Heuristics

1 Designing Heuristic Functions

What do we know already about heuristic functions? (i) They're problem-specific; (ii) they indicate the promise of states, not actions; (iii) they're estimates of the cost of the cheapest path to the goal; (iv) $h(n) = 0$ if n is a goal node; and (v) admissible heuristics never overestimate and can be used to give complete and optimal searches.

What else can we say? While we seek an h that never overestimates, we also prefer its estimates to be as close to the actual path costs as possible. (If, as an idealisation, its estimates equal the path costs, then the search will never be side-tracked and will move unfailingly and directly to the goal.) If we have two heuristic functions, h_1 and h_2, such that $h_1(n) < h_2(n)$ for all non-goal nodes, n, then an A^* search using h_2 will, on average, expand fewer nodes than an A^* search using h_1.

But there is another factor at play: how much it costs to compute $h(n)$. We need h to be a function that we can apply with low computational cost. Unfortunately, in general, the better heuristic functions are often the more computationally expensive ones. If some function h is too computationally expensive, it might be better to abandon it, to use a computationally simpler function that gives less good estimates, and put up with the extra search that is incurred from using this less good heuristic: it's possible the extra search will take less time than it would take to use the expensive h.

So how do we set about designing a heuristic function for some state space? You require a good knowledge of your problem-domain and an amount of ingenuity. However, there is some advice on how to go about it.

Imagine a *relaxed problem*. A relaxed problem is one just like the one you're trying to solve, but in which the operators have less demanding preconditions. The cost of an exact solution to the relaxed problem is often a good heuristic for the original problem.

For example, suppose your original problem is to find a road route from one city, along roads through a number of other cities, to some goal city. A relaxed version of this problem might be one where you have some magic vehicle that allows you to journey directly from city to city instead of winding your way along the roads. In this relaxed problem, the actual cost of each action will be the straight-line distance ('as the crow flies') between the cities, and that's why this is one possible heuristic for the original problem.

Similarly, a relaxed version of the 8-puzzle is one where a tile is no longer constrained to slide from one position to an adjacent empty position in each move. Instead, any tile can be moved to any other position in the grid (even an occupied position) in a single move. In this relaxed problem, the actual cost of the solution path will be the number of tiles out of place (since each can be put in its rightful place by a single move), and that's why this is one possible heuristic for the 8-puzzle.

The process of relaxing problems and generating heuristics from the relaxed problems has even been automated in a system called ABSOLVER. But there is another automatic approach to coming up with heuristic functions: machine learning.

2 Learning Heuristic Functions

Heuristic functions can be adjusted on the fly. Systems that do this would improve their performance on subsequent problems: initially they will perform badly but as they are used for more and more searches, they will get better.

In the 1950s, Arthur Samuel wrote several programs for playing tournament-level checkers/draughts. His program could learn, and, by playing itself thousands of times, quickly learned how to beat its creator. It worked in a way that was a very early precursor to the techniques I'm about to describe.

This is a different kind of learning from what we looked at before. Previously, we were looking at *supervised learning*. A 'teacher' supplies examples of inputs and target outputs. Now, we are looking at an example of *reinforcement learning*. The agent is trying out actions in different states and receives (at some point) a *reward* (which may be negative in which case it's more like a punishment) for the actions it chooses. It uses this reward to make it more or less likely that it will choose that action in similar states in the future. The difference here is that there is no 'teacher' telling the agent what the 'correct' (target) output (in this case, the action) should be.

This is quite a general form of learning, and can be used in many areas of AI. We'll look at it in the context of learning a heuristic function.

3 Learning a Heuristic Function for a Small State Space

In this section, we're going to make an assumption. It's not a very realistic assumption, but we get a more easily understood algorithm by making this assumption. We'll abandon the assumption in the next section. The assumption is that the state space contains k states, and that k is small: small enough for us to be able to hold, in memory, a table (having k rows) of all possible states in the state space.

We build in memory a $k \times 2$ table, which is where we'll be recording our learned values for h. In the first column of the table, we put the k states; in the second column, we put zeros. We then carry out an A^* search. The evaluation function is, of course, $f(n) = g(n) + h(n)$. The values for $h(n)$ come from the table.

During the search, after expanding state n_i to produce its successors, $\mathrm{succ}(n_i)$, we update the corresponding row in h:

$$h(n_i) = \begin{cases} 0 & \text{if } n_i \text{ is a goal state} \\ \infty & \text{if } n_i \text{ is a dead-end (no successors)} \\ \min_{n_j \in \mathrm{succ}(n_i)}[c(n_i, n_j) + h(n_j)] & \text{otherwise} \end{cases}$$

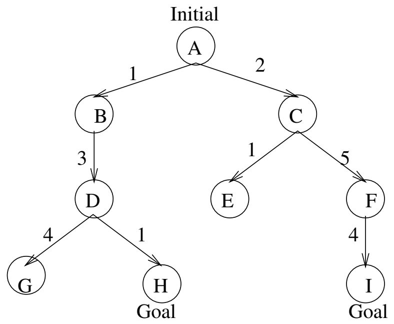
where $c(n_i, n_j)$ is the cost of the move from state n_i to n_j.

The first time you do an A^* search, the table contains all zeros, so your search is really just a least-cost search: the heuristic is contributing nothing. But information is being gathered that will help subsequent searches.

In subsequent A^* searches (from possibly different initial states but with the same states being goal states), some of the nodes that you visit (ones you visited before) may now have a non-zero heuristic value. Over the course of several searches, better and better values will be propagated back from the goal nodes to earlier nodes in the search tree. (It's hard to explain why in words, but you'll see it in the example.)

Eventually, the tabulated heuristic function should converge to give values that perfectly 'estimate' the costs of the cheapest paths to the goal.

We'll illustrate the algorithm on this state space:



4 Learning a Heuristic Function for a Large State Space

Suppose, as is more normal, that the state space is too large: we cannot tabulate the function h. We must represent and learn h in some more compact form, in which generalisations over different states are captured. The approach we'll look at has similarities with the neural net learning we looked at before: both involve adjusting weights.

The idea is that we, as the designers, come up with a set of subfunctions that might be good components of a heuristic function. For example, in the 8-puzzle, we might use h_1, the number of tiles out of place, h_2, the sum of the Manhattan distances between current and desired positions, and any other functions we can think of that relate a tile's current position to its goal position. The overall heuristic function is then a weighted combination of these subfunctions:

$$h(n) \triangleq w_1 h_1(n) + w_2 h_2(n) + \ldots$$

We start off with some arbitrary values for the weights w_1, w_2, etc. We conduct a number of A^* searches. Whenever we expand a node, we adjust the weights in h. Learning involves adjusting the weights to get a function that, over time, better estimates the required values.

The maths doesn't matter, but I'll give it anyway for those who are interested.

The current value for h at node n_i is $h(n_i)$. But, a better estimate (as per the simpler algorithm in the previous section) is $\min_{n_j \in \mathrm{succ}(n_i)}[c(n_i, n_j) + h(n_j)]$, i.e. a better estimate is the cost of getting to a successor plus that successor's estimate, and we want the cheapest of these. The error in $h(n_i)$ is therefore the difference between the better estimate and the current value:

$$\min_{n_j \in \mathrm{succ}(n_i)}[c(n_i, n_j) + h(n_j)] - h(n_i)$$

That's how much $h(n_i)$ should change by. But we'll multiply this by α, a learning rate, so that we don't make adjustments that are too large (as per learning in neural nets):

$$\alpha \times \min_{n_j \in \mathrm{succ}(n_i)}[c(n_i, n_j) + h(n_j)] - h(n_i)$$

But, we must 'divvy' this change out among the different weights.

To update weight w_1, we use:

$$w_1 := w_1 + \alpha \times (\min_{n_j \in \mathrm{succ}(n_i)}[c(n_i, n_j) + h(n_j)] - h(n_i)) \times \frac{\delta h}{\delta w_1}$$

(and similarly for $w_2, \ldots$).

In fact, if $h(n) \triangleq w_1 h_1(n) + w_2 h_2(n) + \ldots$, then $\frac{\delta h}{\delta w_1}$ is simply $h_1(n)$, and so the weight update function simplifies to:

$$w_1 := w_1 + \alpha \times (\min_{n_j \in \mathrm{succ}(n_i)}[c(n_i, n_j) + h(n_j)] - h(n_i)) \times h_1(n)$$

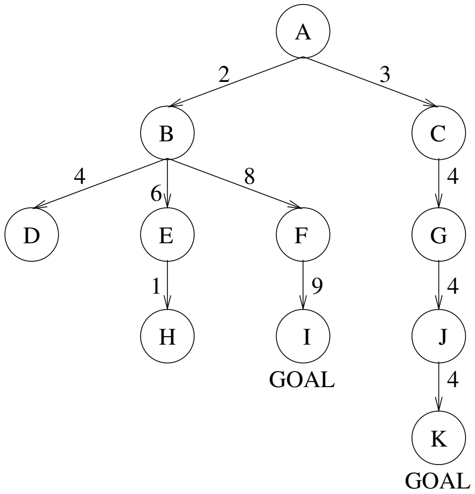
5 Temporal Difference Learning (Optional)

What we've been looking at is an example of a quite general form of reinforcement learning called *temporal difference learning*. It's called this because the adjustments are made on the basis only of two *successive* values. (Of course, for convergence, the process has to be performed iteratively over many searches.) It was the basis of Samuel's checkers/draughts program (although it wasn't called 'temporal difference learning' in those days), and it can be used in many other learning situations. Here are two variations:

- Suppose an agent knows what actions it can carry out, but it doesn't know the effects of those actions or the costs of those actions. It can learn the effects of actions, costs of actions and a heuristic function all at the same time. But the learning will have to take place in the real world (by executing actions for real) rather than in its head. It will initially choose random actions until it is lucky enough to reach its goal. But on subsequent runs, it will have some information from previous runs that will help it to choose actions. Note however that optimal paths are not guaranteed because they might not be the ones visited by the random actions. So you sometimes allow the agent to take occasional random actions as a way of avoiding get stuck in a non-optimal rut.
- We've been looking at the case where the agent gets rewarded positively just once, when it reaches a goal; and it gets rewarded negatively (punished) every time it takes an action (by the amount of the action's cost). But temporal difference learning can be applied in scenarios where, instead of there being a goal condition, there is an ongoing task: the agent carries out actions, it sometimes receives positive or negative rewards, and it must maximise its total reward. Note that this is different from learning a heuristic. Heuristics evaluate the 'promise' of states, but what I'm describing here is an evaluation of the 'promise' of actions.

Exercises

1. State whether each of the following statements is *true* or *false*. To obtain credit, you must in each case **explain** your answer correctly and in detail.
   (a) The terms *state space* and *search tree* are synonymous (i.e. they mean the same).
   (b) A goal state in a state space always has no successors.
   (c) A goal node in a search tree is always a leaf node.
   (d) Breadth-first search is a complete strategy becauses it searches exhaustively (i.e. it visits every state at least once).
   (e) Depth-first search is an optimal strategy because it searches exhaustively.
   (f) If A^* expands a node and one of the node's children is a goal node, then A^* terminates immediately.
   (g) If $g(n)$ is the length of the path to node n, and $h(n) = g(n)$, then A^*, using $f(n) = g(n) + h(n)$ as its evaluation function, will perform a breadth-first search.
   (h) If $g(n)$ is the length of the path to node n, and $h(n) = -g(n)$, then A^*, using $f(n) = g(n) + h(n)$ as its evaluation function, will perform a depth-first search.
2. Consider the following state space in which the states are shown as nodes labelled A through K. A is the initial state, and I and K are the goal states. The numbers alongside the edges represent the costs of moving between the states.



Use the algorithm covered in lectures to learn a heuristic function for this state space. Show your working. Stop after you have carried out two A^* searches.