CITS2200 Data Structures and Algorithms Project Report

Evan Huang 20916873

Introduction

By viewing links between pages as directed edges, and individual pages as vertices, the page structure of Wikipedia can be viewed as a form of graph structure. The project's goal was to analyse portions of graphs based upon page link data from Wikipedia.

4 problems were tasked to be solved involving arbitrarily selected subsets of this data.

1. Finding the shortest path that exists (in number of links) between a pair of pages.
2. Finding a Hamiltonian path through a set of pages, if it exists.
3. Find every set of strongly connected component that exists in a set of pages.
4. Find the center points (Jordan centers) of a set of pages.

Solving these problems involved selection of appropriate algorithms, given the nature of the Wikipedia data, as well as an appropriate form of graph data structure.

Wikipedia and Graph Implementation

The data structure chosen to represent the set of Wikipedia pages and links between them is a ArrayList of adjacency lists in a LinkedList data structure. The page url string data is held in another String ArrayList. When a new page is added to the graph, the page string is added to this ArrayList, whilst a blank LinkedList is created in the adjacency list ArrayList, with the edges added as addEdge() is called.

Wikipedia page data can be categorised as sparse, i.e. each Wikipedia page (vertex) has a low number of links (edges) relative to the number of pages (vertices) in the graph, as seen in datasets analysed by Alshomrani & Iqbal (2012) and Rossi *et. al* (2013)

Whilst the algorithms used in this project may not necessarily be the most efficient comparing in terms of worst-time complexity, Rossi *et. al* (2013) notes that in the case of real-world networks, such as a weakly connected Wikipedia graph, where E (number of edges) is far less than V (number of vertices), cases where worst-case complexity apply lessen. The algorithms have been chosen on the basis of processing sparse but large graphs.

Given the input from previous academic literature on this subject then, it seems expedient to design a graph data structure which is optimised for a sparse graph that is nevertheless large in diameter. Whilst an adjacency matrix would be the most optimal in terms of time complexity, space complexity is $O(V^2)$ versus an adjacency list based implementation [O(V+E)]. This form of implementation makes more sense given this information.

Another advantage of an adjacency list based implementation is the constant time complexity to add new vertices, versus the need to create a new matrix (2D array), an operation which requires $O(V^2)$ time complexity

In order to get around the issue of a query(u,v) method that has a theoretical worst-case time complexity of O(V), due to the need to iterate through all elements of the linked list, the method getEdges(u) is provided instead, with a time complexity of O(1) instead returns all possible neighbours of the vertex u, and the algorithms have been designed around this method instead of calling query(u,v) (as well as saving a method creation system call).

Of further note is that regardless of implementation (matrix vs adjacency list), the method getIndex(string) has a time complexity of O(V).

	Adjacency Linked Lists	Adjacency Matrix
Storage Requirements	O(V+E) [storage]	$O(V^2)$ [storage]
addPage(u)	O(1)	$O(V^2)$
addEdge(u)	O(1)	O(1)
getSize()	O(1)	O(1)
queryEdge(u,v)	O(V)	O(1)
getEdges(u)	O(1)	O(V)
getIndex(s)	O(V)	O(V)

Shortest Path

To solve the shortest path problem, Dijkstra's algorithm (Dijkstra 1959) for the single-pair shortest path solution was utilised, implemented by using a min-heap binary heap structure, to implement the priority queue, along with an array of size V to hold the distances from the originating vertex.

Firstly, for a given source vertex (page), the algorithm first adds this item to the heap, which initialises the heap. The main loop is then entered, the item at the top of the priority queue is popped and each neighbouring edge that originates from this vertex is found. Next, a loop is entered which compares the distance of every neighbouring vertex to the vertex at the top of the priority queue with the values in the key[] array. If this distance is lower than what is stored in the array, the neighbouring vertex is then inserted into the min-heap priority queue. If this vertex is the destination vertex, the dist[] array is then returned for further processing to find the page url strings of the vertexIDs (this occurs in constant time). Once all neighbouring vertices have been processed, the priority queue is examined and if it has items within it, the main loop executes again.

Pseudocode

```
Dijikstra(source, dist)
      dist[source] <- 0

create min-heap priority Q

for each vertex v in Graph
      if v =/= source
            key[u] <- MAX_VALUE

Q.poll(v, key[v])

while Q is not empty:                        // O(V)
      u <- Q.pop()                           // O log(V)
      for each neighbour v of u              // O(E)
            if u = target
                  return key[]
            alt <- key[u] + length(u,v)      // O log(V)
            if alt < key[v]
            dist[v] <- alt

return dist[]
```

Time Complexity

Unlike utilising a List-based priority queue, this ensures that the poll() [change] and pop() [extract-min] methods called during the two loops have a worst-case time complexity of O(log V), as opposed

to O(V). This worst-case complexity arises when a node at every "level" (number of levels = log V) in the heap must be changed to maintain the *heapify* property, which keeps the items at lower priority at the root of the heap. The min-heap binary heap used is taken from the *java.util.PriorityQueue* library class (which does not have a priority decrease option).

The second interior loop is called at most E times [change], the resulting time complexity of this algorithm is O(VLog V + E log V) = **O(E log V).**

This time could further be theoretically improved by implementation of a Fibonacci heap (although the large hidden time constants may negate any theoretical gains in real world application), along with a heuristic guided method which complements the basic Dijkstra algorithm, however a suitable heuristic could not be found and given time constraints implementation of a Fibonacci heap was not attempted.

Hamilton Path

To solve the Hamiltonian path problem, which is a special case of the travelling salesman problem, which is NP-Complete, an exponential time algorithm devised by Bellman, Herd and Karp was used, based on the principles of dynamic programming (Held & Karp 1962). This is theoretically superior to a brute force approach of examining every possible permutation of paths, which has a worst case time complexity of O(V!). A backtracking and DFS (depth-first search) based approach was too rejected as recursive calls would represent a high hidden time constant for the algorithm.

The storage requirements for this algorithm were a 2D array (matrix) of size $V * 2^V$, and an array of size V to return the path (after further string processing).

Bit-masking was utilised in this implementation, which is based upon several sources (*HackerEarth* 2017; *Algorithms and Data Structures* 2017)

The dynamic programming method solves the Hamiltonian path problem by breaking it down into easier to calculate subsets.

Firstly, it follows that if a Hamiltonian path exists in a graph with *n* vertices and a vertex *n+1* is added which has an edge which originates from vertex *n*, a Hamiltonian path exists in the new graph, with *n+1* vertices.

i.e. A Hamiltonian path exists in graph G originating from u terminating at v if such a path exists terminating at n. [{0, 1, 2, … }->n-1 and a link exists where (n-1)-> n.

It is known that the number of subsets in the graph is related exponentially with the number of verticies in the graph (2^n).

Once a Hamiltonian path has been found, the algorithm then accesses the dp[][] matrix from the last bitmask index to redraw the path. The algorithm loops through all the subsets {0, 1, 2…}->V backwards until the Hamiltonian condition is true for a subset. n is then added to the result array, and all the subsets {0, 1, 2…}->n | $V \notin S$ are then examined. This continues until the bitmask value is 0 (and all vertices along the path have been added to the result array).

Pseudocode

```
BellmanHeldKarp()
for bitmask = 0 to 2^n
      for i = 0 to n
            dp[i][bitmask] <- false

for i = 0 to n
      dp[i][bitmask] = true

for bitmask = 0 to 2^n                    // O(2^V)
      for u = 0 to n                      // O(V)
            for each neighbour v of u     // O(E)
                  if dp[u][bitmask XOR 2^v] == true
                  [v][bitmask] <- true
                  continue

// Recreate Path
cur = 2^n - 1

for a = 0 to n                            //O(V)
      i = -1
      for v = 0 to n                      //O(V)
            if dp[v][cur] == true
                  i <- v
                  continue
      if i == -1
            return null

      result[a] <- i
      cur <- cur XOR 2^i  // Examine subsets without {i} on next iteration

return result
```

Bit-masking leverages the knowledge about the exponential nature of the subsets that exist as the number of vertices increases combined with the properties of binary numbers and bit manipulation.

If a vertex may only be visited once, there is no need to consider sets where a vertex is visited multiple time, i.e. {0, 1, 1, 2, 2…}. Thus it is possible to utilise each bit of the index integer of the array as a flag as to whether a vertex has already been visited. When the 3rd vertex in the graph is visited for example, the 3rd bit can be flipped to 1 to indicate this. That column [VERTEX][4] then represents the set {0, 3}->*VERTEX* for every vertex in the matrix.

Traversing through the columns in the matrix is possible by using the XOR and bitwise AND operators. For instance, running dp[VERTEX][4] **XOR** 1 << 4 , flips the 4th bit of the index and move to dp[VERTEX][12] which represents the set {0,4}-> VERTEX

dp[*VERTEX*][**MASK**]

MASK

0	0	0	1	0	0

{3}->VERTEX

-> (XOR)->

0	0	**1**	0	0	0

0	0	**1**	1	0	0

{3,4}->VERTEX

0	0	0	**1**	0	0

0	0	0	**1**		0	0

0	0	0	**1**	0	0

Time Complexity

The masking loop runs for 2^V times, which is reflected in the worst as well as amortised time complexity. The second loop runs for O(V) times, as the algorithm has to be calculated originating from every vertex in the graph. Finally, the final loop runs, in the worst case, in O(V) time, assuming all vertices have edges with every other vertex. This part of the algorithm has a worst time complexity of $O(V^2 2^V)$. Retracing the path through the graph requires $O(V^2)$ time in the worst case.

Thus the worst case time complexity of this algorithm is $O(V^2) + O(V^2 2^V) = O(\mathbf{V^2 2^V})$.

Strongly Connected Components

There are multiple possible approaches to solving the strongly connected component problem, which can be done in linear time, all are which are based around one or multiple passes of a depth-first search routine. This project implements a variation of Gabow's path-based strong component algorithm, based upon several sources (Alshomrani, S & Igbal; *Sanfoundry* 2017)

The selection of the Gabow algorithm was informed by literature comparing different path-based algorithms on various real-world networks, with Gabow being cited as faster but more memory intense than other implementations, such as Tarjan's algorithm (Alshomrani & Igbal 2012). This implementation utilises two stacks and several arrays of size V, together with an ArrayList of ArrayList<Integer> objects to store the component groups. The Stacks and ArrayList are implemented by the Java util library.

The first array added[] keeps track of which vertices have been added into a strongly connected component group.
The second array visited[] keeps track of which vertices have already been reached by the depth first search.
The third array search[] keeps track of how far from an originating parent vertex a vertex is.
The first and second stacks define the upper and lower bounds of verticies in a strongly component group (more on this later).

Firstly, for every vertex in the graph, a depth-first search routine is run. This calls a recursive function dfs(edge) for every edge of that vertex which have not been traversed before. Every time dfs is called, a class variable searchCount is incremented and a value stored in the search[searchCount] array, and the ID of the vertex is added to both stacks.

When a vertex (v) is reached which has edges to verticies which have already been visited by the depth first search or added to the scg ArrayList<ArrayList<Integer>> collection, a while loop is entered. This while loop processes the lower stack, peeks at the top of the stack, using this value as an index for the search[] array, popping from the stack while the search[peek] distance is greater than the search[v] distance. This moves backwards through the search array, and ends with the lower stack holding the value of the vertex at the "top" of the search. This occurs repeatedly for every vertex which is at a similar dead end.

A check is then made, if the lower stack's top item is equal to the vertex u for dfs(u), the difference between the two stacks is added to the scg ArrayList as a strongly connected group. This usually occurs further up the depth-first search chain, unless the vertex is in a single-vertex group.

Pseudocode

```
Gabow()
for v = 0 to n
      if visited[v] == false
            dfs(v)

return scg

dfs(int u)
      visited[u] <- true
      search[u] <- searchCount + 1
      upper.push(u)                        // Upper bound
      lower.push(u)                        // Lower bound

      for each neighbour v of u                        // O(E)
            if visited[u] == false
                  dfs(v)

            else if added[u] == false
                        while search[lower.peek()] > search[v]
                              lower.pop()

      if lower.peek() == u
            lower.pop()

            do
                  v <- upper.pop()
                  added[v]<- true
                  list.add(v)


            while v != u
            scg.add(list)
```

Time Complexity

The main loop (in Gabow()) runs, in the worst-case, runs the depth-first search V times, a O(V) worst-case time complexity.

The loop within the depth-first search recursive method runs in the worst case once for every edge in the graph, a O(E) worst case time complexity. Combined this represents a linear time complexity of O(V+E), however there are significant hidden time constants in regards to the recursive method calling of dfs().

Graph Centers

The graph center of a graph, going by the definition of a Jordan center, occurs where the greatest distance to any vertex in the graph is minimised. Finding the graph centers of the Wikipedia page graph was thus solved by firstly considering the all-paths shortest path problem, in order to find the distances between every vertex and the rest of the vertices in the graph.

Initially, two algorithms, Dijkstra's algorithm repeated for every vertex and the Floyd-Warshall algorithm were used, depending on graph density. However later performance studies deprecated the use of the Dijkstra's algorithm based solution in favour of an implementation entirely based on Floyd-Warshall.

The Floyd-Warshall algorithm incrementally processes the data, and represents another use of dynamic programming in this project.

Let d_{ij}^{M} represent the distance between I and j along a path that uses at most *M* edges, and D^{M} the matrix whose I,j entry is the value dij. The matrix $D^{(V-1)}$ contains the table of the all-pairs shortest path problem (as there can only be at most V-1 edges between two verticies).

It is possible to use D^{x-1} to calculate D^{x}. By breaking down the problem into a series of steps (D^{1}, D^{2}, D^{3} …) we can calculate $D^{(V-1)}$.

D^{1}, the length of the shortest path using at most one edge, is simply a copy of an adjacency matrix A, which is where we initialise the algorithm. (In the case of this program, as data is stored in an adjacency list, the information has to be firstly processed into an adjacency matrix.)

It can also be proven that $D^{M-1} * A$ gives D^{M} .

Therefore, by repeatedly conducting matrix multiplication of D by the adjacency matrix (in the first step, simply multiplying the adjacency matrix by itself) V-1 times, it is possible to solve the all-pairs shortest path problem.

<u>Pseudocode</u>

```
FlyodWarshall()

for int i = 0 to n
      if i != j
            dist[i][j] <- MAX_VALUE/2
      else
            dist[i][j] <- 0


for int u = 0 to n
      for each neighbour v of u
            dist[u][v] <- 1


for int k = 1 to n
      for int i = 1 to n
            for int j = 1 to n
                  if (dist[i][k] + dist[k][j]) < dist[i][j]
                        dist[i][j] = dist[i][k]+ dist[k][j]

return dist

getCenter(int[] dist)

minDist <- MAX_VALUE/2+1
for int i = 0 to n
      currE <- 0
      currMD <- 0


      for int j = 0 to n
            currDist <- dist[i][j]
            if currDist != MAX_VALUE/2 && currDist != 0
                  currE++
                  if currDist > currMD
```

```
                        currMD <-currDist

      if currE < minE
                  break

      if currMD <= minDist && currMD != 0
            if currMD < minDist
                  centerNames.clear()
            minDist = currMD
            centerNames.add(i)

return centerNames;
```

Time Complexity

Calculating time complexity for this algorithm is fairly simple, as it consists of three loops which run V times through a loop, giving a worst time complexity of $O(V^3)$

Performance Analysis

In order to analyse performance, a random function was utilised to create a random graph by specifying the number of edges and vertices. To simulate a sparseness similar to Wikipedia page data, the number of edges was set at d V^2, with a density (d) of 0.25 and various graphs of size 10, 20, 50, 100 and 1000. Whilst this is not a perfect approximation of the nature of page graph data for Wikipedia, it is still a useful heuristic to measure algorithm efficiency and performance and for ensuring that memory usage is kept in check (as this has not been a primary objective of my implementations).

Shortest Path

Execution time of the algorithm was below 1ms, even for the largest (sparse) graph, validating the graph structure and algorithm chosen for this problem.

Acounting function was thus implemented to calculate the amount of times Q.poll() and Q.push() are called.

Q.pop/Q.poll method calls

Trials \V	**10**	**20**	**50**	**100**	**1000**
1	9	24	58	55	1747
2	8	33	65	119	391
3	5	8	65	93	812
4	6	35	38	127	155
5	12	15	56	154	1305
Average	8	23	56	110	882

Graph Centers

When comparing an looped implementation of Dijkstra's algorithm to solve the all-pairs shortest path problems, versus the Floyd-Warshall algorithm, in terms of worst-case time complexity, the Dijkstra implementation was seen to be theoretically more efficient (Pettie 2004), with a worst-case time complexity of $O((V^2 + VE) \log V)$ versus the worst-case time complexity of Floyd-Warshall $O(V^3)$, as the graphs were assumed to be sparse, i.e. $E < V^2$.

This was tested by utilising the previous sparse graphs, as well as denser graphs (with a density of

Trials \V	**10**	**20**	**50**	**100**	**1000**
1	0	1	5	6	2982
2	0	0	7	8	2990
3	0	0	6	9	2940
4	0	1	5	8	2858
5	0	0	5	11	2811
Average	**0**	**0**	**6**	**8**	**2916**

0.8), and it was expected that the Floyd-Warshall algorithm would only outperform Dijkstra when graph density was higher.

Sparse Graphs - Floyd-Warshall (time ms)

Dense Graphs - Floyd-Warshall (time ms)

Trials \V	**10**	**20**	**50**	**100**	**1000**
1	0	1	6	9	2925
2	0	1	5	8	3104
3	0	0	7	10	2924
4	0	2	5	10	2901
5	0	0	4	8	3001
Average	**0**	**1**	**5**	**9**	**2971**

Sparse Graphs - Dijkstra (time ms)

Trials \V	**10**	**20**	**50**	**100**	**1000**
1	1	1	14	20	6251
2	1	2	15	21	6299
3	0	2	13	23	6160
4	1	2	14	25	6085
5	1	2	15	21	6297
Average	**1**	**2**	**14**	**22**	**6218**

Trials \V	**10**	**20**	**50**	**100**	**1000**
1	1	2	26	69	19548
2	1	3	28	70	19602
3	0	3	30	68	19304
4	1	2	33	63	19555
5	1	3	30	65	19431
Average	**1**	**3**	**29**	**67**	**19488**

As can be seen, the efficiency of Dijkstra when applied for the all-paths shortest path problem seems to be extremely far off from predictions, with a calculation time of more than double with a graph size of 1000 (6218 ms on average vs 2916) for sparse graphs, and 6.33 times longer for a dense graph with 1000 verticies.

A possible reason for this outcome is that the Dijkstra implementation has a high hidden constant time cost in the form of system calls for method creation, versus the simple set and get nature of manipulating arrays in Floyd-Warshall. Thus even when handling sparse graph data, the Floyd-Warshall algorithm was superior and thus this was chosen as the final implementation in my getCenters() solution.

References

Algorithms and Data Structures 2017, *Shortest Hamiltonian path in O(2^N * N^2)* Available from: https://sites.google.com/site/indy256/algo/shortest_hamiltonian_path

Alshomrani, S & Igbal, G 2012, 'Analysis of Strongly Connected Components (SCC) Using Dynamic Graph Representation'*, International Journal of Computer Science Issues*, vol. 9, no. 1, pp 94-100. Available from: http://www.ijcsi.org/papers/IJCSI-9-4-1-94-100.pdf

Dijkstra, E 1959, 'A Note on Two Problems in Connexion with Graphs', *Numerische Mathematik, vol. 1, pp.* 269-271.

HackerEarth 2017, *Hamiltonian Path.* Available from: https://www.hackerearth.com/practice/algorithms/graphs/hamiltonian-path/

Held, M & Karp, RM 1964, 'A Dynamic Programming Approach to Sequencing Problems'. *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196-210. Available from: JSTOR

Pettie, S 2004, 'A new approach to all-pairs shortest paths on real-weighted graphs', *Theoretical Computer Science, vol. 312, no. 1,* pp. 47-74.

Rossi, RA, Gleich, DF, Mostofa, AP 2013, 'A Fast Parallel Maximum Clique Algorithm for Large Sparse Graphs and Temporal Strong Components', *Purdue University.* Available from: http://docs.lib.purdue.edu/ccpubs/525/

Sanfoundry 2017, *Gabow Algorithm.* Available from: http://www.sanfoundry.com/java-program-gabow-algorithm/