1 Compiling and Running under CLI/*nix

The implementation is split into the following Java files:

ArgumentParser.java Gridworld.java GridworldState.java PathFinder.java

To compile the program, type "javac PathFinder.java". If your grid input file is named "grid.txt", then the simplest way to run the solver is to type "java PathFinder grid.txt". But the program also takes a few command line options.

1.1 Pretty print

By default, the puzzle is printed with reference labels across the vertical and horizontal axis. The labels across the top are characters, e.g., "A B C ...", adding 1 to the ASCII value of the character for however many columns there are, eventually going into unprintable characters if there are too many columns. The labels across the sides are numbers, starting with 1. When the path is printed after the puzzle, the nodes visited are of the form (row, column), with the labels from the vertical and horizontal axis.

This behavior can be changed by running the program with "-pp0", i.e., pretty print off. No labels across the top and sides will be printed, and the path nodes will be of the form (row column), where row and column are both numbers. The counting starts from zero, so the first row and first column is labeled (0 0).

1.2 Omniscient

The normal implementation of the solver is a Repeated Forward A* search, as described in the relevant section in this document. Running the program with "-omni1" causes all blocked cells to be revealed to the agent before the first path is computed, causing the agent to take an optimal path to the goal state by navigating around all blocked cells. When this mode is turned on, the agent will either find the goal in the first path computation or it will determine that the goal cannot be reached, i.e., it becomes a simple Forward A* search that does not repeat.

1.3 Breaking Ties

When adding a state to the OPEN list (actually a priority queue), it is possible that the state being added has the same f-value as a state already in the list. If this happens, the default behavior is to compare the g-values of the two states to break the tie (smaller is better). This can be changed by running the program with the option "-h", which will cause the agent to break ties by comparing the g-values of the two states.

1.4 Example

"java PathFinder -pp0 -omni1 -h grid1.txt" would solve the puzzle grid1.txt by breaking ties with h-values, it would know about all blocked cells before the first path is computed and it would not print labels around the puzzle. See "sample_run.txt" for an example output with different options.

2 Repeated Forward A* implementation

First, the puzzle is read in and stored in an instance of Gridworld. Gridworld keeps track of global information about the puzzle, including which cells are visible to the agent.

A path from the start state to the goal is computed. Initially, the agent only knows about those blocked cells it is adjacent to. The agent will never expand states of cells it knows to be blocked, but it will expand any cell it doesn't know is blocked. After the first path is computed, the agent moves along it, and either reaches the goal and the search is over, or it tries to go into a blocked cell. Along the path that it moves, it keeps track of every cell it went on and every adjacent cell and makes them visible, i.e. it remembers any blocked cells it has seen.

If the agent was stopped along its path, it computes another path from its current state to the goal in the same way as above until the goal is reached or it determines the goal could not be reached. I have described how the agent uses Forward A* many times until it reaches its goal; below I describe the actual Forward A* algorithm that is used to compute a path from a start state to a goal state.

First, the start state is put into the OPEN priority queue. Elements in OPEN are sorted by their f-values; given state s,

$$f(s) = h(s) + g(s) \tag{1}$$

$h(s)$ is an heuristic estimate of the distance between state s and the goal. $g(s)$ is the distance between the start state and state s; thus, $f(s)$ is an estimate of the total distance between the start state and the goal. Therefore, the lower $f(s)$ is for state s, the more we prefer to go through state s.

We take the first element from OPEN, e.g., the state with the lowest f-value, and *expand* it. Expanding state s means we apply every valid operator to state s (move up, move down, move left, move right) to generate up to four new states. The idea is to add those states back to OPEN, which will sort them by f-value, and expand the next state in OPEN with the smallest f-value, until finally the state in front of OPEN is the goal. The path is reconstructed in reverse by following the goal back to the state that generated the goal, and the state before that, until the start state is reached (this is possible because whenever we expand a state, the new state points to the state that generated it).

There are a few more details. First, whenever we expand a state s to generate up to four new states, state s is put into CLOSED. We never expand any state that is in CLOSED. This is to ensure the algorithm terminates if a goal cannot be reached, i.e., it won't keep regenerating old states forever if it cannot find a goal. Second, it is possible that when we expand state s to generate new states, one of those generated states is already in OPEN. In this case, we just keep the state with the smallest g-value and throw out the other one. Lastly, to compute $h(s)$, we use Manhattan distance, which is the sum of the absolute values of the distance between the row of s and the row of the goal, and the column of s and the column of the goal.

In Gridworlds the Manhattan distance is an admissible heuristic, which means A* will always find an optimal path to the goal if one exists. However, Repeated Forward A* is not optimal, because it is misguided by its heuristic which leads it into blocked paths the agent does not know about. The "omniscient mode" included with the program implements an optimal A*.