CITS2200 Data Structures and Algorithms

Project 1 2023: Graph Representation

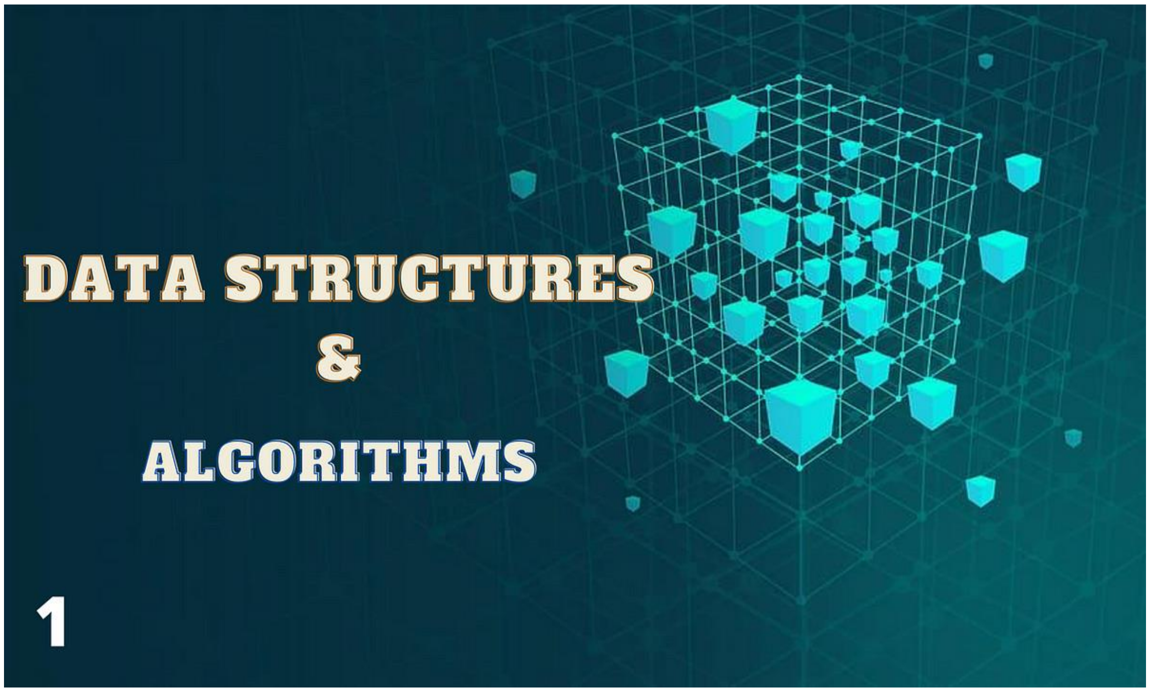

Image 1: Towards Data Science

University of Western Australia

Professor: Amitava Datta

Author 1: Ryan Alexander Hartono 23360449

Author 2: Edward Le 23020568

Ryan Alexander Hartono 23360449

Edward Le 23020568

Introduction

The goal of this project is to develop a class that can store and analyze various features of a Wikipedia page graph. The class will work with subsets of pages and the links between them. The main tasks to be implemented are as follows:

1. Write a method to find the minimum number of links required to reach from one page to another.
2. Write a method to find a Hamiltonian path in the page graph.
3. Write a method to find every strongly connected component of pages.
4. Write a method to find all the centers of the page graph.

This report will provide an overview of the implemented algorithms, their complexity analysis, and performance studies on different size graphs

Method Descriptions

Question 1: Minimum Number of Links

To find the minimum number of links between two pages, we utilize the breadth-first search (BFS) algorithm. BFS explores all the neighbors of a vertex before moving to the next level. We apply BFS starting from the source page and terminate when the target page is reached. The level at which the target page is found represents the minimum number of links required. The graph is represented using an adjacency list data structure. Each vertex in the graph corresponds to a unique URL. The way the algorithm works is described below:

1. We start by initializing an array called distance to keep track of the distance from the starting vertex to each vertex in the graph. We set the initial distance of all vertices to -1 to indicate that they haven't been visited yet.
2. We enqueue the starting vertex and set its distance to 0.
3. While the queue is not empty, we dequeue a vertex and explore its neighbors.
4. If a neighbor vertex has not been visited before, we update its distance as the distance of the current vertex plus 1 and enqueue the neighbor vertex.
5. We repeat steps 3-4 until we either reach the destination vertex or traverse all reachable vertices.
6. Finally, we return the distance of the destination vertex, which represents the shortest path in terms of the number of links. If there is no path, we return -1.

The worst-case time complexity of a Breadth First Search algorithm is **O(V + E)**. For each vertex, we examine its neighboring vertices. In the worst-case scenario, where every vertex is connected to every other vertex, the number of edges **E** becomes significant. Thus, the time complexity of BFS becomes **O(V + E)**, where **V** represents the number of vertices and **E** represents the number of edges. The algorithm traverses each vertex once and processes each edge once, encompassing the entire graph.

For the space complexity of BFS, As we traverse the graph, we need to keep track of visited vertices to avoid revisiting them. Additionally, we employ a queue to manage the order of exploration. The space complexity, therefore, is determined by the maximum size of the queue and the storage for visited vertices.

In the worst-case scenario, where all vertices are enqueued before being dequeued, the queue's size reaches its maximum. As a result, the space complexity of BFS is **O(V),** where **V** represents the number of vertices. This is because we need to store all vertices in the queue during the exploration process.

PERFORMANCE ANALYSIS

The execution time is found by using the System.nanoTime method, as shown below

```
long startTimescc = System.nanoTime();
String[][] stronglyConnectedComponents = proj.getStronglyConnectedComponents();
long endTimescc = System.nanoTime();
```

Then the nanoTime is converted to seconds. (note other questions will also use nanoTime for performance analysis)

Execution time for 5 edges: 1.834E-4 seconds

Execution time for 50 edges: 2.137E-4 seconds

Execution time for 500 edges: 3.302E-4 seconds

Question 2: Hamiltonian Path

A Hamiltonian path in a graph is a path that visits every vertex exactly once. As this question is a well-known computationally intensive problem, so there are at most 20 vertices. The algorithm applied for this question is backtracking, which behaves similarly to DFS, but it will backtrack to the previous vertice and mark it as “unvisited” if the Hamiltonian path is not found. The algorithm recursively explores until the path is found or no more vertex is left. Here is the flow of this algorithm:

1. Firstly, it checks the number of vertices in the graph, as mention above this algorithm is inappropriate for more than 20 vertices so it will return an empty array otherwise continue.
2. A boolean array “visited” to track visited vertices and an ArrayList “path” to store the Hamiltonian path are initialized.
3. The algorithm then loops through every start vertex, it starts by marking all the array “visited” as “false” meaning “unvisited” as well as remove all current element in the ArrayList “path” for every new vertex looped.
4. After that, it checks if there is already the Hamiltonian path of the current vertex and return that path, otherwise loop other vertexes.
5. Return an empty array in case no path is found after all loop.

The “hasHamiltonianPath” boolean method is basically the recursive DFS and backtracking method. It takes current vertex and mark as “visited”, also add that vertex to “path” list. The base case is when length of path is equal to length of the graph, it means all the vertex are added and the path is found, so this method returns true. This method keeps exploring unvisited vertexes recursively until the base case is met. If all visited neighbors have been looped through without finding the path, it backtracks the previous one, marking it as unvisited and remove current vertex from the path, finally return “false”.

The time complexity also depends on the numbers of verties and edges in the graph. This algorithm iterates through every possible start vertex in N times and performs DFS and backtracking inside the loop. In the worst case, we have to traverse every vertex with all paths as there are permutatons of each vertex, leaving the worst case time of V! So on, the time complexity in general including the loop is **O(V * V!).**

The space complexity of it is dominated by the space used for visited array or path arraylist. In general, the maximum space required is when the path is found, at that stage all vertex are included in the path list, leaving space complexity of **O(V).**

PERFORMANCE ANALYSIS

Execution time for 6 vertices: 3.07E-5 seconds

Execution time for 10 vertices: 1.058E-4 seconds

Execution time for 18 vertices: 4.42E-5 seconds

Question 3: Strongly Connected Components

To find the strongly connected components in the page graph, we can use Kosaraju's algorithm. Kosaraju's algorithm utilizes depth-first search (DFS) to perform two passes over the graph. The way the algorithm works is described below:

1. We first initialize an empty list called result to store the strongly connected components.
2. We perform the first DFS to determine the finishing times of each vertex. This helps us process the vertices in the correct order during the second DFS.
3. We process the vertices in descending order based on their finishing times.
4. For each vertex in the sorted order, if it has not been visited, we perform a DFS to discover the strongly connected component it belongs to.
5. We add each strongly connected component (represented as a stack of vertices) to the result list.
6. Finally, we convert the strongly connected components from stacks of vertices to string arrays of URLs.

The worst-case time complexity of Kosaraju's algorithm is **O(V + E)**. While traversing a Directed Graph which is represented using an adjacency list, we process each vertex and all the edges connecting it to its neighboring nodes. This gives us a net Time Complexity of **O(V)** + **O(E)** = **O(V + E)**, where **V** is the total number of vertices and **E** is the total number of edges.

In the second pass, we explore the transpose graph, reversing the direction of edges, which takes **O(E)** time. Then again, we perform a DFS on each vertex, visiting only those that haven't been visited in the first pass. It effectively utilizes the information obtained from the first DFS traversal to efficiently find SCCs in the transposed graph during the second DFS traversal. Once more, this second pass has a linear time complexity of **O(V + E)**. The sum **O(V + E)** + **O(V + E)** simplifies to **O(2V + 2E),** but we can drop the constant coefficient, resulting in **O(V + E**).

Kosaraju's algorithm uses stack to store the vertices while performing first DFS so the space used by stack is equal to size of the vertices of the graph. So, the space complexity of kosaraju's algorithm is **O(V**), where **V** is the total number of vertices in the graph.

PERFORMANCE ANALYSIS

Execution time for 5 edges: 2.7E-4 seconds

Execution time for 50 edges: 0.0010654 seconds

Execution time for 500 edges: 0.0016351 seconds

Question 4: Graph Centers

The implementation aims to find the center(s) of a page graph. It calculates the eccentricity of each vertex in the graph, where the eccentricity is the maximum shortest path distance from a vertex to any other vertex. It utilizes a breadth-first search (BFS) algorithm to find the shortest path distances from a vertex to all other vertices. The algorithm iterates over each vertex, performing BFS to calculate the maximum distance. It stores the eccentricities in an array and then finds the minimum eccentricity value. Finally, it identifies the vertices with the minimum eccentricity as the center(s) of the graph. To find the centers, we performed the following steps:

1. We calculated the eccentricity of each vertex, which is the maximum length of the shortest paths from that vertex to all vertex in the graph using BFS algorithm.
2. We stored all the eccentricites in an array called “eccentricities”.
3. Next, we found the minimum eccentricity value from the eccentricities array using findMinimum method.
4. We created an empty ArrayList called centerList to store the URLs of the vertices that its eccentricity is equal to the minimum eccentricity.
5. Finally, we returned the centerList as an array of URLs, representing the centers of the graph.

The time complexity of this algorithm is dominated by the BFS called in a nested loops. In this case, the loop iterates through every vertex, and each vertex performs the loop through every vertex again including itself to calculate distance from this vertex to all vertex. In the nested loops, the BFS method with time complexity of O(V + E) as mentioned above is called to calculate the distance, leaving the general time complexity of **O(V^2 * (V + E))** which reduces to **O(V^3 + V^2 E).**

The space complexity of the algorithm is determined by the amount of memory required to store data structures and variables during its execution. In this case, the algorithm uses an array to store the eccentricities of vertices, which requires space proportional to the number of vertices in the graph **O(V)**. Additionally, an ArrayList is used to store the URLs corresponding to the center vertices, which can be at most **V** (all vertices are centers). Therefore, the space complexity for centerList is **O(V)**. Overall, the space complexity is proportional to the number of vertices in the graph.

PERFORMANCE ANALYSIS

Execution time for 5 edges: 2.7E-4 seconds

Execution time for 50 edges: 0.0010654 seconds

Execution time for 500 edges: 0.0016351 seconds

REFERENCES

Understanding Hamiltonian Path, backtracking algorithm:

YouTube video: "Hamiltonian Path and Backtracking" - https://www.youtube.com/watch?v=dQr4wZCiJJ4

YouTube video: "Backtracking - Hamiltonian Path" - https://www.youtube.com/watch?v=Q4zHb-Swzro&t=614s

CodeCrucks ariticle: “Hamiltonian Cycle using Backtracking” -

https://codecrucks.com/hamiltonian-cycle-using-backtracking/

Understanding Strongly Connected Components:

GeeksforGeeks article: "Tarjan's Algorithm to find Strongly Connected Components" - https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/

YouTube video: "Tarjan's Algorithm for Strongly Connected Components" - https://www.youtube.com/watch?v=5wFyZJ8yH9Q

YouTube video: "Strongly Connected Components - Kosaraju's Algorithm" - https://www.youtube.com/watch?v=RpgcYiky7uw&feature=youtu.be

OpenGenus IQ article: "Kosaraju's Algorithm for Strongly Connected Components" - https://iq.opengenus.org/kosarajus-algorithm-for-strongly-connected-components/

GeeksforGeeks article: "Strongly Connected Components" - https://www.geeksforgeeks.org/strongly-connected-components/

YouTube video: "Graph Theory: Strongly Connected Components" - https://www.youtube.com/watch?v=ee6zIj4J3-Y&feature=share (covers adjacency lists)

Understanding Graph Centers:

YouTube video: "Graph Theory - Centers of a Graph" - https://www.youtube.com/watch?v=nzF_9bjDzdc&t=201s

Image 1:

Towards Data Science article: “Data Structures and Algorithms Journey” -

https://towardsdatascience.com/data-structures-and-algorithms-journey-80d225cfbbd8