CS 440: Mini-Project 1 - Search

01:198:440

Due July 11, 2019 at 11:55pm on Gradescope (PDF) and Sakai (code)

This project is intended as an exploration of variants of the A* search algorithms covered in class, in the traditional application of robotic path planning.

1 Setup

Consider the following problem: an agent in a gridworld has to move from its current cell to the given cell of a stationary target, where the gridworld is not fully known. The gridworld in this case is a discretization of terrain into square cells that are either unblocked (traversible) or blocked. 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.

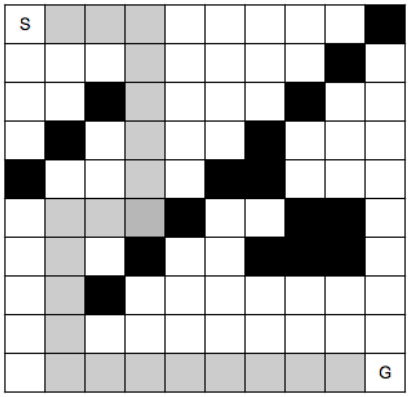

Figure 1: A successful path in a gridworld.

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 and north) to any adjacent cell, as long as that cell is unblocked and still part of the gridworld. All moves take one time step for the agent and thus have cost one. The agent always knows which (unblocked) cell it is 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 always observe the blockage status of its four adjacent cells, which corresponds to its field of view, and remember this information for future use. The objective of the agent is to reach the target as effectively as possible.

A common-sense and tractable movement strategy for the agent is the following: The agent assumes that cells are unblocked unless it has already observed them to be blocked and uses search with the "freespace assumption". In other words, it moves along a path that satisfies the following three properties:

1. It is a path from the current cell of the agent to the target.
2. It is a path that the agent does not know to be blocked and thus assumes to be unblocked, i.e., a presumed unblocked path.
3. It is a shortest such path.

Whenever the agent observes additional blocked cells *while it follows its current path*, it remembers this information for future use. If such cells block its current path, then its current path might no longer be a "shortest presumed-unblocked path" from the current cell of the agent to the target. Then, the agent stops moving along its current path, searches for another "shortest presumed-unblocked path" from its current cell to the target, taking into account the blocked cells that it knows about, and then moves along this path. The cycle stops when the agent:

- either reaches the target or
- determines that it cannot reach the target because there is no presumed-unblocked path from its current cell to the target and it is thus separated from the target by blocked cells.

In the former case, the agent reports that it reached the target. In the latter case, it reports that it cannot reach the target. This movement strategy has two desirable properties:

1. The agent is guaranteed to reach the target if it is not separated from it by blocked cells.
2. The trajectory is provably short.
3. The trajectory is believable since the movement of the agent is directed toward the target and takes the blockage status of all observed cells into account but not the blockage status of any unobserved cell.

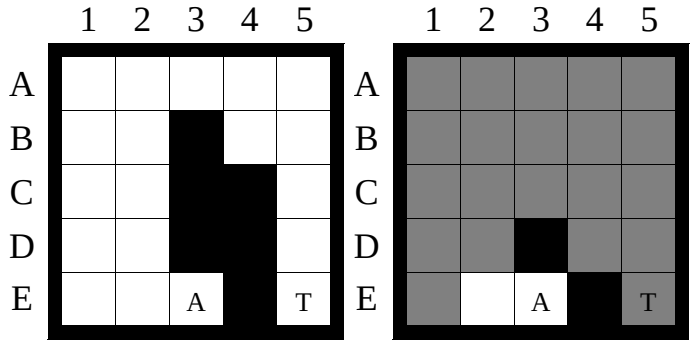

Figure 2: First Example Search Problem (left) and Initial Knowledge of the Agent (right)

As an example, consider the gridworld of size 5×5 shown in Figure 2 (left). Black cells are blocked, and white cells are unblocked. The initial cell of the agent is marked A, and the target is marked T. The initial knowledge of the agent about blocked cells is shown in Figure 2 (right). The agent knows black cells to be blocked and white cells to be unblocked. It does not know whether grey cells are blocked or unblocked. The trajectory of the agent is shown in Figure 3. The left figures show the actual gridworld. The center figures show the knowledge of the agent about blocked cells. The right figures again show the knowledge of the agent about blocked cells, except that all cells for which it does not know whether they are blocked or unblocked are now shown in white since the agent assumes that they are unblocked. The arrows show the "shortest presumed-unblocked paths" that the agent attempts to follow. The agent needs to find another "shortest presumed-unblocked path" from its current cell to the target whenever it observes its current path to be blocked. The agent finds such a path by finding a shortest path from its current cell to the target in the right figure. The resulting paths are shown in bold directly after they were computed. For example, at time step 1, the agent searches for a "shortest presumed-unblocked path" and then moves along it for three moves (first search). At time step 4, the agent searches for another "shortest presumed-unblocked path" since it observed its current path to be blocked and then moves along it for one move (second search). At time step 5, the agent searches for another "shortest presumed-unblocked path" (third search), and so on. When the agent reaches the target it has observed the blockage status of every cell although this is not the case in general.

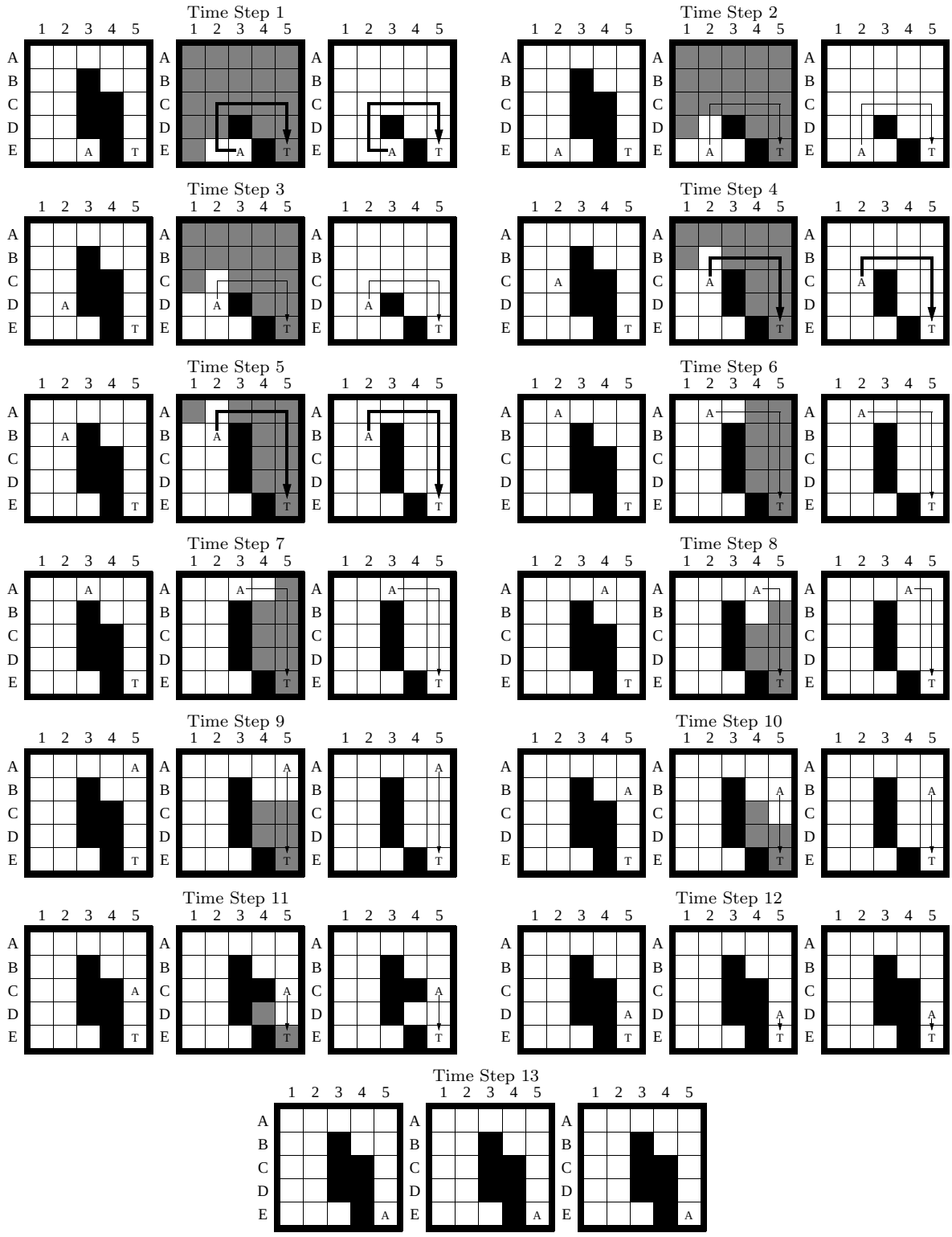

Figure 3: Trajectory of the Agent for the First Example Search Problem

2 Modeling and Solving the Problem

The state space of the search problem arising is simple: The states correspond to the cells, and the actions allow the agent to move from cell to cell. Initially, all action costs are one. When the agent observes a blocked cell for the first time, it increases the action costs of all actions that enter or leave the corresponding state from one to infinity or, alternatively, removes the actions. A shortest path in this state space then is a "shortest presumed-unblocked path" in the gridworld.

Thus, the agent needs to search in state spaces in which action costs can increase or, alternatively, actions can be removed. The agent searches for a shortest path in the state space whenever the length of its current path increases (to infinity). Thus, the agent has to search repeatedly until it reaches the target. It is therefore important for the searches to be as fast as possible.

In the following, we use A* to determine the shortest paths, resulting in Repeated A*. A* can search either from the current cell of the agent toward the target (= forward), resulting in Repeated Forward A*, or from the target toward the current cell of the agent (= backward), resulting in Repeated Backward A*.

3 Repeated Forward A*

A brief sketch of Repeated Forward A* follows. S denotes the finite set of states. $s_{agent} \in S$ denotes the start state of the A* search (which is the current state of the agent), and $s_{target} \in S$ denotes the goal state of the A* search (which is the state of the target). $A(s)$ denotes the finite set of actions that can be executed in state $s \in S$. $c(s, a) > 0$ denotes the action cost of executing action $a \in A(s)$ in state $s \in S$, and $succ(s, a) \in S$ denotes the resulting successor state. A* maintains five values for all states s that it encounters:

1. a g-value $g(s)$ (which is infinity initially), which is the length of the shortest path from the start state to state s found by the A* search and thus an upper bound on the distance from the start state to state s;
2. an h-value (= heuristic) $h(s)$ (which is user-supplied and does not change), which estimates the goal distance of state s (= the distance from state s to the goal state);
3. an f-value $f(s) := g(s) + h(s)$, which estimates the distance from the start state via state s to the goal state;
4. a tree-pointer $tree(s)$ (which is undefined initially), which is necessary to identify a shortest path after the A* search;
5. and a search-value $search(s)$, which is described below.

A* maintains an open list (a priority queue which contains only the start state initially). A* identifies a state s with the smallest f-value in the open list. If the f-value of state s is no smaller than the g-value of the goal state, then the A* search is over. Otherwise, A* removes state s from the open list and expands it. We say that it expands state s when it inserts state s into the closed list (a set which is empty initially) and then performs the following operations for all actions that can be executed in state s and result in a successor state whose g-value is larger than the g-value of state s plus the action cost.

1. Set the g-value of the successor state to the g-value of state s plus the action cost.
2. Set the tree-pointer of the successor state to (point to) state s

3. Insert the successor state into the open list or, if it was there already, changes its priority. (We say that it generates a state when it inserts the state for the first time into the open list.)

4. Repeat Steps 1-3.

Repeated Forward A* itself executes the above described A* search. Afterwards, it follows the tree-pointers from the goal state to the start state to identify a shortest path from the start state to the goal state in reverse. Repeated Forward A* then makes the agent move along this path **until it reaches the target or action costs on the path increase**. In the first case, the agent has reached the target. In the second case, the current path might no longer be a shortest path from the current state of the agent to the state of the target. Repeated Forward A* then updates the current state of the agent and repeats the procedure.

Repeated Forward A* does not initialize all g-values up front but uses the variables *counter* and $search(s)$ to decide when to initialize them. The value of *counter* is x during the x^{th} A* search. The value of $search(s)$ is x if state s was generated last by the x^{th} A* search (or is the goal state). The g-value of the goal state is initialized at the beginning of an A* search since it is needed to test whether the A* search should terminate. The g-values of all other states are initialized directly before they might be inserted into the open list [Lines 7 and 20] provided that state s has not yet been generated by the current A* search ($search(s) < counter$). The only initialization that Repeated Forward A* performs up front is to initialize $search(s)$ to zero for all states s, which is typically automatically done when the memory is allocated.

4 Implementation Details

Your implementation of Repeated A* should use a binary heap to implement the open list. You will use the following heuristics:

- **Euclidean Distance**
$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \quad (1)$$
- **Manhattan Distance**
$$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|. \quad (2)$$
- **Chebyshev Distance**
$$d((x_1, y_1), (x_2, y_2)) = \max(|x_1 - x_2|, |y_1 - y_2|). \quad (3)$$

Your implementation needs to be efficient in terms of processor cycles and memory usage. Make sure that you never iterate over all cells except to initialize them once before the first A* search since your program might be used in large gridworlds. For example, do not determine which cells are in the closed list by iterating over all cells (represent the closed list explicitly instead). Make sure that the agent indeed always follows a "shortest presumed-unblocked path" if one exists and that it reports that it cannot reach the target otherwise. Make sure that each A* search never expands a cell that it has already expanded (= is in the closed list).

5 Analysis

Part 0 - Setup your environments: In order to properly compare your algorithms, they need to be run multiple times (at least 50) over a variety of gridworld environments. Generate your gridworlds in the following way: for a given dimension **dim** construct a **dim** $\times$ **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 gridworlds for a given **dim** and p.

For **Part 1** and **Part 2**, you may assume that the environment is fully known to the agent ahead of time. Thus, you will only need to run your A* Search **once** to identify a shortest path if it exists.

Part 1 - Gridworld solvability: (a) We will call a gridworld *solvable* if a shortest path exists between the start and the goal state. Given **dim**, 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.

(b) Instead of running A* search here, is there a better algorithm that can be used to determine the solvability of an environment quickly? Discuss.

Part 2 - Heuristics: (a) Among environments that are solvable, is one heuristic uniformly better than the other for running A^*? How can they be compared? Plot the relevant data and justify your conclusions.

(b) Pritish claims that if two consistent heuristics $h_1(s)$ and $h_2(s)$ are available for the same problem, then at every state s, either $h(s) = \max(h_1(s), h_2(s))$ or $h(s) = \min(h_1(s), h_2(s))$ will be a consistent heuristic. Is he correct? Argue mathematically or provide a counter-example.

For **Part 3** and **Part 4**, the agent **does not know initially which cells are blocked**. Thus, you must run Repeated A* Search to identify a shortest path if it exists. Report your results on environments where **dim** $= 101$, and $p = \min(p_0, 0.33)$ (using p_0 identified in **Part 1**). Use the heuristic which you identified as best in **Part 2** .

Part 3 - Effect of ties: (a) 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 what you observed and give a reason for the observation.

(b) Aravind claims that the priority of a cell can be calculated as $c \times f(s) - g(s)$, where c is a constant. For what values of c will this priority favor larger g-values? For what values of c will it favor lower g-values? Try to be as specific as possible.

Part 4 - Forward vs. Backward A*: (a) Implement and compare Repeated Forward A* and Repeated Backward A* with respect to their runtime or, equivalently, number of expanded cells. 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 using the best strategy you identified in **Part 3** and remaining ties in an identical way, for example randomly.

(b) On the search problem illustrated in Fig 2, illustrate (either using ASCII printouts or any other visualization tool), the operation of Repeated Backward A*. The illustration must look similar to Fig 3, i.e. you must show at each time step:

- the current location of the agent,
- the agent's knowledge about blocked and unknown cells, and
- the trajectory (path) computed by the agent at that timestep.

(c) Neelesh remarks that the path found by Repeated A* is a *shortest* path, but not necessarily the *optimal path* from the start to the goal. Provide an example to prove him right.

Extra Credit. Will be uploaded by next week.