Introduction to Artificial Intelligence Minimax

Tree Search

A tree structure is a hierarchy of linked nodes where each node represents a particular state. Nodes have none, one or more child nodes. A solution is a path from the "root" node (representing the initial state) to a "goal" node (representing the desired state). Tree search algorithms attempt to find a solution by traversing the tree structure - starting at the root node and examining (expanding) the child nodes in a systematic way.

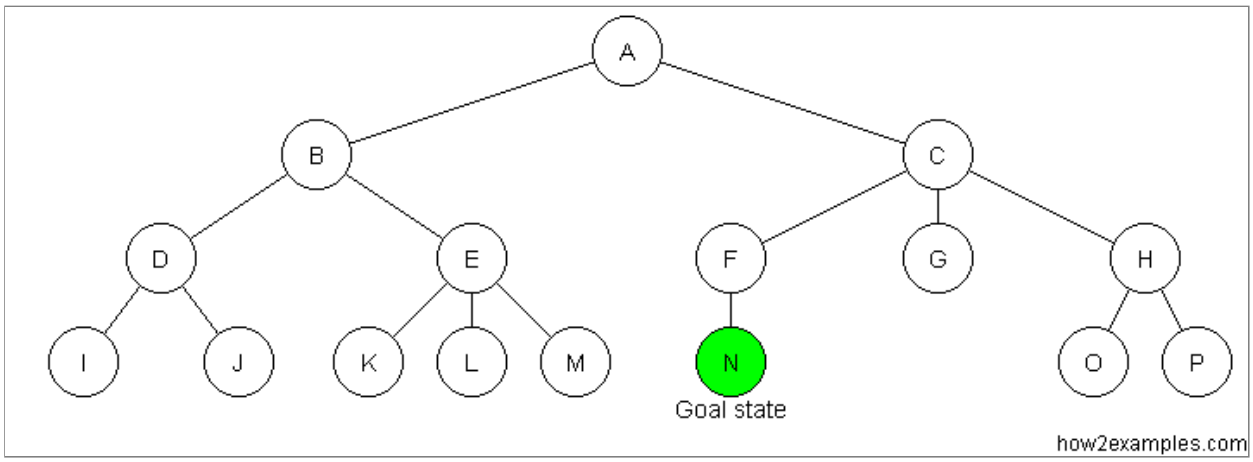
Tree search algorithms differ by the order in which nodes are traversed and can be classified into two main groups:

- *Blind search* algorithms (e.g. "Breadth-first" and "Depth-first") use a fixed strategy to methodically traverse the search tree. Blind search is not suitable for complex problems as the the large search space (number of different possible states to search) makes them impractical given time and memory constraints.
- *Best-first search* algorithms (e.g. "Greedy" and "A*") use a heuristic function to determine the order in which nodes are traversed, giving preference to states that are judged to be most likely to reach the required goal. Using a "heuristic" search strategy reduces the search space to a more manageable size.

A search strategy is *complete* if it is guaranteed to find a solution if one exists. A search strategy is *optimal* if it is guaranteed to find the best solution when several solutions exists.

Breadth-First Search

Breadth-first search starts at the root of the tree and examines all nodes at the same level before examining nodes at the next level.



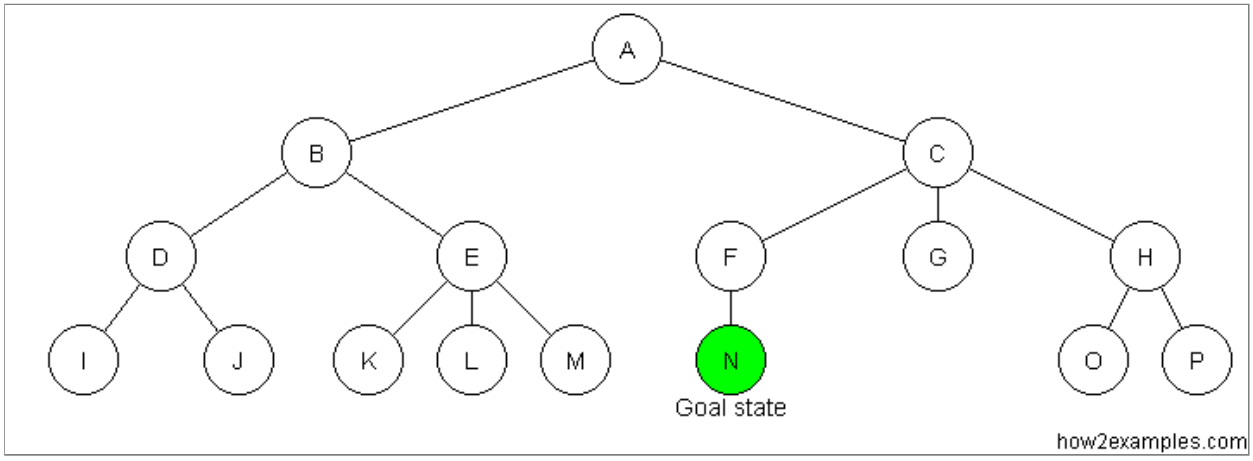
Example of Breadth First Search

As breadth-first search exhaustively examines every node at a particular depth before progressing to the next level, it is guaranteed to find the solution, if one exists, with the shortest path from the initial state. A disadvantage of breadth-first search is that it can have a high memory requirement - as a record needs to be maintained of every expanded node.

Depth-First Search

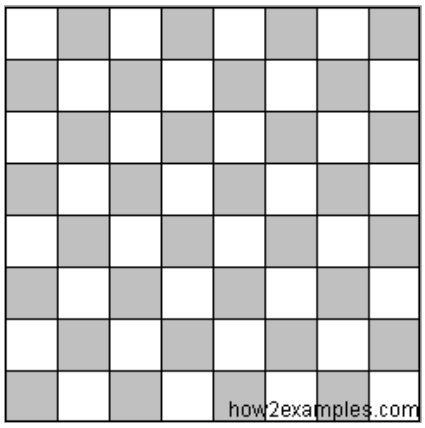
Depth-first search starts at the root node and continues down a particular path (branch) - selecting a child node at the deepest level of the tree to expand next. Only when the search hits a dead end (a node that has no child

nodes) does the search "backtrack" - continuing the search from the last node it encountered whose child nodes have not been fully examined.



Example of Depth First Search

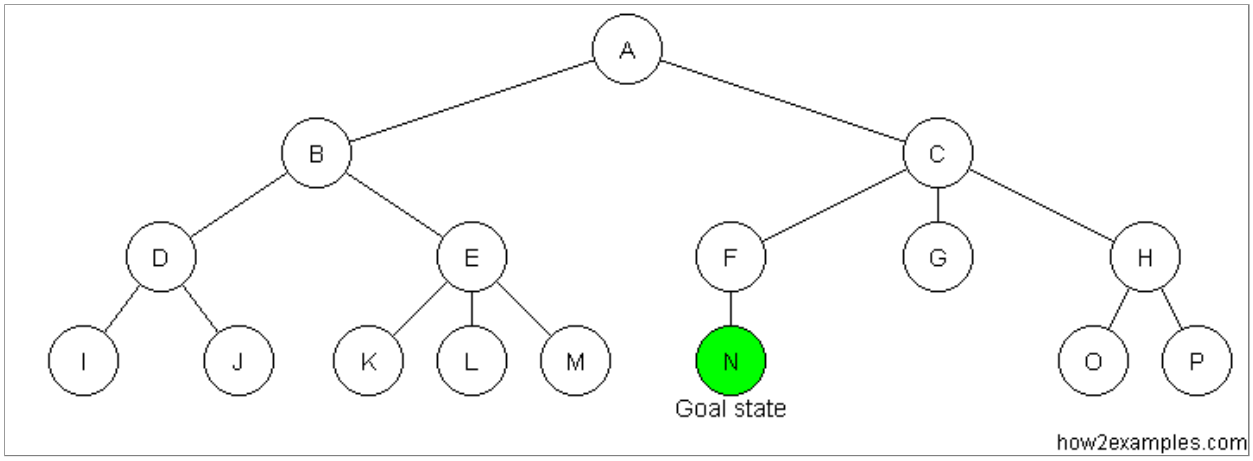
Unlike breadth-first search, depth-first search is not guaranteed to find the solution with the shortest path. As it is possible for depth-first search to proceed down an infinitely long branch, without ever returning to explore other branches, there is no guarantee that depth-first search will ever find a solution, even when one exists. The memory requirements of depth-first search are more modest than breadth-first search. Only a single path from the root node to the current node, plus any unexpanded nodes on the path, need to be stored.



Example of solving the Eight Queens puzzle using Depth First Search

Iterative Deepening Depth-First Search

Iterative deepening depth-first search (IDDFS) operates like depth-first search - apart from that the algorithm imposes a limit on how deep the search traverses. Until a goal state is found, the search is repeated with an increased depth limit.

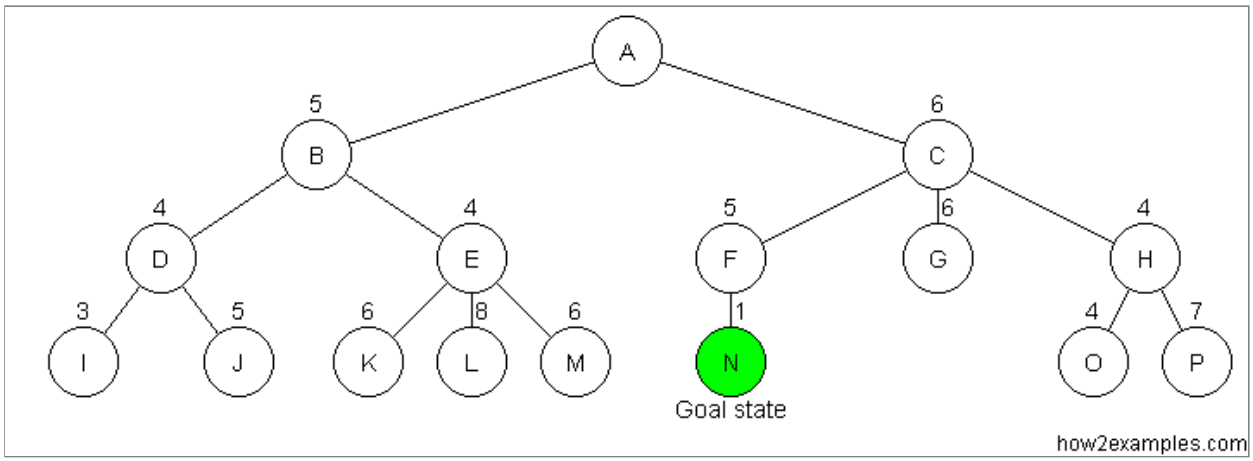


Example of Iterative Deepening Depth First Search

Iterative deepening depth-first search combines advantages of both breadth-first and depth-first search. By continuously incrementing the depth limit by one until a solution is found, iterative deepening depth-first search has the same strength as breadth-first search regarding always finding the shortest path to a solution. By using a depth-first approach on every iteration, iterative deepening depth-first avoids the memory cost of breadth-first search.

Greedy Search

Nodes are evaluated using a heuristic function. The heuristic function estimates how close a node is to the goal state. The sequence in which nodes are traversed is ordered, with the nodes considered closest to the goal state being expanded first.

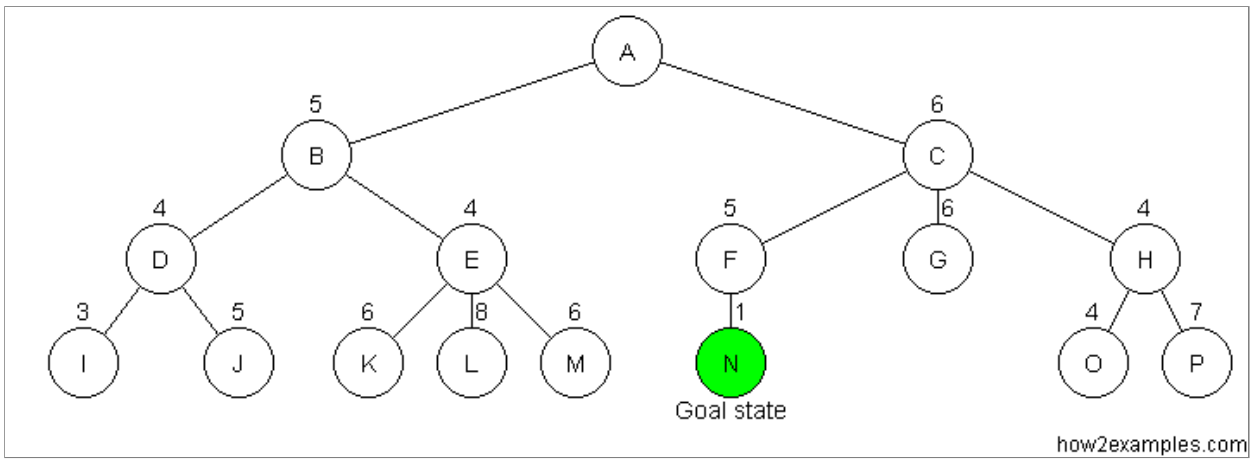


Example of Greedy Search

Like depth-first search, greedy search is not complete. Greedy search is not guaranteed to find the solution with the shortest path. It is possible for greedy search to proceed down an infinitely long branch without finding a solution, even when one exists.

A* Search

A* (A star) is a search strategy used for finding an efficient path between two points (represented as nodes in the tree structure). Like greedy search, a heuristic function is used to guide the order in which nodes are expanded. Unlike greedy search, with A* the heuristic function also takes into account the existing cost from the starting point to the current node. The "cost" is calculated as the sum of a) the cost from the starting point to the current node and b) the estimate of how close the current node is to the goal state.

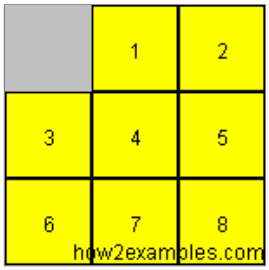


Example of A* Search

Like breadth-first search, A* search is complete - it will always find a solution if one exists. For A* search to be optimal it must be used with an *admissibile heuristic*. An admissibile heuristic, also known as an optimistic

heuristic, never overestimates the cost of reaching the goal.

When A* search reaches a goal state it has found a solution with a total cost less than or equal to the estimated cost of any unsearched paths. If the estimated costs are optimistic then the true cost of any solutions discovered by traversing the unsearched paths are guaranteed to be no better than solution already found.



Example of solving a sliding tiles puzzle using A* Search

A disadvantage of A* search is that, as it needs to maintain a list of unsearched nodes, it can require large amounts of memory. Variations of A* that require less memory include Iterative Deepening A* (IDA*) and Simplified Memory Bounded A* (SMA*).

Introduction to Artificial Intelligence Minimax