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 edges that form the 'link' between these vertices (*Graph (abstract data type)* 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 graph 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 (Datta, A 2019) using a first-in-first-out queue 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 node. Considering a Wikipedia graph, G, *figure* 1 depicts the pseudo code used to find the shortest path between vertex *v* and vertex *y*.

```
Procedure BFS(v, y) //v is the starting node, y is the destination node
    Push v on to the tail of Q
    dist[v] ← 0
    Mark v as 'visited'
    while Q is not empty
        Pop vertex w from the head of Q
        for each vertex x adjacent to w do
            if x is marked as 'unvisited' then
                dist[x] ← dist[w]+1
                if x equals y then
                    return dist[x]
                end if
                Mark x as 'visited'
                Push x on to the tail of Q
            end if
        end for
    end while
```

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 defined edges in the graph. This has been derived by summing the time complexity of enqueuing all vertices and examining all edges within the graph; O(V) and O(E) respectively (Datta, A 2019). A BFS provides optimal time complexity across an unweighted graph (*Shortest path problem* 2019).

Finding a Hamiltonian Path

Hamiltonian paths are paths within a graph that visit each vertex exactly once (*Hamiltonian path* 2019). The naive solution in finding 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-factorial *(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 for-each 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 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"* (*Hamiltonian path* 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 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 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 $O(2^n \cdot n^3)$ enumerating though all possible sets of function calls (Joshi 2017) comprising of each bit-set mask, $O(2^n)$ and each possible directed edge configuration of time complexity $O(n^2)$. 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 ($O(n)$). The resulting solution is hence an optimised solution of the naive backtracking method.

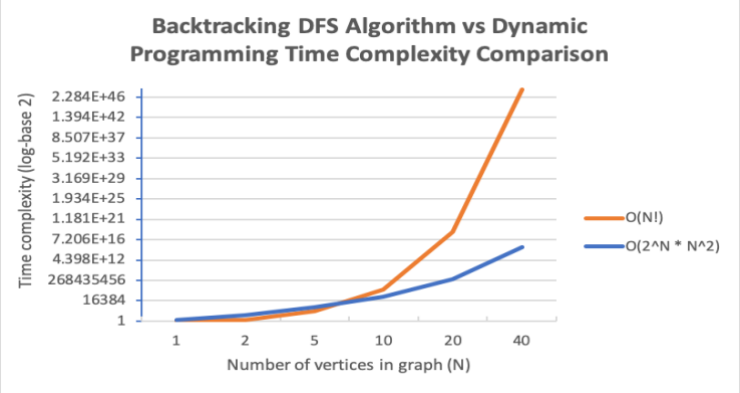

Figure 2
Time complexity comparison in establishing whether a Hamiltonian path exists

Determining Strongly Connected Components

A subgraph containing vertices that are proper subsets of a graph's set of 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 repeated on the transposed graph (i.e the edges are reversed). The reasoning behind Kosaraju's algorithm is that the transposed graph will contain exactly the same strongly connected components as the original graph (*Kosaraju's algorithm* 2019). 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 (Datta et al. 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 $O(n^2)$. In the first DFS I have used a stack to implement a non-recursive DFS, enqueueing vertices as they are seen into a first-in first-out queue data structure. The order at which the first DFS visits vertices is of great importance and needs to be preserved, since this order 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 forest[1] 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 environment it can be shown that there are no significant 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.

```
procedure DFS(w)
    Mark w as 'visited'
    for each vertex x adjacent to w do
        if x is 'unvisited' then
            DFS(x)
        end if
    end for
```

Figure 3
Recursive Depth-First Search

```
procedure DFS(w)
    initialize stack S
    push w onto S
    Mark w as 'visited'
    while S not empty do
        x ← pop off S
        if x is marked as 'unvisited' then
            Mark x as visited
            for each vertex y adjacent to x do
                if y is marked as 'unvisited' then
                    push y onto S
                end if
            end for
        end if
    end while
```

Figure 4
Iterative Depth-First Search

[1] Forest is an undirected graph in which any two vertices are connected by at most one path (Wikipedia 2019)

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 $2 \cdot (V + E)$, when using an adjacency list. $O(V + E)$ is an asymptotically optimal time complexity in finding strongly connected components (*Kosaraju's algorithm* 2019).

Finding The Set of Centre Vertices

The centre of a graph is the set of all vertices for which the greatest distance to all other vertices is minimal (*Graph Center* 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 (Weisstein 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. I have hence 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 $O(V + E)$. 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, $O(V)$, resulting in an overall algorithm complexity of $O(V * (V + E))$, simplified further to $O(V^2 + V \cdot E)$ as the BFS for each node remains the dominant term.

References

Graph (abstract data type) 2019, Wikipedia. Available from: https://en.wikipedia.org/wiki/Graph_(abstract_data_type). [27 May 2019].

Shortest path problem 2019, Wikipedia. Available from: https://en.wikipedia.org/wiki/Shortest_path_problem#Unweighted_graphs. [27 May 2019].

Hamiltonian path 2019, Wikipedia. Available from: https://en.wikipedia.org/wiki/Hamiltonian_path. [28 May 2019].

Jaimini, V n.d, *Hamiltonian Path*, HackerEarth. Available from: https://www.hackerearth.com/practice/algorithms/graphs/hamiltonian-path/tutorial/. [24 May 2019].

Zaveri, M 2019, *An intro to Algorithms (Part II): Dynamic Programming*, Medium. Available from: https://medium.com/free-code-camp/an-intro-to-algorithms-dynamic-programming-dd00873362bb. [28 May 2019].

Joshi, V 2017, *Speeding Up The Traveling Salesman Using Dynamic Programming*, Medium. Available from: https://medium.com/basecs/speeding-up-the-traveling-salesman-using-dynamic-programming-b76d7552e8dd. [28 May 2019].

Strongly Connected Components n.d, GeeksforGeeks. Available from: https://www.geeksforgeeks.org/strongly-connected-components/. [23 May 2019].

Kosaraju's algorithm 2019, Wikipedia. Available from: https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm. [25 May 2019].

Datta, A 2019, Tree and Graph Traversals, lecture notes distributed in CITS2200 Data Structures and Algorithms. Available from: http://teaching.csse.uwa.edu.au/units/CITS2200/material.html.

Datta, A, Gozzard, A, Sullivan, C & Taleb-Beniab, A 2019, CITS2200 Project Editorial, Notes distributed with the project outline. Available from: http://teaching.csse.uwa.edu.au/units/CITS2200/Labs/project-2019/project-2019.html. [16 May 2019].

Graph Center 2019, Wikipedia. Available from: https://en.wikipedia.org/wiki/Graph_center. [26 May 2019].

Weisstein, EW 2019, *Graph Eccentricity*, WolframMathWorld. Available from: http://mathworld.wolfram.com/GraphEccentricity.html. [26 May 2019].