Chao 3

DFS

Depth-first search (DFS) is a traversal algorithm used for searching or traversing through a graph or tree data structure. The DFS algorithm starts from a root node and explores as far as possible along each branch before backtracking.

The algorithm works as follows:

1. Start from the root node.
2. Visit the root node and mark it as visited.
3. Explore each adjacent node that has not been visited.
4. If an adjacent node has not been visited, repeat step 2 and step 3 on that node.
5. If all adjacent nodes have been visited, backtrack to the previous node and repeat step 3 and step 4 for the next adjacent node.

DFS can be implemented using a recursive or iterative approach. In the recursive approach, the function calls itself to traverse each unvisited node. In the iterative approach, a stack data structure is used to keep track of

the nodes to be visited, and the algorithm continues until all nodes have been visited.

BFS

Breadth-first search (BFS) is a graph traversal algorithm that visits all the nodes of a graph in breadth-first order, i.e., it visits all the nodes at the same level before moving on to the nodes at the next level.

BFS starts at the root node (or any arbitrary node) of the graph and visits all the nodes that are at distance one from the root node. Then, it moves on to visit all the nodes that are at distance two from the root node, and so on, until it has visited all the nodes of the graph.

BFS uses a queue to keep track of the nodes to visit. The algorithm starts by adding the root node to the queue. Then, it removes the first node from the queue and visits all its neighbors. The neighbors that have not been visited yet are added to the queue. This process continues until the queue is empty.

Depth Limited Search

Depth-limited search is a search algorithm that is similar to depth-first search, but limits the depth of the search to a specified level. The algorithm starts at the root node of the

search tree and explores all its children before moving on to the next level of nodes.

The depth-limited search algorithm continues this process until it reaches the specified depth limit or until it finds the goal state. If the algorithm reaches the depth limit without finding the goal state, it backtracks to the last node with unexplored children and continues the search from there.

Depth-limited search is useful when the search space is very large, and depth-first search would take too long to explore the entire search tree. By limiting the depth of the search, the algorithm can avoid exploring parts of the search space that are unlikely to lead to the goal state.

However, a disadvantage of depth-limited search is that it may miss the goal state if it is deeper than the specified depth limit. To address this issue, iterative deepening depth-first search can be used, which performs a series of depth-limited searches with increasing depth limits until the goal state is found.

Iterative Deeping

Iterative deepening depth-first search (IDDFS) is a variant of depth-first search (DFS) where the search is done in a series of depth-limited searches, each one increasing the

depth limit of the previous one. It is a combination of depth-first search and breadth-first search algorithms.

The idea behind IDDFS is to gradually increase the depth limit until a solution is found. This means that the algorithm will first search the nodes at a depth of 1, then at a depth of 2, then at a depth of 3, and so on until the goal node is found.

The advantage of IDDFS over other search algorithms is that it uses much less memory than breadth-first search, while still being guaranteed to find the optimal solution in a finite tree. However, IDDFS is slower than breadth-first search and A* search, especially for large search spaces, because it repeatedly searches the same nodes at different depths.

Bideractional Seach

Bidirectional search is a graph search algorithm that simultaneously searches from the start and end nodes towards each other. The two searches meet somewhere in the middle, hopefully finding a path connecting the two nodes. It can be faster than a regular breadth-first or depth-first search when searching for a path between two nodes in large graphs.

The basic idea is to use two search algorithms, one starting from the start node and the other from the goal node. Both searches use the same algorithm, but with reversed edges. The algorithm stops when the two searches meet in the middle, which is equivalent to finding a path between the start and goal nodes.

The advantage of bidirectional search is that it can reduce the search space by searching from both ends towards the middle, which can be faster than searching from one end to the other. However, it can also require more memory, as both searches need to be stored in memory simultaneously. Additionally, it is not always possible to use bidirectional search, for example if the graph is not well-connected or if the start and goal nodes are not known in advance.

Greedy best first search

Greedy Best-First Search is an informed search algorithm that expands nodes based on their heuristic value or estimated cost to the goal. At each step, it selects the node that appears to be closest to the goal, as determined by the heuristic function. This makes it a "greedy" algorithm, as it always makes the locally optimal choice at each step.

The algorithm can be described as follows:

1. Initialize the queue with the start node.
2. While the queue is not empty:
    - Dequeue the node with the lowest heuristic value.
    - If the dequeued node is the goal node, return success.
    - Otherwise, generate the successors of the dequeued node and compute their heuristic values.
    - Enqueue the successors with their heuristic values.
3. Return failure if the queue becomes empty.

Greedy Best-First Search is not guaranteed to find the optimal solution, as it may get stuck in a local minimum. However, it can be very efficient in practice, especially if the heuristic function is well-designed and provides accurate estimates of the distance to the goal.

NOTE: uses a heuristic function and we need to chose the lowest one

A*

is a popular pathfinding algorithm that is widely used in computer games, robotics, and other applications that require finding the shortest path between two points on a graph or a grid.

The algorithm works by evaluating each possible next step based on two values: the cost of getting from the starting node to that node (known as the "g-value"), and an estimate of the cost of getting from that node to the destination node (known as the "h-value"). The total cost of a node is calculated as the sum of its g-value and h-value.

At each step, the algorithm considers the node with the lowest total cost (i.e., the lowest f-value) as the next node to explore. The algorithm then evaluates all the nodes that are reachable from that node and calculates their f-values. It adds those nodes to the list of nodes to be considered for the next step.

The algorithm continues until it reaches the destination node or until there are no more nodes to consider. Once the destination node is reached, the algorithm retraces its steps back to the starting node to determine the shortest path.

A *is considered an improvement over other search algorithms, such as Dijkstra's algorithm, because it is more efficient in terms of the number of nodes it examines. By using a heuristic function to estimate the cost of getting from a node to the destination, A* can prioritize nodes that are more likely to lead to a solution and avoid examining unnecessary nodes.

NOTES: A* uses a cost function and a heuristic which is a cost function to the goal

Important Notes

If you think that you can improve this documentation feel free to send a pull request or to send me an email, :D