Informed Search

1 Heuristic Search Strategies

In least-cost search, the agenda is a priority-ordered queue, ordered by path cost. By always taking nodes from the front of the queue, the path that we select to extend is always the cheapest so far. In *informed search* (also called *directed search* and *heuristic search*), we continue to use a priority-ordered queue. The ordering is now determined by an *evaluation function*, which for each node on the agenda returns a number that signifies the 'promise' of that node. Perhaps counter-intuitively, we use the convention that smaller numbers designate higher 'promise', so the nodes on our queue will be in ascending order.

One of the most important kinds of knowledge to use when constructing an evaluation function is an estimate of the cost of the cheapest path from the state to a goal state. Functions that calculate such estimates are called *heuristic functions*. (In AI, the word 'heuristic' is not used only in the context of 'heuristic functions'. It is also used for any technique that might improve average-case performance but does not necessarily improve worst-case performance.)

It's important to be clear that a heuristic function is used to evaluate the promise of a *state*. We choose which node to expand next using the heuristic value of its state. Heuristic functions do not evaluate *operators*, i.e. if several operators can be used to expand a node, heuristic functions do not say which is the most promising action.

Heuristic functions are problem-specific. We must design (or learn) different functions for different problem domains. Here are examples of heuristic functions for the 8-tiles puzzle:

$$h_1(n) \triangleq \text{the number of tiles out of place in this state relative to the goal state}$$

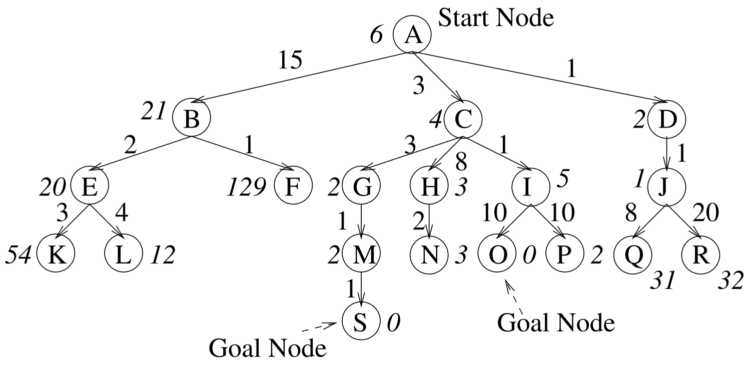
$$h_2(n) \triangleq \text{the sum, for each tile, of the Manhattan distance between its position in this state and its position in the goal state}$$

Both functions estimate the number of moves we'll have to make to modify the current state into a goal state. In fact, both h_1 and h_2 *underestimate* the costs of the cheapest paths in this state space, and this turns out to be a significant property (see later).

One general property of a heuristic function is that $h(n) = 0$ if n is a goal.

We'll discuss heuristics more in the next lecture.

We'll be illustrating the strategies in the lecture on the following simple state space. Path costs are shown along the edges in this graph. The results of applying the heuristic function are shown alongside the nodes.



2 Greedy Search

In *greedy search*, the agenda is a priority-ordered queue, ordered only using a heuristic function, h. This means that the node that we select to expand will always be the one whose state is judged, by the heuristic function, to be the one that is nearest to a goal.

It takes very little to change our Java implementation.

Heuristics evaluate the promise of states, so we need to change the Java interface, `IState`, by adding another method. It becomes:

```
public interface IState
{
   public boolean isGoal();

   public int getHeuristicValue();

   public Iterator getSuccessors();
}
```

And classes such as `EightPuzzleState` and `WaterJugsState` would then need to include an implementation of the method `getHeuristicValue()`. (We won't show those here.)

Then we need a subclass of `NodeComparator` used for ordering nodes on the agenda. Our new subclass orders them not by path cost (as per `NodeCostComparator`), but by the heuristic value of their states:

```
public class NodeGreedyComparator
   extends NodeComparator
{
   public boolean isLessThan(Node aNode, Node otherNode)
   {  return aNode.getState().getHeuristicValue() <
         otherNode.getState().getHeuristicValue();
   }
}
```

We should now evaluate this search strategy.

Complete? Greedy search is not complete. *Why?*

Optimal? Greedy search is not optimal. *Why?*

Time complexity Its worst-case time complexity is $O(b^m)$, the same as least-cost search (and for similar reasons).

Space complexity It has to hold on to all unfinished paths in case it later wishes to explore them further. In its space requirements, it is therefore again similar to least-cost search: $O(b^m)$.

The news doesn't sound good for greedy search. But, the whole point about heuristics is that they should help performance on typical problems (ones that arise in practice). So, with a well-chosen heuristic function, greedy search might perform well on the problems we place in front of it. The time and space demands might be quite reasonable on these problems. (Imagine, e.g., in the best case that the heuristic leads unfailingly to the goal node.)

3 A^* Search

We've looked at least-cost search and seen that, while complete and optimal, it may not focus the search enough and can therefore be inefficient. And we've looked at greedy search and seen that it may prove too focused (which is a problem if it focuses effort on the wrong parts of the state space). The idea in A^* search is to combine these two approaches.

In A^* *search*, we treat the agenda as a priority-ordered queue but the evaluation function, f, that we use to determine the ordering is:

$$f(n) \triangleq g(n) + h(n)$$

where $g(n)$ is the cost of the path to node n and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to a goal.

We'll therefore be ordering the agenda based on estimates of full path costs (not just the cost so far, and not just the estimated remaining cost, but the two together).

In our Java implementation, we need another subclass of `NodeComparator`:

```
public class NodeAStarComparator
   extends NodeComparator
{
   public boolean isLessThan(Node aNode, Node otherNode)
   {  return (aNode.getPathCost() +
         aNode.getState().getHeuristicValue()) <
         (otherNode.getPathCost() +
         otherNode.getState().getHeuristicValue());
   }
}
```

The full definition of A^* search places a requirement on function h. (So, a search strategy that uses g and h as above should not be described as A^* search unless h satisfies this condition).

The condition is that h *never overestimates* the cost of the cheapest path to the goal. (Informally, we might say that it must be an optimistic function!) Heuristic functions that meet this condition are called *admissible heuristic functions*.

The heuristics h_1 and h_2 that we gave for the 8-puzzle (number of tiles out of place and sum of Manhattan distances) were both admissible.

If you were doing route planning, your heuristic function could be straight-line distances between two points ('as the crow files'), since this will never overestimate the true distance following roads/rail links/etc. on the ground.

So, let's evaluate A^* search. (Remember A^* uses $f(n) \triangleq g(n) + h(n)$ and requires h to be admissible.)

Complete? A^* search is complete. *When and why?*

Optimal? A^* search is optimal. *When and why?*

Time complexity The worst-case time complexity is the same as greedy search, $O(b^m)$.

Space complexity The worst-case space complexity is the same as greedy search, $O(b^m)$.

Although the worst-case time and space complexities are bad, we hope (as we did for greedy search) that the heuristic will focus the search on the typical problems that we face in the real world. But now we have the advantage, over greedy search, of completeness and optimality.

[Advanced point. *Ignore this if it makes no sense.*

Remember the need to avoid re-exploration? Remember the hash table solution? And remember that for least-cost search the whole approach has to be made more complicated because you sometimes throw away the node you're about to add to the agenda, and other times you throw away something that's already in the search tree, in order to keep the cheapest path to a node at all times?

This extra complication sometimes does not arise in A^* search: the simpler implementation sometimes suffices! We can place an extra condition on h such that, whenever A^* expands a node, the path it will have found to that node is guaranteed to be the one with the smallest value of f. So if we later find another path to that node, this is the one we discard. The extra condition is: if n_j is a successor of n_i and $c(n_i, n_j)$ is the cost of the operator from n_i to n_j, we require $h(n_i) - h(n_j) \leq c(n_i, n_j)$.]

4 Variations on A^*

Search is so important in AI that people are always inventing new strategies. So I just wanted to mention that there are lots of search strategies that we have not looked at. I mention two in the next two paragraphs, but they're non-examinable.

IDA^*, *iterative-deepening* A^*, is heuristic search but is inspired by iterative-deepening search. Like iterative-deepening, it is complete and optimal but it has linear space requirements, which may prove very important to you if you're trying to use A^* but keep running out of space. It basically does repeated depth-first searches. But the searches are not limited by a simple depth-bound. Instead, a path in one of these depth-first searches is discontinued if its f value exceeds some cut-off value. In the first search, the cut-off is $h(n_0)$, the heuristic value of the start node. In subsequent searches, the cut-off is the lowest $f(n)$ for nodes, n, that were visited but not expanded in the previous search.

SMA^*, *simplified memory-bounded* A^*, places a size limit on the agenda. It discards the least-promising nodes from the agenda, if it needs to, in order to keep the agenda within the size limit. However, it keeps enough information to allow these discarded paths to be quickly re-generated should they ever be needed.

You can see from these two examples that a major focus of research into new search strategies is keeping the space requirements manageable, while retaining completeness and optimality.

Exercises (Past exam questions)

1. A^* search uses an evaluation function f:

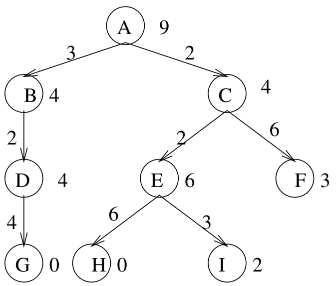
   $$f(n) \triangleq g(n) + h(n)$$

   where $g(n)$ is the cost of the path from the start node to node n and $h(n)$ is an estimate of the cost of the cheapest path from node n to a goal node.

   Breadth-first search and least-cost search are actually special cases of A^* search.

   - How could you define g and h so that the search carried out is actually a breadth-first search?
   - How could you define g and h so that the search carried out is actually a least-cost search.

2. Consider the following state space in which the states are shown as nodes labeled A through I. A is the initial state, and G and H are the goal states. The numbers alongside the edges represent the costs of moving between the states. To the right of every state is the estimated cost of the path from the state to the nearest goal.



Show how each of the following search strategies finds a solution in this state space by writing down, in order, the names of the nodes removed from the agenda. Assume the search halts when a goal state is removed from the agenda. (In some cases, multiple answers are possible. You need give only one such answer in each case.)

(a) Breadth-first;

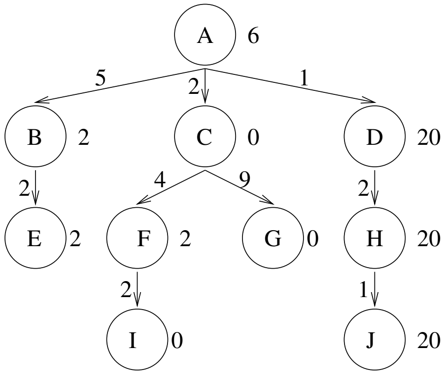
(b) Depth-first;

(c) Iterative-deepening;

(d) Least-cost search; and

(e) Heuristic search using $f(n) = g(n) + h(n)$ as the heuristic function, where $g(n)$ is the cost of the path to node n, and $h(n)$ is the estimated cost of the path from node n to the nearest goal.

3. Consider the following state space in which the states are shown as nodes labeled A through J. A is the initial state, and G and I are the goal states. The numbers alongside the edges represent the costs of moving between the states. To the right of every state is the estimated cost of the path from the state to the nearest goal.



Show how each of the following search strategies finds a solution in this state space by writing down, in order, the names of the nodes removed from the agenda. Assume the search halts when a goal state is removed from the agenda. (In some cases, multiple answers are possible. You need give only one such answer in each case.)

(a) Breadth-first;

(b) Depth-first;

(c) Iterative-deepening;

(d) Least-cost;

(e) Greedy search, i.e. heuristic search using $f(n) = h(n)$ as the heuristic function, where $h(n)$ is the estimated cost of the path from node n to the nearest goal; and

(f) Heuristic search using $f(n) = g(n) + h(n)$ as the heuristic function, where $g(n)$ is the cost of the path to node n, and $h(n)$ is as before.

4. Explain precisely and concisely why A^* search is complete and why it is optimal.

5. *Beam search* is a form of heuristic search using $f(n) = g(n) + h(n)$. However, it is parameterised by a positive integer k (much as depth-bounded search is parameterised by a positive integer l). Having computed the successors of a node, it only places onto the agenda the best k of those children. (The k with the lowest $f(n)$ values.)

(a) Give the worst-case time complexity of beam search. **Briefly** justify your answer.

(b) Give the worst-case space complexity of beam search. **Briefly** justify your answer.

(c) Beam search is *not* complete. Draw a small state space showing an admissible heuristic and a solution path, but give a value for k for which beam search on your state space would fail to find that solution path.

(d) Beam search is *not* optimal. Draw a small state space showing an admissible heuristic and more than one solution path, but give a value for k for which beam search on your state space would only find the more costly of the two solution paths.

6. A sliding tiles puzzle has room for seven tiles but contains only three black tiles (B) and three white tiles (W). The initial configuration is:

B	**B**	**W**		**B**	**W**	**W**

The goal configuration is:

B	**B**	**B**	**W**	**W**	**W**	

Tiles can move into the adjacent position if it is empty, with a cost of 1. They can also hop over *at most* two other tiles into an empty position, with a cost equal to the number of tiles hopped over (1 or 2).

(a) Give an iconic representation for the *states* of this puzzle.
Give the *initial state*.
Give the *goal state*.
Give the *operators*. (Extra marks will be given for precision.)

(b) One possible heuristic function, $h_1(n)$, for the sliding tiles puzzle described above is: if n is a goal state, then $h_1(n) = 0$; for all other states, $h_1(n) = 1$.

- What is the value of $h_1(n)$ applied to the initial configuration above?
- Is $h_1(n)$ admissible? Carefully justify your answer.

(c) Another possible heuristic function, $h_2(n)$, for the sliding tiles puzzle described above is: for each tile position in the current state, excluding the one that is empty, if its current contents do not equal its goal contents, then add one to the estimate.

- What is the value of $h_2(n)$ applied to the initial configuration above?
- Is $h_2(n)$ admissible? Carefully justify your answer.

(d) An AI student claims that $h_2(n)$ is a better heuristic function for this puzzle than $h_1(n)$. Do you agree? Carefully justify your answer.

7. Suppose we have an *admissible* heuristic function h for a state space. Also, for all states n in the state space, $h(n) \geq 0$ and all action costs are positive.

   Hence, state whether each of the following is *true* or *false*. To obtain credit, you must in each case **explain** your answer correctly and in detail.

   (a) If n is a goal state, then $h(n) = 0$.

   (b) If $h(n) = 0$, then n is a goal state.

   (c) If n is a 'dead-end' (i.e. it is a non-goal state from which a goal state canot be reached), then $h(n) = \infty$.

   (d) If $h(n) = \infty$, then n is a 'dead-end'.