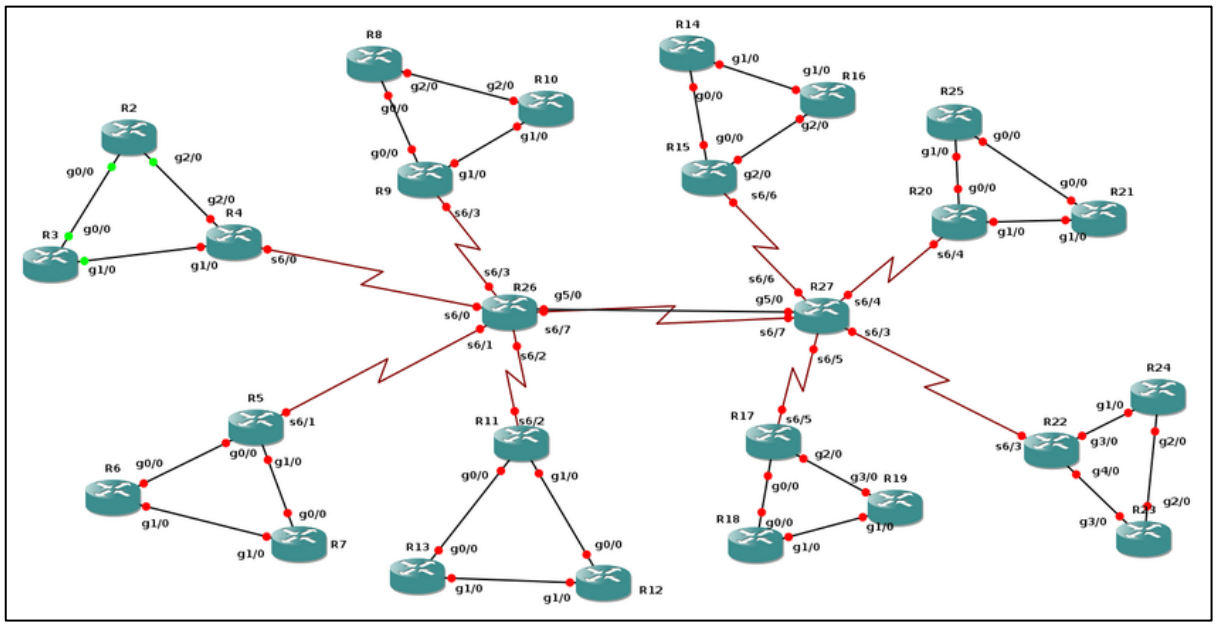
THE UNIVERSITY OF WESTERN AUSTRALIA

CITS2200 DATA STRUCTURES AND ALGORITHMS PROJECT 2021

Vlad Gavrikov (22471649) & Nicodemus Ong (22607943)

Introduction

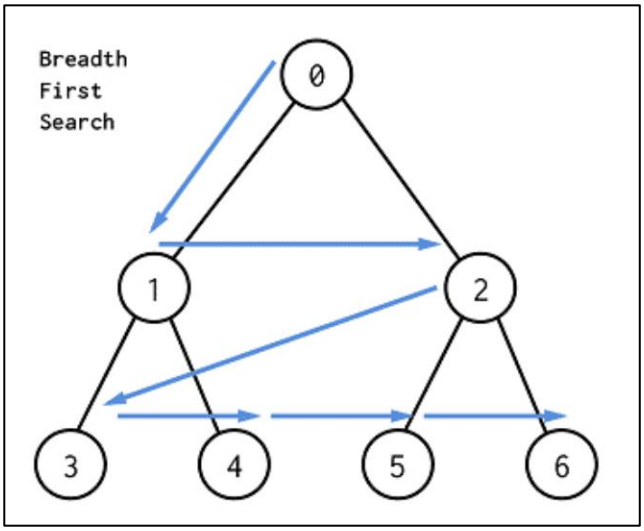
A graph is a non-linear data structure that consist of both vertices and edges connecting them. The vertices can also be referred to as nodes and the edges are often represented as a line connecting two vertices in the graph. A graph can have directions and weight on the edges. For this project we will be simulating a simple computer network. A network contains devices that are connected by a physical link. Some links may transmit in both directions, that does not mean however that every link can transmit in both directions. If a pair of devices are not linked physically, they can still send packets to each other if there is a sequence of links that links it from the source to its destination device. Every time the packet goes through a device it's called a hop.



The figure above shows an example of what a computer network looks like.

1. Algorithms

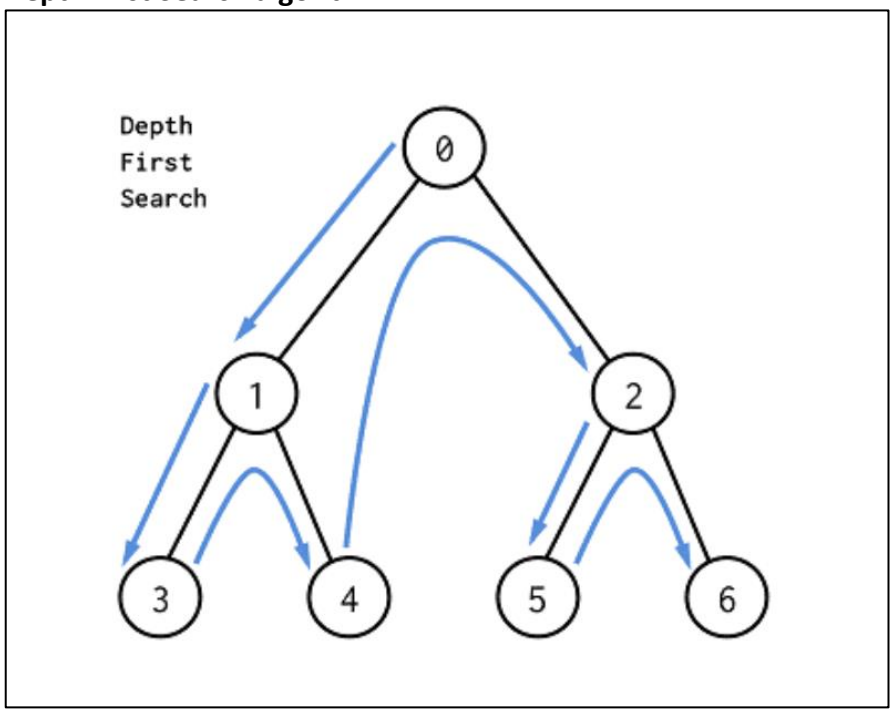
Breadth First Search algorithm



*The figure above shows an example of how BFS would traverse a graph

The Breadth First Search algorithm starts from an arbitrary node and does a level order traversal in the graph. It does this by exploring every neighbor of the node before moving on to the next level. BFS utilizes a Queue data structure for its First in First Out property. At every iteration, an unvisited node is added to the queue. This results in a search that goes level by level. The complexity of the BFS is represented as O(V + E) where V is the number of vertices and E being the number of edges in the graph.

Depth First Search algorithm



*The figure above shows how a DFS would traverse a graph

Depth First Search starts at an arbitrary node and recursively traverse through the graph. Unlike breadth first search where it is a level order traversal, DFS traverses the graph by going deep. It goes down from each vertex to the next till it reaches the bottom of the tree. Once the traversal has hit the bottom, it starts to backtrack till it finds an unexplored node. The recursion does not stop till the entire graph has been traversed. Depth first search is ideal for analyzing and mapping out different paths from a node, hence why we have decided to implement this algorithm to solve this problem.
The Time complexity of the Depth First Search is O(V + E), where V is the number of vertices and E being the number of edges in the graph.

Edmonds Karp algorithm

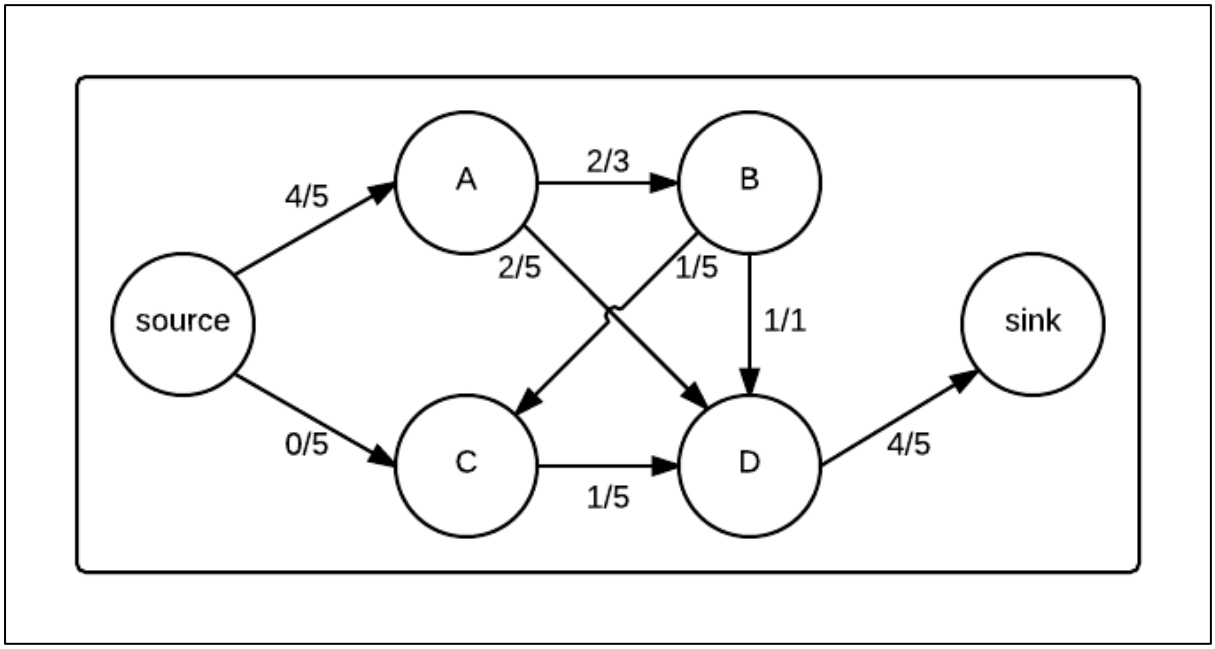


*****Figure above shows what a maximum flow graph generally would look like

A network is often abstractly defined as a graph G with a set of vertices V and connecting set of edges E. There is a source node S and a sink or destination node T which represents where the flow is coming from and going to. Edmond's Karp algorithm is a variation of the Ford-Fulkerson algorithm which improves its complexity of O(E*f) to O(VE^2). Unlike Ford-Fulkerson's, Edmond Karp's algorithm uses BFS to get the next augmented path. By doing so it ensures that it will always choose every shortest augmented path possible from the source to the sink.
The time complexity of Edmond's Karp algorithm is O(VE^2). It is improved from ford Fulkerson algorithm which runs on O(E*f), f being the maximum flow of the graph, it removes the dependency of the max flow.

2. Methods implementation and complexity

- allDevicesConnected(int[][] adjlist)
- numPaths(int[][] adjlist, int src, int dst)
- closestInSubnet(int[][] adjlist, short[][] addrs, int src, short[][] queries)
- maxDownloadSpeed(int[][] adjlist, int[][] speeds, int src, int dst)

AllDevicesConnected(int[][] adjlist):

This method takes in just one input, the adjacency list. To find out if this graph is strongly connected, we have chosen to use a Depth First Search approach. Our DFS comes in two parts, DFSForward and DFSInverse. For DFSForward, it executes the first DFS and populates an inverse adjlist. If all the vertices are visited in DFSForward, it will proceed to DFSInverse, else return false. It checks the state of each vertex by going through the visited array and checks if each vertex has been visited. DFSInverse executes the second DFS on the inverse graph. If all vertices have been visited, this means that the graph is strongly connected and will return true, else return false. Like DFSForward, it checks the visited array to see if all vertices have been visited in the inverse graph. As for the Check method, it calls the DFSForward and DFSInverse and if both methods return true, the graph is strongly connected and will return true, else it returns false.

For the complexity of our method, as we are using DFS twice, the complexity is O(2D+2L) which can be simplified to O(D+L). It can be interpreted as O(N) in terms of the number of vertices D in the graph.

numPaths(int[][] adjlist, int src, int dst):

This method takes in three inputs: The adjacency list, the source vertex, and the destination vertex. The method executes a recursive DFS of unvisited vertices. In each iteration of DFS, it marks the vertices that have been visited as true, if the current vertex has neighbors, DFS will recursively execute on those neighbors. At the end of each DFS Iteration, the source vertex will be set to false. A pathlist arraylist is initialized to keep track of all possible paths from the source to the destination. A counter is also initialized to keep track of the total number of possible paths available. Every time source vertex is equal to the destination vertex, the counter will be incremented by 1 and returned. Once the recursive DFS is complete, the method will return the counter that has the total number of possible paths in the graph from the source vertex to destination vertex.

For the complexity of this method, as it runs DFS, it is O(D+L). It can be interpreted as O(N) in terms of the number of vertices D in the graph.

closestInSubnet(int[][] adjlist, short[][] addrs, int src, short[][] queries):

This method takes in four inputs: The adjacency list, the addresses for each vertex array, the source vertex, and the queries array. The method first executes a topological sort and places all the sorted vertices into a stack. It than checks if the source node can reach the queried IP address. The algorithm pops each element from the stack one at a time and checks if each deleted element reaches the queried IP address. If the deleted element reaches the IP address, it saves the distance in the array of answers. Once all queries in the queries array have been processed, the method will return the array of answers. The TopSort() method performs a recursive topological sort starting from the source, places each item onto the stack and returns the stack once each vertex has been added. Class Node() creates a node of each vertex and holds the IP address of the vertex. CheckQuerie() checks if the queried IP

address matches the subnet of the vertex's IP address through its node, if it matches returns true.
The complexity of this algorithm is O(D + L). in the worst case, the stack will have D number of vertices to be popped and L number of edges. This gives the overall complexity of O(D + L). It can be interpreted as O(N) in terms of the number of vertices V in the graph. The algorithm is executed for each Querie Q, hence performing this method will take O(N + Q).

maxDownloadSpeed(int[][] adjlist, int[][] speeds, int src, int dst):
Takes in 3 inputs: the adjacency list of the graph, the speed of each vertex array, the source vertex, and the destination vertex. The method first checks if the source and destination are the same vertex if they are the same return -1. The method will first convert the adjacency list to a residual matrix. The method will run for every possible shortest path in the graph. This is achieved by running BFS through every iteration to get each possible path. While True, the algorithm will first find the minimum residual capacity of the edges along the path filled by BFS. After which it updates the residual capacities of the edges and reverse edges along the path. It adds the path flow to the overall flow. Once all possible paths have been visited, it will exit the while loop and return the maximum flow of that graph. The convert() method takes in 2 inputs: the adjacency list and the speeds array. It than creates a residual graph that will be used in the algorithm.
The overall complexity of Edmond Karp's algorithm is O(DL^2). There can at most be O(DL) moments when any of the edges has fully utilize its capacity and finding an augmented path takes O(L) times. Hence the algorithm will be O(DL*L) which translate to O(DL^2).

References

Edmonds-Karp Algorithm | Brilliant Math & Science Wiki. (2021). Retrieved 14 May 2021, from https://brilliant.org/wiki/edmonds-karp-algorithm/

Maximum flow - Ford-Fulkerson and Edmonds-Karp - Competitive Programming Algorithms. (2021). Retrieved 14 May 2021, from https://cp-algorithms.com/graph/edmonds_karp.html#:~:text=Edmonds%2DKarp%20algorithm%20is%20just,BFS%20for%20finding%20augmenting%20paths.&text=The%20intuition%20is%2C%20that%20every,again%20in%20an%20augmenting%20path.

Breadth First Search or BFS for a Graph - GeeksforGeeks. (2021). Retrieved 14 May 2021, from https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

Depth First Search Tutorials & Notes | Algorithms | HackerEarth. (2021). Retrieved 14 May 2021, from https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

Shortest Path in Directed Acyclic Graph - GeeksforGeeks. (2021). Retrieved 20 May 2021, from https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/