Artificial Intelligence: Heuristic Search

Nathan Morgenstern, Seo Bo Shim

November 26, 2017

Contents

Introduction to GUI

When the application first starts the user has the option to select either their own map, or a map that has been included by default. If the user wants to load their own map, they should selected user generated map and input the file path into the file input text field. Otherwise, they can simply select a pre-generated map from the drop down menu along with its variation.

There are a total of 4 algorithms included in the main menu that the user can then select from to find a path from the start to goal. Options include: Uniform cost search, A*, Weighted A*, and Sequential A*. If the user selects weighted or sequential A* they should input two weights into the weight text fields. Once all the parameters are selected by the user, they can select confirm and the map along with its solution using the selected algorithm will be loaded.

Figure 1: The main menu of the application

Correct Solutions from instantiations of heuristic algorithm

Figure 2: Uniform Cost Search example

Figure 3: A* Search example - Heuristic 1

Figure 4: Weighted A* Search example - Heuristic 2 w1 = 1.25, w2 = 2

Figure 5: Sequential A* Search example - w1 = 1.25, w2 = 2

Optimization of heuristic algorithm

Overall Optimizations

When searching for the neighboring nodes to expand to, we can ignore the neighbor that is the parent because the parent was already explored. This saves the computation time of calculating the path and adding/removing from the binary heap.

The table that keeps the explored set of nodes is a 2 dimensional array. Since our grid is a 2d table, each node is assigned a coordinate. In the grid object that contains the graph of the map and all the nodes, we store a two dimensional array of nodes that represents the grid.

Also for keeping the fringe or the frontier of the graph search, we use a priority queue which is equivalent to a priority heap. We desire a minimal time for a search operation, and a binary heap provides this in O(log(n)) time. Finding the highest element also takes O(1), which is also significant in choosing this data structure to store our possible nodes in our frontier.

Since the overall algorithm is the same for all the types of searches, there were no optimizations specific to each type of search. Each search is different in how it calculates the value, which can consist of the actual path cost to the node or include the heuristic as well. IN the abstract algorithm we call an abstract method, where the implementation of this method is the only difference between the searches.

Heuristic Proposals

Best admissible/consistent heuristic

The best heuristic can be achieved by considering the minimum path cost from the start to end goal.

We will consider the shortest path for three cases, and generalize the results

-A path from the start node to the goal node is a straight line.

The shortest possible path can occur if a river runs straight from the start to the goal node. This is possible because the river can possibly continue straight across the whole grid. This configuration allows the minimal past cost for this situation. The heuristic value of a node, the predicted cost from the node to the goal:

s is the current node, s_g is the goal node coordinates.

$t = 0.25$: multiplier for the type of block.

$d = 1.0$: multiplier for direction of movement.

$$h(s) = t * d * max(|s^y - s_g^y|, |s^x - s_g^x|)$$

$$h(s) = 0.25 * 1 * max(|s^y - s_g^y|, |s^x - s_g^x|)$$

-A path from the start node to goal node has diagonals and straight lines.

We note that it costs less to move non-diagonally on highways than than moving diagonally along non-rivers. The cost of moving diagonally is:

$$h(s)4 * 0.25 * \sqrt{2} * \sqrt{2} * min(|s^y - s_g^y|, |s^x - s_g^x|))$$

We note that moves can only be made vertically or horizontally on a highway. So we calculate the total

amount of vertical and horizontal moves.

$$h(s) = t * d * (vert_dist + horiz_dist)$$

$$h(s) = 0.25 * 1 * (|s^y - s_g^y| + |s^x - s_g^x|)$$

We can clearly see that this heuristic is minimized.

Inadmissible heuristics

We argue that inadmissible heuristics should be used for the A* algorithm on this graph because the range of possible values for the path cost in this grid is high. Since the range is high, to select a admissible heuristic, we must shoot low and calculate the heuristic conservatively to not exceed the minimum path cost. Ideally, we wish for the minimum path cost to be close to the average, or the range to be smaller so that the heuristic value is not much smaller than the cost. Condition:

$$h(s) <= c(s, s') + h(s')$$

$$h(s) - h(s') <= min(c(s, s'))$$

To find the maximal heuristic that fits this condition, we must find the minimum possible cost value. With a high range of values for the cost we find that:

$$h(s) - h(s') << average(c(s, s'))$$

In other words, the case where the path is strictly on a river or highway is very unlikely, yet it is still possible. This in turn reduces the maximum heuristic value to force the heuristic to be admissible, and in turn renders the heuristic value to minimally affect the estimated path cost. This is why the admissible and consistent A* implementation almost exactly follows the Uniform Cost Search algorithm.

H2: Type-based Heuristic

Currently, the abstract algorithm tends to follow rivers because the path cost to these rivers is small. However this is computationally expensive, as these nodes will have to continuously be expanded along and not along the river. We can argue that the path should follow the river as long as this does not increase the Euclidian distance from the current node to the goal.

If the current node is on a river, then the heuristic can be calculated to be:

$$h(s) = 0.38 * 1 * (|s^y - s_g^y| + |s^x - s_g^x|)$$

If the river takes us away from the goal, then the heuristic will be large enough that a step of 1 will be enough to have a lower path cost than following the river away from the goal.

The result is that nodes along the river going away from the goal will be expanded less, thus reducing computation time when the path is on a river.

When the current node is a hard to traverse cell, we make the assumption that this region will be full, so we avoid these cells by raising the heuristic value of. We calculate that traversing around a 31x31 grid of hard-to-traverse cells is slightly shorter than traversing through the 31x31 grid. Since there's only a 50 percent chance of a hard-to-traverse cell of being. Note: traveling through the difficult blocks on average will yield a path cost of 1.5.

Traversing around the 31x31 block: $30 + 30 + \sqrt{2} = 61.41$

Traversing through the 31x31 block

$$\sqrt{2} * 30 * (1 + (1 * (0.50))) = 65.76$$

$61.41 < 65.76$. It is advantageous to travel around the 31x31 grid. Though it may have a higher path cost to travel around the block, because with a larger number of blocks. So this heuristic should be designed to have the path best avoid these hard-to-traverse cells, not necessarily avoid them completely.

$$h(s) = 1.5 * 1 * (|s^y - s_g^y| + |s^x - s_g^x|)$$

When the current node is a regular unblocked cell:

$$h(s) = t1 * d1 * max(vert_dist, horiz_dist) + t2 * d2 * min(vert_dist, horiz_dist)$$

$$h(s) = 1 * 1 * min(|s^y - s_g^y|, |s^x - s_g^x|) + 1 * \sqrt{2} * max(|s^y - s_g^y|, |s^x - s_g^x|)$$

s is the current node, s_g is the goal node coordinates.

$t1 = t2 = 1$: multiplier for the type of block.

$d1 = 1.0$, $d2 = \sqrt{2}$: multiplier for direction of movement.

This heuristic is advantageous because since it uses the current node type, it is computationally inexpensive.

H3: Expected Value

This method will take into account the number of blocked cells and the number of hard to traverse cells. The value that's generated from the heuristic will take the probability of . The heuristic value will be the average cost of each step combined with a value thats proportional to the node's distance to the goal.

Average cost of step = summation of: probability of scenario * cost of scenario

$$average(s) = \sum P(s) * C(s)$$

]

$$average(s) = (0.8 * 1 + 8 * 0.05 * 0.5 * 1.5 - 7 * 0.05 * 0.5 * 1.5))$$

There are 8 possible moves from a single node. 20 percent of nodes will be blocked, on average leaving 6.4 possible moves from a node. Of

H4: Neighbor-checking Heuristic

This heuristic will check it's neighbors to determine it's heuristic value.

H5: ?

Calculate the average path cost.

Sub-optimality of Sequential A*

$$g_0(s) <= w_1 * c^*(s)$$