SKILRY

POPULAR SEARCH ALGORITHMS

Single Agent PathfindingProblems

The games such as 3X3 eight-tile, 4X4 fifteen-tile, and 5X5 twenty four tile puzzles are singleagent-path-finding challenges. They consist of a matrix of tiles with a blank tile. The player is required to arrange the tiles by sliding a tile either vertically or horizontally into a blank space with the aim of accomplishing some objective.

The other examples of single agent pathfinding problems are Travelling Salesman Problem, Rubik's Cube, and Theorem Proving.We delve into the realm of single-agent pathfinding problems, a fundamental concept in the field of artificial intelligence (AI). Pathfinding involves finding the optimal route from a starting point to a destination while navigating through obstacles. Single agent pathfinding refers to scenarios where there's only one agent traversing the environment, which is a common scenario in many real-world applications.

Search Terminology

- Problem Space - It is the environment in which the search takes place. (A set of states and set of operators to change those states)
- Problem Instance - It is Initial state &plus Goal state.
- Problem Space Graph - It represents problem state. States are shown by nodes and operators are shown by edges.
- Depth of a problem - Length of a shortest path or shortest sequence of operators from Initial State to goal state.
- Space Complexity - The maximum number of nodes that are stored in memory.
- Admissibility - A property of an algorithm to always find an optimal solution.
- Time Complexity - The maximum number of nodes that are created.
- Branching Factor - The average number of child nodes in the problem space graph.
- Depth - Length of the shortest path from initial state to goal state.

Brute-Force Search Strategies

They are most simple, as they do not need any domain-specific knowledge. They work fine with small number of possible states

Requirements –

- State description
- A set of valid operators
- Initial state
- Goal state description

Breadth-First Search

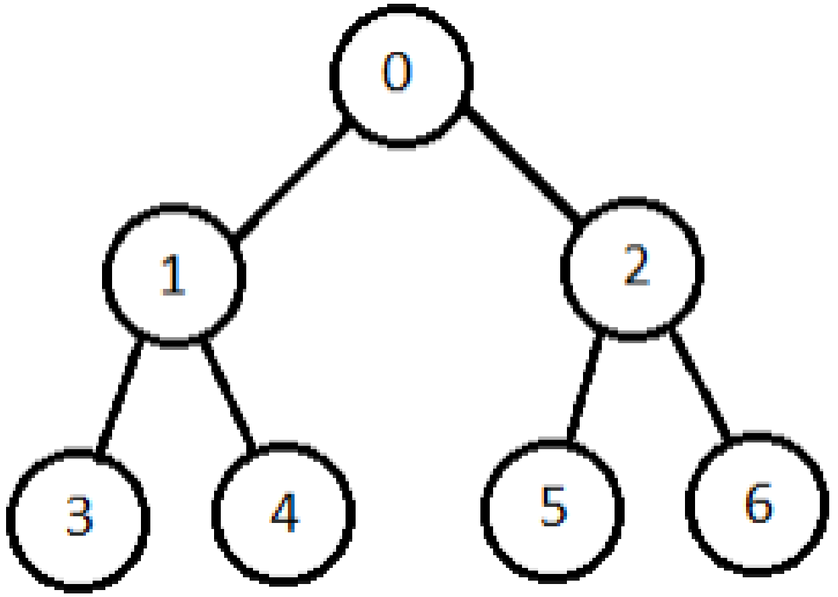
It starts from the root node, explores the neighboring nodes first and moves towards the next level neighbors. It generates one tree at a time until the solution is found. It can be implemented using FIFO queue data structure. This method provides shortest path to the solution.

If branching factor (average number of child nodes for a given node) = b and depth = d, then number of nodes at level d = bd.

The total no of nodes created in worst case is b + b2 + b3 + … + bd.

Disadvantage – Since each level of nodes is saved for creating next one, it consumes a lot of memory space. Space requirement to store nodes is exponential.

Its complexity depends on the number of nodes. It can check duplicate nodes.



- ## Mechanism of Breadth-First Search

  In BFS, nodes are traversed level by level, starting from the initial state. The algorithm expands all the neighboring nodes of the current state before moving on to the next level. This process continues until the goal state is reached or all reachable nodes have been explored.

- ## Data Structures in Breadth-First Search

  BFS typically utilizes a queue data structure to maintain the frontier of nodes to be explored. Nodes are added to the queue as they are discovered, and they are removed in a first-in-first-out (FIFO) fashion. Additionally, a set or hash table is often used to keep track of visited nodes to prevent revisiting already explored states.

- ## Completeness and Optimality of Breadth-First Search

  BFS is both complete and optimal for finding the shortest path in a graph with uniform edge costs. It guarantees that the shortest path to a goal state is found if one exists. However, BFS may be impractical for large search spaces due to its memory requirements and time complexity.

- ## Applications of Breadth-First Search

  BFS has widespread applications beyond pathfinding, including network traversal, web crawling, and puzzle solving. In pathfinding scenarios, BFS is particularly useful for finding the shortest path in unweighted graphs or when the goal is to explore all reachable states within a certain radius.

- ## Limitations and Considerations

  While BFS guarantees optimality and completeness, it may not be suitable for certain scenarios. In graphs with branching factors that grow exponentially, BFS may consume significant memory and time resources. Additionally, BFS may not perform well in scenarios where the cost of moving between states is not uniform.

- ## Optimizations and Variants

  Several optimizations and variants of BFS exist to address its limitations. Techniques such as iterative deepening BFS, bidirectional BFS, and uniform-cost search modify the basic BFS algorithm to improve efficiency or address specific requirements of the problem domain.

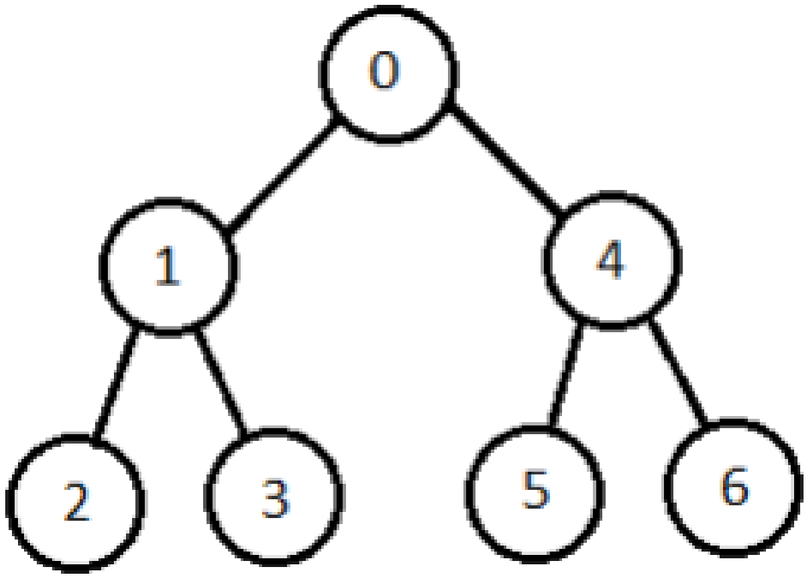
Depth-First Search

It is implemented in recursion with LIFO stack data structure. It creates the same set of nodes as Breadth-First method, only in the different order.

As the nodes on the single path are stored in each iteration from root to leaf node, the space requirement to store nodes is linear. With branching factor b and depth as m, the storage space is bm.

Disadvantage - This algorithm may not terminate and go on infinitely on one path. The solution to this issue is to choose a cut-off depth. If the ideal cut-off is d, and if chosen cut-off is lesser than d, then this algorithm may fail. If chosen cut-off is more than d, then execution time increases.

Its complexity depends on the number of paths. It cannot check duplicate nodes.



- **Key Characteristics**

  - **Completeness:** DFS is not guaranteed to find a solution if one exists, especially if the graph is infinite or has cycles. However, it is complete in finite graphs.

  - **Space Complexity:** The space complexity of DFS is proportional to the depth of the search tree. In the worst-case scenario, where the graph is a linear path, the space complexity can be O(n), where n is the number of nodes.

  - **Time Complexity:** The time complexity of DFS is O(V + E), where V is the number of vertices (nodes) and E is the number of edges in the graph.

  - **Applications:** DFS is commonly used in maze-solving, topological sorting, finding connected components, and solving puzzles.

- # Advantages and Disadvantages

  - ## Advantages:

  - Simplicity: DFS is relatively easy to implement.
  - Memory Efficiency: It requires less memory compared to breadth-first search (BFS) as it only stores nodes along the current path.
  - Space Efficiency: DFS typically requires less space compared to BFS, making it suitable for large graphs.

  - ## Disadvantages:

  - Completeness: It may get stuck in infinite loops if the graph contains cycles, making it incomplete for certain types of graphs.
  - Suboptimal Solutions: DFS does not necessarily find the shortest path between two nodes. It may find a solution, but not the most optimal one.

- # Bidirectional Search

It searches forward from initial state and backward from goal state till both meet to identify a common state.
The path from initial state is concatenated with the inverse path from the goal state.
Each search is done only up to half of the total path.
Bidirectional Search is a pathfinding algorithm that operates by simultaneously exploring the search space from both the start and goal nodes towards each other. By conducting searches from both ends of the problem space, Bidirectional Search aims to converge towards a common node, ultimately finding the shortest path between the start and goal nodes more efficiently than traditional algorithms. This approach is particularly beneficial in scenarios where the search space is large, and the branching factor is high, as it reduces the overall search time by exploring from both ends simultaneously. Bidirectional Search typically utilizes standard search algorithms like Breadth-First Search or Depth-First Search, with the termination condition being the intersection of paths from both directions. However, it requires careful consideration of data structures and termination conditions to ensure efficiency and completeness. Overall, Bidirectional Search offers a promising approach to pathfinding, especially in situations where computational resources are limited, and finding the shortest path quickly is paramount.

- ## Uniform Cost Search

Sorting is done in increasing cost of the path to a node. It always expands the least cost node. It is identical to Breadth First search if each transition has the same cost. It explores paths in the increasing order of cost.

Disadvantage - There can be multiple long paths with the cost ≤ C*. Uniform Cost search must explore them all.

- ## Iterative Deepening Depth-First Search

It performs depth-first search to level 1, starts over, executes a complete depth-first search to level 2, and continues in such way till the solution is found.

It never creates a node until all lower nodes are generated. It only saves a stack of nodes. The algorithm ends when it finds a solution at depth d. The number of nodes created at depth d is bd and at depth d-1 is bd-1.

Informed (Heuristic)SearchStrategies

Informed search strategies, also known as heuristic search, utilize domain-specific knowledge to guide the search process towards the goal more efficiently. Unlike uninformed search algorithms, which lack knowledge about the problem domain, informed search algorithms incorporate heuristic functions that estimate the cost or distance from the current state to the goal. These heuristic functions provide valuable information to prioritize the exploration of more promising paths, potentially leading to faster convergence and optimal solutions.

Popular examples of informed search algorithms include A* Search, Greedy Best-First Search, and Iterative Deepening A* (IDA*). A* Search, in particular, combines the advantages of both breadth-first and depth-first search by using a heuristic function to evaluate the cost of reaching the goal from each node. While informed search strategies generally require more computational resources to compute and store heuristic information, they often outperform uninformed search algorithms, especially in large and complex problem spaces. Understanding and implementing informed search strategies are essential for efficiently solving pathfinding and optimization problems in various domains, including robotics, logistics, and game AI.

Heuristic Evaluation Functions

Heuristic evaluation functions serve as informed estimates of the cost or distance to reach the goal in pathfinding algorithms. These functions leverage domain-specific knowledge to guide search algorithms towards the goal state efficiently. By estimating the remaining cost or distance from a given state to the goal, heuristic evaluation functions enable informed decision-making in algorithms like A* Search. The effectiveness of these functions heavily relies on their accuracy in approximating the true cost, balancing between computational efficiency and solution optimality.

Pure Heuristic Search

It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes. In each iteration, a node with a minimum heuristic value is expanded, all its child nodes are created and placed in the closed list. Then, the heuristic function is applied to the child nodes and they are placed in the open list according to their heuristic value. The shorter paths are saved and the longer ones are disposed.

Local Search Algorithms

They start from a prospective solution and then move to a neighboring solution. They can return a valid solution even if it is interrupted at any time before they end.

Local search algorithms are a class of optimization techniques that iteratively explore the solution space by making incremental modifications to a current solution, aiming to improve its quality. Unlike global search algorithms, which explore the entire search space, local search algorithms focus on finding satisfactory solutions within a limited neighborhood of the current solution. Examples of local search algorithms include Hill Climbing, Simulated Annealing, Genetic Algorithms, and Tabu Search. These algorithms are particularly effective for optimization problems with large and complex solution spaces, where finding an optimal solution is computationally expensive or impractical. Local search algorithms trade-off between exploration and exploitation, often providing efficient solutions for real-world optimization challenges across various domains, including scheduling, routing, and machine learning.

Hill-Climbing Search

Hill-Climbing Search is a local search algorithm used in optimization problems to iteratively improve a solution by moving towards the best neighboring solution. The algorithm starts with an initial solution and iteratively evaluates neighboring solutions, selecting the one that maximally improves the objective function. This process continues until a local maximum is reached, where no neighboring solution provides a better outcome.

One of the key characteristics of Hill-Climbing is its simplicity, making it easy to implement and understand. It requires minimal memory overhead as it only maintains the current solution and its neighbors. However, its simplicity comes with limitations. Hill-Climbing is prone to getting stuck in local optima, where the best solution in the vicinity is not the global optimum. Additionally, it does not provide any mechanism for escaping local optima, leading to the possibility of premature convergence.

Despite its limitations, Hill-Climbing remains valuable in certain scenarios, especially when computational resources are limited, and quick but suboptimal solutions are acceptable. Variants of Hill-Climbing, such as Simulated Annealing and Genetic Algorithms, address some of its shortcomings by introducing mechanisms for exploring the search space more effectively. Understanding Hill-Climbing provides a foundational understanding of local search algorithms and their applications in optimization problems.

Local Beam Search

Local Beam Search is a heuristic search algorithm used for solving optimization problems, particularly in the context of pathfinding and combinatorial optimization. Unlike traditional beam search, which explores a single path, local beam search maintains multiple states, known as beams, at each iteration.

At the beginning of the search, it randomly generates a set of initial states, called the beam. It then evaluates these states using a heuristic function and selects the top k states as candidates for the next iteration. This process continues iteratively, with each iteration potentially generating new states or refining existing ones.

One key feature of local beam search is its ability to maintain diversity among the beams, allowing it to explore multiple promising paths simultaneously. However, this also means that it may get stuck in local optima if the search space is not sufficiently explored.

Local beam search is particularly suitable for problems where the search space is large and it is desirable to explore multiple potential solutions concurrently. It has applications in various domains, including scheduling, routing, and machine learning. Despite its limitations, local beam search remains a valuable technique for addressing optimization problems efficiently.