CITS2200 Project Report
Henri Scaffidi
23245207

Graph Representation:

This project implements various algorithms to accomplish tasks related to graph analysis. The graph in question is the Wikipedia graph. The webpages of Wikipedia act as vertices of the graph, and links between webpages act as directed edges. The implemented class works with arbitrarily selected subsets of pages and links (although, part two is only executed if there is less than or equal to 20 pages in the graph, due to its computational cost).

This implementation represents the graph using an adjacency list - a list of lists, where each list contains the adjacent vertices of a vertex. This representation uses *O(E)* memory (where E is the number of directed edges in the graph). The algorithms in this class implementation do not require testing of edge existence. Therefore, the extra memory of an adjacency matrix, coupled with its redundant benefit of constant time edge existence checking, renders the adjacency list the best choice. The adjacency list is primarily created using the addVert() and addEdge() functions, creating representations for both the original graph and its transpose (for later use in Part 3). Unless otherwise stated, the implementations imply analysis and traversal of the original graph.

Since vertices are links/URLs in this context, we assign each URL an integer ID in the range [0, V), serving as the index into the adjacency list. Conversion between the URL string and the integer ID is enabled using an array of URL strings (which can be accessed using the IDs) and a hash map (which can be accessed using URLs to retrieve the ID). On average, both are constant time operations, allowing for an efficient solution.

Part One: Shortest Path

Algorithm:
A method which returns the minimum number of links to follow from one page to another is, effectively, a shortest path problem. Breadth First Search (BFS) was utilised for this method:

- Check both page URLs are in the graph. If not, return -1.
- Check if both page URLs are identical. If so, return 0.
- Utilise an array of distances, representing the shortest path distance to each page from the source page (initialize them all to very large number, except for setting the source page's distance to 0)
- Do BFS starting at the source page:
    - Initialize queue of pages and enqueue the source page
    - While the queue is not empty:
        - Dequeue the current page
        - For every page adjacent to the current page:
            - If distance of the adjacent page is more than the distance of the current page + 1:
                - Set distance of the adjacent page to distance of the current page + 1 (means we have found an improvement in path length)
                - Enqueue the adjacent page
- Check distance of the destination page from the source page:
    - If it is equal to the very large number originally set, no path exists, return -1.
    - Else, return the distance, as this will be the shortest distance between the source page and the destination page.

Breadth First Search, as an algorithm, was adapted here from the implementation taught in CITS2200 Lecture 12.

Complexity Analysis:
The initialisation of the distances array is *O(V)* time complexity. BFS visits every page, and every adjacent page of each page, which is *O(V + E)* time complexity. This appears as a time complexity, overall, of 2V + E. We take the dominant terms, ignoring constants, yielding worst-case time complexity of *O(V + E)* for the algorithm. The space complexity is also *O(V + E)* as the graph is stored in an adjacency list of size *O(V + E)*, as well as maintaining a queue and a distance array of size V. This analysis demonstrates the algorithm is optimal. The use of Dijkstra's algorithm was also considered, however it serves the identical function as BFS for an unweighted graph, such as the Wikipedia graph. It is simply a slower option in this context.

Performance Study:

V	8	10	12	14
E	10	10	15	18
V + E	18	20	27	32
Time (ns)	73802	80403	103425	117924

Four test graphs were utilised for this performance study. Each graph's structure was relatively similar, and the shortest path was computed between two points which were furthest away from each other in that graph. Hence, the primary variable being manipulated was V + E. As expected, the execution time varied approximately linearly with respect to V + E, confirming the previous complexity analysis.

Part Two: Hamiltonian Path

Algorithm:
The algorithm below is a backtracking Depth First Search (DFS). The idea is sourced from the CITS2200 Project Editorial.

- Check if number of graph vertices exceeds 20. If so, return empty array.
- For each page URL (**starting vertex** for DFS):
    - Initialise Stack called pathTracker to track vertices in the Hamiltonian Path
    - Initialise array of integers, where 2 is fully explored (black), 1 is partially explored (grey) and 0 is unvisited (white). This represents colouring of each vertex.
    - Call dfsHamiltonian(), beginning at **starting vertex,** which does:
        - Mark current vertex as grey and push it into pathTracker
        - For each adjacent vertex of current vertex
            - If adjacent vertex is white
                - Call dfsHamiltonian, beginning at adjacent vertex, and return true if this call returns true
        - If the size of the pathTracker stack is equal to the number of vertices in the graph (we've created a Hamiltonian path)
            - Return true
        - Otherwise, set the current vertex colour back to white, and pop it from the stack (since it's the last one that's been pushed). Return false. Effectively, this symbolises hitting a dead end in the DFS and needing to backtrack.
    - If the main dfsHamiltonian() call returns true (path was found):
        - Pop all the elements off the stack and insert them into an array (from back to front), constructing the path to return. Return the array.
- If the implementation reaches this point, no path was found, so return an empty array.

Complexity Analysis:
The initialisation of the colour array and the pathTracker stack each take $O(V)$ time complexity, where V is the number of vertices. The helper function dfsHamiltonian() is called from each vertex and traverses all possible permutations of the vertices for each of these calls, yielding an overall worst-case time complexity of $O(V\,V!)$. The space complexity is $O(V + E)$ as the graph is stored in an adjacency list of size $O(V + E)$. Additionally, a colour array of size V and a pathTracker stack that can store up to V elements are maintained. Given the inherent complexity of the Hamiltonian Path problem, no polynomial-time algorithm is currently known, and this implementation is among the most efficient possible. The Held-Karp algorithm has a better time complexity of $O(V^2 2^V)$. It was not implemented in this case simply due to difficulty and unfamiliarity with bitsets and binary operations.

Performance Study:

V	8	10	12	14
V V!	322560	36288000	5748019200	1.22×10^{12}

Time (ns)	384021	460036	544892	945421

Four test graphs were utilised for this performance study. Each graph's structure was relatively similar. Hence, the primary variable being manipulated was V. This demonstrates the fast increase in execution time due to the factorial term in the time complexity. The study further cements the choice of only allowing for graphs of V <= 21 to undergo this algorithm.

Part Three: Strongly Connected Components

Algorithm:
This implementation utilises Kosaraju's algorithm. This algorithm is based on the observation that the strongly connected components of the original graph are identical in the transpose. Since it relies on having the transpose of the graph, it is created as an adjacency list at the same time as the original graph's adjacency list is created.

- Initialise array of integers, where 2 is fully explored (black), 1 is partially explored (grey) and 0 is unvisited (white). This represents colouring of each vertex. Colour them all white.
- Declare a stack called postOrder to track the order in which the first Depth First Search (DFS) traverses the vertices.
- For each **vertex**:
    - If **vertex** is white:
        - Call dfsOG() starting at this **vertex**
            - Mark the current vertex as grey
            - For each adjacent vertex of the current vertex in the non-transposed graph
                - If the adjacent vertex is white
                    - Call dfsOG() starting at the adjacent vertex
            - Mark the current vertex as black (fully explored)
            - Push the current vertex onto the postOrder stack to record the finish time order
- Reset all the colours to white
- Declare a 2D ArrayList to hold the strongly connected components (SCCs). Will call this **components**.
- While the postOrder stack is not empty:
    - Pop vertex off the stack.
    - If the vertex is white:
        - Create a new ArrayList to hold a new SCC
        - Add the vertex to the SCC
        - Call dfsBack() starting at the vertex:
            - Mark the current vertex as grey
            - For each adjacent vertex of the current vertex in the **transposed graph**
                - If the adjacent vertex is white
                    - Add the adjacent vertex to the SCC
                    - Call dfsBack() starting at the adjacent vertex
            - Mark the current vertex as black
        - Add the finished SCC to the **components** array
- If the size of the **components** array is 0, return an empty array.
- Otherwise, the **components** array represents an array of the strongly connected components of the graph. In our implementation, this was a 2D ArrayList, which we then copied to a 2D Array, and returned it.

Complexity Analysis:

The execution of Kosaraju's algorithm first initiates a DFS on the original graph, tracking vertices' post-order in a stack. This operation, along with initializing the colour array, has a time complexity of $O(V + E)$, as it visits all vertices and edges. Then, the colour array is reset, which incurs $O(V)$ time complexity. The second phase performs DFS on the transposed graph, again costing $O(V + E)$, while tracking the strongly connected components. Thus, overall, the algorithm exhibits a time complexity of $O(V + E)$ considering the dominant terms. In terms of space complexity, it's governed by the colour array and stack, with each storing at maximum V elements. However, the adjacency lists of size $O(V + E)$ again overshadows this, resulting in an overall space complexity of $O(V + E)$.

Performance Study:

V	8	10	12	14
E	8	10	12	14
V + E	16	20	24	28
Time (ns)	413935	471352	520353	573593

Four test graphs were utilised for this performance study. Each graph's structure was relatively similar. Hence, the primary variable being manipulated was V + E. As expected, the execution time varied approximately linearly with respect to V + E, confirming the previous complexity analysis.

Part Four: Graph Centres

Algorithm:
As per the CITS2200 Project Editorial, this implementation assumes two things:

- The eccentricity of a vertex is the maximum length shortest path **from** the vertex to any other
- Centres of the graph are the vertices with the minimum eccentricity

Therefore, in a directed graph, such as the Wikipedia graph, vertices which are dead ends yield eccentricities of 0 (in this case), and thus are centres.

Breadth First Search (BFS) was utilised for this method:

- Initialise array of distances (size = V)
- Initialise minEccentricity to very large value (Integer.MAX_VALUE was used)
- Initialise ArrayList of centers
- For each **page**
    - Fill distance array with Integer.MAX_VALUE
    - Set distance of **page** to 0
    - Do BFS starting at **page (source)**:
        - Initialize queue of pages and enqueue the **source**
        - While the queue is not empty:
            - Dequeue the current page
            - For every page adjacent to the current page:
                - If distance of the adjacent page is more than the distance of the current page + 1:
                    - Set distance of the adjacent page to distance of the current page + 1
                    - Enqueue the adjacent page
    - Initialise **eccentricity** to -2
    - Iterate through distance array, find the max value that is NOT Integer.MAX_VALUE, and set **eccentricity** to this value
    - If **eccentricity** is less than minEccentricity, set minEccentricity to **eccentricity,** clear the centers ArrayList, and add the **page** to the centers
    - Else if **eccentricity** is equal to minEccentricity, add the **page** to the centers
- If the size of the centers ArrayList is 0, return empty array.
- Otherwise, copy the centers ArrayList to an array and return it.

This implementation was primarily sourced from the CITS2200 Project Editorial.

Complexity Analysis:
This implementation of the breadth-first search (BFS) algorithm is deployed to identify the centers of the page graph. The algorithm first initializes a distance array of size V, equating to a time complexity of *O(V).* The BFS traversal is then performed for each vertex in the graph, marking the time complexity as *O(V(V + E)),* since each vertex and edge are processed once per vertex. The time complexity of finding the eccentricity, i.e., the maximum distance from one vertex to all other vertices, is *O(V)* as it involves a scan through the distance array. This operation is also performed V times, once for each vertex, adding another $O(V^2)$ to the time complexity. Overall, considering dominant terms, the algorithm exhibits a time complexity of $O(V^2 + VE)$. In terms of space complexity, it's primarily

governed by the distance array and BFS queue, both storing at maximum V elements. However, the adjacency list of size *O(V + E)* again overshadows this, resulting in an overall space complexity of *O(V + E).*

Performance Study:

V	8	10	12	14
E	8	10	12	14
$V^2 + VE$	128	200	288	392
Time (ns)	70241	75294	84235	104217

Four test graphs were utilised for this performance study. Each graph's structure was relatively similar. This demonstrates the quadratically increasing execution time, in alignment with the time complexity of the algorithm.