BFS

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It is implemented using a queue data-structure.

Algorithm Steps

1. Initialization:

- Start with the initial node and add it to the queue.
- Initialize an empty set or list for visited nodes.

2. Processing Nodes:

- While there are nodes in the queue:
  - **Dequeue** the front node from the queue.
  - **Mark** this node as visited (add it to the visited set).
  - **Check** if this node is the goal. If it is, return the path to the goal from the root.
  - If the node is not the goal state, **Expand** the node by generating its successors (neighbors).
  - For each successor (neighbor):
    - **Check** if the successor is not in the visited set.
    - If it's not visited, **enqueue** the successor to the queue and mark it as visited. This prevents the algorithm from reprocessing nodes and helps maintain BFS's level-wise exploration.
  - The queue is used to manage the nodes to be processed in the correct order, ensuring that nodes are explored level by level.
  - Repeat until the goal state is found or the queue is empty.

3. Completion:

- If the queue is empty and the goal is not found, return failure.

Prevent Infinite Loops

In graphs with cycles, marking nodes as visited only after they are dequeued prevents the algorithm from getting stuck in an infinite loop by revisiting nodes that have already been processed.

Advantages

Breadth-first search (BFS) is an algorithm used in artificial intelligence to explore a search space systematically. Some advantages of BFS include the following:

Completeness

BFS is guaranteed to find the goal state if it exists in the search space, provided the branching factor is finite.

Simplicity:

BFS is easy to understand and implement, making it a good baseline algorithm for more complex search algorithms.

No redundant paths:

BFS does not explore redundant paths because it explores all nodes at the same depth before moving on to deeper levels.

Disadvantages

Memory-intensive:

BFS can be memory-intensive for large search spaces because it stores all the nodes at each level in the queue.

Time-intensive:

BFS can be time-intensive for search spaces with a high branching factor because it needs to explore many nodes before finding the goal state.

Inefficient for deep search spaces:

BFS can be inefficient for search spaces with a deep depth because it needs to explore all nodes at each depth before moving on to the next level.

Not provide optimal solution:

BFS is considered an uninformed search (also known as blind search) because it does not use any domain-specific knowledge or heuristics to guide its search. It only uses the information provided by the problem's structure (i.e., the adjacency relationships in the graph).

BFS is guaranteed to find the shortest path (minimum number of edges) from the start node to the goal node in an unweighted graph. However, BFS does not always find the optimal solution in terms of path cost if the edges have different weights.

Time and Space complexities:

If (b) is the branching factor (the average number of child nodes for each node) and (d) is the depth of the shallowest goal node, then

Note: *As per text-book reference mentioned in last section, in algorithm, we are not generating one additional depth. Hence our depth is not (d+1), its just (d).*

Time Complexity is (O(b^d))

This is because BFS explores all nodes level by level, and in the worst case, it may need to explore all nodes at each level up to the depth (d).

Space Complexity is also (O(b^d))

This is because BFS needs to store all nodes at the current depth level in the frontier (queue), which can be as large as (b^d) in the worst case.

Summary:

Time Complexity: (O(b^d)) Space Complexity: (O(b^d)) BFS can be very memory-intensive because it needs to store all nodes at the current depth level, which can grow exponentially with the depth of the search tree.

RBFS

RBFS is a variant of the best-first search that uses a recursive approach to handle the search process, keeping memory usage low by only storing the current path from the root to the leaf and a limited amount of information about alternative paths. It combines the depth-first search's space efficiency with the heuristic guidance of best-first search.

Algorithm Steps:

1. Initialization:

- Start with the initial state as the root node.
- Use a priority queue (or similar structure) to manage nodes based on their f-cost (a combination of the cost to reach the node and the estimated cost to the goal), often denoted as f(n) = g(n) + h(n).

2. Recursive Search:

- The algorithm recursively explores the most promising node (the node with the lowest f-cost) first. When expanding a node, generate its successors.
- Calculate the f-cost for each successor.

- If a successor node appears to be promising (i.e., its f-cost is lower than a threshold), recursively apply RBFS to that successor.
- If a successor is less promising, store its f-cost and return to consider other nodes.

Steps Involved in the Comparison

1. Expand the Current Node:
    - Generate the successors of the current node.
    - Calculate the f-cost for each successor, where f(n)=g(n)+h(n).
2. Sort Successors by f-Cost:
    - Sort the list of successors based on their f-costs in ascending order. The successor with the lowest f-cost is considered the most promising path.
3. Set the Initial Threshold:
    - The initial threshold is set to the f-cost of the current node.
4. Select the Best and Second Best Alternatives:
    - Identify the best successor (the one with the lowest f-cost) and the second-best successor (the one with the second-lowest f-cost).
5. Recursive Call on the Best Successor
    - Recursively apply RBFS to the best successor. This involves passing the current threshold and the minimum of the successor threshold and the f-cost of the second-best successor as the new threshold.
6. Comparison Before Goal Test:
    - Before performing the goal test on the best successor, compare its f-cost with the threshold i.e. upper bound keeps track of the f-value of the best alternative path available form any ancestor of the current node
    - The upper bound keeps track of the f-value of the best alternative path available form any ancestor of the current node.If the f-cost of the best successor exceeds this threshold during the recursive call, RBFS prunes this path and backtracks to explore the next best alternative.
    - If the f-cost of the best successor is within the threshold, proceed with the goal test.
7. Updating Threshold:
    - The threshold is dynamically updated to the minimum f-cost of the best alternative path when backtracking.
    - This allows the algorithm to reconsider previously pruned paths if they become more promising.
    - The upper bound allows the algorithm to choose better paths rather than continuing indefinitely down the current path.
8. Completion:
    - The search continues recursively until the goal state is found or all paths have been explored without finding the goal.
    - When the goal state is found, the path to the goal is returned.

- If no solution is found, the algorithm returns failure.

Time and Space complexities:

If (b) is the branching factor (the average number of child nodes for each node) and (d) is the depth of the shallowest goal node, then

Time Complexity is (O(b^d))

This is because, in the worst case, RBFS may need to explore all nodes at each level of the search tree up to the depth (d).

Space Complexity is also (O(bd))

This is because RBFS only needs to store the current path from the root to a leaf node, along with the siblings of each node on the path. This results in a linear space complexity with respect to the depth of the search tree.

Summary:

Time Complexity: (O(b^d)) Space Complexity: (O(bd)) RBFS is more memory-efficient than other best-first search algorithms like A* because it uses linear space, but it may require more time due to re-expanding nodes.

References:

We have referred below text-book for algorithm approach, function names etc.

Stuart Russell and Peter Norvig, "Artificial Intelligence – A Modern Approach", Third Ed, Pearson Education, 2010