# Assignment 1

Anhelina Mahdzyar, Boyang Fu, Yao Shi

February 25, 2018

* - indicates unfinished question

## 1 Part 1 - Understanding the methods

### 1.1 Section a

When the agent begins in the maze program, it does not initially know which cells are blocked. It has to survey its surroundings as it takes each proceeding step. In Figure 8, the agent takes a step to the east rather than north because A* would calculate that it would take a shorter distance to the goal to go east, rather than first north and then east. Our agent can only see which cells are blocked at each subsequent step, so at its initial position, it cannot see that the path directly east is blocked both to the right and from above. Only once it moves east will it realize that it is not the correct path, and then move back to go north.

### 1.2 Section b

In this gridworld, the agent is guaranteed to reach the target of the maze (goal) or discover it is impossible to as A* search leads our agent through every possible cell in the grid to the goal. If the agent detect that the goal state is blocked on all sides by obstacles such as the grid edge or blocked cells, in the next A* search, the open list would be empty before the the algorithm pushing the goal stage node into the open list, and therefore we know we can't reach to the goal stage. Otherwise, the agent will keep moving along each cell until it reaches goal.

Use induction to prove: "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.

Let n be the total number of cells, let m be the number of unblocked cells, and let k be the total number of moves the agent took

Base case: n = 2, when there is only start state and goal state. If the goal state is blocked, then the total move would be k = 0, m = 1.

$$k < m^2 - - - check$$

If the goal state is unblocked, then m = 2, k = 1, because we directly move from start to goal with a single step.

$$k < m^2 - - - check$$

Inductive step: For n > 2, Suppose

$$k < m'^2$$

is true for all m'∈[1, m),need to prove

$$k < m^2$$

is also true. We know that for each unblocked cell, the agent would at most visit it twice (the first time follow the path to the goal, and later come back with the updated blocked information). Therefore, $k <= 2m$.

$$(\frac{2m}{m^2}) = (\frac{2}{m})$$

, since $m > m' >= 1$, therefore $m >= 2$, and further

$$(\frac{2}{m}) <= 1$$

. Thus k'/2m <= 1. Therefore $k < m^2$ hold true. Based on the base step and inductive step, we know $k < m^2$ for all m >= 1.

## 2 Part 2 - The Effects of Ties

After implementing tie-breaking in regards to the g-value, we ran our project 10 times for Repeated Forward A* and compared the average run time. In our A* code, implementation of tie-breaking begins around line 254. Here are the run-times:

Repeated Forward A*

**Choosing smaller g-value** Average run time: 1.733 s

**Choosing larger g-value** Average run time: 0.584 s

We observed that tie-breaking by the larger g-value proved to be almost three times faster than the other method. *Need to flesh out observations

## 3 Part 3 - Forward vs Backward

For this part, we compared Repeated Forward and Backward A* with tie-breaking in favor of cells with larger g-values. We used out results for Forward A* from the previous question. Here are our results:

Choosing larger g-value for tie-breaking **Repeated Forward A*** Average run time: 0.584 s

**Repeated Backward A*** Average run time: 0.862 s

We observed that Repeated Forward A* ran faster than Backward A*. *Need to flesh out observations

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

This part asks us to prove that Manhattan distances are consistent in grid worlds in which the agent can move only in the four main compass directions. The heuristic function h is said to be consisten if

$$\forall(n, a, n') : h(n) \leq c(n, a, n') + h(n'),$$

where c(n, a, n') is the step cost for going from n to n' using action a.

In this case, the step cost for each move is 1, therefore,

$$c(n, a, n') + h(n') = h(n') + 1$$

, and h(n) is either h(n') + 1 or h(n') - 1, whenever the agent makes the next move.

In the case of h(n) = h(n') + 1, h(n) = c(n, a, n') + h(n').

In the case of h(n) = h(n') - 1, h(n) < c(n, a, n') + h(n').

Thus, the statement "the Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions" is true.

The second part asks us to prove that Adaptive A* leaves initially consistent h-values consistent even if action costs can increase. The heuristic function h is said to be consisten if

$$\forall(n, a, n') : h(n) \leq c(n, a, n') + h(n'),$$

where c(n, a, n') is the step cost for going from n to n' using action a.

For each Adaptive process, let the step cost for each move to be denoted by the variable A where A is an integer greater than 0, thus,

$$c(n, a, n') + h(n') = h(n') + A$$

, and h(n) is either h(n') + A or h(n') - A, whenever the agent makes the next move.

In the case of h(n) = h(n') + A, h(n) = c(n, a, n') + h(n').

In the case of h(n) = h(n') - A, h(n) < c(n, a, n') + h(n').

Thus, the statement "Adaptive A* leaves initially consistent h-values consistent even if action costs can increase" is true.

## 5 Part 5 – Heuristics in the Adaptive A*

*Will complete analysis after Yao finishes this part Friday

# 6 Part 6 – Memory Issues

In regards to memory consumption, there are a couple of strategies that can be taken in order to reduce it. One method would be to use different variable types in order to save on some space. One simple example, for a variable that indicates whether a cell is blocked or not, this should be a simple boolean value rather than a numerical value. Another would be to limit the number of variables associated with our node class and store less information in general in each node. One of the most important ways is to examine our data structures and see if we may have implemented any other types to save on memory consumption. Another way is to avoid generating the entire map on the agent's side, and only update the nodes that the agent observes. By storing the fog of war, we are unnecessarily using space when the agent can simply create the map as it walks along the gridworld. Our implementation does this but we can see other versions of this project unnecessarily doing this. FUrhermore, we can even put the information of all fields inside of one field. Forexample, our old class cell is written as:

```
class Cell:
    def __init__(self, xPos, yPos, if_blocked, ifVisited=False):
        self.x = xPos
        self.y = yPos
        self.ifBlocked = if_blocked
        self.visited = ifVisited
```

For the sake of saving memory, our new class cell can be written as the following with get's methods:

```
class Cell:
   def __init__(self, xPos, yPos, if_blocked, ifVisited):
      # xPos is the 4-digit x-coordinate, yPos is the 4-digit y-coordinate
        # if_blocked and ifVisited either 1 or 0 in order to indicate a boolean variable
        self.xxxxyyyybv = xPos*1000000+yPos*100+if_blocked*10+ifVisited

   def getx(self):
        temp = self.xxxxyyyybv
        return int(temp / 1000000)

   def gety(self):
        temp = self.xxxxyyyybv - getx(self) * 1000000
        return int(temp / 100)

   def getIfBlokced(self):
        temp = self.xxxxyyyybv - getx(self) * 1000000 - gety(self) * 100
        return int(temp /10)

   def getIfVisited(self):
        temp = self.xxxxyyyybv
        return temp % 2
```

In this way, each object we create will just be an int type and an int type has a size of 4 bytes and we can do something similar to the class node. Let's take 4MB as 4e6 bytes, then the maximun number of grids we can have will be 4e6/4/2 = 500000. $\sqrt{500000} = 707$, which means we will can support a 707x707 gridworld for the worst case. For a 1001x1001 gridworld, it will take 1001*1001*4*2 = 8016008 bytes of memory which is roughly 8 MB of memory.

*For last question, we have to figure out the lowest amount of memory that one node can hold (examine our current node class and see if there is anything we can trim) and then divide 4MBytes by that number to see the maximum number of cells and thus gridworld that can be generated. *Calculating amount of memory needed to operate on 1001x1001 is a similar process. Calculate node size, multiple by gridworld size, get memory usage.