Bfs Dfs ComplexityAnalysis

Amith C A

2020MCS120003

DSC523 Project 1

Computational Complexity

Computational complexity is a field from computer science which analyzes algorithms based on the amount resources required for running it. The amount of required resources varies based on the input size, so the complexity is generally expressed as a function of n, where n is the size of the input.
It is important to note that when analyzing an algorithm we can consider the time complexity and space complexity. The space complexity is basically the amount of memory space required to solve a problem in relation to the input size. Even though the space complexity is important when analyzing an algorithm, in this story we will focus only on the time complexity.

Time Complexity

In computer science, the time complexity is the computational complexity that describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

Big-O Notation

Big-O notation, sometimes called asymptotic notation, is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.
In computer science, Big-O notation is used to classify algorithms according to how their running time or space requirements grow as the input size (n) grows. This notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

Table of common time complexities

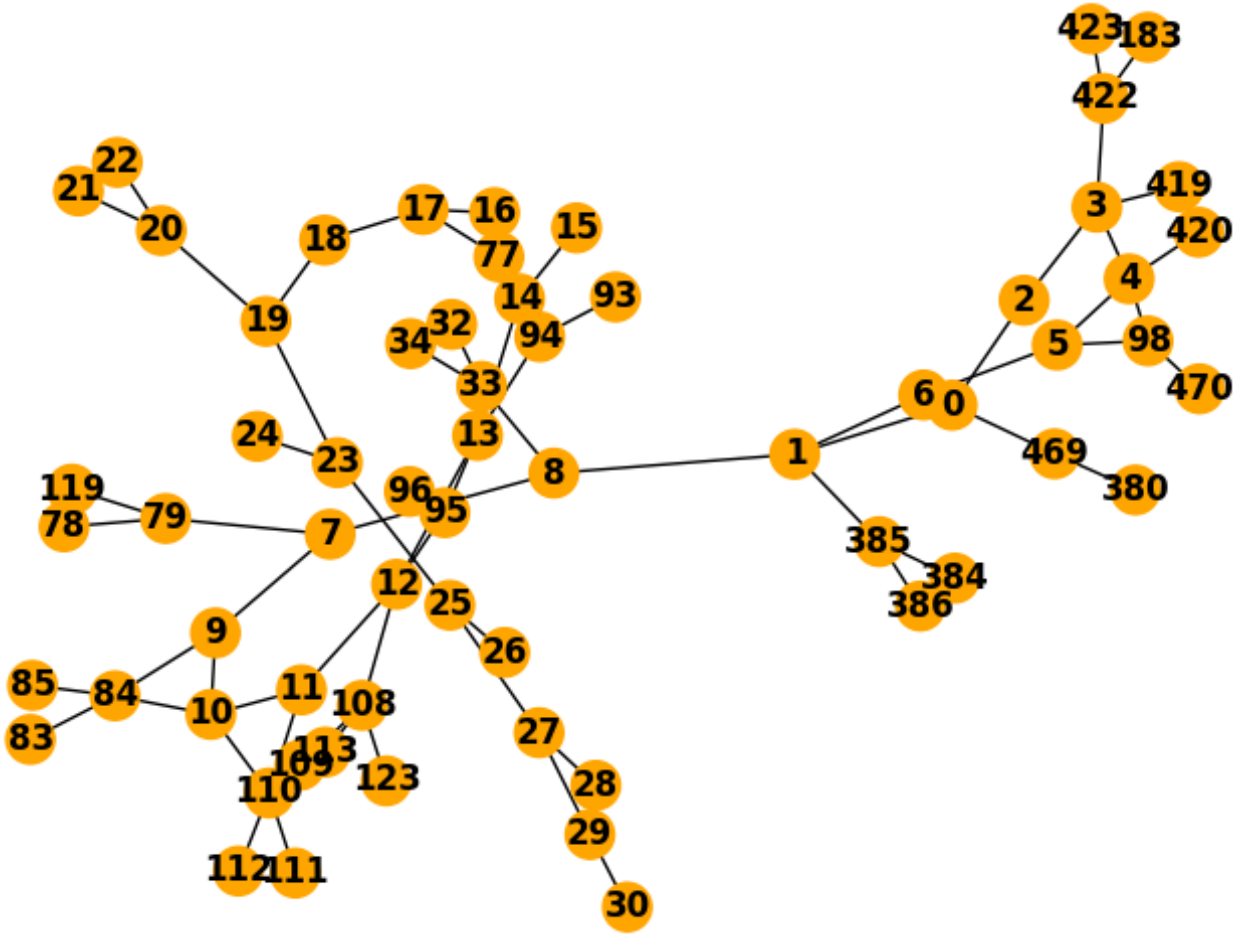
Name	Time Complexity
Constant Time	O(1)
Logarithmic Time	O(log n)
Linear Time	O(n)
Quasilinear Time	O(n log n)
Quadratic Time	O(n^2)
Exponential Time	O(2^n)
Factorial Time	O(n!)

Input Graph

Consider the data roadNet-CA.txt, which shows the road network data from California.

The graph constructed using the data from roadNet-CA.txt is shown below, which serves as the input graph to our algorithms.

Input graph plot:



Depth First Search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.Depth-first search is like walking through a corn maze. You explore one path, hit a dead end, and go back and try a different one

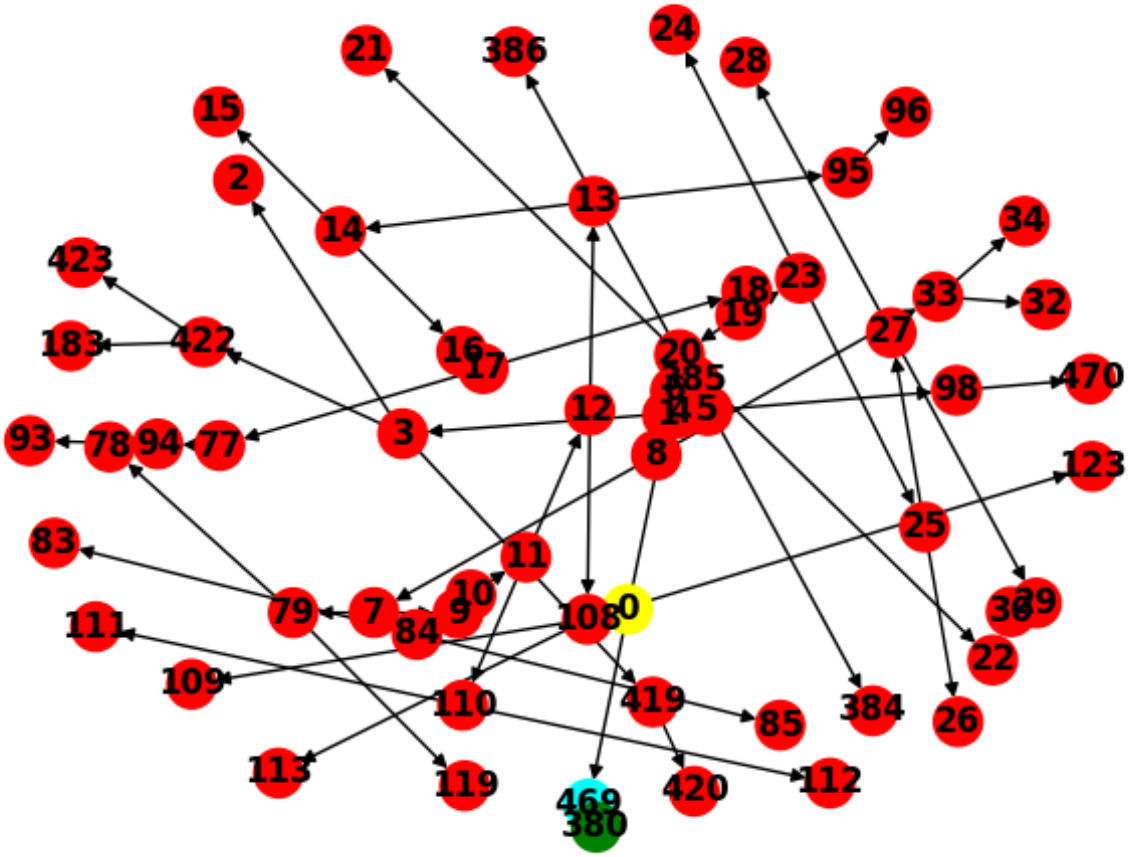
Uses of DFS :
Find path between two given vertices u and v.
Perform topological sorting is used to scheduling jobs from given dependencies among jobs.
Find strongly connected components of a graph.

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.
This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:
Pick a starting node and push all its adjacent nodes into a stack.
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

DFS Tree

The Shortest path from start node 0 to goal node 380 is : [0, 469, 380]

DFS Tree plot:



Breadth First Search

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.

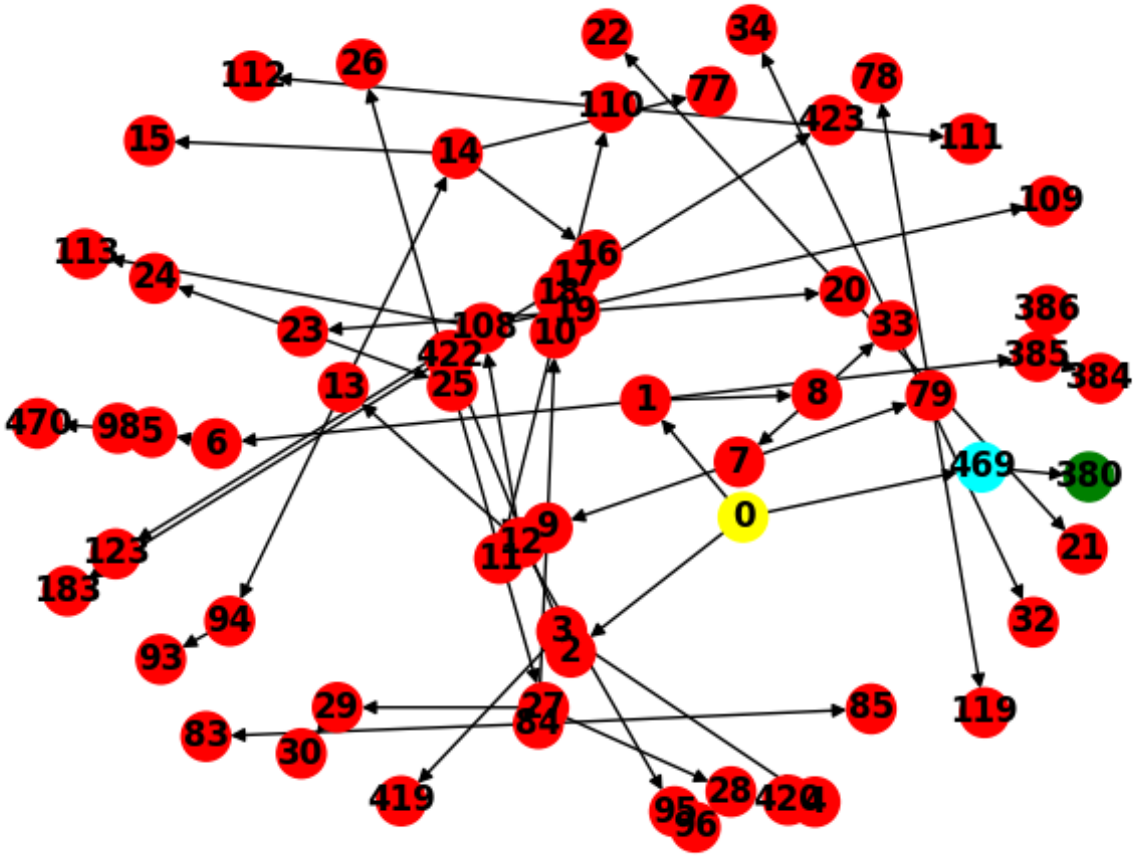
It uses the opposite strategy of depth-first search, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes.

BFS and its application in finding connected components of graphs were invented in 1945 by Konrad Zuse, in his (rejected) Ph.D. thesis on the PlankalkÃ¼l programming language, but this was not published until 1972. It was reinvented in 1959 by Edward F. Moore, who used it to find the shortest path out of a maze, and later developed by C. Y. Lee into a wire routing algorithm (published 1961).

BFS Tree

The Shortest path from start node 0 to goal node 380 is : [0, 469, 380]

BFS Tree plot:

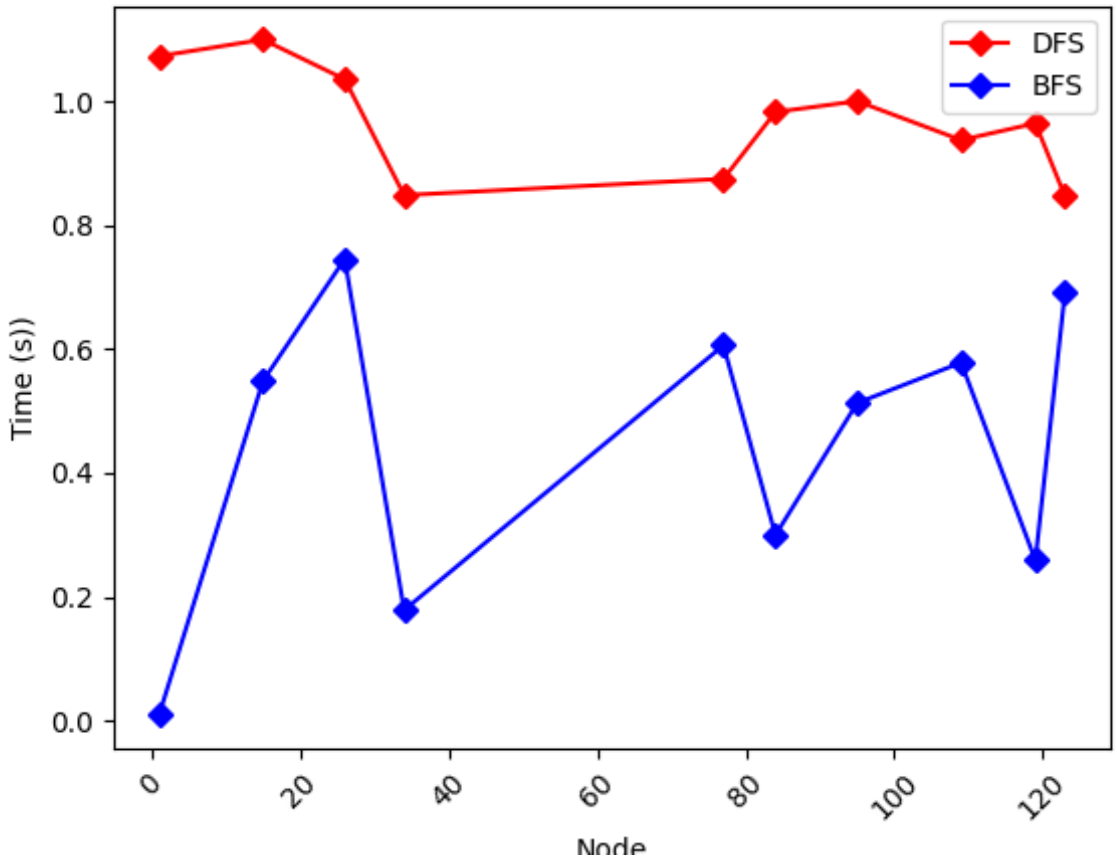


BFS vs DFS

BFS	DFS
BFS stands for Breadth First Search.	DFS stands for Depth First Search.
BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.
BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win
The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.

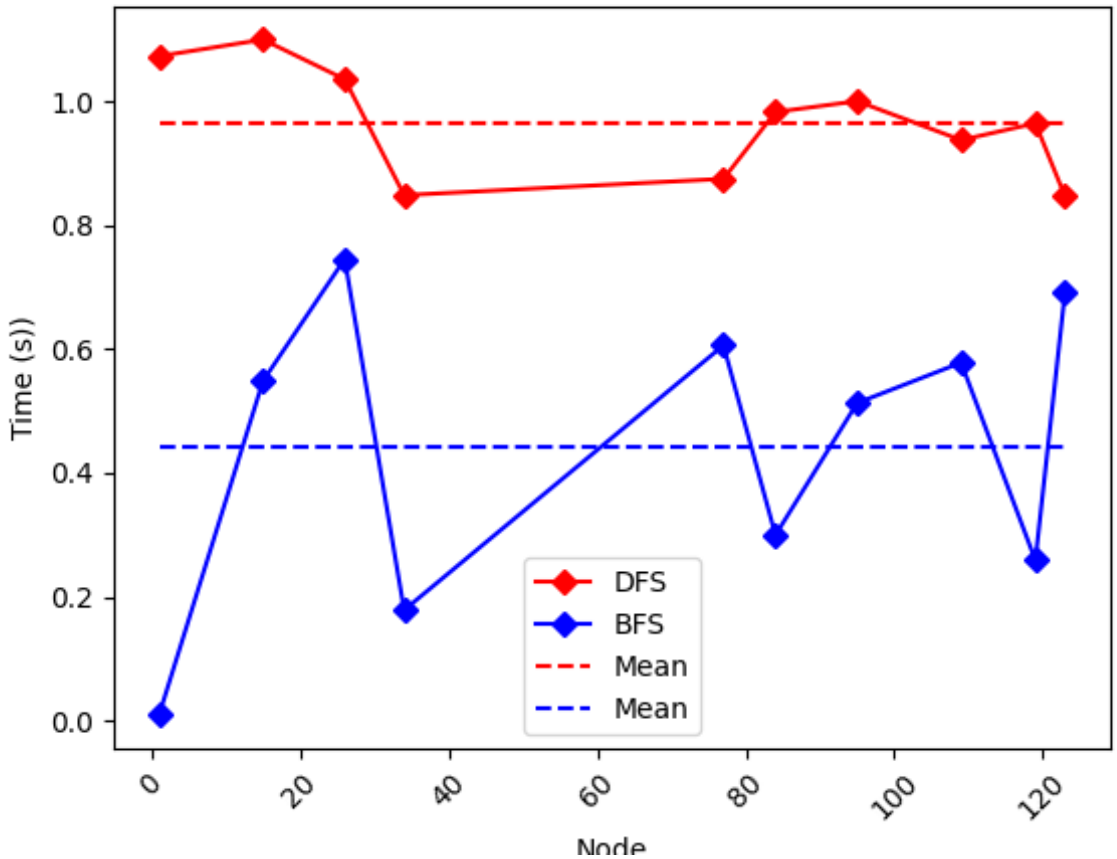
Time complexity analysis of BFS vs DFS

The time complexity plot of BFS vs DFS is as below:



Time complexity analysis of BFS vs DFS (Average)

The time complexity plot of BFS vs DFS is as below:



Conclusion

Memory requirements:

The stack size is bound by the depth whereas the queue size is bound by the width. For a balanced binary tree with n nodes, that means the stack size would be log(n) but the queue size would b O(n).

Speed:

For a full search, both BFS and DFS visit all the nodes without significant extra overhead. If the search can be aborted when a matching element is found, BFS should typically be faster if the searched element is typically higher up in the search tree because it goes level by level. DFS might be faster if the searched element is typically relatively deep and finding one of many is sufficient.