CITS2200

Discrete Structures and Algorithms

Project Report

Semester 1: 2019

Seharsh Srivastava – (22248457)

Sai Prateek Bandari – (22230847)

Introduction:

This project dives into the analysis of graphs which is a key example of a data structure.

These were the tasks we were required to complete:

- Method which computes the minimum number of edges that must be traversed to get between two pages. (Shortest Path)
- Method which finds a Hamiltonian path within the graph. (Only for max 20 vertices)
- Method that computes the most strongly connected components of the graph.
- Method that finds the centre of the graph.

We are applying our analysis on a graph based on the online website "Wikipedia". The graph will be directed but unweighted. Nodes are represented by pages on the website and edges are represented by links between these pages.

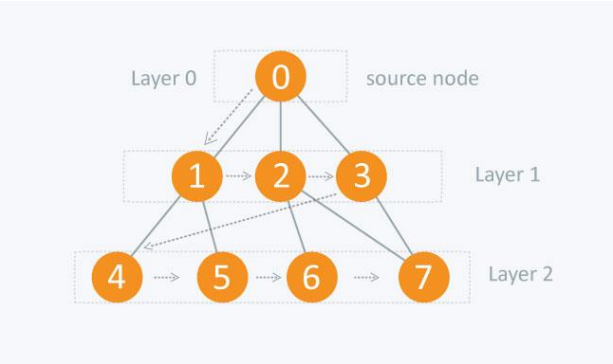
Shortest Path:

As the name suggests we are simply required to find the shortest path between two nodes if such a path exists (For a directed and unweighted graph). Each node is represented by a Wikipedia page. Due to the graph being unweighted the distance is dictated by the number of edges that we must traverse to get from one page to another. Hence, shortest path simply refers to the minimum number of edges that we must traverse to get from our initial node to our final node.

To compute the shortest path, we used the breadth first search (BFS) algorithm. We convert the graph into layer basis and traverse it breadthwise starting from the initial node.

We go through the first layer's connections, then second, then third……

Making sure that we do not repeat any unnecessary connections. This method ensures that we visit the minimum amount of edges possible. We tested our method against the test data and it gave us the correct output.



Source: (https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/)

Pseudocode we referred to:

Pseudocode

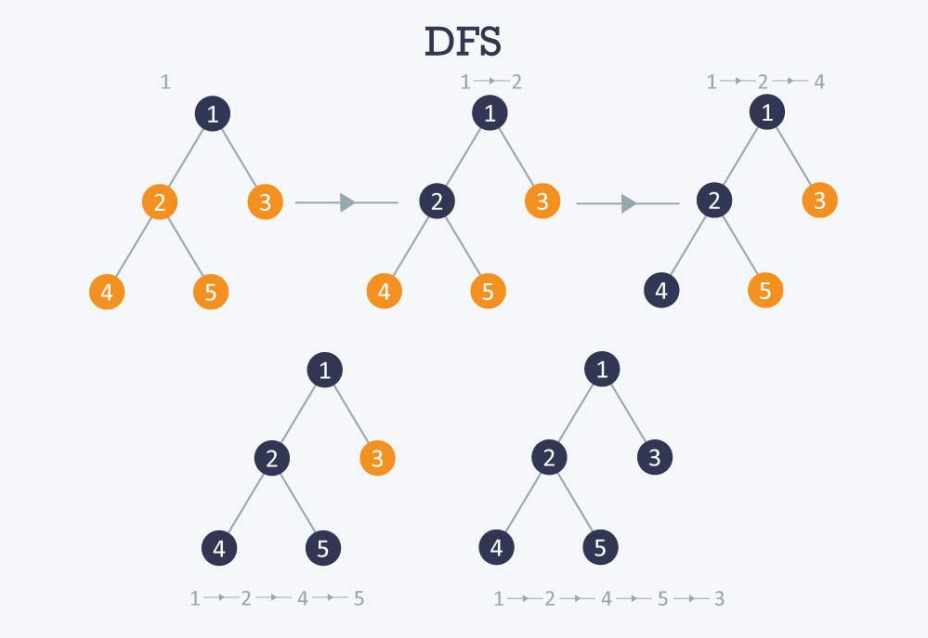
```
BFS (G, s)                    //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

      mark s as visited.
      while ( Q is not empty)
            //Removing that vertex from queue,whose neighbour will be visited now
            v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                         Q.enqueue( w )              //Stores w in Q to further visit its neighbour
                         mark w as visited.
```

Source: (https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/)

Complexity: O(V+E)
V: Number of vertices/nodes (Pages) E: Number of edges (Links)
Runtime: 130 milliseconds

Hamiltonian Path:

A Hamiltonian path is a path that visits each node in a graph exactly once. To compute a Hamiltonian path we used a depth first search (DFS) algorithm. The DFS algorithm uses recursion and applies the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.



Source: (https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/)

Pseudocode we referred to:

Pseudocode

```
DFS-iterative (G, s):                                          //Where G is graph and s is source vertex
  let S be stack
  S.push( s )              //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
      //Pop a vertex from stack to visit next
      v  =  S.top( )
     S.pop( )
     //Push all the neighbours of v in stack that are not visited
    for all neighbours w of v in Graph G:
        if w is not visited :
                  S.push( w )
                 mark w as visited


DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

Source: (https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/)

We implemented the DFS algorithm by using a stack. The process starts by choosing an initial node and pushing all the adjacent nodes into a stack. We then pop a node from the stack to select the next node to visit and then again we push all the adjacent nodes into a stack. We simply repeat this process until we have an empty stack. (During this process we must also ensure we are not visiting the same node more than once)

A downside to the DFS algorithm, is that it is computationally intensive. However we were only required to run our program on graphs that had at most 20 vertices, hence it was not too big of an issue. We tested our method with the test data and it gave us the correct output.

Complexity: O(V V!)

Runtime: 2081 milliseconds

Strongly Connected Components:

A graph is strongly connected if each vertex has a path to ever other vertex. A Strongly connected component (SCC) is a set of vertices that can reach all other vertices (maximally connected subgraph). To compute SCC's we applied Kosaraju's algorithm and used a bit of our own intuition. There are many other alternative algorithms we could have applied, such as, Tarjan's algorithm or Dijkstra's algorithm. We chose to apply Kosaraju's as it was easier for us to implement.

The intuition behind Kosaraju's algorithm is that apply a repeated DFS. We created an empty stack and execute a DFS traversal of our graph. We push vertexes into the stack once we complete the recursion. We then reverse the directions of all the edges in order to obtain the transpose graph. Finally we pop the vertexes from our stack and run a DFS on the vertex that is popped. The DFS beginning from the vertex that was popped will output the most strongly connected component.

Pseudocode we referred to:

1. For each vertex *u* of the graph, mark *u* as unvisited. Let *L* be empty.
2. For each vertex *u* of the graph do Visit(*u*), where Visit(*u*) is the recursive subroutine:
    If *u* is unvisited then:
        1. Mark *u* as visited.
        2. For each out-neighbour *v* of *u*, do Visit(*v*).
        3. Prepend *u* to *L*.
    Otherwise do nothing.
3. For each element *u* of *L* in order, do Assign(*u*,*u*) where Assign(*u*,*root*) is the recursive subroutine:
    If *u* has not been assigned to a component then:
        1. Assign *u* as belonging to the component whose root is *root*.
        2. For each in-neighbour *v* of *u*, do Assign(*v*,*root*).
    Otherwise do nothing.

Source: (https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm)

When testing our data with the wiki test graph posted in the CITS2200 project page, we managed to get all the outputs except the final output of Angola. This would have been due to a small error. We were not able to debug and fix this error as we were restricted by time constraints.

Complexity: O(V+E)

Runtime: 120 milliseconds

Graph Centres:

The centre of a graph is the point from which the most distant other point is as close as possible (The vertex from which the most distant other point is as close as possible). In some situations, it is possible for a given graph to have more than one centre.

We implemented the average eccentricity to calculate the centres of the graph. For the get center method, we made use of the already implemented get shortest path method. Using this we calculated the distances of every vertex to other vertices and then for each individual vertex we calculated the average distance of said vertex to other vertices. We then picked the one with the smallest average to be the centre of the graph. When we tested our method with the test data, we received /wiki/Nicaragua, this was not the correct output. However we believed our intuition was correct behind out implementation hence we stuck with our implementation.

Complexity: O(V+E)

Runtime: 2560 milliseconds

Conclusion:

We found this project very challenging, mostly due to time constraints. It required heavy research and understanding into algorithms. We were not able to implement all the required methods, as we did not have enough time to debug our code. I believe after completing this project we gained more understanding into algorithms and how they are applied to graphs.

References:

HackerEarth. (2019). *Breadth First Search Tutorials & Notes | Algorithms | HackerEarth*. [online] Available at: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/ [Accessed 31 May 2019].

En.wikipedia.org. (2019). *Kosaraju's algorithm*. [online] Available at: https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm [Accessed 31 May 2019].

GeeksforGeeks. (2019). *Strongly Connected Components - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/strongly-connected-components/ [Accessed 31 May 2019].

Cs.usfca.edu. (2019). *Breadth-First Search*. [online] Available at: https://www.cs.usfca.edu/~galles/visualization/BFS.html [Accessed 31 May 2019].

Algs4.cs.princeton.edu. (2019). *Directed Graphs*. [online] Available at: https://algs4.cs.princeton.edu/42digraph/ [Accessed 31 May 2019].