COMP8620: Assignment 1
Trajectory Replanning for Computer Games

August 6, 2008

1 Introduction

[1]Source: Sven Koenig and William Yeoh, *A Project on Fast Trajectory Replanning for Computer Games for "Introduction to Artificial Intelligence" Classes*, Technical Report, Department of Computer Science, University of Southern California, Los Angeles (California, USA).

Figure 1: Total Annihilation by Cavedog

Consider characters in real-time computer games, such as Total Annihilation shown in Figure 1. To make them easy to control, the player can click on known or unknown terrain, and the game characters then move autonomously to the location that the player clicked on. They always observe the terrain within their limited field of view and then remember it for future use but do not know the terrain initially (due to "fog of war").

We study a variant of this search problem in a gridworld where an agent has to move from its current cell to the given cell of a non-moving target. Gridworlds are commonly used in real-time computer games. They are discretizations of terrain into square cells that are either blocked or unblocked. 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,

[1]*

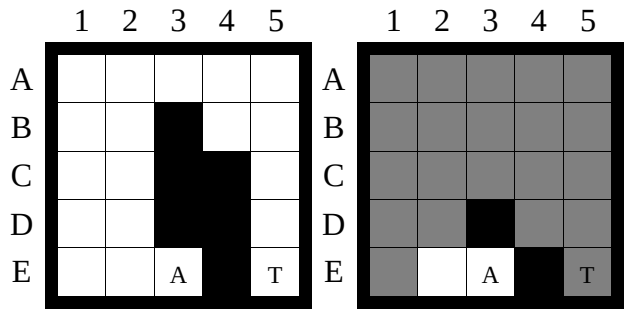

Figure 2: First Example Search Problem (left) and Initial Knowledge of the Agent (right)

it can always observe the blockage status of its four adjacent cells (which is its field of view) and remember this information for future use. The objective of the agent is to reach the target.

A common-sense and tractable movement strategy for the agent is the following one: The agent assumes that cells are unblocked unless it has already observed them to be blocked and uses search with the freespace assumption, that is, always moves along a path that satisfies 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 (= 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 (but not necessarily optimal). 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.

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 centre 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

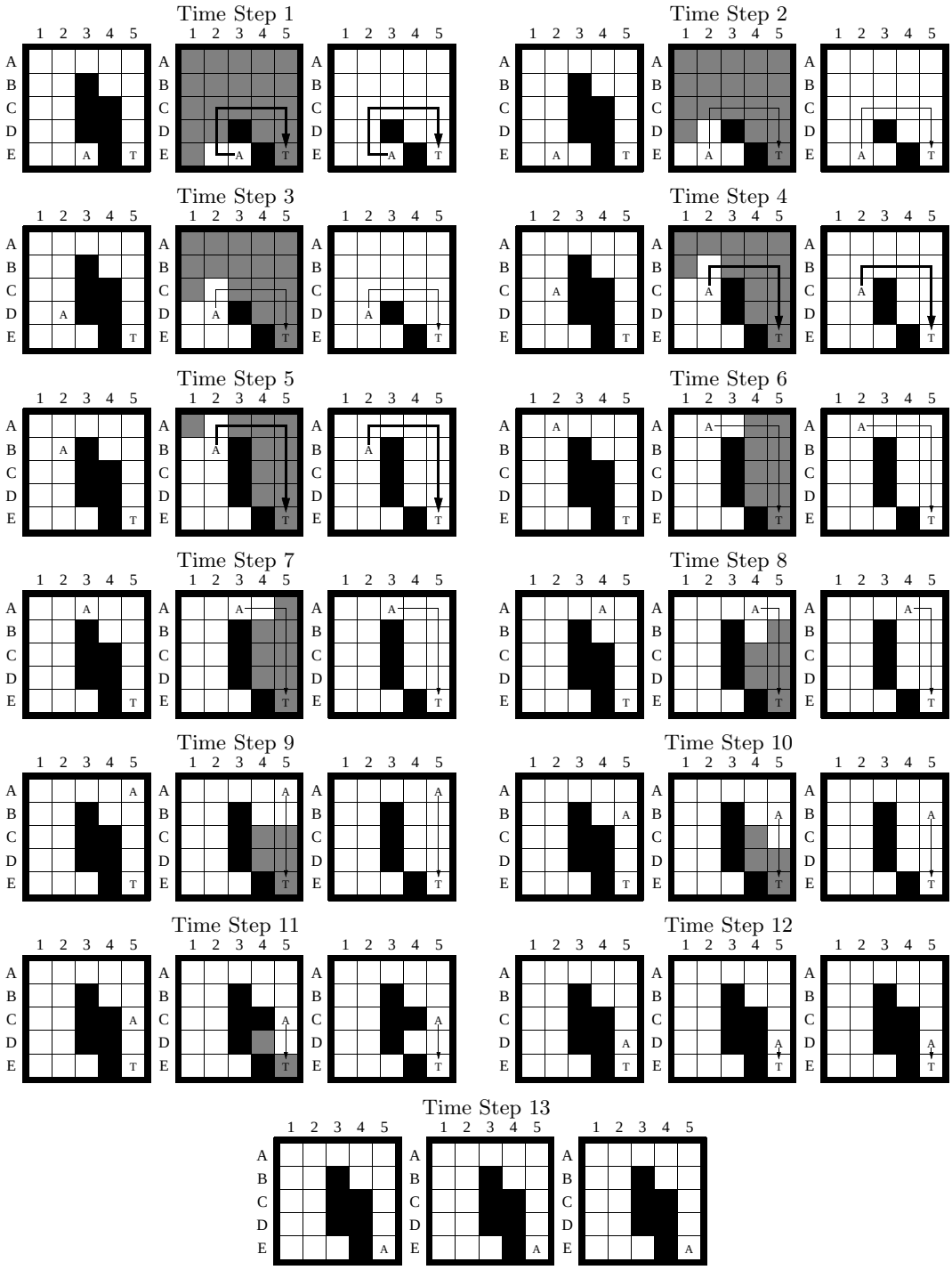

Figure 3: Trajectory of the Agent for the First Example Search Problem

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.

1.1 Modelling and Solving the Problem

The state spaces of the search problems are 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 (of the typically many agents in real-time computer games) has to search repeatedly until it reaches the target. It is therefore important for the searches to be as fast as possible to ensure that the agent responds quickly and moves smoothly even on computers with slow processors in situations where the other components of real-time computer games (such as the graphics and user interface) use most of the available processor cycles. Moore's law does not solve this problem since the number of game characters of real-time computer games will likely grow at least as quickly as the processor speed. 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*.

1.2 Repeated Forward A*

The pseudocode of Repeated Forward A* is shown in Figure 4. (The minimum over an empty set is infinity on Line 2.) It performs the A* searches in ComputePath(). A* is described in your textbook and therefore only briefly discussed in the following, using the following notation that can be used to describe general search problems rather than only search problems in gridworlds: S denotes the finite set of states. $s_{\text{start}} \in S$ denotes the start state of the A* search (which is the current state of the agent), and $s_{\text{goal}} \in S$ 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: 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; 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); an f-value $f(s) := g(s) + h(s)$, which estimates the distance from the start state via state s to the goal state; a tree-pointer $tree(s)$ (which is undefined initially), which is necessary to identify a shortest path after the A* search; 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 [Line 2]. 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 [Line 3] 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) [Line 4] 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 [Lines 5-13]: First,

```
1  procedure ComputePath()
2  while g(s_goal) > min_{s' ∈ OPEN}(g(s') + h(s'))
3    remove a state s with the smallest f-value g(s) + h(s) from OPEN;
4    CLOSED := CLOSED ∪ {s};
5    for all actions a ∈ A(s)
6      if search(succ(s, a)) < counter
7        g(succ(s, a)) := ∞;
8        search(succ(s, a)) := counter;
9      if g(succ(s, a)) > g(s) + c(s, a)
10       g(succ(s, a)) := g(s) + c(s, a);
11       tree(succ(s, a)) := s;
12       if succ(s, a) is in OPEN then remove it from OPEN;
13       insert succ(s, a) into OPEN with f-value g(succ(s, a)) + h(succ(s, a));
14 procedure Main()
15 counter := 0;
16 for all states s ∈ S
17   search(s) := 0;
18 while s_start ≠ s_goal
19   counter := counter + 1;
20   g(s_start) := 0;
21   search(s_start) := counter;
22   g(s_goal) := ∞;
23   search(s_goal) := counter;
24   OPEN := CLOSED := ∅;
25   insert s_start into OPEN with f-value g(s_start) + h(s_start);
26   ComputePath();
27   if OPEN = ∅
28     print "I cannot reach the target.";
29     stop;
30   follow the tree-pointers from s_goal to s_start and then move the agent along the resulting path
       from s_start to s_goal until it reaches s_goal or one or more action costs on the path increase;
31   set s_start to the current state of the agent (if it moved);
32   update the increased action costs (if any);
33 print "I reached the target.";
34 stop;
```

Figure 4: Pseudocode of Repeated Forward A*

it sets the g-value of the successor state to the g-value of state s plus the action cost [Line 10]. Second, it sets the tree-pointer of the successor state to (point to) state s [Line 11]. Finally, it inserts the successor state into the open list or, if it was there already, changes its priority [Line 12-13]. (We say that it generates a state when it inserts the state for the first time into the open list.) It then repeats the procedure.

Remember that h-values $h(s)$ are consistent (= monotone) iff they satisfy the triangle inequalities $h(s_{\text{goal}}) = 0$ and $h(s) \leq c(s, a) + h(succ(s, a))$ for all states s with $s \neq s_{\text{goal}}$ and all actions a that can be executed in state s. Consistent h-values are admissible (= do not overestimate the goal distances). A* searches with consistent h-values have the following properties. Let $g(s)$ and $f(s)$ denote the g-values and f-values, respectively, after the A* search: First, A* searches expand all states at most once each. Second, the g-values of all expanded states and the goal state after the A* search are equal to the distances from start state to these states. Following the tree-pointers from these states to the start state identifies shortest paths from the start state to these states in reverse. Third, the f-values of the series of expanded states over time are monotonically nondecreasing. Thus, it holds that $f(s) \leq f(s_{\text{goal}}) = g(s_{\text{goal}})$ for all states s that were expanded by the A* search (that is, all states in the closed list) and $g(s_{\text{goal}}) = f(s_{\text{goal}}) \leq f(s)$ for all states s that were generated by the A* search but remained unexpanded (that is, all states in the open list). Fourth, an A* search with consistent h-values $h_1(s)$ expands no more states than an otherwise identical A* search with consistent h-values $h_2(s)$ for the same search problem (except possibly for some states whose f-values are identical to the f-value of the goal state) if $h_1(s) \geq h_2(s)$ for all states s.

Repeated Forward A* itself executes ComputePath() to perform an 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 [Line 30]. 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 initialise all g-values up front but uses the variables *counter* and $search(s)$ to decide when to initialise them. The value of *counter* is x during the xth A* search, that is, the xth execution of ComputePath(). The value of $search(s)$ is x if state s was generated last by the xth A* search (or is the goal state). The g-value of the goal state is initialised at the beginning of an A* search [Line 22] since it is needed to test whether the A* search should terminate [Line 2]. The g-values of all other states are initialised 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 initialisation that Repeated Forward A* performs up front is to initialise $search(s)$ to zero for all states s, which is typically automatically done when the memory is allocated [Lines 16-17].

1.3 Implementation Details

Your version of Repeated A* should use the Manhattan distances as h-values. The Manhattan distance of a cell is the sum of the absolute difference of the x coordinates and the absolute difference of the y coordinates of the cell and the cell of the target. The reason for using the Manhattan distances is that they are consistent in gridworlds in which the agent can move only in the four main compass directions.

Your implementation of Repeated A* needs to be efficient in terms of processor cycles and memory usage since game companies place limitations on the resources that trajectory planning has available. Thus, it is important that you think carefully about your implementation rather than use the pseudocode from Figure 4 blindly since it is not optimised. (For example, the closed list in the pseudocode is shown only to allow us to refer to it later when explaining Adaptive A*.)

Do not use code written by others but test your implementations carefully. For example, 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).

We now discuss the example search problem from Figure 2 (left) to give you data that you can use to test your implementations. Figure 5 shows the first two searches of Repeated Forward A* for the example search problem from Figure 2 (left). Figure 5 (left) shows the first A* search, and Figure 5 (right) shows the second A* search. In our example problems, A* breaks ties among cells with the same f-value in favour of cells with larger g-values and remaining ties in an identical way. All cells have their user-supplied h-values in the lower left corner, namely the Manhattan distances. Generated cells (= cells that are or were in the open list) also have their g-values in the upper left corner and their f-values in the upper right corner. Expanded cells (= cells that are in the closed list) are shown in grey. (The cell of the target does not count as expanded since the A* search stops immediately before expanding it.) The arrows represent the tree-pointers, which are necessary to identify a shortest path after the A* search. Similarly,

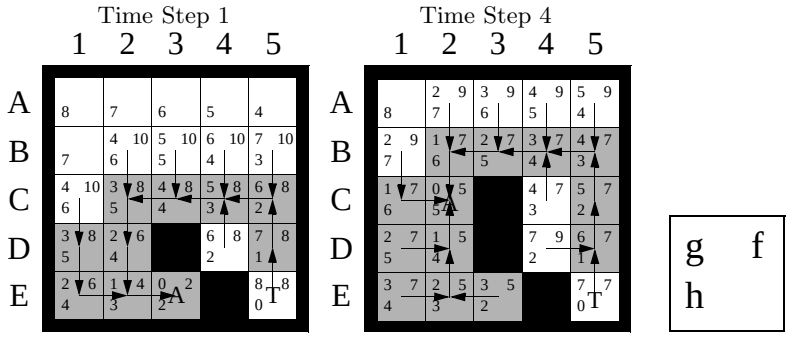

Figure 5: Repeated Forward A*

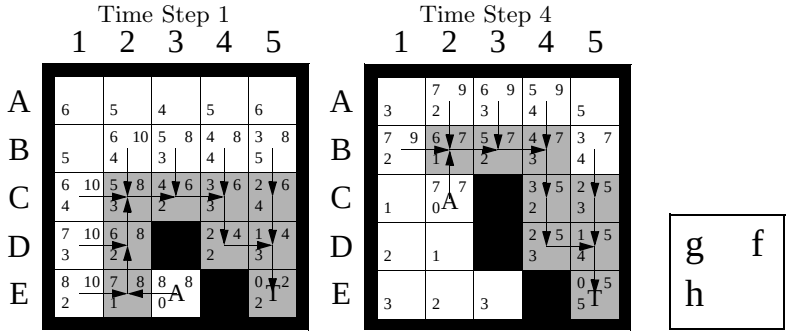

Figure 6: Repeated Backward A*

Figure 6 shows the first two searches of Repeated Backward A*, which searches from the target to the current cell of the agent.

1.4 Improving Repeated Forward A*

Adaptive A* uses A* searches to repeatedly find shortest paths in state spaces with possibly different start states but the same goal state where action costs can increase (but not decrease) by arbitrary amounts between A* searches.[2] It uses its experience with earlier searches in the sequence to speed up the current A* search and run faster than Repeated Forward A*. It first finds the shortest path from the current start state to the goal state according to the current action costs. It then updates the h-values of the states that were expanded by this search to make them larger and thus future A* searches more focused. Adaptive A* searches from the current state of the agent to the target since the h-values estimate the goal distances with respect to a given goal state. Thus, the goal state needs to remain unchanged, and the state of the target remains unchanged while the current state of the agent changes. Adaptive A* can handle action costs that increase over time.

To understand the principle behind Adaptive A*, assume that the action costs remain unchanged to make the description simple. Assume that the h-values are consistent. Let $g(s)$ and $f(s)$ denote the g-values and f-values, respectively, after an A* search from the current state of the agent to the target. Let s denote any state expanded by the A* search. Then, $g(s)$ is the distance from the start state to state s since state s was expanded by the A* search. Similarly, $g(s_{\text{goal}})$

[2]Start state refers to the start of the search, and goal state refers to the goal of the search. The start state of the A* searches of Repeated Forward A*, for example, is the current state of the agent. The start state of the A* searches of Repeated Backward A*, on the other hand, is the state of the target.

is the distance from the start state to the goal state. Thus, it holds that $g(s_{\text{goal}}) = gd(s_{\text{start}})$, where $gd(s)$ is the goal distance of state s. Distances satisfy the triangle inequality:

$$\begin{aligned} gd(s_{\text{start}}) &\leq g(s) + gd(s) \\ gd(s_{\text{start}}) - g(s) &\leq gd(s) \\ g(s_{\text{goal}}) - g(s) &\leq gd(s). \end{aligned}$$

Thus, $g(s_{\text{goal}}) - g(s)$ is an admissible estimate of the goal distance of state s that can be calculated quickly. It can thus be used as a new admissible h-value of state s (which was probably first noticed by Robert Holte). Adaptive A* therefore updates the h-values by assigning

$$h(s) := g(s_{\text{goal}}) - g(s)$$

for all states s expanded by the A* search. Let $h_{\text{new}}(s)$ denote the h-values after the updates.

The h-values $h_{\text{new}}(s)$ have several advantages. They are not only admissible but also consistent. The next A* search with the h-values $h_{\text{new}}(s)$ thus continues to find shortest paths without expanding states that have already been expanded by the current A* search. Furthermore, it holds that

$$\begin{aligned} f(s) &\leq gd(s_{\text{start}}) \\ g(s) + h(s) &\leq g(s_{\text{goal}}) \\ h(s) &\leq g(s_{\text{goal}}) - g(s) \\ h(s) &\leq h_{\text{new}}(s) \end{aligned}$$

since state s was expanded by the current A* search. Thus, the h-values $h_{\text{new}}(s)$ of all expanded states s are no smaller than the immediately preceding h-values $h(s)$ and thus, by induction, also all previous h-values, including the user-supplied h-values. An A* search with consistent h-values $h_1(s)$ expands no more states than an otherwise identical A* search with consistent h-values $h_2(s)$ for the same search problem (except possibly for some states whose f-values are identical to the f-value of the goal state, a fact that we will ignore in the following) if $h_1(s) \geq h_2(s)$ for all states s. Consequently, the next A* search with the h-values $h_{\text{new}}(s)$ cannot expand more states than with any of the previous h-values, including the user-supplied h-values. It therefore cannot be slower (except possibly for the small amount of runtime needed by the bookkeeping and h-value update operations), but will often expand fewer states and thus be faster. You can read up on Adaptive A* in Koenig and Likhachev, Adaptive A* [Poster Abstract], Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 1311-1312, 2005.

Figure 7 shows the first two searches of Adaptive A* for the example search problem from Figure 2 (left). The number of cell expansions is smaller for Adaptive A* (20) than for Repeated Forward A* (23), demonstrating the advantage of Adaptive A* over Repeated Forward A*. Figure 7 (left) shows the first A* search of Adaptive A*. All cells have their h-values in the lower left corner. The goal distance of the current cell of the agent is eight. The lower right corners show the updated h-values after the h-values of all grey cells have been updated to eight minus their g-values, which makes it easy to compare them to the h-values before the

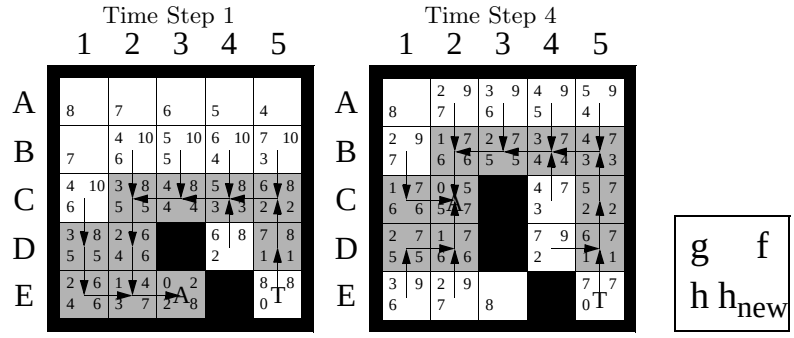

Figure 7: Adaptive A*

h-value update in the lower left corners. Cells D2, E1, E2 and E3 have larger h-values than their Manhattan distances to the target, that is, the user-supplied h-values. Figure 7 (right) shows the second A* search of Adaptive A*, where Adaptive A* expands three cells (namely, cells E1, E2 and E3) fewer than Repeated Forward A*, as shown in Figure 5 (right).

1.5 Experiments

Perform all experiments in the same 50 gridworlds of size 101×101. Generate their corridor structure with a depth-first search (with random tie breaking) and then make 100 randomly chosen blocked cells unblocked. See the program at pjk/teaching for details, which you are free to use. Note that sometimes more than one shortest presumed unblocked path exists. Different search algorithms might then move the agent along different shortest presumed unblocked paths (as seen in Figures 5 and 6), and the agent might observe different cells to be blocked. Then, the shortest presumed unblocked paths and the trajectories of the agent can start to diverge. This is okay since it is difficult to make all search algorithms find the same shortest presumed unblocked path in case there are several of them.

2 The Task

Implement Repeated Forward A*, Repeated Backward A* and Adaptive A* in Java.

Included in the release are 10 random mazes. The number on the first line is the size of the maze. There follows the maze with "X" denoting an obstacle, "A" denoting the start and "T" denoting the target.

Run your code on each of these mazes.

Write a report describing the main data structures in your code, and the results of your experiments. Also note the method used to break ties when the f $(= g + h)$ values for two nodes are the same. Please report

- The path length (number of steps required) between start and target
- The total number of nodes *expanded* (i.e sum over all iterations of the code)

for each of the methods, for each of the mazes.

At the end of solving the maze, have your code print a new copy of the maze in a format matching the input, but add an "O" marking the path between start and finish identified by your program.

Please make either a zip or tar bundle with

- your code
- example output for Adaptive A* only
- your report

and submit them through the usual mechanisms.