CITS2200 Data Structures and Algorithms

Final Project Report

Fang Kai Gan 21896665
Michael Dorrell 21989175

A report for final project for the unit CITS2200 at
The University Of Western Australia
June 2016

Introduction

This report is written as part of the submission process for the final project for the unit CITS2200 - Data Structures and Algorithms. This report will discuss about the implementation of algorithm we did to complete the project. The report also explains the thought process behind deciding on each algorithm and why we think it is the best implementation. Furthermore, we will also include pseudocodes, complexity analysis and performance studies to fulfil the requirements stated in the project prompt.

Graph Implementation

In this project, we were not given a graph implementation as we did in the lab throughout the semester. We wrote a graph class to represent a graph. The graph is implemented internally as an arraylist. Every time an edge is added, the arraylist expands. And calling the method getIntArray returns the graph as an adjacency matrix.

Question 1 - Shortest Path

Question 1. *Write a method that, given a pair of pages, returns the minimum number of links you must follow to get from the first page to the second.*

1.1 Analysis

The requirement of the question requests that we write an algorithm that returns the number of links to get from the shortest legal path/link between two pages. If none are found, return -1 to indicate absence.

1.2 Algorithm and Pseudocode - Breadth First Search

We believe the most efficient and fastest algorithm to find the links is to use Breadth First Search(BFS). The pseudocode for the implemented algorithm is implemented below. It is adapted from the course reader.

```
procedure BFS(V_start ,V_end)
Push V_start to the tail of Q
while Q is not empty
        Pop vertex w from the head of Q
        for each vertex x adjacent to w do
                if x is not visited then
                        dist[x] ← dist[w]+1
                                If x is  V_end exit entire loop
                        visited[x] ← true
```

```
                    Push x to the tail of Q
                end if
            end for
            visited[w] ← black
end while
```

1.3 Algorithm Complexity

During the search, each vertex is enqueued and dequeued once from the queue. As each operation takes constant time, thus the entire process takes O(V) time. As each vertex is examined, the adjacency vertices are examined too. Therefore it takes O(E) time to inspect every adjacent vertex from the dequeued. Thus, the entire breadth first search operation takes a total time of O(V+E).

However, in this implementation, we included an early return/exit when the end vertex is found. Whenever the V_{end} is found, it must be the shortest path. The search operation is ended early. This will potentially reduce runtime as the algorithm do not need to transverse the entire graph when the endpoint is found.

1.4 Performance Studies

To illustrate the performance of this algorithm. The search is ran with random starting and ending vertices for 100 times for each density. Running each algorithm on a 2015 Macbook Pro machine on directed graph with densities 1,0.75,0.5,0.25 yields a relatively constant compute time in the magnitude of 10^{-3} milliseconds.

Question 2 - Hamiltonian Path

Question 2. *Write a method that finds a Hamiltonian path in a Wikipedia page graph. A Hamiltonian path is any path in some graph that visits every vertex exactly once. This method will never be called for graphs with more than 20 pages.*

2.1 Analysis

The question calls for a search for the Hamiltonian path. It requires that every vertex of the graph to be visited exactly once. The beginning and ending vertex of the graph will be different. However, if they are the same, it is called a Hamiltonian cycle. Thus a Hamiltonian cycle must also be a Hamiltonian path, but not vice versa. The algorithm returns the name in the order of the hamiltonian path in the form of String array and returns an empty array if it is not found. The algorithm is referenced from https://www.hackerearth.com/practice/algorithms/graphs/hamiltonian-path/tutorial/

2.2 Algorithm and Pseudocode - Dynamic Programming

Hamiltonian Path is said to be a NP-Complete problem. The obvious answer is to use brute force to produce all possible iterations or use a backtracking(depth first search) algorithm to transverse the matrix to find the path. However, these operations will be slow as they will take O(N!) operations to iterate through the entire graph. Thus, we used the Bellman-Held-Karp algorithm which uses dynamic programming to solve the problem. This algorithm is significantly faster than brute force/depth first search algorithm.

In our implementation, we used bitmasks to represent subsets.

```
function check_using_dp(int[][] graph)
n = number of vertices

    for i = 0 to 2^n
        for j = 0 to n
            dp[j][i] = false
    for i = 0 to n
        dp[i][2^i] = true
    for i = 0 to 2^n
        for j = 0 to n
            if j^th bit is set in i
                for k = 0 to n
                    if j != k and k^th bit is set in i and adj[k][j] == true
                        if dp[k][ i  XOR 2^j ] == true
                            dp[j][i]=true
                            break
for i to n
```

```
        for j to n
            if (dp[j][currentPath] == 1) {
                outputArray += dp[j][currentPath]
                currentPath = currentPath AND ~(1 << y);
                break;
            }
        }
    }
```

This algorithm also has an early exit implemented, if the last column in dp is found, the program is ended early. Essentially, the algorithm quits when any single path exists regardless if there is any other paths.

2.3 Algorithm Complexity

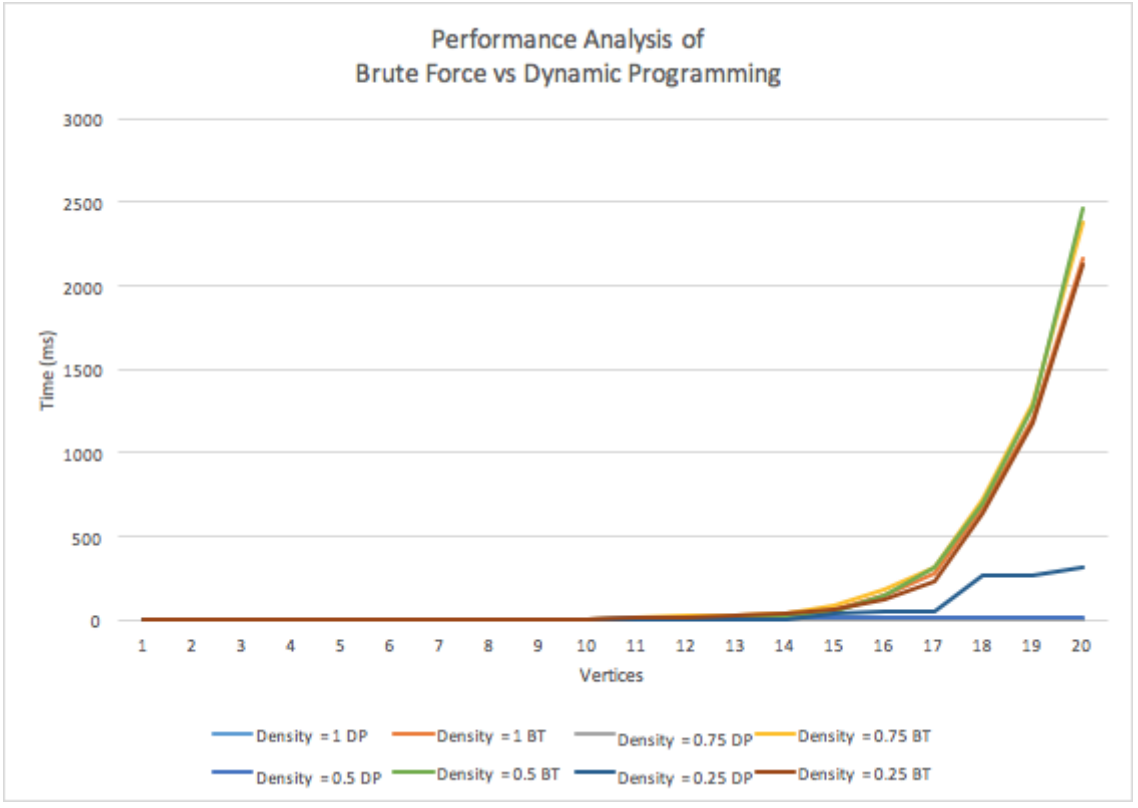
With this algorithm all subsets of the path is produced which is $O(2^n)$. Each subset is examined, and the vertex added is computed which is $O(n^2)$. Thus the complexity of this algorithm is $O(2^n \cdot n^2)$.

2.4 Performance Studies

While we have described the time complexity in doing the search, it is easier to visualise in real world terms. The graph below is some data to illustrate the performance of graph operations performed on a 2015 MacBook Pro. Time are measured in milliseconds. The graphs are generated using CITS2200 Graph class. While the numbers alone is not the best measure to show algorithm efficiency, we think it is sufficient to compare between backtracking(BT) and Held-Karp, dynamic programming (DP) algorithms.

Vertices	Density = 1		Density = 0.75		Density = 0.5		Density = 0.25	
	DP	BT	DP	BT	DP	BT	DP	BT
1	1.78	0.01	1.15	0.01	2.14	0.01	0.09	0.01
2	1.90	0.01	1.17	0.01	2.16	0.01	0.87	0.01
3	1.82	0.02	1.18	0.02	2.18	0.02	0.88	0.01
4	1.83	0.02	1.20	0.03	2.20	0.03	0.89	0.02
5	1.85	0.05	1.23	0.06	2.24	0.05	0.93	0.04
6	1.88	0.12	1.26	0.14	2.37	0.13	0.96	0.11
7	1.93	0.32	1.30	0.34	2.43	1.35	1.10	0.30

8	1.97	0.74	1.33	0.66	2.57	1.79	1.30	0.74
9	2.06	2.09	1.53	5.42	2.64	2.88	1.78	1.75
10	2.16	3.07	1.60	6.69	2.77	5.52	2.35	7.53
11	2.26	13.63	1.69	10.59	10.75	7.30	2.62	11.99
12	2.58	16.80	1.77	24.50	10.84	12.25	2.68	19.31
13	2.73	22.43	1.80	29.46	10.94	18.06	2.75	24.92
14	2.80	34.56	1.90	43.00	10.98	32.00	5.56	36.32
15	2.89	62.34	2.00	91.00	11.00	58.00	43.62	62.48
16	2.98	144.00	2.27	184.00	11.07	149.50	46.18	122.45
17	3.09	276.90	2.30	321.00	11.10	313.00	46.28	233.46
18	3.13	671.30	2.42	729.00	11.16	705.00	266.20	639.00
19	3.34	1203.00	2.47	1297.00	11.20	1275.00	269.00	1176.00
20	3.42	2157.00	2.56	2382.00	11.24	2457.00	317.00	2129.00



As according to the above graph, brute force programming takes exponential time compared to dynamic programming. At about 14 - 15 vertices, the time taken to calculate paths rises greatly while dynamic programming remains relatively constant.

Question 3 - Strongly Connected Component

Question 3. *Write a method that finds every 'strongly connected component' of pages. A strongly connected component is a set of vertices such that there is a path between every ordered pair of vertices in the strongly connected component.*

3.1 Analysis

This question requires a method that can return all strongly connected paths as a string array. The method provided uses Kosaraju's algorithm to perform 2 depth first searches to find the strongly connected components. The first search is to store (in a stack) the order in which every other vertex is reached from any arbitrary vertex. The second search uses a reversed version of the graph to find the set of vertices which can be traversed both forwards and backwards (a strongly connected component). This is repeated until all strongly connected components are found, then the result is converted to a string array (because it couldn't be stored this way initially) and returned.

3.2 Algorithm and Pseudocode

Kosaraju's Algorithm works based on the fact that a graph with reversed edges has the same strongly connected components as the original graph. This algorithm was written in english due to being dissatisfied with the lack of detail and coherency in the online pseudocode.

```
Main Body of Algorithm:
initialise Stack stack and Set visited
for every vertex:
        if (visited doesn't contain vertex)
                Subroutine DFS vertex
        end if
end for
reverse graph
clear visited
initialise result variable (To store all sets of strongly connected components)
while (Stack is not empty)
        vertex = Top of stack
        if (visited doesn't contain vertex)
                initialise temp ArrayList (To hold a set of strongly connected components)
                Subroutine reverseDFS vertex
                add temp to results
        end if
end while
return results
```

```
Subroutine DFS:
add vertex to visited
for every neighbour of vertex
        if(visited doesn't contain vertex)
                Subroutine DFS neighbour vertex
        end if
end for
add vertex to stack

Subroutine reverseDFS:
add vertex to visited
add vertex to temp ArrayList
for every neighbour of vertex in reversed graph
        if(visited doesn't contain vertex)
                Subroutine DFS neighbour vertex
        end if
end for
```

It should be noted that since an adjacency matrix was used in our implementation, the 'reverse graph' section of code could be ignored (it will be explained why we used this implementation in the next section). Due to this we only needed to replace adjacencyMatrix[vertex][index] with adjacencyMatrix[index][vertex] to achieve the same effect.

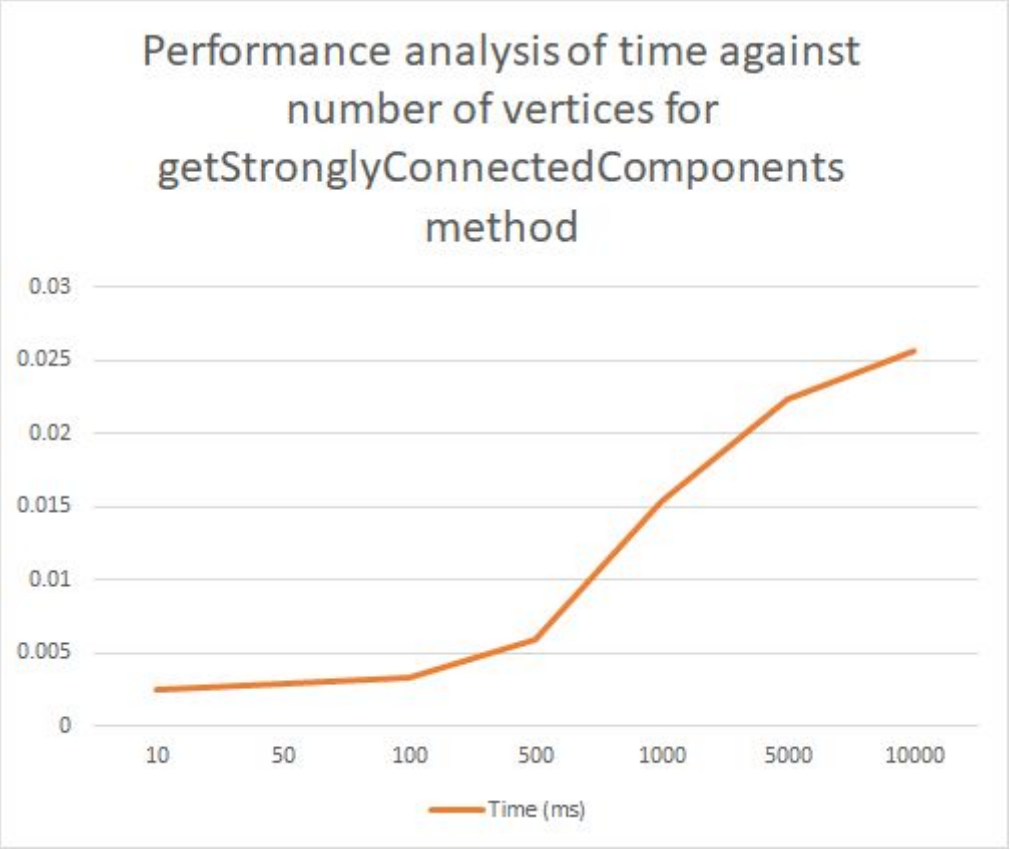
3.3 Algorithm Complexity

The complexity of our implementation of Kosaraju's algorithm is O(V^2) since we used an adjacency matrix. Although the efficiency could be improved to O(|V|+|E|) using an adjacency list this was not implemented because it would require O(V^2) time to reverse the graph using our storage implementation, which would require a complete overhaul of our solution to reduce to linear time. Since many other solutions used the adjacency list and work well with the current implementation, it was deemed impractical to rewrite the implementation to improve the time complexity of this solution.
Kosaraju's Algorithm was selected because it is an elegant and simple algorithm and considering the use of an adjacency matrix, Tarjan's algorithm and the path-based algorithm are not significantly faster despite being far more complex.

3.4 Performance Studies

This performance studies shows how the method's time complexity grows from 1-1000 vertices. This test was performed on windows 10 64-bit OS and the results were averaged out of 10 trials on a random graph of 0.8 density.

Number of Vertices	Time (ms)
10	0.00248
50	0.0029
100	0.00333
500	0.00598
1000	0.01539



It is illustrated in this graph that the time seems to be growing rapidly as the number of vertices increase. More input would be required to generate a more compelling graph, but the tester ran out of space and couldn't produce results above 10000 vertices.

Question 4 - Center of Graph

Questions 4. *Write a method that finds all the centers of the Wikipedia page graph. A vertex is considered to be the center of a graph if the maximum shortest path from that vertex to any other vertex is the minimum possible.*

4.1 Analysis

This question asks to return the set of all centers of a directed graph (as wikipedia pages are) as a string array. A center is any vertex by which every other vertex could be reached by the shortest path. The method provided uses the Floyd-Warshall algorithm to produce a matrix of the shortest paths, which is manipulated to find the centers.

4.2 Algorithm and Pseudocode

This solution is implemented by first producing an adjacency matrix, then using dynamic programming (Floyd-Warshall algorithm) to produce a matrix containing the length of every shortest path. The minimum eccentricity is then calculated by finding the smallest distance by which any vertex could reach all others, which is then used to search for each center and build the string array. Since the latter operations are trivial, only the Floyd-Warshall algorithm will be expanded upon.

The Floyd-Warshall algorithm uses dynamic programming to build up the shortest path matrix by updating the adjacency matrix. Each node in the graph is tested to see if it could shorten the path between every other node and the adjacency matrix is updated to reflect the results. By the nth iteration, the matrix of shortest paths is generated. The pseudo-code below was demonstrated in the lecture slides and represents the Floyd-Warshall algorithm.

$D^{(0)} \leftarrow A$
for k = 1 to V do
    for i = 1 to V do
        for j = 1 to V do
            $d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})$
        end for j
    end for i
end for k

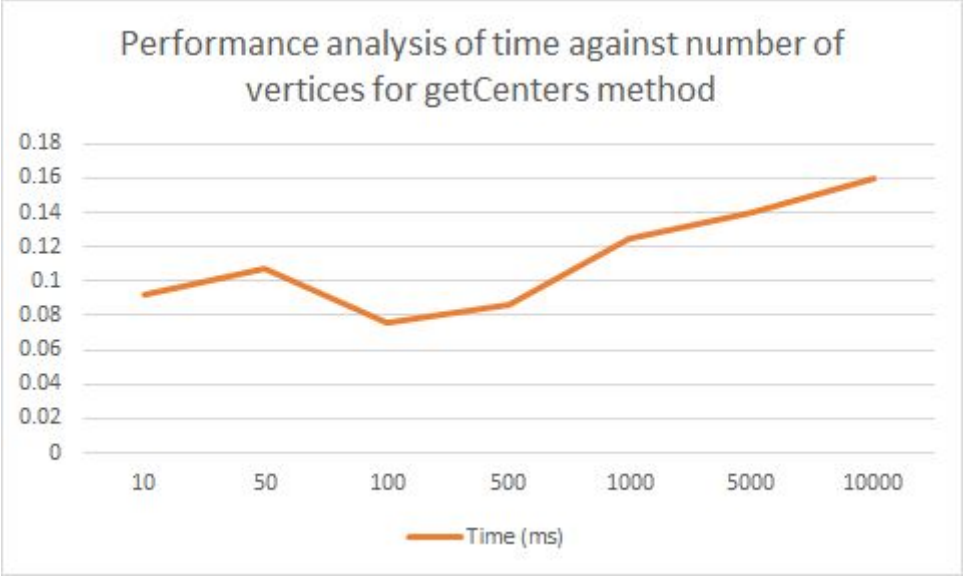
If there is no path between 2 vertices the value of the path should be infinity, in our implementation INFINITY was chosen as |v| + 1 because for an unweighted graph the maximum distance of a path could be |v| - 1 so INFINITY will never be reached for any amount of vertices tested. This is appropriate since the wikipedia pages generate an unweighted graph.

4.3 Algorithm Complexity

In our implementation the dominating operation is the Floyd-Warshall algorithm, which require $O(V^3)$ time. Unfortunately this is the base case, not the worst case because each element of the array must be examined V times. This leads to very regular complexity, no matter how dense the graph is, and Floyd-Warshall has very tight loops so it runs better than V calls to dijkstra's algorithm in practice, despite both running in $O(V^3)$ time.

4.4 Performance Studies

Number of Vertices	Time (ms)
10	0.09241419
50	0.10718126
100	0.07559898
500	0.08649623
1000	0.12485051
5000	0.14004412
10000	0.16029788



As can be seen in this analysis, there is an increase in time as more vertices are added. As in the previous test, the tester ran out of memory past 10000 vertices so more testing. This result certainly isn't to the extent of V^3, but more results may prove that relationship.