Informed/Heuristic Search

Arnav Gupta

October 18, 2024

Contents

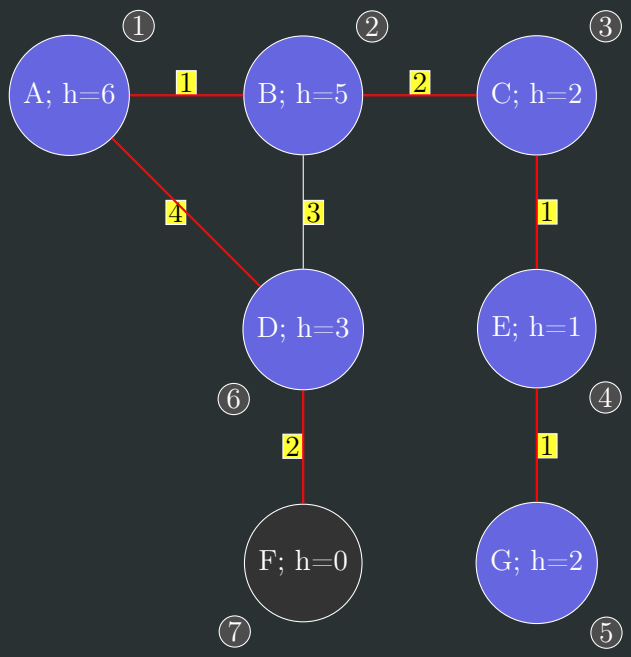
1 Heuristic Search

Use **heuristics** to guide the search towards the goals. $h(n)$ is an estimate of the cost of the shortest path from node n to a goal node. Computing the heuristic must be much easier than solving the problem.

$h(n)$ is an **underestimate** if there is no path from n to a goal that has path length less than $h(n)$.

1.1 Greedy Best-First Search

Select the path whose end is closest to a goal according to the heuristic function, that is minimal h value. Treat the frontier as a priority queue ordered by h.



1.1.1 Properties of GBFS

Properties:

- *space complexity*: exponential
- *time complexity*: exponential
- *completeness*: no, could be stuck in a cycle
- *optimality*: no, may return sub-optimal path first

1.2 Heuristic Depth-First Search

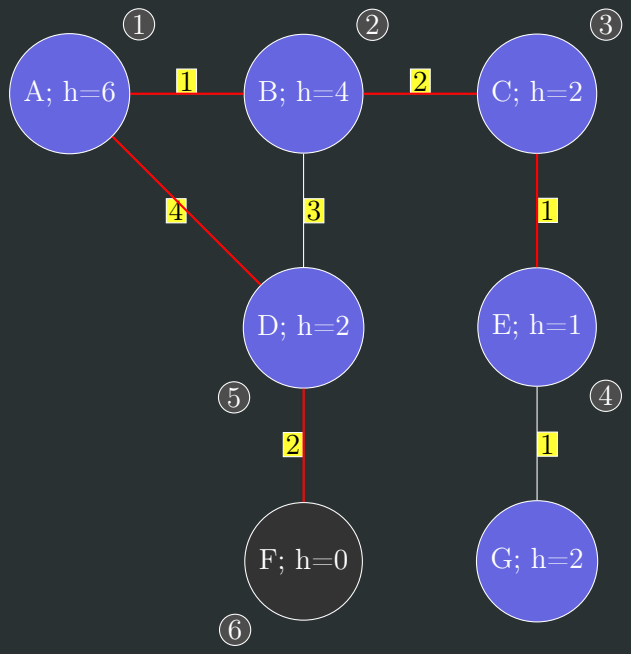
Do a DFS but add paths to the stack ordered by h.

Same properties as DFS.

2 A* Search

Use both path cost and heuristics. The sum of the path cost and heuristic from the end of the path to the goal is the **total path cost**.

Treats the frontier as a priority queue ordered by this sum. Always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.



2.1 Admissibility of A*

A* always finds an optimal solution as the first path to the goal, if:

- the branching factor is finite
- arc costs are bounded above 0
- $h(n)$ is a lower bound on the cost of the shortest path from n to a goal node

Admissible heuristics never overestimate the cost to the goal.

A* halts since the cost of the paths on the frontier keeps increasing and will eventually exceed any finite number.

To construct an admissible heuristic:

1. define a relaxed problem by simplifying or removing constraints on the original problem
2. solve the relaxed problem without search
3. the cost of the optimal solution to the relaxed problem is an admissible heuristic for the the original problem

Preferred heuristics have higher values and are very different for different states (should help in choosing which path to take).

2.1.1 Dominating Heuristic

Given heuristics $h_1(n)$ and $h_2(n)$, $h_2(n)$ dominates $h_1(n)$ if:

- $\forall n\ h_2(n) \geq h_1(n)$
- $\exists n\ h_2(n) > h_1(n)$

If $h_2(n)$ dominates $h_1(n)$, A* using h_2 will never expand more nodes than A* using $h_1(n)$.

2.2 Properties of A* Search

Properties:

- *space complexity*: exponential
- *time complexity*: exponential
- *completeness*: yes, with above assumptions
- *optimality*: yes, with above assumptions

A* is optimally efficient: no other algorithm with the same start node and same heuristic can find the optimal path to the goal and expand fewer nodes. This is because any algorithm that does not expand all nodes $f(n) < \text{cost}(s, g)$ run the risk of missing the optimal solution.

2.3 Multi-Path Pruning and A*

With A*, it is possible that a subsequent path to some node is shorter than the first path to it. To avoid this, ensure the heuristic is monotone.

A heuristic h is **monotone** is $h(m) - h(n) \leq \text{cost}(m, n)$ for every arc $\langle m, n \rangle$. This ensures the heuristic estimate of the path cost between any two adjacent nodes is always less than the actual cost.

If h satisfies the monotone restriction, A* with multi-path pruning always finds the shortest path to a goal.

3 Adversarial Search

Find the best option for the player on the nodes they control (MAX nodes). Assume competitor takes options worst for the player (MIN nodes). Recursively search to leaf nodes to find state evaluations and percolate values upward through the tree.

Algorithm 1 Minimax Algorithm

```
1:  function MINIMAX(node, depth, isMax)
2:      if depth = 0 or node is a terminal node then
3:          return the heuristic value of node
4:      end if
5:      if isMax then
6:          bestValue ← −∞
7:          for each child of node do
8:              v ← MINIMAX(child, depth - 1, False)
9:              bestValue ← max(bestValue, v)
10:         end for
11:     else
12:         bestValue ← +∞
13:         for each child of node do
14:             v ← MINIMAX(child, depth - 1, True)
15:             bestValue ← min(bestValue, v)
16:         end for
17:     end if
18:     return bestValue
19: end function
```

Alpha-beta pruning: method that allows ignoring portions of the search tree without losing optimality (useful in practice but does not change worst-case)

Can stop search early at non-leaf nodes via heuristics, but optimality no longer guaranteed.

3.1 Bidirectional Search

Searching is **symmetric**: it is the same to find path from start nodes to goal node or goal node to start node.

Forward branching factor: number of arcs out of a node

Backward branching factor: number of arcs into a node

Search complexity is b^n, so should use forward search if forward branching factor is less than backward branching factor. (not possible if graph is dynamically constructed)

Can search simultaneously backwards from goal and forward from start, which can result exponential saving in time/space.

In bidirectional search, frontiers must meet. Often done with one breadth-first method that builds a set of locations to the goal and the other direction using another method to find paths to these locations.

3.2 Island Driven Search

Find a set of islands between the start and goal, which gives smaller problems to solve.

This reduces the time complexity to $mb^{k/m}$ when $m-1$ islands are used (m smaller problems to solve).

Difficult to guarantee optimality when identifying islands the path must pass through.