Djikstra Algorithm Implementation

Ndindi Rachael Kilonzo

31/05/21

1 INTRODUCTION

The shortest path problem is one of finding the distance from one spot to another/other spots in a network. Using a road network of roads in Rome obtained from `http://www.diag.uniroma1.it//challenge9/data/rome/rome99.gr` this project aims, to implement the Dijkstra algorithm in two different ways while finding shortest path. A Graphs is a collection of vertices connected by vertices. Vertices will be places in a map objects and edges represent the distance of a road stretch connecting two vertices. are implemented commonly through adjacency matrices or adjacency lists which have their prospective pros and cons. No negative weights are encountered as the entries represent physical road networks and the graph is directed .

2 Djikstra Algorithm with Adjacency Matrix

The first implementation is through an adjacency matrix for the graph which is more innate and simpler to visualize in comparison to adjacency lists. The algorithm as studied in class is:

- Set all vertices as unseen/unvisited and create a set of such vertices.
- Assign provisional distances as zero for the source vertex and infinity for all the other vertices.
- Create an empty set for storing visited/seen vertices. Sets are used so that we have $\mathcal{O}(1)$ search time rather than $\mathcal{O}(n)$ had we used arrays.
- Have an array representing the jumps taken to include the source vertex itself.
- While all vertices have not been visited:
- Set current vertex to the vertex with the smallest provisional distance in the entire graph. For the first iteration, it will be the source vertex because that distance is 0.
- Add current vertex to the visited vertices set.
- Update the provisional distance of each of current vertices neighbors to be the distance from current vertex and the edge length from current node to that neighbour if that value is less than the neighbour's current provisional distance. If this neighbor has never had a provisional distance set, remember that it is initialized to infinity and thus must be larger than this sum. If we update provisional distance, also update the "hops" we took to get this distance by concatenating current node's hops to the source node with current node itself.
- End While.

3 Heaps

The issue with the first implementation using the adjacency matrix is that during the iteration process, we search through a "queue" to find the vertex with smallest provisional distance which is a $\mathcal{O}(n)$ process n times at every iteration. The algorithm is actually $\mathcal{O}(n^2)$. An improvement may be to use a a heap that employs priority queues. To obtain the smallest provisional distance, Fibonacci heaps implement them in $\mathcal{O}(1)$ because heaps are used to provide a vertex with the smallest provisional distance readily since they are an implementation of priority queues. The priority

being the minimum. When the minimum, taken from the heap, we ensure that the heap updates itself to provide the next relevant shortest distance. The Heap Property is that for a minimum heap, every parent must be less than or equal to both of its children. Worth mentioning is that with heaps we can implement our graph with an Adjacency List, where each vertex has a list of its linked vertices rather than having to look through all vertices to see if a link exists.

3.1 Dijkistra with Fibonacci Heaps

Named so because Fibonacci numbers are used in running time analysis, Fibonacci heaps are a collection of trees that can have any shape even to the extent of single vertex. For the case of Dijkistra, we use the trees with minimum heap property. It will have a pointer to the minimum key which is usually the root. They are implemented using circular and doubly linked root list where each vertex has a two pointers(one for parent and the other for the children). Children for each vertex are doubly linked in a circular linked list. Merge operations links two heaps, insertion adds a new tree with a single node With the Fibonacci heap as a priority queue the Dijkstra runs at $\mathcal{O}(E + v \log V)$ and asymptotically fastest time complexity known for Dijkstra.

Operation	Performance cost
make-heap	1
is-empty	1
insert	1
extract-min	log n
decrease-key	1
delete	log n
union	1
find-min	1

Insertion

1. Create a new singleton tree of a vertex "new".
2. Check whether the heap is empty or not.
   - Make new the only vertex in the root_list.
   - Set pointer of the minimum of the heap as new.
3. Else:
   - Insert new into the root_list and update the minimum of the heap.

Merge / Union of Heaps

1. Join the root_lists of the two Fibonacci heaps H_1 and H_2 to make one heap H.
2. If min(H_1) ¡ min(H_2),
   - min(H)=min(H_1)
3. Else:
   - min(H)=min(H_2)

Extracting the minimum value It deletes the minimum vertex and reassigning the pointer to the the minimum value of what's left in the heap.

1. Delete minimum vertex
2. Set the head to the next minimum vertex and then meld all the children of the deleted into the root_list.
3. Create an array of degree pointers of the size of the deleted vertex.
4. Set degree pointer to the current node.
5. Move to the next node.
   - If degrees are different then set degree pointer to next node.
   - If degrees are the same then join the Fibonacci trees by union operation
6. Repeat steps 4 and 5 until the heap is completed

A Fibonacci heap as a priority queue, runs in$\mathcal{O}(E + VlogV)$

3.2 Implementation of Fibonacci

4 References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 1990. Introduction to algorithms mit press. Cambridge MA.
2. Kozen, D.C., 1992. The design and analysis of algorithms. Springer Science & Business Media.

3. http://bioinfo.ict.ac.cn/~dbu/AlgorithmCourses/Lectures/Fibonacci-Heap-Tarjan.pdf

4. https://www.cosc.canterbury.ac.nz/research/reports/HonsReps/1999/hons_9907.pdf