Introduction to Artificial Intelligence: Assignment #1 Fast Trajectory Replanning

Yuyang Chen. – yc791
Simiao fan – sf578
Zhaohan Yan – zy134

Rutgers University — February 27, 2019

Introduction

Info: This document is Homework 1 for 198:440 Introduction to Artificial Intelligence at Rutgers University Spring 2019.

1 Part 1 - Understanding the methods

Read the chapter in your textbook on uninformed and informed (heuristic) search and then read the project description again. Make sure that you understand A* and the concepts of admissible and consistent h-values.

1.1

Question 1

Explain in your report why the first move of the agent for the example search problem from Figure 8 is to the east rather than the north given that the agent does not know initially which cells are blocked.

Answer:

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

Question 2

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.

Answer:

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' belong [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 Effect of Ties

Repeated Forward A* needs to break ties to decide which cell to expand next if several cells have the same smallest f-value. It can either break ties in favor of cells with smaller g-values or in favor of cells with larger g-values. Implement and compare both versions of Repeated Forward A* with respect to their runtime or, equivalently, number of expanded cells.

Question 3

Explain your observations in detail, that is, explain what you observed and give a reason for the observation.

Answer:

After implementing tie-breaking in regards to the g-value, we ran our project 50 times for Repeated Forward A* and compared the average run time. In our A* code, implementation of tie-breaking begins around line 34 in "classinfor". Here are the run-times:

Repeated Forward A*

Choosing smaller g-value Average run time: 5.512 s

Choosing larger g-value Average run time: 0.594 s

We observed that tie-breaking by the larger g-value proved to be almost ten times faster than the other method. From the A* algorithm, f(s) = g(s) + h(s). In terms of the tie-breaking, we can conclude that having a bigger g-value also means having a smaller h-value. In our case, the g-value represents the cost of the path the agent has finished while the h-value is a prediction of the best case in the future which can potentially increase a lot. In other words, having a larger h-value gives the agent more risks of increasing its overall cost in the future. With the same f-values, the one with the larger g-value will more likely to finish its path with less cost to be added to its original prediction. It is also why the method of choosing the larger g-value is faster than the method of choosing the smaller g-value.

hello.py
```
#! /usr/bin/python

import sys
sys.stdout.write("Hello␣World!\n")
```

Fusce eleifend porttitor arcu, id accumsan elit pharetra eget. Mauris luctus velit sit amet est sodales rhoncus. Donec cursus suscipit justo, sed tristique ipsum fermentum nec. Ut tortor ex, ullamcorper varius congue in, efficitur a tellus. Vivamus ut rutrum nisi. Phasellus sit amet enim efficitur, aliquam nulla id, lacinia mauris. Quisque viverra libero ac magna maximus efficitur. Interdum et malesuada fames ac ante ipsum primis in faucibus. Vestibulum mollis eros in tellus fermentum, vitae tristique justo finibus. Sed quis vehicula nibh. Etiam nulla justo, pellentesque id sapien at, semper aliquam arcu. Integer at commodo arcu. Quisque dapibus ut lacus eget vulputate.

Command Line
```
$ chmod +x hello.py
$ ./hello.py

Hello World!
```

Vestibulum sodales orci a nisi interdum tristique. In dictum vehicula dui, eget bibendum purus elementum eu. Pellentesque lobortis mattis mauris, non feugiat dolor vulputate a. Cras porttitor dapibus lacus at pulvinar. Praesent eu nunc et libero porttitor malesuada tempus quis massa. Aenean cursus ipsum a velit ultricies sagittis. Sed non leo ullamcorper, suscipit massa ut, pulvinar erat. Aliquam erat volutpat. Nulla non lacus vitae mi placerat tincidunt et ac diam. Aliquam tincidunt augue sem, ut vestibulum est volutpat eget. Suspendisse potenti. Integer condimentum, risus nec maximus elementum, lacus purus porta arcu, at ultrices diam nisl eget urna. Curabitur sollicitudin diam quis sollicitudin varius. Ut porta erat ornare laoreet euismod. In tincidunt purus dui, nec egestas dui convallis non. In vestibulum ipsum in dictum scelerisque.

3 Part 3 - Forward vs. Backward

Implement and compare Repeated Forward A* and Repeated Backward A* with respect to their runtime or, equivalently, number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both versions of Repeated A* should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly

Answer:

Repeated Forward A*:

```
Time:  NaN Expanded node:  16709
Time:  0.312089204788208 Expanded node:  3750
Time:  0.2952127456665039 Expanded node:  2474
Time:  0.29090309143066406 Expanded node:  2526
Time:  0.3249638080596924 Expanded node:  4483
Time:  0.307081937789917 Expanded node:  3370
Time:  0.32608509063720703 Expanded node:  4493
Time:  0.3079369068145752 Expanded node:  4758
Time:  0.27632594108581543 Expanded node:  2829
Time:  0.28101491928100586 Expanded node:  2739
Time:  0.331723690032959 Expanded node:  4103
Time:  0.33367395401000977 Expanded node:  5298
Time:  0.3019721508026123 Expanded node:  2213
Time:  0.3311481475830078 Expanded node:  4909
Time:  0.30945587158203125 Expanded node:  3481
Time:  0.312636137008667 Expanded node:  3830
Time:  0.3522989749908447 Expanded node:  7106
Time:  0.2915058135986328 Expanded node:  2588
Time:  0.2895338535308838 Expanded node:  2476
Time:  NaN Expanded node:  15878
Time:  0.30850911140441895 Expanded node:  3231
Time:  NaN Expanded node:  14028
Time:  0.32351088523864746 Expanded node:  4445
Time:  0.3034937381744385 Expanded node:  2965
Time:  0.30852389335632324 Expanded node:  3884
Time:  0.3090989589691162 Expanded node:  4224
Time:  0.3077859878540039 Expanded node:  3537
Time:  0.30930304527282715 Expanded node:  3711
Time:  0.2875702381134033 Expanded node:  1993
Time:  0.31240200996398926 Expanded node:  3778
Time:  0.29363584518432617 Expanded node:  2519
Time:  0.2942039966583252 Expanded node:  2550
Time:  0.3056199550628662 Expanded node:  3639
Time:  0.3101179599761963 Expanded node:  3587
Time:  0.33412790298461914 Expanded node:  6021
Time:  0.3113059997558594 Expanded node:  4003
Time:  0.3174560070037842 Expanded node:  4814
Time:  0.34110021591186523 Expanded node:  6625
Time:  0.2971642017364502 Expanded node:  2075
Time:  0.3433341979980469 Expanded node:  7437
Time:  0.29790377616882324 Expanded node:  4200
Time:  NaN Expanded node:  15063
Time:  0.29868173599243164 Expanded node:  3958
Time:  0.2665388584136963 Expanded node:  2005
Time:  0.29404497146606445 Expanded node:  3545
Time:  0.3823089599609375 Expanded node:  10273
Time:  NaN Expanded node:  17829
Time:  0.2738497257232666 Expanded node:  2837
Time:  0.27971816062927246 Expanded node:  2692
Time:  0.3167598247528076 Expanded node:  6188
```

Repeated Forward A*: Average run time = 0.3089696089426676
Repeated Forward A*: Average expanded node = 3959.15555556

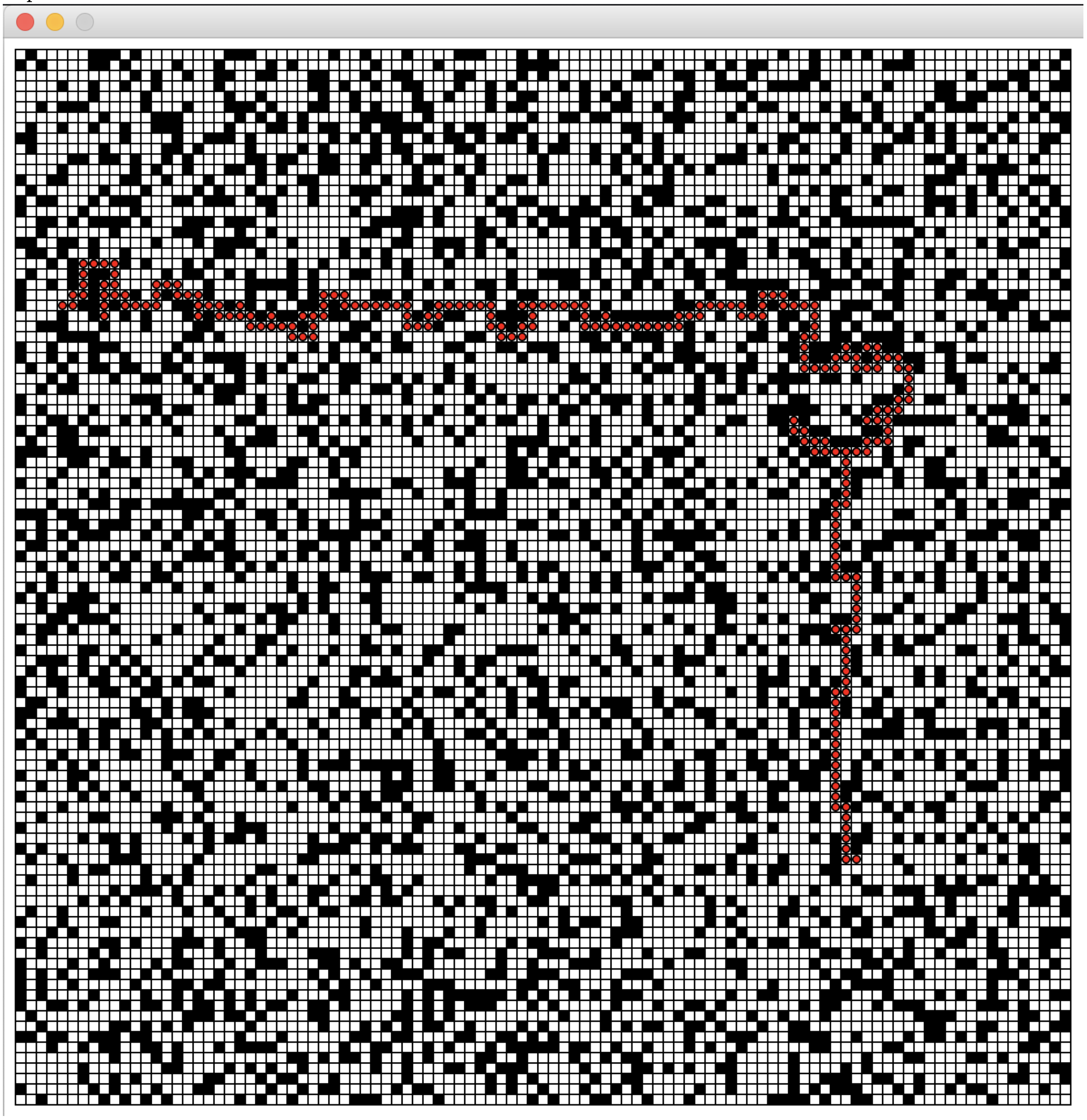
Repeated Backward A*:

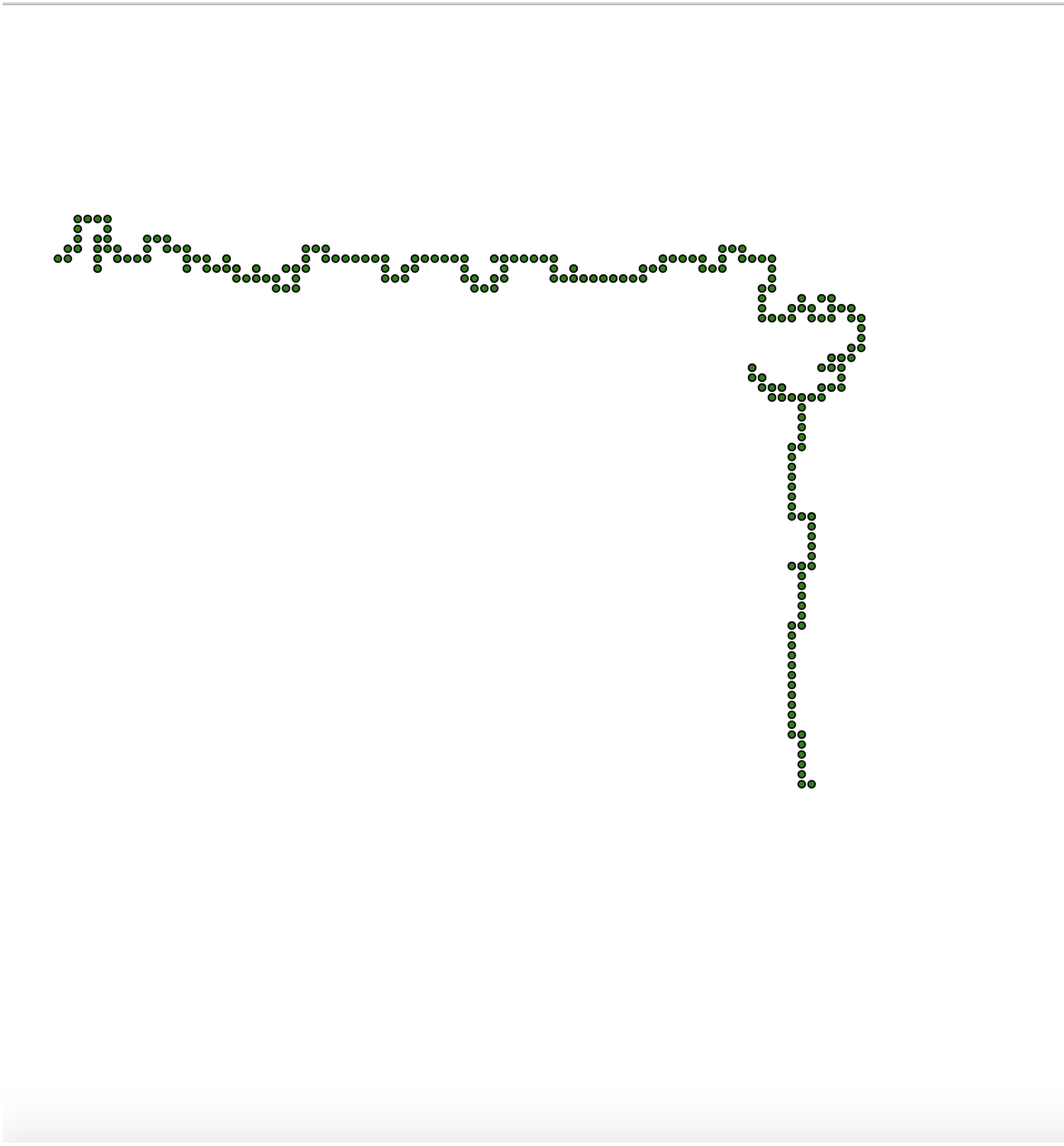
```
Time:  0.7747209072113037 Expanded node:  25899
Time:  0.3565709590911865 Expanded node:  32247
Time:  0.49796199798583984 Expanded node:  45263
Time:  0.3031759262084961 Expanded node:  47828
Time:  0.29739904403686523 Expanded node:  51167
Time:  0.3695061206817627 Expanded node:  58327
Time:  0.41967105865478516 Expanded node:  67505
Time:  0.3038358688354492 Expanded node:  71083
Time:  0.3830540180206299 Expanded node:  79943
Time:  NaN Expanded node:  81508
Time:  0.38150811195373535 Expanded node:  89416
Time:  0.3882591724395752 Expanded node:  96893
Time:  NaN Expanded node:  107088
Time:  0.29601502418518066 Expanded node:  108993
Time:  0.4219069480895996 Expanded node:  118402
Time:  0.3200681209564209 Expanded node:  121746
Time:  0.3117208480834961 Expanded node:  125012
Time:  0.3887453079223633 Expanded node:  131970
Time:  0.3076472282409668 Expanded node:  134875
Time:  0.30680322647094727 Expanded node:  137853
Time:  0.3159210681915283 Expanded node:  141105
Time:  0.32382893562316895 Expanded node:  145255
Time:  NaN Expanded node:  148260
Time:  0.29843711853027344 Expanded node:  150641
Time:  0.3123779296875 Expanded node:  154062
Time:  0.34702014923095703 Expanded node:  159583
Time:  0.7250399589538574 Expanded node:  184992
Time:  0.5247812271118164 Expanded node:  202611
Time:  0.2683238983154297 Expanded node:  204314
Time:  0.28710198402404785 Expanded node:  207338
Time:  0.26576995849609375 Expanded node:  208995
Time:  0.30237913131713867 Expanded node:  213396
Time:  0.2961390018463135 Expanded node:  217640
Time:  0.29099607467651367 Expanded node:  221052
Time:  0.27379631996154785 Expanded node:  223018
Time:  0.3632688522338867 Expanded node:  231695
Time:  0.3521432876586914 Expanded node:  238770
Time:  NaN Expanded node:  248965
Time:  0.3310661315917969 Expanded node:  254992
Time:  0.29046010971069336 Expanded node:  258009
Time:  0.2807750701904297 Expanded node:  260544
Time:  0.3036940097808838 Expanded node:  264673
Time:  0.3140289783477783 Expanded node:  268752
Time:  0.3415558338165283 Expanded node:  275268
Time:  0.2754850387573242 Expanded node:  277999
Time:  0.35358381271362305 Expanded node:  283134
Time:  0.3477909564971924 Expanded node:  289280
Time:  0.34853506088256836 Expanded node:  297395
Time:  0.2776679992675781 Expanded node:  299857
Time:  0.28941798210144043 Expanded node:  303213
```

Repeated Backward A*: Average run time = 0.3506512123605479
Repeated Backward A*: Average expanded node = 176390.76087

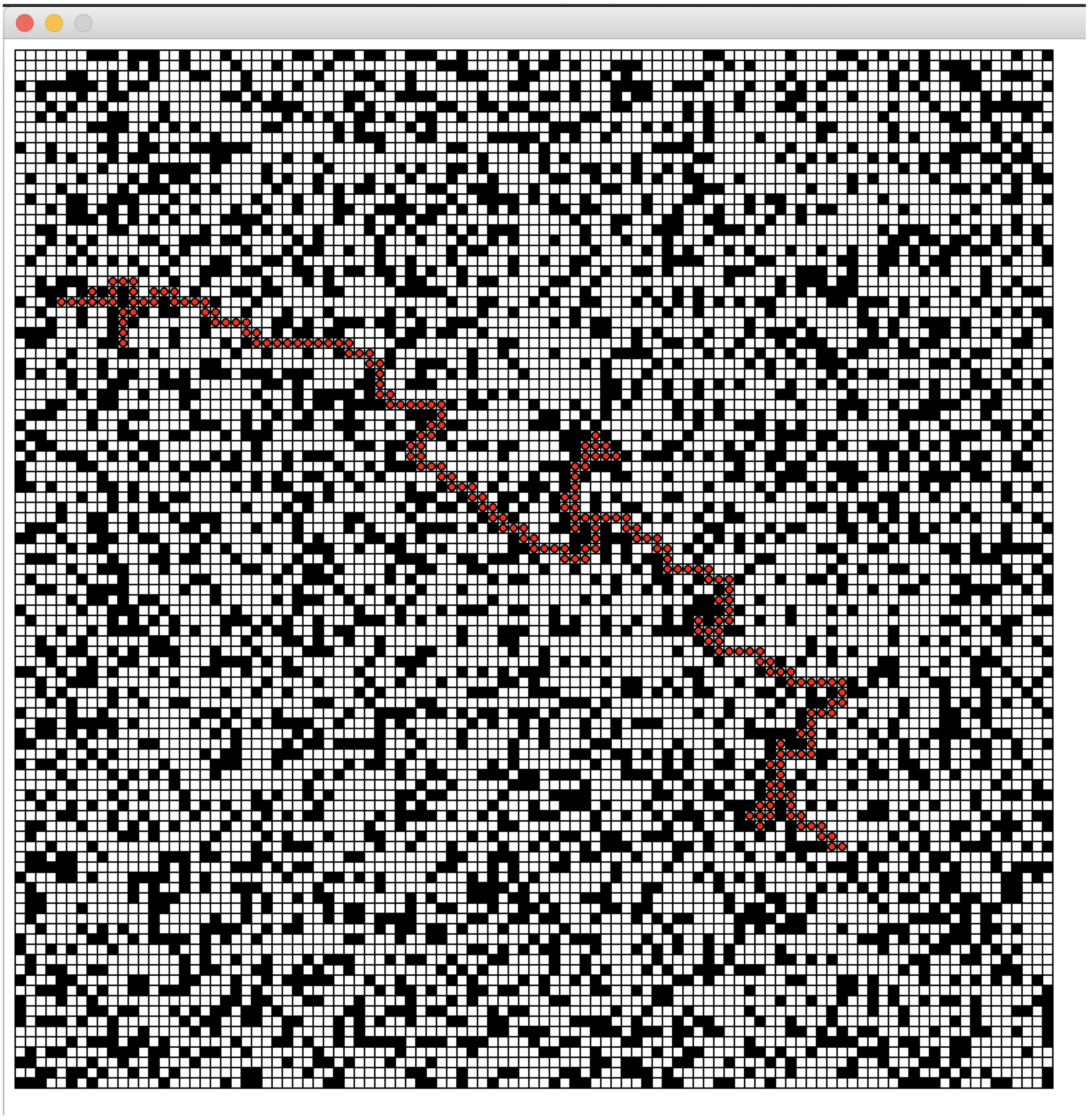
Obersvation : After comparing the run time and the number of expanded node of Repeated Forward and Backward, we can easily find that Repeated Forward A* ran faster than Backward A. But in this situation, the origin node and destination node are not the same.
By setting another python file to test, we set the origin node is same as the destination node. We use it to compare it again:

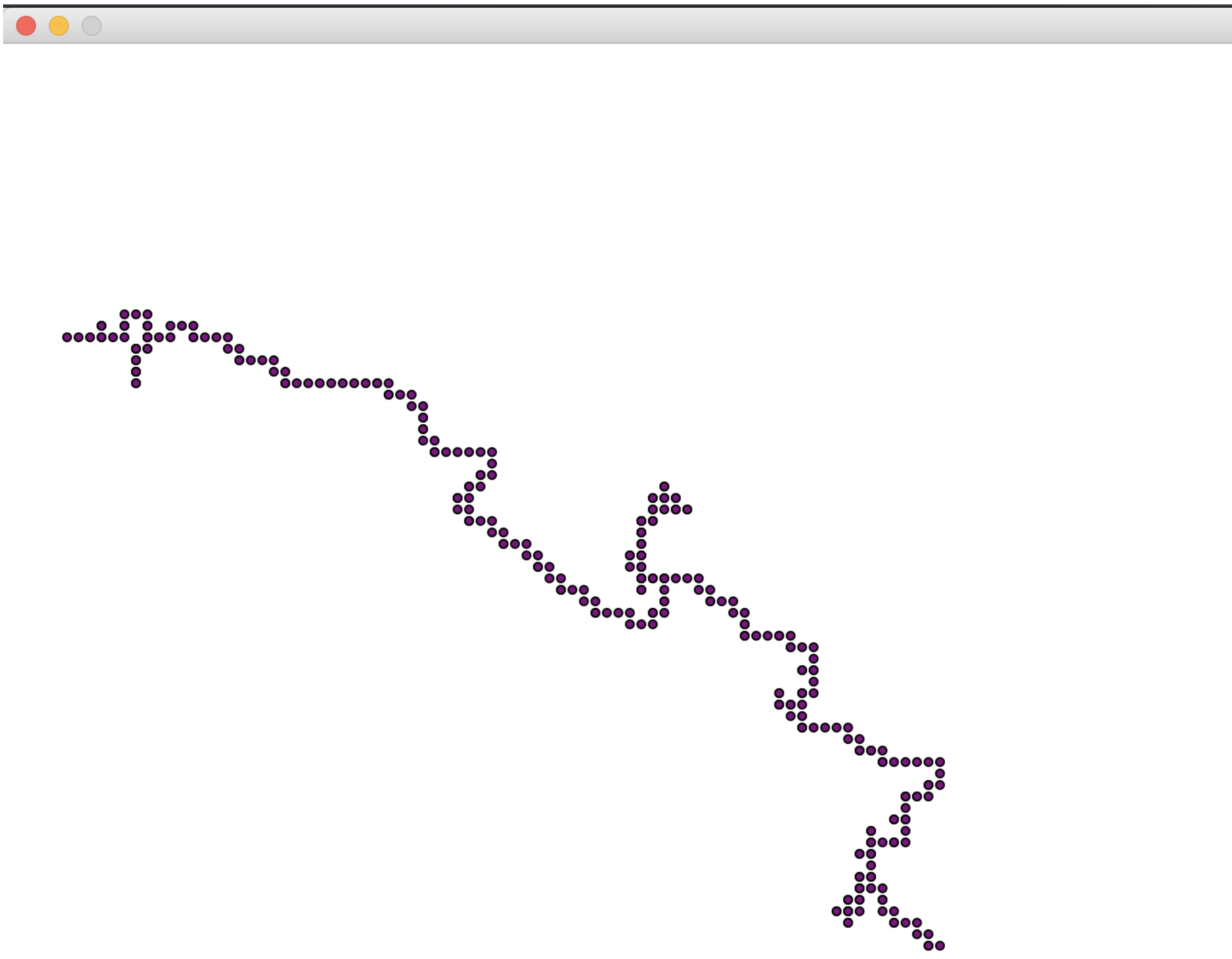
Repeated Forward:

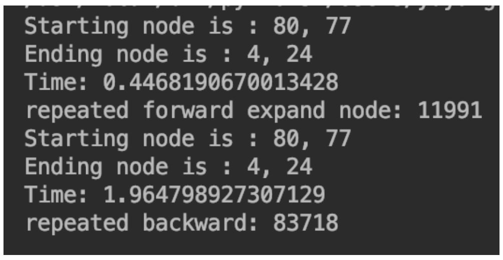




Repeated Backward:







```
Starting node is : 80, 77
Ending node is : 4, 24
Time: 0.4468190670013428
repeated forward expand node: 11991
Starting node is : 80, 77
Ending node is : 4, 24
Time: 1.964798927307129
repeated backward: 83718
```

By comparing these two condition, we can easily say that the running time of Repeated Forward A* is obviously faster than the Repeated Backward A*.

Explanation:

4 Part 4 - Heuristics in the Adaptive A*

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

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 costs can increase.

Answer:
The heuristic function h is said to be consistent 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 grid-worlds 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 consistent 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,

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

.

As is known, for Adaptive A* search, $h_{new} = g(s_{goal}) - g(s)$, and we need to prove:

$$h_{new}(s) \leq h_{new}(s') + A,$$

which is equal to:

$$g(s_{goal}) - g(s) \leq g(s_{goa})) - g(s') + A$$
$$=> g(s') \leq (s) + A,$$

which is true.

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*

Implement and compare Repeated Forward A* and Adaptive A* with respect to their runtime. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both search algorithms should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly.

Answer:

6 Part 6 - Memory Issues

You performed all experiments in gridworlds of size 101 × 101 but some real-time computer games use maps whose number of cells is up to two orders of magnitude larger than that. It is then especially important to limit the amount of information that is stored per cell. For example, the tree-pointers can be implemented with only two bits per cell. Suggest additional ways to reduce the memory consumption of your implementations further. Then, calculate the amount of memory that they need to operate on gridworlds of size 1001 × 1001 and the largest gridworld that they can operate on within a memory limit of 4 MBytes.

Answer:

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 grid-world. Our implementation does this but we can see other versions of this project unnecessarily doing this. Furthermore, we can even put the information of all fields inside of one field. For example, 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 vari
        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 maximum number of grids we can have will be 4e6/4/2 = 500000. $\sqrt{500000} = 707$, which means we will support a 707x707 grid-world for the worst case. For a 1001x1001 grid-world, it will take 1001*1001*4*2 = 8016008 bytes of memory which is roughly 8 MB of memory.