§CS 520: Assignment 1
Fast Trajectory Replanning

Xiaoyang Xie
167008240

Yikun Xian
168000142

Department of Computer Science
Rutgers University, New Brunswick, NJ

09 October 2015

Abstract

Heuristic search algorithms like A* can be adapted to solving path planning problems of directed goal and unknown environment. In this report, we mainly discuss and evaluate three variants of A* algorithms, namely Repeated Forward A*, Repeated Backward A* and Adaptive A*. In the experiment, we first generate two sets of 1000 random grid-based maps, where one follows the assignment requirement containing approximate 30% obstacles and 70% roads, and the other is the set of corridor-like mazes generated by DFS with randomly expanded nodes. Then we compare three algorithms by such evaluation indices as number of expanded nodes, number of explored nodes, total cost of steps, optimal cost of steps, etc. The result shows that 1) the average number of expanded nodes per map and explored nodes per map of Adaptive A* are respectively 28.06% and 24.00% less than those of Repeated Forward A*; 2) cost of steps for all three algorithms is pretty much the same. This indicates that Adaptive A* is greatly optimized in path replanning phase, while there is no significant improvement in actual moving phase. Finally, we discuss and calculate how to optimize data structures to store states as many as possible within only 4M memory. This is the practical problem in the situation where computational resources are rare and precious.

Part 0 Setup Environment

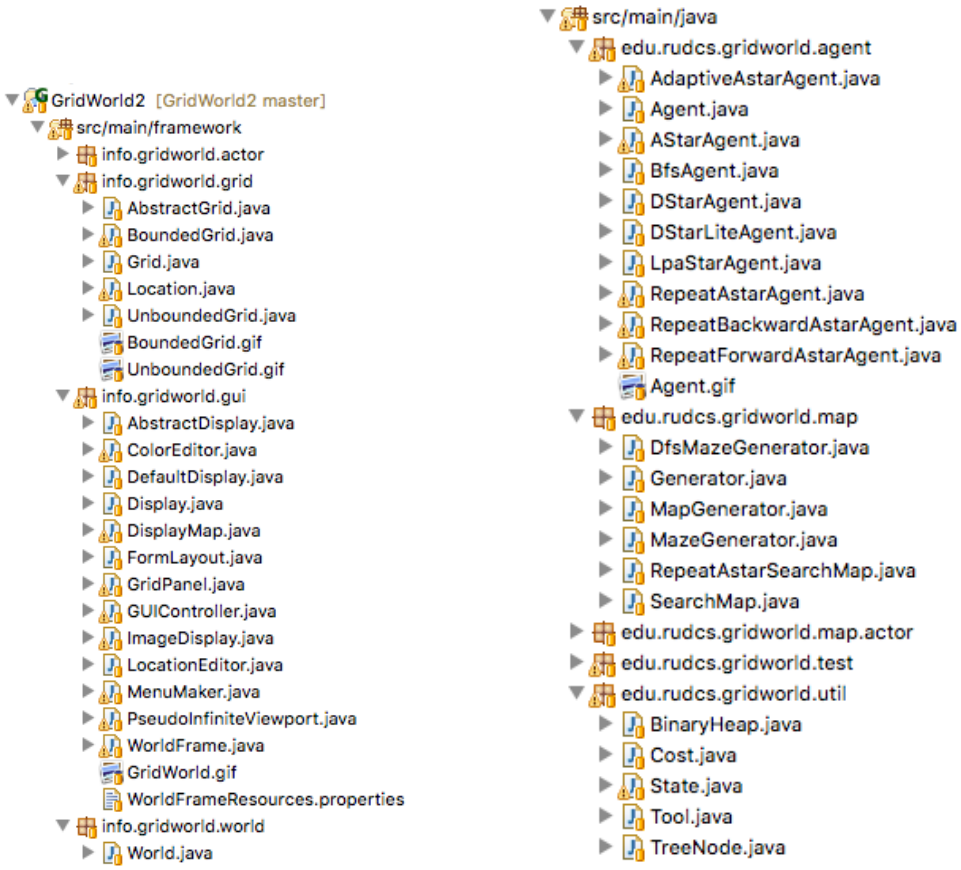
We simulate all path finding processes based on the framework of GridWorld[1], an AP case study project from collegeboard [1]. It provides graphical user interface based on Java AWT where visual objects can interact and perform customized actions in a two-dimensional grid map. In the next part, we will first illustrate original GridWorld framework and our enhancement of displaying colored path. This mainly involves the engineering work, so if you want to directly delve into algorithm analysis, please skip it.

0.1 GridWorld Architecture and Modification

The source code of original GridWorld project is placed in $src/main/framework$ folder and its structure can be divided into four parts, as shown in Figure 1a.

The *actor* package contains objects whose behavior on the map can be arbitrarily defined by rewriting *act* method in each inherited class:

[1] https://www.collegeboard.org/



(a) GridWorld structure (b) Project structure

Figure 1: Structure of GridWorld framework and path-finding project

```
@Override
public void act()
```

The *grid* package defines features on bounded and unbounded grid map as well as connections between map and actors. The *gui* package encapsulates low-level Java AWT to provide APIs for visualization and interactions of map and actors. The *world* package provides high-level integration of actors and world.

In order to visualize the presumed unblocked path and the set of nodes expanded and explored after each planning, we enhance the *GridPanel* class in the GUI package by implementing the method below:

```
private void drawColoredLocations(Graphics2D g2)
```

Meanwhile, following five abstract methods related to colored grid are added to *Grid* interface and classes like *BoundedGrid* and *UnboundedGrid* are responsible

to implement color configuration on each grid.

```
ArrayList<Location> getColoredLocations();
Color getColor(Location loc);
void putColor(Location loc, Color color);
void removeColor(Location loc);
void resetColors();
```

The source code related to assignment is placed in *src/main/java* folder, as shown in Figure 1b, which are divided into five packages. The *agent* package defines the simulated agents equipped with specific navigation algorithm for solving goal-directed path-finding problem. The *map* package provides APIs for generation of random map and corridor-like maze. The *map.actor* package defines some static object on the map like obstacles and goal. The *test* package contains simulation program for experiment and testing. The *util* package includes some self-defined data structures to be used in storing intermediate result of algorithm.

0.2 How to Run

Our project is built and packed by Gradle[2], an open-source build automation tool. To run the project, you should first install and setup Gradle environment on your computer, and install Gradle plugin for Eclipse, called "Gradle Integration for Eclipse". Then, download complete source code of the project, which is attached in Sakai or available on the Github (`https://github.com/orcax/GridWorld2`). Finally, inside Eclipse, click *File* $\rightarrow$ *Import* $\rightarrow$ *Gradle* $\rightarrow$ *Gradle Project* to import project. All runnable programs are placed in *test* package.

0.3 Maze Generation Algorithm

In the experiment, we mainly use two kinds of grid map, namely random map with discrete obstacles and random maze with consecutive obstacles. The random map is generated according to the assignment requirement, so for each map, there are about 30% obstacles and 70% roads. The start position and the goal position can be placed either randomly or on diagonal corners. Difference between these two ways lies in the absolute distance between start and goal, although, the experimental result implies that this initial distance in this type of random map has nearly no influence on the performance of three algorithms. On the other hand, the longer the distance is, the more probable it will lead to unreachable goals, so

[2]http://gradle.org/

in this case, we simply collect maps that exist a path between the start position and goal position.

Random maze is the other type of maps tested in the experiment. It is generated according to DFS by expanding neighbors randomly. The generation process is described as Algorithm 1.

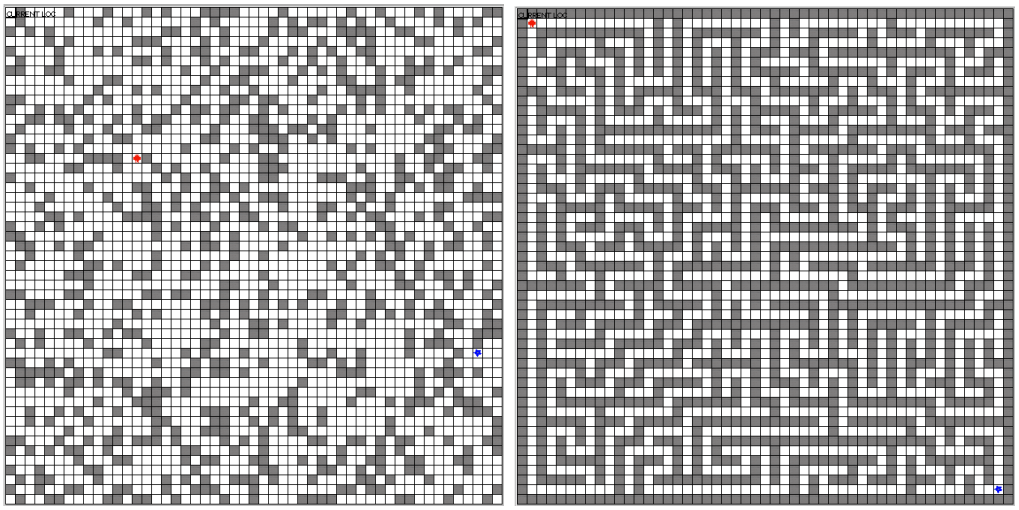
Algorithm 1 Maze Generation Algorithm by DFS

```
1: function GENERATE-BY-DFS(map)
2:     set all cells in map as WALL
3:     r = random(⌊map.rows/2⌋) * 2 + 1
4:     c = random(⌊map.cols/2⌋) * 2 + 1
5:     map[r][c] = ROAD
6:     call RANDOM-DFS(map, r, c)
7:     map[1][1] = START
8:     map[map.rows − 2][map.cols − 2] = GOAL

1: function RANDOM-DFS(map, row, col)
2:     (row′, col′)=RANDOM-NEIGHBOR(map, row, col)
3:     while (row′, col′) ≠ NULL do
4:         map[row′][col′] = ROAD
5:         map[(row + row′)/2][(col + col′)/2] = ROAD
6:         call RANDOM-DFS(map, row′, col′)
7:         (row′, col′)=RANDOM-NEIGHBOR(map, row, col)

1: function RANDOM-NEIGHBOR(map, row, col)
2:     initialize neighbors as empty list
3:     if row − 2 > 0 and map[row − 2][col] ≠ ROAD then
4:         neighbors.add((row − 2, col))
5:     if row + 2 < map.rows and map[row + 2][col] ≠ ROAD then
6:         neighbors.add((row + 2, col))
7:     if col − 2 > 0 and map[row][col − 2] ≠ ROAD then
8:         neighbors.add((row, col − 2))
9:     if col + 2 < map.cols and map[row][col + 2] ≠ ROAD then
10:        neighbors.add((row, col + 2))
       return neighbors[random(neighbors.size)]
```

In this algorithm, the map is initialized as two-dimensional array and set all cells as obstacles (WALL). Roads will be expanded randomly and paved only on the cells of odd index. To generate roads, we first begin with randomly picking a cell on the map and set it as road. Then, randomly select one of its non-road neighboring nodes (of odd index), set it and the node between them as roads. Repeat this



(a) Map with discrete obstacles (b) Corridor-like maze

Figure 2: Example of random map and random maze

procedure until there is no valid neighboring nodes. If this happens, traceback to its parent node and do the same thing again. At last, all nodes of odd index are traversed and some nodes of even index are set as road to connect two adjacent nodes. Finally, without loss of generality, we simply set top-left odd bottom-right respectively as start position and goal position.

It is easy to prove that there is only one depth first tree generated during one complete search. The connectivity of single depth first tree make the maze always reachable, namely there always exists one path from start to goal. Example of two kinds of map of size 50*50 is shown in Figure 2.

Part 1 Understanding the Methods

1.1 Problem A

Problem Explain in your report why the first move of the agent for the example search problem from Figure 3 is the east rather than then north given that then agent does not know initially which cells are blocked.

Solution In Figure 3, the start state A is at $(E, 2)$ and goal state T is at $(E, 5)$. Let's denote A_N as the north cell $(D, 2)$ of A, so $g(A_N) = 1$ and $h(A_N) = 3$, where

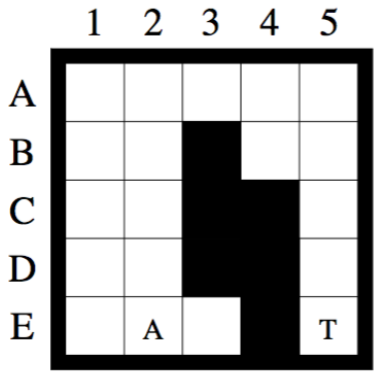


Figure 3: Second Example Search Problem

$h(A_N)$ is calculated as Manhattan distance from A_N to T. Then, we have

$$f(A_N) = g(A_N) + h(A_N) = 4$$

Equivalently, let's denote A_E as the east cell $(E, 3)$ of A, so $g(A_E) = 1$, and $h(A_E) = 2$. Then

$$f(A_E) = g(A_E) + h(A_E) = 3$$

For each iteration in ComputePath, the state with smaller $f-value$ will be chosen from the $OPEN$ list. In this case, A_E is chosen since $f(A_E) < f(A_N)$, the agent will first move to the east rather than to the north.

1.2 Problem B

Problem 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.

Solution For the first question, a finite gridworld can be seen as an undirected graph, which is divided by the blocked cells into several connected components. The start cell of agent and the goal must be assigned to any of these connected components. In a connected component, every cell is reachable from all other cells in the component. This means an agent can uses search algorithms like DFS or A*

to traverse all the cells in the same connected component in finite time. If start cell and goal cell lie on the same component, the agent can definitely reach the goal, otherwise, it is impossible for agent to reach the goal after all the cells are traversed.

For the second question, suppose there are totally K unblocked cells in a map. Now, let's consider the number of moves in each iteration. For the 1st iteration, the agent will just move from the start cell to a nearby cell, so the number of moves is 1. For the 2nd iteration, the agent may either move to an adjacent new cell or trace back to the start cell and move to the other new cell next to the start cell, so the largest number of moves is 2. Generally, for the ith iteration, the agent will visit exact one new cell, where a "new cell" is said to be the cell that has never been visited in previous $i-1$ iterations. Therefore, the largest number of moves is i. Since there are K unblocked cells, there are only $K-1$ iterations. The worst case is that in each iteration, the agent always traverse the longest path to reach the new cell. Let's denote $N(k)$ as the total number of moves after ith iteration and we have

$$N(k) = \sum_{i=1}^{i=k-1} = \frac{k(k-1)}{2} = O(k^2)$$

If the start cell and the goal cell are in the same connected component of the map, $k = K$ so that $N(k) = N(K) = O(K^2)$, otherwise, $k < K$ so that $N(k) < N(K) = O(K^2)$. Therefore, in the end, the number of moves of the agent is bounded from above by the number of unblocked cells squared.

Part 2 The Effects of Ties

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

Solution

Part 3 Forward vs. Backward

Part 4 Heuristics in the Adaptive A*

Part 5 Heuristics in the Adaptive A*

Part 6 Memory Issues

Solution: The data structure of states in our project show below.

state			
type	name	bits	description
Integer	column	10	the column of state in grid
	row	10	the row of state in grid
Boolean	bSearch	1	reflect whether the cell of this state have been found
Boolean	canMove	1	reflect whether the cell of this state is viable

Denote s as the the space of each state, S as the space of the array of states.Using this data structure, s = 22 bits. In the map of size 1001 $\times$ 1001, then

$$S = s * 1001 * 1001 = 2.6Mbyte.$$

Denote g(s) as the cost from start state to state s, during each iteration, namely the procedure of compute path, the array g need to store all notes which have been explored. Denote h(s) as the estimate cost from state s to goal state, during each iteration, namely the procedure of compute path, the array h need to store all notes which have been explored. Denote search(s) as the explore times of state s, during each iteration,the array s need to store all notes which have been explored. Denote close(s) as the state which have been expanded,during each iteration, the array cost close need to store all states which have been expand. Denote open(s) as the state s will be expand in a priority heap sorted by the sum of heuristic value and g(s). The heap open need to store nearly all notes which have been explored. The type of the value of each elements in the data structure above is integer. Assume the start state and goal state are in the opposite endpoint in the diagonal line of the map. Then each iteration, the number of expand will near the sum of rows and columns, which rows is the total number of rows in map and columns is the total number of columns. So the number of expand $\approx$ 2000. Then

$\times$ 1001, then

$$g = 4 * 2000 * (32 + 34) \approx 65KByte$$
$$h = 4 * 2000 * (32 + 34) \approx 65KByte$$
$$search = 4 * 2000 * (32 + 34) \approx 65KByte$$
$$close = 2000 * 34 \approx 8KByte$$
$$open = 4 * 2000 * 34 \approx 65KByte$$
$$tree = 4 * 2000 * (32 + 34) \approx 65KByte$$

so, the total space used: $S_{total} = S + g + search + close + open + tree \approx 3MByte$, which is less than 4 Mbyte.Then the project can operate within a memory limit of 4 Mbyte, using a map of size 1001 $\times$ 1001.

References

[1] CollegeBoard. AP central gridworld case study. `http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/151155.html`. [Online; accessed 2-October-2015].

[2] Sven Koenig and Maxim Likhachev. Real-time adaptive a*. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 281–288. ACM, 2006.

[3] Wikipedia. Maze generation algorithm — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Maze_generation_algorithm&oldid=679876968`. [Online; accessed 4-October-2015].