Assignment 1

Fast Trajectory Replanning

Author:
Brian Y. Mahesh A.
brianyi14@gmail.com
ma1700@scarletmail.rutgers.edu

Supervisors:
Abdeslam Boularias
Aravind S.

CS520: Artificial Intelligence
Rutgers University

March 1, 2020

1 Setup Environments

Frameworks

We use the following main frameworks: numpy, pandas, matplotlib, and heapq. Numpy and pandas are used for data collecting and processing purposes. Matplotlib is used to visualize the maze worlds. Heapq is used for the binary heap data structure.

Data Structures

We use the following data structures: 2D array, linked list, and binary heap. We use a 2D array to contain the information for the maze world. We opt for a linked list structure for our A* algorithm closed lists for O(1) runtime when adding new nodes while still being able to track the entire closed list for algorithm debug purposes. We use a binary heap for our A* algorithm open lists to implement the priority queue that sorts the states in our desired order. We also choose a binary heap since its implementation allows us to easily remove the highest priority state, which is the primary function of our open list.

Maze World / States

Each maze world is a 101 x 101 sized grid where each state (cell) has a 30% of being blocked. Each maze world also contains a randomly generated spawn and target that is never blocked. While measuring algorithm runtimes, we set the spawn and target states to [5,5] and [95, 95] respectively for less variance in our results. We do not choose the corners since it is more likely that the surrounding blocks can be blocked, resulting in no available path. Each state represent an individual cell on the map, and contains both the necessary information for our A* algorithms to run and our agent's knowledge of the map. Note that our state class differs slightly between our three algorithms due to different heuristic update implementations.

Methods

We have three main A* algorithms: Forward A*, Backward A*, and Adaptive A*. Running each A* method will output whether the agent reaches the target, the runtime, and a visualization of the path the agent takes. Even though all three methods use the same 50 generated maze worlds, each method consists of three parts that differ in implementation. Each A* algorithm method consists of a state class, compute_path method, and main method that corresponds to the respective algorithm. The compute_path method is essentially the A* algorithm that determines the best path from agent to target state at any point in time. The main method uses the path calculated from the A* algorithm, and moves the agent accordingly as well as handles the output of the overall A* algorithm. In addition to our three main algorithms, we also came up with three print methods, `print_grid`, `print_agent_fpath`, and `print_agent_map` that we used to validate results. `print_grid` prints any maze so that we can see where the spawn state, target state, and blocked states are. `print_agent_fpath` prints out the final path the agent takes throughout the maze to get from the spawn state to target state; this visualization includes all of the blocked states in the maze. `print_agent_map` prints the current agent map at any given compute_path() iteration; this visualization shows only the blocked states the agent has encountered along with all the expanded states.

2 Understanding the Methods

2.1 East or North

Explain in your report why the first move of the agent (yellow) for the example search problem from Figure 2.1 (below) is to the east rather than the north given that the agent does not know initially which cells are blocked. Target state is in green.

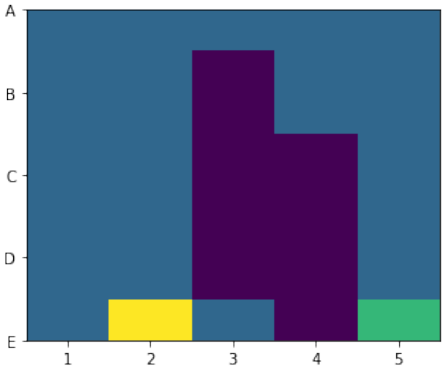

Figure 2.1: East or North

Given A is the start state and T is the goal state. Initially, the agent does not know which cells are blocked. Each state has 3 values, which are g(s),h(s) and f(s).

- g(s) :- g-value is the length of the shortest path from the start state to state s (present state) found by the A* search.
- h(s) :- h-value is the heuristic manually defined by the user. Here we are using Manhattan distance as the h-value.
- f(s) :- f-value estimates the distance from start state to goal state via current state.

Now we perform A* search from start state (A) to goal state (T). We push A (E2 cell) into the open list since it is the start state and we expand A (E2) and add its neighbour cells to the open list which are E1, E3, D2 cells. we move state A (E2) into the closed list as it is already expanded. Now we calculate the f-values of each cell that is in the open list. We choose the smallest f-value cell as the next cell to expand. Open list contains the following 3 cells:

- cell E1 :- g(E1) = 1 h(E1) = 4 f(E1)= g(E1) + h(E1) = 1 + 4 = 5
- cell E3 :- g(E3) = 1 h(E3) = 2 f(E3)= g(E3) + h(E3) = 1 + 2 = 3
- cell D2 :- g(D2) = 1 h(D2) = 4 f(D2)= g(D2) + h(D2) = 1 + 4 = 5

Cell E3 has the smallest f-value among all the cells in the open list so we expand cell E3. We add E3's neighbours, D3 and E4, into the open list. We move cell E3 into the closed list as it is already expanded. Since E2 is already expanded and is in the closed list, we will not put it into the open list again. Open list contains the following 4 cells:

- cell E1 :- g(E1) = 1 h(E1) = 4 f(E1)= g(E1) + h(E1) = 1 + 4 = 5
- cell D2 :- g(D2) = 1 h(D2) = 4 f(D2)= g(D2) + h(D2) = 1 + 4 = 5
- cell D3 :- g(D3) = 2 h(D3) = 3 f(D3)= g(D3) + h(D3) = 2 + 3 = 5
- cell E4 :- g(E4) = 2 h(E4) = 2 f(E4)= g(E4) + h(E4) = 2 + 2 = 4

Cell E4 has the smallest f-value among all the cells in the open list so we will expand cell E4. We push cell E4 neighbours into the open list, which are D4 and E5 (Goal state). We move cell E4 into the closed list as it is already expanded. Since E3 is already expanded and is in the closed list, we will not put it into the open list again. Open list contains the following 5 cells:

- cell E1 :- g(E1) = 1 h(E1) = 4 f(E1)= g(E1) + h(E1) = 1 + 4 = 5
- cell D2 :- g(D2) = 1 h(D2) = 4 f(D2)= g(D2) + h(D2) = 1 + 4 = 5
- cell D3 :- g(D3) = 2 h(D3) = 3 f(D3)= g(D3) + h(D3) = 2 + 3 = 5
- cell D4 :- g(D4) = 3 h(D4) = 2 f(E4)= g(D4) + h(D4) = 3 + 2 = 5
- cell E5 :- g(E5) = 3 h(E5) = 1 f(E5)= g(E5) + h(E5) = 3 + 1 = 4

Cell E5 (goal state) has the smallest f-value among all the cells in the open list so we expand E5. Since it is the goal state(T), A* search stops here. Path found by the A* search is:

- E2(A) –> E3 –> E4 –> E5(T)

Since the agent does not know which cells are blocked or not, the agent follows this path found by A* search and moves to the east initially.

2.2 Finite Maze Worlds and Target Discovery

This project argues that the agent is guaruteed to reach the target if it is not separated from it by blocked cells. Give a convincing argument that the agent in finite gridworlds 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.

Given that the agent is in finite gridworlds, this means there are a finite number of cells. These cells can be either blocked or unblocked. The agent moves only in the four compass directions north, west, south, east to its neighbouring cell given that it is unblocked. We find the shortest unblocked path using A* search and then move the agent along this path. If the agent encounters a blocked state, we increase the action costs of the blocked cell and restart the A* search with the new agent location. If there is any blocked region in the gridworld, the agent is outside this region and will never enter since it can see that the region is blocked. Similarly, for every unblocked region, the agent can traverse through its entirety. Thus, given that a path exists, the agent finds the shortest path to the goal state given it is a finite gridworld. If no path exists, the agent discovers that it is impossible to reach the goal state.

A* search never expands a cell that is already expanded, so it will never put duplicate expanded cells into the closed list. As a result, A* search never enters an infinite loop. In every case, the agent in finite gridworlds is guaranteed to reach the target or discover that no path exists.

Assume that total number of unblocked cells in the given gridworld is N. The whole execution of repeated forward A* or repeated backward A* mainly considers the two following functions:

- compute_path() :- Finds path from the current state to goal state if it exists using the current knowledge of the gridworld.
- main method :- We move the agent along the path that is found by compute_path(). If a cell is blocked, we increase the action costs of the corresponding states and rerun compute_path().

Now, each iteration of the compute_path() takes at most N moves since we do not expand each cell more than once. Let the number of states our agent travels through each iteration of computepath() as M. The worst case scenario is that compute_path() expands every unblocked cell and the agent travels along this path.

$$M \leq N \tag{2.1}$$

Once the path is found, the agent moves along this path. If the agent is blocked, then we recompute the shortest path. The worst case scenario is that each time the agent tries to move along the path, it gets blocked after one step so that each time we have to rerun compute_path(). Let's assume the number of times the agent is blocked or compute_path() is executed as I. Since the number of blocked cells is always less than or equal to N:

$$I \leq N \tag{2.2}$$

Using the above two equations we get,

$$MI \leq N^2 \tag{2.3}$$

N is total number of unblocked cells in the gridworld.
M is number of moves agent travels during each iteration.
I is the number of times compute_path() is run.
MI is total number of moves travelled by agent.
Thus, the number of moves the agent takes until it reaches the target or discovers that no path exists is bounded from above by the number of unblocked cells squared.

3 The Effects of Ties

Implementation Details: Our original Forward A* algorithm (`forward_a`) breaks ties between states with the same f-value by selecting the state with the higher g-value. We implement this into our priority queue (binary heap) by calculating a single integer priority value, $c \times f(s) - g(s)$, and selecting the smaller priority value. This priority value is directly proportional to $f(s)$ and is indirectly proportional to $g(s)$ when there is a tie between $f(s)$. We adjust our Forward A* algorithm to settle tie breakers by selecting the state with the lower g-value (`forward_a2`); for comparison purposes, our implementation simply first compares f-values, and then g-values if necessary. We run both algorithms on all 50 mazes and collect data on the runtimes. The runtime for an algorithm is equivalent to the total expanded states of that algorithm. We choose to measure total expanded states rather than the total number of uniquely expanded states since we feel that this metric would be more telling of runtime. If our A* algorithm searches over the same expanded states each iteration, taking up valuable time, we would like our runtime measurement to reflect that. Keep in mind that our spawn and target states are always at [5,5] and [95,95] respectively for a more consistent comparison. Table 3.1 (below) shows the runtime for both algorithms and the number of times slower breaking ties with smaller g-values is compared to greater g-values for the first ten mazes.

Table 3.1: Greater vs Smaller g-value Runtime Comparison

	gGreaterRuntime	gLesserRuntime	timesSlower
0	9270	236332	24.494283
1	8547	332937	37.953668
2	8924	258128	27.925146
3	5229	275652	51.716007
4	9669	297389	29.756955
5	8153	164678	19.198455
6	6398	180787	27.256799
7	7421	178488	23.051745
8	7467	207075	26.732021
9	6180	258832	40.882201

Using the data collected from all 50 runs, we determine that settling tie breakers by selecting for smaller g-values is always slower compared to selecting for greater g-values. We use the data from Table 3.1 to generate Figure 3.1 (below), a visualization of the runtime discrepancy.

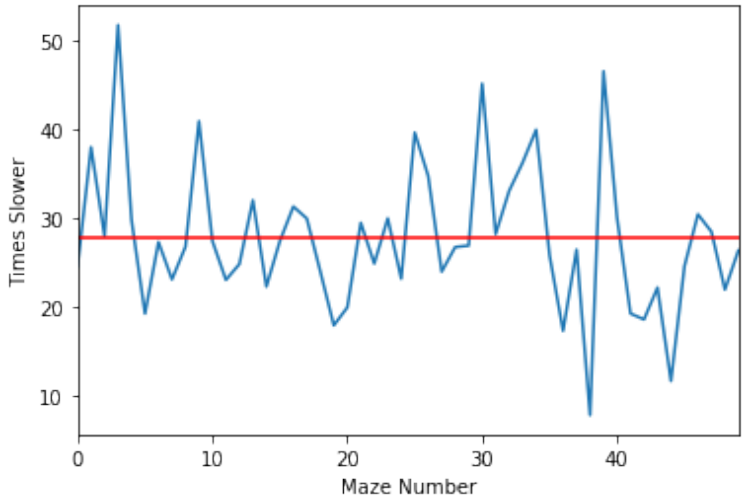

Figure 3.1: Greater vs Smaller g-value Runtime Comparison

Figure 3.1 (above) is the visualization of column 3 (`timesSlower`) of Table 3.1, and shows how many times slower `forward_a2` is with respect to `forward_a`. Since the graph is always positive for all fifty runs, breaking ties by selecting for larger g-values is always faster than selecting for smaller g-values in our test mazes. On average, selecting for a smaller g-value is 27.74 times slower than selecting for a greater g-value, as represented by the horizontal red line. A brief explanation is that by opting for states with a larger g-value, we are selecting for states that are farther away from the spawn state and assumedly closer to the goal state. Let's delve into this explanation with the following example.

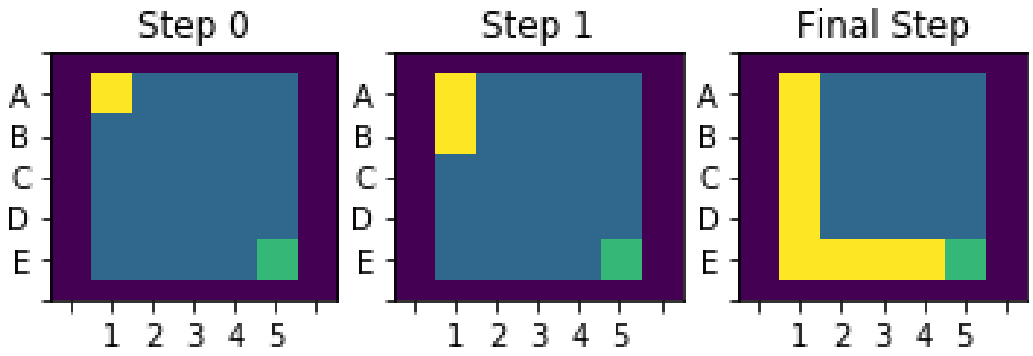

Figure 3.2: Forward A* (Larger g-value)

Figure 3.2 (above) shows how the Forward A* algorithm uses a larger g-value to solve tie breaks and computes a path from the agent to the target state. During our analysis, we will not reference our integer value ($c \times f(s) - g(s)$) implementation, but rather use the standard $f(s) = g(s) + h(s)$ metric for clarity. Step 0 is the initial maze where the spawn state is the yellow block at A1 and the goal state is the green block at E5.

Step 0: At A1, the only two states our algorithm can choose from our open list to expand are B1 and A2 based on their $f(s)$ value. $f(s) = g(s) + h(s)$, where $g(s)$ is the cost to travel from the spawn state and $h(s)$ is the distance to the goal state by the Manhattan heuristic. B1 and A2 have a $f(s)$ value of 8 since they both have a g-value of 1 being one step away from the A1, and both also have an h-value of 7. Since the g-values are equivalent, the Forward A* algorithm selects a random state and chooses B1, as seen by the extension of the yellow block, which now represents states expanded. It then removes A1 and adds C1 and B2 to the open list.

Step 1: At B2, the algorithm can now choose from C1, B2, and A2 from the open list to expand. A2 still has a $f(s) = g(s) + h(s) = 1 + 7 = 8$, B2 has a $f(s) = 2 + 6 = 8$, and C1 has a $f(s) = 2 + 6 = 8$. Since all three states have the same f-value, Forward A* now selects the largest g-value. In this case, it is a tie between B2 and C1, and one of the two is randomly selected. Regardless of the state we choose between B2 and C1, by selecting a larger g-value, Forward A* will always choose a state farther away from the spawn state.

Final Step: Fast forward a few steps to when our algorithm reaches the target state, and we see the benefit of breaking tie-breakers with selecting a larger g-value. Keep in mind, our algorithm doesn't hug an outer wall or move in an L shape, we choose B2, C2, D2, and E2 for the tie breakers in our representation to have a more intuitive visualization of the benefits of choosing a larger g-value. Now let's compare this to what happens when we use a Forward A* that selects the smaller g-value to break states with the same f-value in Figure 3.3 (below).

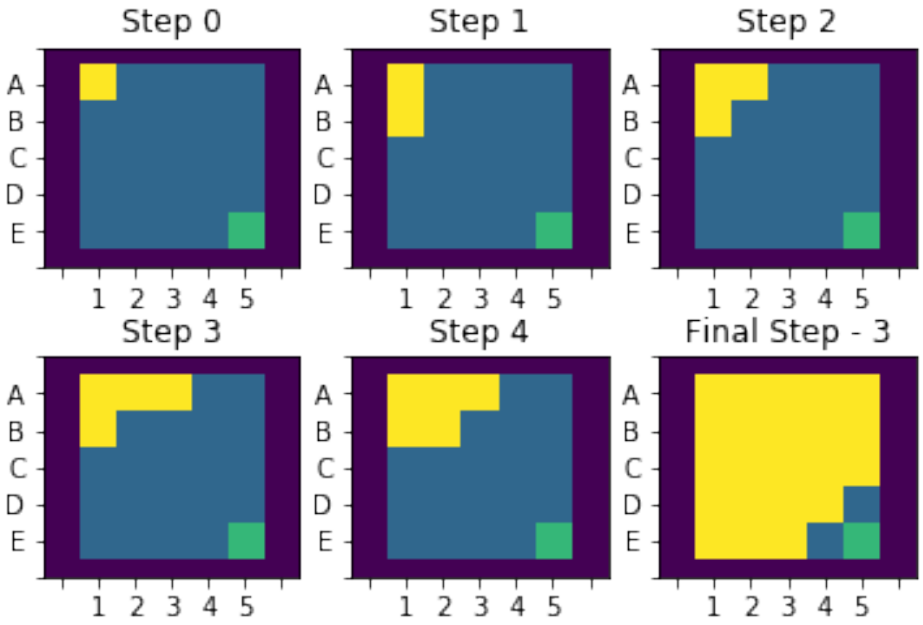

Figure 3.3: Forward A* (Smaller g-value)

Figure 3.3 (above) shows how the Forward A* algorithm uses a smaller g-value to solve tie breaks and computes a path from the agent to the target state. Again, step 0 is the initial maze where the spawn state is the yellow block at A1 and the goal state is the green block at E5.

Step 0: The same situation that occurs in Forward A* occurs again in that the only two states that can be expanded are B1 and A2, where both have a value of 8. Since the g-values are equivalent, the Forward A* algorithm selects a random state and chooses B1, as seen by the extension of the yellow block, which now represents states expanded. It then adds C1 and B2 to the open list.

Step 1: At B2, the algorithm can now choose from C1, B2, and A2 from the open list to expand. Every cell in the grid, other than the spawn state and target state, actually has the same f-value of 8 and we will conduct our future analysis with this understanding in mind. Since all three states have the same f-value, Forward A* now selects the smallest g-value. In this case, Forward A* chooses A2 since it has the smallest g-value of the three, and removes B2 while adding A3 to the open list.

Step 2: At A2, the algorithm now can choose from C1, B2, and A3 from the open list to expand and all actually have the same g-values. A3 is randomly selected between the three, and removes A2 while adding A4 and B3 to the open list.

Step 3: At A3, the algorithm can now choose between A4, C1, B2, and B3 from the open list. A4 and B3 have g-values of 3, while C1 and B2 have smaller g-values of 2. Thus, B2 is randomly selected between C1 and B2, removed from the openlist, and C2 is added to the open list.

Step 4: At B2, the algorithm can now choose between A4, B3, C1, and C2 from the open list. A4, B3, and C2 all have g-values of 3 while C1 has a smaller g-value of 2. Thus, C1 is selected and we can now start to see a pattern occurring. Since Forward A* is always selecting the smallest g-value on the open list, the algorithm will expand unnecessary states that are farther away from the target state.

Final Step - 3: Fast forward a few steps to when our algorithm almost reaches the target state, and we see the downside of breaking tie-breakers with selecting a smaller g-value. Since the algorithm always breaks states with the same f-values by choosing a state closer to the spawn state, we can intuitively realize that the algorithm will choose to expand "horizontally" with respect to the target state instead of propagating towards the target state in a directed manner. This is why the average runtime for Forward A* that solves tie breakers by selecting smaller g-values is almost 28 times slower than if it selected for larger g-values.

4 Forward vs Backward A*

Implementation details: Note that both our Forward and Backward A* implementations break ties between states with the same f-value by selecting the greater g-value. We run both algorithms on all 50 mazes and collect data on the runtimes. The runtime for an algorithm is equivalent to the total expanded states of that algorithm. Again, we choose to measure total expanded states rather than the total number of uniquely expanded states since we feel that this metric would be more telling of runtime. If our A* algorithm searches over the same expanded states each iteration, we would like our runtime measurement to reflect that. Keep in mind that our spawn and target states are always at [5,5] and [95,95] respectively for a more consistent comparison. Table 4.1 (below) shows the runtime for both algorithms, and how much slower Backward A* is compared to Forward A* for the first ten mazes.

Table 4.1: Forward A* vs Backward A* Runtime Comparison

	forwardRuntime	backwardRuntime	timesSlower
0	9270	110189	10.886624
1	8547	103198	11.074178
2	8924	108192	11.123711
3	5229	84779	15.213234
4	9669	142384	13.725825
5	8153	88336	9.834785
6	6398	125408	18.601125
7	7421	115264	14.532139
8	7467	87396	10.704299
9	6180	58531	8.471036

Figure 4.1 (below) is the visualization of column 3 (`timesSlower`) of Table 4.1, and shows how many times slower Backward A* is with respect to Forward A*. On average, Backward A* is about 12 times slower than Forward A*, as represented by the horizontal red line. Since the graph is always positive for all fifty runs, Backward A* is always slower compared to Forward A* without exception for our test cases. Taking a closer look, we see that there are many spikes in Backward A*'s runtime throughout the graph. The reason for these runtime differences seen at these peaks between the two algorithms can be discerned by observing where the algorithm searches begin.

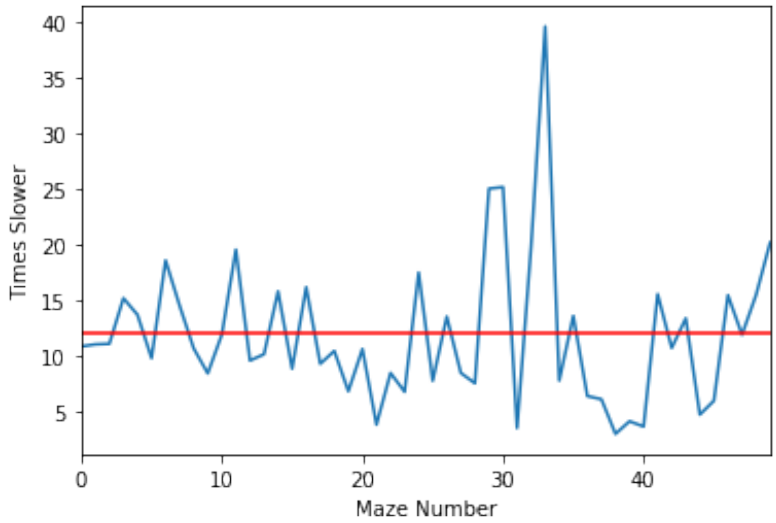

Figure 4.1: Backward A* vs Forward A* Runtime Comparison

One main reason why Backward A* is slower than Forward A* is the result of the opposite starting points of the two search algorithms. Forward A*, as the name suggests, begins at the state where the agent resides and searches in the direction of the target state. The greatest amount of information regarding blocked states comes from the states directly adjacent to the agent (not including the states the agent already passed through). Thus, Forward A* can avoid calculating paths to the target state that are blocked off since the start of each algorithm always considers the agent's immediate surroundings. On the other hand, Backward A* begins at the target state and searches in the direction of the agent. The area around the target state is completely unseen since the agent is assumedly not nearby and cannot see whether any of the neighboring states around the target state are blocked. Thus, Backward A* cannot account for any blocked states until it either trespasses already seen states or gets directly next to the agent. And at that point, it is already too late to bring out a search algorithm's full potential. Take a look at the following example, Figure 4.2 (below), that exemplifies our explanation of why Backward A* is much slower than Forward A*.

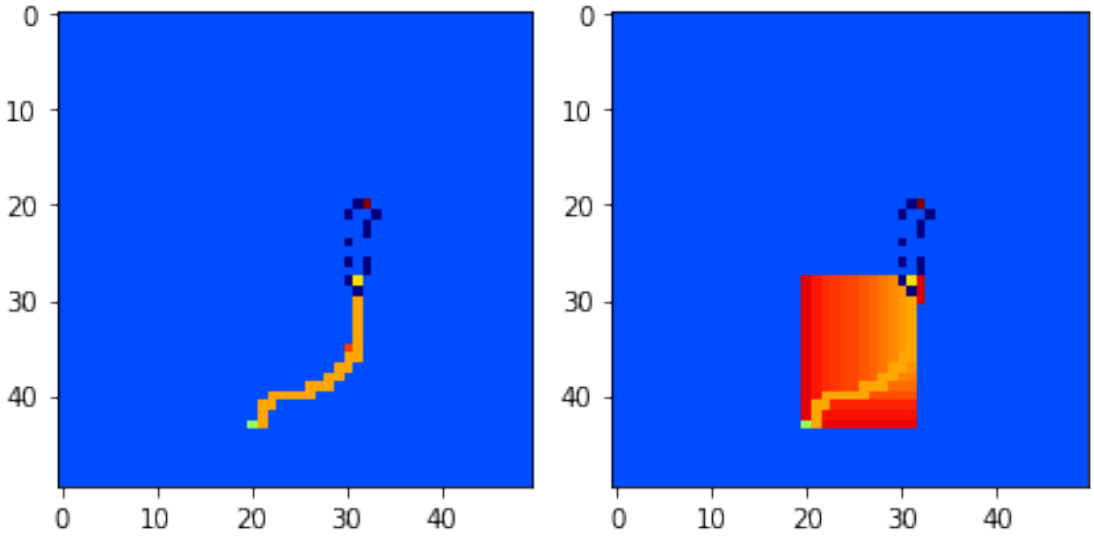

Figure 4.2: Backward A* Bottleneck Iteration

Figure 4.2 (above) shows what our agent sees at a bottleneck iteration (#3) of the Backward A* algorithm that ends up expanding way too many states before finding a successful path. First consider the left graph: the target state [43, 20] is green and our current agent location [28, 31] is yellow (original spawn location [20, 32] is dark brown). The light orange path is the current Backward A* search that begins at the target state and propagates towards the agent location. However, right before it reaches the agent location [28, 31], it meets a blocked state at [29,31]. This blocking phenomenon would never occur for Forward A* towards the end of a search, because all the blocked states are usually at the origin of the search, where the agent is located. What ends up happening in this case is that the surrounding states around [29,32] all have a higher f-value than some previous f-value that is "farther away" than the end of the search. In this case, our Backward A* ends up doubling back and selecting the state at [35, 30], as shown in the darker orange box in the left graph. Fast forward the current search until completion, shown in the right graph, and we can see how Backward A* inefficiently expands all these other states with lower f-values before finally finding that detour to the agent location [28, 31].

5 Heuristics in the Adaptive A* (Part 1)

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

Our project uses Manhattan distances as heuristics or h-values. Manhattan distance is the sum of the horizontal and vertical distances between points (cells in our case) in a grid. Consider (x, y) and (x_1,y_1) cells in a grid. Then Manhattan distance m is

$$m = |x - x_1| + |y - y_1| \tag{5.1}$$

To prove any heuristic h is consistent, we have to prove the following property:

$$h(n) \leq c(n, a, n_1) + h(n_1) \tag{5.2}$$

$h(n)$ is the h-value of n.
$c(n, a, n_1)$ is the cost function to reach n_1 from n.
$h(n_1)$ is the h-value of n_1.

In our Manhattan distances case, we prove this property and heuristic consistency through proof by mathematical induction. In the grid world, the action cost to move between any two neighbouring cells is the same, which is 1. If (n, n_1), (n_1, n_2) and (n_2, n_3) are pairs of neighbouring cells, then

$$c(n, a, n_1) = c(n_1, a, n_2) = c(n_2, a, n_3) = 1 \tag{5.3}$$

Now apply the given property to some cells n_1 and n as its neighbour.

$$h(n) \leq c(n, a, n_1) + h(n_1) \tag{5.4}$$

Since we know that the action cost to move between any two neighbouring cells is 1,

$$h(n) \leq 1 + h(n_1) \tag{5.5}$$

Now add 1 to both sides of the inequality:

$$h(n) + 1 \leq 1 + h(n_1) + 1 \tag{5.6}$$

Consider that there is a next cell n_2 which is a neighbour of n_1.

$$h(n_1) \leq c(n_1, a, n_2) + h(n_2) \tag{5.7}$$

Given that consistency property holds for the cell n_2, by induction, this also holds for all values of n. Thus, Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions.

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.

Adaptive A* search uses previous A* search experience to update the h-values of the cells or states that were expanded in the previous A* search. $g(s)$ is the smallest distance from start state to current state and $g(s_{goal})$ is the distance from the start state to goal state. Here we update the h-values for expanded cells with $g(s_{goal})$ - $g(s)$ as h_{new}-values. With the current knowledge of the gridworld, $g(s_{goal})$-$g(s)$ is the smallest distance from cell s to goal state. In the future, Adaptive A* is going to expand more unblocked cells and the cost will increase. So the new h-values of Adaptive A* are consistent and have the smallest cost from the present state to goal state.

$$h_{new}(s) = g(s_{goal}) - g(s) \tag{5.8}$$

To prove that h_{new}-values are consistent, we assume that h_{new} is consistent. From the definition of heuristic consistency:

$$h_{new}(s) \leq c(s, a, s_{next}) + h_{new}(s_{next}) \tag{5.9}$$

Using the above two equations we get,

$$g(goal) - g(s) \leq c(s, a, s_{next}) + g(goal) - g(s_{next}) \tag{5.10}$$

After solving the above equation we get,

$$g(s_{next}) - g(s) \leq c(s, a, s_{next}) \tag{5.11}$$

From the A* search algorithm we know that $g(s)$ is the smallest distance from the start state to current state s, and $g(s_next)$ is the smallest distance from the start state to state s_next. We know that the smallest distance between s state and s_{next} state is the action cost between s state and s_{next} state. By representing this in equation form we get,

$$g(s_{next}) - g(s) = c(s, a, s_{next}) \tag{5.12}$$

From this, we proved that for every s,a,s_{next} value,

$$g(s_{next}) - g(s) \leq c(s, a, s_{next}) \tag{5.13}$$

This proves that, Adaptive A* leaves initially consistent h-values consistent even if the action costs increase.

6 Heuristics in the Adaptive A* (Part 2)

Implementation Details: Our implementation of Forward and Adaptive A* solves ties between states with the same f-value by selecting the state with the greater g-value. We again run both algorithms on all 50 mazes and collect data on the runtimes. The runtime for an algorithm is equivalent to the total expanded states of that algorithm. Again, we choose to measure total expanded states rather than the total number of uniquely expanded states since we feel this metric would be more telling of runtime. If our A* algorithm searches over the same expanded state each iteration, we would like our runtime measurement to reflect that. Keep in mind that our spawn and target states are always at [5,5] and [95,95] respectively for a more consistent comparison. Table 6.1 below shows the runtime for both algorithms, whether Adaptive A* was faster than Forward A*, and how many times slower Forward A* is compared to Adaptive A* for the first ten mazes.

Table 6.1: Forward A* vs Adaptive A* Runtime Comparison

	forwardRuntime	adaptiveRuntime	adaptive_faster	timesSlower
0	9270	8166	True	0.135195
1	8547	13929	False	-0.386388
2	8924	9466	False	-0.057258
3	5229	8375	False	-0.375642
4	9669	9616	True	0.005512
5	8153	5716	True	0.426347
6	6398	5781	True	0.106729
7	7421	7695	False	-0.035608
8	7467	7300	True	0.022877
9	6180	5446	True	0.134778

Figure 6.1 (below) is the visualization of column 4 (`timesSlower`) of Table 6.1, and shows how many times slower Forward A* is with respect to Adaptive A*. On average, the runtime for Adaptive A* is about 1.33% faster with respect to Forward A* as represented by the horizontal red line. This is an incredibly small percent improvement for our limited sample size, such that we can claim that Adaptive A*'s runtime improvement is negligible in comparison to Forward A* in our test mazes. We see that the graph is not always positive for all fifty runs, indicating that in many cases Adaptive A* is actually slower than Forward A*. In fact, for our 50 test mazes, Adaptive A* is only faster 42% of the time. Such a negligible average runtime difference, represented by the red line, almost contradicts our expectation that Adaptive A* is faster than Forward A*.

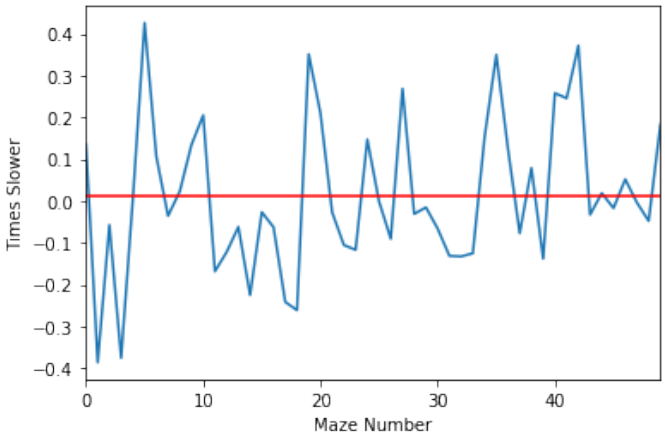

Figure 6.1: Adaptive A* vs Forward A* Runtime Comparison

The main reason for such a lack of difference between the algorithms is due to the construction of our maze world. We generate our maze world where each block has a 30% of being blocked. As you can see from Figure 6.2 (below), this style of maze creation leads to a very random design that isn't very maze-like with long corridors. Instead, it's more like a sea of random blocks where this randomness of blocked states is the crux to why the Adaptive A* algorithm fails to perform better than Forward A*.

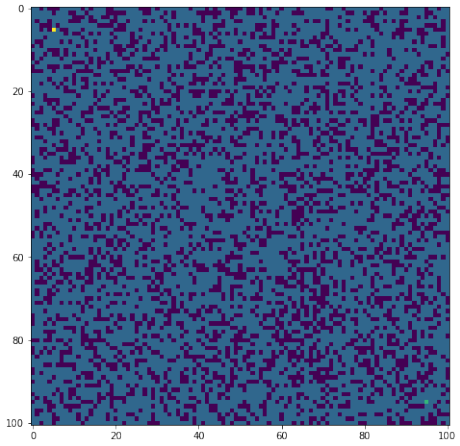

Figure 6.2: Maze 1

A typical maze, depending on the algorithm that builds it usually has some winding paths or corridors. To put it simply, there are stretches of cells that are in a straight line going vertically or horizontally. Consider the following situation where our agent computes a path down one of these corridors and moves down this path until hitting a blocked cell. Adaptive A* shines in that the path the agent just took is less likely to be chosen again in the next few iterations since all the h-values are updated to account for the cost of moving through these states. As a result, our next search is more likely to not expand the same states and waste time traversing the same path the agent just took, opting for a new path in the direction of the target state. However, in our maze world where these straight line corridors are less likely to appear, Adaptive A*'s effectiveness is greatly reduced. The agent cannot travel many steps until a random block appears and stops its progress, so the algorithm ends up re-exploring many of the same states either way. This causes Adaptive A* to have a similar runtime in comparison to Forward A*.

7 Statistical Significance

Performance differences between two search algorithms can be systematic in nature or only due to sampling noise (= bias exhibited by the selected test cases since the number of test cases is always limited). One can use statistical hypothesis tests to determine whether they are systematic in nature. Describe for one of the experimental questions above exactly how a statistical hypothesis test could be performed.

One can use statistical hypothesis tests to determine performance between two search algorithms to be systematic in nature or only due to sampling noise. Hypothesis testing is a type of statistical inference that can be used in many ways to evaluate whether one's sample results is representative on a population level. There are different types of hypothesis tests like z-test, t-test etc. One chooses the hypothesis testing depending on the sample size, known or unknown population parameters and various other factors.

Let us validate whether Repeated Forward A* and Repeated Backward A* have a difference in mean runtime. In our case, the two sample sizes are 50 since we collected sets of runtime results from Repeated Forward A* and Repeated Backward A* search execution on the same set of 50 gridworlds. We decide to use a paired t-test on an $alpha$ = 5% level to compare the Repeated Forward A* and Repeated Backward A* search algorithms because both algorithms are running on the same set of gridworlds.

Consider N as the sample size of both Repeated Forward A* and Repeated Backward A* search algorithms. Let the difference between runtimes of Repeated Forward A* and Repeated Backward A* be δ.

$$\delta = F - B \tag{7.1}$$

F is the runtime of Repeated Forward A* on a single gridworld.
B is the runtime of Repeated Backward A* on a single gridworld.

After finding all the δ values of 50 gridworlds, we calculate the mean value and standard deviation of the δ values: x_δ is the mean value of all differences.
S_δ is the standard deviation of all differences.

Now we formulate the null and alternative hypothesis. The null hypothesis assumes that the mean difference in runtimes between the two search strategies is zero, and the alternative hypothesis suggests that the difference in means is not equal to zero.
H_0:$\mu_\delta = 0$.
H_1:$\mu_\delta \neq 0$.

Now we can validate this null hypothesis with by calculating the test statistic for our hypothesis test, represented by t_δ.

$$t_\delta = \frac{x_\delta - \mu_\delta}{\frac{S_\delta}{\sqrt{N_\delta}}} \tag{7.2}$$

N_δ is the number of pairs i.e number of gridworlds.

After calculating the test statistic, we calculate the p-value, which is the chance that our results are insignificant by chance. The p-value can be obtained through our test statistic with $N_\delta - 1$ degrees of freedom. The lower the p-value, the lower the probability that our results were caused by chance and we can reject the null hypothesis. Since our hypothesis test is evaluated on a 5% level, our p-value needs to be less than 5% for us to conclude that there is enough data to reject the null hypothesis and say the alternative hypothesis is true. If our alternative hypothesis is true, we can say that the difference in mean runtimes between our algorithms is not zero whereas we know that Backward A* is slower than Forward A*. Note that the possibility that the null hypothesis is true can never be ruled out.