A close up of a logo

Description automatically generated

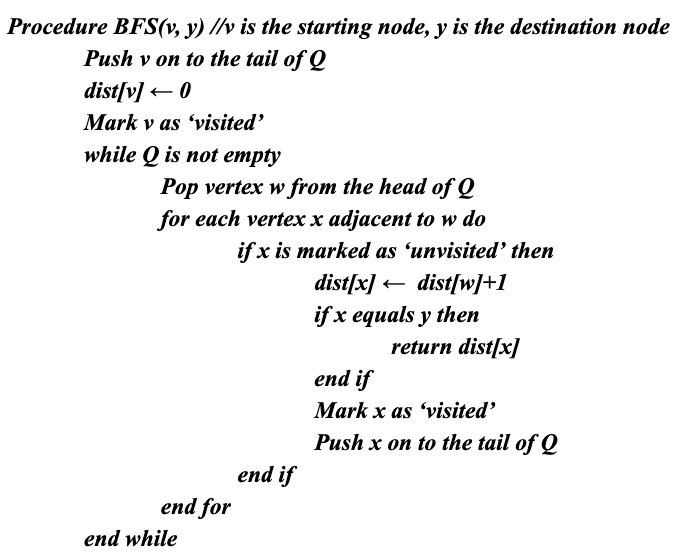
Analysing Graph Properties Within Wikipedia Pages

By Clayton Herbst (22245091)

Graphs are data structures that consist of a finite set of vertices and a set of ordered or unordered edges that form the link between these vertices (Wikipedia 2019). A single Wikipedia page (the vertex) can be considered a member of a set of Wikipedia pages (the graph) interconnected through hyperlinked URL’s (the directed edges) from which various characteristics and properties can be determined.

## Shortest Path Between Two Pages

In order to find the minimum number of URL links a user must follow to navigate from one Wikipedia page to another, a Breadth First Search (BFS) approach can be implemented starting at the specified starting vertex and ending at the destination vertex. The BFS algorithm starts at a vertex and construct a spanning tree for the given graph called the breadth-first tree (CITS2200 Topic 12) using a first-in-first-out queue abstract data structure. Modifying this common graph search technique allows the algorithm to keep track of the distance of each node within the breadth-first spanning tree from its root. Considering a Wikipedia graph, G, the pseudo code to find the shortest path between vertex *v* and vertex *y* can be seen in figure 1.



**Figure 1**

The time complexity of this shortest path algorithm implementation is O(V + E) in the worst case where V is the number of vertices in the graph and E is the number of edges in the graph. This has been derived by summing the time complexity of enqueuing all vertices and examining all edges which are O(V) and O(E) respectively (CITS2200 Topic 12). The BFS provides optimal time complexity across an unweighted graph (Wikipedia 2019).

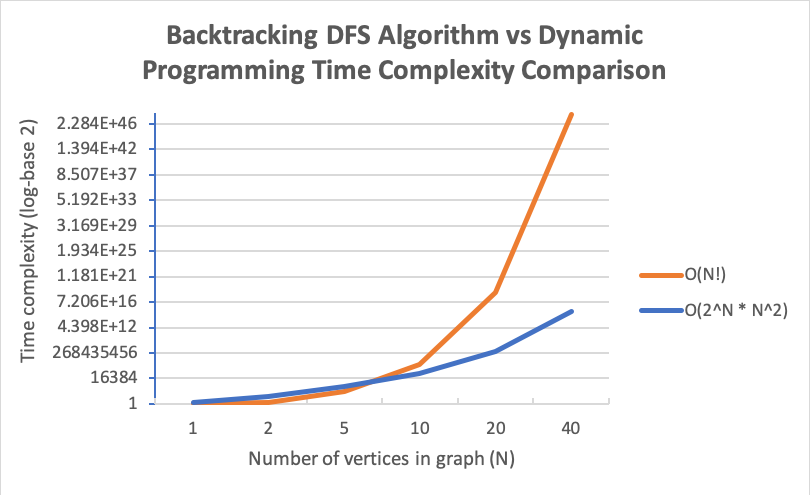
## Finding a Hamiltonian Path

Hamiltonian paths are paths within a graph that visits each vertex exactly once (Wikipedia 2019). The naive solution to find Hamiltonian paths within a graph is to explore all possible configurations or permutations of the set of vertices within the Wikipedia graph. As a result, there will be *n!* possible configurations where n is equal to the number of vertices in the graph. The naive solution hence has a time complexity of O(n!). One such implementation is the Backtracking Algorithm which performs a DFS starting at every vertex contained within the graphs’ vertex set, pushing all nodes visited on to the stack if at least one adjacent vertex is set as ‘unvisited’ (HackerEarth n.d). If the number of elements contained within the stack at any point is equivalent to the number of elements set as ‘visited’, a Hamiltonian path exists. This solution is extremely inefficient due to the repetitive computation of known results.

Dynamic programming addresses this computational problem by keeping records that map previously computed results (Zaveri M 2019) and thus optimises computation time. My Hamiltonian path algorithm uses an adapted implementation of the Bellman-Held-Karp algorithm by implementing their assumption that *“every sub-path of a path of minimum distance is itself of minimum distance”* (Wikipedia 2019*)* using tabulation to memorise previously computed results. This is achieved by using a loop to set particular bits in a bit-set data structure in order using a *bottom-up* approach (Zaveri M 2019). The *bottom-up* approach starts with the smallest sub-problem, computing a solution for these problems and building on these solutions until the larger problem is solved (Joshi V 2017). Hence, the *bottom-up* approach directly implements the underlying assumption of the Bellman-Held-Karp algorithm. The resulting worst-case time complexity of this implementation is enumerating though all possible sets of function calls (Joshi V 2017) comprising of each bit-set mask, and each possible directed edge configuration of time complexity . Finally, in the worst case, if a Hamiltonian path does exist, the ‘Hamiltonian walk’ is constructed by backtracking from the destination node to the starting node consuming order-n time (). The resulting solution is hence an optimised solution of the naive backtracking method.

**Figure 2**

Time complexity comparison in establishing whether a Hamiltonian path exists



## Determining Strongly Connected Components

A subgraph containing vertices that are proper subsets of a graph’s vertices can be considered a strongly connected component if there is a path between all pairs of vertices and this path is maximised (GeeksforGeeks n.d). I have chosen to implement Kosaraju’s algorithm in order to find these strongly connected components by using a depth-first search to traverse the Wikipedia graph once and then another time on the transposed graph (i.e the edges are reversed). The reasoning behind Kosaraju’s algorithm is that the transposed graph will have exactly the same strongly connected components as the original graph (Wikipedia 2019). Hence if vertices *i* and *j* are to form strongly connected component, there must be a directed edge from *i* to *j* in the original graph and a directed edge from *j* to *i* in the transposed graph (Assessment Editorial 2019). If this condition is not satisfied a single vertex serves as a strongly connected component.

In my implementation, I have assumed the transposed graph is prepared before the algorithm starts computation since transposing an adjacency list is a computationally expensive task; time-complexity of . In the first DFS I have used a stack to implement a non-recursive DFS and a first-in first-out queue data structure to enqueue vertices as they are seen. The order at which the first DFS visits vertices is of great importance and needs to be preserved since it will need to be replicated in the second DFS performed. For the transposed graph, a recursive DFS transversal was implemented. This second traversal produces a forest¹ consisting of all the strongly connected components or a single spanning tree, if all vertices are strongly connected within the graph. In the run-time it can be shown that there are no computational efficiencies gained by managing your own stack in the iterative depth-first search when compared to the recursive method. The pseudo code for each implementation is listed in figure 3 and 4 respectively.

|  |  |
| --- | --- |
| **Figure 3**  Recursive Depth-First Search | **Figure 4**  Iterative Depth-First Search |

Kosaraju’s algorithm hence visits each vertex only twice and each edge twice. This results in an overall computation time relative to the number of vertices (V) and the number of edges (E) of , when using an adjacency list. is an asymptotically optimal time complexity in finding strongly connected components (Wikipedia 2019).

## Graph Centre

The centre of a graph is the set of all vertices where the greatest distance to all other vertices is minimal (Wikipedia 2019). Eccentricity is a property of a vertex *u* that is a member of the connected graph *G, that is* equivalent to the maximum path distance between *u* and all other vertices of G (Wolfam Alpha 2019). The centre of a graph can hence be more formally defined as the set of all vertices with minimum eccentricity. Disconnected graphs are considered to have infinite eccentricities, thus within this implementation I have chosen to only consider those vertices with defined edges.

A breadth-first search provides a linear order traversal of a graph allowing for the calculation of the eccentricity of each vertex with a graph. I have chosen to implement a BFS traversal of the Wikipedia graph. A single breadth first search visits each vertex (*V)* and each edge *(E)* once resulting in a time complexity of . Each vertex needs to be re-visited to calculate the set of minimum eccentricities for each starting vertex. Thus, the BFS is performed order-V times, , resulting in an overall algorithm complexity of , simplified further to .

References

Graph definition: <https://en.wikipedia.org/wiki/Graph_(abstract_data_type)>

Shortest Path: <https://en.wikipedia.org/wiki/Shortest_path_problem#Unweighted_graphs>

Hamiltonian Path: <https://en.wikipedia.org/wiki/Hamiltonian_path>

Hamiltonian Path: <https://www.hackerearth.com/practice/algorithms/graphs/hamiltonian-path/tutorial/>

Dynamic Programming: <https://medium.com/free-code-camp/an-intro-to-algorithms-dynamic-programming-dd00873362bb>

Keld-Karp Bottom-up Approach: <https://medium.com/basecs/speeding-up-the-traveling-salesman-using-dynamic-programming-b76d7552e8dd>

Strongly connected components: <https://www.geeksforgeeks.org/strongly-connected-components/>

Kosaraju’s Algorithm: <https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm>

Graph Centre: <https://en.wikipedia.org/wiki/Graph_center>

Eccentricity: <http://mathworld.wolfram.com/GraphEccentricity.html>