CITS2200 Data Structures and Algorithms

Semester 1 Project 2023

23362432

23484481

Introduction

This project introduces practical applications of various algorithms learnt in CITS2200, onto a Wikipedia graph, given the following questions:

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.
2. Write a method that finds a Hamiltonian path in a Wikipedia page graph.
3. Write a method that finds every ‘strongly connected component’ (hereon referred to as SCC) of pages.
4. Write a method that finds all the centers of the Wikipedia page graph.

This report aims to analyze the problems presented to us, the approaches taken to solve them, and the analysis of the solutions and their efficiencies.

I. The Wikipedia Graph

In this project, we require the usage of the graph data structure, and thus simulate the Wikipedia pages provided as a graph.

The problem assumes Wikipedia pages as vertices, and the links connecting pages as the edges in a graph data structure, allowing us to find distance and paths.
In Wikipedia, some pages may have links to other pages, but those pages may not necessarily bring us back to the previous page. These links also do not hold any ‘weight’, as only a click on the link will bring us to another page. This allows us to assume that the Wikipedia graph is directed and unweighted, which are necessary in implementing and analyzing the algorithms we want to use.

In order to construct the Wikipedia graph, we take a file that consists of the adjacencies of the different pages of the sample graph, and then use an addEdge() method. It is important to note

in what form we build our graph as, as there are two formats: adjacency matrix and adjacency list.

The adjacency list was deemed the most optimal format for our purposes, as it has better time and space complexity compared to an adjacency matrix.The adjacency list takes O(E) space while the matrix takesO(V^2). The benefits of an adjacency matrix is to check the existence of an edge between vertices in constant time, however, this is not required in this project. The requirements are to enumerate the edges from one vertex to another, which the adjacency list can do in O(V) time, while an adjacency matrix will do that in O(V^2).

II. Finding the Shortest Path

This problem requires us to find the shortest path, or the minimum distance given two vertices. This means we are required to find the minimum number of edges it takes from one vertex to another.

In the method getShortestPath(), it takes the starting url and the endpoint url, and returns the minimum number of links it takes to reach the endpoint url from the starting url,, -1 if the input urls are non-existent in the graph or if there are no paths found between the two urls.

A. Approach

The algorithm used for this problem is a Breadth First Search, which traverses the vertices by level and enumerates the edges based on its distance from the starting vertex. This means, it starts from the input starting url and then checks for all the pages of distance 1 from the starting url, and then continues on to those 2 edges away, and so on and so forth. The information is then stored as a key-value pair, with the vertex being the key and the distance being the updating value as the BFS traverses each level.

B. Pseudocode

```
procedure getShortestPath(startV, endV) ▷ startV is the start vertex
and endV is the end vertex
      ▷ initializing structures
      Let Q be queue for BFS
      Let D be array of distances
      Let V be array of visited

      enqueue startV to Q
      D[startV] ← 0

      While Q is not empty
            current ← q.poll()
            If current is equal to endV
```

```
            return D[w]

      for all vertices u adjacent to current
            if not in V
                  enqueue u to Q
                  D[v] ← D[w] + 1
```

C. Algorithm Analysis

1. Time Complexity

The BFS algorithm analyzes each vertex and edge once, in the worst case, and therefore has a worst-case time complexity of O(V+E); for every vertex visited, all the adjacent vertices (or edge expansions) are also checked. The actual queue operations work in constant time.

2. Space Complexity

The BFS algorithm uses space based on the number of vertices it has, and may require additional space for the queue holding visited vertices. However, ultimately, the worst-case space complexity will be O(V).

3. Performance on Different Sized Graphs

The execution time of the BFS algorithm is dependent on the number of vertices in the graph, as the BFS visits each vertex once, and then visits all vertices adjacent to that vertex. Each vertex visit takes O(V) time, and every check for an adjacent vertex takes O(E). Thus, the worst case time complexity is O(V+E), which is linear in behavior.

The space complexity is dependent on the number of vertices in the graph as the algorithm needs to be conscious of the number of vertices that can be stored in the queue of checking, which is all the vertices. Therefore the worst case space complexity will be O(V). This means that the more vertices a graph has, the more space will be required for the algorithm. While it is positive that the space complexity is linear, we also have to take into account an extra data structure to store the visited vertices. So the bigger the graph, the more space will be used linearly.

III. Finding the Hamiltonian Path

This question requires us to find the hamiltonian path of the graph; a hamiltonian path refers to a path that visits all the vertices in a graph, once.

The method getHamiltonianPath() takes no inputs, and returns an array of the order of vertices of the hamiltonian path in the graph, -1 if there is no hamiltonian path.

A. Approach

The Bellman Held-Karp algorithm was used to solve this problem. The algorithm utilizes dynamic programming, which breaks down the problem into smaller problems to be solved, and then builds up the solutions of these smaller problems into a final solution. The main problem presented is whether there is a path ending at vertex u that visits every vertex exactly once in subset S. This question is broken down into a simpler and similar form of the question, leading to the following realization: such a path exists if and only if there exists some vertex in S for which the question is true and has an edge to u, if we can remove u from S. This reduces the question to whether there is a path that ends in u that visits u exactly once, which is true.

An analogy is a traveling salesman who wants to visit a bunch of countries but only wants to visit each one exactly once. First, after listing all the countries to be visited, consider all combinations of countries he can visit. Then find the cost to travel for each combination ending at a specific country. If there is a cheaper route ending at that same country for the same combination, take note of it. Once all costs for all combinations have been found, find the combination with the minimum cost, and the last country of that combination is the optimal end country. Repeat this with all the combinations, minus the previously found combination and keep taking the last country of that combination to build up the path.

B. Pseudocode

Given G is the graph:

```
procedure getHamiltonianPath()
      Let vertices be an array of vertices in the graph
      Let dp be a 2D array storing subsets of vertices and their
positions

      for subset S ← 2 to (1 <<  size[G])
            for vertex v ← 1 to size[G]
                  if v is in S
                        for vertex u ← 1 to size[G]
                              if u is in S AND u is adjacent to v
                                    dp[v][S] ← min(dp[v][S], dp[u][S XOR
(1<<v)] + 1)

      for i ←1 to size[G]
            if minDist > dp[i][endSubset]
                  minDist ← dp[i][endSubset]
                  end ← i

      Let path be output
      for i ← size[G] to 1 step -1
```

```
            path[i] ← vertices[end]
            prevMask ← currMask XOR (1 << end)
            for j ← 1 to size[G]
                 if j in prevMask AND dp[end][currMask] is equal to
dp[j][prevMask] + 1 AND end in j
                      parent[end] ← j
                      break
            currMask ← prevMask
            end ← parent[end]
      return path
```

C. Algorithm Analysis

a. Time complexity

As the algorithm makes use of subsets, there are O(2^V) subsets in a graph G(V,E), as these subsets are based on the question of “Does the vertex exist in this vertex or not?” In each subset, we ask that question to every vertex. Each vertex is represented as a bitset, which works in constant time, thus giving O(V) for every answer. Then we have to compute these answers in reverse to build up the hamiltonian path, giving another O(V) as all the vertices would be part of the path. Thus ultimately, the worst case time complexity will be O(V^2 * 2^V).

b. Space Complexity

The space complexity of this algorithm is made up of the subsets and the vertices in the graph as we need to store them for analysis, thus the worst case space complexity is O(V*2^V).

c. Performance on Different Sized Graphs

Due to the exponential nature of the time and space complexities, the algorithm can only work at decent speeds up to 20 vertices, which is taken into account when the problem is presented to us. However, beyond it, the exponential nature of its time complexity makes the execution time so long, it becomes too impractical to use this algorithm as a solution.

IV. Finding the SCC in a Graph

This question requires us to find the SCCs of the graph; SCCs are defined as a set of vertices where there is a path to every vertex from any starting vertex.

The method getStronglyConnectedComponents() takes no inputs, and returns an array of arrays of strings, where each internal array is the list of vertices that make up an SCC.

A. Approach

While there are many known algorithms to solve this, Kosaraju's algorithm was used in this instance because it has the simplest implementation, despite not being the most efficient. The algorithm is a Depth First Search based algorithm, making use of it twice. The first DFS is performed on the original graph to get the vertices in post-order, helping us keep track of the order vertices finish in DFS. Then a second DFS is performed on the transposed graph to find the SCCs, by checking if they have the same connected components in order and post-order. If it is, it is considered an SCC and stored in a list to be returned at the end.

B. Pseudocode

```
procedure getStronglyConnectedComponents()
      Let postOrderStack be a stack to store the transposed graph
      Let visited be a Hashmap to store visited urls

      ▷ Do first DFS
      for every url u in graph G
            if u is not in visited
                  do dfs1(u, visited, postOrderStack);

      ▷ Do second DFS
      clear visited
      Let sccList be a list to store the list of SCCs in the graph
      while postOrderStack is not empty
            url ← postOrderStack.pop()
            if visited does not contain url
                  Let scc be the array that keeps the urls part of the
SCC
                  do dfs2(url, visited, scc, reversedGraph)
▷reversedGraph is defined at the beginning of the class
                  add scc to sccList

      Let sccArray be the output array of SCCs
      for i ← 1 to size[sccList]
            sccArray[i] ← sccList.get(i) as String

      return sccArray

▷ dfs for the first dfs
procedure dfs1(url, visited, postOrderstack) ▷ inputs are current
vertex, visited stack and transposed graph
      add url to visited
      for every adjUrl in G
            if adjUrl not in visited
```

```
                do dfs1(adjUrl, visited, postOrderStack)
      push url to postOrderStack

▷ dfs for the second dfs
procedure dfs2(url, visited, list, graph) ▷ inputs are current
vertex, visited stack, list to store scc and graph being traversed
      add url to visited
      add url to list

      for every adjUrl in graph
            if adjUrl not in visited
                  do dfs2(adjUrl, visited, list, graph)
```

C. Algorithm Analysis

a. Time Complexity

The worst case time complexity of this algorithm is O(V + E). This is because for all the operations in the algorithm, they're all dependent on the number of vertices and edges present in the graph. In the reversing of the graph, it has to visit all the vertices and edges to reconstruct it and both DFS functions also have to visit all the vertices and edges only once each. Therefore, it is O(V+E). Another algorithm, called Tarjan's algorithm has a similar time complexity and is generally preferred due to simplicity and efficiency but as beginners, Kosaraju's algorithm is preferable with its direct and constructed implementation.

b. Space Complexity

The space required for this algorithm is to store all the vertices visited as the stack fluctuates during the DFS.In the worst case, all the vertices of the graph may be part of the SCC so the worst case space complexity is O(V), where V is the number of vertices in the graph.

c. Performance on Different Sized Graphs

Due to the linear behavior of the time and space complexities of the Kosaraju algorithm, as the size of the graph increases, the time and space complexities will increase in a linear manner as well. The performance of these is similar to how it would be for the first question.

V. Finding the Centers of a Graph

This question requires us to find the centers of the graph. A vertex is considered a center should its maximum shortest path to any other vertex be the minimum value of all maximum shortest paths, i.e. has the most minimum eccentricity.

The method getCenters() takes no parameters, and returns an array of strings of the centers found in the graph.

A. Approach

A BFS was also used for this problem to solve it. First an eccentricity map is initialized to store the eccentricity of each vertex to analyze later. A BFS is run on the graph to find the shortest path for each vertex to every other vertex. Then the maximum shortest path is searched for all the shortest paths for each vertex. Then we look for the vertex with the smallest eccentricity or with the smallest, maximum shortest path. Those vertices with the smallest eccentricities are then returned as the output.

B. Pseudocode

```
procedure getCenters()
     Let eccentricityMap be a hashmap that stores vertices and their
eccentricities

     for every url in graph G
          Let distance be the distances of every url
          for every vertex v in G
               distance[v] ← infinite
          distance[url] ← 0

          add url to queue

          while queue is not empty
               current ← q.dequeue()
               for every neighbor in graph[current]
                    enqueue neighbor to queue
                    distance[neighbor] ← distance[current] + 1

          maxDistance ← 0
          connected ← true

          for every vertex v in G
               if distance[v] is equal to infinite
                    connected ←  false
                    break
               if distance[v] is greater than maxDistance
                    maxDistance ←  distance[v]
          if is connected
               eccentricityMap[url] ← maxDistance

     minEccentricity ← infinite
     for each eccentricity in eccentricityMap
```

```
            if eccentricity is less than minEccentricity
                  minEccentricity ← eccentricity

      Let centers be the output array to hold the graph centers
      for each entry in eccentricityMap
            if entry.value is equal to minEccentricity
                  add entry.key to centers
      return centers
```

C. Algorithm Analysis

a. Time Complexity

The algorithm performs a BFS on every vertex, which has a time complexity of O(V+E). After, for each vertex we need to find the maximum shortest path to every other vertex, which itself has a time complexity of O(V). Thus the time complexity for the entire thing is O(V(V+E)).

b. Space Complexity

The space we need for this algorithm is just to store the vertices into queues in the breadth first search so therefore the worst-case is having to store all the vertices in the queue. Thus the worst case space complexity is O(V).

c. Performance on Different Sized Graphs

The performance on different graphs is dependent on the number of vertices and edges in the graph. The performance of this algorithm on larger graphs will be exponentially increased due to the time complexity of O(V^2 + V E). The increased input is also gonna affect the space complexity in a linear manner. Thus as the graph increases, the performance will drop exponentially for time, but linear for space.

VI. Sources

The following are the sources in doing this project:

- CITS2200ProjectEditorial.pdf : provided on the project page

Algorithm Definitions and Understanding

- ChatGPT
- https://en.wikipedia.org/wiki/Shortest_path_problem
- https://en.wikipedia.org/wiki/Hamiltonian_path
- https://en.wikipedia.org/wiki/Strongly_connected_component
- https://en.wikipedia.org/wiki/Graph_center

Code

- ChatGPT

- https://www.programiz.com/dsa/graph
- https://www.w3schools.com/java/java_hashmap.asp
- https://www.programiz.com/dsa/graph-adjacency-matrix
- https://www.geeksforgeeks.org/add-and-remove-edge-in-adjacency-list-representation-of-a-graph/
- https://www.programiz.com/dsa/graph-bfs
- https://www.programiz.com/dsa/graph-dfs
- https://www.geeksforgeeks.org/shortest-path-unweighted-graph/
- https://github.com/leviznull/CITS2200
- https://stackoverflow.com/questions/8379785/how-does-a-breadth-first-search-work-when-looking-for-shortest-path
- https://www.softwaretestinghelp.com/java-graph-tutorial/
- https://www.freecodecamp.org/news/exploring-the-applications-and-limits-of-breadth-first-search-to-the-shortest-paths-in-a-weighted-1e7b28b3307/#:~:text=We%20say%20that%20BFS%20is,give%20us%20the%20shortest%20path
- https://stackoverflow.com/questions/56338255/hamiltonian-path-algorithm-on-directed-unweighted-graph-uses-non-existent-edges
- https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/
- https://www.programiz.com/dsa/dynamic-programming
- https://www.geeksforgeeks.org/strongly-connected-components/
- https://www.programiz.com/dsa/strongly-connected-components
- https://www.topcoder.com/thrive/articles/kosarajus-algorithm-for-strongly-connected-components
  https://github.com/jbpt/codebase/blob/master/jbpt-core/src/main/java/org/jbpt/algo/graph/StronglyConnectedComponents.java