# Routing Table Optimization using t-spanner Technique

*Thesis submitted in partial fulfillment of the requirements for the award of degree of*

**Master of Engineering**
in
**Computer Science and Engineering**

*Submitted By*
**Navadiya Hareshkumar**
**(Roll No. 801132011)**

Under the supervision of:
**Dr. Deepak Garg**
Associate Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

**June 2013**

## Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Routing Table Optimization using t-spanner Technique"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. Deepak Garg* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Hareesh

**(Navadiya Hareshkumar)**

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

**(Dr. Deepak Garg)**
Associate Professor
Computer Science and
Engineering Department
Thapar University
Patiala

**Countersigned by:**

**(Dr. Maninder Singh)**
Associate Professor and Head
Computer Science and Engineering Department
Thapar University
Patiala

**(Dr. S. K. Mohapatra)**
Dean (Academic Affairs)
Thapar University
Patiala



Page | i

# Acknowledgement

I would like to express my deep sense of gratitude to my supervisor, **Dr. Deepak Garg**, Associate Professor, Computer Science and Engineering Department, Thapar University, Patiala, for his invaluable help and guidance during the course of thesis. I am highly indebt to him for constantly encouraging me by giving his critics on my work. I am grateful to him for giving me the support and confidence that helped me a lot in carrying out the research work in the present form. And for me, it's an honor to work under him.

I also take the opportunity to thank **Dr. Maninder Singh**, Associate Professor and Head, Computer Science and Engineering Department, Thapar University, Patiala, for providing us with the adequate infrastructure in carrying out the research work.

I would also like to thank my parents and friends for their inspiration and ever encouraging moral support, which went a long way in successful completion of my thesis.

Above all, I would like to thank the almighty God for His blessings and for driving me with faith, hope and courage in the thinnest of the times.

**Navadiya Hareshkumar**

# Abstract

Dynamically changing graphs are used in various applications of graph algorithms. The scope of these graphs is in graphics, in communication networks and VLSI design where graphs are subjected to change, such as addition and deletion of links and nodes. There is a rich body of the algorithms and data structures used for dynamic graphs. The thesis contains techniques and data structures used in various dynamic algorithms.

There are many applications in distributed systems and communication networks where spanner appears as the underlying graph structures. For instance in dynamically changing graphs where the edge weights are changing can be visualized as a communication network where the load on the links is changing.

In this thesis, a new clustering technique is introduced that is based on two parameters; radius and degree of node. This thesis work proposes an algorithm of constructing sparse t-spanners for arbitrary undirected weighted graphs.

This thesis also explains fully dynamic algorithm for maintaining t-spanner of undirected weighted graphs under a sequence of update operations like insertion and deletion of links.

This algorithm is applied on communication networks to optimize routing table space and also for good routing scheme by taking weight as congestion load of network link among two routers.

All the algorithms are suggested in this thesis are practically implemented and tested for undirected weighted graph by varying number of nodes and number of link of graph.

# Table of Contents

# List of Figures

## List of Tables

# List of Algorithms

# Chapter 1
# Introduction

## 1.1 Dynamic Graphs

A Graph is a collection of nodes and links which represents the network of entities and association between these nodes. Dynamic graphs are not fixed with respect to time, but can evolve through local changes of the graph. Any problem associated with dynamic graph should be solved quickly as the new changes arrive after each modification. In the current scenario, no problem is truly static, so each problem may have some dynamic changes which make the problem dynamically rather than absolutely static. In communication networks for instance, a network changes its routes as nodes and links go down due to failures and repairs. The dynamic graphs are used in almost every application, which is dynamically changing [1]. The World Wide Web is the biggest example of dynamic graphs as the new servers and hosts keep on adding in the graph and make the graph dynamic. While the updation takes place in the graph, the dynamic graph maintains the various properties of graph like, graph connectivity, planarity, spanning forest of the graph [2] [3].

## 1.2 Dynamic Graph Algorithm

The aim of a dynamic graph algorithm is to provide an efficient solution of a problem after dynamic changes, rather than having to recalculate it from scratch each time. Because of dynamic nature, it is not surprising that dynamic algorithms are often more difficult to design and analyze than their static counterpart [4].

A dynamic graph problems can be classified according to the types of updates performed. In particular, a dynamic graph problem is said to be **fully dynamic** if the update operations include unrestricted insertions and deletions of edges or vertices. A dynamic graph problem is said to be **partially dynamic** if only one type of update, either insertions or deletions, is performed. A problem of dynamic graph is said to be incremental if only insertions are performed, while it is said to be decremental if only deletions are performed [5].

The dynamic graph algorithms maintain the updates operation on directed and undirected graph. For undirected graph, the dynamic graph algorithm uses the

different techniques like clustering, sparsification and randomization by using different data structure tools. These algorithms reserve the properties of dynamic graphs like vertex and edge connectivity, minimum spanning tree [2] [3].

For directed graph, the dynamic graph algorithm deals with the two problems for maintaining the updates and query operation.

- In the fully dynamic transitive closure problem a directed graph G(V, E) is maintained under an intermixed sequence of the following operations [6]:
  - **Add(p, q)**: add an edge from p to q.
  - **Remove(p, q)**: remove an edge from p to q.
  - **Query(p, q)**: return yes if q is reachable from p, and return no otherwise.
- In the fully dynamic All Pairs Shortest Path (APSP) problem [7] [8] a directed graph G(V, E) is maintained with real-valued edge weights under an intermixed sequence of the following operations:
  - **Update(e(p, q), w)**: update the weight of edge (p, q) to the real value w, this will includes as a special case both edge insertion (if the weight w is set from 0 to $+\infty$) and edge deletion (if the weight is set to $w = +\infty$).
  - **Distance(p, q)**: output the shortest distance from p to q.
  - **Path**(p, q): report a shortest path from p to q, if any [2].

There has been a lot of research in the area of directed graph where these two problems are widely discussed and provided with some solution. The main goal in case of directed graph is to minimize the running time of the algorithms. The undirected graph in dynamic graph algorithm techniques achieves the better time bound as compared to directed graph. Undirected graphs provide the algorithmic solution in poly-logarithmic time bounds as compared to polynomial time bound of directed graphs. The thesis provides the comparison of various dynamic graph algorithmic techniques, which are used in undirected graph and compares the time bound for update and query operation for various dynamic graph properties.

The dynamic graph algorithm provides answer to the following queries and many more [9].

- Check whether the vertex can be reachable from the given vertices?

- Retain the minimum spanning forest, vertex connectivity and bipartiteness like properties of graph.
- Various updates taking place in dynamic graph when the new edges and vertices are added and deleted from the graph.

So, as the internet expands and the users on the web increases, there is a lot of requirement for the efficient dynamic graph algorithms.

## 1.3 Dynamic Graph Algorithm Techniques

There are three techniques for dynamic graph algorithm as mentioned below.

### 1.3.1 Clustering

The clustering is the subdivision of graph vertex set into groups. Frederickson [10] has suggested this clustering technique. **Clusters** are partitions of the graph into a smaller subdivision of connected sub graph. To store the information about the graph edges and vertices, the tree data structure is used.

**Definition**: A clustering C(G) of a graph G(V, E) is a subdivision of vertices V into disjoint and nonempty subset of $\{C_1, C_2, C_3 \ldots C_k\}$ where $C_i \epsilon C$

The technique works as follows:

- It is based upon the decomposition of vertex set V into the sub graph called clusters and the decomposition applied recursively to the higher level. The information about the sub graph is combined with topology tree [3].
- The clustering technique can be improved further in which the edges also divided into multiple clusters, and only one edge will be selected depending upon the topology of the spanning tree.

Fully dynamic algorithms based only on a single level of clustering obtain typically time bounds of the order of $O(m^{2/3})$ [11] [12]. When the recursive partition gives the better $O(m^{1/2})$ time bounds and it can be achieved by using 2-dimensional topology trees [10] [13].

### 1.3.2 Sparsification

This technique was introduced by Dijkstra [14] and it is a general technique which can be used as a black box in designing algorithms. The technique decreases a size of the graph and makes the dynamic algorithm an efficient algorithm. This technique improves the time bound of the algorithm and become analogous to the sparse graphs.

This technique does not demand the structural detail of the graph. In a large number of situations both clustering and sparsification have been combined to produce an efficient dynamic graph algorithm [4].

### 1.3.3 Randomization

Clustering and sparsification allow one to design efficient deterministic algorithms for fully dynamic problems. This technique introduced by Henzinger and king [15] for dynamic graph algorithm uses the power of randomization for improving the faster update time.

The technique advances the lower time bound for fully dynamic graph algorithm for properties like connectivity, bipartiteness and minimum spanning forest of a graph based on random sampling and graph decomposition. The result of this technique achieves the faster fully and partially dynamic algorithm.

## 1.4 Application of Dynamic Graph

In many situations, graphs are likely to change and need to be updated to maintain the graph properties. According to the changes the dynamic graph algorithms can be classified into categories. The algorithm is incremental when it deals with the insertion of edges in the graph; and the graph is decremental when in deals with the deletion of edges in the graph. The algorithm is fully dynamic when the edges can be inserted and deleted at same time and can be handled by the same algorithm.

There are lots of applications for dynamic graph algorithm where the better time bound is required to solve the complex problems like,

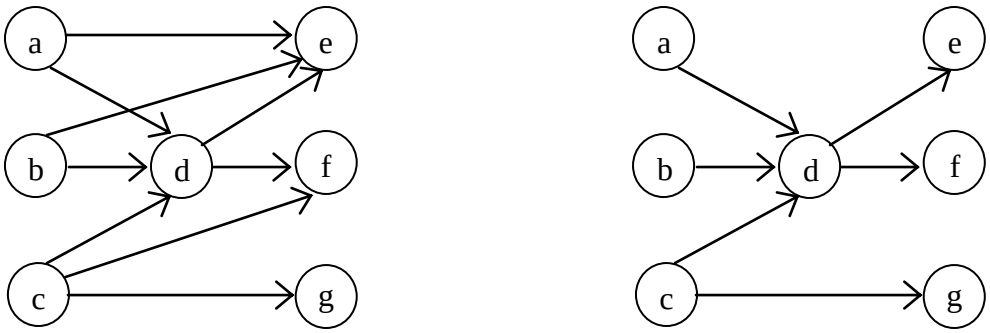
- Communication networks
- Assembling planning
- Chip design
- Graphics

These algorithms have a wide area of research for various applications some of them are mentioned below:

- Graph Spanner
- Incremental cycle detection or topological ordering occurring in circuit evaluation [16]
- pointer analysis [17]
- management of compilation dependencies [18] and deadlock detection [19]

## 1.5 Spanner Graph

A spanner is sparse sub-graph of directed or undirected graph that keep approximate distance between all pairs of nodes [20]. Let denote node set and link set of a graph G by N and L respectively. $d_G(p, q)$ is the distance from node p to node q in graph G. Given an integer $\mathbf{t \geq 1}$, a spanning sub-graph $G'(N, L_S)$ of the graph G(N, L) is called a t-spanner of G if $d_S(p, q) \leq t * d_G(p, q)$ for every $p, q \in N$ and $L_S \subset L$. Here t is called the **stretch factor** associated with the t-spanner and $L_S$ is the size of spanner graph G′. In unweighted graph, distance $d_G(p, q)$ should be considered as number of links between node p and node q. while in weighted graph, it should be sum of weights of links to reach from node p to node q. In worst case, weight between any two nodes may go up to t times the original weight of the selected link.




(a) (b)

Figure 1.1 (a) A dense unweighted graph G (b) A sparse graph G′

A tree t-spanner of a graph G is a spanning subtree T of G in which the distance between every pair of vertices is at most t times their distance in G [21]. Throughout the thesis, graph G(N, L) and its spanner graph $G'(N, L_S)$ is taken and ***m*** and ***n*** denote by number of links and nodes in G, Respectively.

## 1.6 Application of Spanner Graph

Graph spanners arise in many applications, including computational biology in the process of reconstructing phylogeny trees from matrices. Entries in the matrix represent genetic distances among contemporary living species, Graph spanner are also used in computational geometry and robotics [22] [23] [24] [20].

There are many applications in distributed systems [25] and communication networks [26] where spanner appears as the underlying graph structures. Awerbuch [23] and Peleg and Ullman [27] show that the quality of a spanner is very closely related to the

time and communication complexity of any synchronizer for the network based on this spanner. For instance in dynamically changing graphs where the edge weights are changing can be visualized as a communication network where the load on the links is changing.

## 1.7 Routing Table

A routing table uses the same idea that one does when using a map in package delivery. It should be known beforehand that send would go from which source node to which target node. If the node cannot directly connect to the target node, so it has to send it through intermediate nodes along a proper route to the target node. Every node do not try to find out which path might work; instead, a node will send a packet to a gateway in network, which have responsibility to route the package of data to the correct target. Each gateway uses a routing table to keep track of which way to deliver various packages of data. A routing table is a database which stores track of various paths, like a map, and helps the gateway to provide this information to the node requesting the information.

With node-by-node routing, each routing table lists, for all reachable targets, the address of the next gateway along the path to that target: the next node. Assuming that the routing tables should be consistent for relaying packets to their target's next hop thus suffices to deliver data anywhere in a network. Node-by-node is the fundamental characteristic of the IP Internetwork Layer [28] and the OSI Network Layer.

The primary goal of a router is to forward a packet toward its target network, which is the target IP address of the packet. For this, a router searches the routing information stored related to source and target in its routing table.

A routing table is a data file in RAM that is used to store route information about directly connected and remote networks. The routing table stores the network or next node associations. This information gives a router that a particular target can be optimally reached by sending the packet to a specific router that represents the next node on the way to the final target. The next hop association can also be the outgoing or exit interface to the final target.

A network that is directly attached to one of the router interfaces is called directly connected network. A router interface is configured using an IP address and subnet mask and this interface is considered as a host on that direct network. Now the routing

table stores the information of the network address and subnet mask of the interface and also entered the information about interface type and number. When router wants to forward a packet to any host, such as a web server, if it is on the same network then it send to a router's directly connected network.

A remote network is a network that is not directly connected to the router. In other way, a remote network is a network that can only be reached by sending the packet to another router. There are two ways to entered remote networks into routing table. One is a dynamic routing protocol and second is configuring static routes. In dynamic routing protocol, router learned remote networks automatically and routes are dynamically added into routing table. In Static routes, network administrator enters the routes entry manually.

In computer networking a routing table, or routing information base (RIB), is a data table stored in a router or a networked computer that lists the routes to particular network targets, and in some cases, metrics (In this thesis, transfer load or congestion load) associated with those routes. The primary function of a router is to forward a packet toward its target network. To do this, a router needs to search the routing information stored in its routing table. A routing table is same as data file stored in RAM that is used to track route information about directly connected networks. The need to record routes to large numbers of devices using limited storage space represents a major challenge in routing table construction [29].

## 1.8 Structure of the Thesis

The rest of thesis is organized in the following order:

**Chapter-2:** This chapter explains the Naive greedy spanner algorithm, Thorup-Zwick spanner and Baswana-Sen spanner algorithm for undirected graph and explained other technique for directed graph with the suitable an example.

**Chapter-3:** This chapter provides the problem statement of the thesis and the methodology used to solve the problem. The clear insight about the problem statement is defined. The new approach is discussed to solve the spanner graph.

**Chapter-4:** This chapter describes the preliminaries data structure needed to perform the designed t-spanner algorithm and introduces the new clustering technique for spanner graph.

**Chapter-5:** This chapter explains all the phase of the algorithm required for computing the graph spanner with detailed case study and this chapter also explains the fully dynamic algorithm for t-spanner to handle the updation of graph at any time.

**Chapter-6:** In this chapter t-spanner algorithm designed in chapter 5 is applied on application such as routing table optimization. This chapter analyzes the result that is concluded from the suggested t-spanner algorithm. The snap shots of designed system of an algorithm are mention in this chapter.

**Chapter-7:** This chapter provides the conclusion for the new technique introduced and gives the insight of usefulness of the technique for the various applications for future scope.

# Chapter 2

# Literature Review

It is known that any graph G has a 1-spanner itself. There is better to find the spanners of small size means containing few edges while at the same time achieving good stretch. There is a tradeoff between the two, one is the complete unweighted graph $K_n$ has 1-spanner itself and this spanner thus has a quadratic number of edges and Larger stretch creates significantly reduction in the spanner size. The various way of spanner constructions shown in this chapter are expected to achieve the best possible tradeoff between size and stretch.

Spanners are well-motivated, both from a practical and a theoretical point of view. They are the underlying graph structure for constructions in distributed systems and communication networks.

## 2.1 The Naïve Greedy Spanner

A greedy spanner is a spanner obtained by a very simple greedy algorithm [22]. In the greedy approach, all the edges of graph are processed in ascending order of weights shown in algorithm 2.1.

---

**Algorithm 2.1** Naïve greedy spanner

---

**Procedure** Naïve-Greedy (G, t)

1. Sort the edges in E by non decreasing order of weight
2. $E_S \leftarrow \phi$
3. ***for*** each e(u, v) $\epsilon$ E ***do***
4. ***if*** $d_{(V, Es)}(u, v) > (2t-1) * w(u, v)$ ***then***
5. $E_S \leftarrow E_S \cup \{e(u, v)\}$
6. ***Output*** $G(V, E_S)$

---

This algorithm takes as input an undirected graph G(V, E) with non-negative edge weights w(e) and an integer $t \geq 1$. It uses variable where $E_S$ is set of edges of (2t - 1)-spanner graph, w(u, v) is weight of edge from u to v and $d_{(V, Es)}(e)$ is sum of weight from one end point to other in spanner graph. It can be implemented in $O(m(n^{1+1/t} + n \log n))$ time and outputs a spanner $G(V, E_S)$ with stretch 2t - 1.

**Lemma 2.1**: Let t ≥ 1 be an integer. Let G(V, E) be an *n*-vertex unweighted and undirected graph whose smallest cycle has length at least 2t + 1 (where the length is defined to be infinite if G is acyclic). Then |E| = $O(n^{1+1/t})$. The (possibly infinite) cycle length in Lemma 2.1 is known as the girth of graph G. It can be shown that the spanner G(V, $E_S$) from algorithm greedy has girth at least 2t + 1 so $E_S$ has $O(n^{1+1/t})$ edges by Lemma 2.1. A girth conjecture of Erdös [30] states that the result in Lemma 2.1 is tight. More precisely, it is conjectured that for any integer t ≥ 1, there are *n* vertex graphs of size ($n^{1+1/t}$) and girth at least 2t + 1. The conjecture has only been settled for t = 1, 2, 3, 5. Here the girth conjecture follows that the greedy spanner achieves optimal tradeoff between size and stretch but with the slow construction time.

Now understand greedy approach with the practical example by taking t = 3. Figure 2.1 shows the undirected weighted graph G(V, E) having 6 vertices, a naive greedy algorithm is applied on this graph and compute the 3-spanner of given graph G. Steps for computing 3-spanner graph G(V, $E_S$) of graph G(V, E) using this algorithm is as follows.

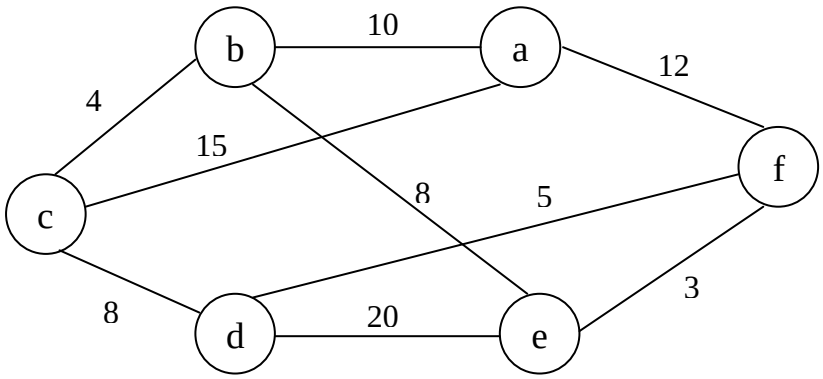



Figure 2.1 A simple undirected weighted graph

**Step1**: E = {(e, f), (c, b), (d, f), (b, e), (c, d), (b, a), (a, f), (c, a), (d, e)} and their weight W = {3, 4, 5, 8, 8, 10, 12, 15, 20} respectively.

**Step2**: Now check for every edge in the set E and here minimum weight edges added in the set $E_S$ that follows the condition given in algorithm.

For example here edges (e, f), (c, b), (d, f), (b, e) are added directly into $E_S$ without any conflict shown in Figure 2.2.

Now further proceed, check for edge (c, d). Here path is already present in 3-spanner graph from c to d with weight of 20 (i.e. path c-b-e-f-d) and based on condition 20 < 24 (i.e. 3*8) so direct edge (c, d) is not added in to $E_S$.

At this stage $E_S$ = {(e, f), (c, b), (d, f), (b, e)}. Edge (b, a) is directly added and edge (a, f) is not insert into $E_S$, because path a-b-e-f and 21 < 36. Same edges (c, a) and (d, e) are not inserted into $E_S$.

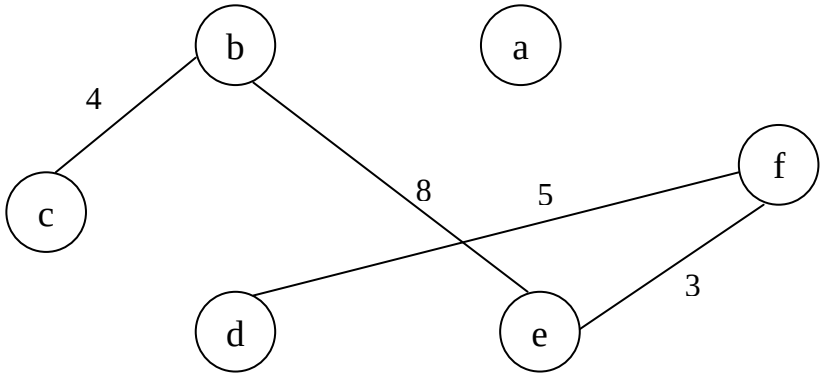



Figure 2.2 Edges added in 3-spanner graph without conflict

**Output:** Now Final spanner edge set $E_S$ = {(e, f), (c, b), (d, f), (b, e), (b, a)} and 3-spanner of graph given in Figure 2.1 is shown in Figure 2.3.

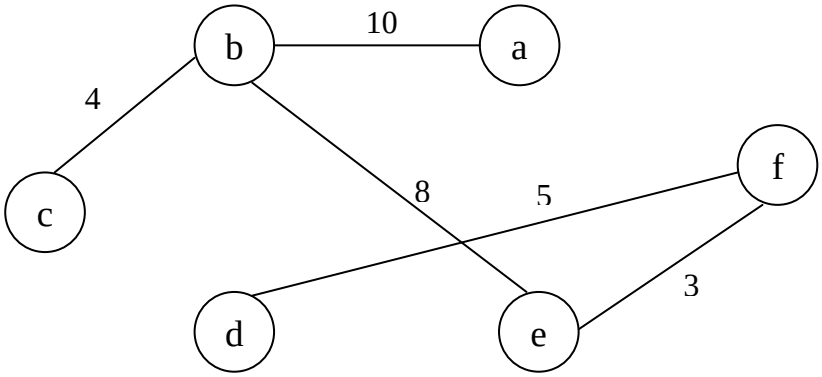



Figure 2.3 A final 3-spanner graph using naïve greedy approach

The time complexity of 3-spanner greedy algorithm is $O(mn \log n)$. There are others better bound algorithm present for spanner graph.

## 2.2 The Thorup-Zwick Spanner

Cohen [31] presented a randomized algorithm with $O(tmn^{1/t})$ expected running time for computing a spanner of $O(tn^{1+1/t})$ size and slightly larger stretch (2t + ε). Thorup and Zwick [32], improved the result of Cohen [31] and gave a (2t - 1)-approximate

distance oracle of size $O(tn^{1+1/t})$, query time $O(t)$, and preprocessing time $O(tmn^{1/t})$. They also showed that the union of shortest path trees of clusters C(v) computed in the preprocessing step is a (2t - 1)-spanner of the input graph G. Since the total size of the shortest path trees is $O(tn^{1+1/t})$, the size of the spanner is thus worse than the greedy spanner by a factor of t. However, its construction time is $O(tmn^{1/t})$ which is significantly faster than that of greedy.

## 2.3 The Baswana-Sen Spanner

This section covers a spanner construction of Baswana and Sen. Their running time is only $O(tm)$ and achieves the same size/stretch tradeoff as Thorup and Zwick. Baswana and Sen.'s construction is randomized; a deterministic algorithm with the same bounds was given by Roditty, Thorup, and Zwick [33].

Baswana and Sen [34] have suggested the linear time algorithm for computing sparse spanner in weighted graphs and the working of their 3-spanner algorithm is explained in this section with an example. There are three phases in this algorithm. Phase 1 randomly selects $\lceil\sqrt{n}\rceil$ vertices from set V that vertex refereed as center of cluster. After Phase 1, vertices are divided in two parts one is center vertices and others are non center vertices. A set of centers is called R and function f(b) gives the center of cluster on which vertex b belongs.

---

**Algorithm 2.2** Random clustering

---

**Procedure** Design_cluster (G, R)

1. ***for*** all vertex u ϵ V – R ***do***
2. ***if*** ∃ v ϵ R and edge e(u, v) ∉ E ***then***
3. $E_S \leftarrow E_S$ U {all adjacent edges on u}
4. f(u) ← 0
5. ***if*** ∃ v ϵ R and edge e(u, v) ϵ E ***then***
6. $E_S \leftarrow E_S$ U {e(u, C(u, R))}
7. ***for*** all adjacent vertex x on vertex u and w(u, x) < w(u, C(u, R)) ***do***
8. $E_S \leftarrow E_S$ U {e(u, x)}
9. ***if*** f(u) = f(v) ***then***
10. $E_S \leftarrow E_S$ - {e(u, v)} (Removing intra cluster edge)

---

Phase 2 performs the algorithm 2.2 in which C(b, R) is a closest adjacent center to vertex b from set R. After performing the second phase, edges are divided into two parts. One are spanner edges $E_S$ and others are not spanner edges $E_1$ where $E_1 \subset (E - E_S)$. Let $V_1$ be the set of vertices that end points of edges present in set $E_1$.

---

**Algorithm 2.3** Inter cluster joining

---

**Procedure** Join_intercluster (G, $V_1$, $E_1$)

1. ***for*** all vertex u $\epsilon$ $V_1$ ***do***
2. ***if*** $\exists$ $E_1$(u, *c*) ***then***
3. $E_S \leftarrow E_S \cup$ {Least weight edge $E_1$(u, *c*)}

---

Phase 3 makes the connection between clusters by executing algorithm 2.3. This algorithm takes input $V_1$ and $E_1$ and gives the spanner graph as output. Let *c* be any cluster of graph G in algorithm.

### 2.3.1 Verifying the Size of Spanner

*A. Verify the number of spanner edges chosen in Phase 2*

In the Phase 1, there are $\lceil\sqrt{n}\rceil$ vertices randomly selected and in phase 2, for each vertex u $\epsilon$ V, the expected number of adjacent edges with weight less than that of e(u, C(u, R)) is $\sqrt{n}$ which are inserted into $E_S$. So the size of spanner after phase 2 is $O(n^{3/2})$.

*B. Verify the number of spanner edges chosen in Phase 3*

In phase 3, $O(n*|R|)$ edges are inserted into $E_S$ because $\lceil\sqrt{n}\rceil$ vertices are selected in phase 1. So, the final size of spanner set $E_S$ is $O(n^{3/2})$.

**Lemma 2.2:** for an edge $\epsilon$ E that is not inserted in spanner G(V, $E_S$), there is a path exist between the end point of that edges which total weight is not more than that edge weight and length is at most 3 edges [34].

### 2.3.2 Execution Time of an Algorithm

An adjacency list representation of a graph is a collection of unordered lists. Each vertex in the graph maintains its own list. Each list describes the set of neighbors of its vertex. Cormen et al. suggest an implementation in which the vertices are represented by index numbers [35]. Here the vertices are numbers 1 to *n* and their representation

uses an array indexed by vertex number and each vertex in the each array cell points to a singly linked list of the neighboring vertices of that vertex.

In this representation, the nodes of the singly linked list may be seen as edge objects; Even so, they do not have the full information about each edge (they only keep one of the two endpoints of the edge) and in undirected graphs there will be two different linked list nodes for each edge (one within the lists for each of the two endpoints of the edge). Set of centers R can be selected in $O(n)$ time and also closest adjacent center to vertex u $\epsilon$ V from R can be calculated by single pass of adjacency list of vertex u.

In algorithm 2.2 of phase 2, by traversing the adjacency list for each vertex u, all the edges adjacent with weight less than that of the edge e(u, C(u, R)) can be selected and inserted into spanner G(V, $E_S$).

In algorithm 2.3 of phase 3, every vertex has assigned the index number from 1 to $n$ and takes an array L[1…$n$] with initial null weight. Now for each vertex u from $V_1$, find the adjacent edge on u from set $E_1$. Mark with weight at index of corresponding end point vertex. Before that check if any other vertex is marked with weight less than that of edge which belongs to same cluster.(Identify cluster center using f(u) function). If any found then mark this vertex index with infinite weight. Now, trace the array L from 1 to $n$ and ignore null and infinite weight. So vertex corresponding indexed and its least edge weight can be got in linear time. Consider that any undirected weighted 3-spanner graph can be computed of size $O(n^{3/2})$ in $O(m)$ time.

### 2.3.3 Example on 3-spanner Algorithm

Let G(V, E) have $n$ = 15 vertex and edges with their corresponding weight are shown in Figure 2.4. Here the value of stretch factor **t** is taken as 3.

In Phase 1, there are 4 (i.e.$\lceil\sqrt{15}\rceil$) centers randomly selected. Let R = {b, p, l, d} and Phase 2 checks for each vertex V-R is shown in Figure 2.5. Dark edge represents that vertex is directly connected with center which inserted into cluster and dotted edge represent the edge whose weight is less than or equal to that clustered edge.

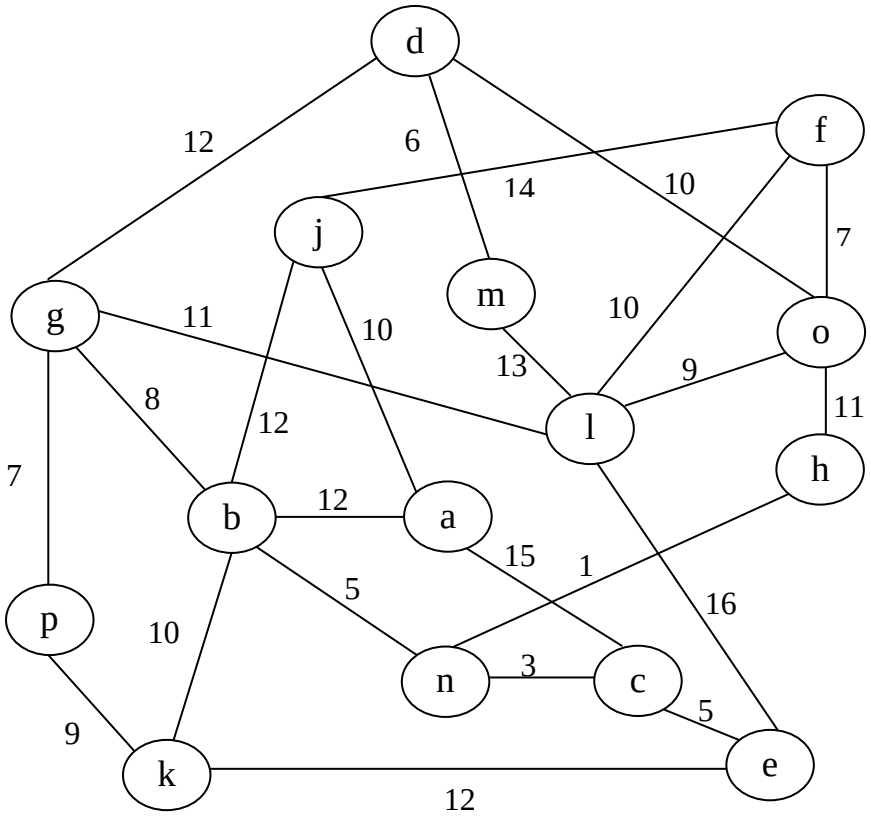
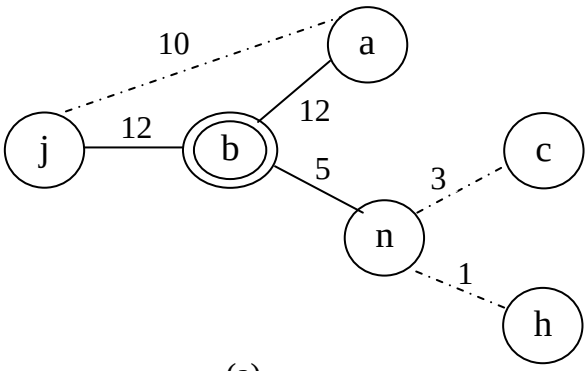


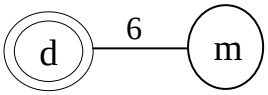

Figure 2.4 An undirected weighted graph

- For vertex **a** and **j**



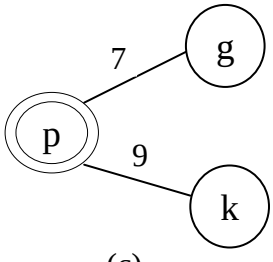

(a)

For vertex **m**



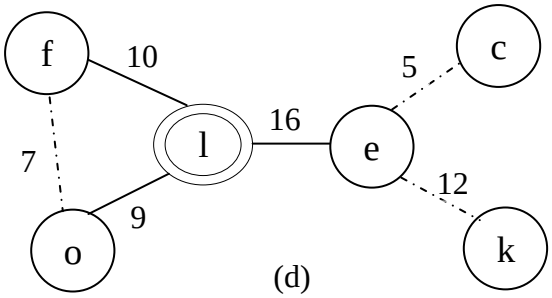

(b)

- For vertex **g** and **k**




(c)

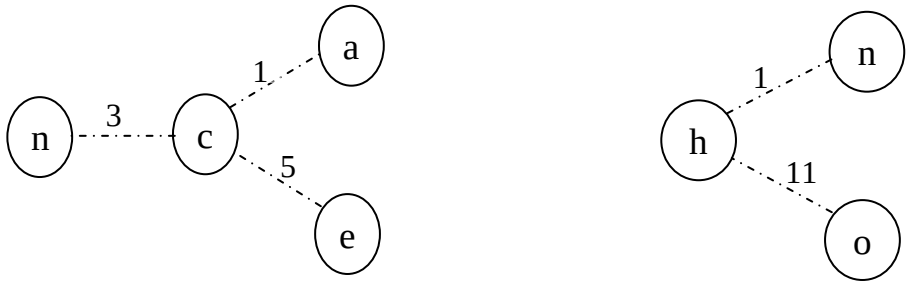
For vertex **e, f** and **o**




(d)

Figure 2.5 Clustered vertices and their corresponding edges in (a), (b), (c) and (d)

➢ For unclustered vertex **c** For unclustered vertex **h**




(a) (b)

Figure 2.6 Unclustered vertices and their corresponding edges in (a) and (b)

The intra cluster edges like (a, j) and (f, o) are removed from $E_S$ and after phase 2, the graph is divided into clusters shown in Figure 2.7.

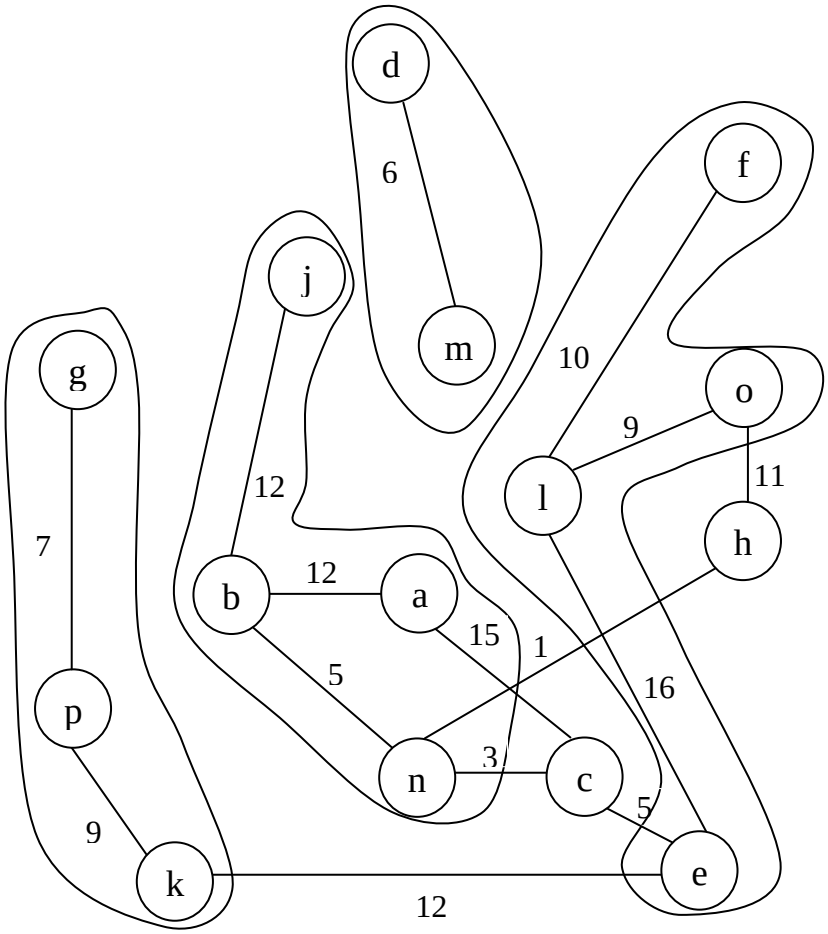



Figure 2.7 A clustered graph after Phase 2

In Phase 3, Let here $E_1$ = {(d, o), (d, g), (j, f), (b, k), (b, g), (g, l), (k, b), (l, m)} and so $V_1$ = {d, o, g, j, f, b, k, l}

This phase makes inter cluster linking and find the least edge weight cluster to which it can connect and follow the Lemma 2.2. Here p centered cluster can connect with b centered cluster with two edges (g, b) and (k, b) with their weight 8 and 10 respectively. Because of least weight condition at line 3 in algorithm 2.3, the edge (g, b) is inserted into $E_S$ while other one is discarded. Same case occurs between edges (d, o) and (m, l) and least weight edge (d, o) is selected for spanner. But edges (b, n) and (j, f) have no confliction so both are inserted into $E_S$. After the performing the phase 3, the given graph is converted into 3-spanner graph, is shown in Figure 2.8.

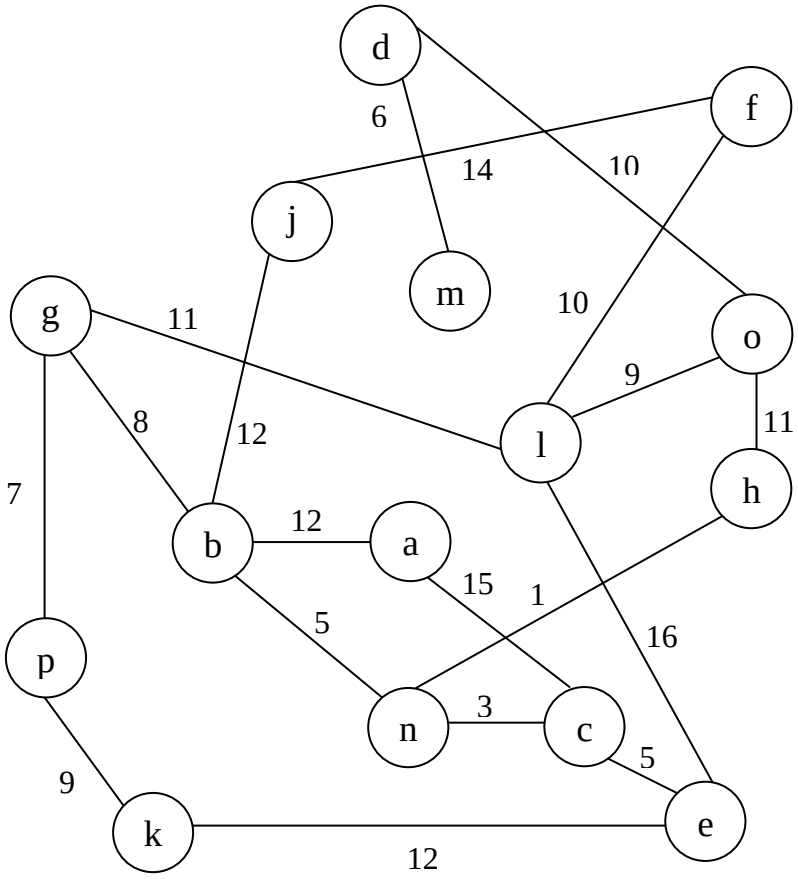



Figure 2.8 The Final 3-spanner graph

This section describes the linear time algorithm for computing 3-spanner graph of undirected weighted graph and size of spanner graph cannot be exceeding the $O(n^{3/2})$. This algorithm can also be applied on distributed environment as well as in communication networks.

Same way, Baswana and Sen [36] present a randomized algorithm that takes expected linear time $O(tm)$ for computing (2t - 1)-spanner of optimal $O(tn^{1+1/t})$ size for a given weighted undirected graph.

## 2.4 Spanner of Directed Graphs

Above all the sections have only considered spanners for undirected graphs. The Spanners for directed graphs is also possible to define. Even so, it can be shown that for any $n \geq 2$ and any $t \geq 1$, there is an *n*-vertex directed graph for which any t-spanner requires $O(n^2)$ edges. Consider the complete bipartite graph with *n/2* vertices in each set and where all edges are directed from one set to the other; every edge is needed in any t-spanner of the graph. Hence, in general, nothing interesting can be said about the size/stretch tradeoff of spanners of directed graphs.

## 2.5 $(\alpha, \beta)$-spanners

The definition of multiplicative spanners can be generalized to also allow additive approximation. For values $\alpha \geq 1$ and $\beta \geq 0$ and an undirected edge-weighted graph G(V, E), an $(\alpha, \beta)$-spanner of G is a subgraph G′(V, $E_S$) spanning all vertices such that for all p, q $\epsilon$ V,

$$d_{G'}(p, q) \leq \alpha\, d_G(p, q) + \beta$$

Again, it also has the inequality $d_G(p, q) \leq d_{G'}(p, q)$ for all p, q $\epsilon$ V since $E_S \subset E$. Note that setting $\beta = 0$ gives a multiplicative spanner. If $\alpha = 1$ then refer to G′ as an additive $\beta$-spanner.

Several results are known for $(\alpha, \beta)$-spanners. The result of Elkin and Peleg [37] shows that any unweighted and undirected graph G with *n*-vertices has an additive 2-spanner of size $O(n^{3/2})$. They have also shown that G has an additive 6-spanner of size $O(n^{4/3})$.

# Chapter 3

# Problem Statement

## 3.1 Gap and Scope Analysis

Althofer et al [22] gave the first algorithm for computing a t-spanner for weighted graphs. Their algorithm is similar to kruskal's algorithm for computing a minimum spanning tree. The best known implementation of this algorithm has running time $O(m.n^{1+2/(t+1)})$. Cohen [31] presented a randomized algorithm with $O(t.m.n^{2/(t+1)})$ expected running time for computing a spanner of $O(t.n^{1+2/(t+1)})$ size and slightly larger stretch $(t + \varepsilon + 1)$. Thorup and Zwick [32] have improved the result of Cohen [31]. The best time bound for computing a t-spanner of a weighted graph with $n$ vertices and $m$ edges is $O(m + n)$, and is given by Baswana and Sen [36]. Their algorithm is randomized and computes spanners of size $O(t.n^{1+2/(t+1)})$. All these existing algorithm for computing spanners in undirected weighted graphs require computation of shortest distance information between many pairs of vertices [22], or computing shortest path trees from a set of $(n^{2/(t+1)})$ vertices [32]. Since there is a bound of $O(m)$ on the best known algorithm for computing a shortest path tree, the earlier algorithms for spanners are unlikely to achieve a running time of $O(m)$.

All existing algorithms assumed that all edge weights are distinct. The clustering method used in existing algorithm that randomly selects the centers of cluster. Due to this randomness, high degree vertex may not be selected and it does improper clustering. Those algorithms do not have any mechanism that handles connected component and unclustered vertices.

Routing messages between pairs of routers is a primary activity of any communication network of routers. Since the cost of sending a message is roughly proportional to the number of links the message has to traverse, it is desirable to route message along short paths. A straightforward approach is to store a complete routing table in each of the $n$ routers (or nodes) of the network, specifying for each destination the next link in some shortest path to that destination. Although this solution guarantees an optimal (shortest path) routing, it is too expensive for large systems since it requires an $(n - 1)$ entry table in each router, or a total of $O(n^2)$ memory space. Thus, a fundamental question in the design of large scale communication network is

whether there exists a routing scheme that produced efficient routes and requires substantially less memory space in the individual routes.

## 3.2 Proposed Objective

This thesis work has proposed a simple algorithm for constructing sparse t-spanners for arbitrary undirected weighted graphs with new clustering techniques based on two parameters radius and degree of node.

This thesis also explains fully dynamic algorithm for maintaining t-spanner of undirected weighted graphs under a sequence of update operations like insertion and deletion of links. This algorithm is applied on communication networks to optimize routing table space and also for good routing scheme by taking weight as congestion load of network link between two routers.

## 3.3 Methodology

The proposed algorithm is divided into six phases.

- It takes an arbitrary weighted undirected graph as input and provides t-spanner graph as output with a given stretch factor t.
- Throughout the thesis, unless stated otherwise, assume that the undirected graph has the augmented adjacency list representation.
- If the initial graph has simple adjacency list representation, augmented adjacency list can be get from its simple representation in $O(m)$ processing time.
- Graph is divided into clusters based on the degree of its node and radius.
- Finds all intermediate nodes from center up to distance radius R.
- Performs inter cluster linking initiated by clustered node but not by center and also initiated by center (Details given in chapter 5).
- Remove the link from $L_S$ which is not necessary means that after removal of link graph still follows t-spanner property.
- Handle connected component and remove unwanted link from unclustered node means that if this links remove from $L_S$ still graph follows the t-spanner property.
- Design routing tables of spanner network. It will be optimal in terms of space and time.
- This routing table can be easily interchange between routers in the network using any appropriate dynamic routing protocol that updates routing table with in some time interval.

# Chapter 4

# Preliminaries and Clustering Technique

## 4.1 Preliminaries

Throughout the thesis, deal with the graph G(N, L) which is undirected and weighted. Assume that the weights of all the links are positive. In the thesis, weight and distance have been used interchangeably. Similarly graph and network means the same things, wherever there have been used. In this thesis, all the diagrams and figures used are just graphical representation of graph (or network) and **should not be taken as actual geometric diagrams.**

***t-spanner property*:** if there is exist link between a node with other node in graph G and its weight multiplied by t is less than the sum of weights of links between these nodes via other nodes in graph G′ then this link will be added in G′. This property should be satisfied every time a link is added to spanner graph G′.

### 4.1.1 Data Structure

In this thesis, undirected weighted graph G(N, L) is represented with augmented adjacency list [34] wherein for each link *l(p, q)* associated with both node p and q have address of each other in list. The fields included in list are shown below in Figure 4.1.

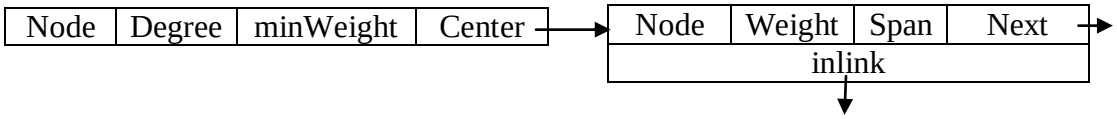



Figure 4.1 Adjacency list field representation

**Example: Adjacency representation of Undirected weighted graph**

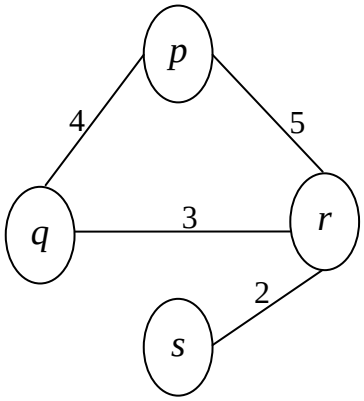



Figure 4.2 An undirected weighted graph

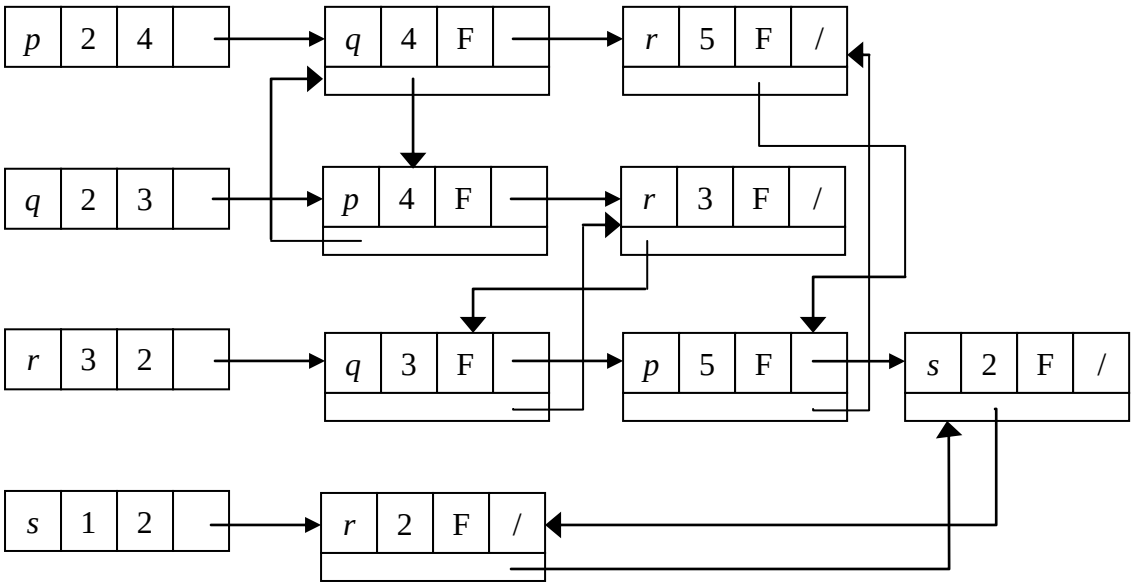



Figure 4.3 An augmented adjacency list representation

Here graph is undirected therefore link for example *l(p, q)* appears twice in list. First link is from p to q and second link is from q to p. This representation is useful in deletion of link *l(p, q)*. In that case, links should be deleted from adjacency list of p and q both. But here link *l(p, q)* can be easily deleted from q while processing adjacency list of p using *inlink* field in constant time.

In above Figure 4.3, the fields are described as follows:

*Node*: Name of Node in graph

*Degree(p)*: Number of adjacent nodes from node p, here 2

*Weight(p, q*): distance between nodes *p* and q, here 4

*minWeight(p)*: minimum distance among all the links those are adjacent to node *p*, here 4

*Center(p)*: center node of cluster to which p belongs, here this field is left blank. This can be filled after clustering technique is applied on graph

*Span(p, q)*: This is Boolean field. It indicates that link *l(p, q)* is included in spanner graph G′(N, $L_S$) or not. If this field is *True* means *l(p, q)* $\epsilon$ $L_S$ otherwise by default is *False.* This will be decided after applying t-spanner algorithm.

*inlink*: Graph is undirected so this field is useful for constant processing time.

*Link_list(p)*: gives the list of nodes that are adjacent to node *p* from adjacency list until *Next* field is not null

### 4.1.2 Notations

The following notations shall be used throughout the thesis in the context of graph G(N, L).

N: set of nodes of graph G

*n*: Number of nodes in graph G

L: set of links present in graph G

*m*: Number of links in graph G

$L_S$: set of links present in Spanner Graph G′

R: Radius of cluster (one of the parameter for clustering)

$C_N$: set of clustered nodes

$U_N$: set of unclustered nodes

$C_C$: set of cluster centers

*Weight(p,* $C_C$): distance from node *p* with any cluster centers

Spandistance(p, *q)*: distance from node p to node q by searching links in $L_S$

$d_{min}$= Lowest degree of Graph (in best case, 0 for singleton graph and 1 for connected graph)

$d_{max}$= Highest degree of Graph (in worst case, *n*-1)

## 4.2 Clustering Technique

A **cluster** is a subset of nodes. A **clustering *C***, is union of clusters. Each cluster will have a unique node which will be called its **center**. Set cluster center of node *p* in *Center(p)* field of node *p* if *p* belongs to any cluster otherwise set *Center(p)* = 0 if *p* is unclustered (does not belong to any cluster)

### 4.2.1 Efficient Construction and Maintenance

Two parameters have been used for clustering: **Degree** of node and **Radius R.** With this method, it is guarantee that the distance between any two nodes cannot exceed R or t times of the distance in G.

Select cluster center based on Degree of node:

There will be constructed at most $\lceil\sqrt{n}\rceil$ clusters. Partial quick sort [38] is applied on the degree of all the nodes; it gives the output in decreasing order of degree of nodes. Complete sorting is not required because need only top $\sqrt{n}$ nodes of higher degrees. Nodes may be arranged in any order. Time complexity of this approach will be

$O(k \,.\, n)$, where $k$ is the number of iterations required to get top $\sqrt{n}$ degree nodes. However practically it will run much faster.

Clustering based on Radius R:

There are *minWeight()* present in adjacency list, now select Radius **R = MAX (*minWeight(p), minWeight(q), minWeight(r),* …) for all *p, q, r,* … ϵ N**. Radius R is computes in $O(n)$ expected time.

---

**Algorithm 4.1** Clustering

---

**Procedure** Design_cluster (G, t, R)

1. $L_S \leftarrow \emptyset$ // initially empty
2. ***for*** all the nodes $x \in N$ ***do***
3. ***if*** node $x \in C_C$ ***then***
4. *Center(x)* ← *x*
5. ***else if*** ∃ node $y \in C_C$ such that *Weight(x, y)* ≤ R ***then***
6. *Center(x)* ← *y*
7. *Span(x, y)* ← True
8. $L_S \leftarrow L_S \cup l(x, y)$
9. ***else***
10. *Center(x)* ← 0

---

In case of tie for example, if there are multiple centers at distance *Weight(p,* $C_C$) ≤ R from node *p,* it is the nearest center that appears first in adjacency list that becomes *Center(p)*. If still tie is not broken, check degree of centers and largest one becomes *Center(p)*. Here there are $\sqrt{n}$ clusters and checking for remaining ( $n - \sqrt{n}$ ) nodes. So time complexity of algorithm 4.1 is also $O(\sqrt{n}.n)$.

To maintain clustering *C,* it takes $O(2i.\sqrt{n})$ update time over any sequence of *i* update operations like insertion or deletion of links in graph.

# Chapter 5

# Algorithm of Graph Spanner

## 5.1 Algorithm of t-spanner for Weighted Graph

After performing clustering algorithm 4.1, There will be three types of nodes in graph shown below in Figure 5.1: Clustered nodes set $C_N$ and unclustered nodes set $U_N$ and Cluster center set $C_C$ ($C_C$ is subset of $C_N$).

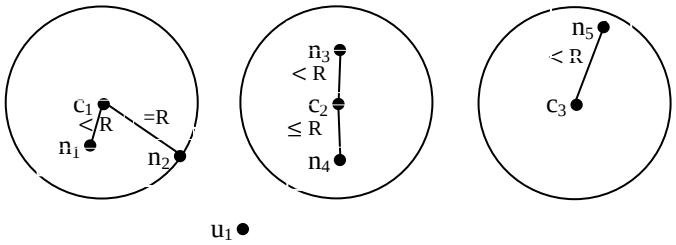



Figure 5.1 Node representation after applying clustering algorithm

Assume that graph G(N, L) in Figure 5.1 where N = {$c_1$, $n_1$, $n_2$, $c_2$, $n_3$, $n_4$, $c_3$, $n_5$, $u_1$} and $n$ = 9, so there are 3 clusters and after applying clustering, set $C_N$ = {$c_1$, $n_1$, $n_2$, $c_2$, $n_3$, $n_4$, $c_3$, $n_5$}, set $C_C$ = {$c_1$, $c_2$, $c_3$} and set $U_N$ = {$u_1$}

*In all the phases, if link from node x to node y is processed then link from node y to node x is not process further because all the links are bidirectional.*

### 5.1.1 Overview of all the Phases of an Algorithm

In proposed algorithm, there are six phases required to design the t-spanner graph from given weighted undirected graph.

Phase 1, finds all intermediate nodes from center up to distance radius R and phase 2 performs inter cluster linking initiated by clustered node but not by center while phase 3 tries to connect center with other center. If found this center to center link, then check center to any node belongs in that cluster which have minimum distance with that center add that link in to $L_S$ and If not found center to center link then check which node have minimum distance, add that link in to $L_S$.

Phase 4 removes the link from $L_S$ which is not necessary means that after removal of link graph still follows t-spanner property. Phase 5 handles connected components [39]. Phase 6 removes unwanted link from unclustered node means that if this links remove from $L_S$ still graph follows the t-spanner property.

### 5.1.2 Flowchart of all the Phases of an Algorithm

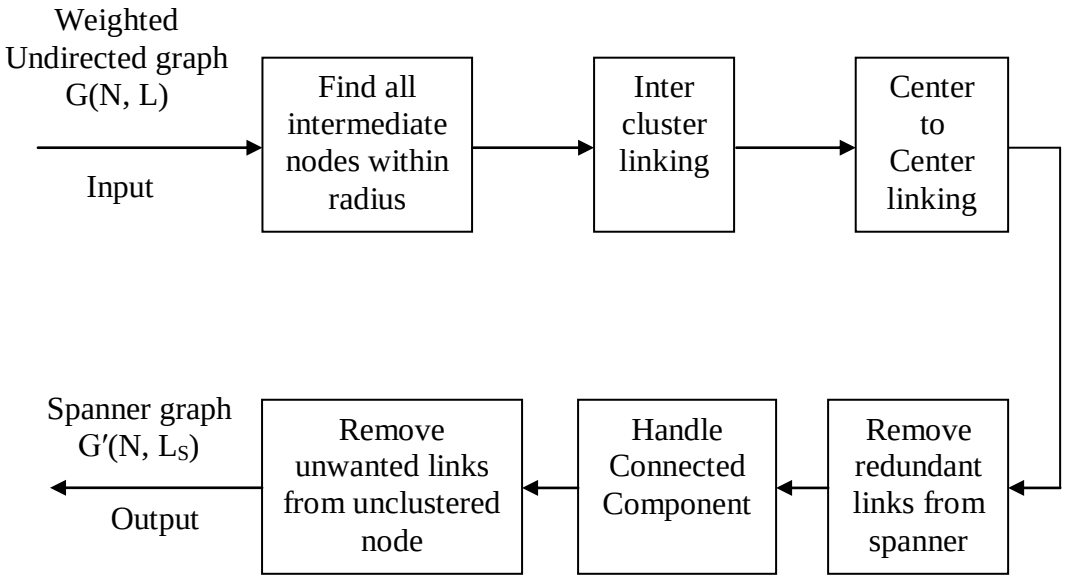



Figure 5.2 Flowchart of all the Phases of t-spanner algorithm

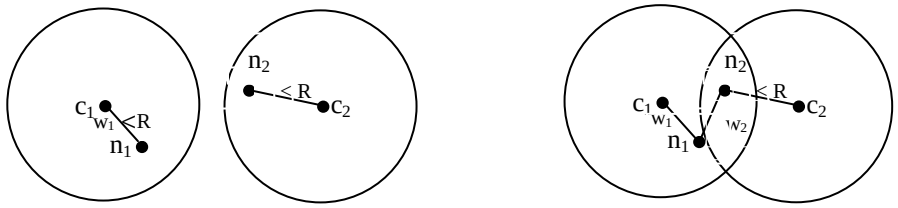
## 5.2 Phase 1: Finding all Intermediate Nodes from Center up to Distance Radius R

It can be seen in Figure 5.1 that distance between centers to node is less than R. so to utilize the remaining distance (R – (distance center to node)) applies this phase on graph. Let's take different cases of graph and check how this phase handles those cases. At the time of clustering, the node will go to the cluster whose center lies at a distance ≤ R from the node. In case there are two or more choices for cluster center selection, center with minimum distance from the node would be selected. Sometimes it might happen that a node linked to its center is at a distance less than R, in that case radius of cluster will not be fully utilized. There may be other clustered nodes (not in the same cluster) or unclustered nodes present in the graph that could be connected with this node but not with this node's center (due to radius constraint). So other nodes linked with this node whoever has distance ≤ (R - (distance from this node to its center)) tries to utilize the full radius as much as possible by linking to this node. During this linking either the two or more clusters overlap (Case 1) or just touch each other (Case 2) with common node on their boundary.

Unclustered node that was not covered by any other clustered node as in case 1 or case 2, because its overall distance from center was >R. Hence it can be connected to

any clustered node or unclustered node whose distance from this node is ≤ R. Consider the case when two or more than two such clustered nodes exist, it will be connected to the one whose distance is minimum(Case 3). The details of all the cases are given below.

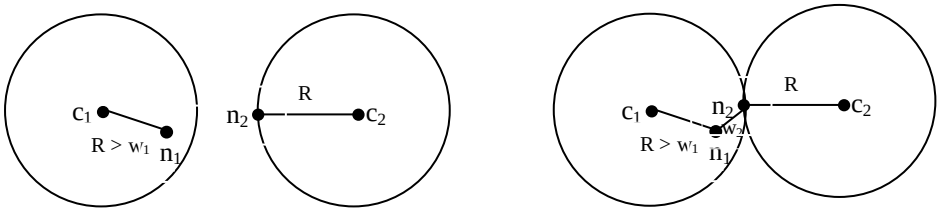
**Case 1:** In Figure 5.3(a), assume that *Weight(*$c_2$, $n_2$*)* < R and *Weight(*$c_1$, $n_1$*)* = $w_1$; ($w_1$ < R) so remaining distance from node $n_1$ is (R - $w_1$). Now check for all node $n_2$ from $n_1$ where $n_1$ and $n_2$ are not in same cluster and $w_2$ = *Weight(*$n_1$, $n_2$*)* ≤ (R - $w_1$). If found such type of node then add this link into $L_S$, so it ensure that **($w_1$ + $w_2$) ≤ R** and clusters are **overlapped to each other** shown in Figure 5.3(b).




(a) (b)

Figure 5.3 For t-spanner algorithm Case 1 (a) After Clustering (b) After Phase 1

**Case 2:** In Figure 5.4(a), assume that *Weight(*$c_2$, $n_2$*)* = R and *Weight(*$c_1$, $n_1$*)* = $w_1$; ($w_1$ < R) so remaining distance from node $n_1$ is (R - $w_1$). Now check for all node $n_2$ from $n_1$ where $n_1$ and $n_2$ are not in same cluster and $w_2$ = *Weight(*$n_1$, $n_2$*)* ≤ (R - $w_1$). If found such type of node then add this link into $L_S$, so it ensure that **($w_1$ + $w_2$) = R** and clusters are **touches each other** shown in Figure 5.4(b).




(a) (b)

Figure 5.4 For t-spanner algorithm Case 2 (a) After Clustering (b) After Phase 1

**Case 3:** In Figure 5.5(a), assume that *Weight(*$c_1$, $n_1$*)* = $w_1$ < R so remaining distance from node $n_1$ is (R - $w_1$). Now check for all node $n_2$ from $n_1$ where $n_1$ and $n_2$ are not in same cluster and $w_2$ = *Weight(*$n_1$, $n_2$*)* ≤ (R - $w_1$). If found such type of node then add this link into $L_S$, but here $n_2$ is not found. Now start checking from all unclustered node $u_1$ such that distance from $u_1$ to that clustered node is ≤ R. Here it is found that *Weight(*$u_1$, $n_1$*)* ≤ R and ensure that **$w_2$ ≤ R**


(a) (b)

Figure 5.5 For t-spanner algorithm Case 3 (a) After Clustering (b) After Phase 1

Algorithm of Phase 1 that handles the above all three cases are shown below.

---

**Algorithm 5.1** Finding all intermediate nodes from center up to distance radius R

---

**Procedure** Collect_indermediate_nodes (G, t, R, $C_C$, $U_N$)

1. Traverse_linklist ← ∅ // this set contains list of links that should be traverse
2. Traverse_distlist ← ∅ // set of weight corresponding link in Traverse_linklist
3. ***for*** all nodes $x \in C_C$ ***do*** // this ***for*** loop handles Case 1 and Case 2
4. ***for*** all nodes $y \in$ *Link_list(x)* and *Center(y)* = $x$ ***do***
5. ***if*** *Weight(x, y)* < R ***then***
6. Traverse_linklist ← Traverse_linklist ∪ *l(x, y)*
7. Traverse_distlist ← Traverse_distlist ∪ *Weight(x, y)*
8. ***for*** all nodes $x \in U_N$ ***do*** // this ***for*** loop handles Case 3
9. ***for*** all nodes $y \in$ *Link_list(x)* and *Weight(x, y)* ≤ R ***do***
10. Traverse_linklist ← Traverse_linklist ∪ *l(x, y)*
11. Traverse_distlist ← Traverse_distlist ∪ *Weight(x, y)*
12. ***for*** all links *l(x, y)* ∈ Traverse_linklist ***do***
13. Findnode_withinR(*x*, R – Traverse_distlist *(l(x, y))*)

**Procedure** Findnode_withinR(*x*, Distleft) // Intra cluster link is not allowed now

1. ***for*** all nodes $y \in$ *Link_list(x)* and *Weight(x, y)* ≤ Distleft
2. and *Center(x)* ! = *Center(y)* ***do***
3. *Span(x, y)* ← True
4. $L_S \leftarrow L_S \cup$ *l(x, y)*
5. Distleft ← Distleft - *Weight(x, y)*
6. ***if*** minWeight*(y)* < Distleft and Distleft > 0 ***then***
7. Traverse_linklist ← Traverse_linklist ∪ *l(x, y)*
8. Traverse_distlist ← Traverse_distlist ∪ Distleft

---

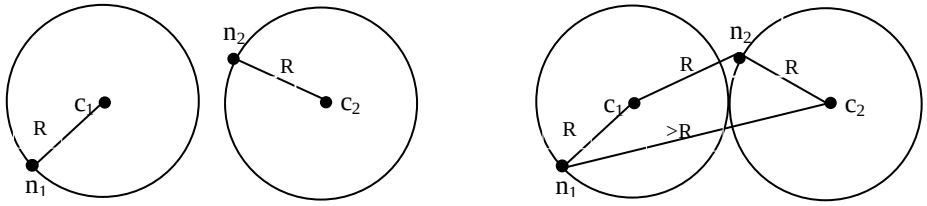
The algorithm 5.1 of Phase 1 handles Case 1 and Case 2 in time $O(\sqrt{n}\,.\,d_{max})$ and Case 3 in time $O(u\,.\,d_{min})$ (where $u$ is number of unclustered nodes) in worst case. Procedure Findnode_withinR takes $O(d_{max})$ and it is called $n$ times. Total time complexity of Phase 1 is $O(n\,.\,d_{max})$ in worst case and it takes $O(\sqrt{n}\,.\,d_{max})$ in average case.

## 5.3 Phase 2: Inter Cluster Linking initiated by Clustered Node but not by Center

In this phase, inter cluster linking initiated by clustered node will take place. Clustered node will try to connect with other cluster nodes if it satisfies t-spanner property i.e. if there exist link between a clustered node with other cluster node in graph G and its weight multiplied by t is less than the sum of weights of links between these nodes via other nodes in spanner graph G′ then this link will be added in graph G′. After this phase, if there is exist a link that is not necessary required in the spanner graph G′ then it will be removed in phase 4. Assume that t = 2 for all cases and details are given below.

For Case 1, assume that graph is G(N, L) where N = {$c_1$, $n_1$, $c_2$, $n_2$} and L = {($c_1$, $n_1$), ($n_1$, $c_2$), ($c_2$, $n_2$), ($c_1$, $n_2$)} and weight of links are W= {R, >R, R, R} respectively and spanner graph G′(N, $L_S$) where $L_S$ = {($c_1$, $n_1$), ($c_2$, $n_2$)}

**Case 1:** Apply algorithm 5.2 on graph shown in Figure 5.6(a) so $x \in \{n_1, n_2\}$. While processing $x = n_1$ and $y = c_2$, check *Weight(*$n_1, c_2$*)* + *Weight(*$n_1, c_1$*)* is (>R + R) ≅ > 2R ≅ 3R and t * *Weight(*$c_2, c_1$*)* is 2 * ∞. Finally condition at line 4 is 3R < ∞. Hence following t-spanner property and add link from $n_1$ to $c_2$ in to $L_S$ (Shown in Figure 5.6(b)). Note that if here t = 3 taken then link $n_1$ to $c_2$ will not be present in $L_S$. Same as while processing $x = n_2$ and $y = c_1$, check *Weight(*$n_2, c_1$*)* + *Weight(*$n_2, c_2$*)* is (R + R) = 2R and t * *Weight(*$c_1, c_2$*)* is 2 * ∞. Finally condition at line 4 is 2R < ∞. Hence following t-spanner property and add link from $n_2$ to $c_1$ in to $L_S$ (See Figure 5.6(b)).




(a) (b)

Figure 5.6 For t-spanner algorithm Case 1 (a) After Phase 1 (b) After Phase 2

For Case 2, assume graph is G(N, L) where N = {$c_1$, $n_1$, $n_2$, $c_2$, $n_3$} and L = {($c_1$, $n_1$), ($n_1$, $c_2$), ($c_2$, $n_2$), ($c_1$, $n_2$), ($c_2$, $n_3$)} and weight of links are W= {R, >R, R, R, <R} respectively and G′(N, $L_S$) where $L_S$ = {($c_1$, $n_1$), ($c_1$, $n_2$), ($c_2$, $n_3$)}

**Case 2:** Apply algorithm 5.2 on graph shown in Figure 5.7(a) so $x \in \{n_1, n_2, n_3\}$. While processing $x = n_1$ and $y = c_2$, check *Weight(*$n_1$, $c_2$*)* + *Weight(*$n_1$, $c_1$*)* is ( > R + R ) ≅ >2R ≅ 3R and t * *Weight(*$c_2$, $c_1$*)* is 2 * ∞. Finally condition at line 4 is 3R < ∞. Hence following t-spanner property and add link from $n_1$ to $c_2$ in to $L_S$. While processing $x = n_1$ and $y = n_3$, check *Weight(*$n_1$, $n_3$*)* + *Weight(*$n_1$, $c_1$*)* is (∞ + R ) ≅ ∞ and t * *Weight(*$n_3$, $c_1$*)* is 2 * ∞. But here *l(*$n_3$, $c_1$*)* is not present in L, so finally link from $n_1$ to $n_3$ is not added in to $L_S$.

Same as while processing $x = n_2$ and $y = c_2$, check *Weight(*$n_2$, $c_2$*)* + *Weight(*$n_2$, $c_1$*)* is (R + R) = 2R and t * *Weight(*$c_2$, $c_1$*)* is 2 * ∞. Finally condition at line 4 is 2R < ∞. Hence following t-spanner property and add link from $n_2$ to $c_2$ in to $L_S$. When $x = n_2$ and $y = n_3$, check *Weight(*$n_2$, $n_3$*)* + *Weight(*$n_2$, $c_1$*)* is (∞ + R ) ≅ ∞ and t * *Weight(*$n_3$, $c_1$*)* is 2 * ∞. But here *l(*$n_3$, $c_1$*)* is not present in L, so finally link from $n_2$ to $n_3$ is not added in to $L_S$. (Shown in Figure 5.7(b))

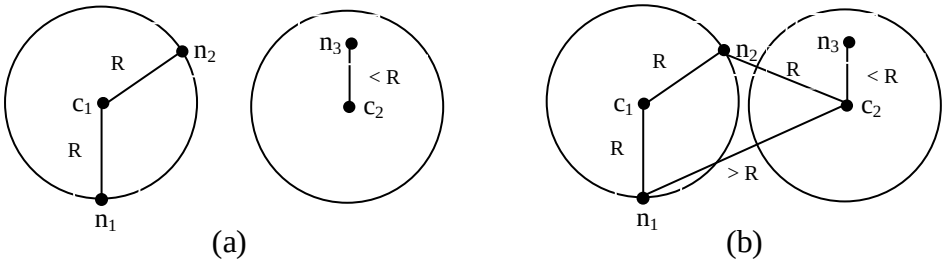
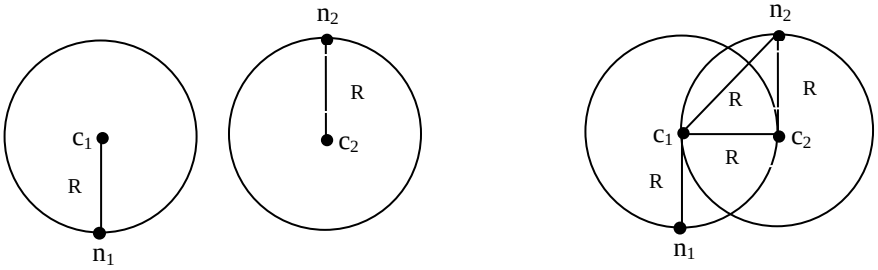



Figure 5.7 For t-spanner algorithm Case 2 (a) After Phase 1 (b) After Phase 2

For Case 3, assume graph G(N, L) where N = {$c_1$, $n_1$, $c_2$, $n_2$} and L = {($c_1$, $n_1$), ($c_1$, $c_2$), ($c_1$, $n_2$), ($c_2$, $n_2$), ($c_2$, $n_1$), ($n_1$, $n_2$)} and weight of links are W = {R, R, R, R, R, R} respectively and G′(N, $L_S$) where $L_S$ = {($c_1$, $n_1$), ($c_2$, $n_2$)}

**Case 3:** Perform algorithm 5.2 on graph shown in Figure 5.8(a) so $x \in \{n_1, n_2\}$ and while processing $x = n_1$ and $y = n_2$, check *Weight(*$n_1$, $n_2$*)* + *Weight(*$n_1$, $c_1$*)* is (R + R) = 2R and t * *Weight(*$n_2$, $c_1$*)* is 2 * R. Hence following t-spanner property and add link from $n_2$ to $c_1$ in to $L_S$. While processing $x = n_1$ and $y = c_2$, check *Weight(*$n_1$, $c_2$*)* + *Weight(*$n_1$, $c_1$*)* is ( R + R ) = 2R and t * *Weight(*$c_2$, $c_1$*)* is 2 * R. So finally link from $c_2$ to $c_1$ is added in to $L_S$. Here $x = n_2$ and $y = n_1$ is not processed because it already proceeded (link is bidirectional).

Same as while processing $x = n_2$ and $y = c_1$, check *Weight(*$n_2, c_1$*)* + *Weight(*$n_2, c_2$*)* is ( R + R ) = 2R and t * *Weight(*$c_1, c_2$*)* is 2 * R. Hence following t-spanner property and add link from $c_1$ to $c_2$ in to $L_S$. It is already present in $L_S$ (Shown in Figure 5.8(b)).




(a) (b)

Figure 5.8 For t-spanner algorithm Case 3 (a) After Phase 1 (b) After Phase 2

Here link from $n_2$ to $c_2$ is present but if it is not present still graph follows t-spanner property. This link removes in the Phase 4. All the above cases can also be checked for t > 2. The algorithm of Phase 2 that handles all the above three cases is given in algorithm 5.2.

---

**Algorithm 5.2** Inter cluster linking initiated by clustered node but not by center

---

**Procedure** Link_InterclusterBy_cnode (G, t)

1. ***for*** all nodes $x \in (C_N - C_C)$ ***do***
2. ***for*** all nodes $y \in$ *Link_list(x)* and *Center(x)* != *Center(y)* ***do***
3. ***if*** (*Weight(x, y)* + *Weight(x, Center(x))*) ≥ t * *Weight(y, Center(x))* ***then***
4. ***if*** link *l(y, Center(x))* ∈ L and Spandistance(*y, Center(x)*) > t * *Weight(y, Center(x))* ***then***
5. *Span(y, Center(x))* ← True
6. $L_S \leftarrow L_S \cup$ *l(y, Center(x))*
7. ***else if*** Spandistance(x, y) > t * *Weight(x, y)* ***then***
8. *Span(x, y)* ← True
9. $L_S \leftarrow L_S \cup$ *l(x, y)*

---

In the algorithm 5.2, outer ***for*** loop repeats $n - \sqrt{n}$ times and inner ***for*** loop calls $d_{max}$ times in worst case. Total time complexity of algorithm of Phase 2 is $O(n \,.\, d_{max})$ in worst case. Spandistance method takes linear time by using Two-way search method [40].

## 5.4 Phase 3: Inter Cluster Linking initiated by Center

This phase try to connect center with other center. If found this center to center link, then check center to any node belongs in that cluster which have minimum distance with that center add that link in to $L_S$. If not found center to center link then check which node have minimum distance, add that link in to $L_S$.

In this phase, inter cluster linking initiated by center node will take place. Center node will search for any other cluster center within distance R. if exists such a cluster center then it will check for links to clustered nodes of that cluster. From all these links, it will select the minimum distance link and add into spanner graph G′ (Case 1).

Case 2 is that if there exist two or more cluster centers that are within distance R from the center node then it will select that cluster which has minimum distance and then a minimum distance link to this cluster will be added as in Case 1.

Case 3 and Case 4 are that if there is no cluster that is within distance R from cluster node then center node will search for clustered nodes of other clusters that are within distance R. if there exist any such links in graph G, it will select a minimum from them and add this link into spanner graph G′ but in case if there already exist path and t-spanner property is not satisfied, this link will be not added in graph G′. The details of all the cases are given below and Assume that t = 2 for all case.

For Case 1, assume G(N, L) where N = $\{c_1, n_1, c_2, n_2\}$ and L = $\{(c_1, n_1), (c_2, n_2), (c_1, n_2), (c_1, c_2)\}$ and weight of links are W = $\{R, R, \leq R, < R\}$ respectively and G′(N, $L_S$) where $L_S = \{(c_1, n_1), (c_2, n_2)\}$

**Case 1:** In Figure 5.9(a), $x \in \{c_1, c_2\}$ and while processing $x = c_1$ and $y = c_2$, in algorithm 5.3 the first condition at line 4 that center $c_1$ can link with other center $c_2$ is true. So check *Weight(*$c_1, c_2$*)* = $w_1 < R$ and distance from center $c_1$ to nodes that belongs to cluster which have center $c_2$. Whichever have minimum distance that link insert into $L_S$. Like here, $z = n_2$ and *Weight(*$c_1, n_2$*)* = $w_2 \leq R$. checking $w_1$ and $w_2$, here $w_1 < w_2$ and Spandistance($c_1, c_2$) = $\infty$. Hence following t-spanner property $\infty > 2^* w_1$ and add link *l(*$c_1, c_2$*)* into $L_S$ (Shown in Figure 5.9(b)).

For Case 2, assume G(N, L) where N = $\{c_1, n_1, c_2, n_2, c_3, n_3\}$ and L = $\{(c_1, n_1), (c_2, n_2), (c_1, c_2), (c_3, n_3), (c_3, c_2), (c_3, c_1)\}$ and weight of links are W= $\{R, R, < R, < R, < R, \leq R\}$ respectively and G′(N, $L_S$) where $L_S = \{(c_1, n_1), (c_2, n_2), (c_1, c_2), (c_3, n_3)\}$

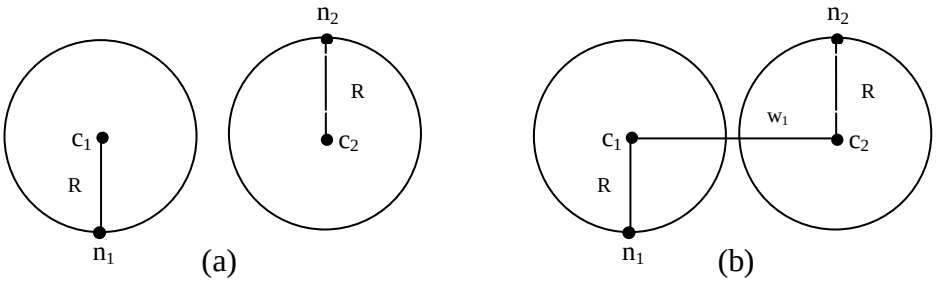



Figure 5.9 For t-spanner algorithm Case 1 (a) After Phase 2 (b) After Phase 3

**Case 2:** In Figure 5.10(a), $x \in \{c_1, c_2, c_3\}$ and while processing $x = c_3$ and $y = c_1$, in algorithm 5.3 the first condition at line 4 that center $c_3$ can link with other center $c_1$ is true. So check *Weight*$(c_3, c_1) = w_3 \leq R$ and distance from Center $c_3$ to nodes that belongs to cluster which have center $c_1$. Whichever have minimum distance that link insert into $L_S$. But here, no node from cluster $c_1$ links to center $c_3$. $z = \emptyset$, checking Spandistance $(c_3, c_1) = w_2 + w_1$ (distance from link $c_3$ to $c_1$ via $c_2$) where *Weight*$(c_3, c_2) = w_2 < R$ and *Weight*$(c_2, c_1) = w_1 < R$. Hence following t-spanner property $(w_2 + w_1) < 2^* w_3$ and do not add link $l(c_3, c_1)$ into $L_S$ because already path in spanner graph G′(Shown in Figure 5.10(b)).

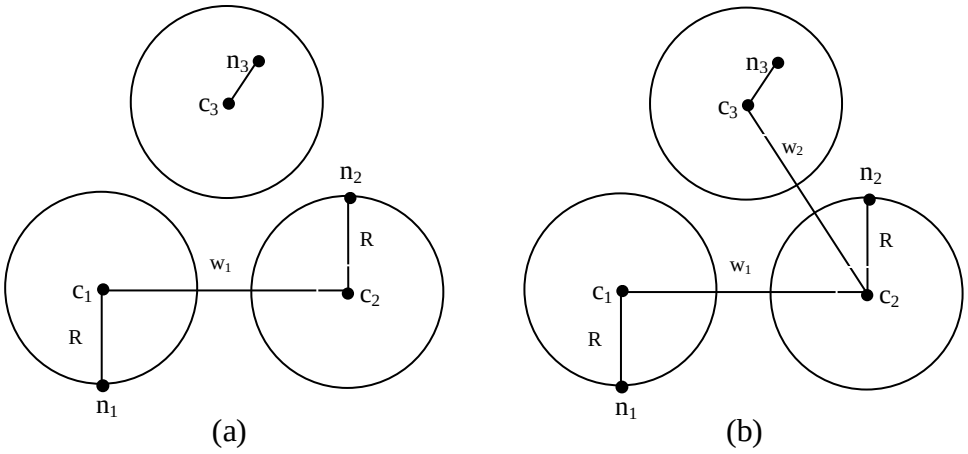



Figure 5.10 For t-spanner algorithm Case 2 (a) After Phase 2 (b) After Phase 3

For Case 3, assume graph G(N, L) where $N = \{c_1, n_1, c_2, n_2, c_3, n_3\}$ and $L = \{(c_1, n_1), (c_2, n_2), (c_3, n_3), (c_1, n_2), (n_3, c_1)\}$ and weight of links are $W = \{R, R, < R, < R, < R\}$ respectively and G′(N, $L_S$) where $L_S = \{(c_1, n_1), (c_2, n_2), (c_3, n_3)\}$

**Case 3:** In Figure 5.11(a), $x \in \{c_1, c_2, c_3\}$ and while processing $x = c_1$ here first condition at line 4 in algorithm 5.3 that Center $c_1$ can link with other center is false. So check whether it can connect with any node other than its cluster. Whichever have minimum distance that link insert into $L_S$. Here $y \in \{n_2, n_3\}$, *Weight*$(c_1, n_2) = w_2 < R$

and *Weight(*$c_1$, $n_3$*)* = $w_1 < R$ checking $w_1$ and $w_2$, here $w_1 < w_2$ and Spandistance($c_1$, $n_3$) = $\infty$. Hence following t-spanner property $\infty > 2^* w_1$ and add link *l(*$c_1$, $n_3$*)* into $L_S$ (Shown in Figure 5.11(b)).

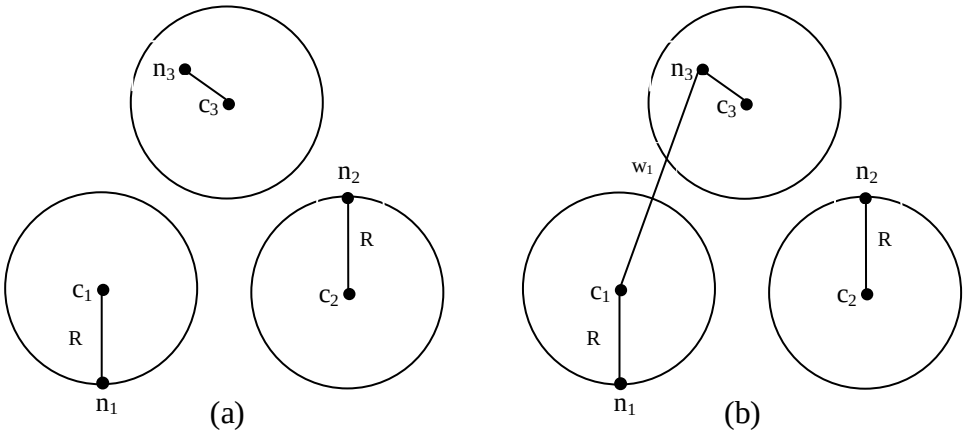



Figure 5.11 For t-spanner algorithm Case 3 (a) After Phase 2 (b) After Phase 3

For Case 4, assume that graph is G(N, L) where N = {$c_1$, $n_1$, $c_2$, $n_2$, $c_3$, $n_3$} and L = {($c_1$, $n_1$), ($c_2$, $n_2$), ($c_3$, $n_3$), ($c_1$, $n_2$), ($n_3$, $c_1$), ($n_1$, $c_2$)} and weight of links are W= {< R, < R, < R, < R, < R, < R} respectively and spanner graph is G′(N, $L_S$) where $L_S$ = {($c_1$, $n_1$), ($c_2$, $n_2$), ($c_3$, $n_3$), ($n_1$, $c_2$)}

**Case 4:** In Figure 5.12(a), $x \in \{c_1, c_2, c_3\}$ and while processing $x = c_1$, in algorithm 5.3 the first condition at line 4 that center $c_1$ can link with other center is false. So check whether it can connect with any node other than its cluster. Whichever have minimum distance that link insert into $L_S$. Here $y \in \{n_2, n_3\}$, *Weight(*$c_1$, $n_2$*)* = $w_4 < R$ and *Weight(*$c_1$, $n_3$*)* = $w_5 < R$ checking $w_4$ and $w_5$, here $w_4 < w_5$ and Spandistance($c_1$, $n_2$) = $w_1 + w_2 + w_3$ (distance from link $c_1$ to $n_2$ via $n_1$, $c_2$) where *Weight(*$c_1$, $n_1$*)* = $w_1 < R$, *Weight(*$n_1$, $c_2$*)* = $w_2 < R$ and *Weight(*$c_2$, $n_2$*)* = $w_3 < R$. Hence following t-spanner property ($w_1 + w_2 + w_3$) < $2^* w_4$ and do not add link *l(*$c_1$, $n_2$*)* into $L_S$ because already path in spanner graph G′(Shown in Figure 5.12(b)).

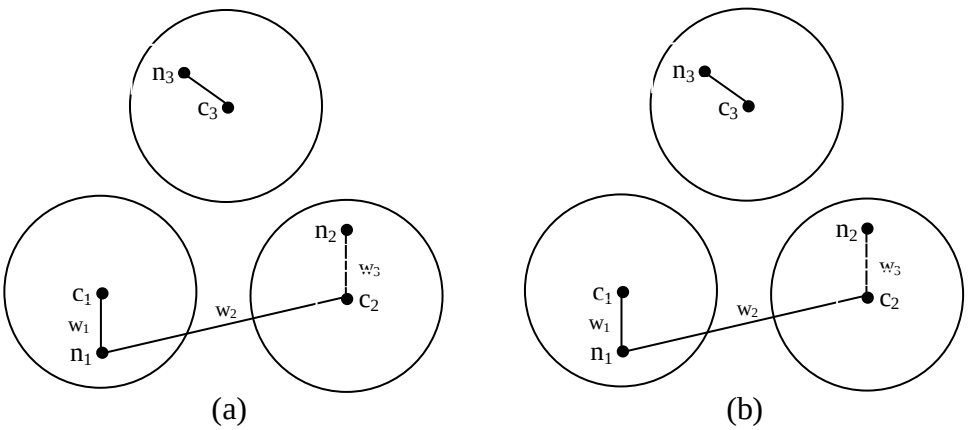



Figure 5.12 For t-spanner algorithm Case 4 (a) After Phase 2 (b) After Phase 3

This all cases can also be handled for t > 2. Above all the cases of Phase 3 are resolve in algorithm 5.3 and algorithm 5.4.

---

**Algorithm 5.3** Inter cluster linking initiated by centers

---

**Procedure** Link_InterclusterBy_center (G, t, R, $C_C$)

1. ***for*** all nodes *x* ϵ $C_C$ ***do***
2. Neighbour_distance ← ∞
3. Neighbour_Node ← ∅
4. ***if*** ∃ node *y* ϵ *Link_list(x)* and *Weight(x, y)* ≤ R and *y* ϵ CC ***then***
   // Center have link with other Center
5. Neighbour_distance ← *Weight(x, y)*
6. Neighbour_Node ← *y*
7. ***for*** all nodes *z* ϵ *Link_list(x)* and *Center(z) = y* ***do***
8. ***if*** *Weight(x, z)* < Neighbour_distance ***then***
9. Neighbour_distance ← *Weight(x, z)*
10. Neighbour_Node ← z
11. ***else***
    // Center does not have link with other Center
12. ***for*** all nodes *y* ϵ *Link_list(x)* and *Weight(x, y)* ≤ R and *Center(y)* != *x* ***do***
13. ***if*** *Weight(x, y)* < Neighbour_distance ***then***
14. Neighbour_distance ← *Weight(x, y)*
15. Neighbour_Node ← y

---

In the algorithm 5.3 of line 4 to line 10 checks that any two center of cluster can be linked with each other. It is possible that more than one center is present which can be link with this center. A tie is broken by selecting the nearest neighbor center. If center cannot linked with other center then line 11 to line 15 checks that weather it can connect with other nearest clustered node.

The algorithm 5.3 checks for all $\sqrt{n}$ cluster centers that can connect with other center or node with minimum distance among selected nodes. If center connected with $d_{max}$ in worst case then time complexity of this algorithm is $O(\sqrt{n} \, . \, d_{max})$.

**Algorithm 5.4** Finding nearest neighbor from center and linked it

**Procedure** Find_Nearest_Neighbor (G, t)

1. ***if*** ( Neighbour_Node != ∅ ) ***then***
2. ***if*** Spandistance(*x*, Neighbour_Node) ≤ t**Weight(x,* Neighbour_Node*)* ***then***
3. *Span(x,* Neighbour_Node*)* ← False
4. $L_S \leftarrow L_S - l(x,$ Neighbour_Node*)*
5. ***else***
6. *Span(x,* Neighbour_Node*)* ← True
7. $L_S \leftarrow L_S \cup l(x,$ Neighbour_Node*)*

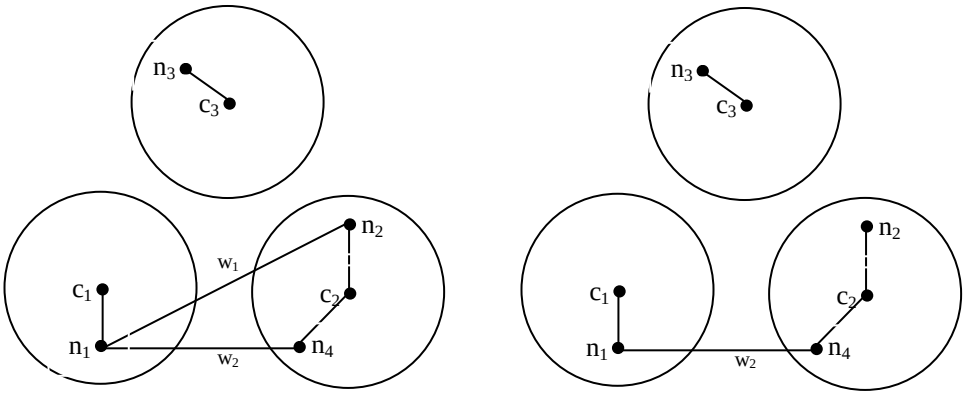
The algorithm 5.4 decides which link has minimum distance by comparing with Spandistance where Spandistance can be computed in linear time by using Two way search method [40] in $O(n)$ in worst case.

## 5.5 Phase 4: Removing Redundant Links from $L_S$ and Linking between Unclustered Nodes

This Phase removes the link from $L_S$ which is not necessary means that after removal of link graph still follows t-spanner property. Case 1 is that if there exist more than one links between any two clusters then the link with minimum distance will retained and all other links will be removed.

Case 2 and Case 3 are that if exists a link between any two cluster nodes of a cluster then the link with maximum distance will be removed. In case 4, unclustered node those are still unconnected will be connected with each other, if possible. Any unclustered node will search for any other unclustered node within distance R and those links will be added into graph G′. The details of all the cases are given below.

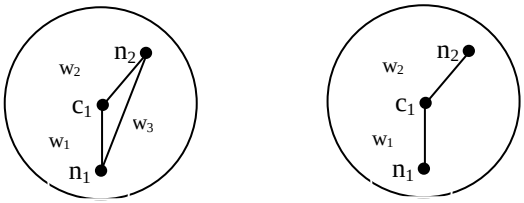
**Case 1:** The spanner graph G′ is shown in Figure 5.13(a), where $x \in \{n_1, n_2, n_3, n_4\}$ and while processing $x = n_1$. Here $n_1$ is directly connected with other cluster nodes $n_2$ and $n_4$ in $L_S$. Whichever have minimum distance that link remains added into $L_S$ and other one is remove. Here $y \in \{n_2, n_4\}$, *Weight(*$n_1, n_2$*)* = $w_1$ and *Weight(*$n_1, n_4$*)* = $w_2$ checking $w_1$ and $w_2$, here $w_2 < w_1$. So remove link *l(*$n_1, n_2$*)* from $L_S$ and do not affect link *l(*$n_1, n_4$*)* in spanner graph G′(Shown in Figure 5.13(b)).




(a) (b)

Figure 5.13 For t-spanner algorithm Case 1 (a) After Phase 3 (b) After Phase 4

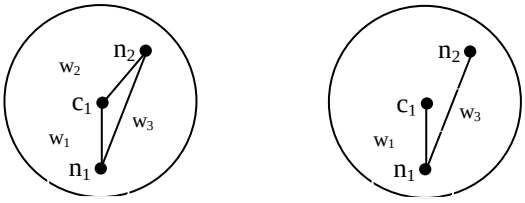
**Case 2:** The spanner graph G′ is shown in Figure 5.14(a), where $x \in \{n_1, n_2\}$ and while processing $x = n_1$. Here $n_1$ is directly connected with same cluster node $n_2$ in $L_S$. *Weight(*$c_1, n_1$*)* = $w_1$, *Weight(*$c_1, n_2$*)* = $w_2$ and *Weight(*$n_1, n_2$*)* = $w_3$ while checking $w_1$ and $w_2$ and $w_3$, here $(w_1 + w_2) < w_3$. So remove link *l(*$n_1, n_2$*)* from $L_S$ and do not affect other links in spanner graph G′(Shown in Figure 5.14(b)).




(a) (b)

Figure 5.14 For t-spanner algorithm Case 2 (a) After Phase 3 (b) After Phase 4

**Case 3:** The spanner graph G′ is shown in Figure 5.15(a), where $x \in \{n_1, n_2\}$ and while processing $x = n_1$. Here $n_1$ is directly connected with same cluster node $n_2$ in $L_S$. *Weight(*$c_1, n_1$*)* = $w_1$, *Weight(*$c_1, n_2$*)* = $w_2$ and *Weight(*$n_1, n_2$*)* = $w_3$ while checking $w_1$ and $w_2$ and $w_3$, here $(w_1 + w_2) > w_3$ and $w_1 < w_2$. So remove link *l(*$c_1, n_2$*)* from $L_S$ and do not affect other links in spanner graph G′(Shown in Figure 5.15(b)).




(a) (b)

Figure 5.15 For t-spanner algorithm Case 3 (a) After Phase 3 (b) After Phase 4

**Case 4:** In Figure 5.16(a) shown only unclustered nodes but not all the clustered node which are present in spanner graph G′(for shake of readability). This phase connects all unclustered node which can link with other unclustered node with Radius R and add those link in to $L_S$ of spanner graph G′. Here unclustered set $U_N = \{u_1, u_2, u_3\}$ and there linking is shown in Figure 5.16(b).


(a) (b)

Figure 5.16 For t-spanner algorithm Case 4 (a) Before Phase 4 (b) After Phase 4

---

**Algorithm 5.5** Removing links from $L_S$ that does not follow t-spanner property

---

**Procedure** Remove_link (G, t, R, $C_N$, $C_C$)

1. From_centerlist ← ∅; To_centerlist ← ∅; To_nodelist ← ∅
2. ***for*** all nodes *x* ϵ N - $C_C$ ***do***
3. ***for*** all nodes *y* ϵ *Link_list(x)* ***do***
4. ***if*** ∃ node *y* ϵ ( $C_N$ - $C_C$ ) and *Weight(x, y)* ≤ R ***then***
5. index ← Find_Node_Index (From_centerlist, To_centerlist, *Center(x)*, *Center(y)*)
6. ***if*** index = -1 ***then***
7. From_centerlist ← From_centerlist ∪ *Center(x)*
8. To_centerlist ← To_centerlist ∪ *Center(y)*
9. To_nodelist ← To_nodelist ∪ *y*
10. ***else if*** *Weight(x,* To_nodelist[index] ) > *Weight(x, y*)
11. ***then*** *Span(x,* To_nodelist[index]*)* ← False
12. $L_S$ ← $L_S$ – *l(x,* To_nodelist[index]*)*
13. To_nodelist ← To_nodelist ∪ *y*
14. ***else if*** ∃ node *y* ϵ $U_N$ and *x* ϵ $U_N$ and *Weight(x, y)* ≤ R ***then***
15. *Span(x, y)* ← True; $L_S$ ← $L_S$ ∪ *l(x, y)*
16. ***if*** To_nodelist != ∅ ***then***
17. ***for*** all nodes *z* ϵ To_nodelist ***do***
18. *Span(x, z)* ← True
19. $L_S$ ← $L_S$ ∪ *l(x, z)*

---

**Algorithm 5.6** Finding the index of node that link with given center and node name

**Procedure** Find_Node_Index (From_centerlist, To_centerlist, Center_x, Center_y)

1. ***if*** Center_y ϵ To_centerlist and Center_x ϵ From_centerlist ***then***
2. Return index of From_centerlist[Center_x]
3. ***if*** Center_x ϵ To_centerlist and Center_y ϵ From_centerlist ***then***
4. Return index of From_centerlist[Center_y]
5. ***Return*** -1

The algorithm 5.5 and algorithm 5.6 of phase 4 is worked on spanner graph G′. It removes unwanted links from $L_S$ and handles all above cases. The algorithm 5.5 analyzes all $n - \sqrt{n}$ nodes and that checks for every intra cluster and inter cluster node linked with it in the spanner graph G′. Time complexity of this phase 4 is $O(n \,.\, d_{max})$ in worst case. This procedure is to find the index of already linked node with given center and node name. It works in linear time.

## 5.6 Phase 5: Connected Component Linking without Constraint of R

The algorithm 5.7 handles connected components [39]. A cluster or node can be connected component if there is no inter cluster link or inter link present from that cluster or node within Radius. It can be more than one connected component present in the graph at a time. In this algorithm, check that any node from connected component can connect with other cluster or connected component (distance within Radius is not necessary). If yes, then link with minimum distance among all inter links from that. In case 1, a connected component will check for other connected component (distance may be > R) and will be connected to the one with minimum distance.

For Case 1, assume G(N, L) where N = {$c_1$, $n_1$, $c_2$, $n_2$, $u_1$, $u_2$, $u_3$, $u_4$} and L = {($c_1$, $n_1$), ($c_2$, $n_2$), ($c_1$, $c_2$), ($u_1$, $u_2$), ($u_1$, $u_3$), ($u_1$, $n_1$), ($u_2$, $c_1$), ($u_2$, $u_3$), ($u_3$, $u_4$), ($u_4$, $c_2$)} and weight of links are W= {R, R, < R, ≤ R, ≤ R, > R, > R, ≤ R, > R, > R} respectively and G′(N, $L_S$) where $L_S$ = {($c_1$, $n_1$), ($c_2$, $n_2$), ($c_1$, $c_2$)}

**Case 1:** Two connected component shown in Figure 5.17(a), where one connected component $cc_1$ = {$u_1$, $u_2$, $u_3$} and second $cc_2$ = {$u_4$}. Now check every connected component one by one, there are three links from first $cc_1$ and its distance are *Weight(*$u_1$, $n_1$*)* = $w_1$, *Weight(*$u_2$, $c_1$*)* = $w_2$ and *Weight(*$u_3$, $u_4$*)* = $w_3$ and whichever have

minimum distance that one is added in to $L_S$ but not other links. Here is $w_1 < w_2 < w_3$ so add only $l(u_1, n_1)$ in to $L_S$ but not $l(u_2, c_1)$ and $l(u_3, u_4)$. Now check for second $cc_2$, $Weight(u_4, c_2) = w_4$ and $Weight(u_4, u_3) = w_3$. Here $w_4 < w_3$ so add $l(u_4, c_2)$ in to $L_S$. There linking is shown in Figure 5.17(b).

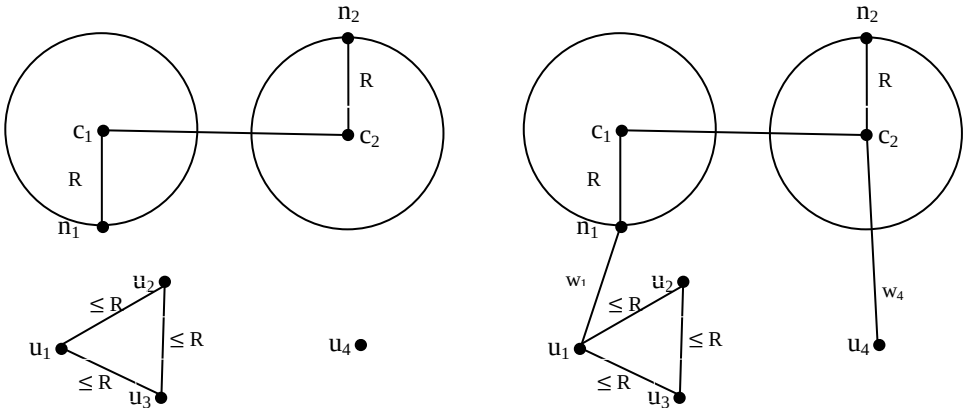



Figure 5.17 For t-spanner algorithm Case 1 (a) After Phase 4 (b) After Phase 5

---

**Algorithm 5.7** Connected component linking and spanning

---

**Procedure** Link_component (G, t, $U_N$)

1. ConnectedComponentList ← ∅
2. ***for*** all nodes $x \in C_C$ ***do***
3. ***for*** all nodes $y \in$ *Link_list(x)* and *Span(x, y)* = False and *Center(y)* != *x*
4. ***do*** ConnectedComponentList[*x*] ← *x* ∪ {for all nodes $y \in$ *Link_list(x)* where *Center(y)* = *x*}
5. ***if*** ConnectedComponentList != ∅ ***then***
6. ***for*** all nodes $x \in$ ConnectedComponentList ***do***
7. min_node ← ∅
8. min_dist ← ∞
9. ***for*** all nodes $y \in$ *Link_list(*ConnectedComponentList[*x*]*)* ***do***
10. ***if*** $y \notin U_N$ and *Center(y)* != *Center(x)* and
11. *Span(x, y)* = False and *Weight(x, y)* < min_dist
12. ***then*** min_dist ← *Weight(x, y)*
13. min_node ← *y*
14. ***if*** min_node != ∅ ***then*** *Span(x,* min_node*)* ← True
15. $L_S \leftarrow L_S \cup l(x,$ min_node*)*

---

It is very less chances that this type of connected component found in the graph still the algorithm 5.7 has mechanism that handles this type of cases. It takes linear time in best case situation and takes $O(\sqrt{n} \,.\, d_{max})$ time in average and worst case scenario.

## 5.7 Phase 6: Removing redundant links from Connected Components

This Phase removes unwanted link from unclustered node means that if this links remove from $L_S$ still graph follows the t-spanner property.

In this case 1, those redundant links will be removed from the connected component which has maximum distance among all the links.

**Case 1:** In Figure 5.18(a) just shown only unclustered nodes not other clustered node for shake of readability. Here $U_N = \{u_1, u_2, u_3\}$ and *Weight(*$u_1$, $u_2$*)* = $w_1$, *Weight(*$u_2$, $u_3$*)* = $w_2$ and *Weight(*$u_3$, $u_1$*)* = $w_3$. So $w_2 < w_3 < w_1$ and remove link *l(*$u_1$, $u_2$*)* from $L_S$ and other remains as it is (Shown in Figure 5.18(b)).


(a) (b)

Figure 5.18 For t-spanner algorithm Case 1 (a) After Phase 5 (b) After Phase 6

In order to follows the t-spanner property of the unclustered nodes, the algorithm 5.8 is formulated. It is a rare chance that some of the nodes remain unclustered. It takes linear time in best case and its complexity is $O(u \,.\, d_{min})$ (where u is number of unclustered nodes) in worst case.

---

**Algorithm 5.8** Removing unnecessary linking of unclustered nodes

---

**Procedure** Remove_Link (G, t, $U_N$)

1. ***for*** all nodes $x \in U_N$ ***do***
2. ***for*** all nodes $y \in$ *Link_list(x)* and *Span(x, y)* = False ***do***
3. ***if*** Spandistance(*x, y)* > t * *Weight(x, y)* ***then***
4. *Span(x, y)* ← True
5. $L_S \leftarrow L_S \cup$ *l(x, y)*

---

## 5.8 Analyzing the Running Time

It is known that $n$ = Number of nodes in Graph, $d_{max}$ = Highest degree of nodes and $d_{min}$= Lowest degree of nodes. Here Phase 1 takes $O(\sqrt{n} \,.\, d_{max})$ expected time in average case and takes $O(n \,.\, d_{max})$ expected time in worst case. While time complexity of Phase 2 is $O(n \,.\, d_{max})$ in worst case. Phase 3 computes in $O(\sqrt{n} \,.\, d_{max})$ in worst case. While Phase 4 performed in $O(n \,.\, d_{max})$ in expected time. Phase 5 gives result in linear time in best scenario and takes $O(\sqrt{n} \,.\, d_{max})$ times in average scenario. Phase 6 computes in $O(u \,.\, d_{max})$ time in worst scenario. Finally Total time complexity of t-spanner algorithm is $O(n \,.\, d_{max})$ in worst scenario and $O(\sqrt{n} \,.\, d_{max})$ in average scenario. After performing t-spanner algorithm, the expected size for the graph is $O(n^{1+\frac{1}{t}})$ at given time [36].

## 5.9 Analyzing the Stretch of the Spanner

**Lemma 5.1** Let c′ is any cluster in *C*. Each node $y \in$ c′ is connected to its center through a path of at most *i* links from $L_S$.

**Proof:** The proof is based on induction on *i* and the number of links in the spanner seen so far. Let *x* be the center of the cluster c′. If c′ is a singleton cluster, there is nothing to prove, so assuming otherwise, let $y \neq x$ be a node which belongs to c′. The node *y* would have become member of c′ only in the following situation in the past. There are links present in the graph that incident on node *y* whose weight ≤ R. Some link (*y*, *z*) appeared in the spanner with node *z* being a member of some cluster c in *C*. The assertion implies that c and c′ have a common node *y* and *x* is its center too. Now applying induction hypothesis, there is path exist in $L_S$ between node *y* to node *w* with an intra cluster link. The link (*y*, *x*) remove in algorithm 5.5 of phase 4 due to the link (*y*, *w*) exists in $L_S$. Now node y still can be reached to its center via node *w* with a length ≤ *i*. if link (*y*, *w*) is not present in $L_S$ but link (*y*, *z*) exist in $L_S$. Node *y* still can access the center of *z* with a length ≤ *i*.

## 5.10 Analyzing the Spanner Size

**Lemma 5.2** Let $K_n$ is the complete graph of *n* nodes and has ${}^nC_2$ links. Let w be the weight of the every link in graph. After performing t-spanner algorithm, the expected number of links in the $L_S$ is at most $n^{3/2}$ in worst case.

**Proof:** The proof is based on induction on *n* and numbers of nodes are selected as cluster center. The clustering algorithm 4.1 selects the $\sqrt{n}$ nodes as center of cluster based on the decreasing order of degree. In complete graph $K_n$, the degree of every node is same *n*-1 and radius is w. So any $\sqrt{n}$ nodes are randomly selected. If every centers are link with every other nodes then only then $\sqrt{n}\ (n - \sqrt{n})$ links can be added into $L_S$ and total size of spanner is $n^{3/2}$ but this case cannot be occurred in proposed algorithm because every phase of algorithm follows the t-spanner property. When the algorithm is performed, the maximum number of links added into $L_S$ is $(n + n/2) < n^{3/2}$. For any stretch factor t, the expected number of links in $L_S$ is at most $n^{3/2}$ in worst case.

## 5.11 Fully Dynamic Algorithm for t-spanner

In communication networks for instance, a network changes its routes as nodes and links go down due to failures and repairs. So updation (insertion or deletion of links) should handle in efficient way. In this thesis, deletion of links is handled by decremental algorithm 5.9 and insertion of links is handled by incremental algorithm.

### 5.11.1 Decremental Algorithm for t-spanner

---

**Algorithm 5.9** Decremental algorithm

---

**Procedure** Deletelink (G, G′, *x*, *y*, $C_N$, $C_C$, $U_N$) // Deleting link *l(x, y)* from network

1. ***if*** link *l(x, y)* ∈ L but link *l(x, y)* ∉ $L_S$ ***then***
   // easily delete link from the data structure of *x* as well as *y*
2. L ← L – *l(x, y)*
3. ***if*** link *l(x, y)* ∈ $L_S$ ***then*** // delete link from the data structure of *x* as well as *y*
4. $L_S$ ← $L_S$ – *l(x, y)*
5. ***if*** node *x* ∈ $U_N$ and node *y* ∈ $U_N$ ***then***
6. After deletion if either *x* or *y* are not link with any other node in spanner then find node *z* from L with minimum distance from *Link_list(x)* or *Link_list(y)* respectively.
7. $L_S$ ← $L_S$ ∪ *l(x, z)* or $L_S$ ← $L_S$ ∪ *l(y, z)*

8. ***if*** node $x \in C_N - C_C$ and node $y \in U_N$ ***then***

9. After deletion $x$ has no problem because it is already link with center but if $y$ is not link with any other node in spanner then find node $z$ from L with minimum distance from *Link_list(y).*

10. $L_S \leftarrow L_S \cup l(y, z)$

11. ***if*** node $x \in C_N - C_C$ and node $y \in C_N - C_C$ ***then***

12. ***if*** *Center(x) = Center(y)* ***then***

// Rare chance of having intra cluster link in cluster

13. ***if*** link $l(x, Center(x)) \notin L_S$ ***then***

14. $L_S \leftarrow L_S \cup l(x, Center(x))$

15. ***else***

16. $L_S \leftarrow L_S \cup l(y, Center(y))$

17. ***if*** node $x \in C_C$ ***then***

18. ***if*** node $x$ is not present in top degree list of $\sqrt{n}$ nodes set ***then***

19. $z$ enters in the top most degree list of $\sqrt{n}$ nodes set and Make-New-Center($z$).

20. ***if*** $x$ is connected with other center $w$ in spanner ***then*** change the cluster and Update-Center($x$) $\leftarrow w$

21. ***if*** *Center(y)* = $x$ and node $y \in C_N - C_C$ ***then***

22. Check *Link_list(y)* and find other node $z \in C_C$ and *Weight(y, z)* $\leq$ R and change a cluster and

23. Update-Center($y$) $\leftarrow z$.

24. ***if*** node $y \in C_C$ ***then***

25. ***if*** any node from cluster of $y$ is linked to any node from cluster of $x$ ***then*** no problem.

26. ***if*** not ***then*** make sure that at least one link is present from cluster $y$ to cluster $x$ in $L_S$.

---

The deletion of link in the spanner graph G′ is performed in algorithm 5.9. Consider the deletion of link ($x$, $y$), it follows the description of the above algorithm 5.9 and its processing time $O$(*Degree(x)*) per link deletion in worst case. *Degree(x)* traversing time of *Link_list(x)* in adjacency list otherwise it takes $O$(1) in best case.

### 5.11.2 Incremental Algorithm for t-spanner

The proposed decremental algorithm can be fully dynamic by handling the link insertion in lazy fashion and rebuilding the entire data structure from the scratch after link insertion. This leads to a fully dynamic algorithm for t-spanner with $O(Degee(x))$ time per link insertion or deletion means that it takes linear update time and the expected size of the t-spanner maintained by this algorithm is $O(n^{1+\frac{1}{t}})$ at given time.

# Chapter 6
# Results and Analysis

Maintaining a t-spanner may be practical in the case of very large networks G, whose links L must be stored in external memory; while spanner links $L_S$ could fit into main memory. A network with million nodes could need Terabytes to store its links, while the links in its t-spanner only need order of Gigabytes, at the cost of a limited weight (or transfer load ) stretch [41].

## 6.1 Optimization of Resources for Routing Table using t-spanner Algorithm

Large routing table slow down the process and occupies more space that can not fit into main memory. To optimize this space, apply t-spanner algorithm on computer networks so that in spanner networks, each router have less entry than original one.

For applying t-spanner algorithm, consider weight of link as congestion load of link. Congestion load is that how much traffic on the link. Congestion load can be calculated as (transfer speed * receiving speed)/bandwidth.

In Figure 6.1, there are 15 routers and their corresponding routing table is shown. Each router is connected with other router with the dedicated link used for data transfer between them. In Figure 6.1, some numeric value is assigned to every links. Each link has different capacity of handling congestion load. Every links can handle assigned numeric value congestion load or transfer load. There are 2 columns in Routing Table: First for Target router, second for congestion load between source router and target router. Source router name is mentioned on first column of first row of routing table (whose second column is blank).

For applying t-spanner algorithm in above computer networks, represents router as node and dedicated link as simple link and congestion load of link as weight of link in Figure 6.2. The Node set N = {a, b, c, d, e, f, g, h, j, k, l, m, n, o, p} and L is set of links. Here taken **t = 2**. It can also works well for t > 2.

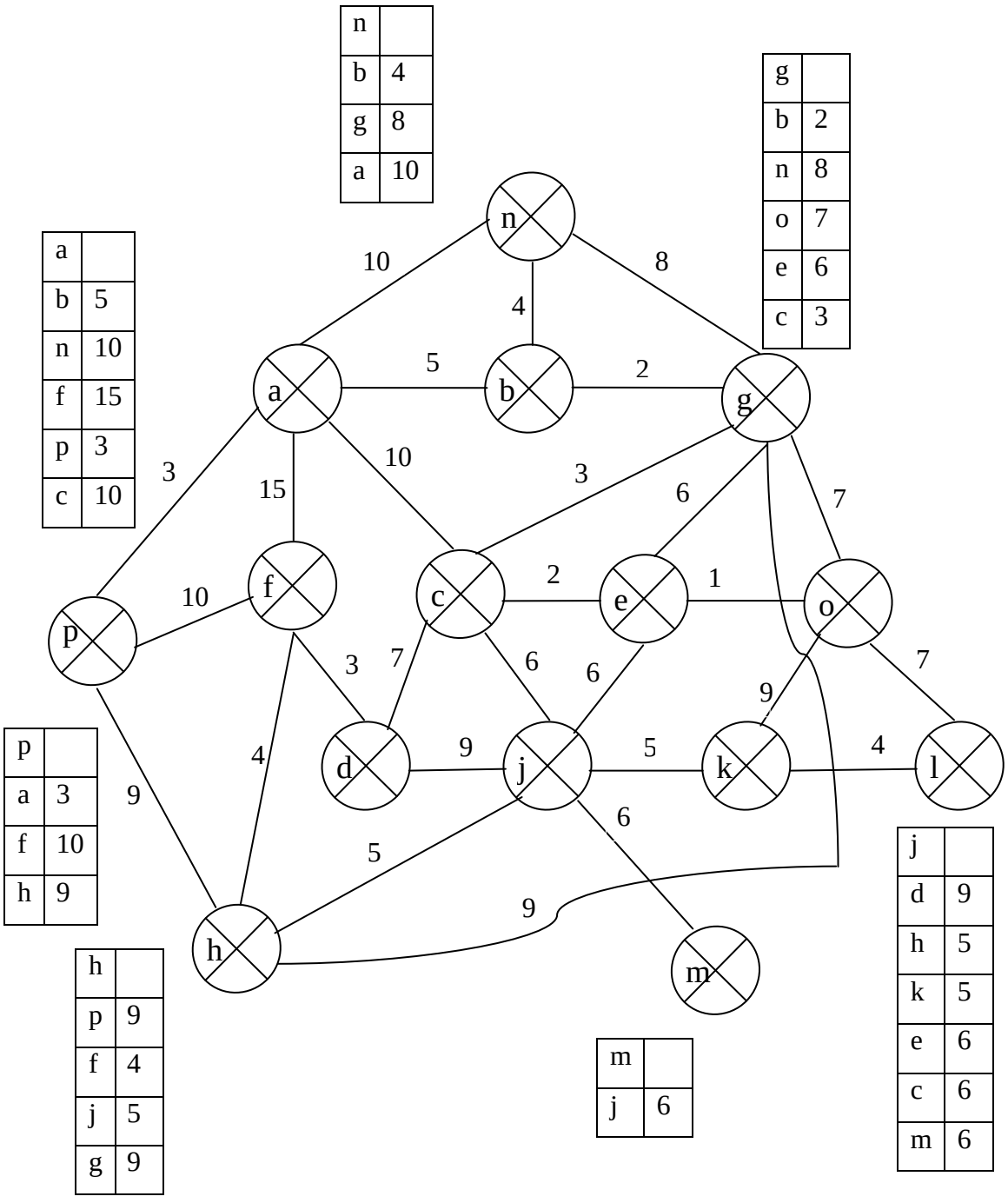



Figure 6.1 The computer networks with routers and routing table

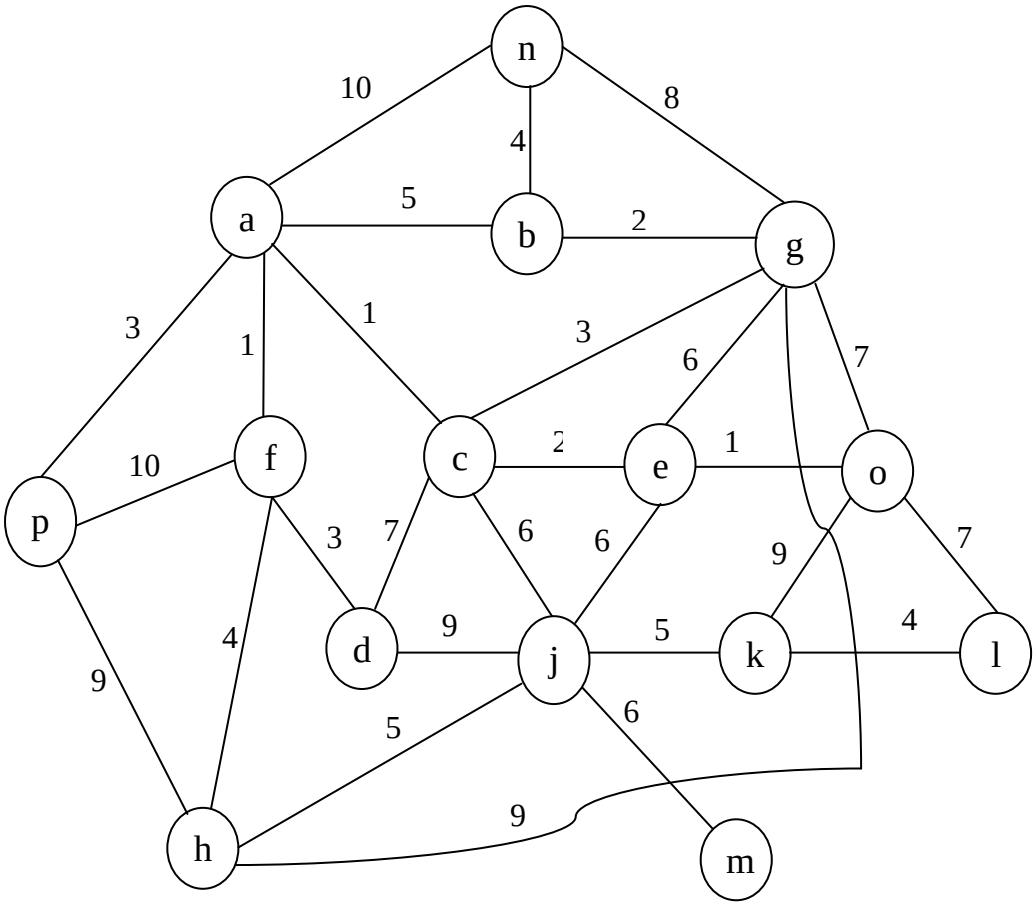



Figure 6.2 The undirected weighted graph representation of networks

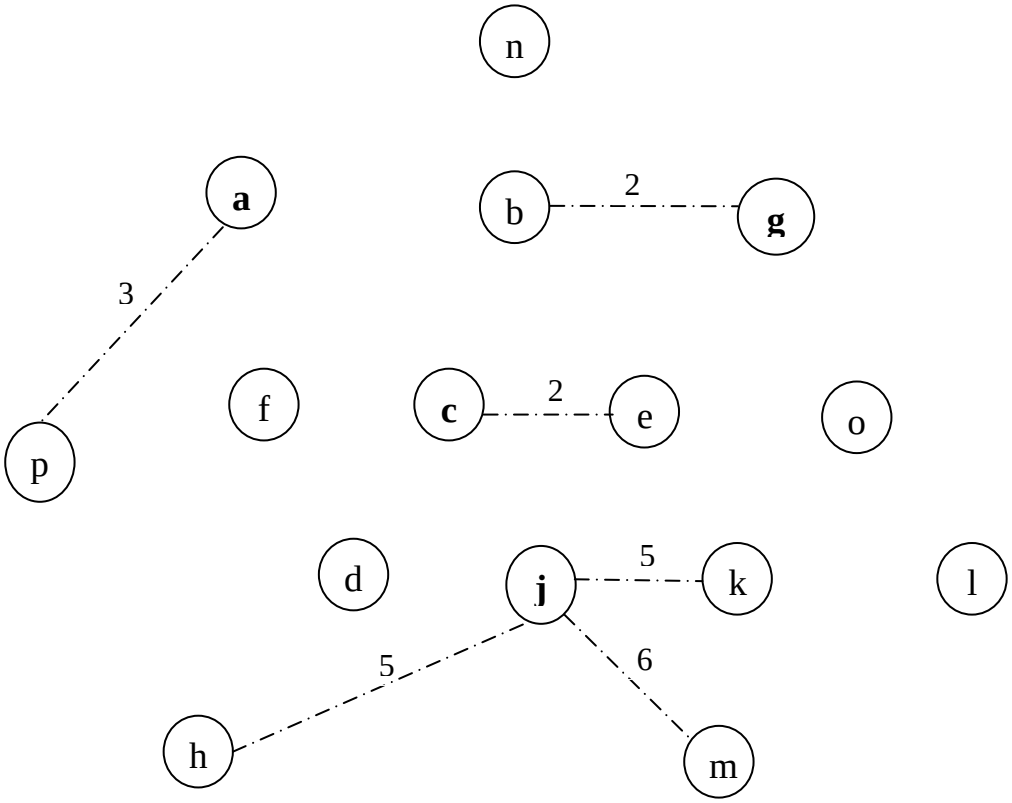



Figure 6.3 The clustered graph after clustering

There are $\lceil\sqrt{15}\rceil$ = 4 cluster centers (shown in Bold in Figure 6.3) are selected based on decreasing order of degree and Here Radius is 6. After clustering, nodes are divided in two set: Clustered set $C_N$ and Unclustered set $U_N$. Here $U_N$ = {n, f, o, d, l} and $C_N$ = {a, p, b, g, c, e, j, k, h, m} where Center set $C_C$ = {a, c, g, j}. The links that are added into $L_S$ by applying current phase are shown as dotted line and already added links are shown as dark line.

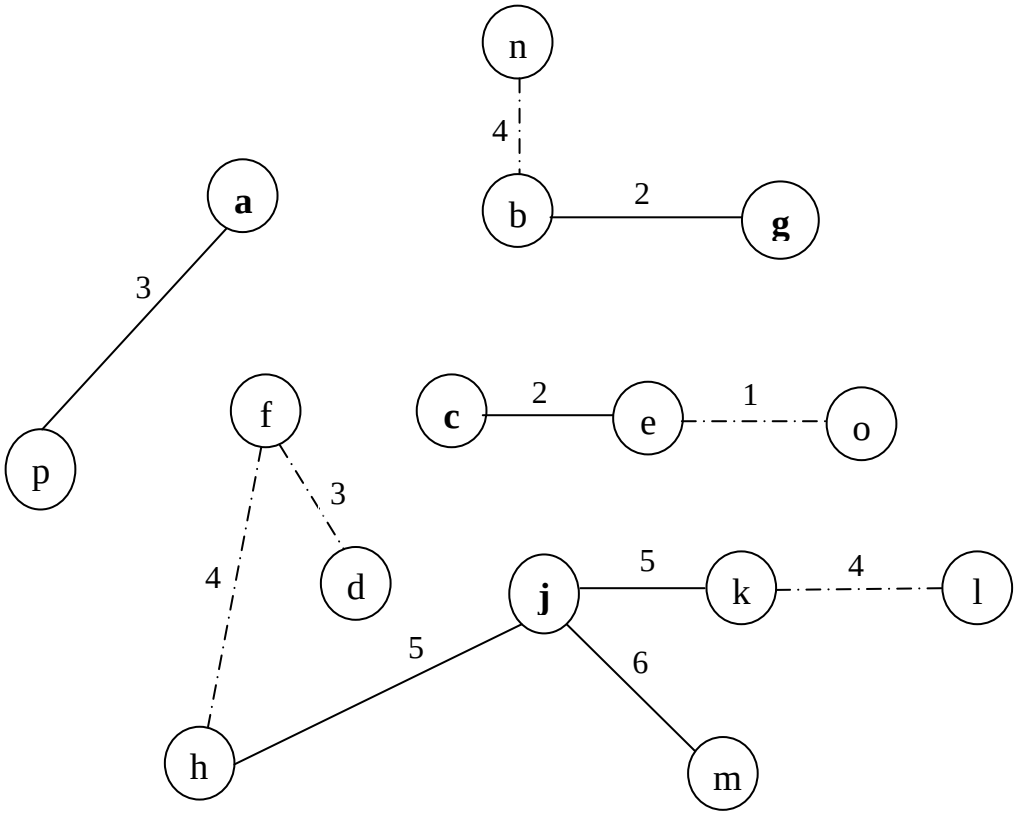



Figure 6.4 After performing phase 1 and phase2

In Phase 1, here undirected node link with clustered node whose have weight ≤ R and also all intermediate nodes from center up to distance R are linked(but here no such link found). No need to perform inter cluster linking here so Phase 2 gives same result as Phase 1.

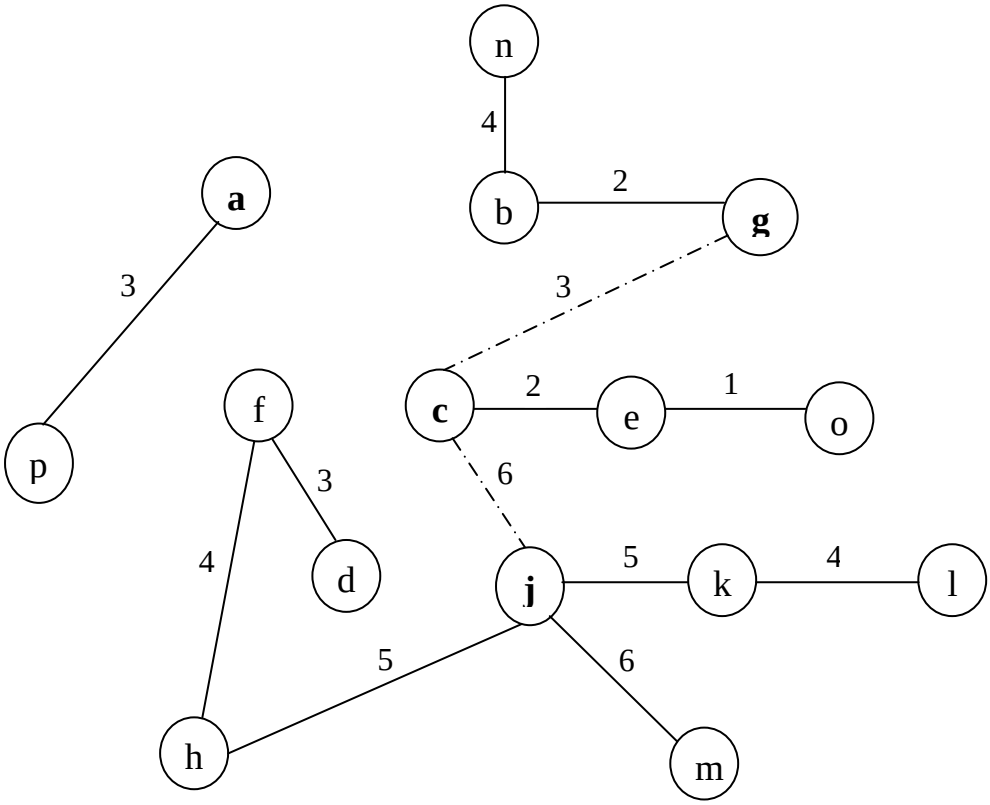



Figure 6.5 After performing center to center linking

In Phase 3, center c is linked with center g and j shown in Figure 6.5.

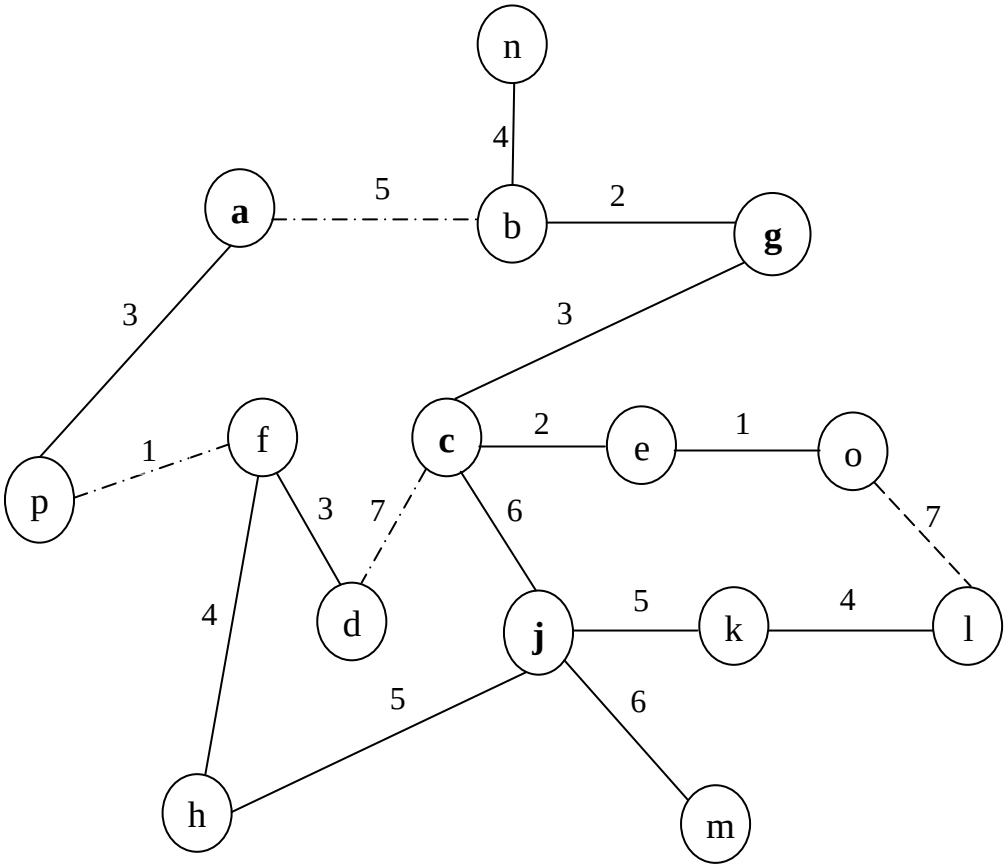



Figure 6.6 After performing phase 4, phase 5 and phase 6

Phase 4, Phase 5 and Phase 6 remove unwanted links; handle connected component and unclustered nodes respectively. Final Spanner Graph is shown in Figure 6.7.

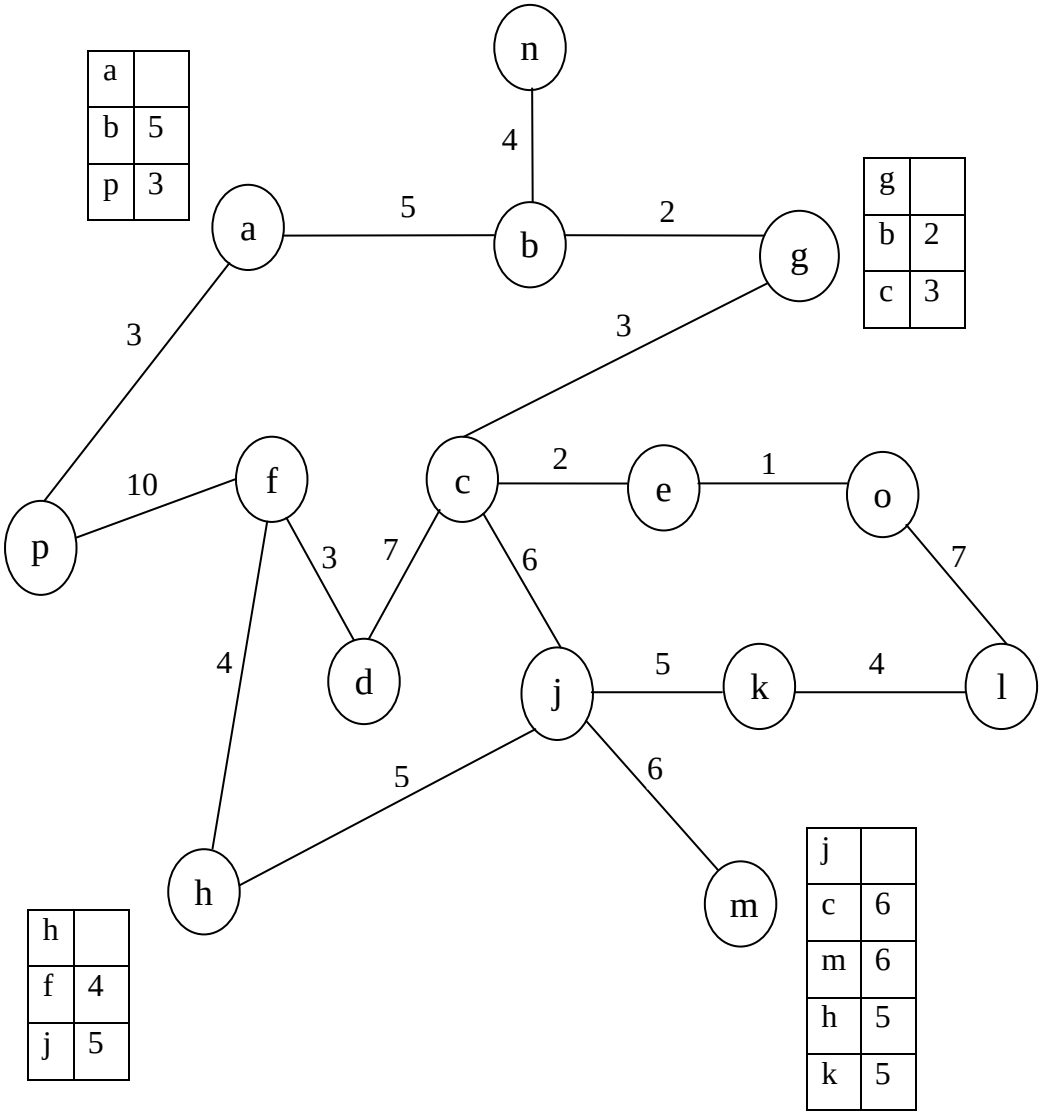



Figure 6.7 The 2- spanner Graph

Figure 6.7 shows that routing table has very less entry than original one and it occupies the less memory space compare to earlier. In this way it can sparse the very large networks and can optimize all the routing tables using t-spanner technique. Updation like insertion or deletion of links can be handled in the proposed fully dynamic algorithm in linear time.

The algorithms designed in this thesis are practically implemented in java language. Each phase of algorithm is represented by function. Graphical user interface is also designed for user to draw undirected weighted graph.

## 6.2 Snapshots of Designed System

Figure 6.8 shows the welcome screen of designed system for user to check the result of spanner graph.

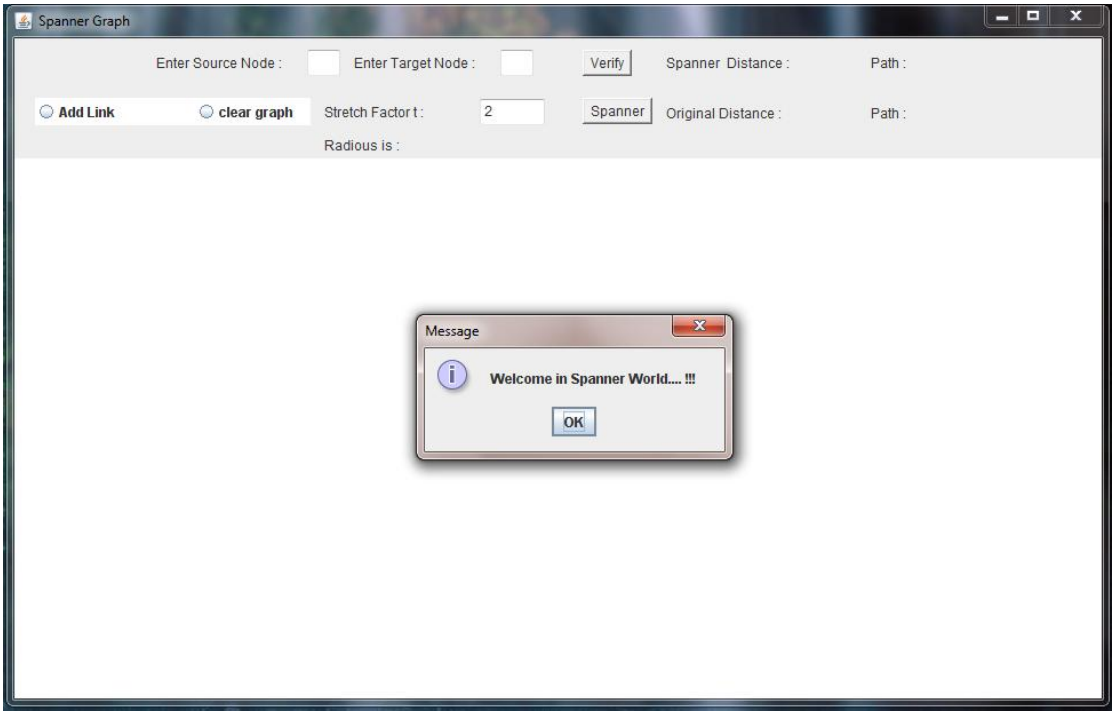



Figure 6.8 Graphical user interface of designed system

GUI shown in figure 6.8 is provides the following facilities:

- This system provides the canvas on which user can draw the nodes and links between them.
- User can give the weight of link of corresponding end nodes.
- After drawn the weighted graph, user can see the spanner graph of input graph on same canvas.
- User can change the value of stretch factor other wise 2 is by default.
- User can verify the distance between two nodes whether it follows the t-spanner property or not.

User clicks on the canvas screen and one node is created at clicked co-ordinate position and user can give the name of node if he wants shown in figure 6.9

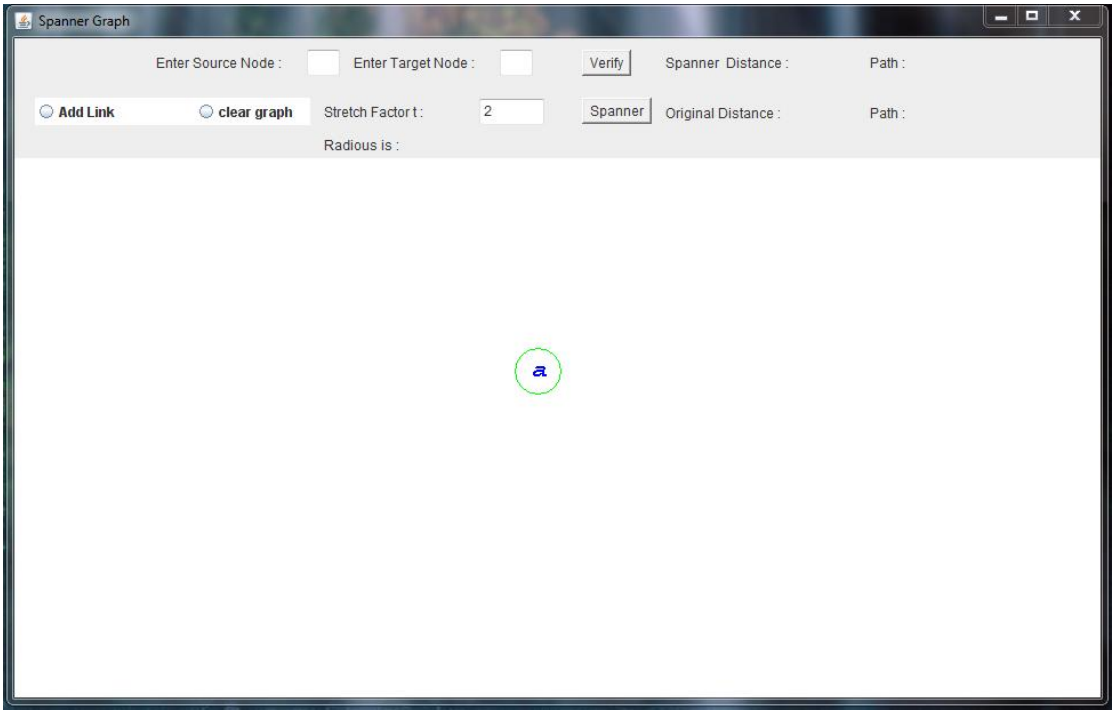



Figure 6.9 Node drawing and naming

Figure 6.10 shows the all the nodes of graph that user has drawn. User can draw more nodes by clicking on canvas.

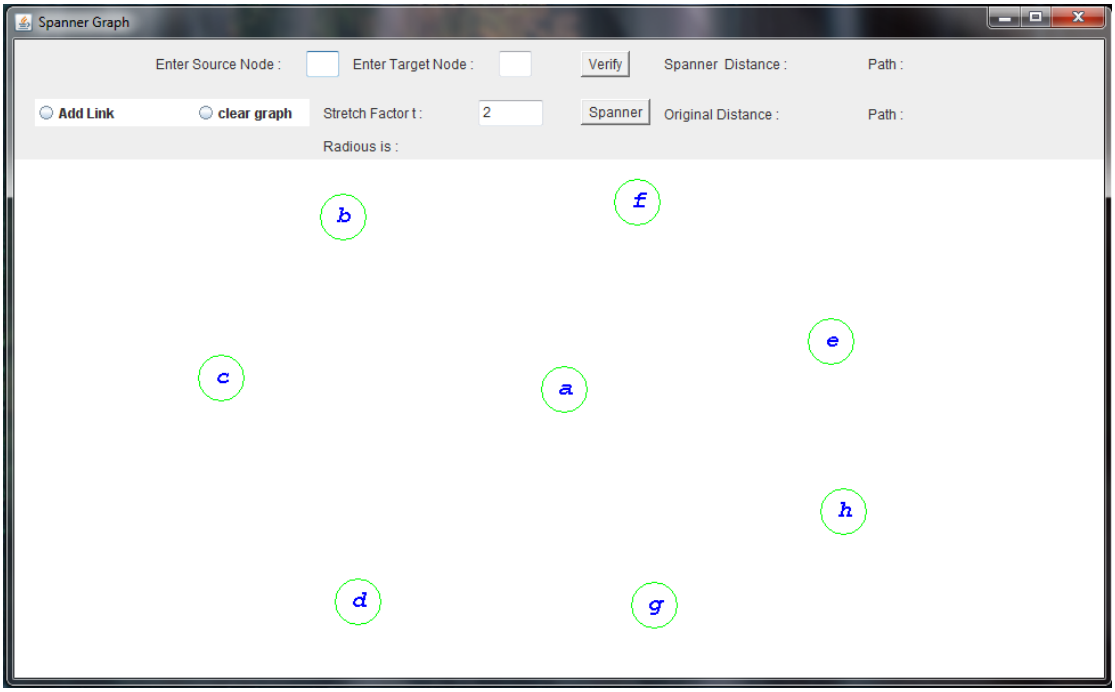



Figure 6.10 Snapshot of node representation of graph

To draw the link, User selects the two nodes and enters the weight of that link into popup textbox.

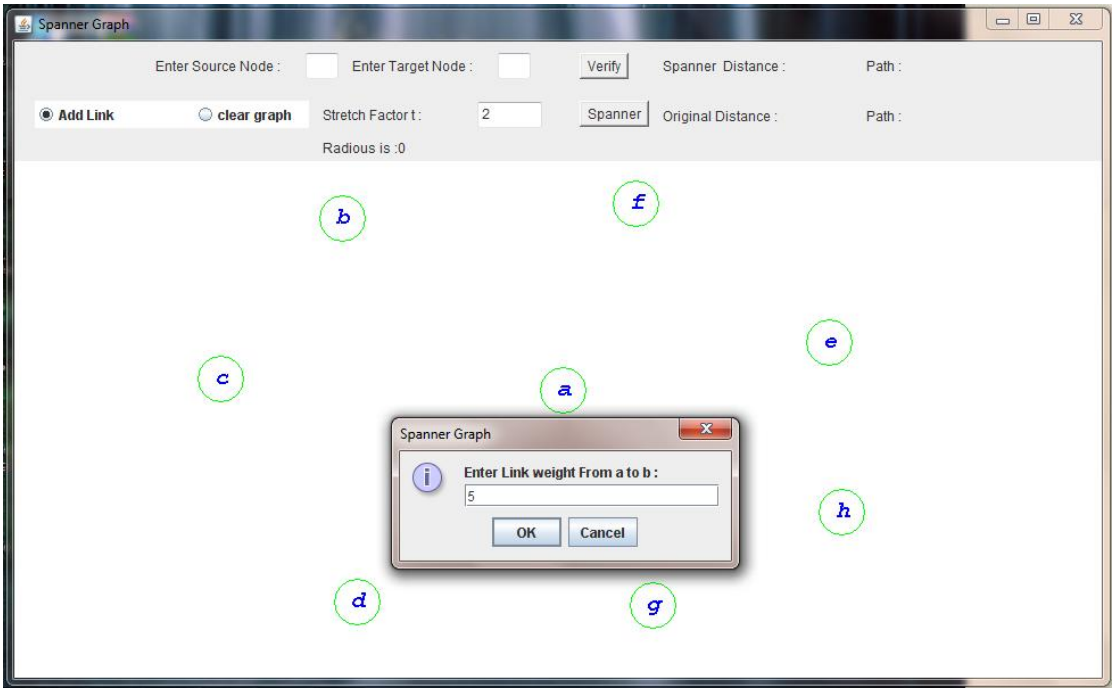



Figure 6.11 Enter weight of link between selected nodes

The user drawn undirected weighted graph by giving weight of all the links is shown in figure 6.12.

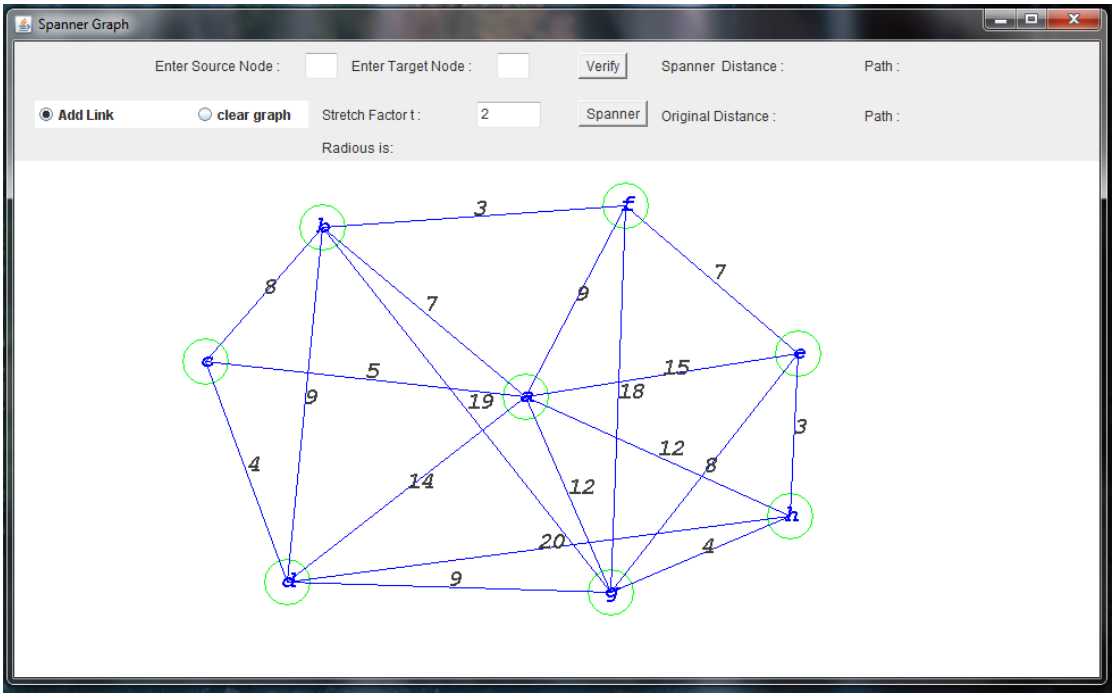



Figure 6.12 Undirected weighted graph drawn by user

User presses the spanner button to see the spanner representation of his drawn graph. User can change the value of stretch factor by entering value in textbox and click spanner button again. The links that are added in spanner graph is shown in red color and center of cluster is also represented in red color shown in figure 6.13

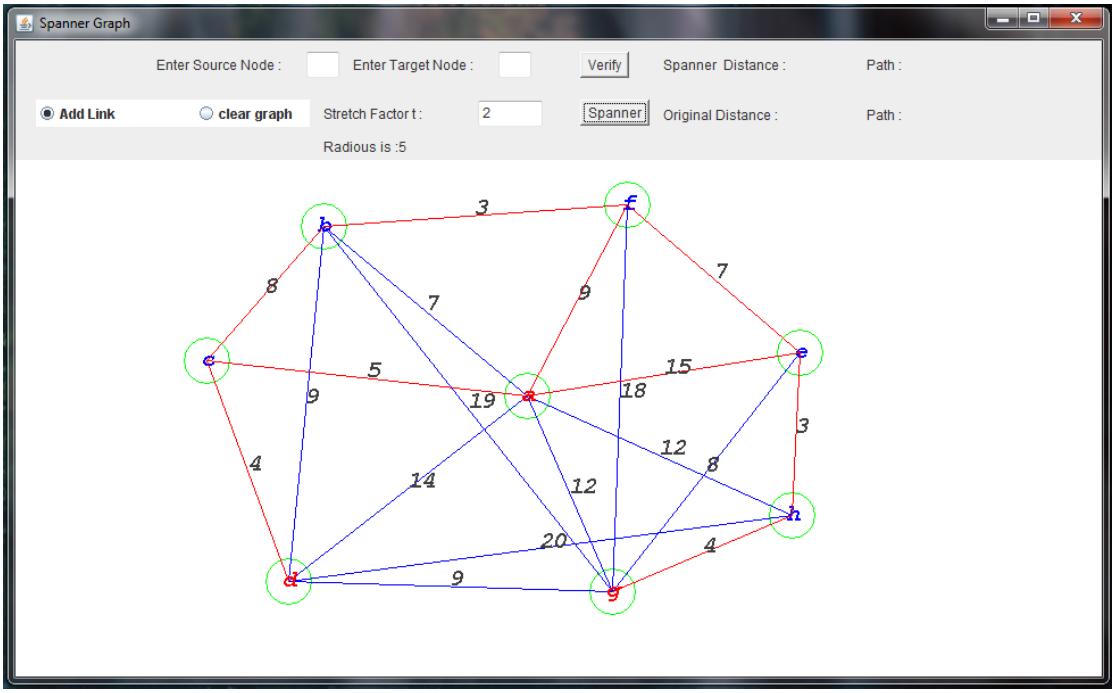



Figure 6.13 Spanner graph representation of user's input graph

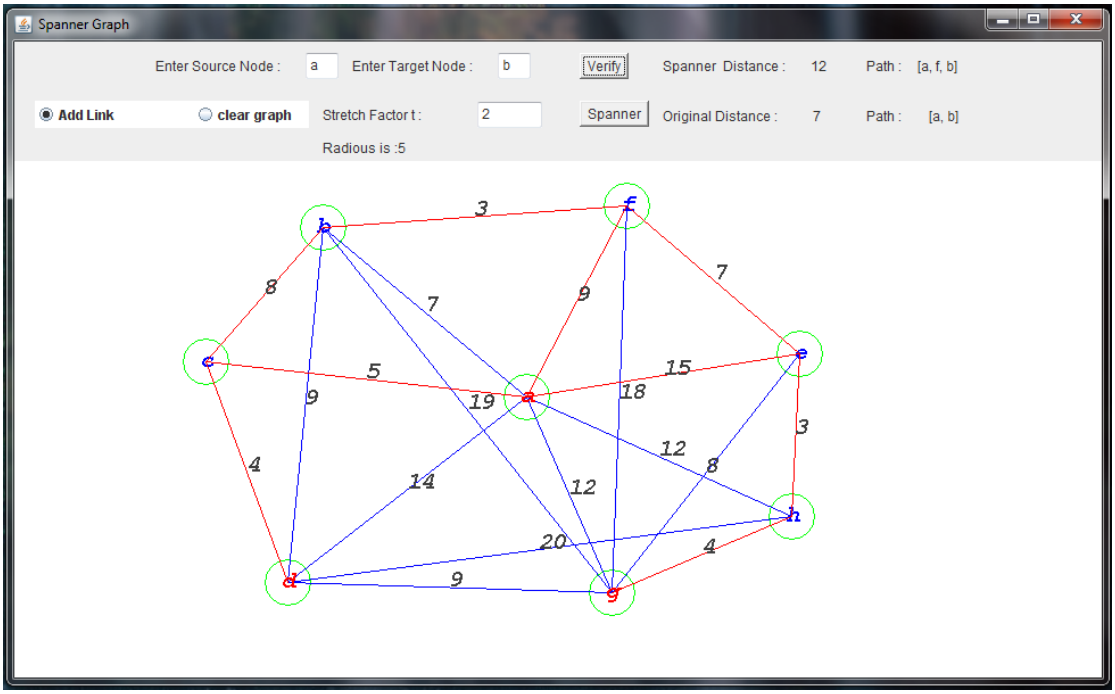



Figure 6.14 Verifying the spanner distance between nodes

User can verify the distance between two nodes that follows the t-spanner property or not by entering source and target node name in textbox and click the verify button.

The spanner distance and original distance between source and target node can be seen in figure 6.14 and path to reach from source to target node is also shown.

## 6.3 Analyzing the Results

The efficiency of routing scheme is measured in terms of its stretch factor. It is the maximum ratio the length of a route computed by the spanner and that of a shortest path connecting the same pair of nodes.

| Number of Node (n) | Number of Links (m) in Graph G | Input stretch factor (t) | Number of Clusters ($\sqrt{n}$) | Radius (R) | Number of Links in Spanner Graph G′ | Output stretch factor (t′) |
|---|---|---|---|---|---|---|
| 7 | 19 | 2 | 3 | 8 | 7 | 1.6 |
| 7 | 19 | 4 | 3 | 8 | 6 | 3 |
| 15 | 86 | 2 | 4 | 11 | 37 | 1.78 |
| 15 | 86 | 3 | 4 | 11 | 35 | 2.4 |
| 15 | 86 | 4 | 4 | 11 | 28 | 3.1 |
| 15 | 86 | 5 | 4 | 11 | 25 | 4 |
| 25 | 125 | 3 | 5 | 12 | 73 | 2.39 |

Table 6.1 Results of experiment on graph

The implemented system is tested by changing the value of number of nodes ($n$), size of graph ($m$) and stretch factor (t). The tested results are shown in table 6.1.

When the value of variables is changed, it affects the size of spanner. Output stretch factor is the maximum ratio the distance of a two end nodes computed by the spanner and that of a shortest path connecting the same pair of nodes.

While keeping number of nodes ($n$) and number of links ($m$) constants, increase in input stretch factor (t) will increase the difference between input stretch factor (t) and output stretch factor (t′) shown in Table 6.1

# Chapter 7
# Conclusion and Future Scope

## 7.1 Conclusion

Dynamic graph algorithm is vast area of research for directed and undirected graphs. There are fully dynamic graph algorithms and partially dynamic graph algorithms for various applications.

The undirected graph maintains the various properties on graph like spanning forest, bipartiteness and planarity. The directed graph algorithm maintains the transitive closure and spanning forest on the graph.

We have optimized space and time complexity of generation of routing table by using t-spanner algorithm and new clustering technique has been introduced using degree and radius as parameter. Fully dynamic algorithm has been developed to handle dynamic changes in the networks.

The algorithm proposed in this thesis are practically implemented and tested on undirected weighted graph with varying size of graph and number nodes.

## 7.2 Future Scope

- The algorithm can be extended for insertion and deletion of nodes. Here only the link insertion and deletion has been handled.
- In this thesis, we have worked on congestion load parameter. This work can be extended for other network parameters like hop-count also.
- It is also applicable for adhoc networks and distributed systems (in inter-processor communication). Another, major challenge is to find good algorithms for directed graphs.
- Cuckoo hashing data structure can be used instead of augmented adjacency list because it provides constant worst case search time.
- The new algorithm can be designed that explores the links in the neighborhood of a node or group of nodes. This feature leads to designing simple external-memory and parallel algorithms for computing sparse spanners, whose running times can be near optimal.

# Bibliography


[1] F. Harary and G. Gupta, "Dynamic graph models," *Mathematical and Computer Modelling,* vol. 25, no. 7, pp. 79-87, April 1997.

[2] M. R. Henzinger and V. King, "Fully dynamic biconnectivity and transitive closure," in *FOCS '95 Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Washington, DC, USA, 1995.

[3] J. Holm, K. d. Lichtenberg and M. Thorup, "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity," *Journal of the ACM,* vol. 48, no. 4, pp. 723-760, July 2001.

[4] G. F. Italiano and I. Finocchi, Dynamic graphs, DISP, Universit`a di Roma "Tor Vergata", 1999.

[5] D. Eppstein, Z. Galil and G. F. Italiano, "Dynamic Graph Algorithms," *SIAM Journal on computing,* vol. 14, no. 3, pp. 41-80, 1999.

[6] C. Demetrescu and G. F. Italiano, "Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures," *Journal of Discrete Algorithms,* vol. 4, no. 3, pp. 353-383, September 2006.

[7] S. Baswana, R. Hariharan and S. Sen, "Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths," in *STOC '02 Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, New York, USA, 2002.

[8] U. Zwick, "All Pairs Shortest Paths in weighted directed graphs-exact and almost exact algorithms," in *FOCS '98 Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA, 1998.

[9] D. Garg and M. Tyagi, "Comparative Analysis of Dynamic Graph Techniques and Data Structure," *International Journal of Computer Applications,* vol. 45, no. 5, pp. 41-46, 2012.

[10] G. N. Frederickson, "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications," *SIAM Journal on Computing,* vol. 14, no. 4, pp. 781-798, 1985.

[11] Z. Galil and G. F. Italiano, "Fully Dynamic Algorithms for 2-Edge Connectivity," *SIAM Journal of Computing*, vol. 1, no. 21, pp. 1047-1069, 1992.

[12] M. R. Henzinger, "Fully dynamic biconnectivity in graphs," *Algorithmica*, vol. 13, no. 6, pp. 503-538, 1995.

[13] G. N. Frederickson, "Ambivalent Data Structures for Dynamic 2-Edge-Connectivity and k Smallest Spanning Trees," *SIAM Journal on Computing*, vol. 26, no. 2, pp. 484-538, April 1997.

[14] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269-271, 1959.

[15] M. R. Henzinger and V. King, "Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation," *Journal of the ACM*, vol. 46, p. 516, 1999.

[16] B. Haeupler, T. Kavitha, R. Mathew, S. Sen and R. E. Tarjan, "Incremental cycle detection, topological ordering, and strong component maintenance," *ACM Transactions on Algorithms*, vol. 8, no. 1, pp. 1-33, January 2012.

[17] B. Steensgaard, "Points-to analysis in almost linear time," in *POPL '96 Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, USA, 1996.

[18] A. Marchetti-Spaccamela, U. Nanni and H. Rohnert, "On-line graph algorithms for incremental compilation," in *Graph-Theoretic Concepts in Computer Science*, Springer Berlin Heidelberg, June 1993, pp. 70-86.

[19] F. Belik, "An Efficient Deadlock Avoidance Technique," *IEEE Transactions on Computers*, vol. 39, no. 7, pp. 882-888, July 1990.

[20] D. Peleg and A. A. Schäffer, "Graph spanners," *Journal of Graph Theory*, vol. 13, no. 1, p. 99–116, March 1989.

[21] D. Handke and G. Kortsarz, "Tree Spanners for Subgraphs and Related Tree Covering Problems," in *Graph-Theoretic Concepts in Computer Science*, Konstanz, Germany, Springer Berlin Heidelberg, June 15–17, 2000, pp. 206-217.

[22] I. Althofer, G. Das, D. P. Dobkin, D. Joseph and J. Soares, "On sparse spanners of weighted graphs," *GEOMETRY: Discrete & Computational Geometry,*, vol. 9, pp. 81-100, 1993.

[23] B.Awerbuch, "Complexity of network synchronization," *Journal of the ACM,* vol. 32(4), pp. 804-823, October-1985.

[24] H. J. Bandelt and A. Dress, "Reconstructing the shape of a tree from observed dissimilarity data," *Advances in Applied Mathematics,* vol. 7, no. 3, p. 309–343, September 1986.

[25] D. Richards and A. Liestman, "Degree-Constrained Pyramid Spanners," *Journal of Parallel and Distributed Computing,* vol. 25, no. 1, pp. 1-6, February 1995.

[26] A. L. Liestman and T. C. Shermer, "Additive graph spanners," *NETWORKS: An International Journal,* vol. 23, no. 4, p. 343–363, July 1993.

[27] D. Peleg and J. D. Ullman, "An optimal synchronizer for the hypercube," *SIAM Journal of Computing,* vol. 18, pp. 740-747, 1989.

[28] F. Baker, Writer, *Requirements for IP Version 4 Routers.* [Performance]. Network Working Group, Cisco Systems, June 1995.

[29] D. Peleg and E. Upfal, "A trade-off between space and efficiency for routing tables," *Journal of the ACM,* vol. 36, no. 3, pp. 510-530, July,1989.

[30] P. Erdös, "On some extremal problems in graph theory," *Israel Journal of Mathematics,* vol. 3, no. 2, pp. 113-116, 1965.

[31] E. Cohen, "Fast algorithms for constructing t-spanners and paths with stretch t," *SIAM Journal on Computing,* vol. 28, pp. 210-236, 1998.

[32] M. Thorup and U. Zwick, "Approximate distance oracles," in *STOC '01 Proceedings of the thirty-third annual ACM symposium on Theory of computing,* 2001.

[33] L. Roditty, M. Thorup and U. Zwick, "Deterministic Constructions of Approximate Distance Oracles and Spanners," in *Automata, Languages and Programming,* Springer Berlin Heidelberg, 2005, pp. 261-272.

[34] S. Baswana and S. Sen, "A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs," *Journal of Random Structures and Algorithms,* vol. 30, no. 4, pp. 532 - 563, July 2007.

[35] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Representations of graphs," in *Introduction to Algorithms,* MIT Press and McGraw-Hill, 2001, p. 527–529 .

[36] S. Baswana and S. Sen, "A Simple Linear Time Algorithm for Computing a (2k-1)-Spanner of $O(n^{1+1/k})$ Size in Weighted Graphs," in *Automata, Languages and Programming*, Berlin, Springer, 2003, pp. 384-396.

[37] M. Elkin and D. Peleg, "$(1 + \varepsilon, \beta)$-Spanner constructions for general graphs," in *In Proceedings of the 33th Annual ACM Symposium on Theory of Computing*, 2001, pp. 608-631.

[38] H. M. Mahmoud, "Distributional analysis of swaps in Quick Select," *Theoretical Computer Science*, vol. 411, no. 16-18, pp. 1763--1769, March, 2010.

[39] S. Even and Y. Shiloach, "An On-Line Edge-Deletion Problem," *Journal of the ACM*, vol. 28, no. 1, pp. 1-4, Jan. 1981.

[40] B. Haeupler, T. Kavitha, R. Mathew, S. Sen and R. E. Tarjan, "Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance," *ACM Transactions on Algorithms*, vol. 8, no. 1, pp. 1-33, January 2012.

[41] G. Ausiello, P. G. Franciosa and G. F. Italiano, "Small Stretch Spanners on Dynamic Graphs," in *Algorithms – ESA 2005*, Palma de Mallorca, Spain, Springer Berlin Heidelberg, October, 2005, pp. 532-543.

# List of Publications

[1] H. K. Navadiya and D. Garg, "Routing Table Optimization using t-Spanner Technique," *Journal of Graph Theory.*[Communicated]

[2] H. K. Navadiya and D. Garg, "Construction of 3-spanner graph in linear time," *in Third International Conference on Computational Intelligence and Information Technology – CIIT 2013,* Oct 17-18, 2013,Mumbai, India. [Accepted]