# CITS2200 PROJECT REPORT

Ruan Scheepers (21980258) and Jainish Pithadiya (21962504)

June 5, 2017

## 1 Introduction

Graphs are a fundamental data structure used within computer science. This projects explores algorithms used to solve frequent queries that can be asked of a particular graph. This project focuses on four main questions...

- The shortest distance between two vertices.
- Finding a Hamiltonian path within a graph.
- What are the strongly connected components of the graph.
- Finding the centers of the graph.

The popular online encyclopedia *Wikipedia* can be considered a Graph by taking each page as a vertex and a URL link between two pages an edge. Using the four algorithms to solve the above problems we can analyze components such as what are the minimum amount of URL click to get from a particular page to another. Look at things like the sizes of strongly connected components to determine if there is an area of interest that can use more work.

The *Wikipedia* graph is a directed and unweighted graph, this means that the distance between adjacent vertices is always 1 and if vertex v is adjacent to vertex w it is not automatically true that w is adjacent to v.

The construction of the graph is done by adding an edge between two vertices. Our class is required to read in the two vertices, and depending if they already exist, create the vertices and the edge. The class is required to store the whole graph so when one of the four methods are invoked we return the solution on that particular graph. Since *Wikipedia* is constantly growing and our class can constantly get edges added to it. We used an Adjacency List to maintain our graph data structure. This allows us to store the graph as $O(|V| + |E|)$ instead of $O(|V|^2)$ if we instead used an Adjacent matrix. It also makes the process of adding vertices easier. An adjacency matrix would require resizing after adding each vertex taking $O(|V|^2)$ time.

## 2 Shortest Path

Shortest path is the path between two vertices in a graph where the sum of the edge weights in this path is minimal. Basically its the minimum distance between two vertices. For the *Wikipedia* graph since its a directed and unweighted graph, the shortest path between two pages will be the minimum number of clicks required to get from the start url to the end url.

### 2.1 Our Approach

The algorithm we used to solve this problem was Breadth First Search which has a worst case time complexity of *O(V+E)* where $V$ is the number of vertices (Pages) and $E$ is the number of Edges (Links) in the *Wikipedia* graph. The shortest path for the *Wikipedia* graph will be the path which requires the lowest amounts of clicks (i.e minimum amount of edges)

Breadth First Search uses a root vertex from which it transverses the graph , in our case that will be the start vertex and we will continue to transverse the graph using BFS till we have reached

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Figure 1: Algorithm for BFS: (Cormen, Leiserson, Rivest, and Stein, 2009)

the end vertex we want to find the distance of from the start vertex. This method will always guarantee that we pick the minimum amount of edges required to get from the start vertex to the end vertex as the method uses a queue which stores the adjacent vertices of a vertex in the order that the parent vertices were visited, from which we continue our transversal. Therefore allowing us to get the minimum distance between two vertices in a graph. With the implementation of our graph to note visited vertices and to avoid visiting already visited vertices we provided each vertex with the gsp variable which either has a value of its distance from the start vertex or a value of -1 to indicate that it hasnt been visited.

## 2.2 Time Complexity

Let our *Wikipedia* graph be *G(V,E)* where $V$ and $E$ are the vertices and edges in the graph.
In the worst case we may need to find the distance between two vertices which require a path which visits every vertex and edge from which we will need to transverse from using BFS. Since the amount of vertices we may use is at most $|V|$ and the amount of vertices we use is at most $|E|$, our worst case time complexity for get shortest path would be *O(|V|+|E|)*.

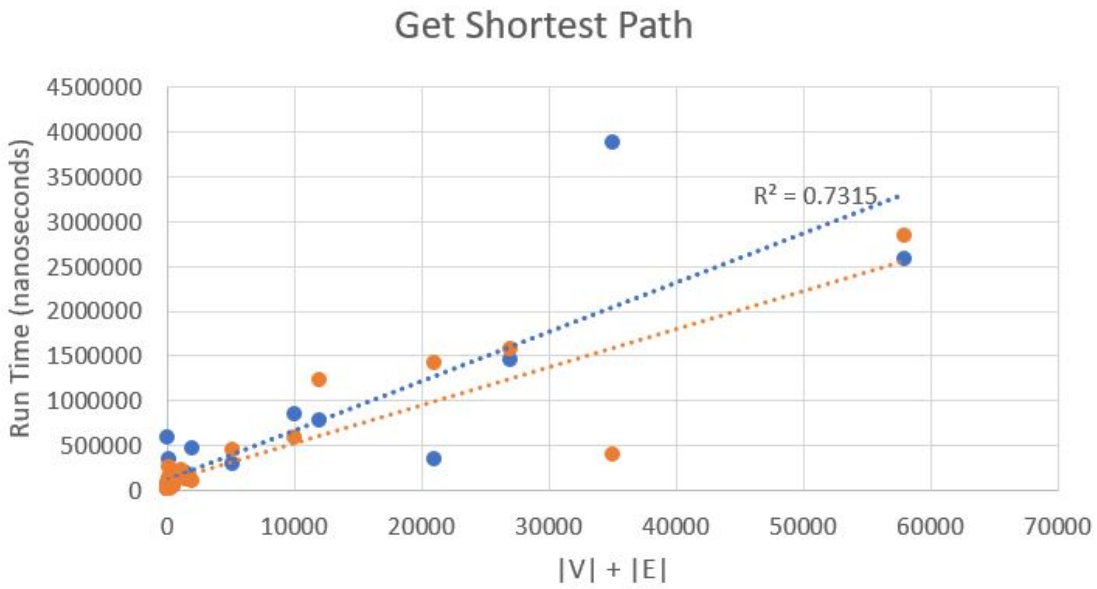


Figure 2: Run time analysis for vertices of amount 10 to 20000 and edges of amount 10 to 50000 of randomly generated graphs. The orange and blue lines represent the getshortestpath for two different pairs of vertices on the same graph. This highlights the fact that the time complexity $O(|V| + |E|)$ is the worst case scenario. Two points can lie very close to each other or might have to transverse all of the edges. Large discrepancies can be seen for example the second last point has a very large difference in run times.

# 3 Centers

Let our *Wikipedia* graph be *G(V,E)* where $V$ and $E$ are the vertices and edges in the graph. The centers of the graph are the set $C \subseteq V$ where $\forall\ c \in C$ where c has the minimum positive distance between the set of vertices in the set *V-c*. This function returns a array of string's which have the name of all the vertices which are centers of the graph. For example consider the unweighted graph A<-B->C with -> representing a edge and A,B and C being vertices of the graph. Here the center's would be just B as it connects to all its complimentary vertices and has the smallest positive distance to them.
Note the center of the graph must be able to reach all its complimentary vertices which means it must have a postive minimal distance between these vertices. Therefore we used our shortest path method as part of get centers.

## 3.1 Our Approach

No particular algorithm was exactly used for this method but basic definitions of graph theory were applied to solve this problem. The following definitions are considered. Information taken from : ("Graph Theory: 51. Eccentricity, Radius and Diameter", 2015).

1. Eccentricity: The eccentricity of a vertex in a graph is the maximum distance between itself and a complimentary vertex in the graph. Lets denote the eccentricity of a vertex as *E(V)*.
2. Radius: The radius of the graph is the value of the minimum eccentricity of all the vertices in the graph. i.e $\forall\ V \in$ *(Vertices of graph), radius= min* $\{E(V_1), E(V_2), E(V_3),..., E(V_n)\}$.

Essentially all the centers of the graph are the vertices with Eccentricity values of the radius of the graph. As mentioned previously for a vertex to be a center it must be able to reach all of its complimentary vertices, which is why each vertex has the variable noOfTouchableVertices which notes the number of vertices a vertex can reach (excluding itself) if and only if the shortest path between the vertices is postive. This is where the getShortestPath method is implemented. The

```
public String[] getCenters() {
    ArrayList<String> center= new ArrayList<String>();
    for(vertex v:vertices){
        v.noOfTouchableVertices=0;
        int w=0;
        for(vertex u:vertices){
            if(0<getShortestPath(v.url,u.url)){
                v.noOfTouchableVertices++;
                if(w<getShortestPath(v.url,u.url)){
                    w=getShortestPath(v.url,u.url);
                }
            }
        }
        v.maxGst=w;
    }
```

Figure 3: Finds the eccentricity for all vertices in the *Wikipedia* graph

variable maxGst notes a vertex's max eccentricity. The next step is to work out the radius of the graph. This must be done in another loop as our eccentricity's of our vertices are now known while previously they were not. Also for a legal radius value the vertex with the minimum eccentricity will have to touch $|V|$ - 1 number of vertices , in other words that vertex's noOfTouchableVertices must be equal to $|V|$ - 1. The final step just involves finding all the vertices with eccentricity values of radius and noOfTouchableVertices equal to $|V|$ - 1 and adding their vertex url to some collection or the String[] itself that we are trying to return. In our case we used a arraylist to hold all the centers and used .toArray to return it as a array of strings. If no centers exist the method just returns a empty array.

## 3.2 Time Complexity

Our approach does the following:

1. Finds the eccentricity for each vertex
2. Find the value of the radius in the graph using eccentricity values from before
3. Finds all vertices with eccentricity of values of radius and ensures that these vertices are connected to their complimentary set of vertices
4. Return all centers found

Procedure 1 has to loop through |V| vertices |V| times and use get shortest path for each vertex. In the worst case scenario all vertices are connected to each other and therefore getshortestpath is run $|V|^2$ times. Since getshortestpath runs at the worst case of $O(|V|+|E|)$ time. Therefore procedure 1 takes $O(|V|^2(|V|+|E|))$ time, resulting in a worst case complexity of $O(|V|^3+|V|^2E)$ time.
Procedure 2 and 3 just loop through |V| vertices once to get their required result and procedure 4 takes O(1) to complete resulting in a overall worst case time complexity of $O(|V|^3+|V|^2E)$ time for this method.

## 3.3 Potential Improvements

Our implementation is evidently slow ,if we could improve our approach to this problem we would try to improve the time complexity of this method. One such way of going about this is to possibly implement Floyd Warshall's algorithm to accomplish procedure 1 which would taken only $O(|V|^3)$ time as opposed to $O(|V|^2(|V|+|E|))$ time. Resulting in a faster time complexity. However Floyd Warshalls algorithm is a recursively called and will have a worse memory complexity compared to the approach we took, especially for larger graphs.

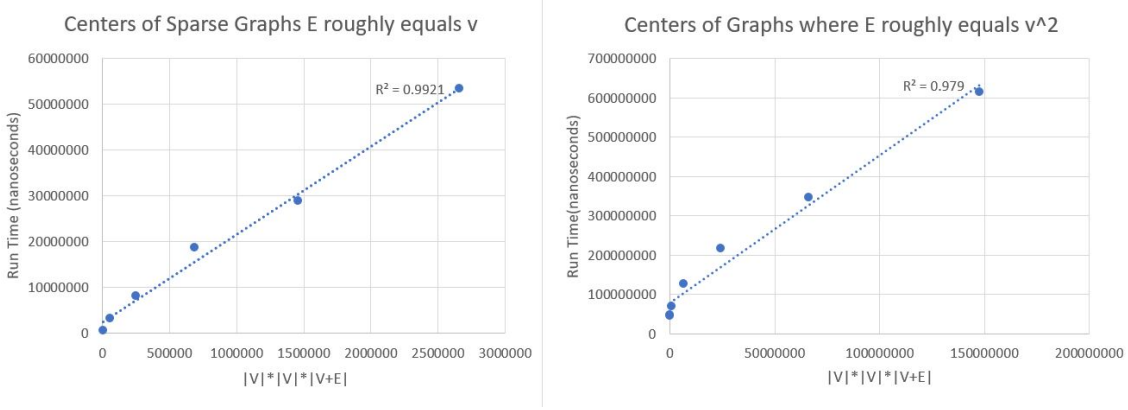


Figure 4: The graph on the left represents performance analysis for sparse graphs whereas the graph on the right represents performance analysis for heavily connected graphs. This is to demonstrate how the worst case time complexity of $O(|V|^3 + |V|^2 E)$ holds for different types of graphs

# 4 Strongly Connected Components

A graph is said to be strongly connected if each vertex has a path to every other vertex. The strongly connected components of a graph $G$ with vertices $V$ is a set $S$ such that $S \subseteq V$ and there exists a path between all pairs of vertices in $S$. The set $S$ also has to be complete. If there exists a vertex 'w' such that $w \notin S$ and w has a path to every vertex in $S$ and every vertex in $S$ has a path to w then the subset is incomplete, since $S \cup w$ is a strongly connected component. This function is to return all the strongly connected components of the graph.

## 4.1 Our Approach

This problem can be solved in linear time, we used Tarjan's algorithm (Tarjan 146-160). Fundamentally the algorithm uses a depth first search, recursively visiting each node once. It also considers each edge at most once. So the goal is to have a time complexity $O(|V| + |E|)$

Tarjan's algorithm runs the breadth first search only once and maintains a stack. It pushes the vertices onto the stack in the order it is visited by the search. They only get popped from the stack if there does not exist a path from that vertex to a previous vertex on the stack. In other words it is the root of a subgraph that is strongly connected.

```
for(vertex v:vertices){
    if(v.strongIndex == -1) index = strongconnect(v, index, s, verticesInOrder);
}
```

Figure 5: Starts the recursion for Depth-First Search

The algorithm has to assign an index to every vertex, the order of which it has been visited, it also needs to maintain a 'lowest' index one that indicates the 'earliest' vertex it can return to already in the stack. This allows us to keep the vertex on the stack if its $index < lowest$

To keep track of each index our vertex private class each has...

int strongIndex;
int strongLowest;
boolean onStack;

*onStack* is used to ensure that looking whether a vertex is on the stack happens in constant time, if we had to iterate through the stack we would lose our time complexity of $O(|V| + |E|)$. The function requires to return a string[][] of the strongly connected components. It is unknown

```
for (vertex adjV:v.adjacents){
    if(adjV.strongIndex == -1){
        index = strongconnect(adjV, index, s, inOrder);
        v.strongLowest = Math.min(v.strongLowest, adjV.strongLowest);
    }else if(adjV.onStack){
        v.strongLowest = Math.min(v.strongLowest, adjV.strongIndex);
    }
}
```

Figure 6: Continues the recursion for the Depth-First Search and calculates the indices

at the start how many components there are, and any declaration such as string[n][n] will have to be made smaller and take $n^2$ time. So to maintain linear time complexity our algorithm counts how many components there are using *private int numOfscc;* and a stack that contains all the vertices in order of their respective 'scc' where a vertex that has strongIndex = strongLowest marks the start of a new 'scc'. So when popping the vertices we fill this stack and increment *numOfscc;*

```
if(v.strongLowest == v.strongIndex){
    vertex w;
    do{
        w = s.pop();
        w.onStack = false;
        inOrder.push(w);
    }while (w.url != v.url);
    numOfscc ++;
}
```

Figure 7: When a vertex gets popped from the main stack we push it onto the 'inorder' stack to retrieve it when constructing the output.

Now when constructing the result we declare String[][][] scc = new String[numOfscc][]; and iterate through our inOrder stack to retrieve our components, maintaining our time complexity $O(|V| + |E|)$.

## 4.2 Potential Improvements

The continual recursion calls requires Java to maintain the previous method on the stack. Now in the worst case scenario where the the graph is very sparse and is such that it creates a something close to a graph where each node has only one descendant A->B->C->D... the recursion will continue for the amount of vertices there are and potentially run out of stack space (with a sufficiently large number of vertices). If this becomes a problem the algorithm can be implemented using a stack instead of recursion.

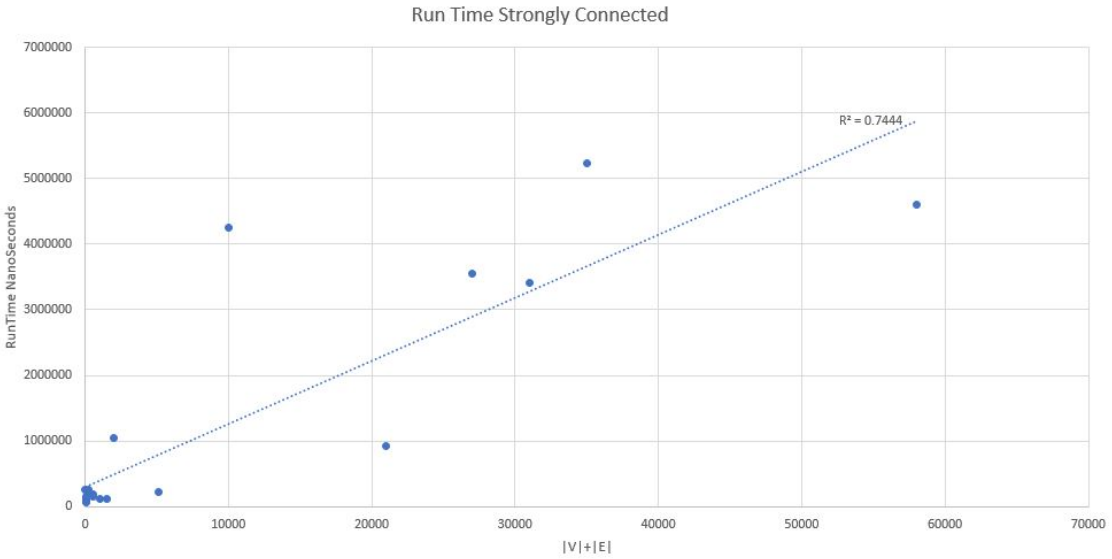


Figure 8: Run time analysis was done on the graph with vertices ranging from 10-10000 and edges 10-50000. The V+E is an upper bound complexity for scc, the run time is highly influenced on the type of graph. For example two graphs one with vertices = 50 and edges = 50 runs twice as fast as a graph with vertices = 10 and edges = 110, even though there respective V+E are 100 and 110

# 5 Hamiltonian Path

A Hamiltonian path in a graph is a path such that it visits each vertex exactly once. To solve this problem we need to find a complete sequence of vertices where $V_n, V_{n+1}$ is an edge. The obvious solution would be to check all the possible ordering of the vertices and see if any are Hamiltonian. Unfortunately this will take very long namely O(V!) and factorial function grows very rapidly.

The Hamiltonian path problem is a NP-Complete problem.
Meaning that no polynomial time algorithm has been found to solve this problem, nor has it been proven that no polynomial time algorithm can exist.

We took a dynamic approach to finding the Hamiltonian Paths (Held and Karp 196-210). Using this method you determine if the subsets of the graphs contain a Hamiltonian path. So let $G$ be our graph and $S \subseteq G$ if there exists a Hamiltonian path in $S$ ending on vertex 'v' and there exists an edge between 'v' and vertex 'w' such that $w \notin S$. It is clear then that $S \cup w$ has a Hamiltonian path ending on w.
Once all the subsets have been determined we can backtrack to find the path. This method requires all the subsets each with all the vertices to be stored. Since the $|\wp(G)| = 2^n$ the solution requires $O(2^n n)$ space where n is the number of vertices.

Each subset can be encoded with binary numbers example let our graph $G$ have N vertices each numbered 0 ... N-1. The subset '0100101' then contains the 1st, 3rd and 6th vertex. To compute a subset containing for example ABC we need to have computed A,B,C,AB,AC,BC already. So '111' requires '001' ... '010' ... '011' ... '100' ... Notice that these are just the binary equivalence to 1,2,3,4... So regarding the ordering of completing the subsets the natural progression from 0 to $2^n$ works correctly.

Let dp[subset][v] be our two-dimensional array of size $2^n n$ where it stores an integer containing the length of the shortest Hamiltonian path in the subset that ends on the vertex 'v'.

Two private methods are used to simplify the algorithm

boolean nbit(int n, int subset)
Which returns true if the subset contains the vertex at position n

int count(int subset)
Which returns the number of vertices contained in the subset

The dp can be calculated with the following formulas
dp[subset][v] = $\infty$; if nbit(v, subset) = false;
dp[subset][v] = 0; if count(subset) = 1 and nbit(v, subset) = true;
dp[subset][v] = $min_{nbit(w,subset)=1,isEdge(v,w)=true}\{dp[subset \text{ xor } 2^v][w] + 1\}$

```
if(nbit(v,m) == false) dp[m][v] = inf;
else if(countSubset(m, numV) == 1) dp[m][v] = 0;
else{
    min = inf;
    for(int j = 0; j < numV; j ++){
        if(nbit(j,m) && isEdge(vertices.get(j), vertices.get(v))){
            tmin = dp[(m) ^ (int)(Math.pow(2,v))][j] + 1;
            if(tmin<min) min = tmin;
        }
    }
    dp[m][v]= min;
}
```

Figure 9: Determines the dp

Notice the bit operation 'xor' in the third formula, we use this to 'lookup' the smaller subset without the vertex 'v'. Example if we are calculating if the set A,B,C has a path ending on B we want to lookup the subset A,C the mask equivalence for A,B,C is 111 and the binary equivalence of vertex 'v' is $2^v$ in this case $2^1 = 010$. The xor operation in this example 111 xor 010 = 101 gives us the required subset without the vertex v. The algorithm loops through each subset $O(2^n)$ and for each subset loops through each vertex $O(n)$ then if formula 3 is required loops through each vertex again $O(n)$ coming to a total of $O(2^n n^2)$
Now that the dp is filled we can check the 'subset' that contains the whole graph, if there exists a 'v' such that dp[$2^n - 1$][v] = n where n is the number of vertices we know that a Hamiltonian path exists. We remove 'v' from the subset and add it to our path, now finding what vertex 'w' on the smaller subset has a Hamiltonian path ending on it and the (w,v) is an edge. Do this repeatedly until the whole path is traced.

The above implementation can easily be altered to find the shortest Hamiltonian path if it is to be implemented for a weighted graph, instead of adding 1 in the third formula you would add the weight of (v,w).

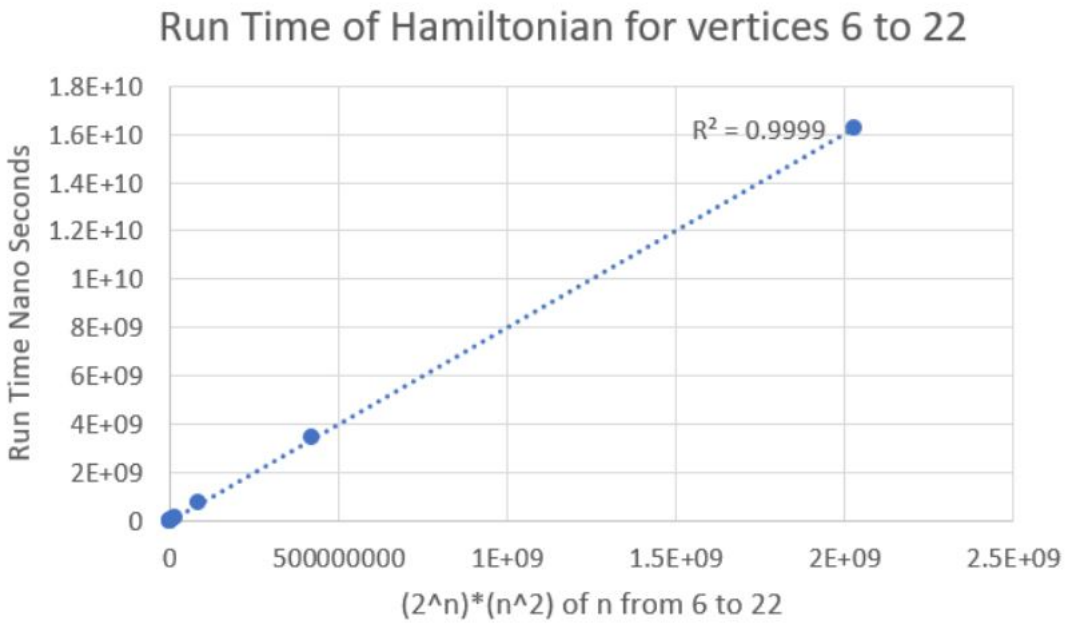


Figure 10: Run time analysis was done on graph with vertices size of 6 to 22, the edges had varying sizes but the Hamiltonian time complexity is not dependent on the number of edges, this can be seen very clearly by the high correlation coefficient

# 6 References

This report was written in *Overleaf* and any graphs was created with *Microsoft Excel*

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009). Introduction to algorithms (3rd ed., p. 595). Cambridge, Massachusetts: The MIT Press.

Jungnickel, Dieter. Graphs, Networks And Algorithms. 1st ed. [Place of publication not identified]: Springer, 2014. Print.

Tarjan, R. E. (1972), "Depth-first search and linear graph algorithms", SIAM Journal on Computing, 1 (2): 146–160

Held, Michael, and Richard M. Karp. "A Dynamic Programming Approach To Sequencing Problems". Journal of the Society for Industrial and Applied Mathematics 10.1 (1962): 196-210. Web.

Graph Theory: 51. Eccentricity, Radius and Diameter. (2015). YouTube. Retrieved 4 June 2017, from https://youtu.be/YbCn8d4Enos

Tarjan's strongly connected components algorithm. (2017). En.wikipedia.org. Retrieved 4 June 2017, from https://goo.gl/DBFO3j