CS440 Assignment 1
Fast Trajectory Replanning

Da Huo

February 21, 2018

Part 0 - Setup your Environments

Architecture

To construct maze/corridor-like gridworlds of size 101×101, we use the object Node, which contains attribute bolcked(boolean), coordinate(tuple of (x,y)), and neighbors(python dictionary which use 'up', 'down', 'left', 'right' as keys and Node object as values) to represent the bottom left corner of each cell.

Method

The function generateMap() in buildMap.py will generate an empty gridworld and setup all neighbors pointers. For cells on the boundries, the neighbors outside the gridworld will be set to None.

The function DFSMaze() will use DFS algorithm to traverse all cells in the given gridworld, randomly assign a cell to blocked with 20% probability

Visualize Maze

We use the python library matplotlib to generate a figure of the given girdworld. We draw 101 horizontal lines and 101 verticle lines to generate the empty gridworld. Then use the list of blocked cells returned by DFSMaze() to fill the corresponding cell to black.

Part 1 - Understanding the methods

a) Explain in your report why the first move of the agent for the example search problem from Figure 1 is to the east rather than the north given that the agent does not know initially which cells are blocked.

Assume A is the start point and T is the goal point, the agent does not know which cells are blocked initially, and we use Manhattan distance as heuristics. When we perform A* from A to T, we first push A to the open list. Then we expand A and add its three neighbors – cell E3, cell D2, and cell E1 to the open list.

Now the open list contains 3 items:

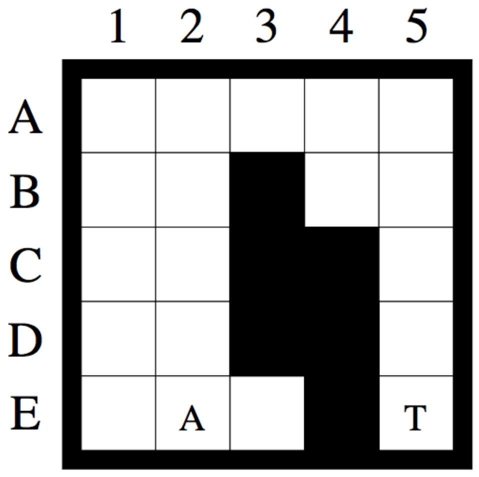

Figure 1: Example Search Problem

- cell E3: cost:1 h-value: 2 f-value: 3
- cell D2: cost:1 h-value: 4 f-value: 5
- cell E1: cost:1 h-value: 4 f-value: 5

Since cell cell E3 has the smallest f-value, so cell E3 will be chosen to expand. We will then push cell E3's neighbors – cell E4 and cell D3 to the open list. We do not push cell E2(A) to the open list because cell E2 is already expanded and is in the close list.

now the open list contains 4 items:

- cell E4: cost:2 h-value: 1 f-value: 3
- cell D3: cost:2 h-value: 3 f-value: 5
- cell D2: cost:1 h-value: 4 f-value: 5
- cell E1: cost:1 h-value: 4 f-value: 5

Since cell cell E4 has the smallest f-value and the agent does not know cell E4 is blocked at this point, so cell E4 will be chosen to expand. We will then push cell E4's neighbors – cell E5(T) and cell D4 to the open list. We do not push cell E3 to the open list because cell E3 is already expanded and is in the close list.

now the open list contains 5 items:

- cell E5: cost:3 h-value: 0 f-value: 3
- cell D4: cost:3 h-value: 2 f-value: 5
- cell D3: cost:2 h-value: 3 f-value: 5
- cell D2: cost:1 h-value: 4 f-value: 5
- cell E1: cost:1 h-value: 4 f-value: 5

Since cell cell E5(T) has the smallest f-value, so cell E5 will be chosen to expand. Before we expand cell E5(T), since E5 is the goal point, so A* can terminate at this point.

We found the path: E2(A) $\rightarrow$ E3 $\rightarrow$ E4 $\rightarrow$ E5(T)

Therefore, the agent will move to east at the beginning

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

In finite gridworlds, we always have a finite number of cells. In addition, each cell are connected to its neighboring cells, that is, we can go from any cell in the gridworld to any other cell in the gridworld by following the neighbor pointers, if we assume there are no blocked cells. Similarly, if we take the blocked cells into account, we can go from any unblocked cell to any other unblocked cells within the same unblocked region. By unblocked region, it means a region of consecutive unblocked cells, namely, region of cells that are surrounded by either the boundries of the gridworld, or the blocked cells, or both.

Since A* algorithm keeps a closeList, so it will never expand cells which are previously expanded. Because that in finite gridworld, we can get to any unblocked cell from an unblocked cell within a unblocked region, So

A* algorithm will either following the neighbor pointers to find a path from start to goal if start and goal are both in the same unblocked region, or it will report this is impossible and no path available after it traverses all the cells within its unblocked region. Since we have finite number of cells, so it will report this is impossible in finite time as well.

The Repeated Forward/Backward Algorithm will execute multiple times of A* algorithm. For each A*, it either find a path from start to goal under current knowledge of the gridworld, or report this is impossible. If it reports this is impossible, it means that indeed there is no such path from start to goal given the blocked cells. If it finds a path from start to goal, then the agent will follow this path to either hits the goal or repeat the process again. Since we have a finite number of cells, so the worst case would be the agent moves through all these cells and discovers the whole map. Since the agent moves a finite number of cells and each time it hits a blocked cell, it will run A* which cost a finite time. So the Repeated Forward/Backward algorithm will either reach the target or report this is impossible in finite time.

To 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, we denote the number of unblocked cells as n.

The process of Repeated Forward/Backward A* can be simplified as the following:

```
Loop:
    computePath()
    moveAgent()
```

Since each A* keeps a closeList, so the path it finds will contains at most n cells. Therefore, the number of moves m, after each A* can be at most n. That is,

$$m \leq n \tag{1}$$

As for the number of interations of the loop, we denote this number as l. For the number of interations of the loop, the worst case would be every time the agent follows the presumed path moves one step, the next step is blocked and needs to run A* from current location again and all cells within current unblocked region has to be traversed at least once, in addition, the worst case also require that there is only one unblocked region. All other cells are blocked, that is we have n cells under the same unblocked region. Since every time the agent moves to a new cell, it will explore the neighboring cells, so we can run A* on each cell at most once. Say we visted cell A once, when the future A* try to find a path, it will take A's blocked neighbors into account so we will never stucked in A again. Therefore, the number of A* searches, which is the number of interations of the loop l, can be at most n. That is,

$$l \leq n \tag{2}$$

Combine (1) and (2), we get

$$lm \leq n^2 \tag{3}$$

Where,
n is number of unblocked cells
l is number of loop interations
m is number of agent moves within one loop interation
lm is total number of agent moves

Therefore, 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.

Part 2 - The Effects of Ties

We implemented Repeated Forward A* algorithm with both tie break in favor of larger g-value and smaller g-value. We performed 50 experiments on these 2 tie break strategy with the same 101*101 gridworld and get the result on figure 2:

We found that Repeated Forward A* with larger g-value tie break strategy is much faster than Repeated Forward A* with smaller g-value tie break strategy in general despite some occasional cases such as fail to found path at the very beginning.

The reason of this is because under the situation of multiple cells have the same f-value, choose cells with smaller g-value will take us back to cells near the start point whereas choose cells with larger g-value will let us explore cells near the goal point. If we choose to expand cells near the start point, we will expland a lot of useless cells.

To illustrate this, we can consider a problem where we want to search a path from A to T within the environment on figure 3:

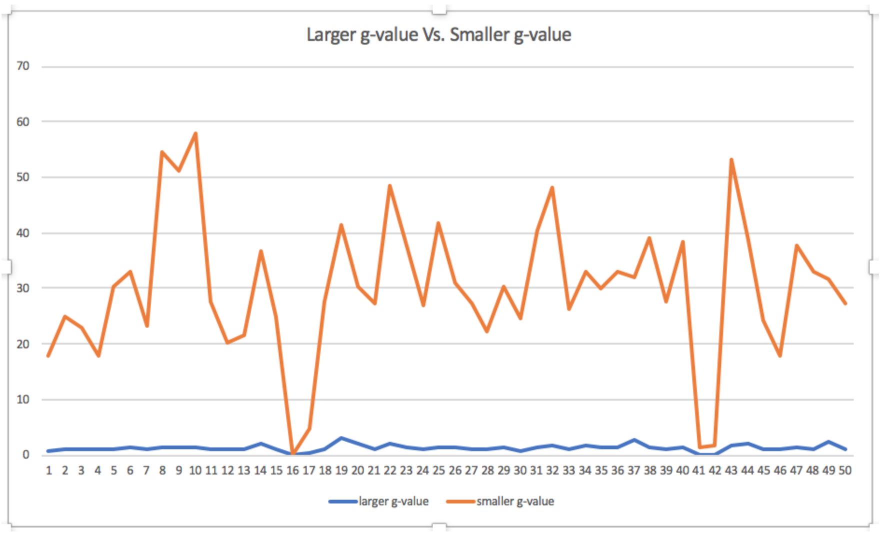

Figure 2: Tie Experiment

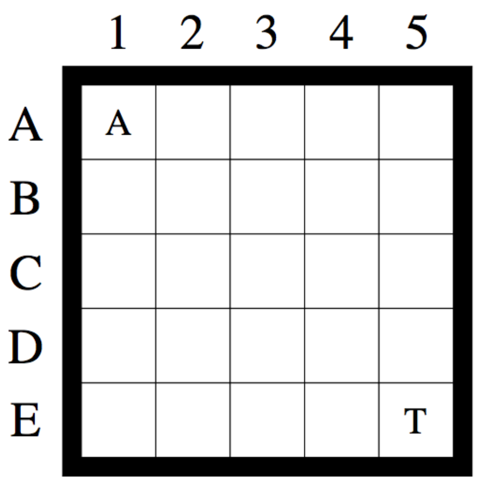

Figure 3: Example Search Problem

Figure 4 shows the first 6 steps of Repeated Forward A* with smaller g-value tie break strategy

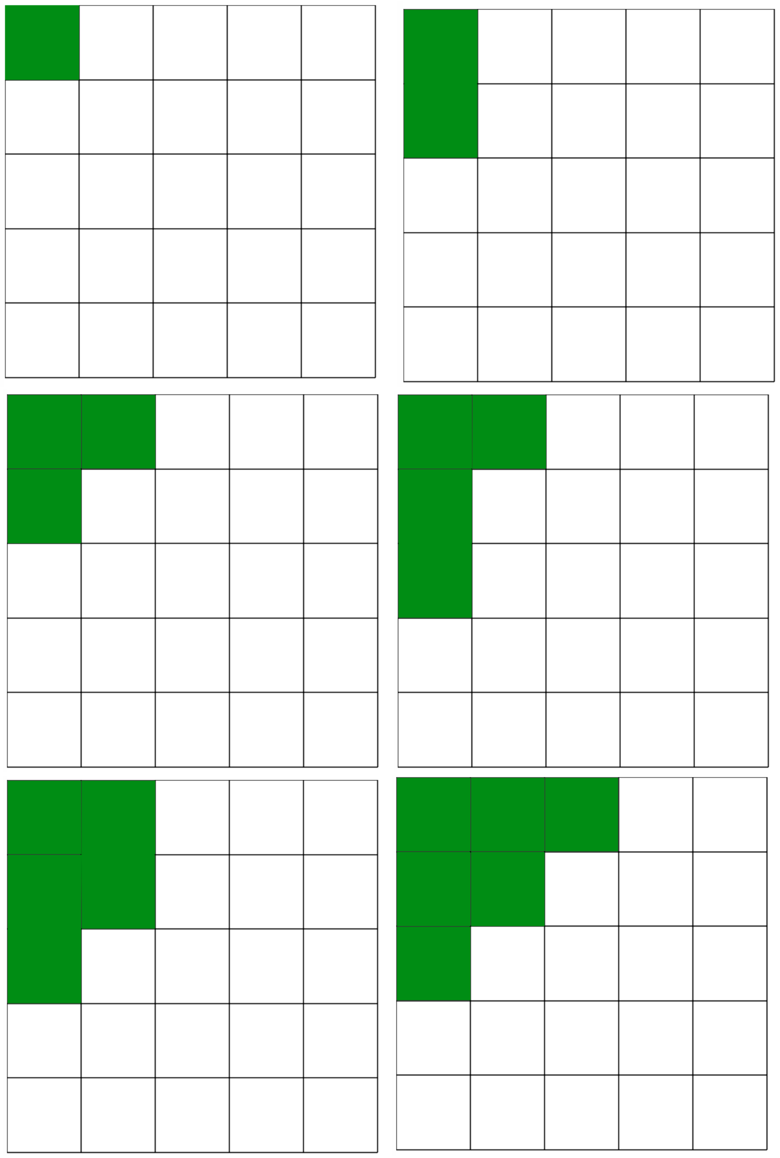
Figure 4: Tie Example

The first cell to be expanded, as the first picture of figure 4 shows, is cell A1, which is the start state. After expanding cell A1, the open list constains

2 cells:

- cell A2: cost:1 h-value: 7 f-value: 8
- cell B1: cost:1 h-value: 7 f-value: 8

In this case, since cell A2 and cell B1 have the same f-value and g-value, so we can randomly choose one to expand, in this case, cell B1. After expanding cell B1, as the second picture of figure 4 shows, the open list now contains:

- cell A2: cost:1 h-value: 7 f-value: 8
- cell B2: cost:2 h-value: 6 f-value: 8
- cell C1: cost:2 h-value: 6 f-value: 8

Now that cell all cells in the open list have the same f-value, if we choose to expand cell with smaller g-value, we will choose cell A2 to expand, as the third picture of figure 4 shows. Afer expanding cell A2, the open list contains:

- cell B2: cost:2 h-value: 6 f-value: 8
- cell C1: cost:2 h-value: 6 f-value: 8
- cell A3: cost:2 h-value: 6 f-value: 8

Now, since all cells in the open list have the same f-value and g-value, so we can randomly choose one to expand, in this case, cell C1. After expanding cell C1, as the fourth picture of figure 4 shows, the open list now contains:

- cell B2: cost:2 h-value: 6 f-value: 8
- cell A3: cost:2 h-value: 6 f-value: 8
- cell C2: cost:3 h-value: 5 f-value: 8
- cell D1: cost:3 h-value: 5 f-value: 8

Now, all cells in the open list have the same f-value, if we choose to expand cell with smaller g-value, we will choose one from cell B2 or cell A3 to expand. In this case, we chose cell B2 as the fifth picture of figure 4 shows. Afer expanding cell A2, the open list contains:

- cell A3: cost:2 h-value: 6 f-value: 8
- cell C2: cost:3 h-value: 5 f-value: 8
- cell D1: cost:3 h-value: 5 f-value: 8
- cell B3: cost:3 h-value: 5 f-value: 8

If we choose to expand cell with smaller g-value again, we will choose cell A3 to expand, as the sixth picture of figure 4 shows.

We can find the pattern from the above steps that under the situaion of all cells have the same f-value, if we break tie in favor of smaller g-values, we will end up with expanding a lot of useless cells near the start point, in this particular cause, we expand all diagonals. Where as if we break tie in favor of larger g-values, we will always expand cells near the goal point, as figure 5 shows.

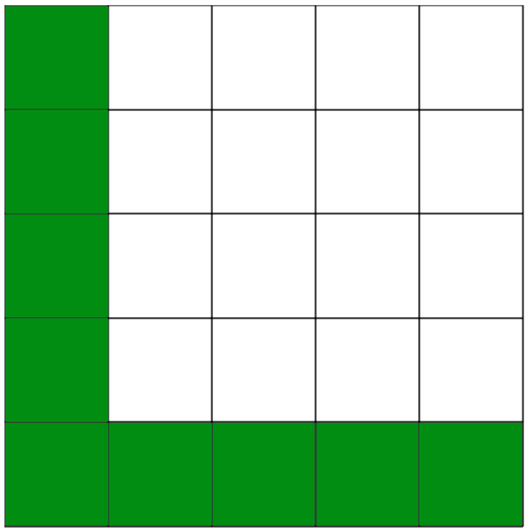

Figure 5: Tie break in favor of larger g-values

Therefore, Repeated Forward A* with larger g-value tie break strategy is much faster than Repeated Forward A* with smaller g-value tie break strategy in general since Repeated Forward A* with smaller g-value tie break strategy will waste a lot of time on expanding useless cells near the start point

Part 3 - Forward vs. Backward

We performed 50 experiments on Repeated Forward A* and Repeated Backward A* with the same 101*101 gridworld and get the following result on figure 6:

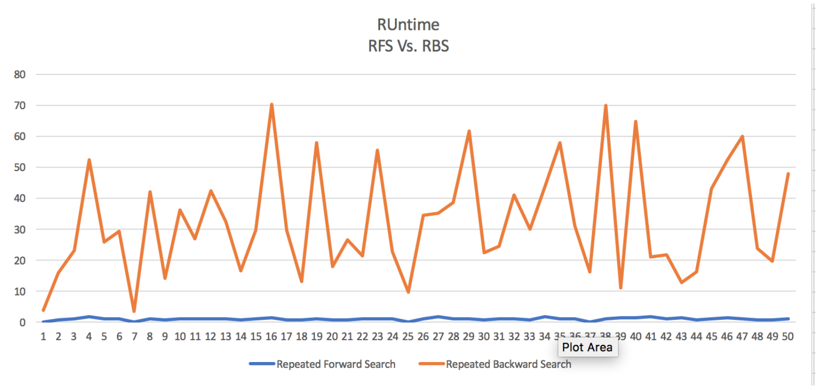

Figure 6: Repeated Forward A* Vs. Repeated Backward A*

We found that Repeated Forward A* is much faster than Repeated Backward A* in general despite some occasional cases such as fail to found path at the very beginning.

The reason for this is because for Repeated Backward A*, each time we perform an A* search, the known world is always near the goal point of current A* search, which is the start point for this Repeated Backward A*. This will cause a lot of unnecessary expansion of cells in most cases because once there is a blocked cell detected on the presumed path, the actual f-value from previous cell to goal will increase at least one. However, most cells near the start point of current A* search, which is the goal point for the Repeated Backward A*, are in the unknown world and remains the smaller original f-value. So Repeated Backward A* will choose to expand these unnecessary cells instead of choosing the one near the goal of current A* search.

For example, under the situation of figure 7,
Repeated Backward A* expanded 8011 cells in this particular A* search, whereas Repeated Forward A* expands only 180 cells. For this particular A* search, we search from (90,90) to (2,1) where the known world is near (2,1).

- The A* expanded cells from (90,90) to (90,1) with all $f-value = 177$

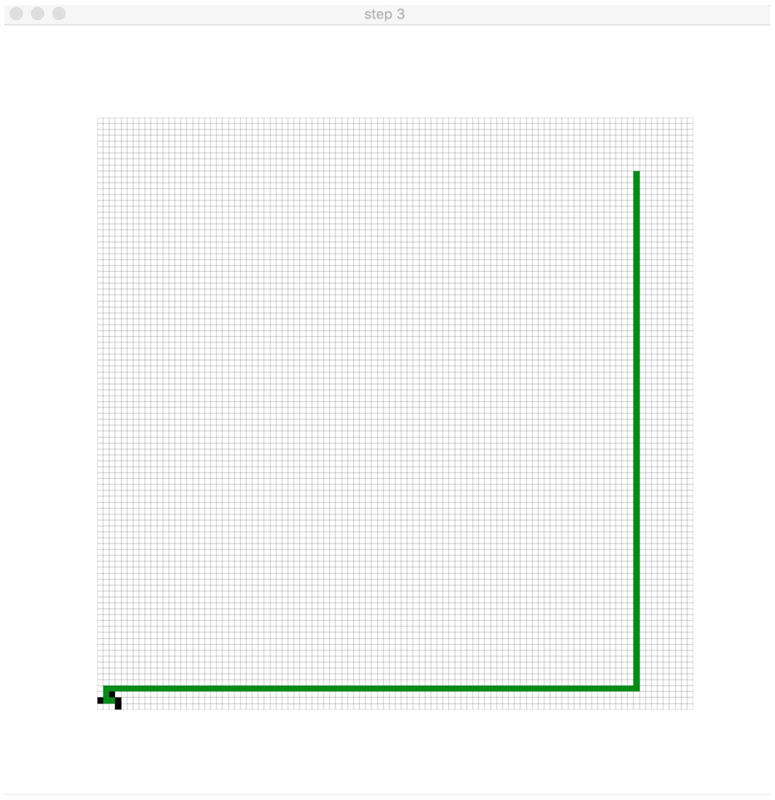

Figure 7: Repeated Backward A* Problem

- It than expanded horizontally from (90,1) until it hit the blocked cell on (3,1) with all $f - value = 177$
- Now that cell (4,2) is in the open list with g-value increased by 1, so $f((4,2)) = 178$. So it will jump back to expand cell (89,2) and goes horizontally again until it hits the blocked cell on (2,2) which is one cell above the search goal.
- Now it requires a detour to go up again with f-value increase 2. So A* will jump back to expand (89,3) which has a smaller f-value of 177 and expand horizontally again, when it expands cell (2,3), it will goes downward and hit the blocked cell on (2,2)
- It will keep expanding cells (89,4) to (89,90) and expand the entire horizontal line of cells for each one of them. this results in a extremely huge number of useless cells to be expanded.
- once every cell in the open list with $f-value = 177$ has been expanded, it will then consider to take a detour and find the correct path.

We will not have this problem if we are using Repeated Forward A* because all the known worlds are near the start point, so we will not have most cells in the open list with a smaller original f-value, and will consider go around the blocked cell very soon.

Therefore, Repeated Forward A* is much faster than Repeated Backward A* in most cases.

Part 4 - Heuristics in the Adaptive A*

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

Since in grid world, agent only move to four directions, that is up, down, left, right. Manhattan Distance is the distance where we let our agent move verticlely until it is on the same horizontal line with the goal, and then moves horizontally to the goal. This is the fastest possbile path in grid world and requires there are no blocked cellls on its path. If we have blocked cells on its path, we would have to detour a few cells and it will increase the cost. If we do not follow the path which has $cost = ManhattanDistance$, if we go up x more cells, we would eventually have to go down the same amount of cells in order to get back to the goal. And this applies also to go down, go left, and go right.

We can prove this using proof by contradiction. Lets assume Manhattan Distance is not a consistant heuristic. So,

$$\exists(n, a, m) : h(n) \geq c(n, a, m) + h(m) \tag{4}$$

where $c(n, a, m)$ is the step cost for going from n to m using action a.
We move h(m) to the left side of equation and get:

$$\exists(n, a, m) : h(n) - h(m) \geq c(n, a, m) \tag{5}$$

Since h(n) and h(m) are the Manhattan Distance from n or m to the goal, so:

$$h(n \Rightarrow m) : |h(n) - h(m)| \tag{6}$$

Where $h(n \Rightarrow m)$ is the Manhattan Distance from n to m.
Combine (5) and (6), we get:

$$\exists(n, a, m) : h(n \Rightarrow m) \geq c(n, a, m) \tag{7}$$

Since Manhattan Distance is just the difference between x and y values of two point, so:

$$h(n \Rightarrow m) = |x(n) - x(m)| + |y(n) - y(m)| \tag{8}$$

We can think of $h(n \Rightarrow m)$ as two sides of a square where one side is $|x(n) - x(m)|$ and another side is $|y(n) - y(m)|$. The only way for $h(n \Rightarrow m) \geq c(n, a, m)$ to be true is that $c(n, a, m)$ is the distance of moving through diagonals of this square or some path that is similar to diagonals, that is lines has z degrees with horizontal line such that $z \neq 0$. However, since only four directions of moves are allowed in grid worlds, which are up down left right, so this $c(n, a, m)$ does not exist in gridworlds. Therefore, Manhattan Distance is indeed a consistant heuristics in gridworlds.

Furthermore, it is argued that the h-values hnew(s) are not only admissible but also consistent. Prove that Adaptive A* leaves initially consistent h-values consistent even if action cost can increase

Since g(goal) is the smallest goal-cost reported by A* search under current knowledge of the map, and g(s) is the smallest cost to s reported by A* search under current knowledge, so $g(goal) - g(s)$ is also the smallest cost from s to goal under current knowledge of the gridworlds. Since as we explore the world, the number of unblocked cells can only increase, that is the cost to goal can only increase as well, so the new Heuristics used by Adaptive A* is the smallest cost from s to goal under current knowledge of the gridworlds.

We can prove this through direct proof. We denote h(s) as the new heuristics used by Adaptive A* from state s to goal. So h(s) can be viewed as following:

$$h(s) = g(goal) - g(s) \tag{9}$$

Assume h(s) is a consistent heuristic, we can have the following:

$$\forall(n, a, m) : h(n) \leq c(n, a, m) + h(m) \tag{10}$$

Subsititute h(n) and h(m) in (10) by (9), we get:

$$\forall(n, a, m) : g(goal) - g(n) \leq c(n, a, m) + g(goal) - g(m) \tag{11}$$

Simplify the above equation, we get:

$$\forall(n, a, m) : g(m) - g(n) \leq c(n, a, m) \tag{12}$$

$c(n, a, m)$ is smallest if n and m are on the same side of a line and no cells between them are blocked. In this case, $c(n, a, m)$ is just the difference in distance between n and m. Assume we get g(m) and g(n) from an A* search with Manhattan Distance, which is a consistent heuristic, as Heuristics. So g(m) and g(n) are both the cost follows the shortest path from start to m and n. So g(m) and g(n) are both smallest cost from start to m and n. So, g(m)-g(n) is the difference in cost or distance between m and n. As a result, we get:

$$g(m) - g(n) = c(n, a, m) \tag{13}$$

if we take the smallest possbile $c(n, a, m)$
Therefore, in general, we proved:

$$\forall(n, a, m) : g(m) - g(n) \leq c(n, a, m) \tag{14}$$

is indeed true.

Therefore, by induction, we proved the new heuristics used by Adaptive A* will indeed remain consistent.

Part 5 - Heuristics in the Adaptive A*

We implemented Repeated Forward A* algorithm with both A* and Adaptive A*. We performed 50 experiments on these 2 search algorithm with the same 101*101 gridworld and get the following result on figure 8:

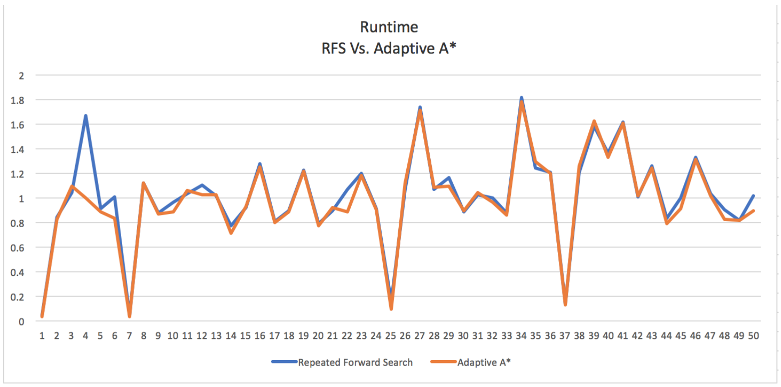

Figure 8: Repeated Forward A* Vs. Adaptive A*

We found that Repeated Forward A* with Adaptive A* is slightly faster than Repeated Forward A* with A* in general despite some occasional cases such other computer CPU usage during experiment.

The reason that Adaptive A* is faster then A* is because Adaptive A* updates the h-value of expanded cells to a higher valued consistent heuristic, which is more accurate, after each computePath(). So when next time Adaptive A* take these cells to the open list, it could make a more wise choice then the ordinary A* does. So the number of cells expanded by each Adaptive will be less than number of cells expanded by ordinary A* expanded in general.

The reason for that the difference between these two algorithms is not too much is because the map I constructed is relatively sparse and lack of pattern as figure 9 shows.

So each time, there are only a few cells near the detour area which is near the start point of current Adaptive A*, will have their h-value changed. All other cells' h-value will remain the same since the rest of girdworld remains unknown. Since the blocked cells are sparse, so the Adaptive A* aalgorithms generally does not search back to the cells which are moved by agent already. So we did not make use of these updated heuristics much. However, we spent a lot of time to traverse through the close list and update the heuristics after

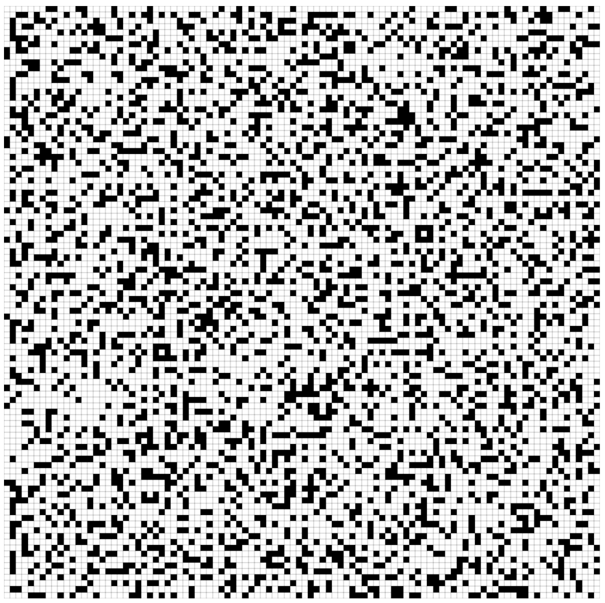

Figure 9: map

each Adaptive A*. Therefore, the difference between these two algorithms is not too much.

Part 6 - Memory Issues

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

In the current implementation of the gridworld, each cell contains 3 field – isBlocked(boolean), coordinate((x,y) tuple), and neighbors(python dictionary that has the structure:'up':list of cells, 'down': list of cells, 'left':list of cells, 'right' lsit of cells. To make future improvement on Memory usage, we can get rid of the four pointers which keeps track of neighbors of current cells since in gridworlds, we can calculate the coordinate of neighboring cells and access them directly. This will save a lot of Memory by getting rid of 4 pointers(about 32 Byte). If we need to even future imporove the Memory usage, we could save both x and y values of a cell into a single int. Since we only need to construct 101*101 gridworlds, so we can save the x value on the first 16 bit of an int, and y value on the second 16 bit of an int. We can use bit shifts to get the values we want when we need them.

Calculate the amount of memory that they need to operate on gridworlds of size 1001*1001

Since each cell in my implementation has 4 pointers, one boolean, 2 ints, so each cell occupies $4 * 8 + 2 * 4 + 1 = 41 Bytes$
so the amount of memory needed for gridworlds of size 1001 * 1001 is:

$$1001 * 1001 * 41 = 41082041 Byte = 40119 MB \tag{15}$$

Calculate the largest gridworld that they can operate on within a memory limit of 4 MBytes

Since Each cell is 41 Byte, so:

$$\frac{4 * 1024}{41} = 99.9024390244 \approx 100 \tag{16}$$

So the largest gridworld that they can operate on within a memory limit of 4 MBytes is 10*10