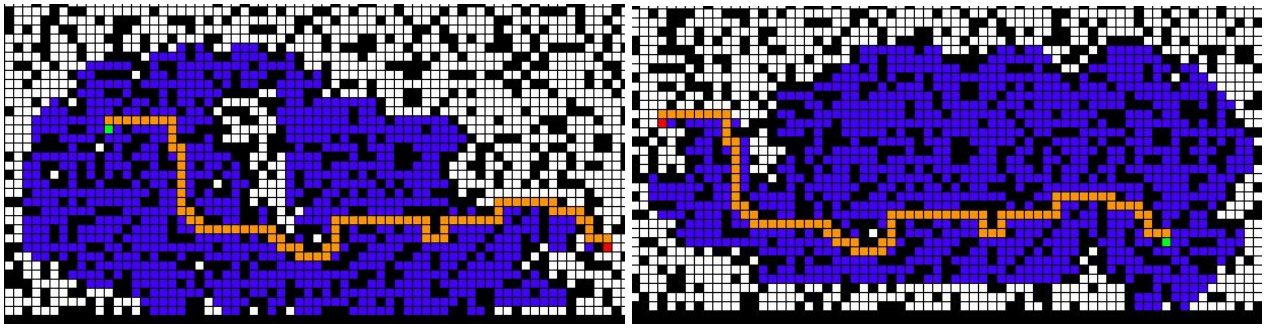
In general I observed that the reverse version of repeated A* performed slightly worse than the forward direction. The above 3 samples show that in general backward exploration explores more cells, even when it returns a path of the same length. I did however, observe a few trials where backward exploration outperformed the forward direction on the same maze. As you can see very clearly in example 3 the backward direction explored the maze in a completely different direction, thus it could run into more or less blocked cells resulting in more or less exploration. I think this evaluation is highly anecdotal, and not accurate in more widespread practice. The algorithm is exactly the same so they have the same complexity if we ran a large number of trials and analyzed the results, their difference would be insignificant. Interestingly enough although

not required by the assignment I also evaluated lower G forward and backward and found some interesting results.

*Forward Lower G - 1* *Forward Lower G - 1*



*Forward Lower G - 2* *Backward Lower G - 2*



| Maze Number | Forward Path Length | Forward Cells Explored | Backward Path Length | Backward Cells Explored |
|---|---|---|---|---|
| 1 | 20 cells | 57 cells | 20 cells | 75 cells |
| 2 | 86 cells | 893 cells | 86 cels | 987 cells |

Interesting that again it appears that backward exploration explores more cells. Lower G is consistent in the fact that the path returned is always the shortest path regardless of what direction we explore the maze. So overall it appears that backward exploration is marginally worse than forward. This appears to be the result of starting in a different location because A* in this form will expand only cells with a cost less than the cost of the optimal path. We encounter obstacles in a different manner which can affect how we expand our search for better or worse.

**Part 4 - Heuristics in the Adaptive A***

*A heuristic $h(n)$ is consistent if, for every node n and every successor $n'$ of $n$ generated by any action $a$, the estimated cost of reaching the goal from $n$ is no greater than the step cost of getting to $n'$ plus the estimated cost of reaching the goal from $n'$*

Let's first show that the values of $f(n)$ are nondecreasing along any path. Given a cell $n$ with successor $n'$ and our heuristic function equal to the Manhattan distance then:

$$f(n') = g(n') + h(n')$$
$$g(n') = g(n) + c(n, a, n')$$

Since we are limited to only moving in the four cardinal directions the h-value of any successor of $n$ can only increase or decrease by one cell. Thus $h(n') = h(n) \pm 1$. If $h(n') > h(n)$ then it follows that

$$f(n') \geq f(n)$$

If $h(n') < h(n)$ we have $f(n') = g(n) + c(n, a, n') + h(n')$

Given that the move cost is greater than zero. We see that $h(n')$ can decrease by at most one cell, but we moved one cell to get from $n$ to $n'$ which offsets the decrease in manhattan distance. Therefore,

$$f(n') \geq f(n) \text{ for all } n$$

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

The initial or first round of heuristic values in Adaptive A* are generated using the Manhattan distance and were already proven to be consistent above. First we show that the heuristic values cannot decrease due to being updated by the Adaptive A*. This is because the heuristic value went from $h(s) = manhattan\ distance$ which is already idealistic (admissible) to $h(s) = g(s_{goal}) - g(s)$. Since $g(s_{goal})$ is the length of the actual shortest path found from the previous A* it follows that $g(s_{goal}) \geq h(s) = manhattan\ distance\ from\ g(s)\ to\ g(s_{goal})$. Taking this one step further we see that $g(s_{goal}) - g(s) \geq h(s)$.

Next we examine the case where step cost increase. Let $c$ denote the initial cost and $c'$ denote the updated cost. Then we have $h(s) \leq h(succ(s, a)) + c(s, a) \leq h(succ(s, a)) + c'(s, a)$. Since we already showed above that updated heuristics cannot decrease we can now declare that the heuristic values are consistent.

**Part 5 - Heuristics in the Adaptive A***

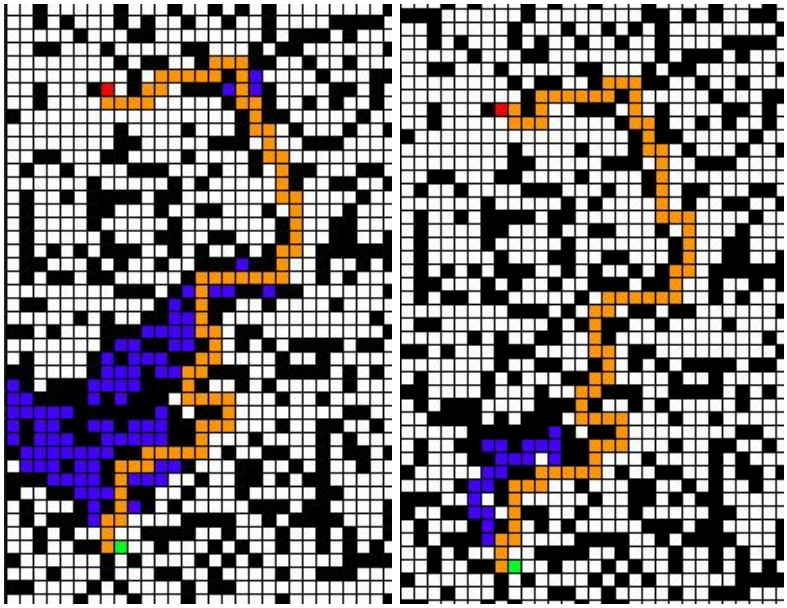
I ran lower g, higher g, and adaptive A* (with 4 rounds) on all of the mazes and got the following average times

```
Average time taken for lower G:     0.006911683082580567
Average time taken for Higher G:    0.0028015804290771485
Average time taken for Adaptive A*: 0.009691429138183594
```

I noticed that the number of rounds for adaptive A* was a general approximation of how much longer it takes to run than A* with higher g tie breaking. For example $0.002801 \times 4 = 0.011204$. It makes sense that Adaptive A* doesn't quite take the full amount of time as each round usually results in less cells being explored, thus a

reduced running time. I also looked at the number of cells explored and the length of the shortest path found here is a sample result.

*Higher G Vs. Adaptive A* - 2 rounds*

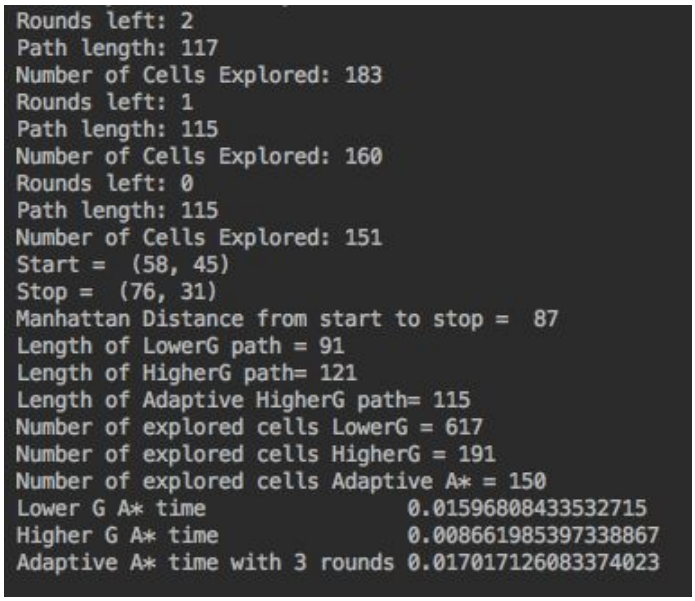


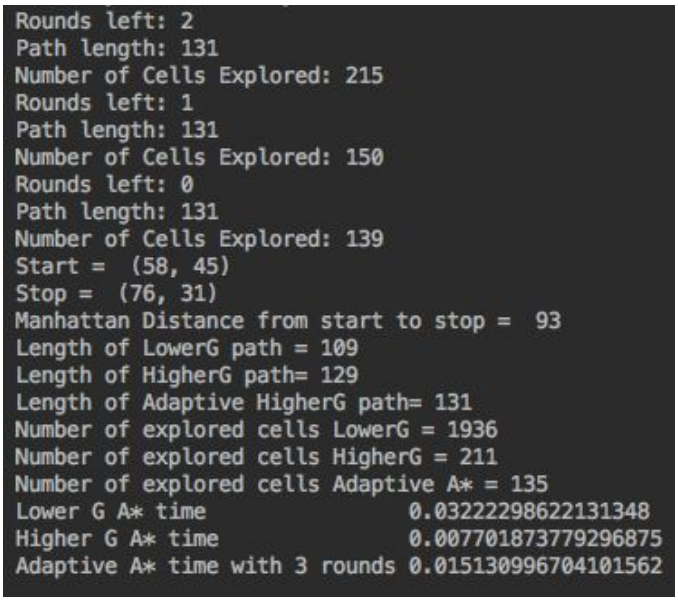
How many rounds of Adaptive A* makes sense?

```
Length of LowerG path = 67
Length of HigherG path= 73
Length of Adaptive HigherG path= 69
Number of explored cells LowerG = 483
Number of explored cells HigherG = 109
Number of explored cells Adaptive A* = 69
Lower G A* time                0.013382911682128906
Higher G A* time               0.0032699108123779297
Adaptive A* time with 2 rounds 0.005064964294433594
```

```
Length of LowerG path = 67
Length of HigherG path= 81
Length of Adaptive HigherG path= 79
Number of explored cells LowerG = 468
Number of explored cells HigherG = 164
Number of explored cells Adaptive A* = 81
Lower G A* time                0.019383907318115234
Higher G A* time               0.009475946426391602
Adaptive A* time with 3 rounds 0.032476186752319336
```

Above you will find the results from running all three algorithms. First I'd like to point out that my results concluded that Adaptive A* only makes sense if you perform less than three rounds. Using the above results you can see that Adaptive A* with 3 rounds takes longer to perform than A* with lower G tie breaking. For this reason it doesn't make any sense to use Adaptive A* considering that lower g tie breaking will return a shortest path and take less time. Another key note here is that adaptive A* often does not make significant improvements to the overall best path length. Looking at the above examples we have higher g paths of 73 and 81 vs adaptive paths of length 69 and 79. This is also coming at a price of approximately double the running time if two rounds are used. Additionally, the largest reduction in the number of expanded cells appears to occur after the first round and decrease rapidly after that.




Looking at the above results again you can see that the largest reduction in explored cells occurs between the first and second round. You will also notice that after a certain number of rounds the number of cells explored cannot be reduced as it is only expanding the shortest path. This is due to the fact that adaptive a* cannot expand cells that weren't expanded in a previous search. It should also be noted that the first round of Adaptive A* and A* with higher g tie breaking expand approximately the same number of cells. Overall, the total number of cells expanded by Adaptive A* depends on the number of rounds and the reductions that occur between rounds. Adaptive A* still expands significantly less cells than A* with lower g tie breaking. However, as the second example above shows Adaptive A* is not guaranteed to find a shorter path that standard A* with higher g tie breaking. This is due to the fact that ties are broken randomly, and successive rounds of Adaptive A* cannot expand cells that were not previously explored. Thus it can only improve on the original path so much. In general Adaptive A* with two rounds seems to be a compromise between running the full A* with lower g and A* with higher g. Overall, it performs right in between the two, expanding significantly less cells than lower g, and generally finds a shorter path than higher g.

**Part 6 - Memory Issues**

If you analyze my code you will see that I did not optimize my results with regards to memory usage. I wanted to be able to display the mazes in a clear fashion so I could visually analyze the results and understand the algorithm better. I chose to explore this question using real world results rather than theoretical sizes. That being said the original gridworld is stored inside a list of lists. In my case all 50 mazes took up about 1MB of space when stored with pickle, but these are rather small mazes. Additionally, the matrix representing the maze is very sparse. Given that each cell has about a 30% chance of being blocked, we could reduce the amount of memory needed to store the maze by 70% just storing the location of the blocked cells in a dictionary or map. Additionally, rather than using a tuple (x, y) as the key we can just assign the cells a single number in row and column major order. We start out with a given start location and goal state, we check our neighboring cells if they are in the dictionary and have a specific value such as -1 then we know they are blocked. If the cell is not in the dictionary then we know it is open and we can move there. We then generate the cell object with all values and store it in the dictionary with the appropriate key. Let's assume that we are using A* with higher g tie breaking. In a 101 x 101 maze this algorithm explored an average of approximately 100 cells. Some of the mazes can be rather short so let's round up to 150 to overestimate the cost. That means our dictionary would store about 3,000 blocked cells plus the 150 we explored so only about 3,150.

Additionally, in each cell object we can just store the f score, and the parent as ints. There is no reason to store the heuristic value as that can always be calculated. Furthermore, we do not need to store the g values. If you have the f score, we can calculate the heuristic, and we can perform $f_{score} - h(n) = g(n)$. I created a sample dictionary in python using pickle and saved the dictionary with all 3,150 values and it took up 34KB of space. Let's see how much memory we'd need to store a 1001 x 1001 maze. Again, let's assume 30% of cells are blocked and let's up our explored cells from 150 to 2,000. So we have 1,002,001 cells about 300,600 will be blocked and we will explore approximately 2,000 cells. That gives us a total of 302,600 cells. Generating simulated tuples and filling a dictionary with the 302,600 values, and saving it with pickle the file took up 4.9MB of space. Thus, we are already over the limit. I dropped the size down to 909 x 909 and I was able to store all the value right at 4MB. This dictionary had 248,500 values which means this is the upper bound on the number of values that can be stored. More importantly, we need to account for a worse case scenario. This is where the entire maze is explored and no path is found. That means there can only be about 248,000 values. This gives us a more realistic limit of a grid world with size 497 x 497. At this point the problem became very interesting as I started to use sys.getsizeof(). I then noticed that pickle was drastically, reducing the size of the file through the use of compression. I then began to explore the actual size of things in python and noticed that an int takes up 28 bytes not 32 bits! Using the following code I was able to find the maximum number of tuples that can be stored in a python dictionary while staying under 4MB.

```
four_mb_max = {}

while sys.getsizeof(four_mb_max) < 4000000:
    x = random.randint(0, 1000000)
    four_mb_max[x] = (x, x)

print(len(four_mb_max))
```

The function returned a dismal value of 87,382. So if we explored the whole maze it could be a max size of 295 x 295. This is much smaller than I expected, but that is largely due to the fact that the actual amount of space taken up by an int is 28 bytes not 32 bits which is seven times larger than anticipated. Overall, I think just storing the blocked cells is a good way to reduce the amount of memory needed, but alternative means to store cell data need to be thought of.