CITS2200: Data Structures and Algorithms Wikipedia Graph Project

Written by Jeremy Boulter SN:22712171

Introduction/Preface

Graph theory is a discipline of mathematics that concerns modelling the interconnectivity of the world around us. In the evermore interconnected digital world, these relationships can be found in vast networks such as Wikipedia, where pages upon pages of information are connected through a complex web of hyperlinks. Given the vastness and complexity of the Wikipedia network, understanding it's structure and characteristics can be a daunting task, but is beneficial for several reasons. It allows people to study the relation of certain topics to one another in a systematic way and create new insights about the dynamics of information sharing in this massive online encyclopedia. More importantly, the study of the Wikipedia network provides valuable empirical data for testing various theoretical models of network formation and analysis. Due to the incredible scale of Wikipedia (with over 6.5 million articles) the assignment asks us to construct a subset of linked pages as vertices, and the link from one page referencing another as a directed edge. In doing this, we can construct a model of the relationships between topics as an unweighted, directed graph.

The task at hand and the topic of this report is the creation of this structure in Java, and the implementation of the given interface and its methods. The methods in the interface along with their desired operation are as follow:

(Q1) Shortest Path

Given two URL's, return the minimum number of links (or edges) that need to be travelled from the first URL in order to reach the second URL.

(Q2) Hamiltonian Path

Calling this method will return a path that visits every page (or vertex) exactly once, given that path exists.

(Q3) Strongly Connected Components

Calling this method returns sets of pages that are 'strongly connected', which means all the pages in the set can reach one another.

(Q4) Centres

This method returns the page that is the centre of the graph; the point from which the most distant other page is as close as possible.

Constructing the Data Structure

As specified in the project editorial I used an adjacency list to represent the elements of the graph. In the java implementation provided; and metaphorically, this can be thought of as a "list of lists" where each list represents a page on Wikipedia, and its elements representing out-going links to other pages. The choice to use an adjacency list rather than a matrix is based on the fact our algorithms will not require us to verify the existence of an edge at any time, but will use the list of out-going edges of a vertex quite frequently, which is done in constant time. For the sake of time efficiency over memory I also chose to store a transposed version of the graph concurrently for use later to compute the strongly connected components as seen later. Our representation uses a unique page ID to reference a given URL to make things less cumbersome and introduces two other data structures to convert between the two efficiently. A list of URL's where the index of a given page corresponds to its ID, and a hash map that maps a URL to its corresponding ID. My answers to the questions will be based on these structures in the following.

Q1: Shortest Path

In order to find the shortest path between two pages I made use of a Breadth-First search algorithm. I first initialise an array the size of the graph and fill it with -1 to represent infinity or "unvisited". The algorithm works by following each available path to a given vertex and stores the distances of its neighbours in the array. Once a vertex is seen as reachable, it's added to a queue. Once all neighbours of the starting vertex are recorded in the distance array, the algorithm then repeats itself for the neighbours of the starting vertex's neighbours. This continues until every vertex reachable to the starting vertex has been checked and their distances from the start are recorded. We then simply check the distance taken to reach our desired endpoint and it will always be the shortest path possible. Since we visit every vertex and every edge exactly once in the worst case, this algorithm can be said to have O(V + E) time complexity which is optimal.

Pseudocode:

1. Initialise a distance array with its size equal to the number of vertices and fill it with -1
2. Set the distance of the source vertex to zero
3. Initialise a queue and add the source vertex
4. While the queue is not empty
    1. Dequeue a vertex
    2. For each neighbour of this vertex
        1. If it has not been visited before
            1. Set its distance to +1 of the current vertex's distance
            2. Add it to the queue
5. Return the distance of the end vertex

Q2: Hamiltonian Path

Finding a Hamiltonian path is a famous problem in computer science and graph theory. The goal is to find a path that visits every vertex exactly once. This problem is renowned for it's difficulty, as there is no polynomial time solution for it. I chose to implement the Held-Karp algorithm as it's one of the best currently known algorithms for solving this problem. The main idea of the Held-Karp algorithm is to use dynamic programming to systematically explore all subsets of vertices and their associated paths. This is achieved by defining a 2D array dp, where dp[i][j] is the shortest path that visits every node in the subset j exactly once, and ends at i. The subsets are represented as binary numbers, where each bit corresponds to whether a vertex is included in the subset or not. This technique is known as bitmasking. In each step, we update dp[i][j] by considering all vertices k that are included in subset j and have a path to vertex i. We aim to minimize dp[i][j] by choosing the optimal vertex k. After filling the dp table, we find a vertex that can be an end of a Hamiltonian path by checking each vertex i and subset j which includes all vertices. This vertex will be the one that minimizes dp[i][j].

The final step is to reconstruct the Hamiltonian Path. We backtrack from the selected vertex using the next array until we reach the start of the path. The time complexity of the Held-Karp algorithm is $O(V^2 2^V)$ because there are 2^V subsets of vertices and it takes $O(V^2)$ time to iterate over all pairs of vertices in each subset. Due to the exponential nature of this solution, the algorithm becomes less effective as the size of the graph increases. To illustrate this, if you were to compute the Hamiltonian path at 1 million operations per second, a graph with 10 vertices may take up to 10 seconds, while a graph with 30 vertices may require multiple days.

Pseudocode:

1. Initialise a Dynamic Programming array and a next array with size N x 2^n
2. For each vertex, set dp[i][1 << i] = 0
3. Initialize a DP array and a next array with size n * 2^n
    1. Iterate over all subsets of vertices in increasing order of size
        1. For each vertex in this subset
            1. For each other vertex in the subset
                1. If the distance from the other vertex to this vertex is smaller than the current distance, update the distance and set the next vertex
4. Find a vertex that can be the end of a Hamiltonian Path
5. If such a vertex exists, reconstruct the Hamiltonian Path by backtracking using the next array. If not, return an empty array

Q3: Strongly Connected Components

As mentioned earlier, the Strongly Connected Components of a directed graph consists of subgraphs of vertices that are all able to reach each other. I used Kosaraju's algorithm to solve this problem which makes use of two depth-first searches. The algorithm employs two Depth-First searches in order to find the strongly connected components. First, the algorithm performs a DSF on the originally graph, and while traversing it tracks the order in which the vertices are found and adds them to a stack. The second DSF is on the transposed graph mentioned earlier (where all the directions of the edges are reversed), in the order of the stack made from the first search, only if they have not been visited yet. All nodes reachable from the starting node will form a strongly connected component, which is stored in the "component" array. The algorithm uses the property of a directed graph that if A and B are in the same strongly connected component there must be a path from A to B in the normal graph, and a path from B to A in the transposed version. Due to this graph requiring two full graph DFSs, each of which visit every vertex exactly once and consider each edge leaving the vertex exactly once and consider each edge leaving exactly once, giving a time complexity of O(V + E).

Pseudocode:

1. Initialise a stack
2. Perform DFS on the graph and push vertices onto the stack in order of their finishing times
3. Reverse the graph
4. While the stack is not empty
    1. Pop a vertex from the stack
    2. If it has not been visited yet
        1. Perform a DFS from this vertex and add all visited vertices to a new component
5. Return the strongly connected components

Q4: Centres

The centre of a graph is a vertex that minimizes the maximum distance to any other vertex, or in other words has the lowest eccentricity. The eccentricity of a vertex V in a graph is the greatest distance between v and any other vertex. Therefore, finding the centres of a graph simply means comparing the farthest reachable vertex of every vertex, and finding the one with the lowest path.

This seems somewhat ambiguous to me as in a directed, unweighted graph, a singular sink vertex (a vertex with no out-going edges) seems to be a centre according to this definition. I digress however I wanted to make this clear as it's what my solution will produce in that case. This is a fairly straightforward algorithm; perform a Breadth-First search on every vertex to find the maximum value from the distances produced, whichever vertex has the lowest, furthest vertex is then considered a centre. This approach requires 1 BFS per vertex, with each BFS requiring the consideration of every vertex and every edge. Therefore, the time complexity of finding the centres of a graph is O(V (V + E).

Pseudocode:

1. Initialise a list of centres and set the minimum eccentricity to infinity
2. For each vertex
    1. Perform a BFS from this vertex to compute the distances to all other vertices
    2. Compute the eccentricity of this vertex (the maximum distance to any other vertex)
    3. If this eccentricity is smaller or equal to the minimum, update it accordingly and reset the list of centres if needed
3. Return the centres