HW1 Solutions

Part 0: Set up the Environments

In this step, we generate the map with Depth-First Search (DFS) approach by using random tie breaking as required. A map is denoted by a two-dimensional array with a boolean value in each cell. True value means this cell is blocked while false means unblocked. In the beginning, the map is initialized as false in all cells. Another matrix of the same size as the map is used to record if a cell has been visited. We also use a stack to store the visited path. The stack is implemented using a vector. We first set its capacity to the size of the map. Once the stack is full, it is resized and capacity doubles.

Starting from a random cell, we randomly choose one of its unvisited neighbors and mark it as blocked with a probability ρ (in our experiment ρ is set to 0.3). Then we push it on the stack, and mark it as visited. Once a dead-end is encountered, we just keep popping cells from the stack until finding a cell that has unvisited neighbor(s). When the stack is empty, all cells have been visited. The pseudocode of the algorithm to generate the map is given in Algorithm 1.

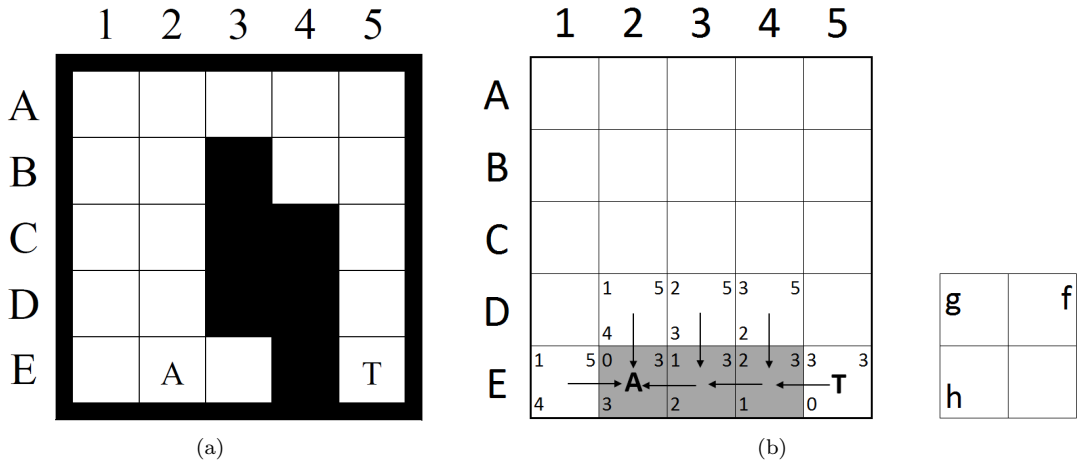
Algorithm 1 DFS with random tie breaking for map generation

Input: an initial map M={false}, an indicator matrix V={false} with the same size of M, an empty stack S, block generation probability ρ and a randomly selected start point P_0 in M.

```
S.push(P_0)
V(P_0) = true
while !S.empty() do
    P_curr=S.top()
    if there is at least one unvisited neighbor then
        randomly choose one neighbor P_next
        set M(P_next) = true with probability ρ
        V(P_next) = true
        if S is full then
            double the capacity of S
        end if
        S.push(P_next)
    else
        S.pop()
    end if
end while
return generated map M
```

Part 1: Understanding the methods

a) In the first search of repeated A^* search, the agent only knows that its neighbor cells (E1, D2 and E3 in figure 1(a)) are not blocked. The f-values of neighbor cells of start cell are $f(E1) = 5, f(D2) = 5, f(E3) = 3$. A^* algorithm always chooses the neighbor which has smallest f-value, thus it will expand cell E3. The final path found by the first search is E2→E3→E4→E5. Therefore, the agent will move to the east first.



(a) (b)

Figure 1: Second Example Search Problem

b1) **Give a convincing argument that the agent in finite gridworlds indeed either reaches the target or discovers that this is impossible in finite time.**

Case 1: the target is not separated by blocked cells. Each time the agent encounters a blocked cell, A^* search algorithm can find a path from the current cell to target cell due to the completeness of the algorithm. Because the gridworld is finite, it will reach the target finally.

Case 2: the target is separated by blocked cells. Each time the algorithm finds a path, the agent will at least move one step before it encounters another blocked cell, and knows more about the gridworld. Because the gridworld is finite, the agent will know the whole gridworld in finite time in the worst case. Then A^* search algorithm cannot find a path to the target, and the agent discovers that it is impossible to reach the target.

b2) **Prove that the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of unblocked cells squared.**

Case 1: the target is not separated by blocked cells. If there is one path to the target state, the path has at most $n-1$ states, where n is the number of unblocked cells. In the worst case, every time the agent moves, it encounters a block in the planned path, which means that the A^* is required to run $n-1$ times at most. In each A^* search, the planned path will not contain the same cell due to the optimality of A^* search. Therefore, the number of moving steps with one A^* is bounded by $n-1$. Thus the number of total moves is at most $(n-1)^2$, less than n^2.

Case 2: the target is separated by blocked cells. In this case, there is no path to the target state. Before determining that the target state cannot be reached, the agent should find all the unblocked states that the start state can reach, the number of which is bounded by $n-1$. The number of moves required to find a unblocked state is at most $n-2$. Obviously, $(n-1)(n-2) < n^2$. Therefore, the number of moves is bounded above by the number of unblocked states squared.

Part 2: The Effects of Ties

Although the answer is **open** to all possible outcomes, the mostly probable result you will encounter is: on average over all experimented maps, i.e. with statistical significance, breaking ties by choosing the candidate with a greater g-value expands less cells than the strategy of choosing the one with a smaller g-value.

We use the toy example given in the problem statement to explain the reason. Figure 2 gives the searching results of A^* in favor of larger and smaller g-values. The cells that have been expanded are marked as grey color. We expand the neighbors in the order of south, east, north and west when f-value and g-value are both equal. We can find that the A^* in favor of smaller g-values expand much more unnecessary cells. The reason lies in that larger g-values encourage A^* to go down a single path whereas smaller g-values will make A^* go back many times to expand the cells closer to the start cell, which costs many extra expansions. Therefore, the A^* in favor of large g-values is better.

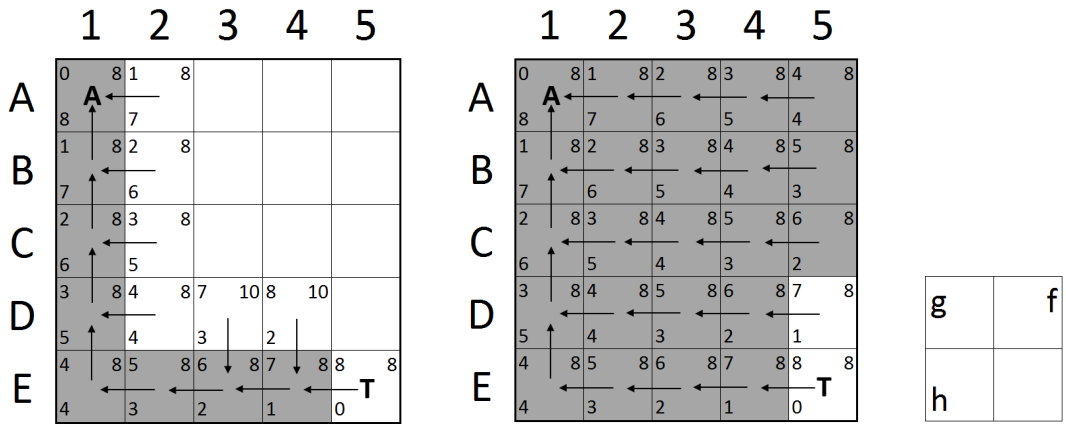


Figure 2: A toy example for the comparison of different tie-breaking method in A^* search: (a) expanded cells in favor of larger g-value, (b) expanded cells in favor of smaller g-values

Part 3: Forward vs. Backward

Although the answer is **open** to all possible outcomes, the mostly probable result you will encounter is: on average over all experimented maps, i.e. with statistical significance, the forward A^* algorithm expands less cells than the backward A^* algorithm.

During each A^*, the agent only knows some blocks lying in the area near the start cell and it assumes there is no blocks in other areas between the start and target cells. For backward A^*, before it reaches the blocks near the start cell, the f-values of cells in the open list are all equal to the Manhattan distance between the start and target states, which is also the smallest f-value during this search. It is very likely that the length of the path between the start and target is larger than their Manhattan distance, so when the f-value of a cell increases, it will go back to expand the cells with the smallest f-value in the open list and these cells will further expand to many more neighbors with the same f-values. The backward A^* will keep expanding all the states with the smallest f-value and then it expands states with larger f-values. In contrast, the forward A^* will probably increase its f-value at the first several movements and then maintain the same f-value till the target cell. When the f-value of current cell increases, there will be much fewer states with smaller f-values in the open list benefiting from the fact that only a few states are searched and blocks in the near area exclude many candidates to be searched. Thus, forward A^* will save many more expansions than backward A^*.

We use an example in figure 3 to better illustrate the reason. Figure 3(a) is the original test map, which only has blocked cells near the start cell. The blocks are black and the expanded states are grey. A and T are the start and target cells. Figure 3(b) is the expansion result of first backward A^* search, all cells have the same smallest f-value. Figure 3(c) is the second backward A^* search when it comes to expand the cell A5. At this time, its neighbor B5 has a f-value of 12, but it is not the smallest one in the open list. The cell B6 will be expanded next because it has the smallest f-value and largest g-value in the open list. After that, when it comes to expand the cell B4 (shown in figure 3(d)), the same thing happens and the cell C6 will be expanded next instead of C4. During this process, backward A^* will expand all cells after column 3 before it chooses to expand cells in column 2, which is much time-consuming. The final expansion result of the second backward A^* search is presented in figure 3(f). For comparison, we also present the result of forward A^* in figure 3(g) if in the same situation.

Part 4: Heuristics in the Adaptive A^*

a) **The project argues that ?the Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions.? Prove that this is indeed the case.**

Because $h(s)$ is defined as the Manhattan distance between state s and s_{goal}, if $s = s_{goal}$, then $h(s_{goal}) = |x_{s_{goal}} - x_{s_{goal}}| + |y_{s_{goal}} - y_{s_{goal}}| = 0$, where x_s and y_s are the indexes of rows and columns

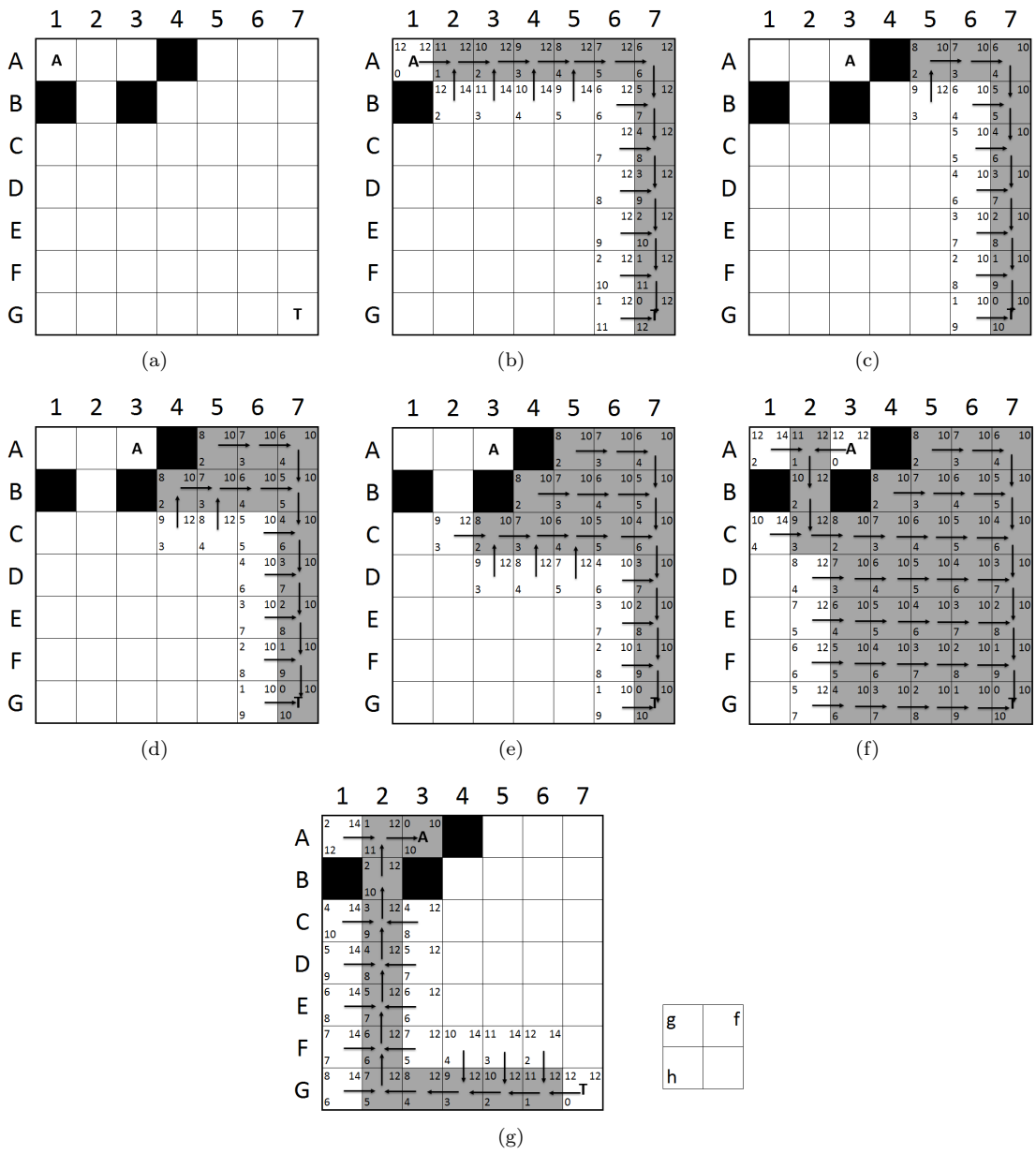


Figure 3: An example of how backward A^* works: (a) the original map; (b) first backward A^* search; (c)$\sim$(f) different steps in second backward A^* search; (g) forward A^* search for comparison.

for state s.

If $s \neq s_{goal}$, let $n = succ(s, a)$ for simplicity, then $h(s) - h(n) = |x_s - x_{s_{goal}}| + |y_s - y_{s_{goal}}| - |x_n - x_{s_{goal}}| - |y_n - y_{s_{goal}}|$. Since s and n are neighbors in grids and the agent can only move in four main compass directions, we have $x_s = x_n$ or $y_s = y_n$. If $x_s = x_n$, then $|y_s - y_n| = 1$. In this case $h(s) - h(n) = |y_s - y_{s_{goal}}| - |y_n - y_{s_{goal}}| \leq |y_s - y_n| = 1$. Similarly, if $y_s = y_n$, then $h(s) - h(n) \leq |x_s - x_n| = 1$. In both cases, $h(s) - h(n) \leq 1 = c(s, a)$, i.e. $h(s) \leq h(succ(s, a)) + c(s, a)$. So the Manhattan distances are consistent in gridworlds in which the agent can move only in four compass directions.

b) **Prove that Adaptive A* leaves initially consistent h-values consistent even if action costs can increase.**

In the adaptive A^*, there are three cases for the h-values of state s and and its following state $succ(s, a)$, denoted as n for simplicity.

Case 1: both s and n were expanded in last search. In this case, both heuristics of state s and n

are the new defined ones, i.e. $h_{new}(s) = g(s_{goal}) - g(s)$ and $h_{new}(n) = g(s_{goal}) - g(n)$. Besides, the A^* search discovers a trajectory from the current state via state s to state n of cost $g(s) + c(s, a)$. The cost may not be the final g-value of state n because there may be a shorter path. Therefore, we can get $g(n) \leq g(s)+c(s, a)$. Thus, $h_{new}(s) = g(s_{goal})-g(s) \leq g(s_{goal})-g(n)+c(s, a) = h_{new}(n)+c(s, a)$.

Case 2: state s was expanded but state n was not in last search. In this case, the h-value of s is the new defined one, i.e. $h_{new}(s) = g(s_{goal}) - g(n)$. But that of state n is still the Manhattan distance to the target state, i.e. $h_{new}(n) = h(n)$. Also, $g(n) \leq g(s) + c(s, a)$ for the same reason described in Case 1. Besides, state n was generated but not expanded, which means the f-value of n is not smaller than that of s, i.e. $f(s) \leq f(n)$. Thus, $h_{new}(s) = g(s_{goal}) - g(s) = f(s_{goal}) - g(s) \leq f(s) - g(s) \leq f(n)-g(s) = g(n)+h(n)-g(s) = g(n)+h_{new}(n)-g(s) \leq g(n)+h_{new}(n)-g(n)+c(s, a) = h_{new}(n)+c(s, a)$.

Case 3: state s was not expanded in last search. In this case, $h_{new}(s) = h(s)$. Besides, the heuristics of the same state are monotonically nondecreasing over time, so we have $h(n) \leq h_{new}(n)$. Thus, $h_{new}(s) = h(s) \leq h(n) + c(s, a) \leq h_{new}(n) + c(s, a)$.

Therefore, the new h-values are consistent even if the action costs can increase.

Part 5: Heuristics in the Adaptive A^*

Over all experiments, you should observe that the adaptive A^* algorithm always outperforms the forward A^* algorithm, in terms of the total number of expanded cells during the search.

In the $n^{th}(n \geq 2)$ adaptive A^*, the h-values of states in the closed list of the $(n-1)^{th}$ adaptive A^* are changed to the real path lengths from those states to the target state when moving along from the path got from the $(n-1)^{th}$ adaptive A^*. The fewer expansions in the adaptive A^* benefit from the usage of the new h-values which are more likely to reflect their true distances to the target state than the Manhattan distances. A state with a small Manhattan distance to the target state may have a large new h-value in the condition that the possible paths with the lengths of Manhattan distance are all blocked. Using the new h-values can avoid to expand the states with large new h-values but small Manhattan distances, while these states will probably be expanded by the repeated forward A^*. Therefore, the adaptive A^* expands no more states than the repeated forward A^*.

Theoretically, the adaptive A^* will not expand more states than the repeated forward A^* for all cases. But there are some cases in the experiments in which the adaptive A^* expanded more states. After exploring, we found that it is the problem of priority queue, which is implemented using binary heap in C++. When the states with same f-values and g-values are insert into the priority queue, it doesn't always obey First-In-First-Out (FIFO) rule for these states. As a result, during a certain search the adaptive A^* may find another path with the same cost as the path forward A^* has found. The block distribution along the path of adaptive A^* may be more complicates and thus the adaptive A^* has to repeat more times afterwards than forward A^*, resulting in larger number of expanded states compared with repeated forward A^*.

Part 6: Memory Issues

The answer is **open**. For possible improvements, we are expecting something like the followings:

- Store the board in its **sparse representation**, i.e. only keeping the information of the blocked/open cells in memory.
- Use a **hash mapped array** to find/insert/delete/modify each cell on the board.