ABSTRACT

A NEW IMPLEMENTATION OF CLUSTERING ALGORITHM AND ITS APPLICATION IN NET-TREE CONSTRUCTION ALGORITHM

Yan Dai

In a vertex $k$-center problem, the goal is to pick some vertexes, called centers, from a given undirected weighted graph so as to minimize the maximum distance of any vertex from its closest center. It is known that not only is $k$-center an NP-complete problem, but approximating the $k$-center problem within a factor better than 2 is still NP-complete [2]. If the distance between any two vertices is given as a distance matrix (so distance computations take constant time), the 2-approximation $k$-center problem can be solved in $O(n \cdot \log k)$ time [2]. But to build such a distance matrix for a large-size road map is impractical because even storing the distance matrix is impractical, because it requires $n^2$ space. In this paper, I propose a new algorithm that gives a 2-approximation to the vertex $k$-center problem in $O(\lambda^2 \cdot \log n \cdot \log \phi)$ time in road maps, where $\lambda$ is the doubling constant and $\phi$ is the spread of the graph, without being given the distance matrix in advance.
A NEW IMPLEMENTATION OF CLUSTERING ALGORITHM AND ITS APPLICATION IN NET-TREE CONSTRUCTION ALGORITHM

A Thesis

Submitted to the
Faculty of Miami University
In partial fulfillment of
The requirements for the degree of
Master of Computer Science
Department of Computer Science
By
Yan Dai
Miami University
Oxford, Ohio
2009

Advisor Dr. Bo Brinkman

Reader Dr. Alton Sanders

Reader Dr. Keith Frikken
Contents
1. Introduction ...................................................................................................................................... 1
   1.1 Motivation .......................................................................................................................... 1
   1.2 Related work on the k-center problem ...................................................................... 1
   1.3 Definitions ..................................................................................................................... 3
   1.4 My Results .................................................................................................................... 5
2. My algorithm ................................................................................................................................... 5
   2.1 The main algorithm ...................................................................................................... 5
   2.2 How to efficiently pick centers .................................................................................. 7
   2.3 Running Time Bounds ............................................................................................... 8
3. Implementation ........................................................................................................................... 10
   3.1 Programs ..................................................................................................................... 10
   3.2 Graph Representation ............................................................................................... 12
   3.3 Binary Max-Heap ..................................................................................................... 12
   3.4 Dijkstra’s Algorithm ................................................................................................. 12
4. Experiments ................................................................................................................................ 13
   4.1 Environment and Map Data .................................................................................... 13
   4.2 Testing our K-center algorithm on road maps ......................................................... 16
   4.3 Determining the doubling dimension for road maps .............................................. 17
   4.4 Conclusion of the experiments ................................................................................ 21
5. Future research directions ......................................................................................................... 22
6. Conclusion ................................................................................................................................... 23
References ......................................................................................................................................... 23
Appendix A: How to export map data from StreetMap USA .................................................... 24
List of Tables

Table 4.1 Maps used in experiments .................................................................6

Table 4.2.1 Experimental results of running times when K = |V| ......................17

Table 4.3.2 .....................................................................................................21
List of Figures

Code 1.2 Outline of Gonzalez’s algorithm ................................................................. 2

Code 2.1: outline of my algorithm ............................................................................. 6

Example 2.1: Proof of correctness of pruning .......................................................... 8

Code 2.2: Detailed pseudo code of our algorithm ..................................................... 8

Code 2.3: Detailed pseudo code of Update_D_and_H(c,D,H) .................................... 8

Figure 4.1.1 ............................................................................................................. 16

Plot 4.3 running time comparison among different K values in USA road maps. The X-axis is in logarithmic scale (base = 10) ........................................................................................................ 17

Figure 4.3.1 The tree on the left side is a net-tree of the map on the right side .......... 19
1. Introduction

1.1 Motivation

The $k$-center problem is a classical and well-known problem of computational geometry, but it is not well solved in road maps. The well-known algorithms for the $k$-center problem have to know the distance matrix beforehand, which would require $n^2$ space (as we note below, this is quite impractical for real maps). The other naïve approach would be to execute a shortest-path query using Dijkstra’s algorithm each time a distance needs to be computed, but this leads to an unsatisfactory running time of $O(n^2 \cdot \log n \cdot \log k)$.

Secondly, it is known that, in metrics with bounded doubling dimension, many traditional graph problems (for example, shortest paths and network routing) can be solved very efficiently. But it is not known whether all road maps in the world have bounded doubling dimension. We can demonstrate, experimentally, that real road maps have bounded doubling dimension, leading to many possibilities for beating the naïve approaches for $k$-center (as well as other problems). One way to calculate the doubling dimension of a graph is to use Sariel Har-Peled and Manor Mendel’s algorithm to build a net-tree for the graph. In their algorithms, $n$ centers need to be computed first. As we mentioned above, it is impractical to use any known algorithms to calculate $n$ centers in large-size real road maps. So, we want to find a new algorithm which not only can solve $k$-center in road maps very efficiently but also does not rely on the distance matrix.

1.2 Related work on the $k$-center problem

The $k$-center problem is a well-known problem of picking $k$ vertexes as the centers in a finite metric space\(^1\) so as to minimize the maximum distance of any vertex from its

---

\(^1\) In general, I use graph terminology instead of metric terminology because we are interested in road maps, which are represented as graphs. For any graph with weighted edges, let the distance between any two vertices be given by the length of the shortest path between them. The result is always a finite metric space. Also note that, by using the complete graph, every finite metric space can be represented as a graph. As a result, I ignore the distinction between graphs and finite metric spaces.
closest center. It can be proved that both $k$-center and its $\alpha$-approximation ($\alpha<2$) problems are NP-complete by reducing Dominating Set [6] to it. So, unless $P=NP$, we could only approximate the $k$-center problem within a factor larger than 2 in polynomial time.

There are several algorithms solving the 2-approximate $k$-center problem in poly-time. Gonzalez [3] achieved a 2-approximation algorithm in $O(kn)$, which was improved by Feder and Greene [2] to $O(n \log k)$ and finally improved to $O(n + k \log k)$ by Har-Peled [4].

The basic idea behind their 2-approximation algorithms follows Gonzalez’s clustering algorithm and can be generalized as follows:

\[
\text{Let } G= \langle V, E \rangle \text{ be the given graph.} \\
\text{Let } C \text{ be a collection of centers.} \\
\text{Initially, pick an arbitrary vertex to be the first center and put it into } C. \\
\text{For } i=2 \text{ to } k \\
\text{Pick the farthest vertex } v \text{ from } C \text{ in } V - C \text{ and set } C = C + \{v\}.
\]

Code 1.2: outline of Gonzalez’ algorithm

As stated above, however, there is a cost associated with “picking the farthest vertex.” We cannot pre-compute the distance matrix, because the storage space required is too large. For example, the US road map consists of more than 26,000,000 vertexes, requiring at least 78697GB to store the distance matrix – even if only one bit is needed per vertex pair.

The new algorithm I designed for $k$-center still follows the same strategy of picking vertexes as Gonzalez’s algorithm, but uses different ways to keep and process the intermediary data so that given a weighted undirected graph, without knowing the distance matrix, it can solve 2-approximation $k$-center in $O(\lambda^2 n \log n \log \phi)$, where $n$ is the total number of vertexes, and $\lambda$ and $\phi$ are values that depend on the geometry of the graph (and are, as we will see, not too large). Please note that this running time is the upper bound for generating all centers in a graph (that is, $k=n$). It is not particularly helpful to derive the dependence of the running time on $k$, because when $k=1$ the running time is still roughly $O(n \log n)$, because my approach is based on Dijkstra’s algorithm. My running time as stated, however, is much better than the $n^2 \log n$ that
would result if Dijkstra’s algorithm was run every time we needed the distance between two points.

I begin with some definitions in Section 1.3. In Section 2, I present my algorithm with proof of the running time and correctness. In Section 3, I introduce an implementation of the algorithm used in experiments. In Section 4, I present the experiments on both $k$-center problem and net-tree construction.

1.3 Definitions
Most of the following definitions are standard, and are only presented for completeness. Those familiar with graph and metric space algorithms can safely skip this section.

1.3.1 Graphs
A graph $G$ consists of a set of vertices $V$, and a set of edges $E$ whose endpoints are the elements of $V$. Each edge $e_i \in E$ is associated with a weight $w_i$. In map problems, the weight of an edge is usually measured as the length of the road connecting its two endpoints (or the travel time between the points).

1.3.2 Metric
A metric is a function $d: X \times X \rightarrow \{0\} \cup R^+$ such that

1. $\forall x, y \in X, d(x, y) = d(y, x)$
2. $\forall x, y \in X, d(x, y) = 0$ if and only if $x = y$
3. $\forall x, y, z \in X, d(x, y) + d(y, z) \geq d(x, z)$

1.3.3 Finite Metric Space
A finite metric space $M$ is a pair $<X, d>$ where $X$ is a finite set and $d$ is the corresponding metric distance function. [5]

1.3.4 Distance to a set
We may sometimes write $d(p, Y)$ or $d(X, Y)$ where $X$ and $Y$ are sets, not single vertices. As is standard in this area, we define:
\[ d(p, C) = \min_{q \in C} d(p, q) \]

and

\[ d(X, Y) = \min_{(x, y) \in X \times Y} d(x, y) \]

1.3.5 K-center problem
Input: A finite metric space \( M \), and an integer \( K \)
Output: a collection of centers \( C \subseteq V \) such that:

1. \( |C| = K \), and
2. \( \max_{p \in V} d(p, C) \) is minimized

1.3.6 Map metric
In a graph that represents a road map, we define \( d(x, y) \) to be the length of the shortest path between vertices \( x \) and \( y \). It is easy to see that, in an undirected graph, this is a metric.

1.3.7 Ball
A ball \( Ball(x, r) \) in a metric space is the set of points \( P = \{ p \in V | d(x, p) \leq r \} \).

1.3.8 Doubling metric / doubling number (\( \lambda \))
A metric space \( M \) has doubling constant \( \lambda \) if any ball in \( M \) can be covered by at most \( \lambda \) balls of half the radius. Formally, we have: \( \lambda \) is the smallest integer so that for any \( x \in V \), \( r \in \mathbb{R}^+ \), there exists a set \( Y \subseteq V \) with \( |Y| \leq \lambda \) such that \( Ball(x, r) \supseteq \bigcup_{y \in Y} Ball(y, r/2) \).

1.3.9 Doubling dimension
The doubling dimension of a metric space \( M \) is given by \( \dim(M) = \log_2 \lambda \).

1.3.10 Aspect ratio or Spread (\( \phi \))
Let \( P \) be a finite metric space. The spread \( \phi \) of \( P \) is given by \( \phi = \max_{s, t \in P} d(s, t) / \min_{u, v \in P} d(u, v) \). Note that this is finite by the assumption that \( P \) is both finite and a metric, so there are no 0 (or infinitesimal) distance.
1.3.11 Packing Property
If a metric space $M$ has doubling constant $\lambda$, any ball in $M$ can cover at most $\lambda^2$ elements that are at a distance of at least half the radius.

1.3.12 Covering Property
If a metric space $M$ has doubling constant $\lambda$, any ball in $M$ can be covered by at most $\lambda$ balls of half the radius.

1.4 My Results
I have three main results:

1. I give a 2-approximation algorithm for k-center that runs in $O(\lambda^2 \cdot n \cdot \log n \cdot \log \varphi)$ time in road maps, where $\lambda$ is the doubling constant of the map metric, and $\varphi$ is the aspect ratio.

2. I demonstrate that the algorithm is efficient in practice.

3. Using my k-center algorithm, I experimentally determine upper bounds of $\lambda$ for a variety of widely used road maps.

The third result suggests that algorithms designed for metrics with bounded doubling dimension may work well, in practice, on road maps.

2. My algorithm

2.1 The main algorithm
I follow the same strategy as Gonzalez’s algorithm to pick centers one by one. That is, my algorithm picks an arbitrary vertex as the first center and puts it into the center collection $C$, and then iteratively picks the farthest vertex $p$ from $C$ to be the new center and updates $C = C + \{ p \}$. This algorithm terminates when $k$ centers are picked.
The following pseudo code presents the basic logic of my algorithm:

```
Input: a weighted undirected graph G<V,E> and an integer k
Output: a collection of centers C
Let C_k be the collection of centers at the end of the kth iteration
Let D be the distance vector, where D_x = d(x,C_k)
Line 1:  C_1 = p;  //p is an arbitrary vertex in V
Line 2:  For each x Є V
Line 3:   D_x = d(x, C_1); // use Dijkstra’s algorithm
Line 4:  For each k = 2,…,K
Line 5:   C_k = C_{k-1} + {c}; // add the new center to the collection
Line 6:   For each x Є V // in order to be efficient, need to be careful here. See the detailed version of the code below
Line 7:   D_x = Min( D_x, d(x, {c}) )
```

In line 1, an arbitrary vertex p is picked as the first center.

Line 2 and 3 actually calculate the shortest path from p to any other vertexes in V. It can be implemented by Dijkstra’s single-source shortest path algorithm [1] which requires \( O(|E|\log|V|) \) time. Because all nodes in a real road map have constant degree, the running time of Dijkstra’s algorithm can also be written as \( O(c|V|\log|V|) = O(|V|\log|V|) \), where c is the maximum degree of the map.

From line 4, the algorithm begins to generate k-1 centers.

In line 5 and 6, the new center is added to the collection by picking the farthest vertex from \( C_{k-1} \). This step requires Dijkstra search and an auxiliary max-heap so that the farthest vertex (the largest element in D) can be found quickly. But there are two problems with this step. First, Dijkstra search runs in \( O(n \log n) \) time. So, to find k centers, we have to run k times Dijkstra search. It’s impractical to do this in larger maps when k is very big. I will introduce a pruning algorithm in next section which can greatly improve performance. Another is the synchronization problem of the auxiliary max-heap and vector D, because some items in D become smaller at the end of iteration in line 8. I will introduce a solution to this issue in next section.

The last two lines update the vector D after a new center being added and the updated vector D will be used to find the new center in next round.
2.2 How to efficiently pick centers

My algorithm suggests the following approach for picking centers.

First, we maintain an auxiliary max-heap, called $H$, for vector $D$ so that the new center can be selected in $O(\log n)$ time at each round. As mentioned in last section, there is a synchronization problem between $H$ and $D$. I use the following approach to solve this problem. First, I create a position mapping from vector $D$ to max-heap $H$. Then, when $D_x$ is updated, use the mapping table to access $x$’s value in the max-heap (in constant time) and update its position in the heap (in logarithmic time).

Secondly, to update vector $D$ at each round, I apply Dijkstra’s algorithm to calculate the shortest distance from the new center $c$ to any other vertexes at each round (See section 3.4 for details of real implementation). But not every element in $D$ need to be expanded and updated every time in Dijkstra search. That is, vertex $x$ will be expanded and $D_x$ will be updated if and only if $c$ becomes the new center of $x$. More specifically, suppose that we are calling Dijkstra’s algorithm to calculate the shortest distance from the new center $c$ to any other vertexes in the graph, and we are going to expand a particular vertex, say $x$. If $d(x,\{c\}) < D_x$ ($x$ is dominated by the new center $c$), then we must update the value of $D_x$ and expand all $x$’s incident vertexes. If $d(x,\{c\}) \geq D_x$ ($x$ is not dominated by the new center $c$), then we keep the old value for $D_x$, and prune the current branch of the search tree immediately. This pruning is very crucial to our algorithm and enhances its performance greatly. Let’s see an example:

**Without loss of generality, let’s suppose that we’ve picked two centers, $v_1$ and $v_3$ into the collection $C$ and $v_3$ is the last center we picked. Now, we are running Dijkstra’s algorithm to calculate the shortest path from $v_3$ to all the other vertexes in order to update the distance vector $D$, and currently we are in $v_2$. We will prune the current branch of the breadth first search tree if and only if $D_{v_2} \leq d(v_2,v_3)$, because if $D_{v_2} \leq d(v_2,v_3)$, distance from $v_3$ to all $v_2$’s incident nodes($v_4$ and $v_5$) within the current branch of search tree will always be greater than the distance from some center in $C$, say $v_1$, to all $v_2$’s incident nodes. In other words, $v_3$ will never become the center of $v_4$ and $v_5$ and theirs values in $D$ will not be changed in the current branch.**
Example 2.1: Proof of correctness of pruning

Now, let me present detailed pseudo-code for the algorithm.

```
//Let C_k be the collection of centers at the end of the k’th iteration
//Let D be the distance vector, where D_x = d(x, C_k)
//Let H be a max-heap containing all elements in D
Line 1: C_1 = p; //p is an arbitrary vertex in V
Line 2: For each x ∈ V // Line 2 and 3 actually can be implemented by Dijkstra’s SSP Algorithm
Line 3: D_x = d(x, C_1);
Line 4: H = Make_Heap(D);
Line 5: For each k = 2, ..., K
Line 6: c = H.Top.Key; // pick the farthest vertex as the new center.
Line 7: C_k = C_{k-1} + {c}; // add the new center to the collection
Line 8: Update_D_and_H(c, D, H); // Dijkstra’s SSP with pruning. See Code 2.3 for details.
```

```
//Let Q be the priority queue used by Dijkstra’s algorithm
// Let M be the position mapping vector of D and Q.
Push c into Q.
while(!Q.empty())
{
    let currentNode be the top element in Q;
    pop the top element of Q;
    if(d(currentNode,c)<D[currentNode]) // c is the new center of currentNode
    {
        D[currentNode] = d(currentNode,c);
        Let i = M[currentNode] //to locate currentNode’s position in H
        Decrease the key of H[i] and update all the elements in M changed in H.
        Expand currentNode’s incidences by Dijkstra’s mechanism
    }
}
```

In order to bound the running time of this algorithm, I focus on counting the running time of Line 8 in Code 2.2.

### 2.3 Running Time Bounds

The following lemma and proof are folklore.

**Lemma 1:** Gonzalez’s algorithm can be divided into $\log(\varphi)$ phases such that at most $\lambda^2$ centers are picked in each phase.
Proof: First, let’s normalize every edge’s distance by dividing by the length of the shortest edge in the map. That is, after the normalization, the length of the shortest edge is 1 and the distance between two farthest vertexes is $\phi$. Now, let $r_k$ be the maximum distance of any non-center nodes to the set of centers when picking the $k+1^{th}$ center. Now, let us divide his algorithm into phases, where phase $i$ contains all steps of picking the $k^{th}$ center such that $r_i/2^i \geq r_k > r_{i+1}/2^{i+1}$. Since $r_k \geq 1$, the total number of phases is $\log(r_1)$.

Moreover, $r_1$ is always less than or equal to the aspect ratio $\phi$, so the total number of phases should be $\log(\phi)$ at most. According to the packing property (see definition 1.3.11) for metrics with bounded doubling, there are at most $\lambda^2$ centers (at a distance of at least $r_i/2^{i+1}$ from each other) picked in phase $i$.

Lemma 2: The total number of updates of all vertexes’ values in $D$ (in line 8) is bounded by $O(|V|\lambda^2 \cdot \log \phi)$ overall.

Proof: Let’s consider a particular vertex $x$ in the map. According to my algorithm, $D_x$ only gets updated when $x$ changes allegiance. By Lemma 1, there are at most $\lambda^2$ picked in each phase. So, $D_x$ gets updated at most $\lambda^2$ times in each phase. Since there are $|V|$ vertexes and $\log \phi$ phases, the total number of updates occurring in $D$ is bounded by $O(|V|\lambda^2 \cdot \log \phi)$

Lemma 3: In line 8, expanding a single vertex takes $O(\log |V|)$ time.

Proof: Updating a single vertex requires changing its key and fixing its places in the heap $H$, which takes $O(\log |V|)$ time. We then must add all children of the vertex to the min-heap used by Dijkstra’s algorithm, or update their keys as appropriate. Because the degree in road maps is constant (certainly less than 16), this also takes $O(\log |V|)$ time.

Theorem 1: The total running time of our algorithm is $O(|V|\lambda^2 \cdot \log \phi \cdot \log |V|)$

Proof: This is immediate from Lemmas 2 and 3.
**Theorem 2:** The total space used by our algorithm is $O(|V|)$.

**Proof:** Our algorithm requires the following data structures:

1. Vector $D$
2. Max-Heap $H$
3. Mapping vector
4. All data structures used by Dijsktra’s SSP algorithm.

All these data structures are linear.

### 3. Implementation

In order to get accurate experimental results of our algorithm’s running time, I implemented the algorithm following these rules:

1. Choose C++ as the programming language. Because C++ doesn’t rely on any virtual environment or framework which would affect running time of our program.
2. Do NOT use any libraries except for the C++ Standard Template Library.
3. Do NOT use Object Oriented Programming

#### 3.1 Programs

Besides implementing our algorithm, I also created some auxiliary programs for the experiment. Followings list all programs I created and their usages:

**3.1.1 K-Center Solver**
Usage: Given a graph and an integer K, it returns a 2-approximation solution for K-center problem.

Input: a graph G\langle V,E\rangle, an integer K

Output: an array of K centers

### 3.1.2 Net-Tree Constructor

Usage: It constructs a Net-Tree for the given graph.

Input: a permutation of vertexes.

Output: a Net-Tree of G

### 3.1.3 Map Data Convertor

Usage: It Converts road map data from ArcGIS Network Dataset to our map data format (Please see 3.2)

Input: ArcGIS Network Dataset

Output: a graph G\langle V,E\rangle represented in adjacency lists.

Note: This program is a plug-in of ArcMap.

### 3.1.4 Net-Tree Loader

Usage: It counts the max number of children of each level in Net-Tree

Input: a net-tree

Output: an array of numbers
3.2 Graph Representation

I use *adjacency list* to store graphs in the program. *Adjacency list* is such a graph representation data structure that keeps all incident vertexes in a list for each vertex. In this project, using *adjacency list* to represent graphs has the following advantages:

1. Since most real road maps are large sparse graphs, using *adjacency list* to represent sparse graphs occupies less space.

2. It’s simple and efficient to traverse a graph which is represented by *adjacency list*.

3.3 Binary Max-Heap

The binary max-heap is implemented by a single array where the first element is always the largest element and the $i^{th}$ non-leaf element’s two children are the $2i^{th}$ and $(2i+1)^{th}$ elements. Its elements consist of a key and an associated attribute. The associated attribute specifies the ID of the vertex. Priority of elements is determined by the key which is the length from the vertex to its nearest center.

The only heap operations used in my algorithm are “*DecreaseKey*” and “*FindMax*”. The *DecreaseKey* operation decreases an element’s key value and rearranges the structure of the max-heap by iteratively swapping elements with its child whose key is greater than the new decreased key value. Its running time is $O(\log |V|)$. *FindMax* works in constant time by just looking at the $0^{th}$ entry of the array.

3.4 Dijkstra's Algorithm

As I mentioned in section 2.2, the pruning algorithm works with Dijkstra’s single-source shortest path algorithm. So I combined the pruning with Dijkstra’s algorithm together into one method called *DijkstraAlgorithmWithPruning()* . This method was implemented by priority queue. It is exactly the same as Dijkstra’s algorithm except the followings:

1. It needs three more variables:

   a. *globalDistanceVector* keeps the distance from each vertex to its closest center. The pruning algorithm works based on the value in this vector.
b. globalDistanceHeap is globalDistanceVector’s data in the max-heap. It must be updated together with the globalDistanceVector to guarantee data consistency.

c. mapVectorToHeap maintains a position mapping from globalDistanceVector’s elements to globalDistanceHeap’s elements. Using this mapping vector, we can find the corresponding element in the heap in \( O(1) \) time. Updating this vector also takes \( O(1) \) time. It also must be updated together with the globalDistanceVector and globalDistanceHeap to guarantee data consistency.

2. Dijkstra’s Algorithm always expands the closest vertex in the heap, whereas DijkstraAlgorithmWithPruning will check the pruning condition before expanding the vertex.

Please see Appendix E for the C++ implementation of DijkstraAlgorithmWithPruning.

4. Experiments

I divided the experiment into two parts. First, I tested this new \( K \)-center implementation on extensive real road maps. Secondly, I determined upper bounds of doubling dimension for those maps by applying Har-Peled and Mendel’s Net-Tree construction algorithms.

4.1 Environment and Map Data

4.1.1 Environment.

All experiments were done on a 64-bit machine with 2.5G memory and 2.0 GHz AMD Athlon(tm) 64 X2 Dual Core Processor 3800+. Since I did NOT use any multithreading, only one of the dual core processor was used in the experiments. All programs were compiled by Microsoft C++ 8.0 and run on Windows XP Professional with SP2.
4.1.2 Map Data.

All map data used in the experiments were extracted from “ESRI Data & Maps and StreetMap USA 2006 CD”. (Please See Appendix A to know how to export the USA street map). I extracted the whole USA road map and 18 road maps of different states and regions from Street Map USA 2006 and all experiments were done on these 19 road maps.

<table>
<thead>
<tr>
<th>Map Name</th>
<th># of vertexes</th>
<th># of edges</th>
<th>$\log_2(\phi)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>2693742</td>
<td>33409995</td>
<td>35.4447</td>
</tr>
<tr>
<td>San Francisco</td>
<td>35350</td>
<td>50095</td>
<td>15.0838</td>
</tr>
<tr>
<td>Canada</td>
<td>1326235</td>
<td>1648134</td>
<td>20.9657</td>
</tr>
<tr>
<td>California, USA</td>
<td>2257072</td>
<td>1854048</td>
<td>21.6192</td>
</tr>
<tr>
<td>Texas, USA</td>
<td>2246739</td>
<td>2823472</td>
<td>23.2171</td>
</tr>
<tr>
<td>Florida, USA</td>
<td>1067161</td>
<td>1355309</td>
<td>21.1373</td>
</tr>
<tr>
<td>Illinois, USA</td>
<td>854250</td>
<td>1089509</td>
<td>23.7206</td>
</tr>
<tr>
<td>Ohio, USA</td>
<td>761443</td>
<td>946122</td>
<td>19.735</td>
</tr>
<tr>
<td>Tennessee, USA</td>
<td>679409</td>
<td>785969</td>
<td>20.6646</td>
</tr>
<tr>
<td>Arizona, USA</td>
<td>601550</td>
<td>740309</td>
<td>20.1876</td>
</tr>
<tr>
<td>Washington, USA</td>
<td>615330</td>
<td>723643</td>
<td>21.7601</td>
</tr>
<tr>
<td>Oregon, USA</td>
<td>592180</td>
<td>692860</td>
<td>20.6117</td>
</tr>
<tr>
<td>New Mexico, USA</td>
<td>524164</td>
<td>643485</td>
<td>19.5673</td>
</tr>
<tr>
<td>New Jersey, USA</td>
<td>341480</td>
<td>438852</td>
<td>19.6749</td>
</tr>
<tr>
<td>Massachusetts, USA</td>
<td>316935</td>
<td>392568</td>
<td>21.1021</td>
</tr>
<tr>
<td>Nevada, USA</td>
<td>286925</td>
<td>342768</td>
<td>19.9952</td>
</tr>
<tr>
<td>Connecticut, USA</td>
<td>161256</td>
<td>198347</td>
<td>20.224</td>
</tr>
<tr>
<td>Alaska, USA</td>
<td>57887</td>
<td>35330</td>
<td>21.0596</td>
</tr>
<tr>
<td>Oxford, OH, USA</td>
<td>4176</td>
<td>5029</td>
<td>15.5596</td>
</tr>
</tbody>
</table>

Table 4.1 information of maps used in experiments

4.1.3 Calculating the spread $\phi$.

To calculate the spread, we must know the distance between two farthest nodes and the distance between two nearest nodes. The latter can be retrieved by scanning all edges in linear time. The former cannot be easily retrieved, so I calculate its 2-approximation value using the following simply algorithm:
**Step 1:** pick an arbitrary vertex $p$

**Step 2:** find the farthest vertex $q$ from $p$, and return $2d(p,q)$ which is a 2-approximation value of the distance between two farthest vertexes.

**Proof:** Suppose $a$ is the distance from $p$ to its farthest point and $L$ is the distance between two farthest points, say $s$ and $t$, in the map(Figure 4.1.1). Now we want to prove the following inequality:

$$L \leq 2a \leq 2L$$

Apparently, we have $a \leq L$. (1)

Because $a$ is the distance from $p$ to its farthest vertex, we have $b+c \leq a+a = 2a$. (2)

Because $L$ is the shortest path between $s$ and $t$, we have $L \leq b+c$. (3)

From (1),(2) and (3), we have $L \leq 2a \leq 2L$

So the logarithm of the spread I calculated in table 4.1 is off by at most +1.

4.1.4 Map Formats.

Various data vendors use various formats to present maps, so map data must be converted into our map format before being processed. I created a program, called Map Data Convertor (See Section 3.2.3), converting map data from ESRI Shapefile Network Dataset to our map format. (Please See Appendix B to know how to convert map data from ESRI Shapefile Network Dataset to our map format)
If you want to duplicate our experiments, please follow the instructions introduced in Appendix C.

4.2 Testing our K-center algorithm on road maps

Table 4.2.1 shows running times of generating $|V|$ centers for different road maps. In order to show that the algorithm’s complexity is close to $O(|V| \log |V|)$, I normalized the results based on the running time of generating the first center, because the first center is generated by Dijkstra’s shortest path algorithm which runs in $O(|E| \log |V|)$. The results were also normalized by directly dividing $|V| \log |V|$.

<table>
<thead>
<tr>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Name</td>
<td>$</td>
<td>V</td>
<td>$</td>
<td>Vertex updates</td>
<td>total running time (seconds)</td>
<td>Running time of finding 1st center (seconds)</td>
</tr>
<tr>
<td>USA</td>
<td>26,662,443</td>
<td>550,376,014</td>
<td>3301.33</td>
<td>278.125</td>
<td>11.86995056</td>
<td>0.000016674</td>
</tr>
<tr>
<td>San Francisco</td>
<td>35,350</td>
<td>405,505</td>
<td>1.703</td>
<td>0.234</td>
<td>7.277777778</td>
<td>0.000010592</td>
</tr>
<tr>
<td>Canada</td>
<td>1,326,235</td>
<td>25,125,361</td>
<td>126.047</td>
<td>9.703</td>
<td>12.9905184</td>
<td>0.000015523</td>
</tr>
<tr>
<td>CA, USA</td>
<td>2,257,072</td>
<td>38,352,084</td>
<td>192.032</td>
<td>17</td>
<td>11.296</td>
<td>0.000013391</td>
</tr>
<tr>
<td>TX, USA</td>
<td>2,246,739</td>
<td>36,502,298</td>
<td>185.344</td>
<td>17.844</td>
<td>10.38690876</td>
<td>0.000012988</td>
</tr>
<tr>
<td>FL, USA</td>
<td>1,067,161</td>
<td>16,303,642</td>
<td>77.078</td>
<td>7.921</td>
<td>9.730842065</td>
<td>0.000011981</td>
</tr>
<tr>
<td>IL, USA</td>
<td>854,250</td>
<td>12,279,152</td>
<td>59.078</td>
<td>6.734</td>
<td>8.73372119</td>
<td>0.000011659</td>
</tr>
<tr>
<td>OH, USA</td>
<td>761,443</td>
<td>10,335,316</td>
<td>49.203</td>
<td>6.078</td>
<td>8.095261599</td>
<td>0.000010986</td>
</tr>
<tr>
<td>TN, USA</td>
<td>679,409</td>
<td>9,858,184</td>
<td>46</td>
<td>4.891</td>
<td>9.405029646</td>
<td>0.000011609</td>
</tr>
<tr>
<td>AZ, USA</td>
<td>601,550</td>
<td>8,587,740</td>
<td>39.031</td>
<td>4.469</td>
<td>8.73372119</td>
<td>0.000011227</td>
</tr>
<tr>
<td>WA, USA</td>
<td>615,330</td>
<td>8,741,481</td>
<td>39.266</td>
<td>4.359</td>
<td>9.00820935</td>
<td>0.00001023</td>
</tr>
<tr>
<td>OR, USA</td>
<td>592,180</td>
<td>8,486,946</td>
<td>38.531</td>
<td>4.25</td>
<td>9.066117647</td>
<td>0.000011272</td>
</tr>
<tr>
<td>NM, USA</td>
<td>524,164</td>
<td>7,721,244</td>
<td>35.39</td>
<td>3.89</td>
<td>9.097686375</td>
<td>0.000011805</td>
</tr>
<tr>
<td>NJ, USA</td>
<td>341,480</td>
<td>4,516,840</td>
<td>20.625</td>
<td>2.5</td>
<td>8.25</td>
<td>0.000010915</td>
</tr>
<tr>
<td>MA, USA</td>
<td>316,935</td>
<td>4,167,699</td>
<td>18.766</td>
<td>2.219</td>
<td>8.456962596</td>
<td>0.000010764</td>
</tr>
<tr>
<td>NV, USA</td>
<td>286,925</td>
<td>4,258,491</td>
<td>19.297</td>
<td>1.922</td>
<td>10.0406243</td>
<td>0.000012323</td>
</tr>
<tr>
<td>CT, USA</td>
<td>161,256</td>
<td>1,943,544</td>
<td>8.359</td>
<td>1.109</td>
<td>7.5374211</td>
<td>0.000009954</td>
</tr>
<tr>
<td>Alaska, USA</td>
<td>57,887</td>
<td>786,214</td>
<td>3.032</td>
<td>0.297</td>
<td>10.20875421</td>
<td>0.000010998</td>
</tr>
<tr>
<td>Oxford, OH, USA</td>
<td>4,176</td>
<td>37,631</td>
<td>0.141</td>
<td>0.031</td>
<td>4.548387097</td>
<td>0.000009325</td>
</tr>
</tbody>
</table>

Table 4.2.1 Experimental results of running times when $K = |V|$.
4.3 Determining the doubling dimension for road maps

4.3.1 Definition of Net-tree
In 2004, Har-Peled and Mendel [5] proposed a new data structure for a graph $G\langle V, E \rangle$, called net-tree, which can be used to determine the doubling dimension for road maps. A net-tree of a finite metric space $P$ is a tree $T$ such that:

1. Its set of leaves is $P$.
2. Let $P_v$ be the set of leaves in the subtree rooted at a vertex $v \in T$. Associate with each vertex $v$ a point $rep_v \in P_v$.
3. Internal vertices have at least two children.
4. Each vertex $v$ has a level $l(v) \in \mathbb{Z} \cup \{ -\infty \}$ . The levels satisfy:
   a. $l(v) < l(p(v))$, where $p(v)$ is $v$’s parent.
   b. All leaves’ levels are $-\infty$
5. Let $\tau$ be the packing dimension, a net-tree must have the following properties:
   a. Covering property: For every vertex $v \in T$
b. Packing Property: For every non-root vertex \( v \in T \)

\[ \text{Ball}(\text{rep}_v, \frac{\log \varepsilon}{1-\varepsilon}) \supseteq P_v \]

\[ \text{Ball}(\text{rep}_v, \frac{1-\varepsilon}{2(1-\varepsilon)} \cdot \varepsilon^{\ell(\frac{1}{2})}) \cap P_v \subseteq P_v \]

c. Inheritance property: For every non-leaf vertex \( u \in T \), there exists a child \( v \in T \) of \( u \) such that \( \text{rep}_v = \text{rep}_u \)

![Figure 4.3.1: The tree on the left side is a net-tree of the map on the right side.](image)

### 4.3.2 Using net-tree to approximate doubling dimension

According to Proposition 9.2 in Har-Peled and Mendel’s paper [5], a \( \varepsilon \)-approximation of a metric’s doubling constant can be quickly calculated by counting the max out degree of its net-tree.

### 4.3.3 Net-tree construction

#### 4.3.3.1 Har-Peled and Mendel’s algorithm

The basic idea behind their net-tree construction algorithms consists of two stages. First, they compute \(|V|\) centers, called *greedy permutation* of \( V \), by applying Gonzalez’s algorithm, and then add every vertex into the net-tree according to the order of the permutation. They claimed that the net-tree structure can be used to obtain improved algorithms for a lot of geometrical problems and to verify whether a graph has a bounded doubling dimension. Their algorithm relies on the following assumption: “*the input is*
given via a black box that can compute the distance between any two points in the metric space in constant time.” With this assumption, they claim that the total running time of greedy permutation for $V$ is $O(|V|\lambda^{O(1)} \cdot \log(\varphi \cdot |V|))$, and they also give an improved version with $O(|V|\lambda^{O(1)} \cdot \log |V|)$ running time. As we’ve mentioned in Section 1, the obvious way to build such a “black box” is to apply Floyd’s all-pairs-shortest-paths algorithm which requires $O(n^3)$ time. It’s impractical to apply this algorithm on graphs containing millions of vertexes.

### 4.3.3.2 My algorithms

The way I construct a net-tree is similar to Har-Peled and Mendel’s except that I use my implementation of the clustering algorithm to generate a permutation for all vertexes. The overall running time for creating a net-tree is still $O(|V|\lambda^{2} \cdot \log \varphi \cdot \log |V|)$.

My algorithm has following advantages when being applied to net-tree construction in real road maps:

1. **The running time is acceptable.** In Section 4.2, experimental results show that our algorithm can be applied to large-size road maps. It took only about 2.7 hours to generate a vertex permutation for the USA road map which contains more than 26 million vertexes.

2. **My implementation does not rely on distance matrix.** The distance between two vertexes is calculated by traversing the graph and no distance matrix is needed to be pre-calculated.

3. **The algorithm is very simple (especially compared to Har-Peled and Mendel).**

The way I create a net-tree also consists of two steps:

**Step 1:** Using my k-centers algorithm to generate a permutation for all vertexes of the graph.
Step 2: Following the order of the permutation, iteratively add each vertex to the net-tree according to some rules.

To construct a net-tree, I slightly modify my $k$-center algorithm in the following ways:

1. The input $k$ was set to the total number of vertexes of the graph.

2. The output contains not only the greedy permutation $\langle p_1, \ldots, p_n \rangle$, but also

   a. $cp_k$, the nearest center to $p_k$ in the previous round.

   b. $r_k = \min_{1 \leq i < j \leq k-1} d(p_i, p_j)$

$cp_k$ can be calculated by duplicating all elements at the end of every round. Since there are at most $\log_\phi$ rounds, the total running time of $cp_k$ calculation is $O(|V|\log_\phi)$. $r_k$ actually is the value of $D_k$, so it just adds $O(|V|)$ overall. Both $O(|V|\log_\phi)$ and $O(|V|)$ are dominated by the overall running time, so the overall running time remains the same which is $O(|V|\lambda^2 \cdot \log_\phi \cdot \log|V|)$.

4.3.4 Experimental results for determining the doubling dimension

Har-Peled and Mendel claimed that, with $\tau \geq 11$, if a metric space has doubling constant $\lambda$, then each vertex of the Net-Tree has at most $\lambda^c$ children, where $c$ is some universal constant $\geq 1$. That is, let $D$ denote the maximum number of children of any node in the net-tree. Then $\log_2 D$ is a constant approximation to the doubling dimension of the space.

I built a net-tree with $\tau=11$ for each road map and computed $D$ for every net-tree. The results are presented in Table 4.3.2. Please note that $\log_2 D$ should be larger than actual doubling dimension, because $\tau=11$ (not 2) was used in net-tree construction. That is, $\log_2 D$ actually is a constant approximation to the “multiplying by 11” dimension which is also a loose upper bound for doubling dimension. For ease of comparison to true doubling dimension, I calculated $D^{(2/11)}$ which is analogous to the doubling constant and $\log_2 D^{(2/11)}$ which is analogous to the doubling dimension. Please note $D^{(2/11)}$ and $\log_2 D^{(2/11)}$
don’t mean anything. They are just arithmetically analogous to real doubling constant and dimension.

<table>
<thead>
<tr>
<th>Map Name</th>
<th>D</th>
<th>log₂D</th>
<th>D''/D</th>
<th>log₂D''</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>422</td>
<td>8.721099189</td>
<td>3.001439105</td>
<td>1.585654398</td>
</tr>
<tr>
<td>TX</td>
<td>451</td>
<td>8.816983623</td>
<td>3.037928534</td>
<td>1.603087932</td>
</tr>
<tr>
<td>Canada</td>
<td>334</td>
<td>8.383704292</td>
<td>2.876490831</td>
<td>1.524309871</td>
</tr>
<tr>
<td>FL</td>
<td>449</td>
<td>8.810571635</td>
<td>3.03547463</td>
<td>1.601922115</td>
</tr>
<tr>
<td>IL</td>
<td>347</td>
<td>8.438791853</td>
<td>2.896530369</td>
<td>1.534325791</td>
</tr>
<tr>
<td>OH</td>
<td>333</td>
<td>8.379378367</td>
<td>2.874923046</td>
<td>1.523523339</td>
</tr>
<tr>
<td>TN</td>
<td>338</td>
<td>8.400879436</td>
<td>2.882723818</td>
<td>1.527432625</td>
</tr>
<tr>
<td>AZ</td>
<td>372</td>
<td>8.539158811</td>
<td>2.933401053</td>
<td>1.552574329</td>
</tr>
<tr>
<td>WA</td>
<td>523</td>
<td>9.030667136</td>
<td>3.120850999</td>
<td>1.641939479</td>
</tr>
<tr>
<td>OR</td>
<td>453</td>
<td>8.82336724</td>
<td>3.040373551</td>
<td>1.604248589</td>
</tr>
<tr>
<td>NM</td>
<td>398</td>
<td>8.636624621</td>
<td>2.969655103</td>
<td>1.570295386</td>
</tr>
<tr>
<td>NJ</td>
<td>313</td>
<td>8.290018847</td>
<td>2.842728176</td>
<td>1.507276154</td>
</tr>
<tr>
<td>MA</td>
<td>259</td>
<td>8.016808288</td>
<td>2.746513735</td>
<td>1.457601507</td>
</tr>
<tr>
<td>NV</td>
<td>248</td>
<td>7.95419631</td>
<td>2.724926868</td>
<td>1.446217511</td>
</tr>
<tr>
<td>CT</td>
<td>255</td>
<td>7.994353437</td>
<td>2.73875233</td>
<td>1.45518807</td>
</tr>
<tr>
<td>Alaska</td>
<td>154</td>
<td>7.266786541</td>
<td>2.49879737</td>
<td>1.321233916</td>
</tr>
<tr>
<td>Oxford</td>
<td>109</td>
<td>6.768184325</td>
<td>2.34661428</td>
<td>1.230578968</td>
</tr>
<tr>
<td>San Francisco</td>
<td>241</td>
<td>7.912889336</td>
<td>2.710778346</td>
<td>1.438707152</td>
</tr>
<tr>
<td>USA</td>
<td>17214</td>
<td>14.07129475</td>
<td>5.890610769</td>
<td>2.558417228</td>
</tr>
</tbody>
</table>

Table 4.3.2

4.4 Conclusion of the experiments

The experiments are designed to validate two hypotheses:

1. All road maps tested have reasonably small doubling constant $\lambda$ and aspect ratio $\phi$.

2. My algorithm is fast enough to use in practice.

From Table 4.3.2, we know that the maximum upper bound of doubling constant is 5.89 and the logarithm of the spread shown in Table 4.1 ranges from 15 to 35. If we assign these constants to $\lambda$ and $\phi$ in our big $O$ notation, we have $O(35^2\cdot 5.89\cdot |V|\cdot \log |V|)$. Although this upper bound is not as good as the results shown in experiments, it’s still within the practical range.
I also experimentally determined that this algorithm is fast enough to use in practice. It took less than one hour to generate a permutation for all vertexes of US road map with more than 26 million vertexes. I also experimentally estimated the upper bounds of the doubling constant $\lambda$ and aspect ratio $\Phi$ so as to theoretically prove that this algorithm can be used in practice.

5. Future research directions

5.1 The Upper bound of the running time

As mentioned in section 4.4, we believe that there exists a better running time upper bound of my algorithm. To get a better upper bound in future, we will consider the following aspects:

1. Can we get a better upper bound for doubling constant $\lambda$? Most people suspect that the doubling constant $\lambda$ of real road maps should be around 5 or 6, but ours retrieved from experiments is far bigger than that. One explanation might be the fact that our greedy permutation algorithm has the possibility of using $\lambda^2$ centers in some situations where $\lambda$ would be sufficient. Is the doubling constant of road maps really in the 20-40 range, or is this an overestimate due to our approach?

2. Can we get a better upper bound for the running time of our algorithm? In particular, a few algorithms exist for metrics with bounded doubling dimension that do not include $\log \Phi$ in the running time. It is possible that a more careful accounting could allow us to remove this factor.

5.2 About Har-Peled and Mendel’s Net-Tree construction algorithm

Har-Peled and Mendel claimed that net-tree can be used to obtain improved algorithms for lots of geometrical problems such as distance oracles and approximate traveling salesman problem. Now, with my algorithm, net-tree can be easily constructed, so future research can attempt to find practical and efficient distance oracles and TSP algorithms for road maps.
6. Conclusion

In this thesis, I present a new implementation of Gonzalez’s clustering algorithm to approximately solve the \textit{k-center} problem in $O(|V|^2 \cdot \log \varphi \cdot \log |V|)$ in real road maps. Compared with other versions of Gonzalez’s clustering algorithm, my algorithm has the following advantages when being applied in real maps:

1. It does rely on distance matrix.
2. It runs in $O(|V|^2 \cdot \log \varphi \cdot \log |V|)$ time and $O(|V|)$ space. Experimental results show that it’s fast enough to be applied in large-scale maps.

This new implementation of clustering algorithm can be applied in the net-tree construction algorithm which requires the permutation of all vertexes in the initial steps.

References


Appendix A: How to export map data from StreetMap USA

The US street map is in Smart Data Compression (SDC) format. It can be completely or partially exported by DDA. DDA is a data distribution program coming with the “ESRI Data & Maps and StreetMap USA 2006” CD.

Followings list the key steps to export data using DDA:

**Step 1: Open the map project file *.axl**

Step 2: Select the features (left window) and the region (main window) you want to export
Step 3: Click the “E” icon to launch “Extraction options”.

Step 4: Specify output format and output directory.
Appendix B: How to convert map data from *ESRI* Shapefile Network Dataset to our map format

Step 1: Create a new document in ArcMap:

Step 2: Add the map data in Shapefile Network Dataset format to the current layer.
Step 3: Add our plug-in “GetMapData” into ArcMap and run it.

Step 4: You must specify the output location and file name of the map. In some shape files, edges might contain more than one attribute such as length and speed limit, so you have to tell the program the index of the length attribute.
Appendix C: Build Network Dataset for a shape file
To traverse a map efficiently in ArcGIS, we need to build a network dataset for it. You can use ArcCatalog to build the network dataset. The operations are very simple, just as shown below.

Appendix D: Steps of duplicating the experiments
Step 1: If your map data is already in Shapefile or Smart Data Compression (SDC) format but without Network Dataset, you must follow Appendix C to build its Network Dataset. Then go to Step 2.

Step 2: If your map data is already in Shapefile or SDC format and contain Network Dataset, then follow Appendix B to convert map data into our format.

Step 3: Use kc.exe to generate k centers for your map. kc.exe will create a binary file, called Centers_of_XXXX_StartFrom=XXXX.bin, under the same working folder.
This file contains all centers and other data that will be used in Net-Tree construction.

Step 4: Use nt.exe to construct a net-tree for your map. nt.exe will output the net-tree to a binary file, called Centers_of_XXXX_StartFrom=XXXX.bin.nettree.

Step 5: Use loadnt.exe to load and verify a net-tree. Loadnt.exe will tell you the number of total nodes contained the net-tree and the maximum number of children of each level.

Here is an example about how to construct a net-tree for USA Street Map 2006 in “ESRI Data & Maps and StreetMap USA 2006” CD,

Step 1: Since the CD contains network dataset of USA street map, we can directly use “GetMapData” add-in to get raw map data.

Step 2: Use RawDataConverter.exe to convert the data format.
RawDataConverter.exe will create a new map file, called usa.map in the same folder. Kc.exe will work on this file in next step.

Step 3: use kc.exe to generate all centers

Kc.exe will create a binary file called Centers_of_usa.map_StartFrom=10000000.bin. This file contains all centers and other data that will be used in net-tree construction.
Step 4: use nt.exe to construct net-tree.

Nt.exe will save the net-tree as a binary file called
Centers_of_usa.map_StartFrom=10000000.bin.nettree.
Step.5: use loadnt.exe to load the net-tree file
Appendix E: C++ implementation of DijkstraAlgorithmWithPruning

```cpp
typedef pair<double, int> di;
#define MAX_DOUBLE 999999999.0
void DijkstraAlgorithmWithPruning (
    const Graph & G,
    int startVertex,
    vector<double> & globalDistanceVector,
    vector<di> & globalDistanceHeap,
    vector<int> & mapVectorToHeap )
{
    vector<double> D (G.size() , MAX_DOUBLE) ;
    priority_queue<di, deque<di>, greater<di> > Q ;
    D[startVertex] = 0.0;
    Q.push(di (0.0, startVertex));
    while(!Q.empty()) {
        di top = Q.top();
        Q.pop();
        int v = top.second ;
        double d = top.first;
        if(d < globalDistanceVector[v]) {
            globalDistanceVector[v] = d ;
            DecreaseKey (globalDistanceHeap,
                mapVectorToHeap[v] ,
                d ,
                mapVectorToHeap[v]);
        }
        for(vector<di>::const_iterator it = G[v].begin(); it != G[v].end(); it++) {
            int v2 = it->first ;
            double cost = it->second;
            if(D[v2] > D[v] + cost) {
                D[v2] = D[v] + cost;
                Q.push(di(D[v2], v2));
            }
        }
    }
}
```

Appendix F: Formats of Binary Files

<table>
<thead>
<tr>
<th>Types</th>
<th>Created by</th>
<th>Processed by</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw Map Data</td>
<td>GetMapData</td>
<td>RawDataConverter</td>
<td>0) 4 bytes: vertex id1 (int32)</td>
</tr>
<tr>
<td></td>
<td>A C# ArcGIS plug-in project</td>
<td></td>
<td>1) 4 bytes: vertex id2 (int32)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2) 8 bytes: distance (double)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3) repeat from 0)</td>
</tr>
<tr>
<td>Map Data in our format</td>
<td>RawDataConverter</td>
<td>K-Center Algorithm</td>
<td>0) 4 bytes: # of edges (int32)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1) 4 bytes: # of vertexes (int32)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2) 4 bytes: vertex id1 (int32)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3) 4 bytes: vertex id2 (int32)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4) 8 bytes: distance (double)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5) repeat from 2)</td>
</tr>
<tr>
<td>K-center saved file</td>
<td>K-Center Algorithm</td>
<td>Net-Tree Algorithm</td>
<td>0) 4 bytes: center id starting from 0 (int32)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1) 4 bytes: center’s vertexId (int32)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2) 4 bytes: distance from last center (float)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3) 4 bytes: center’s center id (int32)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4) repeat from 0</td>
</tr>
<tr>
<td>Net-Tree saved file</td>
<td>Net-Tree Algorithm</td>
<td>Loadnt</td>
<td>0) 4 bytes: father’s (int32)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1) 4 bytes: level (int32)</td>
</tr>
</tbody>
</table>