CSCI 360 – Project #3

Programming Part: Navigation in Partially Known Environments - 7 Points

In this part of the project, you will implement A* and a variant of A*, called Adaptive A*, for navigation in partially known environments.

Problem Description

Figure 1: Warcraft II by Blizzard Entertainment

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

We study a variant of this search problem on a grid where an agent has to move from its current cell to a given destination cell. The agent can move one step in any of the four compass directions (with cost 1), as long as the adjacent cell in that direction is unblocked.

The agent always knows which (unblocked) cell it is in and which (unblocked) cell its destination is in. The agent knows that blocked cells remain blocked and unblocked cells remain unblocked but, initially, has only partial knowledge about the blocked cells (it knows that some cells are blocked, but does not know whether the remaining cells are blocked). However, it can always observe the blockage status of its eight adjacent cells (which is its field of view) and remember this information for future use.

Repeated Forward A*

A possible solution to the problem described above is to use the "freespace assumption." If we do not know whether a cell is blocked, we simply assume that it is unblocked. Under the freespace assumption, we repeatedly perform the following two steps:

1. Perform a forward A* search (that is, an A* search from the current cell of the agent to the destination cell) to find a shortest path from the current cell of the agent to the destination cell. (If no path exists under the freespace assumption, that means there is no solution.)

2. Execute the first move on this path and then update the knowledge of the environment by observing the blockage status of the adjacent cells.

Adaptive A*

Adaptive A* uses forward 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. 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 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 destination 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 destination 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 destination. 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, $g(s_{\text{goal}})$ 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. 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 preceeding 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. The above properties remain unchanged if the action costs increase over time.

Tie-Breaking

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. Hint: For the implementation part, priorities can be integers rather than pairs of integers. For example, you can use $c \times f(s) - g(s)$ as priorities to break ties in favor of cells with larger g-values, where c is a constant larger than the largest g-value of any generated cell.

Provided Code

We provide you with code that models the partially known grid ("PartiallyKnown-Grid.h" and "PartiallyKnownGrid.cpp"). It keeps track of the agent's current cell and all blocked cells that have been observed by the agent. The struct `xyLoc` is used to represent the x- and y-coordinates of cells. The coordinates of the top-left corner of the grid are (0,0). The `PartiallyKnownGrid` class provides the following functions:

- `GetWidth()`, `GetHeight()`: Returns the width (associated with the x-coordinate) and height (associated with the y-coordinate) of the grid, respectively.
- `IsValidLocation(xyLoc` l`)`: Returns true if and only if cell l is within the boundaries of the grid.
- `IsBlocked(xyLoc` l`)`: Returns true if and only if cell l is known to be blocked. A cell is assumed to be unblocked if the agent has not observed it in the past.

The `PartiallyKnownGrid` class also has the member function `MoveTo(xyLoc` l`)` that moves the agent to cell l if l is an unblocked neighbor of the current cell of the agent. We list this function separately since you will not call it, but it is how the agent moves around on the map and observes blocked cells.

We also provide you with code that can simulate the agent moving around on the map ("Simulation.h"), by repeatedly asking your code for a shortest path to the destination cell (under the freespace assumption) and moving the agent one step along the path, until the agent reaches the destination. You can customize how the simulation works (for debugging purposes) from the main function, using the following functions:

- `SetStepsPerSecond(double n)`: Number of moves the simulation makes per second.
- `SetConfirmationAfterEachMove(bool set)`: If set, the simulation will stop after each move until you hit "enter."

After each move, the simulation displays the grid on the terminal:

```
########################################
#0#.......##................##.........#
#.#................H...................#
#.........##................##.........#
#..###############.##############.######
#..###############H##############H######
#.##..#...............##....##$........#
#.#...#.####..##########..#.#######....#
#...#......H..............#............#
########################################
```

The meanings of the symbols are as follows: 0 represents the current cell of the agent, *$* represents the destination cell, # represents a cell that is known to be blocked, *H* represents a blocked cell that the agent has not yet observed and thus is assumed to be unblocked, and a dot represents an unblocked cell.

Implementation

We provide you with a skeleton implementation of the `GridPathPlanner` class. You will extend this class for your project. Specifically, you need to implement the following methods:

- `GridPathPlanner(PartiallyKnownGrid* grid, xyLoc destination, bool adaptive, bool larger_g)`: This constructor should initialize your class.
  - `PartiallyKnownGrid* grid`: This is the grid that you need for path-planning. You should initialize any of your internal variables that depend on the height and width of the grid (for instance, a table for h-values).
  - `xyLoc destination`: The destination cell is one of the parameters of the constructor since the destination cell is fixed. This also allows you to initialize the h-values during construction, if you prefer.
  - `bool adaptive`: Determines whether your code should operate as Repeated Forward A* (if `adaptive = false`) or as Adaptive A* (`adaptive = true`). Notice that the only difference between Repeated Forward A* and Adaptive A* is that Adaptive A* updates the h-values between the A* searches and uses the updated h-values during the next search.
  - `bool larger_g`: Determines whether your code should break ties in favor of cells with larger g-values (if `larger_g = true`) or smaller g-values (if `larger_g = false`).
- `void FindPath(xyLoc start, vector⟨xyLoc⟩ & path)`: This function should run a Forward A* or Adaptive A* search (based on how the parameter `adaptive` is initialized during construction) to find a shortest path from the start cell (parameter `xyLoc start`) to the destination cell under the freespace assumption. Note that you do not need to make the freespace assumption explicitly, as the `IsBlocked(xyLoc)` function of the `PartiallyKnownGrid` class returns whether a cell is blocked under the freespace assumption. This function should fill in the pass-by-reference parameter `path` as its output. The first cell on the path should be the start cell, the last cell on the path should be the destination cell, every cell on the path should be unblocked (under the freespace assumption), and every two consecutive cells on the path should be adjacent on the grid. All of these criteria are checked by the `Simulator` class, which prints an error message if one of them is violated. The `Simulator` class does not check whether the path your code returns is a shortest path. If your code works correctly, the agent should reach the destination in 140 steps.
- `int GetHValue(xyLoc l)`: This function should return the estimate distance from the given cell l to the destination cell. If the class is initialized with `adaptive = false`, it should return the Manhattan Distance (see the next section for details). Otherwise, it should return the updated h-value of the cell.

- `int GetNumExpansionsFromLastSearch()`: This function should return the number of states expanded by the most recent Forward A* or Adaptive A* search. During the call to the FindPath function, you should keep track of the number of expanded states.

You are not allowed to use any external libraries for your code, except for STL. We have tested and verified that the provided code (and a solution to the project that extends the provided code) compiles and runs on Ubuntu 16.04. You are free to develop your code on any platform that you choose, although we have not tested whether our code works on non-Linux operating systems (and we might not be able to help you to get your code to work on your preferred platform). We will test your code on a Linux machine that supports C++11.

Further Requirements

- h-value: Your Forward A* implementation should use the Manhattan Distance to the destination as the h-value of each cell. Your Adaptive A* implementation should initialize the h-value of each cell with its Manhattan Distance to the destination, which can then be updated according to the update rule of Adaptive A*. The Manhattan Distance between cells $l1$ and $l2$ is calculated as follows:

$$|l1.x - l2.x| + |l1.y - l2.y|$$

- Although we do not expect your code to be optimized at the coding level, we expect it to be optimized at the algorithmic level. That is, you should use the proper data structures and use duplicate detection to avoid expanding the same state twice. We will both time your code and look at its number of expansions. You will lose points if you expand a state more than once during a search or if you use improper data structures (such as using a vector for implementing the open list).

Experimental Results

Run both Repeated Forward A* and Adaptive A* with tie-breaking towards both larger and smaller g-values (for a total of four different runs) and record the average number of expansions in each run, then answer the following questions. For each question, explain your observations in detail, that is, explain what you observed and give a reason for the observation.

1. Compare Repeated Forward A* and Adaptive A* (using both tie-breaking strategies).
2. Compare the two tie-breaking strategies (for both Repeated Forward A* and Adaptive A*). For the explanation part, consider which cells both versions of Repeated Forward A* expand on a 5×5 grid with no blocked cells.

Theoretical Part: Constraint Satisfaction - 3 Points

Download the AIspace `Consistency Based CSP Solver` Java applet from `http://aispace.org/constraint/index.shtml`. You might need to add `http://www.aispace.org/` to the `Exception Site List` available in the `Security` tab of the `Java Control Panel` to get the applet to run. Read the documentation available at the download site and try it out. Do this well before the deadline to ensure that it works on your computer.

You are given a 2×3 grid with two rows (0 and 1) and three columns (0, 1 and 2). You need to paint each cell in one of three colors: Blue, Red, and Green. No two cells that share a border can have the same color, and you are told that the bottom-center cell should be Green and the top-center cell should be Blue.

1. Build the constraint network for the given problem. Remember that you need to specify the variables, domains, and constraints. Save your network as "Part2.xml". Include a screenshot of your network in your submission.

2. Run the arc consistency (AC) algorithm on the network (using the "fine step" option in the applet). What was the first inconsistent arc found by AC? How was it made consistent? What was the second one? How was it made consistent? Can AC solve this problem? Include a screenshot of your network in your submission after running the AC algorithm.

3. Assume that you are given the additional constraint that the cells in the top-right and bottom-left corners must be painted in the same color. Can AC solve this more constrained problem? Why or why not?

Submission

You need to submit your solutions through the blackboard system by Thursday, November 15, 11:59pm. For the programming part, you should submit your modified "GridPathPlanner.h" and "GridPathPlanner.cpp" files, and a PDF file named "Part1.pdf" that explains your answers to the questions about your experimental results. For the theoretical part, you should submit a PDF file named "Part2.pdf" that explains your answers and contains the necessary screenshots, as well as the the xml file "Part2.xml" that contains your constraint network (using `File` → `Save program` to save your network as an xml file). Upload the five files individually and not as an archive. If you need to resubmit a single file, resubmit all five of them (we will download and grade only the last attempt). If you have any questions about the project, you can post them on campuswire (you can find the link on the course wiki) or ask a TA or CP directly during office hours. If you have a bug in your code that you have been unable to find, you can ask for help from a CP.