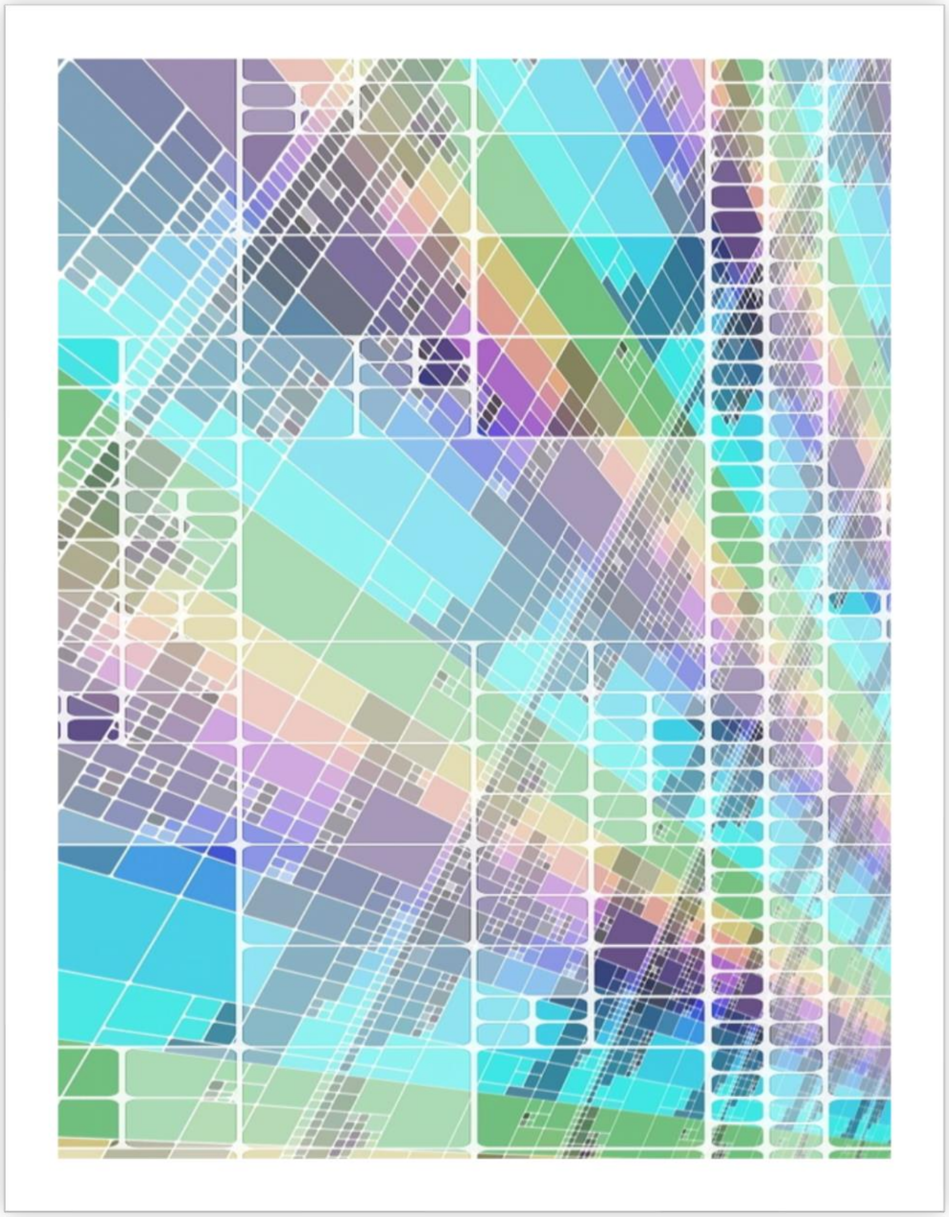

CITS2200 Data Structures & Algorithms

PROJECT 2019

Alexander Varano della Vergiliana | Student ID: 22268701 | May 31, 2019

Introduction

The following report details the design and implementation of the requirements for the CITS2200 2019 project, which encompasses four different algorithms solving a distinct problem each on directed, unweighted graphs. The four implementations that will be discussed are: Finding the shortest path between two nodes (SP), finding the Hamiltonian path of a graph, finding the strongly connected components (SCC) of a graph, and finally finding the centers of a graph.

Performance studies were carried out on each of the algorithm implementations utilizing a modern computer of Intel x86-64 architecture with a central processing unit speed of 4.8Ghz, 16 gigabytes of random-access memory and a solid-state storage drive. The runtime was recorded for each algorithm and three main datasets were utilised as part of this process. One provided as part of the project, a small graph of Wikipedia nodes and edges and two much larger datasets acquired from the Stanford Large Network Dataset Collection (snap.stanford.edu/data/index.html) which is part of the Stanford Network Analysis Project (SNAP). The details of which are below:

- EU Email Communications Network:
    - Details: Sent and received emails between European Union research institutes.
    - Graph type: Directed, unweighted.
    - Nodes: 265,214
    - Edges: 420,045
- University of Notre Dame Web Graph
    - Details: Webpages and hyperlinks between them from nd.edu
    - Graph type: Directed, unweighted.
    - Nodes: 325,729
    - Edges: 1,497,134

Loading the Dataset

Two Java methods have been developed to facilitate the loading of graph data into an adjacency list data structure and also creating a maintaining an index of all vertices and their position within the adjacency list. The functions are getVert() and advert(). The adjacency list and the associated index are used throughout this project as the means to access the dataset for each solution implemented. Two adjacency lists are maintained, an original and a transposed version. Both taking O(V + E) time to create. Two indexes are created at runtime and added to as each vertex is created in the adjacency lists. It takes the form of a HashMap, with the key being a String value of the vertex and the data being an index number, correlating with the array position of that vertex in the adjacency list. Therefor a relationship between the HashMap and the adjacency list is formed which allows us fast lookup of vertex array positions by querying the HashMap. Testing of load times against our test data was performed, and it is presented below:

- CITS2200 Wikipedia Graph: 25ms
- EU Email Communications Network: 2,214,033ms (~37 minutes)
- University of Notre Dame WG: 2,455,128ms (~41 minutes)

Shortest Path

This problem required the implementation of a solution which given a pair of pages, would return the minimum number of links you must follow to get from the first page to the second, or in other words to solve the shortest path. The implementation taken was that of a breadth first search, utilizing a queue data structure. The following pseudo-code outlines the approach taken:

```
Get variables urlFrom and urlTo
Create new queue Q
Set each vertices visited = False
Add urlFrom -> Q
urlFrom visited = True
While Q != Empty
    Dequeue vertex V <- Q
    Parent = V
    For each adjacent vertex A of V
        If A != visited
            Add A -> Q
            A visited = True
            A Distance = Parent Distance + 1
        If A = urlTo
            Destination reached = True
            Break to return statement
Return Distance A
```

The above implementation in a worst-case scenario will iterate over every vertex at most once and then over every adjacent vertex or edge of that vertex. Therefore, the problem is solved with a time complexity of O(V + E). Real world run-time testing was performed on Java implementation of this algorithm against the four datasets and the results are detailed below:

- CITS2200 Wikipedia Graph: < 1ms
- EU Email Communications Network: 90ms
- University of Notre Dame WG: 187ms

Results from running the java implementation against the CITS2200 graph where urlFrom = '/wiki/Flow_network' and urlTo = '/wiki/Push%E2%80%93relabel_maximum_flow_algorithm' provided a result for shortest path of 2.

Hamiltonian Path

This problem required an implementation that would try to find a Hamiltonian Path from a supplied graph. The approach taken was that of a brute force implementation, performing a recursive depth first search on all vertices, we then mark vertices as visited and maintain them in a list, the the case of reaching a dead end we then use backtracking functionality, removing vertices from this list and marking them unvisited. Doing so, if the list eventually grows to the same size as the total number of vertices, we have found a Hamiltonian Path. The pseudo-code for this implementation is presented below:

```
Path[] = NULL
For each vertex V of graph G
    add V -> Path[]
    {begin: hamsDFS(V)
        V visited = True
        If size of Path[] = number of vertices in G
            Hamiltonian path found. Return Path[]
        Else
            For each adjacent vertex A of V
                If A != visited
                    add A -> Path[]
                    hamsDFS(A)
                    Backtrack: A visited = false
                    Backtrack: remove A -> Path[]
    {end; hamsDFS()}
    If size of Path[] = number of vertices in G
        break
Return Path[]
```

Due to the brute force nature of this implementation, the time complexity of this implementation in a worst-case scenario is O(V!). Therefore, running the java implementation of this solution against anything more than 20 vertices, would be futile. There real-world testing for this implementation were limited to the CITS2200 graph. The results are below:

- CITS2200 Wikipedia graph: 35ms

No Hamiltonian Path was found with the above dataset.

Strongly Connected Components

This problem required the design and implementation of an algorithm which would find all the strongly connected components (SCC's) from the sample graph. The approach taken was to implement a known solution to this problem, that of Kosaraju's algorithm. This approach required two versions of the graph's adjacency list, that being of the original and that of a transposed version of the original. The transposed version utilised was created during the original loading of the graph data in the addEdge() and advert() functions. The basic premise behind the implementation of Kosaraju's algorithm is to build a stack data structure containing vertices obtained from a recursive depth first search of the graph, filling the stack with vertices ordered in finishing time. Once we have this stack, we pop the vertices from it one by one and use this order of vertices to perform another recursive depth first search, however this time on the transposed version of the graph. Doing so provides us with the SCC's for each vertex which we can then add to our String array for returning. The array returned will either be a tree or forest structure, and hence the String[][] data structure is used as our return value. Below is the pseudo-code implementation taken:

```
Create new stack S
Set each vertices visited = False
For each vertex V of graph G
    If V != visited
    {begin: fillStackDFS (V)}
        V visited = True
        For each adjacent vertex A of V
            If A != visited
                fillStackDFS(A)
        Add V -> S
    {end: fillStackDFS()}

Get transposed graph as T
While S != Empty
    Pop vertex V <- S
    If V != visited
    {begin: stackDFS(V)}
        V visited = True
        foundSCC <- V
        Get vertex Vt from T
        For each adjacent vertex At of Vt
            If At != visited
            stackDFS(At)
    {end: stackDFS()}

Return foundSCC
```

The above implementation of Kosaraju's algorithm for finding SCC's utilised two depth first searches as well as a function to transpose the original graph adjacency list. The time complexity of building the transposed list is no different to that of creating the original list, therefore it is O(V+E). Our depth first searches are also an O(V + E), therefore our overall time complexity for this implementation is O(V + E). The real-world runtime results for this implementation against the test graphs is as below:

- CITS2200 Wikipedia Graph: 1ms
- EU Email Communications Network: 338ms
- University of Notre Dame WG: 274ms

Results of running the java implementation against the CITS2200 graph produce the following SCC's:

/wiki/Nowhere-zero_flow	
/wiki/Flow_network	/wiki/Approximate_max-flow_min-cut_theorem /wiki/Ford%E2%80%93Fulkerson_algorithm /wiki/Dinic%27s_algorithm /wiki/Edmonds%E2%80%93Karp_algorithm /wiki/Max-flow_min-cut_theorem /wiki/Gomory%E2%80%93Hu_tree /wiki/Minimum_cut /wiki/Maximum_flow_problem /wiki/Circulation_problem /wiki/Multi-commodity_flow_problem /wiki/Minimum-cost_flow_problem /wiki/Network_simplex_algorithm /wiki/Out-of-kilter_algorithm /wiki/Push%E2%80%93relabel_maximum_flow_algorithm
/wiki/Braess%27_paradox	

Graph Centers

This problem required a solution to find all the graph centers, or all vertices of minimum eccentricity. One assumption was made in the design of this solution, in that a vertex would only be considered a center if a path existed from the vertex to all other vertices. The solution implemented utilised iterative calls to a previous discussed function, that being of finding the Shortest Path. By calling the shortest path breadth first search algorithm on all vertices and recording the largest shortest path found for each, we could then return the vertices which had the smallest of these paths, that also satisfied the requirement of being able to reach all other vertices from itself. The below details the pseudo-code to the implementation taken, it references call the above implementation of Shortest Path.

```
For all vertices V of graph G
    For all vertices J of G
        If V != J
            SP = {getShortestPath(V, J)}
        If SP = -1
            MAX = -1
            Break
        Else If SP > MAX
            MAX = SP
        MAXSP[V] = MAX
For all V in MAXSP[]
    If MAXSP[V] < MIN and MAXSP != -1
        MIN = MAXSP[V]
For all V in MAXSP[]
    If MAXSP[V] = MIN
    Add V -> CENTERS[]

Return CENTERS[]
```

The above implementation utilised iterative calls to our getShortestPath implementation, with its bread first search it has a time complexity of O(V + E). Because we are doing this for each vertex to find our centers this expands to O(V x (V + E)), or O(V^2 + VE). The real-world run-time results against our test data are as follows:

- CITS2200 Wikipedia Graph: 10ms
- EU Email Communications Network: 4,245,969ms (~70 minutes)
- University of Notre Dame WG: DNF. > 5 hours

Results of running this java implementation against the CITS220 graph produced a center:

/wiki/Nowhere-zero_flow

References

EU Communications Network dataset. Retrieved from http://snap.stanford.edu/data/email-EuAll.html

Notre Dame Web Graph dataset. Retrieved from http://snap.stanford.edu/data/web-NotreDame.html