CITS2200 Project Report

Oliver Dean 21307131

Libero Piffaretti 23134294

Abstract

This report presents an analysis of the WikiPageGraph class, which represents a graph of wiki page URLs as a tree of interconnected links. The class provides methods to; calculate the shortest path between vertices, identify strongly connected components, find all graph centers, and find the Hamiltonian path. Several pre-existing algorithms are applied in the class including breadth first search (BFS), Kosaraju's algorithm, and the Held-Karp algorithm. The report details the implementation of these algorithms and their time complexities. The report details testing protocols and how the time complexity approaches the theoretical prediction described by the big O complexity.

Introduction

The WikiPageGraph class offers a representation of a directed graph, where vertices represent URLs of wiki pages, and edges represent links between these pages. The class provides methods for adding edges, calculating the shortest path between two pages, finding all graph centers, strongly connected components, and finding the Hamiltonian path in the graph. It also provides methods for loading a graph from a file.

The main data structure used is a Hash Map (adjacencyList) that represents the graph, where each URL is mapped to a list of URLs representing pages directly linked from it. This adjacency list offers an efficient representation for scattered graphs, which is generally the case in web networks. Specifically, an adjacency list is used as opposed to an adjacency matrix in this context due to efficiency in representing sparse graphs. An adjacency matrix requires space proportional to the square of the number of vertices O(V^2) in the graph, regardless of the number of edges. An adjacency list requires space proportional to the number of edges plus the number of vertices O(V+E). In the case of a large graph an adjacency list is a suitable choice due to its space efficiency and the nature of the operations performed.

Question 1: Shortest Path

public int getShortestPath(String urlFrom, String urlTo)

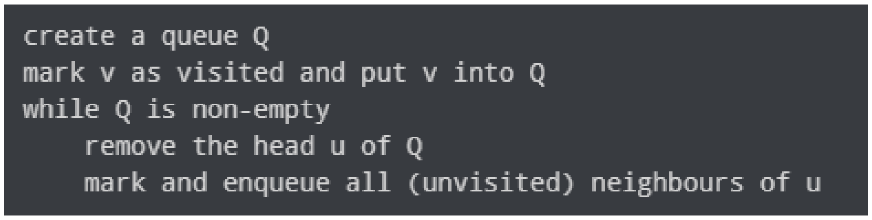
This method calculates the shortest path between two vertices (URLs) in the wiki page graph. It uses breadth first search BFS to traverse the graph from a starting vertex (urlFrom) to the ending vertex (urlTo). The BFS algorithm stores a list of vertices to visit and a map to store the minimum number of links, the distance to each vertex. The method returns the shortests number of links to the ending URL or -1 if no path exists. Performing a BFS is more efficient in an unweighted graph than using Dijkstra's algorithm

The time complexity of the BFS is O(V+E) where V is the number of vertices and E is the number of edges.

A Hash Map is a data structure that uses an associative array abstract data type, a structure that maps keys to values. In the context of our project, the HashMap is used to represent the adjacency list of the graph. The keys are the URLs (or vertices of the graph) and the values associated with the keys are lists of URLs that represent edges from one URL to other URLs. (*Hash Table/Hash Map Data Structure*, n.d.)

HashMaps offer several advantages in our code. For instance, the getShortestPath method uses the HashMap to quickly find all neighboring URLs (vertices) of a URL in order to perform a BFS.

BFS algorithm pseudocode (*BFS Graph Algorithm(With Code in C, C++, Java and Python)*, n.d.)



```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
    remove the head u of Q
    mark and enqueue all (unvisited) neighbours of u
```

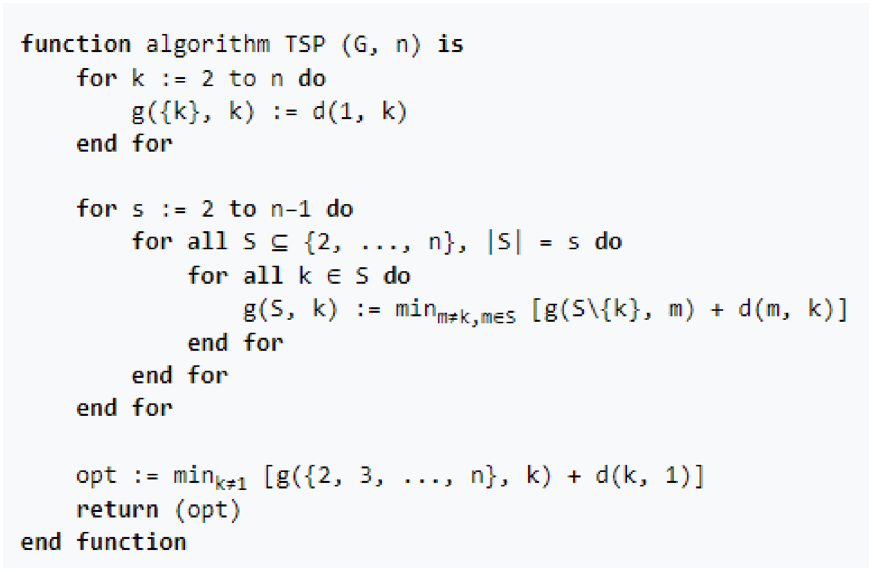
Question 2: Hamiltonian Path

public String[] getHamiltonianPath()

This method finds the Hamiltonian path in the wiki page graph if it exists. A Hamiltonian path is a path in a graph that visits each vertex exactly once. This algorithm uses the “Herald-Karp algorithm” that uses dynamic programming (DP) (Cormen, Leiserson, Rivest, & Stein, 2009).

The method uses DP and bit masking to represent subsets of vertices, it iterates through subsets and updates the shortest path from a given vertex to any other vertex if it is shorter. This method calculates the shortest path between two URLs in the Wiki page graph. It uses breadth-first search (BFS) to traverse the graph from the starting URL (urlFrom) to the ending URL (urlTo). It returns an array containing the hamiltonian path as a sorted array, or an empty array if none was found.An error will be thrown if this method is called on a graph with more than 20 vertices due to its exponential time and space complexity.

Herald-Karp algorithm pseudocode (*Herald–Karp Algorithm*, n.d.)



```
function algorithm TSP (G, n) is
    for k := 2 to n do
        g({k}, k) := d(1, k)
    end for

    for s := 2 to n-1 do
        for all S ⊆ {2, ..., n}, |S| = s do
            for all k ∈ S do
                g(S, k) := $\min_{m \neq k, m \in S}$ [g(S\{k}, m) + d(m, k)]
            end for
        end for
    end for

    opt := $\min_{k \neq 1}$ [g({2, 3, ..., n}, k) + d(k, 1)]
    return (opt)
end function
```

The time complexity of this algorithm is O(N^2 * 2^N) where N is the number of vertices. This is because for every vertex, it goes through all subsets of vertices to check for a path.

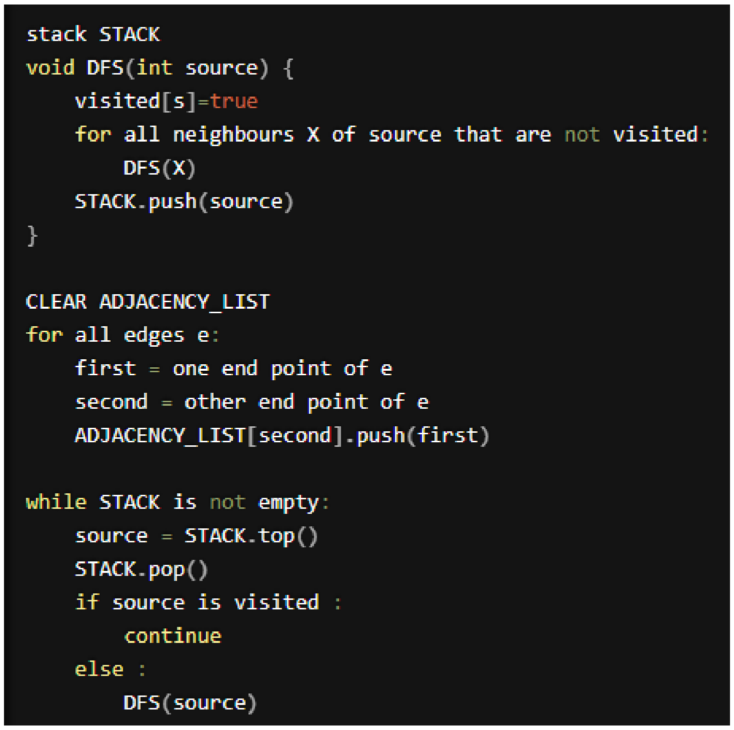
Question 3: Strongly Connected Components

public String[][] getStronglyConnectedComponents()

This method uses Kosaraju's algorithm to find all the strongly connected components (SCC) in the wiki page graph data (Cormen et al., 2009). A SCC of a directed graph is a subset where every vertex is reachable from every other vertex. First a depth first search (DFS) is performed on the graph storing the vertices in post order. Secondly the graph is transposed, reversing all edges. Finally a DFS is done on the reversed graph, exploring the vertices in the order of the post-order. It returns an array of arrays, where each subarray contains URLs forming a SCC.

The algorithm involves two DFS traversals of the main graph, hence the time complexity is O(V + E),

Kosaraju’s algorithm pseudocode (Tatiparti, n.d.)



```
stack STACK
void DFS(int source) {
    visited[s]=true
    for all neighbours X of source that are not visited:
        DFS(X)
    STACK.push(source)
}

CLEAR ADJACENCY_LIST
for all edges e:
    first = one end point of e
    second = other end point of e
    ADJACENCY_LIST[second].push(first)

while STACK is not empty:
    source = STACK.top()
    STACK.pop()
    if source is visited :
        continue
    else :
        DFS(source)
```

Question 4: Graph Centers

public String[] getCenters()

This method finds all the centers of the wiki page graph. The center of a graph is the vertex with the minimum eccentricity, where eccentricity refers to the maximum distance from a vertex to all other vertices in the graph (Cormen et al., 2009). To find the centers this method calculates the eccentricity of each vertex and collects the smallest ones, it then returns an alphanumerically sorted array of URLs corresponding to the centers of the graph.

The complexity of this function is driven by two nested loops iterating over all the vertices in the graph, giving it a base complexity of O(V^2), where V is the number of vertices. Inside the inner loop BFS is performed between the current vertex and all other vertices. This has a complexity of O(E), because it is called in a nested loop the total complexity of the function is O(EV^2)

Time Tests and Results

The testing suite was made in conjunction with chatGPT to help with boilerplating and for the production of additional test data (OpenAI, 2023). For the supplied dataset timed tests on the functions are described below. Specifically testing the graph centers, all the permutations of data for the shortest path method, finding the SCC, and finally the Hamiltonian path.

Test passed: Centers are [/wiki/Braess%27_paradox] | Expected: [/wiki/Braess%27_paradox] | Time: 5 ms (5222500 ns)

Test passed: Shortest path /wiki/Flow_network -> /wiki/Flow_network | Expected: 0 | Result: 0 | Time: 28300 ns

Test passed: Shortest path /wiki/Non_existent_page -> /wiki/Flow_network | Expected: -1 | Result: -1 | Time: 13000 ns

Test passed: Shortest path /wiki/Dinic%27s_algorithm -> /wiki/Ford%E2%80%93Fulkerson_algorithm | Expected: 1 | Result: 1 | Time: 95400 ns

Test passed: Shortest path /wiki/Edmonds%E2%80%93Karp_algorithm -> /wiki/Minimum_cut | Expected: 3 | Result: 3 | Time: 12000 ns

Test passed: Strongly connected component found: [[/wiki/Approximate_max-flow_min-cut_theorem, /wiki/Circulation_problem, /wiki/Dinic%27s_algorithm, /wiki/Edmonds%E2%80%93Karp_algorithm, /wiki/Flow_network, /wiki/Ford%E2%80%93Fulkerson_algorithm, /wiki/Gomory%E2%80%93Hu_tree, /wiki/Max-flow_min-cut_theorem, /wiki/Maximum_flow_problem, /wiki/Minimum-cost_flow_problem, /wiki/Minimum_cut, /wiki/Multi-commodity_flow_problem, /wiki/Network_simplex_algorithm, /wiki/Out-of-kilter_algorithm, /wiki/Push%E2%80%93relabel_maximum_flow_algorithm], [/wiki/Braess%27_paradox], [/wiki/Nowhere-zero_flow]] in 0.7ms (715900 ns)

Test passed: Hamiltonian path found: [] in 3185 ms (3185172200 ns)

Discussion / conclusion

The WikiPageGraph class provides a set of algorithms for graph analysis, with methods for identifying: shortest paths; Hamiltonian paths; strongly connected components; and graph centers. As observed from the test results, the timing of each method aligns with their predicted time complexities.

The getShortestPath method, implementing a breadth-first search (BFS), displays a low execution time which is consistent with its time complexity of O(V+E). This shows BFS's efficiency when dealing with graphs, where the algorithm ensures the shortest path is found in the fewest number of iterations.

The getStronglyConnectedComponents method, which uses Kosaraju's algorithm (Cormen et al., 2009), involves two depth-first search (DFS) traversals, with a time complexity of O(V+E). The execution time observed in the tests verifies this, reflecting the speed of DFS in exploring all vertices of the graph.

The getCenters method, however, requires BFS to be run on every vertex, leading to a higher time complexity of O(EV^2). This was confirmed in our test run where the time taken was noticeably longer. Despite this, the method is still capable of accurately identifying the centers of the graph, which is crucial in network analysis for identifying key vertices.

The getHamiltonianPath method, which employs the Held-Karp algorithm, displays the largest time complexity at O(N^2 * 2^N). This complexity is caused as the method examines all subsets of vertices for every vertex. As expected, the execution time for this method took the longest among the tests. However, its ability to find a Hamiltonian path provides valuable insights into the connectivity and navigability of the network, despite the computational complexity cost.

The timing and complexity of these methods highlight a key trade-off in graph analysis and algorithm design. Higher complexity algorithms often provide more comprehensive or delicate analysis (e.g., Hamiltonian paths), but at the cost of increased computational resources. Simpler algorithms (e.g., BFS for shortest path) may offer less complete insights but are more suitable for large graphs in order to minimize complexity.

In conclusion, the time complexity of the implemented algorithms and the performance tests corroborate that this class efficiently represents a wiki page graph and performs various operations. However, some performance improvements could be investigated for methods dealing with more complex problems such as the Hamiltonian Path and the graph centers. The real-world applicability of such a class is immense, including in network analysis, search engines, data mining, and information retrieval. Future work could include extending this model to deal with weighted edges, enabling it to model more diverse and complex networks.

References

BFS Graph Algorithm(With code in C, C++, Java and Python). (n.d.). Programiz. Retrieved May 20, 2023, from https://www.programiz.com/dsa/graph-bfs

Herald–Karp algorithm. (n.d.). Wikipedia. Retrieved May 20, 2023, from https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm#Pseudocode[4]

Tatiparti, A. (n.d.). *Kosaraju's Algorithm for Strongly Connected Components*【*O(V+E)*】. OpenGenus IQ. Retrieved May 20, 2023, from https://iq.opengenus.org/kosarajus-algorithm-for-strongly-connected-components/

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.

OpenAI. (2023). *ChatGPT* (May 12 version) [Large language model]. https://chat.openai.com/chat

Hash Table/Hash Map Data Structure. (n.d.). Interview Cake. Retrieved May 24, 2023, from https://www.interviewcake.com/concept/java/hash-map