CS 520: Assignment 1 - Path Planning and Search Algorithms 16:198:520

Due 10/2/2017 by Midnight

This project is intended as an exploration of various search algorithms, both in the traditional application of path planning, and more abstractly in the construction and design of complex objects. You will first generate and solve simple mazes using the classical search algorithms (BFS/DFS/A^*). Once you have written these algorithms, you will utilize other search algorithms *to generate mazes that your initial algorithms have trouble solving.*

1 Part 1: (Shortest?) Path Planning

Generating Environments: In order to properly compare these algorithms, they need to be run multiple times over a variety of environments. A **map** will be a square grid of cells / locations, where each cell is either empty or occupied. An agent wishes to travel from the upper left corner to the lower right corner, along the shortest path possible. The agent can only move from empty cells to neighboring empty cells in the up/down direction, or left/right - each cell has potentially four neighbors.

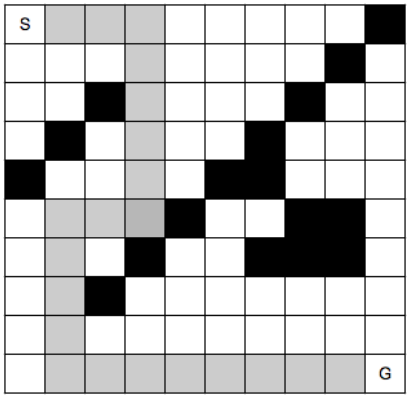

Figure 1: Successful - A path exists from start to finish.

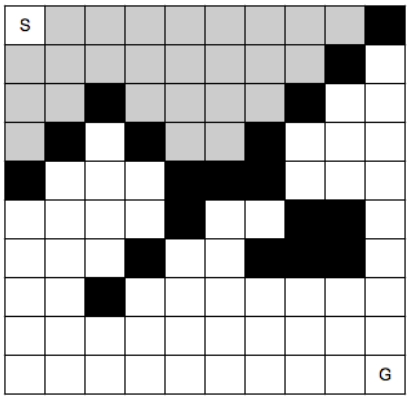

Figure 2: Unsuccessful - No path exists from start to finish.

Maps may be generated in the following way: for a given dimension **dim** construct a **dim x dim** array; given a probability p of a cell being occupied ($0 < p < 1$), read through each cell in the array and determine at random if

it should be filled or empty. When filling cells, exclude the upper left and lower right corners (the start and goal, respectively). It is convenient to define a function to generate these maps for a given **dim** and p.

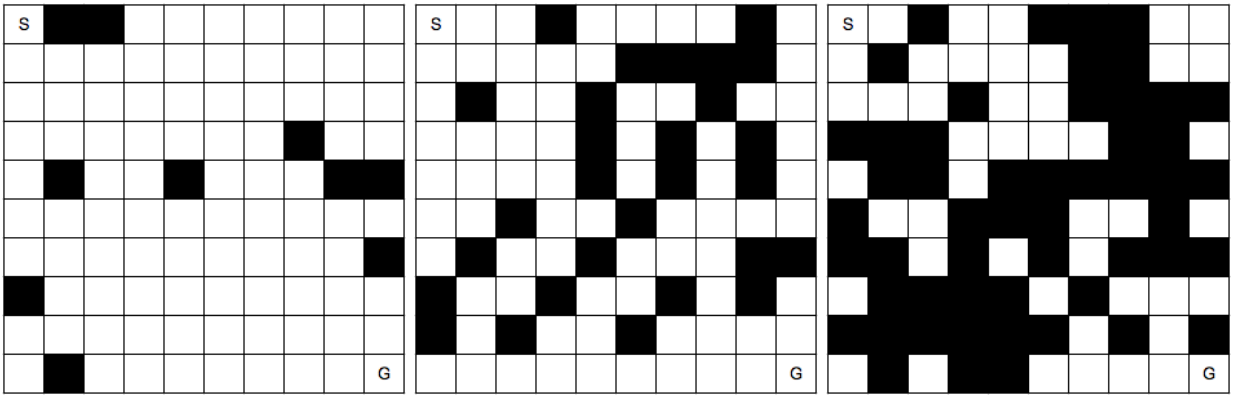

Figure 3: Maps generated with $p = 0.1, 0.3, 0.5$ respectively.

Path Planning: Once you have the ability to generate maps with specified parameters, implement the ability to search for a path from corner to corner, using each of the following algorithms:

- Depth-First (Graph) Search
- Breadth-First (Graph) Search
- A^*: where the heuristic is to estimate the distance remaining via the **Euclidean Distance**

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{1}$$

- A^*: where the heuristic is to estimate the distance remaining via the **Manhattan Distance**

$$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|. \tag{2}$$

For any specified map, applying one of these search algorithms should either return failure, or a path from start to goal in terms of a list of cells taken. *(It may be beneficial for some of these questions to return additional information about how the algorithm ran as well.)*

Questions:

1) For each of the implemented algorithms, how large the maps can be (in terms of **dim**) for the algorithm to return an answer in a reasonable amount of time (less than a minute) for a range of possible p values? Select a size so that running the algorithms multiple times will not be a huge time commitment, but that the maps are large enough to be interesting.

2) Find a random map with $p \approx 0.2$ that has a path from corner to corner. Show the paths returned for each algorithm. (Showing maps as ASCII printouts with paths indicated is sufficient; however 20 bonus points are available for coding good visualizations.)

3) For a fixed value of **dim** as determined in Question (1), for each $p = 0.1, 0.2, 0.3, \ldots, 0.9$, generate a number of maps and try to find a path from start to goal - estimate the probability that a random map has a complete path from start to goal, for each value of p. Plot your data. Note that for p close to 0, the map is nearly empty and the path is clear; for p close to 1, the map is mostly filled and there is no clear path. There is some threshhold value p_0 so that for $p < p_0$, there is usually a clear path, and $p > p_0$ there is no path. Estimate p_0. Which path finding algorithm is most useful here, and why?

4) For a range of p values (up to p_0), generate a number of maps and estimate the average or expected length of the shortest path from start to goal. You may discard all maps where no path exists. Plot your data. What path finding algorithm is most useful here?

5) For a range of p values (up to p_0), estimate the average length of the path generated by A^* from start to goal (for either heuristic). Similarly, for the same p values, estimate the average length of the path generated by DFS from start to goal. How do they compare? Plot your data.

6) For a range of p values (up to p_0), estimate the average number of nodes expanded in total for a random map, for A^* using the Euclidean Distance as the heuristic, and using the Manhattan Distance as the heuristic. Plot your data. Which heuristic typically expands fewer nodes? Why? What about for p values above p_0?

7) For a range of p values (up to p_0), estimate the average number of nodes expanded in total for a random map by DFS and by BFS. Plot your data. Which algorithm typically expands fewer nodes? Why? How does either algorithm compare with A^* in Question (6)?

Bonus 1) Why were you not asked to implement UFCS?

2 Part 2: Building Hard Mazes

In the previous section, mazes were generated essentially distributing obstructions at random through the environment. Examining these mazes, you found that certain algorithms had various advantages and disadvantages. In this section, you are going to try to construct mazes that are hard for these algorithms to solve, either in terms of a) the length of the shortest path, b) total number of nodes expanded, or c) the maximum size of the fringe.

One potential approach would be the following: for a given solution algorithm, generate mazes at random and solve them, and keep track of the 'hardest' maze you've seen so far. However, this search approach necessarily does not learn from any of its past results - having discovered a particularly difficult maze, it has no mechanism for using that to discover new, harder mazes. Every round starts again from scratch.

One way to augment this approach would be a **random walk**. Generate a maze, and solve it to determine how 'hard' it is. Then at random, add or remove an obstruction somewhere on the current maze, and solve this new configuration. If the result is harder to solve, keep this new configuration and delete the old one. Repeat this process. This has some improvements over repeatedly generating random mazes as above, but it can be improved upon still. For this part of a project, you will pick one of the **local search algorithms** (other than a random walk) and implement it to try to discover hard to solve mazes. Mazes that admit no solution may be discarded, we are only interested in solvable mazes.

Questions:

8) What local search algorithm did you pick, and why? How are you representing the maze/environment, to be able to utilize your chosen search algorithm? What design choices did you have to make to apply this search algorithm to this problem?

9) Unlike the problem of solving a maze, for which the 'goal' is well-defined, it is difficult to know when we have constructed the 'hardest' maze. That being so, what kind of termination conditions can you apply to your search algorithm to generate 'hard' if not the 'hardest' mazes?

10) For each of the following algorithms, do the following: Using your local search algorithm, for each of the following properties indicated, generate and present three mazes that attempt to maximize the indicated property. Do

you see any patterns or trends? How can you account for them? What can you hypothesize about the 'hardest' maze, and how close do you think you got to it?

a) DFS

a.i) Length of solution path returned

a.ii) Total number of nodes expanded

a.iii) Maximum size of fringe during runtime

b) BFS

b.i) Length of solution path returned

b.ii) Total number of nodes expanded

b.iii) Maximum size of fringe during runtime

c) A^* with Euclidean Distance Heuristic

c.i) Length of solution path returned

c.ii) Total number of nodes expanded

c.iii) Maximum size of fringe during runtime

d) A^* with Manhattan Distance Heuristic

d.i) Length of solution path returned

d.ii) Total number of nodes expanded

d.iii) Maximum size of fringe during runtime