CS 520: Assignment 1 - Search

16:198:520

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.

1 Environments and Algorithms

Generating Environments: In order to properly compare pathing 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.

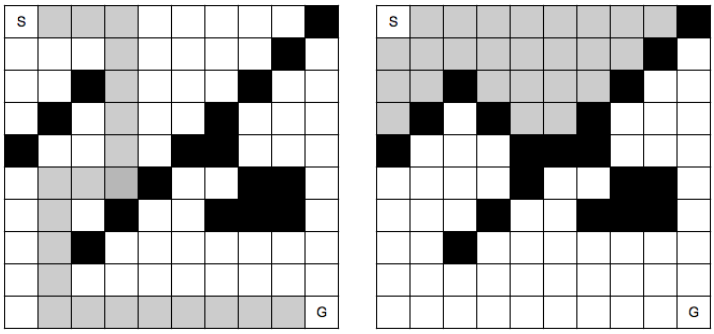

Figure 1: Successful and Unsuccessful Maze Environments.

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.

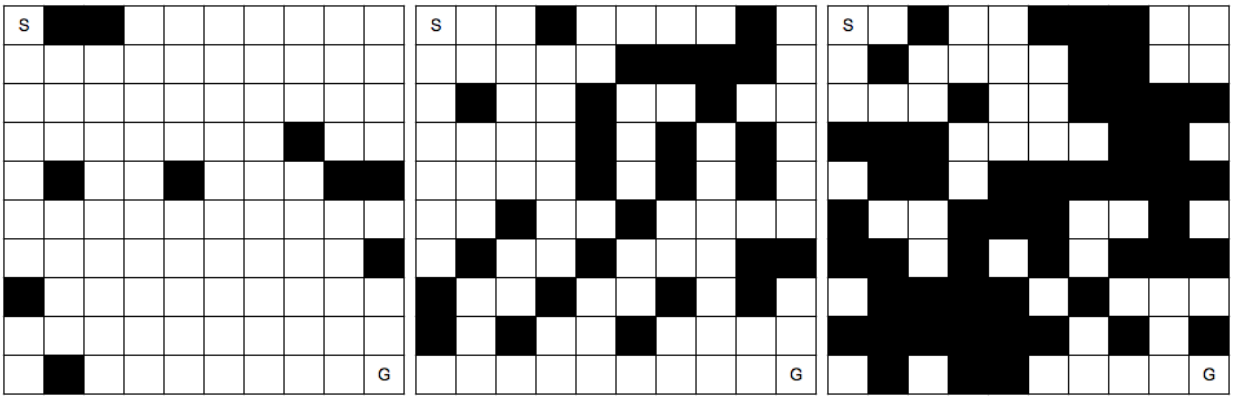

Figure 2: Maps generated with $p = 0.1, 0.3, 0.5$ respectively.

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 Search
- Breadth-First 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.)*

2 Analysis and Comparison

Having coded four path-generating algorithms, we want to analyze and compare their performance. This is important not only for theoretical reasons, but also to check to make sure that your algorithms are behaving as they should.

- Find a map size (**dim**) that is large enough to produce maps that require some work to solve, but small enough that you can run each algorithm multiple times for a range of possible p values. *How did you pick a **dim**?*
- For $p \approx 0.2$, generate a solvable map, and show the paths returned for each algorithm. *Do the results make sense? ASCII printouts are fine, but good visualizations are a bonus.*
- Given **dim**, how does maze-solvability depend on p? For a range of p values, estimate the probability that a maze will be solvable by generating multiple mazes and checking them for solvability. What is the best algorithm to use here? Plot *density vs solvability*, and try to identify as accurately as you can the threshold p_0 where for $p < p_0$, most mazes are solvable, but $p > p_0$, most mazes are not solvable.
- For p in $[0, p_0]$ as above, estimate the average or expected length of the shortest path from start to goal. You may discard unsolvable maps. Plot *density vs expected shortest path length.* What algorithm is most useful here?
- Is one heuristic uniformly better than the other for running A^*? How can they be compared? Plot the relevant data and justify your conclusions.
- Is BFS will generate an optimal shortest path in this case - is it always better than DFS? How can they be compared? Plot the relevant data and justify your conclusions.
- Do these algorithms behave as they should?
- For DFS, can you improve the performance of the algorithm by choosing what order to load the neighboring rooms into the fringe? What neighbors are 'worth' looking at before others? Be thorough and justify yourself.

Bonus: How does the threshold probability p_0 depend on n? Be as precise as you can.

3 Generating Hard Mazes

So far we have looked only at randomly generated mazes, and looked at the average behavior of algorithms over these mazes. But what would a 'hard' maze look like? Three possible ways you might quantify hard are: a) how long the shortest path is, b) the total number of nodes expanded during solving, and c) the maximum size of the fringe at any point in solving.

One potential approach to generating hard mazes would be the following: for a given solving algorithm, generate random mazes and solve them, keeping track of the 'hardest' maze generated so far. This would, over time, result in progressively harder mazes. However, this does not learn from past results - having discovered a particularly difficult maze, it has no mechanism for using that to discover new, harder mazes. Each round starts 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 must design a **local search algorithms** (other than the directed random walk described here) 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.

- What local search algorithm did you pick, and why? How are you representing the maze/environment to be able to utilize this search algorithm? What design choices did you have to make to make to apply this search algorithm to this problem?
- Unlike the problem of solving the maze, for which the 'goal' is well-defined, it is difficult to know if you have constructed the 'hardest' maze. What kind of termination conditions can you apply here to generate hard if not the hardest maze? What kind of shortcomings or advantages do you anticipate from your approach?
- Try to find the hardest mazes for the following algorithms using the paired metric:
  - DFS with Maximal Shortest Path
  - DFS with Maximal Fringe Size
  - A^*-Manhattan with Maximal Nodes Expanded
  - A^*-Manhattan with Maximal Fringe Size
- Do your results agree with your intuition?

4 Thinning A^*

Consider an alternate approach to A^* in the following way: For a given maze, strip out some fraction of the obstructions, and use one of the previous algorithms to solve this 'simpler' maze - the shortest path on this simpler maze then becomes a lower bound on the length of the shortest path in the original maze; the solution to the simpler maze acts as a heuristic that you can use for solving the harder maze.

Suppose q represents the fraction of obstacles removed when simplifying the maze. Note that when $q = 1$, the simplified maze is empty, and the 'solution' is just the manhattan distance. When $q = 0$, the maze is not simplified at all and solving the 'simple' maze is as hard as solving the original maze.

Is this a viable strategy? Is there a value of q where solving the thinned maze first as a heuristic for solving the original maze actually makes solving the maze easier? Note that for fair comparison, you must include the cost of computing the heuristic. Be thorough, and justify yourself.

5 What if the maze were on fire? (Hard, Bonus)

All solution strategies discussed so far are in some sense 'static'. The solver has the map of the maze, spends some computational cycles determining the best path to take, and then that path can be implemented, for instance by a robot actually traveling through the maze. But what if the maze were changing as you traveled through it? You might be able to solve the 'original' maze, but as you start to actually follow the solution path, the maze may change and that solution may no longer be valid.

Consider the following model of the maze being on fire: any cell in the maze is either 'open', 'blocked', or 'on fire'. Starting out, the upper right corner of the maze is on fire. You can move between open cells or choose to stay in place, once per time step. You cannot move into cells that are on fire, and if your cell catches on fire you die. But each time-step, the fire may spread, according to the following rules:

- If a free cell has no burning neighbors, it will still be free in the next time step.
- If a free cell has k burning neighbors, it will be on fire in the next time step with probability $1 - (1/2)^k$.

How could you start to think about this problem? How does this time-evolving environment change your previous maze-solving strategies? See if you can design an algorithm for solving a maze that is on fire.