Breadth-First Search (BFS) algorithm

Source: ***https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms***

1. What is BFS algorithm?

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) and explores the neighbor nodes first, before moving to the next level neighbors.

The algorithm works as follows:

1. Start at the root node and add it to the queue.
2. Take the front node from the queue and explore its children.
3. If the node has any children, add them to the queue.
4. Repeat steps 2 and 3 until the queue is empty.

BFS is useful for finding the shortest path between two nodes in a graph. It has a time complexity of O(n) for traversing n nodes in the worst case. It requires more memory than other traversal algorithms, as it needs to store the entire breadth of the tree at each level in the queue.

BFS can be implemented using an iterative or a recursive approach. It can also be implemented using a stack instead of a queue, in which case it is called a depth-first search (DFS).

2. Code example.

```cpp
#include <iostream>
#include <queue>
#include <unordered_set>
using namespace std;
bool bfs(unordered_set<int>* graph, int start, int goal) {
  queue<int> queue;
  unordered_set<int> visited;
  queue.push(start);
  visited.insert(start);
  while (!queue.empty()) {
    int node = queue.front();
    queue.pop();
    if (node == goal) {
      return true;
    }
    for (int neighbor : graph[node]) {
      if (visited.find(neighbor) == visited.end()) {
        queue.push(neighbor);
        visited.insert(neighbor);
      }
    }
  }
  return false;
}
int main() {
  unordered_set<int> graph[3];
  graph[0].insert(1);
  graph[0].insert(2);
  graph[1].insert(2);
  graph[2].insert(0);
  graph[2].insert(1);

  cout << bfs(graph, 0, 2) << endl;  // 1
  cout << bfs(graph, 1, 0) << endl;  // 0
  return 0;
}
```

3. *Code explanation.*

In the example of the Breadth-first search (BFS) algorithm, the bfs function takes in three parameters:

- graph: an array of unordered sets representing the graph. Each set contains the neighbors of the corresponding node.
- start: the starting node for the search.
- goal: the goal node that the search is trying to reach.

The function returns a Boolean indicating whether the goal node is reachable from the start node.

The function begins by initializing an empty queue and an empty set of visited nodes. It then adds the start node to the queue and marks it as visited.

The function then enters a loop that continues until the queue is empty. In each iteration of the loop, the function takes the front node from the queue, removes it from the queue, and checks whether it is the goal node. If it is, the function returns true. If it is not, the function adds the node's neighbors to the queue and marks them as visited.

Once the queue is empty, the function returns false, indicating that the goal node was not found.

In the main function, the graph is initialized as an array of sets. Each set contains the neighbors of the corresponding node. For example, the neighbors of node 0 are nodes 1 and 2, and the neighbors of node 1 are node 2.

The function is then called with the start and goal nodes as arguments, and the result is printed to the console.

In this graph, there are three nodes labeled 0, 1, and 2. Node 0 is connected to nodes 1 and 2, and node 1 is connected to node 2.

If the bfs function is called with start node 0 and goal node 2, the search would proceed as follows:

1. Add 0 to the queue and mark it as visited.
2. Add 1 and 2 to the queue and mark them as visited.
3. Remove 0 from the queue.
4. Remove 1 from the queue.
5. Remove 2 from the queue. Return true, because the goal node has been found.

If the bfs function is called with start node 1 and goal node 0, the search would proceed as follows:

1. Add 1 to the queue and mark it as visited.
2. Add 2 to the queue and mark it as visited.
3. Remove 1 from the queue.
4. Remove 2 from the queue. Return false, because the goal node has not been found.

4. Time complexity.

The time complexity of the Breadth-first search (BFS) algorithm in the example is O(n), where n is the number of nodes in the graph.

In the worst case, the algorithm may need to visit all the nodes in the graph before finding the goal node or determining that it is not reachable. This means that the time complexity of the algorithm is linear in the number of nodes.

The space complexity of the algorithm is also O(n), as it uses a queue to store the nodes to be explored, and this queue may contain up to n nodes at any given time.

In practice, the time and space complexity of the algorithm may be lower if the graph is sparse, meaning that it has a relatively small number of edges compared to the number of nodes. In a sparse graph, the algorithm may be able to find the goal node or determine that it is not reachable before visiting all the nodes in the graph.

I wish you happy learning,

Yours Rosen Rusev