

# Project Report for Intro to AI HW1

Daniel Pattathil, Kush Baldha, Frank Jiang  
dp865, kb727, fj99

February 27, 2019

## Contents

<b>1</b>	<b>Understanding the Methods behind A* Search</b>	<b>2</b>
1.1	Why would the agent travel east? . . . . .	2
1.2	Is the A* Algorithm consistent? . . . . .	2
1.2.1	Validity . . . . .	2
1.2.2	Number of moves . . . . .	2
<b>2</b>	<b>Tie-breaking between Low and High G-Values</b>	<b>3</b>
<b>3</b>	<b>Differences between Forwards and Backwards</b>	<b>4</b>
<b>4</b>	<b>Tables and figures</b>	<b>5</b>
<b>5</b>	<b>Differences between Repeated and Adaptive</b>	<b>5</b>
<b>6</b>	<b>Memory Issues - Optimize the Problem</b>	<b>6</b>

## Setting up the environment

We decided to use Python to create and store our mazes for this project. In the case of the code's source, we referenced an online algorithm that focused on the two main features of the maze, density and complexity. We wanted fairly complicated mazes, but consistently kept paths available for the agent to reach the target. We have stored the different mazes as text files where 0's and 1's represent the open and blocked walls respectively. The mazes can be read back in for usage in the different search types and are displayed using numpy and matplotlib.

## 1 Understanding the Methods behind A\* Search

### 1.1 Why would the agent travel east?

To go into why the agent would first attempt to move east, we have to look at the general logic and decision making involved in the A\* Search that we are implementing. The primary influencing factor for this specific search is the  $h(s)$  or the Manhattan distance that we are using. Because we do not know where blocks are located yet, the agent believes traveling to the east would be the fastest. When deciding in which cardinal direction to go, we would check the different  $h(s)$  values and see that the goal state causes the lowest overall  $f(s)$  ( $= g(s) + h(s)$ ) to be on the east side for the initial state.

### 1.2 Is the A\* Algorithm consistent?

#### 1.2.1 Validity

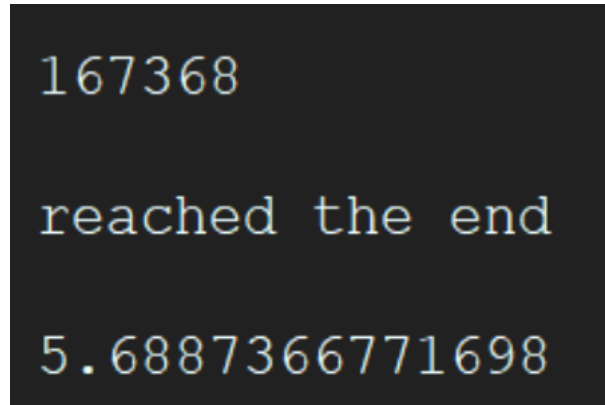
In the case of this search algorithm, as with many others, it should be true that either the search will find the target or find that there is no possible path. The A\* search always checks open spaces surrounding the current location of the agent. When the agent realizes that there are no nearby options to move locally, it checks other locations on the queue and searches from there. Because we do not limit the amount of movement, we will eventually reach/consider every accessible open space. This does not mean that we will access every open space, however, it does mean that the path will be found if open spaces lead to the target. If the open spaces end up being boxed off or the target is somehow blocked, the search will end, because all accessible spaces will have been explored and it will return that there was no possible path. The different closed and open lists are utilized to ensure that everything will be explored without redundancy.

#### 1.2.2 Number of moves

In the case of a maze that has all of the unblocked squares connected together, with the agent and target far apart, the total amount of moves for the agent would be  $n$  which we will say represents the amount of unblocked squares. Then, if the path itself is a swirl surrounding the target or some other formation with many collisions, every move could cause a restart of the A\* search. With both of these parts considered, it would end up that the upper bound would be  $n^2$  overall.

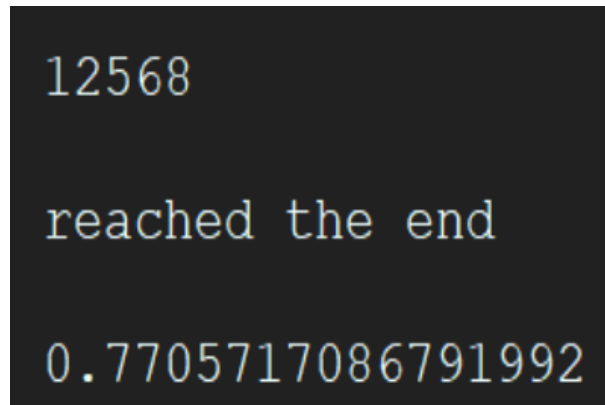
## 2 Tie-breaking between Low and High G-Values

In the case of our implementation, we found and concluded that using larger g-values when running Repeated Forward A\* yields a significantly lower number of comparisons and faster runtime. The findings are depicted below:



```
167368  
reached the end  
5.6887366771698
```

Figure 1: Running Repeated Forward A\* favoring Smaller G-values



```
12568  
reached the end  
0.7705717086791992
```

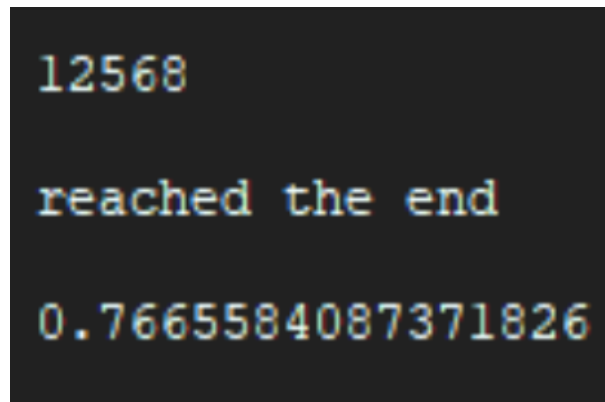
Figure 2: Running Repeated Forward A\* favoring Larger G-values

We have deduced that the reason as to why favoring larger g-values results in a significantly lower number of comparisons and runtime versus favoring smaller g-values is because we are able to choose cells/locations closer to the target. The g value is the length of the path that we have currently taken and preferring a larger g value means we are preferring a longer path. We have expanded more on the larger g value path and therefore would be a better decision to prefer this path. Conversely, if we favor smaller g-values, it would take us more comparisons and time to compute because the agent is exploring cells that are more likely to be further away from the goal and closer to its starting position. Typically, when we favor smaller g-values we are exploring numerous cells that cause us to waste comparisons and exhaust more running time. From the agent's perspective, it does not make sense to explore areas near the starting position when you want

to reach the goal as fast as possible. Therefore, if we favor larger g-values, we are able to traverse through the cells that have a higher success and probability of reaching the target. Furthermore, a larger g value means that we have a smaller h value. A smaller h value means that we are more likely to be closer to the target.

### 3 Differences between Forwards and Backwards

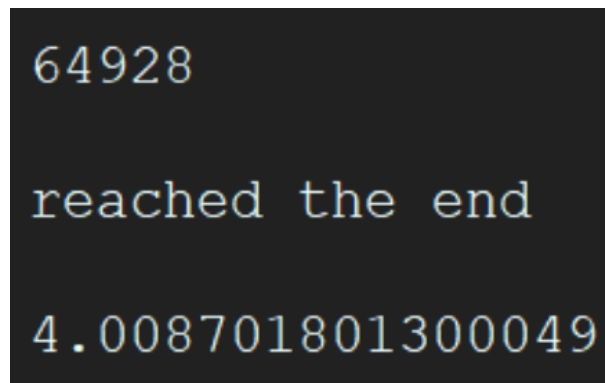
For our particular types of mazes, we regularly noticed that Forward Repeated A\* Search ended up having faster search times. Our results can be depicted below:



12568  
reached the end  
0.7665584087371826

A terminal window with a black background and yellow-green text. It displays three lines of output: the number of comparisons, a status message, and the runtime.

Figure 3: Running Repeated Forward A\*



64928  
reached the end  
4.008701801300049

A terminal window with a black background and yellow-green text. It displays three lines of output: the number of comparisons, a status message, and the runtime.

Figure 4: Running Repeated Backward A\*

As can be seen through the different numbers in our pictures, both the total amount of comparisons and the runtime are drastically different. The reason why Repeated Forward A\* performs massively better than Repeated Backwards A\* is because in the latter case each time we run A\* search, since we are traversing from the target cell to the start cell the larger f-value causes the agent to explore unnecessary cells when traversing backwards. In our actual

experiment, the results support our claim, because the number above the time indicates how many cells were expanded in the backwards search. From an immediate look, we can see that the amount of comparisons and time required increased by a factor larger than 5. In the case of the backwards having so many more comparisons, we believe it has to do with all of the open options that are available to the agent when going from the backwards path. This is very different from the Forwards A\* which manages to see less and choose a more efficient path without opening too many splitting paths.

## 4 Tables and figures

It is important to note the reason as to why we can prove that the argument: "the Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions." We can do a proof by contradiction to prove that Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions.

Let's say that it is not consistent so:  $h(a) > c(a, b) + h(b)$

We can then subtract  $h(b)$  from both sides and get  $h(a) - h(b) > c(a, b)$

Here we notice that  $h(a) - h(b)$  is the subtraction of the Manhattan distances of  $a$  and  $b$ . We can also write this as  $(x(a) - x(b)) + (y(a) - y(b))$ .

The difference of  $x$  is the distance between of both coordinates in the  $x$  axis and the difference of  $y$  is the distance between of both coordinates in the  $y$  axis. We can look at the addition of these difference as moving first in the  $x$  axis and then in the  $y$  axis. For the  $c(a, b)$  to be less than  $h(a) - h(b)$ , we must take a path that is diagonal and moves in the  $x$  and  $y$  plane at the same time. We know that this is not possible because we must move in the four main compass directions. Therefore we have proved this by contradiction.

## 5 Differences between Repeated and Adaptive

In our experiment, we found that Repeated Forward A\* with Adaptive was moderately faster than Repeated Forward A\*. The results are depicted as below:

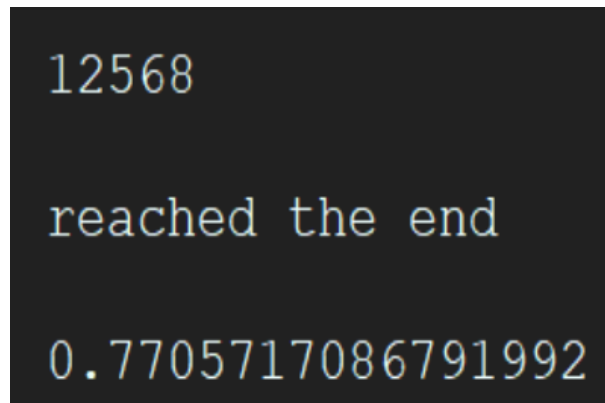


Figure 5: Running Repeated Forward A\*

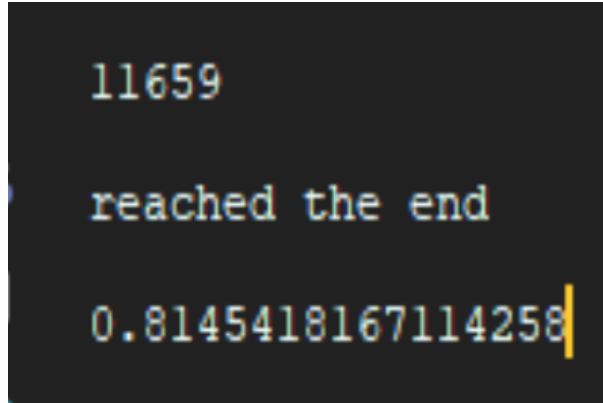


Figure 6: Running Repeated Forward A\* with Adaptive

It is important to note that while the number of comparisons for Repeated Forward Adaptive A\* is indeed lower than Repeated Forward A\*, the slight discrepancy in the running time can be attributed by the GPU’s hardware capabilities and other simultaneous CPU usage. The reason why Forward Adaptive A\* is more efficient than normal Forward A\* is because we are updating h-values each time and focusing our path to take expand less unnecessary nodes. Consequently, this slightly more dynamic approach allows us to avoid traversing less cells that the agent has previously visited before. By doing this we are efficiently choosing the path that the agent takes to get to the target cell and intuitively decrease the number of comparisons that are being computed.

## 6 Memory Issues - Optimize the Problem

In the case of increasing the gridworld size that we are working with, we have to make sure that our implementation is properly scalable. A good example of utilizing less memory is the usage of two bits per cell for pointers from our tree. The literal maze itself still requires to keep a matrix of values which could represent whether or not the maze is blocked, open, or blocked and seen cell values (three value types means two bits). We also need to keep values in order to know what the g and h values are for our actual calculations. Because an int is a larger data type and we know that g/h values will not be excessively large, we could try to store the values in separate shorts or specify bits to a the maze size. By keeping matrices, it is easy to reference coordinates simply by using row and column. If we wanted to be very space efficient, we could store the tree pointers bits and the maze open/blocked value bits inside of the same short int. In this case, there would be approximately 3 \* (short int) bytes per cell being used in memory. This would come out to about 12 bytes per cell with our estimation. For an 1001x1001 grid, we would have 12 \* 1001 \* 1001 which comes out to approximately 12024012 bytes or approximately 12MB. If we want to estimate how many cells we can have using just 4MB of memory, we would divide 4MB/12B and see that there are about 340 cells that we could keep track of. In the case of the square grid, that would come out to approximately an 18x18 grid.