CS 520 : Project 1 - Voyage Into the Unknown   16:198:520

Due: 9/28 by 11:55 PM, Submitted via Canvas

This project is intended as an exploration of a variant of the A* search algorithm covered in class, in the traditional application of Robotic Path Planning. In particular, we are going to look at a situation where the environment is not fully known in advance, and must be learned by the agent as it moves through it.

A **gridworld** is a discretization of terrain into square cells that are either unblocked (traversable) or blocked. Consider the following problem: an agent in a gridworld has to move from its current cell (S) to a goal cell, the location of a stationary target. The layout of the gridworld (what cells are blocked or unblocked) is unknown. These kinds of challenges arise frequently in robotics, where a mobile platform equipped with sensors builds a map of the world as it traverses an unknown environment.
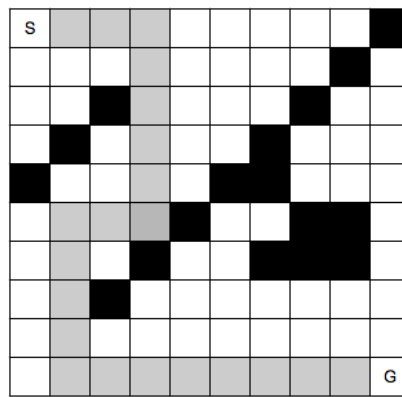

Figure 1: A successful path in a gridworld.

Assume that the initial cell of the agent is unblocked. The agent can move from its current cell in the four main compass directions (east, south, west, north) to any adjacent cell, as long as that cell is unblocked and still part of the gridworld. All moves take one timestep for the agent, and thus have cost 1. (We can consider this in terms of energy consumption for the agent, for instance.) The agent always knows which (unblocked) cell it is currently in, and which (unblocked) cell the target is in. The agent knows that blocked cells remain blocked and unblocked cells remain unblocked but **does not know initially which cells are blocked**. However, it can observe or sense the status of some of its surrounding cells (corresponding to its field of view) and remember this information for future use. By exploring the maze, collecting information on blockages, and integrating that into the whole, the agent must reach the target as efficiently as possible.

We may structure a basic agent in the following way: the agent assumes that all cells are unblocked, until it observes them to be blocked, and all path planning is done with the current state of knowledge of the gridworld under this *freespace assumption*. In other words, it moves according to the following strategy:

- Based on its current knowledge of the environment, it plans a path from its current position to the goal.

- This path should be the shortest (presumed) unblocked path available.

- The agent attempts to follow this path plan, observing cells in its field of view as it moves.

- Observed blocked cells or unblocked cells are remembered, updating the agent's knowledge of the environment.

- If the agent discovers a block in its planned path, it re-plans, based on its current knowledge of the environment.

- The cycle repeats until the agent either a) reaches the target or b) determines that there is no unblocked path to the target.

- **Question 1:** Why does re-planning only occur when blocks are discovered on the current path? Why not whenever knowledge of the environment is updated?

- **Question 2:** Will the agent ever get stuck in a solvable maze? Why or why not?

- **Question 3:** Once the agent reaches the target, consider re-solving the now discovered gridworld for the shortest path (eliminating any backtracking that may have occurred). Will this be an optimal path in the complete gridworld? Argue for, or give a counter example.

---

1   Implementation

Because this kind of strategy depends on being able to do a lot of searches, it is important these searches be as fast as possible. To that end, we are going to implement **Repeated Forward A***, where each forward path planning step is done using the A* search algorithm. Recall from class that as A* runs, it maintains four values for each cell n *that it encounters*:

- $g(n)$ - this represents the length of the shortest path discovered from the initial search point to cell n so far.

- $h(n)$ - the heuristic value, estimating the distance from the cell n to the goal node.

- $f(n)$ - $f(n)$ is defined to be $g(n) + h(n)$, which estimates the distance from the initial search node to the final goal node *through cell n*

- parent(n) - a pointer to the previous node along the shortest path to n.

A* maintains its fringe as a *priority queue* (initially containing the initial search point). The priority of the fringe is given by the value of f for every node in the fringe. Under some assumptions on h, the first time the goal node comes off the fringe represents the discovery of a shortest path from start to goal node. Until then, A* removes the cell n with the smallest value of f and expands it:

- Generate the children of n (neighbors believed or known to be unoccupied).

- The successors of n are the children n' that are newly discovered, or $g(n') > g(n) + 1$.

- For each successor n', re-set $g(n') = g(n) + 1$, representing the newly discovered shortest path from the start node to n'.

- For each successor n', set parent$(n') = n$.

- For each successor n', insert n' into the fringe at priority $f(n') = g(n') + h(n')$, or update its priority if it is already in the queue.

The above is repeated until termination, representing either a) an exhaustion of all possible paths with no success, or b) successfully identifying a minimal path from start to finish. Afterwards, it follows the parents from the goal node to the initial node to identify a shortest path (in reverse).

Note: g may be taken to be infinite for any node that has not yet been discovered.

Repeated Forward A* moves the agent along the identified path, until it either successfully reaches the goal or it encounters an obstacle along its planned path. In the first case, the agent has reached the target; in the second case, A* is re-called using the agents current position and any updated information about the location of obstacles. This is repeated until one of the two termination conditions for this strategy is met.

2 Execution, Analysis, and Report

In order to analyze algorithms for this problem, we need to generate consistent environments for comparison. Do the following: for a **dim** by **dim** array, let each cell independently be blocked with probability p, and empty with probability $1 - p$. Exclude the upper left corner (chosen to be the start position) and the lower right corner (chosen to be the end position) from being blocked. Write a function to generate random gridworlds in this way for a given value of p ($0 < p < 1$).

> **Note:** Provide in an appendix (of max. 2 pages), the code for the main functions you have implemented for Repeated A*. Please comment your code carefully to describe what your data structures and methods represent. In addition, each block/module of code should be annotated with a description of what that block does and should directly correspond to the steps in the algorithm.

> - **Question 4: Solvability** A gridworld is solvable if it has a clear path from start to goal nodes. How does solvability depend on p? Given **dim** $= 101$, how does solvability depend on p? For a range of p values, estimate the probability that a maze will be solvable by generating multiple environments and checking them for solvability. 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. Is A* the best search algorithm to use here, to test for solvability? Note for this problem you may assume that the entire gridworld is known, and hence only needs to be searched once each.
>
> - **Question 5: Heuristics** (a) Among environments that are solvable, is one heuristic uniformly better than the other for running A^*? Consider the following heuristics:
>
>   - **Euclidean Distance**
>
> $$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{1}$$
>
>   - **Manhattan Distance**
>
> $$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|. \tag{2}$$
>
>   - **Chebyshev Distance**
>
> $$d((x_1, y_1), (x_2, y_2)) = \max(|x_1 - x_2|, |y_1 - y_2|). \tag{3}$$
>
>   How can they be compared? Plot the relevant data and justify your conclusions. Again, you may take each gridworld as known, and thus only search once.

In the next questions, the blocks of gridworld are taken to be *unknown*, but as the agent moves through the environment it may observe the cells in its field of view and record whether or not they are blocked or open. You may

discard unsolvable gridworlds and re-generate new ones when collecting data.

- **Question 6: Performance** Taking **dim** = 101, for a range of density p values from 0 to $\min(p_0, 0.33)$, and the heuristic chosen as best in **Q5**, repeatedly generate gridworlds and solve them using Repeated Forward A*. Use as the field of view each immediately adjacent cell in the compass directions. Generate plots of the following data:

  - Density vs Average Trajectory Length

  - Density vs Average (Length of Trajectory / Length of Shortest Path in Final Discovered Gridworld)

  - Density vs Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld)

  - Density vs Average Number of Cells Processed by Repeated A*

  Discuss your results. Are they as you expected? Explain.

- **Question 7: Performance Part 2** Generate and analyze the same data as in **Q6**, except using only the cell in the direction of attempted motion as the field of view. In other words, the agent may attempt to move in a given direction, and only discovers obstacles by bumping into them. How does the reduced field of view impact the performance of the algorithm?

The final questions asks you to be somewhat speculative and explore (you may pick one, doing both will be viewed as extra credit). Both require code and data to answer completely:

- **Question 8: Improvements** Repeated A* may suffer in that the best place to re-start A* from may not be where you currently are - for instance if you are at the dead end of a long hallway, you can save some effort by backtracking to the end of the hallway (recycling information you already have) before restarting the A* search. By changing where you restart the search process, can you cut down the overall runtime? What effect does this have on the overall trajectory (given that you have to travel between the current position and the new initial search position)?

- **Question 9: Heuristics** A* can frequently be sped up by the use of inadmissible heuristics - for instance weighted heuristics or combinations of heuristics. These can cut down on runtime potentially at the cost of path length. Can this be applied here? What is the effect of weighted heuristics on runtime and overall trajectory? Try to reduce the runtime as much as possible without too much cost to trajectory length.

Extra Credit: Repeat **Q6, Q7**, using Repeated BFS instead of Repeated A*. Compare the two approaches. Is Repeated BFS ever preferable? Why or why not? Be as thorough and as explicit as possible.