CITS2200

Algorithms and Data Structures

Project

Levente Zombori and Jong Su Jang

22228596 - 22522128

Semester One 2019

Preface

This project covers various algorithms that is used to find different elements in a graph, an essential data structure in computer science. The project asks us to investigate four different problems, which are:

1. Given a pair of vertices, find the shortest path to get to the second from the first
2. Find out the Hamiltonian path in the graph
3. Find out every strongly connected components in the graph
4. Find out all the vertices that are the centre of the graph

The topic of this project is to analyse parts of the Wikipedia graph. The Wikipedia can be viewed as a graph by considering each page as a vertex and the URL to each vertex as edges. This way we can find out various features of the Wikipedia graph, such as whether a Hamiltonian path exists, or the minimum amount of length it takes to get from one page to another.

Within the Wikipedia graph, the URLs have a direction; it may be possible to get from one page to another; however, it may not be possible to do the reverse. This means that the edges in the graph are directed, and if vertex a is adjacent to vertex b, vertex b may not be adjacent to vertex a. The edges are assumed to be unweighted, as it just takes one click to go from one page to another. Due to this, the edge weights for this Wikipedia graph is assumed to be 1.

There are two common ways to represent graphs; the use of adjacency lists and adjacency matrix. For adjacency list, it takes constant time to enumerate the edges when leaving the vertex, whereas for adjacency matrix, it takes O(V). Therefore, for this project we have decided to use an adjacency list instead of an adjacency matrix.

1 Shortest Path Algorithm

The shortest path is the path that takes minimum edges to get to one vertex to another. In the method getShortestPath(), it returns the minimum number of edges that is required to get from the start and the finishing vertex, which are the inputs. If it is not possible to get from one vertex to another, it will simply return -1. If the start vertex and the finish vertex is the same, it will return 0.

1.1 Pseudocode:

Algorithm 1: `Breadth First Search`

Input: A graph $Graph$ and a starting vertex $root$ of Graph
Result: The $parent$ links trace the shortest path back to $root$

```
procedure BFS(G, startv):
begin
    let S be a queue;
    S.enqueue(startv);
    while S is not empty do
        v = S.dequeue();
        if v is the goal: then
            return v;
        end
        for all edges from v to w in G.adjacentEdges(v) do do
            if w is not labeled as discovered: then
                label w as discovered;
                w.parent = v;
                S.enqueue(w);
            end
        end
    end
end
```

See appendix 5.1 for reference

1.2 Implementation & Time Complexity:

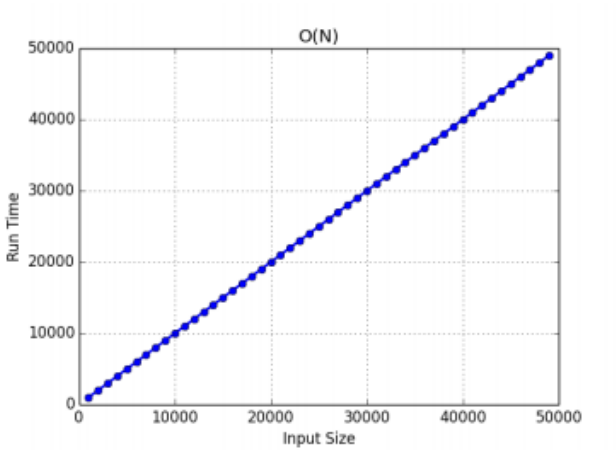
The algorithm we used for this question was a Breadth First Search (BFS). Breadth First Search enumerates vertices starting from the origin, until the vertex that is the destination is reached. The BFS method will visit our starting vertex and will maintain a queue that will store the adjacent vertices going from the starting vertex to the end vertex. This method ensures that the minimum distance between two vertices of the graph is recorded.

When analysing our Wikipedia graph of G(V,E) where V is the vertices of the graph (pages) and E is the edges of the graph (URLs). BFS at most visits each vertex and edge once, the worst-case time complexity of the BFS search is O(V+E), which is optimal.

1.3 Performance:

Worst-Case

Breadth-First Search: $\mathcal{O}(V + E)$



The time complexity of a BFS algorithm depends directly on how much time it takes to visit a node. Since the time it takes to read a node's value and enqueue its neighbour doesn't change based on the node, we can say that visiting a node takes constant time, or, O(1) time. Since we only visit each node in a BFS graph traversal exactly once, the time it will take us to read every node just depends on how many nodes and edges there are in the graph. If our graph has 15 nodes and 10 edges, it will take us O(25); but if our graph has 1500 nodes and 1000 edges, it will take us O(2500). Thus, the time complexity of a breadth-first search algorithm takes linear time, or O(V + E), where V is the number of nodes and E is the number of edges in the graph.

Space complexity is similar to this, has more to do with how much our queue grows and shrinks as we add the nodes that we need to check to it. In the worst-case situation, we could potentially be enqueuing all the nodes in a graph, which means that we could possibly be using as much memory as there are nodes in the graph. If the size of the queue can grow to be the number of nodes in the graph, the space complexity for a BFS algorithm is also linear time, or O(V), where V is the number of nodes in the graph.

2 Hamiltonian Path

A Hamiltonian path is an NP-complete problem that finds a path that visits every vertex in a graph exactly once. This means that there is no polynomial time algorithm that has been found to solve this problem. The getHamiltonianPath() method returns the array that contains the array of vertices in order of the Hamiltonian path, if none is found it will return an empty array.

2.1 Pseudocode

```
function check_using_dp(adj[][], n)
    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 = 0 to n
        if dp[i][2^n-1] == true
            return true
    return false
```

See appendix 5.2 for reference

2.2 Implementation & Time complexity

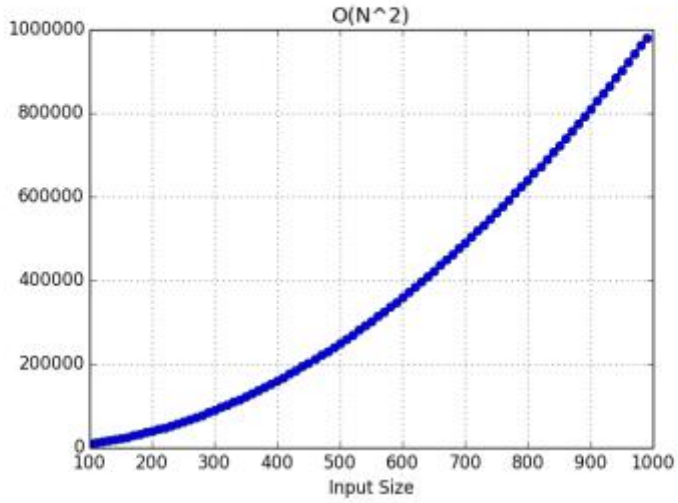
In this implementation, we used Bellman-Held-Karp algorithm. As mentioned above, there is no polynomial time algorithm found for this problem. As a result, we used the Bellman-Held-Karp algorithm, which is dynamic programming algorithm which divides the graph into subsets of the original graph that can be used to find the Hamiltonian path. When analysing our Wikipedia graph of G(V,E) where V is the vertices of the graph (pages) and E is the edges of the graph (URLs). The algorithm has worst-case time complexity of $O(V^2 2^V)$.

Additionally, we sped up our code by using arrays of primitives (as it's likely to have to better memory layout than a list of objects) and operating on bitmasks directly. This allowed for constant time as opposed to the built in Math.pow() method.

2.3 Performance

Complexity Analyis

Bellman-Held-Karp: $\mathcal{O}(V^2 + 2^V)$



The worst-case time complexity for this algorithm is $O(V^2\ 2^V)$.

The Bellman-Held-Karp algorithm is answering the question "Is there a path ending at vertex *A* that visits every vertex in the subset *P* exactly once?". This question can be broken down into the base question of "Is there a path ending at the vertex *A* that visits *A* exactly once?". There are $O(2^V)$ subset vertices in the graph, which results in $O(2^V\ V)$ questions of the above form. We can represent the subsets into bitset, where using each binary digit in an integer, we can represent if the vertices are in the set or not. We can store the answer to this as it is computed, so we won't have to re-evaluate it again. The answer to each of these questions is in O(V) and therefore the worst-case time complexity for the Bellman-Held-Karp algorithm is $O(2^V\ V^2)$.

The space complexity of the Bellman-Held-Karp algorithm is $O(2^v\ V)$. The Bellman-Held-Karp algorithm is answering the question m "Is there a path ending at the vertex *A* that visits *A* exactly once?" and there are $O(2^V)$ subset vertices in the graph, which results in $O(2^V\ V)$ questions of the above form.

3 Strongly Connected Components

The strongly connected components (SCC) is a group of vertices of a graph where every vertex can all reach the other. The method getStronglyConnectedComponents() partitions the vertices in the graph into the SCCs that contain them.

3.1 Pseudocode

Algorithm 8.7 *Kosaraju_SCC*

1: **Input** : $G = (V, E)$
2: **Output** : SCCs of G
3: **stack** $S \leftarrow \emptyset$
4: **while** $S \neq V$ **do**
5: **pick** an arbitrary vertex $v \notin S$
6: **run** $DFS(G, v)$ by putting finished vertices on stack S
7: **end while**
8: **reverse** direction of edges to obtain G^T
9: **while** $S \neq \emptyset$ **do**
10: $u \leftarrow pop(S)$
11: **run** $DFS(G, u)$ ▷ form a DFS tree for each vertex on S
12: **end while**
13: vertices in each tree rooted at stack vertices are the SCCs of G

See appendix 5.3 for reference

3.2 Implementation & Time complexity

In this implementation, we used Kosaraju's algorithm as it is conceptually the simplest efficient algorithm. It is not as efficient as Tarjan's SCC algorithm and the path-based SC algorithm which traverses the graph only once. Kosaraju's algorithm uses a transposed graph where if vertices are in the same SCC, the vertices need to be reachable for both sides. In other words, vertex a can be reached by vertex b and vertex b can be reached by vertex a.

We can compute the SCC of a graph by doing a DFS on the original graph and a transposed graph. We first call DFS on the original graph which will separate the graph into subsets of the graph that can be reachable for a given vertex. We then create a transposed graph which has all the edges reversed, and we perform another DFS on the transposed graph. For the given two vertices, if they have the same connected components in both the in-order and post-order, then they are strongly connected.

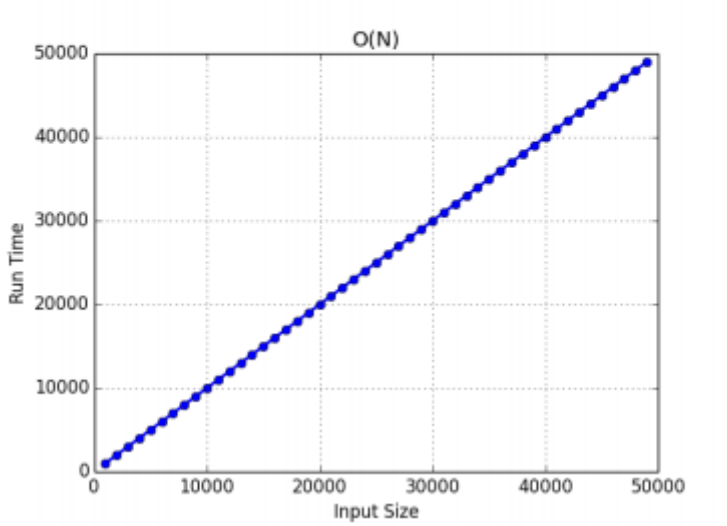
Kosaraju's algorithm uses 2 Depth First Search (DFS) which visits each edge and each vertex exactly once, giving the algorithm the time complexity of O(V+E)

3.3 Performance

Complexity Analyis

Kosaraju's: $\mathcal{O}(V + E)$

Graph Transposal: $\mathcal{O}(V + E)$



The time complexity of the Kosaraju's algorithm depends on the time it takes to visit the node. Since the time it takes to push the vertex into the stack doesn't change with what node it is, we can say that visiting the node takes constant time. We must visit each node twice using the DFS (for in-order and post-order traversal) which means the time it takes us to read every node depends on how many nodes and edges there are in the graph. If the graph has 20 nodes and 15 edges, it will take us O(35); but if the graph has 2000 nodes and 1500 edges, it will take O(3500). Hence the time complexity of Kosaraju's algorithm takes linear time, or O(V+E) where V is the number of nodes in the graph and E is the number of edges in the graph

As for space complexity, it is about how much our stack grows or shrinks as we add more vertices that needs to be checked. In the worst-case situation, we could be pushing all the vertices into the stack, which means we need as much memory as there are vertices in the graph. Therefore the space complexity of Kosaraju's algorithm is O(V) where V is the number of vertices in the graph.

4 Graph Centres

Graph centre can be defined with the same definition of the centre of the circle; the minimum distance to the furthest point. In this Wikipedia graph, this could be represented as a vertex from which the furthest vertex is as close as possible; the vertex with the minimum eccentricity.

4.1 Pseudocode

```
function getCenters() {
  let center be an array
  radius = graph.size();
  for i = 0 to graph.size()
    eccentricity is -1
    for a in bfs(i)
      if a is not -1 and eccentricity is less than a
        eccentricity = a
    if eccentricity is not -1 and eccentricity < radius
      radius = eccentricity
      clear the previous center
    if eccentricity = radius
      add i to center
}
```

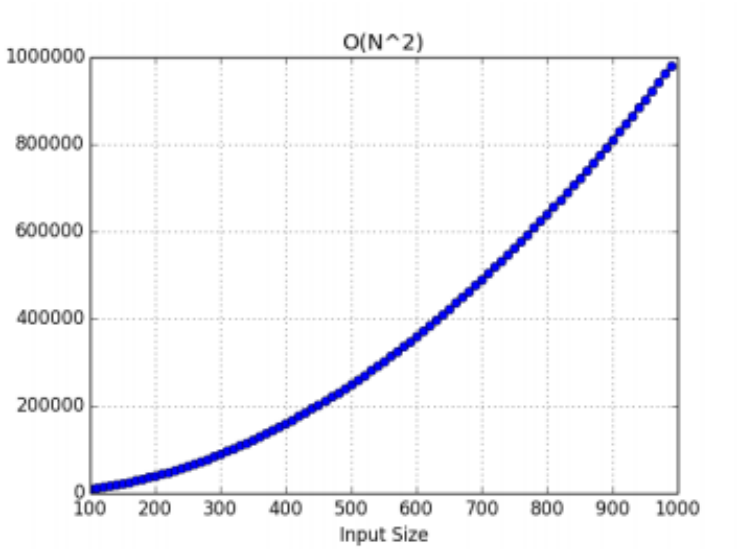
4.2 Implementation & Time complexity

Using the BFS implementation in Q1, the vertices with the minimum eccentricity can be evaluated by doing a BFS for each vertex and storing the maximum value from the distances in an array. For this method, the Breadth First Search is performed for each vertex which gives us the worst-time complexity of $O(V * (V+E))$, simplified to $O(V^2 + VE)$ which is optimal.

4.3 Performance

Complexity Analyis

Graph Centers: $\mathcal{O}(V(V+E))$



The time complexity of the graph centres algorithm depends on the number of vertices and the BFS traversal of the graph. To get the graph centre, we need to do a BFS for each vertex in the graph. Since the time complexity of the Breadth First Search is O(V+E), the time complexity of getting the graph centres is the number of vertex multiplied by BFS, as we need to perform BFS for each vertex, giving the overall time complexity of $O(V^2 + VE)$ where V is the number of vertices and E is the number of edges in the graph.

Similar to other algorithms, the space complexity depends on how many vertices are in this graph This is because we need to do a BFS for each vertex in the graph, which means we need to store the graph inside a data structure (i.e an array). Therefore, the space complexity of this algorithm is O(V) where V is the number of vertices in the graph.

5 Appendices

5.1 Breadth-First Search Algorithm

Pseudocode

Wikipedia.com. Breadth-First Search.

Available at: www.tinyurl.com/cdpn8t2

Understanding

Hackerearth.com. Breadth First Search Tutorial.

Available at: www.tinyurl.com/y3heq724

5.2 Bellman–Held–Karp Algorithm

Pseudocode

Hackerearth.com. Held–Karp algorithm.

Available at: https://tinyurl.com/ychxux44

Understanding

Medium.com. Dynamic Programming.

Available at: www.tinyurl.com/y49w4tha

Reference Implementation

Stackoverflow.com. Speeding up the Algorithm.

Available at: www.tinyurl.com/y3pgucqp

5.3 Kosaraju's Algorithm

Pseudocode

Guide to Graph Algorithms: Sequential, Parallel and Distributed. (2019).

Available at: www.tinyurl.com/y55gohrb

Understanding

Geeksforgeeks.org. Strongly Connected Components.

Available at: www.tinyurl.com/y3vnxhhe

Programcreek.com. Reference implementation of Kosaraju's Algorithm.

Available at: www.tinyurl.com/yxme8oeny

5.4 Graph Centers

Pseudocode

Codeforces.com. Center of a Graph.

Available at: www.tinyurl.com/y428m7bj

Understanding

Geeksforgeeks.org. Graph Theory.

Available at: www.tinyurl.com/y4kt2dcp

5.5 Graph Representations

Graphs

Lukasmestan.com. Big-O notation.

Available at: www.tinyurl.com/y5nw6o96