Single Shortest Distance Problem

Design and Analysis of Algorithms Assignment - 3

Department of Information Technology

Indian Institute of Information Technology - Allahabad, India

Jaidev Das
IIT2019197

Deeptarshi Biswas
IIT2019195

Abstract: ***This paper discusses the methods involved in solving the shortest distance problem. First we go over a brief introduction of the algorithms used, namely Dijkstra and Bellman Ford algorithms. Then we go over the algorithms themselves. Finally, there's an analysis of time and space and complexities of both algorithms and a brief comparison ending with a conclusion.***

Index Terms:Arrays, XOR bitwise operator, Binary numbers, Dynamic programming

INTRODUCTION

Problem Statement The shortest distance problem involves, given a graph and a source vertex in the graph, finding shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm Dijkstra's algorithm provides an efficient solution to this problem. It is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST(minimum spanning tree), we generate a SPT (shortest path tree) with a given source as root. We maintain two sets, one set contains vertices included in the shortest path tree, the other set includes vertices not yet included in the shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap)

Bellman Ford algorithm An alternative algorithm to solve this problem is Bellman Ford's algorithm. . While Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suits well for distributed systems. But the time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

Floyd-Warshall algorithm The Floyd–Warshall algorithm (also known as Floyd's algorithm, the Roy–Warshall algorithm, the Roy–Floyd algorithm, or the WFI algorithm) is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm. Versions of the algorithm can also be used for finding the transitive closure of a relation R, or (in connection with the Schulze voting system) widest paths between all pairs of vertices in a weighted graph.

Contents This report further contains:

- Algorithm Designs
- Algorithm Analysis
- References

ALGORITHM DESIGN

BFS(for unweighted graph: First we look at an algorithm which works for unweighted graphs to solve the single shortest distance problem.

```
bool BFS(vector<int> adj[], int src, int dest,
    int v,
                int pred[], int dist[])
{

        list<int> queue;

        bool visited[v];

        for (int i = 0; i < v; i++) {
                visited[i] = false;
                dist[i] = INT_MAX;
                pred[i] = -1;
        }
        visited[src] = true;
        dist[src] = 0;
        queue.push_back(src);


        while (!queue.empty()) {
                int u = queue.front();
                queue.pop_front();
                for (int i = 0; i < adj[u].size
                    (); i++) {
                        if (visited[adj[u][i]]
                            == false) {
                                visited[adj[u][i
                                    ]] = true;
                                dist[adj[u][i]]
                                    = dist[u] +
                                    1;
                                pred[adj[u][i]]
                                    = u;
                                queue.push_back(
                                    adj[u][i]);
                                if (adj[u][i] ==
                                     dest)
                                        return
                                            true
                                            ;
                        }
                }
        }

        return false;
}


void printShortestDistance(vector<int> adj[],
    int s, int dest, int v)
{

        int pred[v], dist[v];

        if (BFS(adj, s, dest, v, pred, dist) ==
            false) {
                cout << "Given source and
                    destination"
                        << " are not connected";
                return;
        }

        vector<int> path;
        int crawl = dest;
        path.push_back(crawl);
        while (pred[crawl] != -1) {
                path.push_back(pred[crawl]);
                crawl = pred[crawl];
        }

        cout << "Shortest path length is : "
                << dist[dest];
        cout << "\nPath is::\n";
        for (int i = path.size() - 1; i >= 0; i
            --)
                cout << path[i] << " ";
}
int main()
{
        int v,e;
  cin>>v>>e;
        vector<int> adj[v];
  int x,y;
  for(int i=0;i<e;i++){
    cin>>x>>y;
    adj[x].push_back(y);
  }
        int v1,v2;
  cin>>v1>>v2;
        printShortestDistance(adj, v1, v2, v);
        return 0;
}
```

Dijkstra's Algorithm (using adjacency matrix): Then we look at the Dijkstra's algorithm using adjacency matrix. This can be considered the naive approach as better algorithms with better efficiencies exist. Below, the algorithm is discussed.

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3. While sptSet doesn't include all vertices

   - Pick a vertex u which is not there in sptSet and has minimum distance value.
   - Include u to sptSet.
   - Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Pseudocode – matrix version

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Name		s	t	x	y	z
	ID	1	2	3	4	5
s	1	0	10	+INF	5	+INF
t	2	+INF	0	1	2	+INF
x	3	+INF	+INF	0	+INF	4
y	4	+INF	3	9	0	2
z	5	7	+INF	6	+INF	0

With adjacency/weight matrix W lines 7-8 become

```
7 for (v=1;v<=n;v++) if (W[u,v]!=INF) // only neighbors
8     Relax(u,v,W) // update distances
```

Computational complexity: $O(V^2)$, two loops, each going through V iterations

Dijkstra's Algorithm (using adjacency list): Here, we go over the Dijkstra's algorithm, this time using adjacency list. This is a fairly efficient algorithm for solving the single shortest distance problem but it should be noted that this algorithm does not work for negative weights in the graph.

1. Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.
2. Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).
3. While Min Heap is not empty, do following
   - Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u.
   - For every adjacent vertex v of u, check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of u-v plus distance value of u, then update the distance value of v.

Bellman Ford's algorithm: Now we take a look at Bellman Ford's algorithm. It's an efficient algorithm for solving the single shortest distance problem but less so than Dijkstra's algorithm. However, this algorithm works for negative weighted edges too in the graph.

1. This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size V with all values as infinite except dist[src] where src is source vertex.
2. This step calculates shortest distances. Do following V-1 times where —V— is the number of vertices in given graph.
   - Do following for each edge u-v
   - If dist[v] ¿ dist[u] + weight of edge uv, then update dist[v] dist[v] = dist[u] + weight of edge uv
3. This step reports if there is a negative weight cycle in graph. Do following for each edge u v
   - If dist[v] ¿ dist[u] + weight of edge uv, then "Graph contains negative weight cycle".

Bellman-Ford algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$ } initialization
do $d[v] \leftarrow \infty$

for $i \leftarrow 1$ **to** $|V| - 1$
do for each edge $(u, v) \in E$
do if $d[v] > d[u] + w(u, v)$ } ***relaxation step***
then $d[v] \leftarrow d[u] + w(u, v)$

for each edge $(u, v) \in E$
do if $d[v] > d[u] + w(u, v)$
then report that a negative-weight cycle exists

At the end, $d[v] = \delta(s, v)$, if no negative-weight cycles.
Time = $O(VE)$.

Pseudocode – list version

We still have the problem of u=ExtractMin(Q)

```
Dijksktra(G,w,s)
 1  InitializeSingleSource(G,s)
 2  InitializeQ(G.V)
 3  while (notEmpty(Q))
 4   u=ExtractMin(Q)
 5   listElement=L[u]
 6   while (listElement!=NIL)
 7     v=listElement.vertexID;
 8     wUV=listElement.edgeWeight;
 9     Relax(u,v,wUV) // here we may
  decrease v.d
10     listElement=listElement->next
```

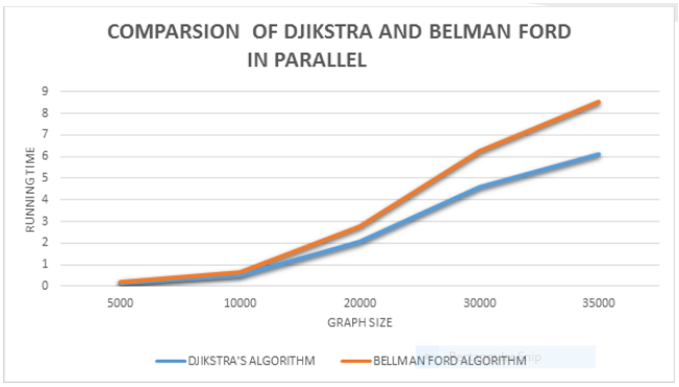
	s	t	x	y	z
ID	1	2	3	4	5
Q	F	T	T	F	F

ExtractMin(Q) in total still costs $O(V^2)$
Can we have faster ExtractMin(Q)?

Floyd-Warshall's algorithm: Now we take a look at Floyd-Warshall's all pair shortest distance method.

1. We initialize the solution matrix same as the input graph matrix as a first step.
2. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
3. When we pick vertex number k as an intermediate vertex, we already have considered vertices 0, 1, 2, .. k-1 as intermediate vertices.

4. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
   - k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
   - k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] ¿ dist[i][k] + dist[k][j].



Data: n, M_{floyd}
Result: M_{floyd}
begin
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      for $k \leftarrow 1$ **to** n **do**
        if $M_{floyd}[i,j] \geq (M_{floyd}[i,k] + M_{floyd}[k,j])$ **then**
          $M_{floyd}[i,j] = M_{floyd}[i,k] + M_{floyd}[k,j];$

ALGORITHM ANALYSIS

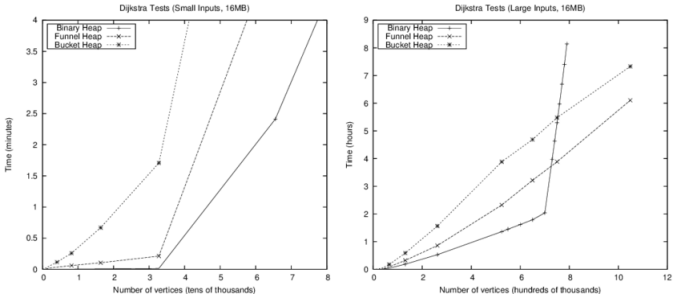
Comparison between Dijkstra's and Bellman Ford's algorithms: With the help of a table and a following graph, we compare the two algorithms. At first glance, Dijkstra's algorithm comes out to be better but we also have to note that only Bellman Ford's algorithm supports negative weights.

Dijkstra's algorithm	Bellman Ford's algorithm
Dijkstra's Algorithm doesn't work when there is negative weight edge.	Bellman Ford's Algorithm works when there is negative weight edge, it also detects the negative weight cycle.
The result contains the vertices containing whole information about the network, not only the vertices they are connected to.	The result contains the vertices which contains the information about the other vertices they are connected to.
It can not be implemented easily in a distributed way.	It can easily be implemented in a distributed way.
It is less time consuming. The time complexity is O(E logV).	It is more time consuming than Dijkstra's algorithm. Its time complexity is O(VE).
Greedy approach is taken to implement the algorithm.	Dynamic Programming approach is taken to implement the algorithm.

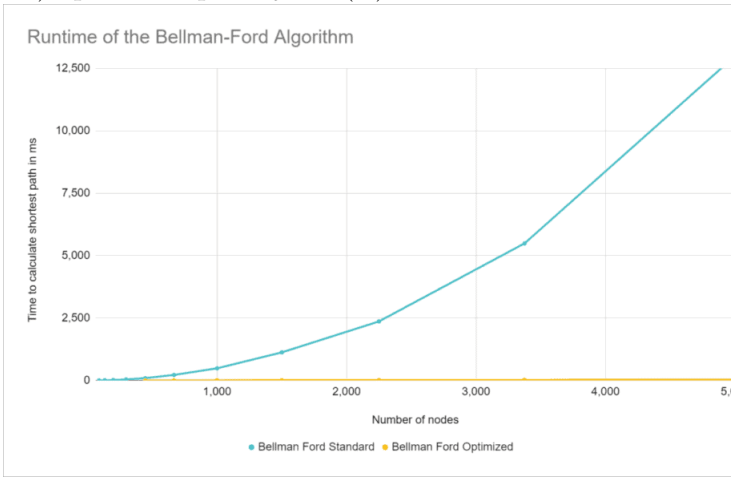
BFS (unweighted graph): Time complexity: $O(V+E)$ **Space complexity:** $O(V+E)$

Dijkstra's algorithm (adjacency matrix): Time complexity: $O(V^2)$ **Space complexity:** $O(V+E)$

Dijkstra's algorithm (adjacency list): Time complexity: $O(ElogV)$ **Space complexity:** $O(V)$



Bellman Ford's algorithm: Time complexity: $O(VE)$ **Space complexity:** $O(V)$



Floyd-Warshall's: Time complexity: $O(V^3)$ **Space complexity:** $O(V^2)$

REFERENCES

1. https://livebook.manning.com/book/grokking-algorithms/chapter-7/1
2. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
3. https://cp-algorithms.com/graph/dijkstra.html#toc-tgt-2
4. https://cp-algorithms.com/graph/bellman_ford.html
5. https://cp-algorithms.com/graph/all-pair-shortest-path-floyd-warshall.html