Single Source Shortest Paths Problem

DAA Assignment 6 - Group 21

Divyesh Rana
IIT2019063

Akash Deep
IIT2019064

Gopal Pedada
IIT2019065

***Abstract*—This paper contains the design and the detailed analysis of the algorithm used to solve the following problem: Single source shortest distance problem - find shortest paths from source vertex to all other vertices in a graph.**

I. Introduction

The shortest path problem is to find a path between two vertices such that the sum of the weights of edges in the path is minimized. The single-source shortest path (SSSP) problem is to find shortest paths from a source vertex v to all other vertices in the graph. It is a classical problem in graph theory and has many real life applications.

This report further contains -

II. Algorithmic Design

III. Algorithm Analysis

IV. Experimental Study

V. Conclusion

II. Algorithmic Design

The source s is given from which the distances of all other nodes are calculated. Given graph is assumed to be **undirected-graph** with n nodes and m edges and stored as adjacency lists.

A. Unweighted Graph

We can find the shortest paths between a single source and all other vertices using standard BFS(Breadth First Search). The distance is the minimum number of edges that you need to traverse from the source to another vertex.

BFS can be implemented using queue q that contains vertices or nodes. Take two arrays $visited$ and $distance$ of size n(no. of nodes), the elements in arrays store boolean value whether a node has been visited and the distance between source and a node respectively. Initialise both arrays with 0.

Push source node s in q and mark $visited[s] = 1$ and $distance[s] = 0$ and proceed the following steps until the q is empty:

1. Take front element in q and store in cur and pop an element from q.

2. For each non-visited nodes k connected with cur, mark $visited[k] = 1$ and $distance[k] = distance[cur] + 1$, and push the node k in q.

B. Graph contains all non-negative weights

Dijkstra's algorithm can be used for this case, Dijkstra's algorithm finds the shortest paths from the starting node to all other nodes. However,the algorithm requires that there are no negative weight edges in the graph.

Dijkstra's algorithm maintains distances to the nodes and reduces them during the search. Dijkstra's algorithm is efficient,because it only processes each edge in the graph once, using the fact that there are no negative edges.

Dijkstra's algorithm calculates the minimum distances from a node s to other nodes of the graph. The graph is stored as adjacency lists so that $adj[u]$ contains a pair (v, w) always when there is an edge from node u to node v with weight w.

An appropriate data structure for this is a priority queue that contains the nodes ordered by their distances. Using a priority queue, the next node to be processed can be retrieved in logarithmic time.

Take a priority queue q contains pairs in form of $(-d, k)$ meaning the current distance to the node k is d. Take an array $distance$ contains the distance to each node, and the array $processed$ indicates whether a node has been $processed$. Initially the distance is 0 to s and ∞ for all other nodes.

Push pair $(0, s)$ in q and repeat the following steps until the q is empty.

1. store second parameter of top element of q in u and pop an element from q.

2. If $processed[u] = false$ then mark $processed[u] = true$ and for each x in $adj[u]$, take two variables v equals first parameter of x and w equals second parameter of x. If $distance[u] + w < distance[v]$, then assign $distance[u] + w$ in $distance[v]$ and push a pair $(-distance[v], v)$ in q.

Here, the priority queue q contains negative distances because the priority queue finds maximum elements but we want to find minimum elements.

C. Graph contains some negative weights

Bellman-Ford algorithm can be used for this case, The Bellman–Ford algorithm finds shortest paths from a source node to all nodes of the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length.

The algorithm keeps track of distances from the starting node to all nodes of the graph. Take an array $distance$, Initially, the distance to the source node s is 0 and the distance to all other nodes in ∞. The algorithm reduces the distances

by finding edges that shorten the paths until it is not possible to reduce any distance.

The design assumes that the graph is stored as an edge list edges that consists of tuples or structure of the form (u, v, w), meaning that there is an edge from node u to node v with weight w.

The algorithm consists of $n-1$ rounds, and on each round the algorithm goes through all edges of the graph and tries to reduce the distances, this can be done using two loops.

Algorithm 1: BFS

```
Input: source s,adjacency list adj
Output: distance array
Function BFS(s, adj):
    visited[s] ← 1
    distance[s] ← 0
    q.push(s)
    while q.empty() = false do
        cur ← q.front()
        q.pop()
        for auto k : adj[cur] do
            if visited[k] = 1 then
                continue
            visited[k] ← 1
            distance[k] ← distance[cur] + 1
            q.push(k)
```

Algorithm 2: Dijkstra's Algorithm

```
Input: source s, no. of nodes n,adjacency list adj
Output: distance array
Function SSSP(s, n, adj):
    for i ← 0 to n − 1 do
        distance[i] ← Inf
    distance[s] ← 0
    q.push(0, s)
    while q.empty() = false do
        u ← q.top().second
        q.pop()
        if processed[u] = true then
            continue
        processed[u] ← true
        for auto x : adj[u] do
            v ← x.first
            w ← x.second
            if distance[u] + w < distance[v] then
                distance[v] ← distance[u] + w
                q.push(−distance[v], v)
```

Algorithm 3: Bellman-Ford Algorithm

```
Input: source s, no. of nodes n, edge list edges
Output: distance array
Function SSSP(s, n, edges):
    for i ← 0 to n − 1 do
        distance[i] ← Inf
    distance[s] ← 0
    for i ← 1 to n − 1 do
        for auto e : edges do
            tie(u, v, w) ← e
            if distance[v] > distance[u] + w then
                distance[v] ← distance[u] + w
```

III. Algorithm Analysis

A. Time Complexity:

Approach 1: the time complexity of breadth first search is $O(n+m)$, where n is the number of nodes and m is the number of edges.

Approach 2: The time complexity of this algorithm is $O(n+mlogm)$, because the algorithm goes through all nodes of the graph and adds for each edge at most one distance to the priority queue.

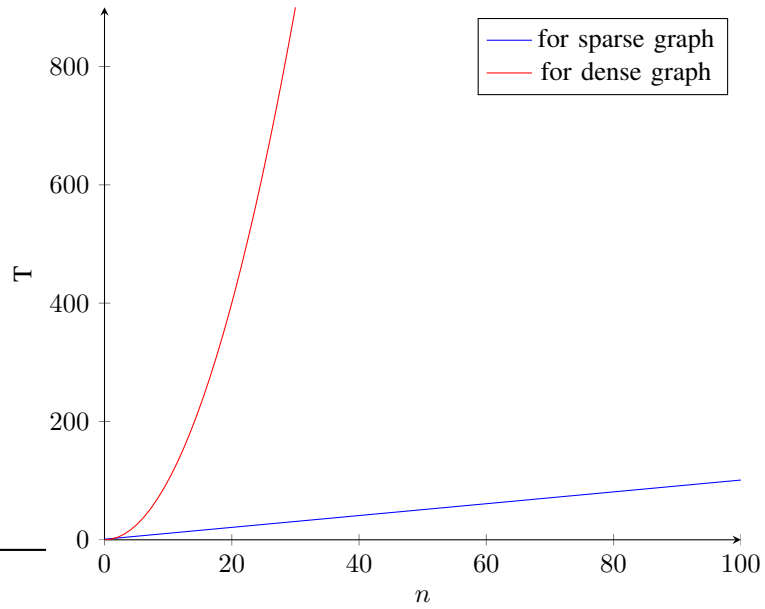
Approach 3: The time complexity of the algorithm is $O(nm)$, because the algorithm consists of $n-1$ rounds and iterates through all m edges during a round.

B. Space Complexity

The space complexity for first two algorithms is $O(n+m)$ for adjacency list + $O(n)$ for other arrays, and for third algorithm, the complexity is $O(m)+O(n)$.

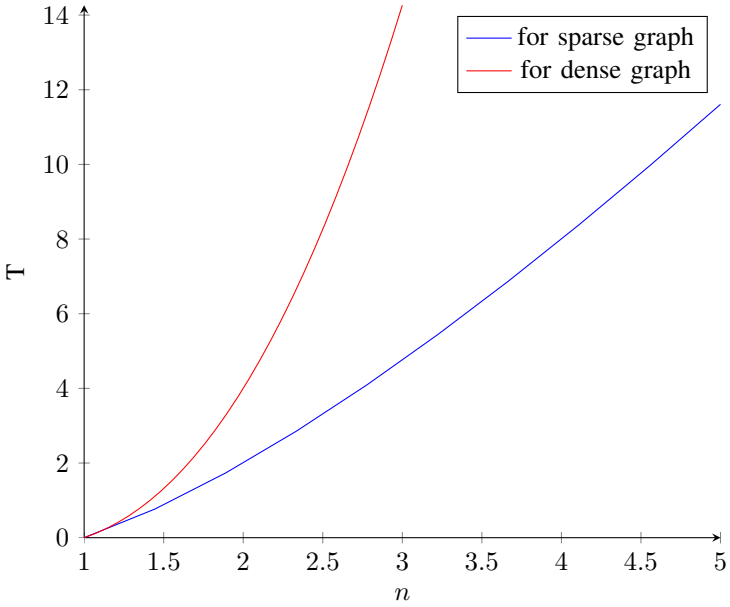
IV. Experimental Analysis

graphical analysis of algorithm - 1



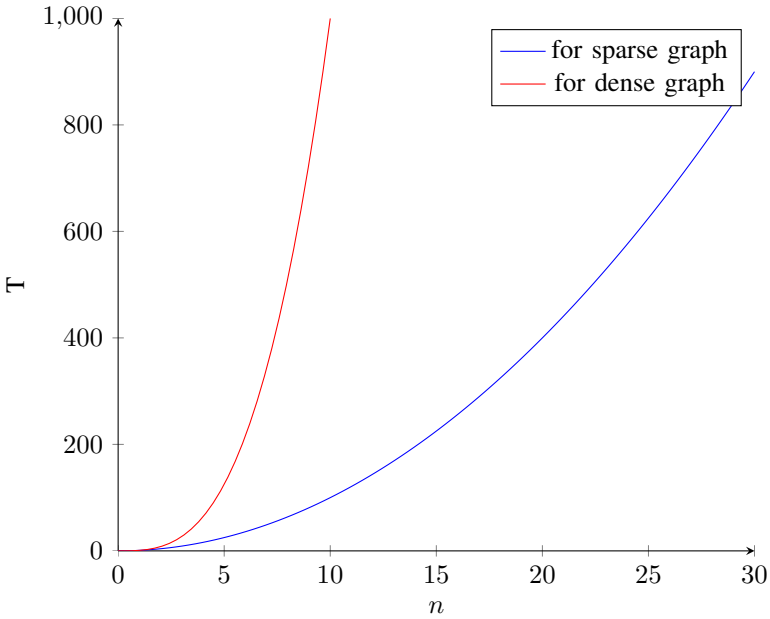
Graph 1 : for sparse graph, no. of edges = n, the graph has linear behaviour and for dense graph, no. of edges = n^2, the graph is showing quadratic behaviour.

graphical analysis of algorithm - 2



Graph 2 : for sparse graph, no. of edges = n, the graph has somewhat linear behaviour and for dense graph, no. of edges = n^2, the graph is showing somewhat quadratic behaviour.

graphical analysis of algorithm - 3



Graph 3 : for sparse graph, no. of edges = n, the graph has quadratic behaviour and for dense graph, no. of edges = n^2, the graph is showing somewhat cubic behaviour.

V. Conclusion

Here we discussed algorithms to solve SSSP for undirected graph. For Unweighted graph, the distances can be computed in almost linear time complexity but for weighted graph, there exist several algorithms for different types of graphs.

VI. Reference

1) https://en.wikipedia.org/wiki/Shortest_path_problem
2) https://www.tutorialspoint.com/
single-source-shortest-paths-arbitrary-weights