CS:440 Intro to Artificial Intelligence

Assignment 1
Fast Trajectory Replanning

Project Report

Group Members:
Ruiqi Wang: (rw555@scartletmail.rutgers.edu) RUID: 192001525
Vivek Rajyaguru: (vpr11@scarletmail.rutgers.edu) RUID: 195003544
Divay Tomar: (dt578@scarletmail.rutgers.edu) RUID: 195004196

Part 0. (RW)

In order to check whether a valid path exists, we simply used DFS search, where the agent will continuously search for the agent and only look back when the current path is not working. The reason why we choose DFS is because we thought that it might be better because BFS tends to search every node surrounding every iteration and, for example, if the agent is at the top left corner while the target is at the bottom, it will take BFS a longer time to reach the target than DFS. But when we think about the other scenario, where the target is closer to the agent, DFS might miss it and will keep going to the end of the grid while BFS will likely find out that there is a valid path, therefore end before DFS. We do not think one is better than the other, therefore the implementation we have is definitely not the most efficient. The most efficient will definitely be A* as DFS and BFS disregards the direction from the agent and target, therefore wasting a lot of time. Since A* uses heuristic values, it will ignore most of the nodes and only head towards the general direction of the target.

Part 1.

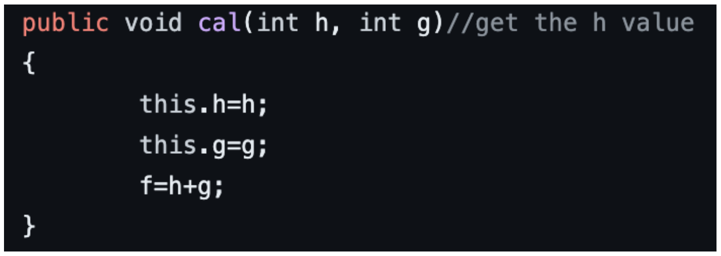
a) (RW)

The reason why the agent's first move is to the east rather than north is because, looking at the graph, the agent doesn't know whether if the road to the east is blocked or not, so it would choose the closest direction to reach T, which in this case is east by using the equation that $f(n) = h(n) + g(n)$. Although the cost of moving in both directions is 1, the heuristic value for north is clearly higher than east, resulting in greater $f(n)$, thus making the agent move to the east, rather than to the north.

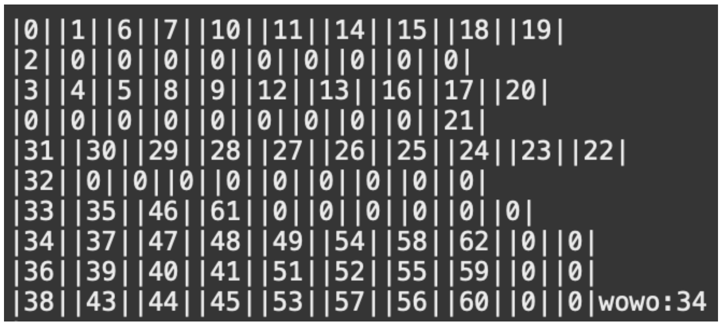
b) (VR)

In a finite world, where the amount of cells in a grid is predetermined, the agent is able to move in a direction towards the target. Given that said direction is blocked off, the agent must find a path in a different direction to bypass the blocked cells, and still reach the target. In the first case, we know that the agent will eventually find the target, given that a path exists. For example, let's assume that the gridworld is a 9x9 square, though it can be set to any limit. Given that either row 5 or column 5 are filled with only blocked cells, and the agent and target are on opposite sides of said row/column. Now, the agent knows which direction the target is in, so based on the heuristic value, it will move towards the agent, until it realizes that the next cell is blocked. To avoid this blocked cell, the agent will test other cardinal directions to find a path, from one edge of the grid to the other. Once it realizes that there is no path through to the target, the agent will note that in a finite time, it is impossible to reach the target. In any other case, the agent will continue to search through nodes, in order based on heuristic values, and eventually will find a path to the target in finite time, since the grid is of finite size. Using Figure 8, we can see that we are working in a finite grid of 5x5, or 25 total cells, 6 of which are blocked, and the other 19 unblocked. The upper bound of moves until the target is reached or deemed impossible to reach would be 19^2, or 361. In any grid world, at max, the moves that the agent can make is 4. In this 5x5 grid, there are 16 cells that only have 3 possible moves (outer edge cells) and 9 cells that can possibly have 4 moves. To move to any of the 19 unblocked cells, there would be at max 19 moves for the agent. We can prove this using proof by contradiction, where we assume that the number of moves required is greater than, or not bounded by above, by the number of unblocked cells squared. In this case, we can say that number of moves > (number of unblocked cells)^2. Filling in the numbers, this in turn gives us $19 > (19)^2 \rightarrow 19 > 361$. We know that this inequality is not true, and therefore, we can see that by contradiction, that the number of moves of the agent until it reaches the target or discovers that it is impossible is bounded from above by the number of unblocked cells squared.

Part 2. (RW, VR)

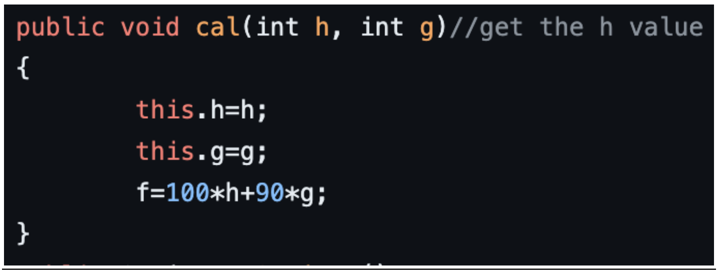


```
public void cal(int h, int g)//get the h value
{
        this.h=h;
        this.g=g;
        f=h+g;
}
```

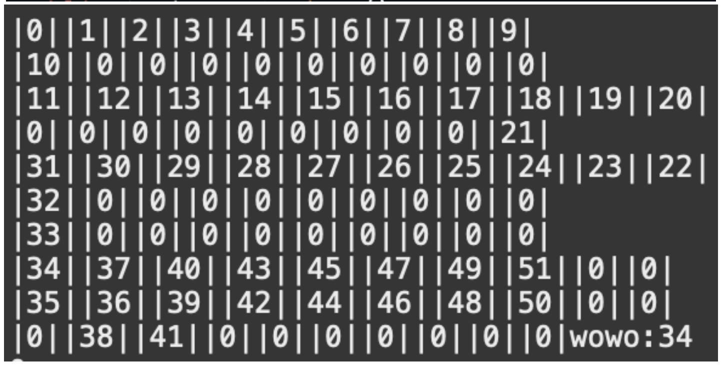


```
|0||1||6||7||10||11||14||15||18||19|
|2||0||0||0||0||0||0||0||0||0|
|3||4||5||8||9||12||13||16||17||20|
|0||0||0||0||0||0||0||0||0||21|
|31||30||29||28||27||26||25||24||23||22|
|32||0||0||0||0||0||0||0||0||0|
|33||35||46||61||0||0||0||0||0||0|
|34||37||47||48||49||54||58||62||0||0|
|36||39||40||41||51||52||55||59||0||0|
|38||43||44||45||53||57||56||60||0||0|wowo:34
```

In the first case, we saw the Repeated Forward A* algorithm break ties with cells in favor of smaller g-values. We limited the grid to a 10 by 10 to help see what exactly was going on. When calculating, we gave the same weight to both h-values and g-values, and to minimize the overall value, the program will prefer those paths associated with smaller g-values.which means that the algorithm will much prefer to stay on its path rather than going back since g-value weighs less. As you can see from the output, the algorithm will prefer to go right because is holds a lower g-value, but sees that the south direction has a g-value of 2, so it will traverse along both paths, until it realizes that at g-value "19", there is no path left to the target, so it can only go along the other path.



```
public void cal(int h, int g)//get the h value
{
        this.h=h;
        this.g=g;
        f=100*h+90*g;
}
```



```
|0||1||2||3||4||5||6||7||8||9|
|10||0||0||0||0||0||0||0||0||0|
|11||12||13||14||15||16||17||18||19||20|
|0||0||0||0||0||0||0||0||0||21|
|31||30||29||28||27||26||25||24||23||22|
|32||0||0||0||0||0||0||0||0||0|
|33||0||0||0||0||0||0||0||0||0|
|34||37||40||43||45||47||49||51||0||0|
|35||36||39||42||44||46||48||50||0||0|
|0||38||41||0||0||0||0||0||0||0|wowo:34
```

In this second case, we saw the Repeated Forward A* program break ties with cells in favor of larger g-values. We limited the grid to a 10 by 10 to help see what exactly was going on. Since in the equation $f = 100 * h + 90 * g$, the value of g was given less-weight than h. Because of this, the program adapts to preferring a larger g-value, which changes the overall output. As you can see in the output, the algorithm will see the g-value of steps in both right and down, and will note that since it is looking for a higher g-value, to explore the south path, which eventually leads to the target.

In our observations, we saw that when the grid was the exact same, the first case, where the lower g-value was preferred, the A* traversed 2 different paths simultaneously, where one path led to a dead end, and the other let to the target. This is because the g-value of the path leading to the dead end was lower after the first step than the path leading to the target. It continuously jumped between both paths, until it realized that the first path is a dead end, and then continued to only traverse through the second path. Comparing this to case 2, the algorithm will prefer the higher g-value, and go directly on the path south, and continue to the target from there. Simply changing the weight that both "h" and "g" hold on the value of f will completely change the way that the algorithm perceives the grid, and will take completely different instructions to find its way to the target.

Part 3. (DT, VR)

After implementing and comparing the Repeated Forward A* and Repeated Backward A* are variants of the A* search algorithm with respect to their runtime or, equivalently, number of expanded cells, we found in most cases the Backward A* algorithms had a faster average run time and expanded cells. Both of the algorithms expand cells based on their f-values, which are the sum of heuristic estimates and costs so far. The algorithm further terminates when reaching the goal state or when no more cells are available to search. For the break ties we found that both versions of Repeated A* break ties among cells with the same f-value in favor of cells with larger g-values, which therefore favors paths with higher cumulative costs and ensures that the algorithm explores more good paths earlier.
Overall, we found that Repeated Backward A* algorithm was quicker and had less expanded nodes compared to Repeated Forward A*. In certain cases, the run time for Repeated Backward A* was almost 2 to 3 times faster than Forward A*. We think this is true based on the amount of accessible paths from either the initial state and goal state. For example, we took 5 different runtimes of 50 grids for each algorithm, and recorded the runtimes in nanoseconds:

Repeated Forward A*	**Repeated Backward A***
101423700 ns	94864900 ns
101954500 ns	104274200 ns
98180200 ns	96582900 ns
92033000 ns	94669800 ns
103845700 ns	88321000 ns
Average: **99487420** nanoseconds	Average: **95742560** nanoseconds

We can see that there is a slight difference in the runtimes, and we think that it is mainly due to the amount of paths available from either state. In a general sense, after running the algorithms across multiple repetitions, we saw that Repeated Forward A* was also quicker than Repeated Repeated Backward A* in given scenarios, which we think is due to the randomness of the grid that it is traversing.

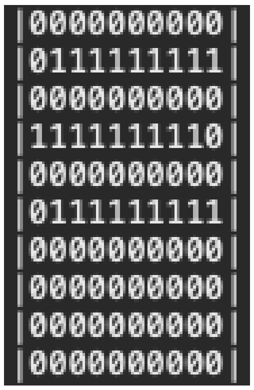
Part 4.(R.W)

The reason why Manhattan distances are consistent in the gridworld in which agents can move only in the four main compass directions is because the most optimized route presumes that there's no blockage, which will be the total x difference and y difference between agent and target. For example if the Target is 10 block east to the agent the agent will simply move 10 block east because it's a straight line which is proved by the Manhattan formula distance= |y1-y2|+|x1-x2| since it calculates both the x and y difference between target and the agent. And in the case where the target is let's say 10 blocks east and 5 blocks south no matter what combination of the steps the agent took it will take the agent 15 moves to get there, which is just the x and y difference between the agent and target.

The reason why the Hs(new) is consistent and admissible is because while the Hs(new) is going to be higher than the original H value, but since A* algorithm looks for the most efficient route, the g(goal state) - g(current state) will be always be the most accurate value during the end of the search. So, Hs(new) will not be overestimated. Furthermore, we can prove it by the given formula of A*. Since g(goalstate) - g(s)<= gd(s) and g(goalstate - g(s)) = h(snew), we can see that h(snew) <= gd(s). As the values of h(snew) are being updated, they will always satisfy the equation of h(snew) <= gd(s), therefore giving it a consistent and admissible value.

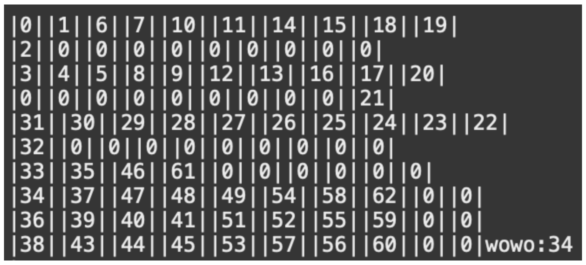
Part 5. (RW, VR, DT)

When looking at Repeated Forward A* and Adaptive A*, we noticed some general differences. In Adaptive A*, the g-values change with every move that is made, and it will easily find the target state compared to the Repeated Forward A* algorithm.



```
|0000000000|
|0111111111|
|0000000000|
|1111111110|
|0000000000|
|0111111111|
|0000000000|
|0000000000|
|0000000000|
|0000000000|
```

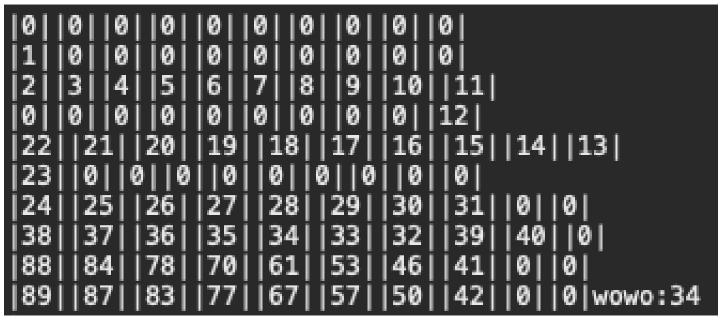
Here we have a 10 by 10 grid, which we scaled down to see exactly, where the 0's mark the unblocked cells, and the 1's mark the blocked cells. Below, you will see the outputs in values that each algorithm took, and how they compare to each other.



```
|0||1||6||7||10||11||14||15||18||19|
|2||0||0||0||0||0||0||0||0||0|
|3||4||5||8||9||12||13||16||17||20|
|0||0||0||0||0||0||0||0||0||21|
|31||30||29||28||27||26||25||24||23||22|
|32||0||0||0||0||0||0||0||0||0|
|33||35||46||61||0||0||0||0||0||0|
|34||37||47||48||49||54||58||62||0||0|
|36||39||40||41||51||52||55||59||0||0|
|38||43||44||45||53||57||56||60||0||0|wowo:34
```

(The number indicated how the algorithm traversed)

Here, we are looking at the output for Repeated Forward A* where the algorithm checks both paths, having to look at the trail going east, and then also checking the trail that is going south. We limited the grid to a 10 by 10 to help see what exactly was going on.



```
|0||0||0||0||0||0||0||0||0||0|
|1||0||0||0||0||0||0||0||0||0|
|2||3||4||5||6||7||8||9||10||11|
|0||0||0||0||0||0||0||0||0||12|
|22||21||20||19||18||17||16||15||14||13|
|23||0||0||0||0||0||0||0||0||0|
|24||25||26||27||28||29||30||31||0||0|
|38||37||36||35||34||33||32||39||40||0|
|88||84||78||70||61||53||46||41||0||0|
|89||87||83||77||67||57||50||42||0||0|wowo:34
```

Here, we are looking at the output for Adaptive A*, we ran this algo twice, the first time starting at the top right of the map which makes the h value much higher on the second search and the second time we start agent on the top left of the map. The algorithm through repeated searches knows that the h value is much higher when traveling east first, so it knows not to check in that direction on the second run. Rather, it just goes to the path in the south, which saves runtime and is proven to be more efficient. We limited the grid to a 10 by 10 to help see what exactly was going on.

Here, we compared 5 different runtimes of Repeated Forward A* to Adaptive A* on the same maps with different starting state but end state, which were recorded in nanoseconds:

Repeated Forward A*	**Adaptive A***
1237800 ns	824900 ns
1205600 ns	1016100 ns
1230600 ns	1052000 ns

1277400 ns	865800 ns
1460900 ns	901300 ns
Average: **1282460** nanoseconds	Average: **932020** nanoseconds

One thing that we did notice between the 2 algorithms was not only was it more efficient, but it was more effective. The Adaptive A* constantly updated the h-values of the cells as it moves, and ultimately determined the cost-effective, optimal path to the target. On the other hand, the Repeated Forward A* had to check every path that it would come along, in order to see the path to the target. Although it may not have to check every cell, it was still relatively slower than Adaptive A*. Since both algorithms see forward, meaning they only know where the agent and target are, and move to cells from the start, they follow the same means. But where Adaptive A* tops Forward A* is that as the search progresses, with each move, the heuristic values are updated. In a sense, the algorithm is “learning” as it goes along, figuring out the most effective next step with each move.

Part 6. (VR)

A statistical hypothesis test could be used to test the runtimes between two different algorithms, and compare them to see which would be most efficient. To set up this experiment, we are going to compare Repeated Forward A* against Repeated Backward A*. We would start by making a claim, or a hypothesis. For this purpose, let’s claim that Repeated Forward A* is a more efficient program than Repeated Backward A* because the program finds a solution based on the data it was given. Our independent variables for this experiment would be programs, the number of tasks given, and the initial instruction given to both programs. The dependent variable would be the time to complete these tasks. We would run this test 50 times, and gather a sample of runtimes for each program. Each time, we would record the runtime of each program, and add to a list. Then we’d calculate the mean of each program, as well as a delta value of the difference between them. Using those values, we can find a distribution of delta using the assumption that Repeated Forward A* = Repeated Backward A*. Using that distribution, we can find the probability that the previous statement is true or not. If probability is low, we can reject the null hypothesis and claim that Repeated Forward A* ≠ Repeated Backward A*. If it is higher, we are uncertain if the statement is true or not.

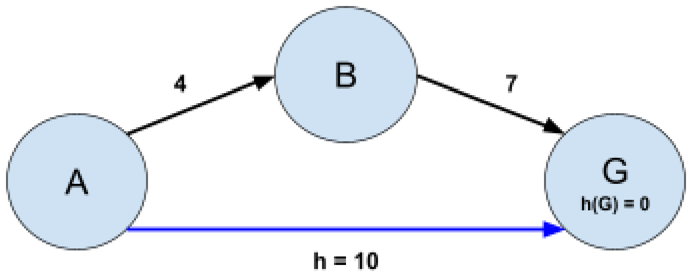
Extra Credit: (DT)

Prove that if a heuristic is consistent, it is also admissible:

- A heuristic is **consistent** if the estimated cost from a given state to the goal, plus the estimated cost from the successor state to the goal, is always greater than or equal to the estimated cost from the given state to the successor state.
- A heuristic is **admissible** if it never overestimates the cost to reach the goal from any given state.

To prove if a heuristic is consistent and also is admissible:

For heuristics to be Consistent: h(G) = 0 and h(n) <= cost(n, B) + h(B)
For heuristics to be admissible: we should have h(n) <=h*(n)

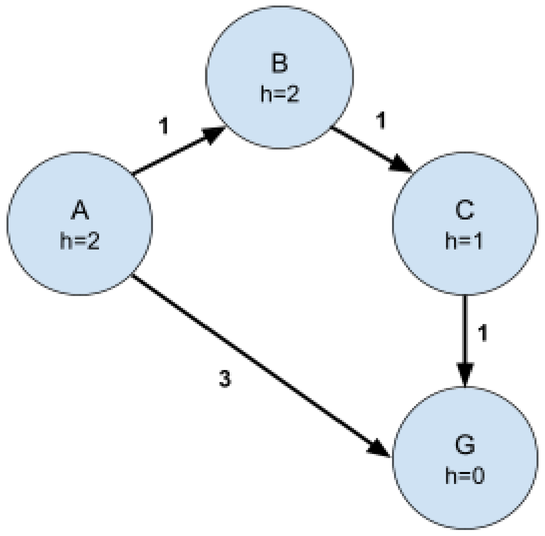


Heuristics Consistent:
h(A) <= 4 + h(B)
h(B) <= 7 + h(G) = 7

Heuristics Admissible: h(n) <=h*(n) for every node(n) where h* is the real cost to the goal, which would be:
h(A) <= 10
h(B) <= 7
h(G) <= 0

Find an example of a heuristic that is admissible but not consistent.

- A heuristic is **not consistent** if the estimated cost from a given state to the goal exceeds the cost of reaching a neighboring state plus the estimated cost from that neighboring state to the goal state.
- For a heuristic to be **admissible** it must never overestimate the cost of reaching the goal from any given state.



A heuristic is a function that estimates the cost of reaching a goal state from a given state. A heuristic is admissible if it never overestimates the cost of reaching the goal. A heuristic is consistent if it satisfies the triangle inequality, which is when the heuristic value of any state is less than or equal to the cost of moving to a neighboring state plus the heuristic value of that neighbor 1.

So, for example if we look at the graph above with four nodes A, B, C and G, where G is the goal state. The edge costs are as follows: c(A,B) = 1, c(B,C) = 1, c(C,G) = 1, c(A,G) = 3. The heuristic function h is defined as follows: h(A) = 2, h(B) = 2, h(C) = 1, h(G) = 0.

This heuristic is admissible because it never overestimates the cost of reaching G. For example, h(A) = 2, which is less than or equal to the actual cost of reaching G from A, which is 3. However, this heuristic is not consistent because it violates the triangle inequality at node B. The heuristic value of B is 2, which is greater than the cost of moving to C (1) plus the heuristic value of C (1). That is, h(B) > c(B,C) + h(C).

A consistent heuristic is also admissible, but not vice versa. Therefore, finding a consistent heuristic is preferable to finding an admissible one because it guarantees that the search algorithm will not revisit any node more than once.