Fast Computation of Clustered Many-to-many Shortest Paths

Final Report

Prepared by

Aniket Sangwan (180001005)
Sarthak Jain (180001047)

Indian Institute of Technology Indore
Design and Analysis of Algorithms (CS254)

Table of Contents

Chapter 1

Introduction

There are certain situations in which it is necessary to efficiently compute shortest paths in a transportation network from each of a number of source nodes lying within a bounded area to each of a number of target nodes outside it. A number of successful algorithms based on a variety of approaches have been proposed in the literature for map matching sparse and noisy trajectories. They involve the computation of shortest paths from each candidate point inside the error region of a location measurement to each candidate point inside the error region of the subsequent location measurement. We aim at implementing an algorithm that helps in computing the MSP with high accuracy and reduced complexity.

1.1 Motivation

There have been recent advances in map matching sparse and noisy trajectories using Wifi based and cellular network based positioning technologies. These technologies are energy-efficient and easily available in GPS-denied environments. But they cause large positioning errors which require computation of the shortest path from each candidate point in the source region to each candidate point in the target region. In case of large errors, it requires a large number of SSP computations. This problem is a special case of the MSP problem in which the source nodes are geographically clustered. Its requirement in map matching motivated us to choose this topic for our project.

1.2 Objectives

- Learning basic shortest path algorithms like Dijkstra, Floyd-Warshall, Bellman Ford, etc.
- Implementing modified Dijkstra Algorithm to reduce the time complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$.
- Implementation and analysis of existing MSP algorithms.
- To efficiently compute shortest paths from a cluster of source nodes to a set of target nodes.

Chapter 2

Related Work

The MSP Problem is not much tackled by researchers. Instead, they work on improving the running time of pre-existing algorithms like Dijkstra. Some of the past researhed are as follows.

- One of the proposed method involves applying a hierarchical acceleration technique to the MSP problem. Their basic idea involves performing limited backward searches from each target node on a hierarchically organized road network and storing the search spaces so the stored information is accessed during the forward searches from each source node. The major drawback for this method is that it requires preprocessing. Thus, a lot of computation is required when there are major changes in road network.
- Another goal-directed method first computes a bounded backward search space from all the target nodes and then uses a lower-bound estimate of the shortest path length to the set of target nodes to accelerate the search from each source node.
- Another method involves goal-directed and bidirectional search techniques based on the concept of landmarks. The major limitation for goal-directed and bidirectional approach is that it is effective only when both the source nodes and target nodes are clustered.

Chapter 3

Possible Methodology

3.1 Assumptions

- We are using a road network of a section of Indore city, derived from OpenStreetMap data.
- The graph is treated as undirected, connected and edge-weighted.
- The source nodes are clustered in a region while the target nodes can be sparse.

There are a number of options available for SSP computation:

1. Bellman Ford - $\mathcal{O}(|E| \cdot |V|)$
2. Floyd-Warshall - $\mathcal{O}(|V|^3)$
3. Dijkstra's Algorithm - $\mathcal{O}(|V| + |E| \cdot \log(|V|))$

Clearly, Dijkstra's algorithm is the best way for SSP computation.

Now the MSP problem can be solved by computing SSP for each source node, which leads to very high computational complexity. Instead we use the fact that the shortest path originating from the source region crosses the source region's boundary through a smaller number of nodes. We work forward on this idea and implement the algorithm detailed in the next section.

Chapter 4

Algorithm Design

At first, a circular region containing the potential source nodes is selected out of which a fixed number of source nodes are selected at random. Along with this, we define a secondary graph V' which contains direct edges from source nodes to exit nodes and exit nodes to target nodes. This is explained in detail in the algorithm. After that, we apply the designed algorithm to find the shortest path distance.

The algorithm is divided into the following three steps :

1. Identifying the exit nodes and computing the shortest path distance between each exit node and target node.
2. Computing the shortest path distance between each source node and exit node.
3. Optimally combining the distances computed in the above steps to find the shortest path distance between the source nodes and target nodes.

4.1 Steps

4.1.1 Step 1

In this step, we identify the exit nodes and their shortest distance to each target node. This process begins with identifying the potential exit nodes. The nodes lying near the boundary or on the boundary of the circular region which either of which always lies in every path from the source nodes to the target nodes are the potential exit nodes. These nodes are identified such that the other endpoint of the edge passing through these nodes lie outside the circular region.

With each of the potential exit nodes pi , we perform an SSP computation which determines the shortest distance between the potential exit node to all the target nodes. This is computed using the Modified Dijkstra Algorithm with a time complexity of O(nlogn). Along with the modification to improve the time complexity, the algorithm is further used to compute the Last Potential Exit Node (LPEN) for each of the target nodes which is basically the last visited potential exit node before reaching the target node.

If, for any target node t_j , LPEN(t_j) equals p_i , we perform the following calculations :

- Mark p_i as an exit node.
- Mark p_i and t_j as nodes in $V^{'}$.
- Insert an edge between p_i and t_j in $V^{'}$ with weight equals the shortest distance calculated between them with the Dijkstra Algorithm.

4.1.2 Step 2

In this step, we compute the SSP from each source node to each exit node. This is done using the reversed graph since the count of the exit nodes is less than the count of source nodes, finding SSP from exit nodes to the source nodes rather than from source nodes to the exit nodes saves computation time. The reversed graph is calculated during the input step. This is also performed using the Modified Dijkstra Algorithm. After that the following calculation is done for each exit node e_i :

- For each source node s_j , nodes e_i and s_j are marked as nodes in $V^{'}$.
- For each source node s_j , an edge between e_i and s_j is added to the $V^{'}$ with weight equal to the shortest distance calculated during SSP computation.

4.1.3 Step 3

This is the final step of the algorithm in which we combine the source node to target nodes passing through the exit nodes and thus finally find the shortest distance between the source nodes and the target nodes. Since this will be a path with at most two edges, the Dijkstra Algorithm can be further modified to work in linear time complexity. This has also been implemented in the algorithm.

Thus in this way, with the above mentioned 3 step algorithm, one can find the shortest path distance between the source nodes to the target nodes efficiently.

Chapter 5

Algorithm Analysis

The conventional solution requires $|S|$ SSP computations and the running time of single SSP computation for Dijkstra's Algorithm is $\mathcal{O}(|V| + |E| \cdot \log(|V|))$.

Let us represent this as D. Hence the baseline algorithm has a runtime complexity of $\mathcal{O}(|S| \cdot D)$.

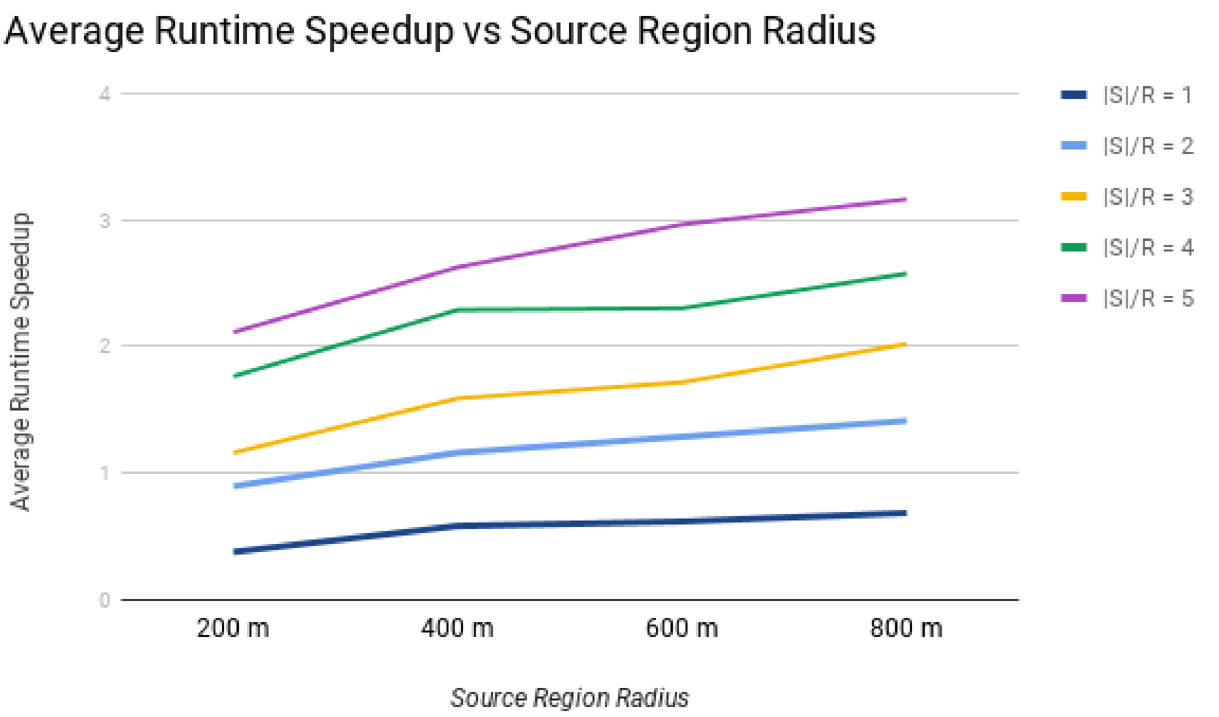
The proposed algorithm, on the other hand, contains $|P|$ SSP computations in Step 1, where $|P|$ is equal to the number of potential exit nodes. In Step 2, $|X|$ SSP computations are performed, where $|X|$ represents the number of exit nodes. As the shortest path in Step 3 consists of only two edges, its complexity is considered negligible. Hence, it involves $(|P| + |X|)$ SSP computations. Here $X \subseteq P$, so the running time of the proposed algorithm is $\mathcal{O}(|P| \cdot D)$.

Thus the speed-up provided by the proposed algorithm over the baseline algorithm is $\mathcal{O}(\frac{|S|}{|P|})$. In cases of map matching, $|S|$ is generally greater than $|P|$. Clearly, the number of potential exit nodes in a circular region will be proportional to its circumference, i.e. $|P|$ is expected to be linearly related to R. Thus, the speedup increases with increase in $\frac{|S|}{R}$.

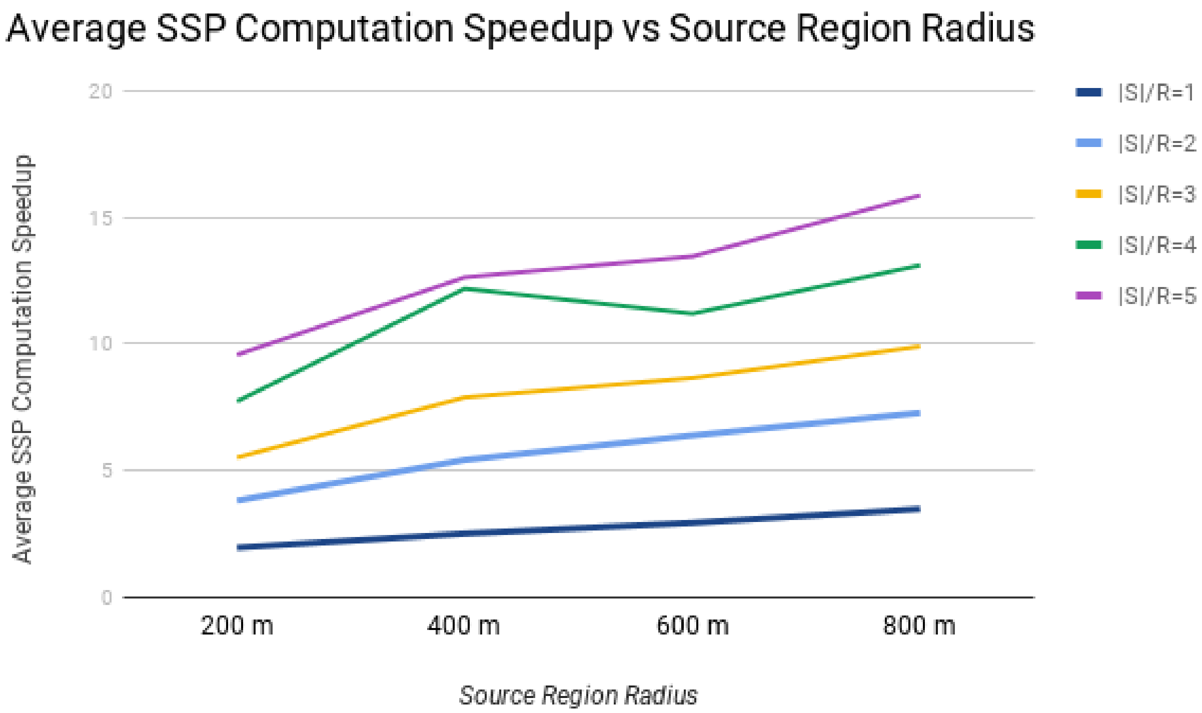
Chapter 6

Evaluation

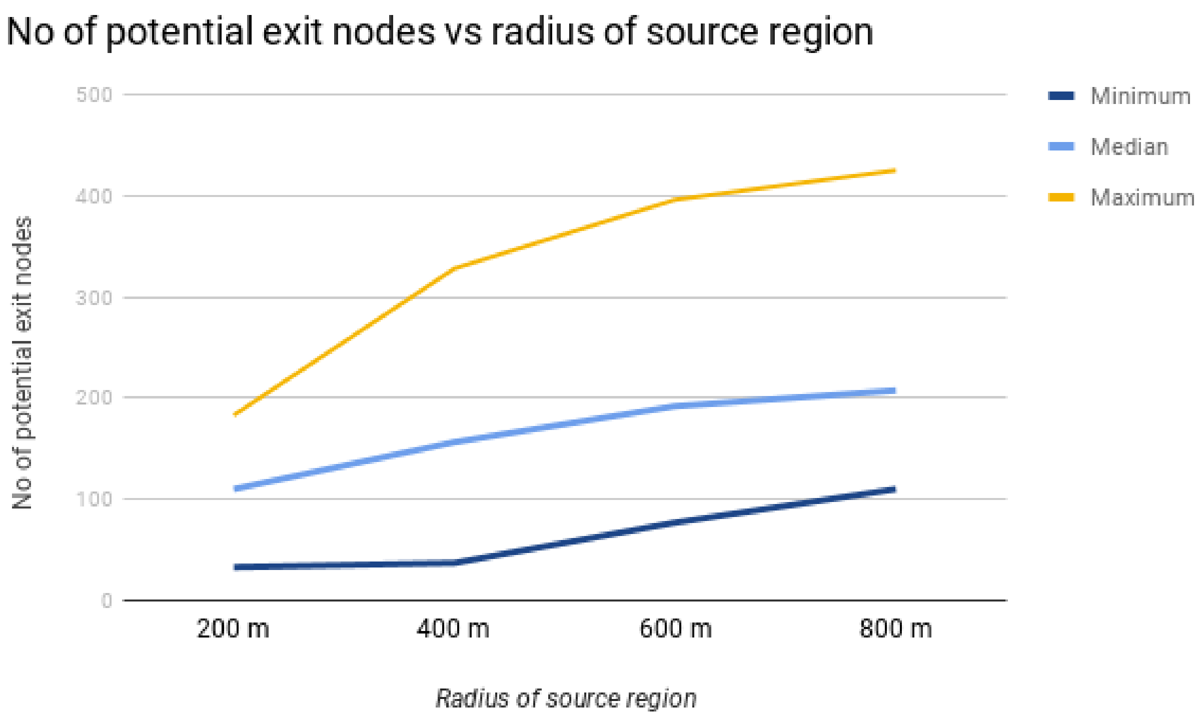
We considered four different values of R, ranging from 200m to 800m. For each value of R, $|S|$ is chosen such that the ratio $\frac{|S|}{R}$ (Density of source nodes in the source region) varies from 1 to 5 in steps of 1. Thus the total number of possible combinations of $|S|$ and R are 20 and we run every combination on 50 randomly generated instances of clustered MSP problems to get better accuracy. In total, 1000 total cases are considered.



The above graph shows the change in average runtime speedup with change in source density and source region radius. The average speed-up provided by our algorithm goes to a maximum of 3.16 when $frac|S|R = 5$. For a certain radius, the speedup increases with increase in source density. This shows that this algorithm is well suited for clustered MSP problems.



The above graph shows the speedup in terms of total number of SSP computations. It shows the same trend as the previous graph, when the source density is increased. But with increase in radius, the increase is not as apparent as the previous graph, because the SSP computation speedup is equal to $\frac{|S|}{(|P|+|X|)}$. Although the $|X|$ SSP computations do not significantly impact the runtime evaluation, but they cause the SSP speedup to be lower.



This graph shows the change in number of potential exit nodes versus the radius of source region. Based on algorithm analysis, for a constant value of $\frac{|S|}{R}$, we expect the speedup to remain constant, but this is not the case because $|P|$ and $|R|$ do not have a linear relationship. The median values of $|P|$ are lower than what they would be if the relationship was linear.

Chapter 7

Implementation

```
#include<bits/stdc++.h>
using namespace std;
#define int long long
#define pi M_PI
#define R 6371e3

vector<vector<pair<int, int>>> v; // basic graph (input)

vector<vector<pair<int, int>>> v1; // reverse graph (input)

vector<pair<double, double>> m; // node to coordinates (input)

vector<int> p; // potential exit nodes

vector<pair<pair<int, int>, int>> edges; // all edges (input)

int n=0; // number of nodes

int ne=0; // number of edges

int ssp_comp=0; // To keep count of no of ssp computations

int center;

vector<int> s; // sources

vector<int> ps; // potential sources

vector<int> r = {200,400,600,800}; // set of radius to check

unsigned seed = rand(); // random seed

vector<int> t; // target nodes

map<int, int> ma; // map for differentiating

vector<int> ext; // exit nodes

vector<vector<pair<int, int>>> v2; // exit nodes to target nodes

map<int, map<int, int>> ans;

int pe_size=0; // No of potential exit nodes (Required for evaluation)
```

```
int base_steps=0; // baseline steps

int distance(int c1, int c2)
{
   double x1=m[c1].first;
   double y1=m[c1].second;
   double x2=m[c2].first;
   double y2=m[c2].second;
   double p1=(x1*pi)/180.0;
   double p2=(x2*pi)/180.0;
   double d1=((x2-x1)*pi)/180.0;
   double d2=((y2-y1)*pi)/180.0;
   double a=sin(d1/2.0)*sin(d1/2.0)+cos(p1)*cos(p2)*sin(d2/2.0)*sin(d2/2);
   double c=2*atan2(sqrt(a), sqrt(1-a));
   return (int)(c*R);
}

int distance2(double x1, double y1, double x2, double y2) // Calculates distance between two points
    (Given their latitudes and longitudes)
{
   double p1=(x1*pi)/180.0;
   double p2=(x2*pi)/180.0;
   double d1=((x2-x1)*pi)/180.0;
   double d2=((y2-y1)*pi)/180.0;
   double a=sin(d1/2.0)*sin(d1/2.0)+cos(p1)*cos(p2)*sin(d2/2.0)*sin(d2/2);
   double c=2*atan2(sqrt(a), sqrt(1-a));
   return (int)(c*R);
}

void input() // initializing the graph
{
   ifstream in1;
   in1.open("nodes.txt"); // To store latitude and longitude of each node
   in1>>n;
   m.resize(n+1);
   v.resize(n+1);
   v1.resize(n+1);
   v2.resize(n+1);
   for(int i=0;i<n;i++)
   {
      double x, y;
      in1>>x>>y;
      m[i+1]={x, y};
   }
   in1.close();
   ifstream in2;
   in2.open("edges.txt"); // Contains information about edges
   in2>>ne;
   edges.resize(ne+1);
   for(int i=0;i<ne;i++)
   {
      int a, b;
      in2>>a>>b;
      int w=abs(distance(a, b));
      edges[i]={{a, b}, w};
      v[a].push_back({b, w});
      v[b].push_back({a, w});
      v1[a].push_back({b, w});
      v1[b].push_back({a, w});
   }
```

```
}

void reset(){
   p.clear();
   ps.clear();
   s.clear();
   t.clear();
   ma.clear();
   ext.clear();
   v2.clear();
   ans.clear();
   pe_size=0;
   ssp_comp=0;
   base_steps=0;
}

// Selects a random node as centre such that it contains atleast |S| nodes
void select_center_and_source(int r, int num) // radius, number of source nodes
{
   while(true)
   {
      int node=(int)((rand()%n)+1);

      int x=m[node].first;
      int y=m[node].second;
      for(int i=1;i<=n;i++)
      {
         int xa=m[node].first;
         int ya=m[node].second;
         if(distance(node, i)<=r)
         {
            ps.push_back(i);
         }
      }
      if(ps.size()<num)
      {
         ps.clear();
         continue;
      }
      shuffle(ps.begin(), ps.end(), default_random_engine(seed));
      for(int i=0;i<num;i++)
      {
         s.push_back(ps[i]);
         ma[s[i]]=1;
      }
      center = node;
      break;
   }
}

//Randomly selects |S| non-source nodes as target nodes
void select_targets(int num)
{
   map<int, int> m1;
   vector<int> rest;
   for(int i=0;i<(int)ps.size();i++)
   {
      m1[ps[i]]=1;
   }
   for(int i=1;i<=n;i++)
   {
```

```
      if(!m1[i])
         rest.push_back(i);
   }
   shuffle(rest.begin(), rest.end(), default_random_engine(seed));
   for(int i=0;i<num;i++)
   {
      t.push_back(rest[i]);
      ma[rest[i]]=2;
   }
}
// Forms a vector with potential exit nodes stored in it
void potential_ext_nodes(int r)
{
   for(int i=0;i<ne;i++)
   {
      int d1=distance(edges[i].first.first, center);
      int d2=distance(edges[i].first.second, center);
      if(d1>d2)
      {
         swap(d1, d2);
         swap(edges[i].first.first, edges[i].first.second);
      }
      if(d1<r && d2>r)
      {
         p.push_back(edges[i].first.first);
         if(ma[edges[i].first.first]==1)
            ma[edges[i].first.first]=13;
         else
            ma[edges[i].first.first]=3;
      }
   }
   pe_size=p.size();
}

// Calculates lpen function described in step 1
// Helps get the list of final exit nodes
void lpen_func()
{
   ssp_comp+=(int)p.size();
   for(int i=0;i<(int)p.size();i++)
   {
      int node=p[i];
      vector<int> dis(n+1, INT_MAX);
      dis[node]=0;
      vector<int> lpen(n+1, 0);
      vector<int> vis(n+1, 0);
      lpen[node]=node;
      int start=node;
      multiset<pair<int, int>> s;
      s.insert({0, node});
      int c=0;
      while(s.size()>0)
      {
         pair<int, int> p=*(s.begin());
         s.erase(s.begin());
         if(ma[p.second]==2)
            c++;
         if(c==(int)t.size())
            break;
         if(ma[p.second]==3 || ma[p.second]==13)
         {
```

```
            start=p.second;
         }
         lpen[p.second]=start;
         if(vis[p.second]==1)
            continue;
         vis[p.second]=1;
         for(int i=0;i<(int)(v[p.second].size());i++)
         {
            if(dis[p.second]+v[p.second][i].second<dis[v[p.second][i].first])
            {
               dis[v[p.second][i].first]=dis[p.second]+v[p.second][i].second;
               s.insert({dis[v[p.second][i].first], v[p.second][i].first});
            }
         }
      }
      int f=0;
      for(int i=0;i<(int)t.size();i++)
      {
         if(lpen[t[i]]==node)
         {
            if(f==0)
            {
               ext.push_back(node);
               f=1;
            }
            v2[node].push_back({t[i], dis[t[i]]});
         }
      }
   }
}

//Reverses the graph for step 2
void reverse()
{
   ssp_comp+=(int)ext.size();
   for(int i=0;i<(int)ext.size();i++)
   {
      int node=ext[i];
      vector<int> dis(n+1, INT_MAX);
      vector<bool> vis(n+1, 0);
      int c=0;
      multiset<pair<int, int>> s1;
      dis[node]=0;
      s1.insert({0, node});
      while(s1.size()>0)
      {
         pair<int, int> p=*(s1.begin());
         s1.erase(s1.begin());
         if(vis[p.second])
            continue;
         vis[p.second]=1;
         if(ma[p.second]==1 || ma[p.second]==13)
            c++;
         if(c==(int)s.size())
         {
            break;
         }
         for(int i=0;i<(int)v[p.second].size();i++)
         {
            if(dis[p.second]+v[p.second][i].second<dis[v[p.second][i].first])
            {
```

```
                dis[v[p.second][i].first]=dis[p.second]+v[p.second][i].second;
                s1.insert({dis[v[p.second][i].first], v[p.second][i].first});
            }
        }
    }
    for(int i=0;i<(int)s.size();i++)
    {
        v2[s[i]].push_back({node, dis[s[i]]});
    }
  }
}

//Step 3
void final_step()
{
  for(int i=0;i<(int)s.size();i++)
  {
    vector<int> dis(n+1, INT_MAX);
    vector<int> vis(n+1, INT_MAX);
    dis[s[i]]=0;
    vector<int> vx;
    vx.push_back(s[i]);
    while(vx.size()>0)
    {
      int node=vx[(int)vx.size()-1];
      vx.pop_back();
      if(vis[node])
        continue;
      vis[node]=1;
      for(int i=0;i<(int)v2[node].size();i++)
      {
        dis[v2[node][i].second]=min(dis[v2[node][i].second], dis[node]+v2[node][i].first);
        if(ma[v2[node][i].first]==3 || ma[v2[node][i].first]==13)
        {
          if(!vis[v2[node][i].first])
            vx.push_back(v2[node][i].first);
        }
      }
    }
    for(int i=0;i<(int)t.size();i++)
    {
      if(dis[t[i]]!=INT_MAX)
      {
        ans[s[i]][t[i]]=dis[t[i]];
      }
    }
  }
}

//Baseline algorithm which uses dijkstra to calculate distance between all pairs of source nodes and exit
    nodes
void baseline()
{
  for(int i=0;i<(int)s.size();i++)
  {
    vector<int> dis(n+1, INT_MAX);
    vector<int> vis(n+1, 0);
    int node=s[i];
    dis[node]=0;
    multiset<pair<int, int>> s1;
    s1.insert({0, node});
```

```
        int c=0;
        while(s1.size()>0)
        {
            pair<int, int> p=*(s1.begin());
            s1.erase(s1.begin());
            if(vis[p.second])
                continue;
            if(ma[p.second]==2)
                c++;
            if(c==(int)t.size())
                break;
            vis[p.second]=1;
            for(int i=0;i<(int)v[p.second].size();i++)
            {
                if(dis[p.second]+v[p.second][i].second<dis[v[p.second][i].first])
                {
                    dis[v[p.second][i].first]=dis[p.second]+v[p.second][i].second;
                    s1.insert({dis[v[p.second][i].first], v[p.second][i].first});
                }
            }
        }
    }
    base_steps=(int)s.size();
}

//Evaluation process
//Stores evaluation data in respective files
void evaluate_and_run(){

    ofstream fout,fout1;
    fout.open( "evaluation_data.txt");
    fout1.open( "potential_exit_nodes.txt" );

    for(int x=0;x<(int)r.size();x++)
    {
        fout<<"radius : "<<r[x]<<endl;
        fout1<<"radius : "<<r[x]<<endl;

        for(int y=1;y<=5;y+=1)
        {
            int num=(r[x]*y);
            double avg_runspeedup=0,avg_ssp_speedup=0;
            for(int itr=1;itr<=50;itr++)
            {
                reset();
                input();

                clock_t start,end;
                double time1,time2;
                start=clock();

                select_center_and_source(r[x],num);
                select_targets(num);
                potential_ext_nodes(r[x]);
                lpen_func();
                reverse();
                final_step();

                end=clock();
                time1=double(end-start)/double(CLOCKS_PER_SEC);

```

```
                start=clock();

                baseline();

                end=clock();
                time2=double(end-start)/double(CLOCKS_PER_SEC);
                avg_runspeedup+=time2/time1;
                avg_ssp_speedup+=(base_steps*1.0)/ssp_comp;
                fout1<<pe_size<<" ";
            }
            avg_runspeedup/=50;
            avg_ssp_speedup/=50;
            fout<<"|S|/R: "<<y<<" avg_runspeedup: "<<avg_runspeedup<<" avg_ssp_speedup
                "<<avg_ssp_speedup<<endl;
        }
        fout<<endl;
        fout1<<endl;
    }
    fout.close();
    fout1.close();
}

int32_t main()
{
    evaluate_and_run();
}
```

Chapter 8

Conclusion

We have designed an efficient approach for computing shortest paths from a clustered set of source nodes to a set of target nodes. This approach involves three basic steps which involve identifying exit nodes, computing SSP from exit nodes to target nodes and computing SSP from source nodes to exit nodes which are then combined to find the required shortest paths from each source node to target nodes. The major advantage of this approach is that it can be applied to networks with dynamically changing connectivity and edge lengths.

This algorithm achieves a significant improvement in runtime over the baseline algorithm. The speed-up increases with increase in both source density and radius of the source region.

The approach can be further optimized by using speed-up techniques such as goal-directed approach along with it. Further optimization can be done by applying various heuristics limiting the selection of potential exit nodes in the first step.

Chapter 9

References

- R. Jagadeesh and Thambipillai Srikantha. *Fast computation of clustered Many-to-many Shortest Paths and Its Application to Map Matching.*
- Mo Chen, Rezaul A. Chowdhury, Vijaya Ramachandran, David L. Roche, and Lingling Tong. 2007. *Priority Queues and Dijkstra's Algorithm*
- Tetsuo Shibuya. 2000. *Computing the nxm shortest path efficiently.*