CSCI 360 – Project #3

Theoretical Part: Constraint Satisfaction - 4 Points

Download the AIspace `'Consistency Based CSP Solver'` Java applet from `http://aispace.org/constraint/`. 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 a different color than from the cells it shares an edge with. You can use only three colors: Blue, Red and Green. Also, 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 an .xml file. 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?

Programming Part: Navigation in Partially Known Environments - 6 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

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 known or unknown environment, and the game characters then move autonomously to the location that the player clicked on. They always observe the environment within their limited field of

Figure 1: Warcraft II by Blizzard Entertainment

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 target cell (of a non-moving target). 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 the target is in. The agent knows that blocked cells remain blocked and unblocked cells remain unblocked but, initially, it 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 target cell) to find a shortest path from the current cell of the agent to the target 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 our knowledge of the environment by observing the blockage status of the adjacent cells.

Adaptive 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. 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, $g(s_{\text{goal}})$ 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.

Provided Code

We provide you with code that models the partially known grid ('PartiallyKnownGrid.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 PartiallyKnownGrid class provides the following functions:

- GetWidth(), GetHeight(): Returns the width and height of the grid, respectively.
- GetCurrentLocation(), GetTargetLocation(): Returns the x and y positions of the cell of the agent and the target, respectively.
- IsBlocked(xyLoc l): Returns true if and only if cell l (with x and y coordinates $l.x$ and $l.y$) is known to be blocked.

The PartiallyKnownGrid class also has a 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 the map and observes blocked cells.

'main.cpp' contains the function 'Simulate' that iteratively asks your code (details below) for a shortest path and then moves the agent one step along the path, until the agent reaches the target. After each move, it displays the grid on the terminal:

```
########################################
#O#.......##.............##.........#
#.#.............H...................#
#.........##.............##.........#
#..###############.##############.######
#..###############H##############H######
#.##..#.............##....##$........#
#.#...#.####..##########..#.#######....#
#...#......H............#............#
########################################
```

The meanings of the symbols are as follows: 0 represents the current cell of the agent, $ represents the target cell, # represents a cell that is known to be blocked, H represents a hidden obstacle (so, IsBlocked returns false for that cell), 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, bool use_adaptive_a_star): This constructor should initialize your class. The partially known grid is one of the parameters of the constructor since you might want to at least get the height and width of the grid for initializing your internal variables. The parameter 'use_adaptive_a_star' determines whether your code should operate as Repeated Forward A* (if 'use_adaptive_a_star' = false) or as Adaptive A* (if 'use_adaptive_a_star' = true). Notice that the only difference between Repeated Forward A* and Adaptive A* is that Adaptive A* updates the h-values between searches and uses the updated h-values during the next search.

- xyLoc GetNextMove(PartiallyKnownGrid* grid): This function should return the cell that the agent should move to next when following a shortest path from its current cell to the target cell. In order to determine the shortest path to the target, you should run a Forward A* or Adaptive A* search (based on the value of the parameter 'use_adaptive_a_star' during construction).

- int GetNumExpansions(): This function should return the number of states expanded by the most recent Forward A* or Adaptive A* search. During the call to the GetNextMove function, you should keep track of the number of expanded states. Currently, we are not calling this function, but you will need it for the Adaptive A* question (see below) and we will check your number of expanded states during grading to see if everything works correctly.

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 `aludra.usc.edu`. You are free to develop your code on any platform that you choose, although we have not tested if our code works on non-Linux operating systems (and we might not be able to help you to get your code work on your preferred platform). We will test your code on a Linux machine that supports C++11.

Further Details

- Heuristic: Your Forward A* implementation should use the Manhattan Distance heuristic. Your Adaptive A* implementation should also start with the

Manhattan Distance heuristic as the base heuristic. The Manhattan Distance between cells $l1$ and $l2$ is calculated as follows:

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

- Tie-Breaking: Both your Forward A* and Adaptive A* implementations should use the following tie-breaking strategy: If two states have the same f-value, prioritize expanding the one with the larger g-value. If two states have the same f- and g-values, prioritize expanding the one with the smaller xyLoc (the '$<$' operator is overloaded for the xyLoc struct in the provided code).

Adaptive A* Question

Compare Repeated Forward A* and Adaptive A* with respect to the number of states they expand. You should compare the number of expansions that both algorithms perform during their first search, the number of expansions during their second search, and the average number of expansions over all searches until the target is reached. Explain your observations in detail, that is, explain what you observed and provide a reason for your observations.

Submission

You need to submit your solutions through the blackboard system by Wednesday, Nov. 30, 11:59pm.

For the theoretical part, you should submit a PDF file named 'Part1.pdf' that contains the required screenshots. You also need to submit an xml file named 'Part1.xml', that contains your constraint network (using `'File'` $\rightarrow$ `'Save program'` to save your network as an xml file).

For the programming part, you should submit your modified 'GridPathPlanner.cpp' and 'GridPathPlanner.h' files, and a PDF file named 'Part2.pdf' that contains your answer to the Adaptive A* question. If you have created additional source files, include them in your submission along with a makefile that compiles your project.

If you have any questions about the project, you can post them on Piazza (you can find the link on the course wiki) or come to the TAs' office hours.