Best First Search

- Now we have a heuristic, we can use it to direct our search towards the goal.
- Best first search simply chooses the unvisited node with the best heuristic value to visit next.
- It can be implemented in the same algorithm as lowest-cost Breadth First Search.
  - This time the priority of each node added to the queue is its heuristic value.

Example

- Loops through algorithm:
  1. L = {<Bi,390>}, V = {}.
  2. L = {<Li,280>, <Mn,280>, <Ca,500>, <Lo,530>, <Pl,620>}, V = {Bi}.
  3. L = {<Mn,280>, <Hu,310>, <Ca,500>, <Lo,530>, <Pl,620>}, V = {Bi, Li}.
  4. L = {<Ca,110>, <Hu,310>, <Ca,500>, <Lo,530>, <Pl,620>}, V = {Bi, Li, Ma}.
  5. L = {<Gl,70>, <Hu,310>, <Ne,150>, <Ca,500>, <Lo,530>, <Pl,620>}, V = {Bi, Li, Ma, Ca}.
  6. L = {<Ed,0>, <Ne,150>, <Hu,310>, <Ca,500>, <Lo,530>, <Pl,620>}, V = {Bi, Li, Ma, Ca, Gl}.

Best First Search Performance

- Best first works pretty well in this case (as long as Birmingham-Manchester gets tried before Birmingham-Liverpool-Manchester).
- However, it need not. If Hull had a heuristic of 270, it would find Birmingham-Hull-Newcastle-Edinburgh as the best route.
  - Overall that has a cost of 558 instead of 551 to go via Manchester.
  - In general, the worse the Heuristic is, the worse Best First search is likely to perform.
  - The problem is that Best First search doesn’t take into account the cost of getting to the node, only the cost to go from there.

A* Search

- A* Search is Best First search where the priority value in the queue is $f(x) = g(x) + h(x)$, where
  - $g(x)$ is the cost to get from the start state to $x$.
  - $h(x)$ is the heuristic cost to get from $x$ to the goal.
- Can think of it as combining lowest-cost breadth first search ($g(x)$) and Best First search ($h(x)$).
- Definition: A heuristic is admissible if it never overestimates the actual cost to the goal.
- Like lowest-cost BFS, A* search with an admissible heuristic is guaranteed to find the shortest path.

Which Search to Use

- If you have a good heuristic, obviously you want to use heuristic search.
  - But for some domains (as we’ll see later) good heuristics are hard to produce.
- If not, there are memory and time considerations.
  - BFS and the like are guaranteed to find short paths, but use a lot of memory and are slow.
  - DFS is much faster, but isn’t guaranteed to find a solution.
  - Even for heuristic search we sometimes just do the equivalent of DFS on the heuristic value.
  - This is known as greedy search.

Game Playing

- One common application of search is in games.
  - Chess computers (dumb ones) work using a heuristic to evaluate board positions, then search as many moves ahead as they can in the time available and use the heuristic to evaluate the final position.
  - Now however, we need to search not only over our moves, but also over the other player’s moves.
  - Zero-sum games are ones in which what you win is what the other player loses.
  - Search in zero-sum games involves choosing the best move for you in your turn, and the worst move for you in your opponent’s turn.
**MINIMAX Search**

- In MINIMAX search we maximise the heuristic in our turn, then minimise it in our opponent’s turn.

- The heuristic is used at the bottom of the tree.

- These values propagate up the tree via max and min.

**Alpha-beta Search**

- Alpha-beta search is like minimax, but it uses the fact that we don’t need to expand all the tree if we know it will never be as good/bad as something we’ve already seen.

- In the first case, we need not expand leaves after the 14, because it’s already more than 10 from the first subtree.

- In the second case, we need not expand after the 9 because it’s already worse than the first subtree.

**Example: Alpha-beta Search**

- This is the same search tree as for MINIMAX search.

- Now we only expand 15 nodes at the third level in the tree, compared with 27 for MINIMAX.

- Savings get even bigger the deeper the tree.

**Algorithm: Alpha-beta Search**

```python
Function max-value(s,a,b)
// s = current state, A to play, a = best score for A, b = best score for B
1. If s is a terminal state, return its heuristic value
2. Else for each s’ a child of s
   3. a = max(a, min-value(s’,a,b))
4. if (a >= b) return b
5. Return a

Function min-value(s,a,b)
1. If s is a terminal state return its heuristic value
2. Else for each s’ a child of s
   3. b = min(b, max-value(s’,a,b))
4. if (b <= a) return a
5. Return b
```

These are recursive: max-value calls min-value, which calls max-value again. This stops when we get to a terminal state (a winning position, or the maximum tree depth).

Algorithm based on the one in Russell and Norvig.

**Final Words**

- Search is the basis for a huge amount of AI.
  - Real-world applications include SAT-NAV systems, planning, fault diagnosis (conflict-directed best-first search), robot navigation (probabilistic roadmap search), chess computers, intelligent opponents for computer games, ...
  - Heuristics are the key to making search efficient.
  - Also key in lots of other parts of AI.
  - Finding good heuristics is an art in itself.
  - Many other AI techniques (genetic algorithms, ant colony optimisation, ...) use ways to do search without explicitly writing out a search space.

**What You Need to Know**

- All the search algorithms we’ve covered.
  - You may have to do examples by hand in the exam.
  - You don’t need to memorise the algorithms, just remember how they work.
  - Using stacks/queue/priority queues.
  - Heuristics and admissibility.