HIERARCHICAL DATA STRUCTURES FOR MOBILE NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jie Gao
August 2004

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Leonidas J. Guibas
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Rajeev Motwani

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Balaji Prabhakar

Approved for the University Committee on Graduate Studies.

Abstract

An ad hoc mobile network is a network on a set of autonomous mobile nodes with wireless communication. The network has no fixed infrastructure and is highly dynamic. In this dissertation I describe several geometric algorithms that extract and maintain combinatorial structures that are local, lightweight, smooth, and stable.

The first part studies a fundamental problem of clustering the nodes - find a set of clusterheads such that any node can communicate with at least one clusterhead directly. We propose a constant approximation that is provably stable under motion. This algorithm is also used to construct a linear-size "backbone" subgraph in the mobile wireless network so that both the number of hops and the Euclidean length of the shortest path on the backbone graph are at most a constant times those in the original communication graph.

The second part studies the tradeoff between two goals of routing algorithms: balancing the load of the nodes and using short routing paths. In general these two goals are in conflict with each other. However, when the wireless nodes are distributed on a line or in a narrow band, we show by a distributed algorithm that both a constant load balancing ratio and a constant stretch factor can be obtained simultaneously. In addition, if the nodes are distributed in the plane, there is a non-trivial tradeoff of the two factors in terms of the density of the nodes.

The third part focuses on clustering the pair-wise distances. In the graph domain, we can represent the pairwise shortest path distances in a unit disk graph by an almost linear-size data structure called the well-separated pair decomposition. In the Euclidean space, we can represent the pairwise distances of a set of mobile points by a linear size spanner graph. Both structures can replace the heavy computation

of all-pairs distances by using simple combinatorial structures of almost linear size. Therefore many proximity problems that involve all pairs distances can be approximately answered in a more efficient way.

Acknowledgements

Over my last five years I have had the privilege to work with a number of people who have made my time at Stanford University enjoyable and rewarding. I'd like to thank all of them. Without them this dissertation would not be possible.

I am especially grateful to my advisor, professor Leonidas Guibas. He is an outstanding researcher with broad knowledge, sharp intuition and grand vision. Leo is also a great advisor. He is very patient and gives me a lot of freedom to explore the field by myself. I did not have many experiences doing research before I came to Stanford. It's not flattering at all to say that Leo basically shaped my academic life.

I'd like to thank all my coauthors, especially those who have made contributions to this dissertation: Leonidas Guibas, John Hershberger, An Nguyen, Li Zhang and An Zhu. It has always been great to work with them. Our collaboration has always been enjoyable and fruitful.

I want to thank all the students and postdocs in Leo's geometry group: Qing Fang, Natasha Gelfand, Niloy Mitra, An Nguyen, Mark Pauly, Daniel Russel, Jaewon Shin, Anthony Man Cho So, Danny Yang, Afra Zomorodian. They make the group not only a wonderful place for discussions on research problems, but also a warm and lively family that I will remember forever.

I want to thank all my friends at Stanford who helped me in various ways. Without them my life at Stanford would not be so colorful and fun.

Finally I want to thank my parents and my husband for their endless love. Every time I felt discouraged and depressed. They were always there offering their understanding, support and encouragement. This dissertation is dedicated to them.

Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction

One of the motivations of computer science is to model, interact with and predict the physical world. While motion is ubiquitous in the physical world, how to model and control objects in a moving and highly dynamic environment has been a big challenge in computer science. These challenges arise in many different scenarios, from constructing efficient ad hoc networks on mobile wireless nodes, to simulating the process of protein folding. While there has been tremendous amount of research on static objects, how to handle motion is still an area that is quite open for computer science researchers. The fundamental problems in this area are the lack of efficient data structures to handle moving objects, and a deep understanding of the underlying philosophy. This dissertation studies several geometric problems that arise from mobile ad hoc networks, all of which have the flavor of designing efficient and flexible structures for continuously moving objects.

An *ad hoc* network is a network on a set of autonomous nodes. For different applications, the nodes in such a network vary in terms of computation, communication and power. From high-end to low-end, the wireless nodes could be laptops, PDAs, or small and inexpensive devices such as berkeley motes and smart dust [85]. In an *ad hoc* network, every node is equipped with wireless communication device such as omnidirectional antenna. In the most popular power-attenuation model [133], a node sends out messages whose signal power drops as $1/r^{\alpha}$, where r is the distance from the transmitter antenna and α is a constant between 2 and 4. Therefore sending a

message to far away nodes will cost a substantial amount of energy. Usually in an *ad hoc* network, each node has a fixed transmission range so that only the nodes within the range can receive the messages. Such a range can be modeled by a disk with fixed radius centered at the node. Packets to the nodes outside the communication range are forwarded by multi-hop routing. What's especially interesting in an *ad hoc* network is that there is no fixed infrastructure nor a central server: all the nodes act as routers to discover and maintain routes to other nodes.

There are different kinds of *ad hoc* networks proposed for various applications. Sensor networks are one class of wireless *ad hoc* networks, where a node is equipped with sensors to sense light, sound, humidity, chemical materials or other physical quantities. Widely deployed sensors would greatly extend our ability to monitor and control the physical environment from remote sites. Another example is the rooftop network, which is a static network with nodes placed on top of buildings, to be used when wired networks fail. Inter-vehicle networks let vehicles moving on the road to communicate with each other and learn about road or traffic conditions [74]. Other applications of *ad hoc* networks are emergency rescue tasks, target tracking and data acquisition in inhospitable or human inaccessible environments [111].

Although *ad hoc* networks vary a lot in terms of equipment and applications, they have some common characteristics which are very different from traditional wired networks. Firstly, ad hoc networks are usually deployed in highly dynamic environments where fixed networks are not feasible. The nodes in an ad hoc network can be changed dynamically (turned on/off arbitrarily) and can move continuously. Therefore the underlying topology of the network changes constantly. Secondly, the wireless nodes are normally energy and memory constrained devices, such as PDAs, cell phones, pagers and battery-powered sensors. These hardware constraints prevent the nodes from performing costly operations as those explored by the traditional wired networks.

The differences of ad hoc networks from wired networks reveal fundamental challenges that require new insights on the design of algorithms and architectures. We use the routing problem as an example. On one hand, the routing table scheme used by the Internet, which lets each router keep the full routing information, is not feasible in

a wireless *ad hoc* network for two reasons. First, a wireless node does not have enough memory to hold the entire routing table. Second, the unstable, constantly changing topology makes shortest path routing very inefficient — shortest paths change too frequently and the control messages of updating the routing tables dominate the information transmission in the network. This furthermore drains out the limited energy of the wireless nodes before any meaningful tasks can be done. On the other hand, the wireless networking community has proposed several routing schemes for ad hoc networks [134]. For *ad hoc* networks with infrequent changes, variants of the routing table schemes have been explored. For networks with frequent changes, on-demand routing [121, 123, 134] has been proposed. In on-demand routing, a node initiates communication to another node and floods the network to discover a path, which is cached for later use. This approach doesn't require any routing table to be constructed and memorized. However, flooding is very expensive, and the lifetime of a cached path is very short if the nodes move fast. To summarize, the routing table approach is too rigid for the wireless networks, the on-demand routing is somewhat too "*ad hoc*" to exploit the possibilities of reusing the resources.

Between the two extremes, for example, the routing table scheme and the on-demand routing, one can extract and maintain some underlying structures of the mobile networks – although the nodes move continuously, the critical changes of the topology can be captured by some combinatorial structures that only change at discrete times. Therefore by maintaining such structures, one can keep track of the topology of the network more efficiently, while still providing near optimal performances. Such kind of algorithms and data structures must be **local, lightweight, smooth, and stable**. Local algorithms are always desirable in environments without central control. Lightweight structures are desirable for the reason of scalability. The structures need to be smooth such that as nodes are inserted/deleted or moving around, we can incrementally update the structures instead of compute everything from scratch. Furthermore, we want stable structures that don't change too many times as the nodes are inserted/deleted or moving around.

1.1 Overview

The dissertation has three parts. The first part includes a problem of clustering the nodes, followed by its application of constructing a spanner routing backbone. The second part of the dissertation is about load balanced routing in ad hoc networks. The third part of the dissertation is about clustering the pairwise distances.

The first part starts from a very fundamental problem of clustering the nodes. Specifically, we want to find discrete clusterheads in a mobile *ad hoc* network, so that any node can communicate with at least one clusterhead directly. The clusterheads allow hierarchical structures to be built on the mobile nodes and enable more efficient use of scarce resources, such as bandwidth and power. We propose an algorithm that generates stable clusterheads of good quality. First, our solution is a constant approximation of the optimal solution, in the sense that the number of clusterheads is at most a constant times the minimum number possible. Secondly, the clusterheads evolve smoothly as the nodes move around. The minimum number of changes of the clusterheads is bounded by a log log factor of the minimum number of changes of *any* clusterheads with a constant approximation ratio. One thing to emphasize is that the density of the clusterheads, i.e., the maximum number of clusterheads inside any unit disk, is bounded by a constant. This property is used to construct a linear-size "backbone" subgraph in the mobile wireless network so that both the number of hops and the Euclidean length of the shortest path on the backbone graph are at most a constant times those in the original communication graph. This subgraph enjoys another nice property that no two edges are crossing each other and therefore can be used in geographical greedy routing for a packet to get out of local minima.

The second part of the dissertation studies load balanced routing in a wireless network. In general there are two goals that people want to achieve on routing: load balancing, i.e., no node is highly congested, and low delay, i.e., the routing path is as short as possible. The load balancing competitive ratio (the ratio of the maximum congestion on any node compared with the optimum off-line algorithm) and the stretch factor (the maximum ratio of the length of a routing path compared with the shortest path length) represent how well we can achieve the two goals respectively.

The shortest path routing in a network minimizes the delay of a message from source to destination, but doesn't consider fairness. It is, however, desirable to distribute the load as evenly as possible to avoid overloading anyone. In general these two goals are in conflict with each other. However, when the wireless nodes are distributed on a line or in a narrow band, we show by a distributed algorithm that both a constant load balancing ratio and a constant stretch factor can be obtained simultaneously. In addition, if the nodes are distributed in the plane, there is a non-trivial tradeoff of the two factors in terms of the density of the nodes.

The third part of the dissertation focuses on clustering the pair-wise distances, i.e., approximating the quadratic number of pairwise distances of the nodes by a linear size structure. Thus all the proximity problems that involve all pairs distances can be approximately answered by a linear size structure. We study two problems in this category. The first problem is to represent the pairwise shortest path distances in a unit disk graph by an almost linear-size data structure called the well-separated pair decomposition. A well separated pair is a pair of sets, blue points and red points, so that the shortest path distance between any blue/red pair doesn't differ by a factor $(1+\varepsilon)$, for any $\varepsilon > 0$. Thus the shortest path distances between all the blue/red pairs can be approximated by a single blue/red pair. A well-separated pair decomposition is simply a set of well separated pairs that "cover" all pairs of points such that for any pair of points, there is a well separated pair that covers it. We show that a well separated pair decomposition with a almost linear number of pairs can be computed in almost linear time. This result has many applications in approximating the (bichromatic) nearest/furthest pair, diameter, median, and minimum spanning/steiner tree.

The second result in this category is to represent the pairwise distances of n points in the Euclidean space by a linear size graph, such that the shortest path distance of any pairs of points on the graph is at most a factor $(1 + \varepsilon)$ of the Euclidean distance, for any $\varepsilon > 0$. Such a graph is called a $(1 + \varepsilon)$-spanner. The advantage of our spanner, compared with the numerous spanners proposed previously, is that it is adaptive under motion. Thus this spanner has numerous applications under the moving domain. For example, wireless communication is usually short range, so the proximity information captured by the spanner can be used to track the near

neighbors of all the nodes. For another example, we can predict the collision of the moving nodes by inspecting the spanner edges only. As a bonus feature, the spanner implies a $(1+\varepsilon)$-WSPD and an 8-approximate k-center, for any given k. This is also the first time we know how to maintain a WSPD and an approximate k-center for moving points.

The very special properties of the wireless *ad hoc* networks raise challenges in algorithm design and implementation. The data structures and algorithms in this dissertation share a common property – they all have hierarchies. Hierarchies are nice divide-and-conquer structures so that every small piece in this structure only takes care of a small number of other pieces. This feature makes the data structures fit into the very dynamic and continuously changing setting, since updates to a small piece can be repaired locally and won't trigger cascading behavior. With only simple and loose interactions between the components, the hierarchical structures achieve scalability and robustness in a mobile environment.

1.1.1 References

Parts of the materials included in this dissertation have been or will be published in international conferences and journals, and thus are under copyright. Here is a list of them.

Chapter 3 is the work coauthored with Leonidas Guibas, John Hershberger, Li Zhang and An Zhu. A paper with title "Discrete Mobile Centers" has been published in the proceedings of the 17th ACM Symposium on Computational Geometry, pages 188–196, June 2001, and Discrete and Computational Geometry, 30(1), 45–65, 2003.

Chapter 4 is the work coauthored with Leonidas Guibas, John Hershberger, Li Zhang and An Zhu. A paper with title "Geometric Spanner for Routing in Mobile Networks" has been published in the proceedings of the 2nd ACM Symposium on Mobile Ad Hoc Networking & Computing, pages 45–55, October 2001, and will appear in IEEE Journal on Selected Areas in Communications Wireless Ad Hoc Networks in 2005.

Chapter 5 is the work coauthored with Li Zhang. A paper with title "Load

Balanced Short Path Routing in Wireless Networks" has been published in the proceedings of the 23rd Conference of the IEEE Communications Society (INFOCOM) in 2004.

Chapter 6 is the work coauthored with Li Zhang. A paper with title "Tradeoffs between Stretch Factor and Load Balancing Ratio in Routing on Growth Restricted Graphs" appeared in the proceedings of the ACM Symposium on Principles of Distributed Computing, pages 189–196, July, 2004.

Chapter 7 is the work coauthored with Li Zhang. A paper with title "Well-Separated Pair Decomposition for the Unit-Disk Graph Metric and its Applications" has been published in the proceedings of the 35th ACM Symposium on Theory of Computing, pages 483–492, June, 2003.

Chapter 8 is the work coauthored with Leonidas Guibas and An Nguyen. A paper with title "Deformable Spanners and Applications" has been published in the proceedings of the 20th ACM Symposium on Computational Geometry, pages 190–199, June, 2004.

Chapter 2

Background and Related Work

This chapter includes the notations and definitions that will be used in this dissertation, as well as the related work in both theory community and mobile wireless networking community. We will introduce the Kinetic Data Structure framework, which is used to analyze algorithms for moving objects and survey the related work on two topics, collaborating mobile devices and routing on ad hoc mobile networks.

2.1 Notations, definitions and basic structures

Approximation algorithms An *optimization problem* Π consists of a set $\mathcal{I}$ of inputs and a cost function C. Every input I is associated with a set of feasible outputs $F(I)$. Every solution $O \in F(I)$ has a cost $C(I, O)$. Given an input I, a minimization (maximization) optimization problem tries to find the solution O^* with the minimum (maximum) cost $C(I, O^*)$. Given any legal input I, an algorithm $\mathcal{A}$ for an optimization problem Π computes a feasible output $\mathcal{A}(I) \in F(I)$. An optimal algorithm OPT for a minimization problem is an algorithm such that for all legal inputs, $\mathrm{OPT}(I) = \min_{O \in F(I)} C(I, O)$. An algorithm $\mathcal{A}$ is a c-approximation algorithm if there is a constant $\alpha \geq 0$ such that for all legal inputs, $\mathcal{A}(I) - c \cdot \mathrm{OPT}(I) \leq \alpha$. The definition of an optimal and approximation algorithm for a maximization problem are symmetric. The ratio c is also called an *approximation ratio*.

An optimization problem where the input is received in an online manner and the

output must be produced online is called an *online problem*. The corresponding algorithm is called an online algorithm. An online algorithm doesn't have the information about the future input and produces output based solely on the current input and the past history. To measure the performance of an online algorithm, we compare the cost of the output with the optimal off-line algorithm that can use the information about the entire input. We call an online algorithm $\mathcal{A}$ a c-competitive algorithm if there is a constant α such that for all finite input sequence I, $\mathcal{A}(I) \leq c \cdot \mathrm{OPT}(I) + \alpha$. Similarly, the ratio c is called the *competitive ratio* of $\mathcal{A}$.

Metric A *metric* π is a distance function $\pi(\cdot, \cdot)$ defined on a (finite or infinite) set of points S, with the following properties:

1. $\forall u \in S$, $\pi(u, u) = 0$;
2. $\forall u, v \in S$, $\pi(u, v) = \pi(v, u)$;
3. Triangle Inequality: $\forall u, v, w \in S$, $\pi(u, w) \leq \pi(u, v) + \pi(v, w)$.

Suppose that (S, π) is a metric space where S is a set of elements and π is the distance function defined on $S \times S$. For any subset $S_1 \subseteq S$, the *diameter* $D_\pi(S_1)$ (or $D(S_1)$ when π is clear from the context) of S is defined to be $\max_{s_1, s_2 \in S_1} \pi(s_1, s_2)$. The *distance* $\pi(S_1, S_2)$ between two subsets $S_1, S_2 \subseteq S$ is defined to be $\min_{s_1 \in S_1, s_2 \in S_2} \pi(s_1, s_2)$.

Euclidean space Denote by $\mathbb{R}^d$ the d-dimensional Euclidean space. For a point $p \in \mathbb{R}^d$, denote by $\overrightarrow{p}$ the vector from the origin to the point p. When there is no confusion, we use p and $\overrightarrow{p}$ interchangeably. The L_p norm of a vector $\overrightarrow{x} = (x_1, x_2, \cdots, x_d)$ is defined as $|\overrightarrow{x}|_k = (\sum_{i=1}^{d} |x_d|^p)^{1/p}$. The L_p distance between two points $x, y \in \mathbb{R}^d$ is defined as $|xy|_p = |\overrightarrow{x} - \overrightarrow{y}|_p$. L_p distance is a metric for any $1 \leq p < \infty$. The L_2 distance is normally written as $|xy|$.

Graph A *graph* $G = (V, E)$ defines a set of *edges* E on a set of *vertices* S, such that each edge $e \in E$ is a 2-tuple $e = (u, v)$, $u, v \in V$. A graph $G' = (V, E')$ is a *subgraph* of $G = (V, E)$ if they are defined on the same vertex set and $E' \subseteq E$.

The graph can be weighted or unweighted. For a weighted graph, each edge e is associated with a weight $w(e) \geq 0$. For an unweighted graph, each edge e is given a weight 1. We denote by $P_G(u, v)$ the shortest path between u, v in the graph G. A graph is *simple* if there are no self-loops and parallel edges. In this dissertation we only consider undirected graphs such that the edge (u, v) and (v, u) are the same. A graph is *connected* if for any two vertices $u, v \in S$ there is a path $P(u, v)$ connecting them. In this dissertation, we will focus on simple connected graphs most of the time. We define the graph metric, i.e., the shortest distance metric π_G, of a graph G such that $\pi_G(u, v)$ is the length of the shortest path $P_G(u, v)$. The subscript is omitted when there is no confusion.

Unit-disk graph For a set of points S in the plane, the *unit-disk graph* $I(S) = (S, E)$ is defined to be the graph where an edge $e = (p, q)$ is in the graph if $|pq| \leq 1$. The graph can be either weighted, i.e., the weight of an edge (p, q) is $|pq|$, or unweighted. Likewise, we can define the unit-ball graph for points in higher dimensions. The *unit-disk graph metric* $\pi = \pi_{I(S)}$ is the shortest distance metric induced by the unit-disk graph of a set of points S in the plane. For a weighted unit-disk graph $I(S)$ and two vertices $u, v \in S$, we denote by $\pi_I(u, v)$ the Euclidean length of the shortest path connecting u and v in $I(S)$. For an unweighted unit-disk graph, we denote by $\tau_I(u, v)$ the topological length, i.e., the number of hops of the shortest path connecting u and v.

Delaunay triangulation and Voronoi diagram For a set of sites in the plane, the *Voronoi diagram* partitions the plane into convex polygonal faces such that all points inside a face are closest to only one site. The *Delaunay triangulation* is the dual graph of the Voronoi diagram, obtained by connecting the sites whose faces are adjacent in the Voronoi diagram. For an edge xy, there is an *empty-circle* rule to determine whether xy is a Delaunay edge: xy is a Delaunay edge if and only if there exists a circle that contains no other points except x, y. Figure 2.1 shows an example of the Voronoi diagram and Delaunay triangulation of a set of points. Delaunay triangulation and Voronoi diagram are classical geometric structures and

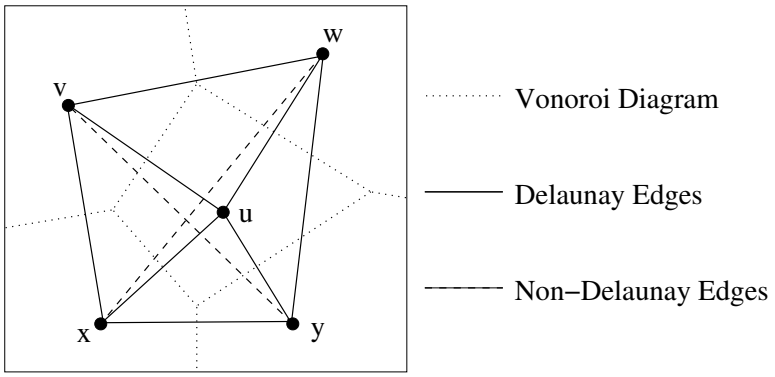


Figure 2.1. Voronoi diagram and Delaunay triangulation of a set of points.

have numerous applications [128].

Growth restricted graphs Graphs with restricted growth rate arise in many practical networks, either due to physical constraints such as in wireless networks and VLSI layout networks, or due to geographical constraints such as in peer-to-peer overlay networks [117, 125, 86, 97]. An unweighted graph has *density* ρ and *growth rate* k (or growth dimension k) if for any vertex v and any $r > 1$, $|B_r(v)| \leq \rho r^k$, where $B_r(v) = \{u|\tau(u,v) \leq r\}$, the ball with radius r centered at v. For example, in a wireless ad hoc network, when the maximum density of the nodes, i.e., the maximum number of nodes covered by a unit disk, is a constant, the unit disk graph has growth rate 2. As another example [117], the Internet network distance defined by round-trip propagation and transmission delay forms a metric with restricted growth rate. Therefore, many algorithms for peer-to-peer networks, such as media file sharing on Internet, content addressable overlay networks, exploit this property [125, 86, 97].

There are several other stronger definitions for capturing metrics with slow growth. For example, a metric defined by the shortest path distances for an unweighted graph has *expansion rate* k (or KR-dimension $\log k$) if $|B_{2r}(v)| \leq k|B_r(v)|$; and a metric has *doubling constant* k (or doubling dimension $\log k$) if $B_{2r}(v)$ is contained in the union of at most k balls with radius r[1]. For metrics induced by unweighted graphs,

[1]We use similar definitions as in [86] and [13]. The original definitions are for general metrics. Here we focus on shortest path metrics defined by unweighted graphs.

both definitions imply that the size of B_r is bounded by $O(k^{\log r}) = O(r^{\log k})$. On the other hand, we can construct a family of graphs, e.g. the comb graphs as shown in Figure 2.2, with constant density and growth rate but unbounded KR-dimension and unbounded doubling dimension. Therefore, for unweighted graphs our definition is broader in the sense that any unweighted graph with KR-dimension or doubling dimension d also has a growth dimension d.

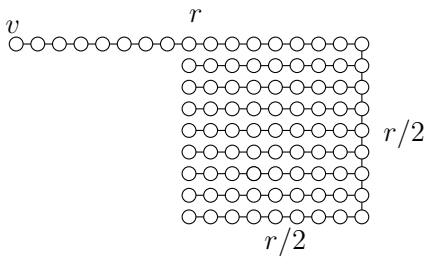


Figure 2.2. The "comb" graph is a unit disk graph with constant bounded density. Therefore it's a graph with growth rate 2. It is not a metric with constant KR-dimension since $|B_{2r}(v)| = \Theta(r^2)$ and $|B_r(v)| = \Theta(r)$. And, it is not a metric with constant doubling dimension: the comb graph has diameter $2r$, it can not be covered by a constant number of balls with diameter r, since the diameter of a set including two teeth of the comb is at least $r + 1$.

2.2 Kinetic data structures

Since this dissertation is about design and analysis of algorithms for moving objects, in this section we will review the Kinetic Data Structure (KDS in short), proposed by Basch, Guibas and Hershberger [28], as a framework to quantify and compare algorithms on objects under motion.

The traditional method to handle moving objects is to sample time discretely and delete/reinsert the objects at new positions for each time step. There are several problems with the discrete sampling method. Firstly, the incremental updating method only gets limited benefit from the coherence of continuous motion. In addition, the sampling rate is hard to control. A small rate results in waste of computation resources. By using a large sampling rate one may possibly miss the events where critical reconfigurations take place.

Compared with the discrete sampling method, a KDS only does work when necessary. The KDS framework assumes that each object follows a publicly posted *flight plan* specifying its motion. It can be either computed algebraically or estimated by interpolation. The KDS employs a novel idea of maintaining a set of proofs which implies the correctness of the desired structure. The proofs are usually combinatorial conditions, called *certificates*. Each certificate is associated with an earliest failure time. When a certificate fails, we say an *event* happens and the KDS certificate repair mechanism is invoked to update the structure, as well as the set of certificates if necessary. If only the certificates need to be updated, the event is denoted as an internal event. Otherwise, it's called an external event. In a typical case, the flight plans are polynomial or rational trajectories and each certificate is a simple algebraic inequality, so the certificate failure time is the next real root of some low-degree polynomials. The set of certificates are organized in a priority queue, according to their first failure times so that the root of the priority queue always contains the certificate with the earliest failure time. Figure 2.3 shows the KDS engine.

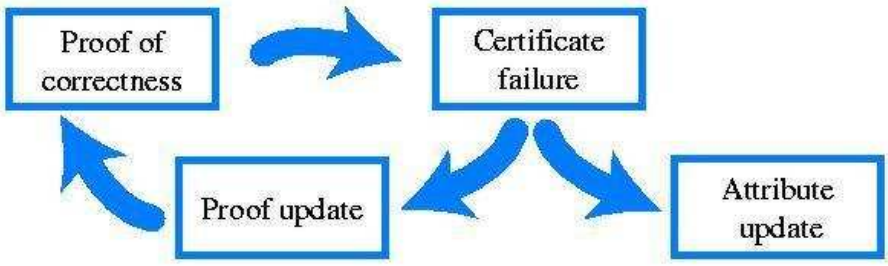


Figure 2.3. The KDS engine, from [27].

The amount of work that a KDS has to do highly depends on the properties of the motion. The motion of a point p is represented by a curve $p(t)$ parametrized by time t. The initial position is denoted as $p(0)$. If the curve $p(t)$ is a pseudo algebraic function with a constant degree, we say the point p moves along pseudo algebraic motion. If a certificate in a KDS only depends on a constant number of points and each point follows pseudo algebraic motion, the same certificate can switch from TRUE to FALSE at most a constant number of times. This observation is very important in the analysis of the efficiency of a KDS.

The challenge of designing a good KDS is to find a set of certificates that evolves gracefully over time. The key point of an efficient KDS is that it captures exactly the critical events that may happen very irregularly through time. The performance of a KDS depends on the number of rigidly moving objects or object parts, denoted by n, as well as the number of flight plan updates. Specifically, there are four criteria for evaluating a KDS.

Responsiveness The responsiveness of a KDS measures the worst-case amount of time needed to update its certificates after an event happens. This requires discovering whether a certificate fails, removing invalid certificates, recomputing new certificates, and updating the maintained structure if necessary. A KDS is called *responsive* if the worst-case update time is of the order $O(\text{polylog}(n))$ or even $O(n^{\varepsilon})$ for some arbitrarily small $\varepsilon > 0$.

Efficiency The efficiency captures how many events a KDS processes, compared with the number of internal events, which provides a lower bound on the cost of any algorithm for maintaining the structure. To be more specific, we call the weak efficiency the ratio of the maximum number of events over all allowed motion over the maximum number of external events. We also call the strong efficiency the worst-case ratio of the total number of events processed to the number of external events, taken over all allowed motion. A KDS is called *weakly/strongly efficient* if the weak/strong efficiency is of the order $O(\text{polylog}(n))$ or even $O(n^{\varepsilon})$ for some arbitrarily small $\varepsilon > 0$.

Locality When an object changes its flight plan, all the certificates in which the object is involved need to be re-evaluated for the first failure time. The locality measures the maximum number of certificates in which one object appears, and therefore the update cost of a flight plan change. A KDS is *local* if this number is of the order $O(\text{polylog}(n))$ or even $O(n^{\varepsilon})$ for some arbitrarily small $\varepsilon > 0$.

Compactness The compactness measures the storage necessary for a KDS. We call a KDS *compact* if the total number of certificates ever present in a proof is almost linear, i.e., of order n times a small quantity.

Since the notion of kinetic data structures was introduced in 1997 [28], many kinetic data structures for moving objects have been designed and analyzed, with applications in various domains: computer graphics, physical simulation, robotics, computational biology, temporal databases and mobile networks [28, 71, 6, 8, 9, 30, 89, 5, 2, 1, 3, 69, 90, 77, 58, 7]. Practical evaluation was also done on the performance of kinetic data structures compared with the traditional sampling method [29].

To further illustrate the ideas in Kinetic Data Structures, we will describe kinetic sorted list as an example of the KDS, which will also be used in a later chapter.

Kinetic sorted list For a set of points $p_1, p_2, \cdots, p_n$ moving on the real line, the problem is to maintain the linear order of the points. The idea lies in the classic line sweeping method. To initialize, we first sort the points $p_1(0), p_2(0), \cdots, p_n(0)$ at time 0. Assume the current sorted sequence is $p'_1, p'_2, \cdots, p'_n$. The certificates are $n-1$ pairs of comparisons: $p'_1 < p'_2, p'_2 < p'_3, \cdots, p'_{n-1} < p'_n$. When a certificate $p'_i < p'_{i+1}$ fails, $1 \leq i \leq n-1$, the sorted list is updated by exchanging p'_i and p'_{i+1}. The certificate set is updated by deleting $p'_{i-1} < p'_i, p'_i < p'_{i+1}, p'_{i+1} < p'_{i+2}$ and adding $p'_{i-1} < p'_{i+1}$, $p'_{i+1} < p'_i$ and $p'_i < p'_{i+2}$.

The structure takes $O(n)$ space. Each event is involved with $O(1)$ certificates. And it costs $O(\log n)$ to update the priority queue. So the update cost at any event is $O(\log n)$. Each point is only involved with at most 2 certificates. All the events are external events. The total number of events is bounded by $O(n^2)$ since any two points can exchange their ordering at most a constant number of times if the points follow pseudo algebraic motion. Therefore the kinetic sorted list is compact, local, efficient and responsive.

2.3 Collaborating mobile devices

Collaborating mobile devices are of interest in diverse applications, from wireless networking to robot exploration. In these applications there are mobile nodes that need to communicate as they move so as to accomplish the task at hand. These tasks can vary from establishing an *ad-hoc* multi-hop network infrastructure that allows

point-to-point communication, to aggregating and assimilating data collected by distributed sensors, to collaboratively mapping an unknown environment. A challenge common to all these tasks is that communication is usually accomplished using low-power radio links or other short-range technologies. Since only nodes sufficiently close to each other can communicate directly, the communication topology of the network is strongly affected by node motion and obstacle interference. The mobile networking community has been especially active in studying such problems in the context of networking protocols allowing the seamless integration of devices such as PDAs, mobile PCs, phones, pagers, etc., that can be mobile as well as switched off and on at arbitrary times. In this section, we will review the related work on two problems in collaborating mobile devices, the mobile clustering problem and proximity maintenance.

2.3.1 Mobile clustering

A principle that has been discussed a number of times for enabling collaborative tasks is the organization of the mobile nodes into *clusters* [26, 45, 60, 140]. Clustering allows hierarchical structures to be built on the mobile nodes and enables more efficient use of scarce resources, such as bandwidth and power. For example, if the cluster size corresponds roughly with the direct communication range of the nodes, much simpler protocols can be used for routing and broadcasting within a cluster; furthermore, the same time or frequency division multiplexing can be re-used across non-overlapping clusters. Clustering also allows the health of the network to be monitored and misbehaving nodes to be identified, as some nodes in a cluster can play watchdog roles over other nodes [112].

Formally, assume we have n wireless nodes each with a fixed communication range as a unit disk, the clustering problem in ad hoc network setting is to find a minimum subset of the nodes, called the *clusterheads*, such that every node can communicate to at least one clusterhead. In the mobile device setting, unlike the general facilities location context, it is appropriate to insist that the clusterheads are located at the nodes themselves, as these are the only active elements in the system; thus we are

interested in "discrete clusterheads" problems.

There is a huge literature on clustering, as the problem has been studied in many variations by several different communities, including operations research, statistics, and computational geometry. First of all, even in the static case, the discrete centers problem is hard. The static version of the discrete centers problem is known to be NP-complete [57]. In fact, it is equivalent to finding the minimum dominating set in the intersection graph of unit disks, defined as follows. The dominating set problem is defined as follows. We build a graph G on all the points and create an edge between two points within distance 1, find a minimum size subset V' of vertices, such that every vertex in $V \setminus V'$ is adjacent to some node in V'. In the theory community, most work has focused on approximation algorithms. The static version of the problem admits a polynomial time approximation scheme (PTAS) [79]. But existing algorithms for the static version cannot be adapted to the mobile case efficiently. Many such algorithms use space partition methods, i.e., they partition space into smaller subregions and solve for each region separately. For instance, one can design a simple constant approximation algorithm by choosing one clusterhead out of every pre-fixed grid square of length $\sqrt{2}/2$. Algorithms of such flavor totally ignore the underlying topology of the node set and, as a result, suffer from many unnecessary solution changes during node motion. For example, if nodes are travelling together with the same velocity, the partition based algorithms will move the nodes from voxel to voxel and select new clusterheads, which is in fact not necessary at all.

Despite the numerous work on static clustering, much less is known, however, about maintaining a clustering on mobile nodes. Sariel Har-Peled proposed a static clustering of moving points, which is compared with the optimal k-center solution at any instance [73]. The k-center problem is to find a subset of k centers such that the maximum distance from any point to the k centers is minimized. Har-Peled shows that for a point set moving with polynomial motion of degree μ, one can find $k^{\mu+1}$ centers such that the maximum radius is at most a constant factor of the radius of the optimal k center of the point set at any time. This clustering is static and the computation requires the full knowledge of the moving trajectories. John Hershberger proposed a kinetic algorithm for maintaining a covering of the moving points in $\mathbb{R}^d$

by unit boxes such that the number of boxes is always within a factor of 3^d of the optimal static covering at any instance [75]. The central idea is to maintain a smooth covering of points on a line by unit intervals and the clustering on higher dimensions is done by projecting points along each axis. Therefore there is fundamental difficulty to extend this method to find approximate covering of the points by unit disks.

In the networking community, there are a number of heuristics for cluster maintenance in the literature, for example, the lowest ID and highest degree algorithms. But to our knowledge there has been very little work on theoretical analysis of the algorithms. The most famous algorithm is the lowest ID clustering algorithm, where each node selects the one with lowest ID inside its communication range. It was originally proposed by Ephremides, Wieselthier and Baker [51] and then revisited for ad hoc mobile networks [60]. Implementation details and comparison with other clustering methods were reported in [108, 44]. Chiang *et al.* have shown that the lowest ID algorithm performs well in terms of stability of clusters [45]. The lowest ID algorithm was also augmented to incorporate various mobility models to produce more stable clusters [31, 26, 32]. A similar idea leads to the Max-Min D-clustering scheme where a clusterhead can cover nodes up to d hops [11]. Another category of clustering algorithm is the highest degree algorithm proposed by Gerla and Parekh [60, 120], where a node selects the one with highest degree inside its communication range as a clusterhead. Experiments demonstrate that the throughput of the system drops and performance degrades as the number of nodes in a cluster is increased. There are other heuristics that assign weights to the nodes and apply the lowest weight heuristic to find a set of clusterheads [43, 155].

2.3.2 Proximity maintenance

Another important component in collaborating mobile devices is to maintain the proximity information between them. Since the mobile nodes, such as manned or unmanned vehicles (cars, airplanes) or people carrying cell phones or other communication devices, are physical objects in the real world, maintaining proximity information between them is an important problem for many application scenarios.

In the vehicle case, collision avoidance is an important consideration where proximity information plays a crucial role. In another setting, a flying aircraft may want to find the closest aerial tanker to get fuel. In the control of teams of collaborating vehicles, as in formation flying, each vehicle must know its near neighbors and be aware of their motion in order to plan its own. A communicating search team in a rescue operation may need proximity information among team members in order to guarantee exhaustive coverage of the search space. Knowledge of proximity is also essential for building and maintaining communication infrastructure within the ad hoc communication network. Mobile nodes will typically use wireless transmitters whose range is limited. Proximity information is essential for topology maintenance, as well as for the formation of node clusters and other hierarchical structures that may aid in the operation of the network.

Because of its central significance, maintaining proximity information among moving objects has been a topic of study in various domains, from robot dynamics and motion planning, to physical simulations across a range of scales from the molecular to the astrophysical. We regard collision detection as a special case of proximity maintenance — indeed many extant approaches to collision detection already noted the similarity between that task and that of distance estimation between objects [109]. Many data structures have been designed for collision detection. The standard approach is based on bounding volume hierarchies. Examples of standard simple bounding volumes include the convex hull, the (smallest, axis-aligned) bounding box, and the (smallest) bounding sphere. A bounding volume hierarchy is simply a tree of bounding volumes. The bounding volume at a given node encloses the bounding volumes of its children. The collection of the bounding volumes at the leaves covers the object. In other words, a bounding volume hierarchy approximates an object by collections of bounding volumes across different scales. While the bounding volume hierarchies work well for the interaction between a small number of moving rigid objects, it does not scale to solve the collision detection between a large number of objects, nor objects that deform. Some efforts towards better deformable bounding volume hierarchies for "linear" objects are the kinematic chains of Lotan *et al.* [110], where the hierarchy allows for quick updates after a single, or few, joints in the chain

move, and the combinatorial sphere hierarchy of Guibas *et al.* [70], where bounding spheres are defined implicitly through feature points on the surface of an object. Both of these structures can perform intersection tests in $O(n^{4/3})$ time in 3-D. A different technique is to use deformable tilings of the free space among moving objects [3], but this is currently limited to 2-D.

We are not aware of much general work on maintaining proximity information in the ad hoc networking community. One related problem is to keep track of the 1-hop neighbors of each node, i.e., the nodes within communication range for every node. Tracking 1-hop neighbors is a fundamental problem that has applications in routing protocols and the overall organization of the network. A usual protocol for topology discovery is to ask all the nodes to send out "hello" beacons periodically. The nodes who receive the beacons respond and thus neighbors are discovered. However, a critical issue in this method is to choose the rate of the "hello" beacons. A high beacon rate relative to the node motion will result in unnecessary communication costs, as the same topology will be discovered many times. A low rate, on the other hand, may miss important topology changes that are critical for the connectivity of the network. As also happens in physical simulations, the maximum speed of any node usually gates this rate for the entire system.

In this dissertation proximity maintenance is studied in two settings, the geometric setting and the graph setting. In the geometric setting, we focus on the Euclidean distances between the mobile nodes. In the graph setting, we measure the distances of the nodes by the shortest path distances in the communication graph.

2.4 Routing in *ad hoc* mobile networks

The eternal goal and the most fundamental problem for any types of networks is to enable efficient information transmission between the peers. A routing protocol, which establishes routes between a pair of nodes efficiently and correctly so that messages can be delivered in a timely manner, is an important component of a network architecture.

The routing protocols for wired networks, e.g., the Internet, has shown to be very

successful and influential. However, they are not suitable for wireless ad hoc networks for a number of reasons. Firstly, the Internet has a carefully designed backbone on powerful routers, which have enormous amount of memory and unlimited supply of energy. It also has a nice embedded hierarchical architecture and few dynamic changes. All these assumptions are no longer true for an ad hoc mobile network, which has no centralized or fixed infrastructure, and is built on nodes that are highly dynamic and resource constrained.

There has been lots of work on routing protocols in *ad hoc* mobile networks [134]. The routing protocols are generally categorized as table-driven and source-initiated on-demand protocols. Figure 2.4 illustrates the classification of the current routing protocols.

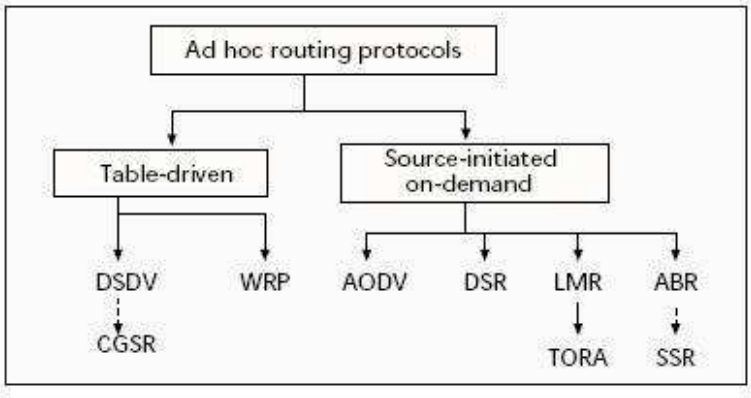


Figure 2.4. The categorization of *ad hoc* routing protocols, solid lines represent direct descendants, dotted lines represent logical descendants. The figure is from [134].

Table-driven protocols Table-driven routing protocols try to maintain consistent and up-to-date information on each node in the network. The protocols require each node to keep one or more tables to store routing information, similar to the routing tables in traditional wired networks. Under topological changes, the nodes propagate updates throughout the network to maintain a consistent network view. This category includes Destination-Sequenced Distance-Vector Routing (DSDV), Clusterhead Gateway Switch Routing (CGSR) and the Wireless Routing Protocol (WRP). As mentioned earlier, these protocols have high requirement on the resources of the

wireless nodes and can not be easily adapted for dynamic updates. Therefore they can't scale to large networks.

Source initiated on-demand protocols Unlike the table-driven protocols that always maintain consistent routing information throughout the network even when there are no routing requests, the source initiated on-demand protocols only create routes when desired by the source node. When a node requires a route to a destination, it initiates a route discovery process within the network. Once a route is established, it's maintained by a route maintenance procedure until either the destination becomes inaccessible or the route is no longer wanted. This category includes Ad hoc On-Demand Distance Vector (AODV), Dynamic Source Routing (DSR), Temporally Ordered Routing Algorithm (TORA), Associativity-Based Routing (ABR) and Signal Stability Routing.

The table-driven protocols guarantee that a route to every other node is always available, but incur substantial signaling traffic and power consumption. The source-initiated on-demand protocols are more lightweight in terms of control traffic, but a node has to wait until a route is discovered before it's able to communicate and the route discovery stage is usually expensive. People then start to look for routing algorithms that are more scalable and energy efficient. We will review the work on these two categories in the following. The first category uses geographic information to help routing. The second category aims on reducing the energy consumption of the routing algorithm.

2.4.1 Location-based routing

Location-based routing algorithms use the locations of the wireless nodes to perform efficient multi-hop routing. The wireless nodes can obtain location information by communicating with satellites if the nodes are equipped with GPS (Global Positioning System) receivers. Alternatively, for indoor environments where GPS doesn't work, the relative distances between neighboring nodes can be estimated on the basis of incoming signal strengths or time delays through direct communications. The relative distances can be used by various localization algorithms to calculate or estimate the

absolute or relative coordinates of the nodes [76, 62, 144, 61].

The advantage of location-based routing algorithms is their scalability, which is crucial in the applicability of large ad hoc networks. It has been experimentally confirmed that the routing protocols such as AODV, DSDV, DSR, are not as scalable as routing protocols that use geographic location information [82, 104]. Simple geographical forwarding [56] combined with GLS (scalable location service [104]) compares favorably with DSR. It delivers more packets and consumes fewer network resources. Furthermore, the performance of GLS degrades gracefully as nodes fail and restart, and is relatively insensitive to node speeds [104].

The reason for the scalability and efficiency of location-based routing algorithms is that they adopt *local algorithms*, i.e., each node makes the decision on which node to forward the packet to, based solely on the location of itself, the neighboring nodes, and the destination. Local algorithms are lightweight, robust and distributed in nature. In contrast, the shortest-path based algorithms require the knowledge of the complete network topology, whose maintenance cost is quadratic in the size of the network – each change in edge or node status (nodes switch on/off/sleep) may trigger routing table update in a large portion of the network. Location-based routing algorithms dramatically reduce such overhead.

Generally speaking, location-based routing protocols have two modes: the greedy mode and the recovery mode.

Greedy mode In this mode, the node currently holding the packet "advances" it towards the destination, based only on the location of itself, the neighbors and the destination. The advance may be defined in many ways. Examples are, Closest to Destination [56, 87], Most Forward within Radius (MFR) [149], Nearest Forward Progress (NFP), Nearest Closer (NC) [146], Geographic Distance Routing (GEDIR) [145] and Compass routing [96]. The most popular way of defining the advance is to examine the Euclidean distance to the destination. The greedy mode routing was shown to nearly guarantee delivery for dense networks, but fail frequently for sparse graphs. For example, the packet may reach a *local minimum* whose neighbors are all further away from the destination from itself.

Recovery mode When the greedy mode fails to advance a packet at some node, the routing process is converted to the recovery mode. The recovery mode defines how to forward the packet at a local minimum, in order to guarantee delivery of the packets. Some examples of the methods to get out of the local minimum are simple flooding [145], terminode routing [33], bread-first search or depth-first search [82], face algorithm [36] and perimeter routing [87].

Here we use a specific routing protocol, the Greedy Perimeter Stateless Routing (GPSR) Protocol [87], to explain the two modes in detail. In the greedy mode, a node forwards the packet to its neighbor who is closest to the destination. This greedy forwarding scheme was originally proposed by Finn [56]. The packet may reach a *local minimum* whose neighbors are all further away from the destination than itself. See Figure 2.5. To get out of the local minimum, the protocol maintains

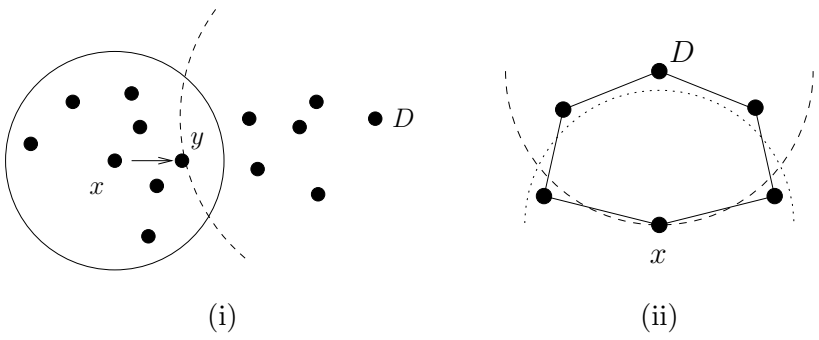


Figure 2.5. (i) In the greedy forwarding mode, x sends the packets to y, the closest neighbor to the destination D; (ii) x is a local minimum whose neighbors are all further away from the destination D than x itself.

a planar and connected subgraph, e.g, *Gabriel Graph* (GG) or *Relative Neighborhood Graph* (RNG) (see Figure 2.6). When a packet gets stuck, the recovery mode applies routing along the faces of the subgraph that intersect the imaginary line between the source and the destination (Figure 2.7). The advantage of this method is that the construction of the planar graphs, as well as the routing decisions in recovery mode, are both local. Furthermore, it guarantees the delivery of a packet to the destination if there is indeed such a way of doing that. This recovery mode has been independently discovered by Bose *et al.* as the face algorithm [36].

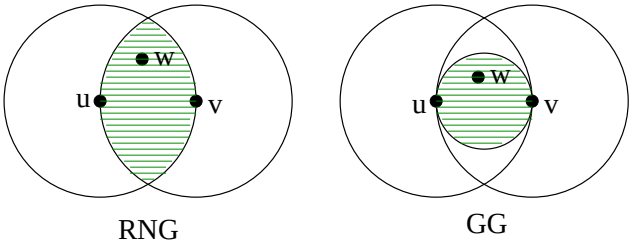


Figure 2.6. In RNG (GG), the edge uv is included if there are no nodes in the shaded area.

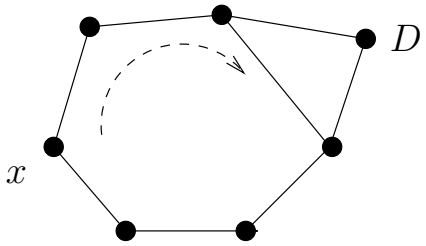


Figure 2.7. Perimeter routing along the face of a planar graph.

2.4.2 Energy-aware routing

Energy-aware routing algorithms take into account the energy constraint of a wireless ad hoc network. The nodes are usually energy constrained as they are normally powered by batteries. Many routing algorithms, for example, the widely used shortest path routing algorithm, aim to find short routing paths. Besides minimizing latency, the shortest path routing is good for overall energy efficiency because energy needed to transmit a packet is correlated to the path length. However, the algorithms that aim to minimize the path length may ignore "fairness" in the routing — for example, the shortest path routing is likely to use the same set of hops to relay packets for the same source and destination pair. This will heavily load those nodes on the path even when there exist other feasible paths. Such an uneven use of the nodes may cause some nodes die much earlier, thus creating holes in the network, or worse, leaving the network disconnected. In addition, unbalanced use of the nodes may discourage the nodes to participate in the routing.

However, load-balanced routing is a very difficult problem. For example, it is

NP-hard to compute the most balanced routes, even in a very simple network. Approximate algorithms have been developed for the problem. For the general case, the unsplittable flow problem where we aim to minimize the maximum vertex congestion is a well-known NP-hard problem that can be approximated to a factor of $O(\log n/\log\log n)$ [132, 131]. The dual problem of fixing the edge or vertex capacity and maximizing the total number of flows, with the disjoint path problem as a special case, has also been studied extensively [93, 94, 92, 95]. There is also extensive study on the on-line load-balancing problems [22, 34]. Among them the on-line virtual circuit routing problem is to find the best routes for each online routing request with the goal of minimizing the maximum congestion on any link [127]. Aspnes *et al.* [21] proposed an algorithm for the online virtual circuit routing problem that achieves $O(\log n)$ competitive ratio, which is also tight in the worst case. But none of the previous algorithms is local, i.e. they require global coordination, and the approximation ratio often has only theoretical interests.

In networking community, energy-aware routing algorithms, which try to maximize the network survivability, have also been studied a lot recently [24, 105, 64, 84, 124, 141, 41, 42, 65, 161, 152, 139, 143, 162, 158]. The energy aware metrics, such as "maximize time to partition" and "minimize maximum node cost", were first proposed by Singh *et al.* [141]. Chang *et al.* [41, 42] used a flow augmentation algorithm and a flow redirection algorithm to balance the energy consumption on different nodes. Their method, however, requires a full knowledge of traffic demands and does not handle node insertion and deletion. Extensions along this approach were addressed in [143, 162]. Li *et al.* [105] studied the online power-aware routing which minimizes the earliest time when a packet can not be sent. They proved that any online algorithm has unbounded competitive ratio and provided algorithms with zone-based heuristics. Stojmenovic *et al.* [146] and Yu *et al.* [161] proposed methods that use the geographical locations of wireless nodes for energy aware routing. Xu *et al.* [158] proposed an algorithm GAF which is designed to reduce the energy consumption by turning off unnecessary nodes. In [160, 72], the traditional energy-unaware routing protocols such as DSR [83] or AODV [123] were re-visited to take into account the energy-aware metric. One thing to note is that all of the energy-aware protocols mentioned above

are heuristics and do not provide any guarantee on the performance.

Part II

Clustering the Points

Chapter 3

Discrete Mobile Centers

3.1 Introduction

In this chapter we study the problem of maintaining a clustering for a set of n moving nodes in the plane. In our setting we assume that all the nodes are identical and each can communicate within a region around itself, which we take to be an L_p ball. For most of the chapter we will focus on a ball in the L_∞ metric, that is an axis-aligned square whose side is of length 1, as this makes the analysis the simplest. We call two nodes *visible* to each other if they are within the communication range of each other. We seek a minimal subset of the n nodes, the *clusterheads*, such that every node is visible to at least one of the clusterheads.

A good algorithm should only undergo solution changes that are necessary. Another desirable property is that the algorithm can be implemented in a distributed manner on nodes with modest capabilities, so as to be useful in the mobile *ad hoc* network setting. As nodes move around, we need efficient ways of incrementally updating the solution, based on local information as much as possible. We'll formalize such properties in our study.

In this chapter we present a new randomized clustering algorithm that provides a set of clusterheads that is an $O(1)$ approximation to the optimal discrete center solution. Our algorithm uses $O(\log \log n)$ rounds of a "clusterhead nomination" procedure in which each node nominates another node within a certain region around

itself to be a clusterhead; a round of the nomination procedure can be implemented in $O(n \log n)$ time. Furthermore, we show how this approximately optimal clustering can be maintained as the nodes move continuously. The goal here is to exploit the continuity of the motion of the nodes so as to avoid recomputing and updating the clustering as much as possible. We employ the framework of *Kinetic Data Structures* (KDS) [28, 71] to provide an analysis of our method. We show that the proposed structure is responsive, efficient, local, and compact.

To summarize, our clustering algorithm has a number of attractive properties:

- We can show that the clustering produced is an $O(1)$ approximation.
- The clustering generated by the algorithm is *smooth* in the sense that a point's movement causes only local clustering changes. Certificate failures and flight-plan updates can be processed in expected time $O(\log^{3.6} n)$ and $O(\log n \log\log n)$ respectively. This is in contrast to the optimal clustering solution, which may undergo a complete rearrangement upon small movements of even a single point.
- In the KDS setting, the algorithm also supports dynamic insertion and deletion of nodes, with the same update bound as for a certificate failure, in addition to the mentioned properties of our KDS.
- Under the assumption of pseudo-algebraic motions for the nodes, we show that our structure processes at most $O(n^2 \log\log n)$ events, i.e., certificate failures. We also give a construction showing that for any constant $c > 1$, there is a configuration of n points moving linearly on the real line so that *any* c-approximate set of clusterheads must change $\Omega(n^2/c^2)$ times. Thus, even though an approximate clustering is not a canonical structure [2], we can claim efficiency for our method.
- The algorithm can be implemented in a distributed fashion: each node only reasons about the nodes visible to it in order to carry out the clustering decisions. In fact, the algorithm can be implemented without any knowledge of the actual positions of the nodes—only knowledge of distances to a node's visible neighbors are necessary.

Because of these properties, our algorithm has many applications to *ad hoc* wireless networks. We defer the detailed discussion to Subsection 3.4.3.

The remainder of the chapter is organized as follows. Section 3.2 introduces the basic algorithm and analyzes the approximation factors for the clusterings it produces. Section 3.3 describes a hierarchical version of the algorithm and proves the constant approximation bound. Section 3.4 shows how this clustering can be maintained kinetically under node motion and analyzes the performance of the algorithm in both centralized and distributed settings.

3.2 Basic algorithm

Before presenting the algorithms, we first give some formal definitions. A d-cube with size r is a d-dimensional axis-aligned cube with *side length* r. When $d = 1$ or 2, a d-cube is also called an interval or a square, respectively. For two points p and q, p is said to be r-*covered* by or r-*visible* from q if p is inside the cube with size r centered at q. For a set of n points (nodes) $P = \{p_1, p_2, \ldots, p_n\}$ in the d-dimensional space, a subset of P is called an r-*cover* of P if every point in P is r-covered by some point in the subset. The points in a cover are also called (*discrete*) *clusterheads*. A minimum r-cover of P is an r-cover that uses the minimum number of points. We denote by $\alpha_P(r)$ (or $\alpha(r)$ if P is clear from the context) the number of points in a minimum r-cover of P. An r-cover is called a c-approximate cover of P if it contains at most $c \cdot \alpha_P(r)$ points. When r is not mentioned, we understand it to be 1. In this chapter, we are interested in computing and maintaining $O(1)$-approximate covers for points moving in the space. For the sake of presentation, we will discuss our algorithms for points in one and two dimensions, but our techniques can generally be extended to higher dimensions. In the rest of the chapter, log is understood to be $\log_2$, and ln to be $\log_e$, unless otherwise specified in the context[1].

In the following, we first present the algorithms for the static version of the problem and later describe their implementation for moving points.

[1]This distinction is important because in a few places, log appears in exponents, and we have to make the base explicit in order to give precise asymptotic bounds.

3.2.1 Description of the basic algorithm

The basic algorithm, which is distributed in nature, is the following: we impose a random numbering (a permutation of $1, 2, \ldots, n$) onto the n points, so that point p_i has an index N_i. In most situations in practice each mobile node is given a unique identifier (UID) at set-up time, and these UIDs can be thought of as providing the random numbering (either directly, or via a hash function on the UIDs). Each point p_i nominates the largest indexed point in its visible range to be a clusterhead. A point can nominate itself if there is no other point with larger index inside its range. All points nominated are the clusterheads in our solution. A cluster is formed by a selected clusterhead and all the points that nominated it.

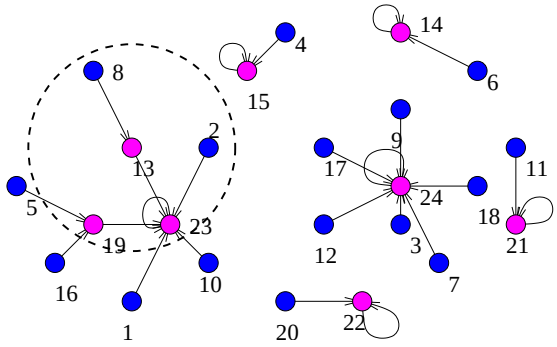


Figure 3.1. The basic algorithm: purple nodes are the nominated clusterheads.

First, we note that randomization is essential for the performance of our scheme. Without randomization, the only approximation bound that holds, even in the one-dimensional case, is the trivial $O(n)$ bound. For example, consider the one-dimensional case in which n points are equally spaced along a unit interval, with their indices increasing monotonically from left to right. Each point in the left half of the set has a different clusterhead, which is the rightmost point within distance $1/2$ of it. Thus the number of clusterheads produced by the algorithm is $n/2$, even though the optimal covering uses only a single clusterhead.

In the following, we are able to show that for *any* configuration, if the ordering is assigned randomly, the basic algorithm yields a sub-linear approximation ($\log n$ in 1-D, and $\sqrt{n}$ in higher dimensions) with high probability.

3.2.2 Analysis for the basic algorithm

Analysis for the one-dimensional case

As a warm-up, we first present the analysis for this algorithm in the 1-D case, where points lie along the real line and the unit square corresponds to the unit interval.

Lemma 3.2.1. *If V' is a subset of the points that are mutually visible to each other, then there is at most one point in V' nominated by points in V'.*

Let the optimal clusterheads be $\{O_i\}$, $i = 1, 2, \ldots, k$. We partition each unit interval U_i centered at O_i into two sub-intervals with O_i as the dividing point. We define the *visible range* of an interval to be all the points on the line that are visible to at least one of the nodes in the interval and call nodes in the visible range the *visible set* for that interval.

Theorem 3.2.2. *The basic algorithm has an approximation ratio of $4 \ln n + 2$ in expectation.*

Proof: It suffices to show that, for each sub-interval S, the number of clusterheads nominated by points in S is at most $2 \ln n + 1$. The visible range of S is an interval of size $\frac{3}{2}$, which contains all the clusterheads nominated by nodes in S, as shown in Figure 3.2. We use S_l to denote the portion of the interval to the left of S and S_r for the right portion. Note that the points in S are mutually visible. Lemma 3.2.1 shows that all the points in S nominate at most one clusterhead in S.

Now we calculate the expected number of clusterheads in S_r that are nominated by points in S. Let $x = |S|$ and $y = |S_r|$ be the number of nodes in the respective subintervals. Scan all points from left to right in S_r. The i^{th} point in S_r can be nominated by a point in S only if it has the largest index compared to all points to its left in $S \cup S_r$. Therefore, the expected number of clusterheads in S_r is no more than $\sum_{i=1}^{y} \frac{1}{x+i} < \ln n$. A similar argument works for S_l, and we can conclude that all points in S nominate at most $2 \ln n + 1$ clusterheads. □

We remark that the approximation bound is asymptotically tight. Consider the following situation in Figure 3.3: the unit interval centered at p is divided into two

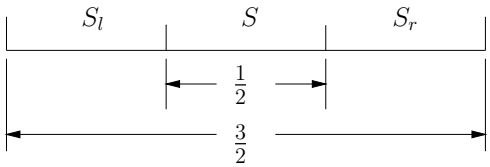


Figure 3.2. S's visible range.

sub-intervals S_l and S_r. S_l contains $\sqrt{n}$ evenly distributed points, each of which can see $\sqrt{n}$ more points in S_r from left to right. In this configuration, with probability $1/2$, the leftmost point q in S_l nominates a point in the first group of $\sqrt{n}$ points in S_r. This is because q sees $2\sqrt{n}$ points ($\sqrt{n}$ in S_l and another $\sqrt{n}$ in S_r). Under a random numbering, the point with the maximum rank falls in S_r with probability $1/2$. In general, a point in the i^{th} group of S_r is nominated by the i-th point in S_l with probability $\frac{1}{i+1}$. Thus the expected number of clusterheads (in S_r alone) is $\sum_{i=1}^{\sqrt{n}} \frac{1}{i+1} = \Omega(\log n)$. But a single clusterhead at p covers all the points.

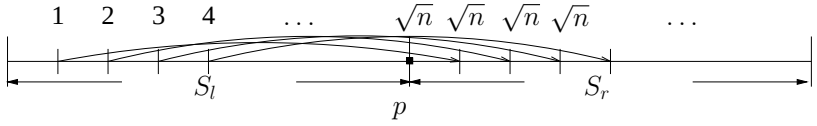


Figure 3.3. Lower bound for the 1-D case.

We can also prove that the $O(\log n)$ upper bound holds with high probability. This fact is useful in our hierarchical algorithm, which achieves a constant approximation factor, and in our kinetic maintenance algorithms.

Theorem 3.2.3. *The probability that the basic algorithm selects more than $ck^* \ln n$ clusterheads is $O(1/n^{\Theta(c^2)})$, where k^* is the optimal number of clusterheads.*

Proof: We divide the optimal intervals in the same way as in the proof of Theorem 3.2.2. Consider a sub-interval S and its right portion S_r. We look for the fraction of random numberings such that points in S nominate not too many clusterheads in S_r. Suppose that $|S \bigcup S_r| = m$. We sort all points in $S \cup S_r$ according to their coordinates from left to right into a sequence of m points. The sequence of their indices can be viewed as a random permutation on numbers $1, 2, \ldots, m$. Each

clusterhead in S_r must have a bigger index than all the other points to its left. Thus, to guarantee that points in S nominate no more than s clusterheads in S_r, it suffices to ensure that the total number of left-to-right maximal indices in the sequence is no more than s. The number of permutations with s left-to-right maxima is known as the Stirling number $C(m, s)$, which is asymptotically equal to $m!\, e^{-\frac{\theta^2}{2}}/\sqrt{2\pi}$, for $s = \ln m + \theta\sqrt{\ln m}$, as $m \to \infty$ and $\theta/m \to 0$ [156]. Let x be the random variable of the number of left-to-right maxima in this permutation and $P(l)$ be the probability that there are $x = l$ left-to-right maxima in a random permutation. Then we have

$$\text{Prob}(x \geq s) = \int_s^\infty P(l)\, dl \leq \int_s^\infty \frac{C(m,l)}{m!}\, dl\,.$$

If we set $s = c \ln n$, this formula becomes

$$\begin{aligned}
\text{Prob}(x \geq c \ln n) &\leq \int_{(c-1)\sqrt{\ln n}}^{\infty} \frac{e^{-\frac{\theta^2}{2}}}{\sqrt{2\pi}} \sqrt{\ln m}\, d\theta \\
&\leq \sqrt{\frac{\ln m}{\pi}} \int_{(c-1)\frac{\sqrt{\ln n}}{\sqrt{2}}}^{\infty} e^{-x^2} dx \\
&\leq \frac{n^{-\frac{(c-1)^2}{2}}}{\sqrt{2\pi}(c-1)} \leq n^{-\Theta(c^2)}\,.
\end{aligned}$$

For $O(k^*)$ sub-intervals, since each needs to be considered only twice for its left and right points, the probability that there are more than $ck^* \ln n$ clusterheads is less than $\Theta(n)\, n^{-\Theta(c^2)}$, which is $O(n^{-\Theta(c^2)})$. □

Analysis for the two-dimensional case

Unfortunately Theorem 3.2.3 does not extend to higher dimensions. We will show that in two (and higher) dimensions, the basic algorithm produces a $\Theta(\sqrt{n} \log n)$ approximate cover with high probability. The analysis is similar to the 1-D case. Again, we consider the sub-squares with side length 1/2. Suppose that L is the visible range of a sub-square S. Clearly, L is a square of side length 3/2 and can be partitioned into 9 sub-squares where S is the center one (Figure 3.4). Now, we have

the following lemma:

Lemma 3.2.4. *Suppose that $|L| \leq m$. Then the number of clusterheads nominated inside S is $O(\sqrt{m})$ in expectation. Furthermore, the probability that S contains more than $8\sqrt{m}\ln m + 1$ clusterheads is bounded by $O(1/m^{\ln m})$.*

Proof: We only need to consider the points inside L. It suffices to bound the number of clusterheads nominated by points in each sub-square S' of L. If $S' = S$, since all the points are mutually visible in S, there can be at most one point nominated.

Now we consider a sub-square $S' \neq S$. Suppose that $x = |S|$, $y = |S'|$, $x + y = m$. A point $p \in S$ can be nominated by a point $q \in S'$ if q finds that p has the largest index in its visible range. Since q sees all points in S', p must have rank higher than all the points in S'. Thus, the probability that p can be nominated is at most $\frac{1}{1+y}$. Thus, in expectation, there are at most $\frac{x}{1+y}$ points nominated. On the other hand, since there are only y points in S', there can be at most y clusterheads nominated by points in S'. The expected total number of clusterheads is therefore no more than $\min(y, \frac{x}{1+y}) \leq \sqrt{x+y+1} - 1 < \sqrt{m}$.

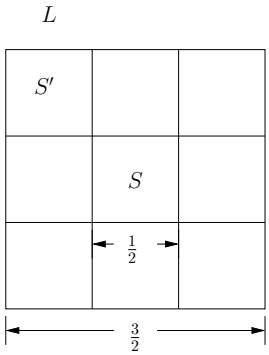


Figure 3.4. S's visible range L.

Next we prove the high probability bound. If $y < \sqrt{m}\ln m$, then we know that S' cannot nominate more than $\sqrt{m}\ln m$ points. Otherwise, S' contains $y > \sqrt{m}\ln m$ points. In order to nominate s points in S, S must contain at least s points with higher ranks than all the points in S'. That is, S must contain the s highest ranked

points in $S \cup S'$. The probability for this to happen is:

$$\binom{x}{s} \Big/ \binom{x+y}{s} = \frac{x!(x+y-s)!}{(x+y)!(x-s)!}$$
$$< \left(\frac{x}{x+y}\right)^s = \left(1 - \frac{y}{x+y}\right)^s < \left(1 - \frac{y}{m}\right)^s < \left(1 - \frac{\sqrt{m}\ln m}{m}\right)^s .$$

Thus, if $s > \sqrt{m}\ln m$, we have

$$\text{Prob}(x \geq s) < \left(1 - \frac{\ln m}{\sqrt{m}}\right)^{\sqrt{m}\ln m} < \left(\frac{1}{e}\right)^{\ln^2 m} = O(\frac{1}{m^{\ln m}}) .$$

Summing over all the 9 sub-squares, we see that the expected number of clusterheads nominated in S is bounded by $O(\sqrt{m})$, and with high probability, the number of clusterheads nominated is bounded by $O(\sqrt{m}\ln m)$. □

By Lemma 3.2.4, it is easy to obtain

Theorem 3.2.5. *For points in the plane, the algorithm has an approximation factor of $O(\sqrt{n})$ in expectation. Further, the probability that there are more than $\sqrt{n}\ln n \cdot k$ clusterheads is $O(1/n^{\ln n - 1})$, where k is the optimal number of clusterheads.*

Proof: Consider an optimal covering U_i, $1 \leq i \leq k$. We partition each U_i in the optimal solution into 4 quadrant sub-squares and apply Lemma 3.2.4 to each sub-square. Since there are at most $O(n)$ sub-squares, the high probability result also holds. □

Again, this bound is asymptotically tight. Consider the configuration in Figure 3.5. The upper left sub-square S_1 has $\sqrt{n}$ points, each of which can see a distinct set of $\sqrt{n}$ points in the lower right sub-square S_2. Each point in S_1 is visible to $2\sqrt{n}$ points: $\sqrt{n}$ points in S_1 and $\sqrt{n}$ points in S_2. So it will nominate a point in S_2 with probability 1/2. Thus the expected number of clusterheads in S_2 is $\Omega(\sqrt{n})$.

We remark that in this analysis, the use of the unit square and the dimensionality is not essential. It is easy to extend the analysis to any centrally symmetric covering shape in any dimension; the constant factors, however, depend on the covering shape

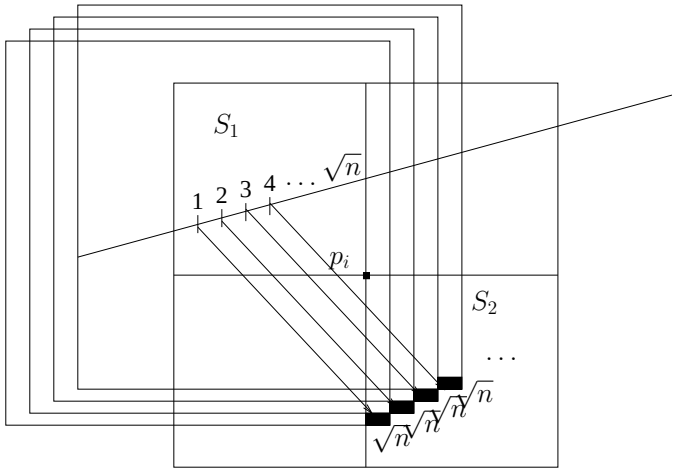


Figure 3.5. Lower bound for the 2-D case.

and the dimensionality. Note also that the worst-case examples in Theorems 3.2.2 and 3.2.5 require a significantly non-uniform distribution of the points. The distributions encountered in practice are much less skewed, and the basic algorithm returns much better results, as experiments show [60].

3.3 Hierarchical algorithms for clustering

The basic algorithm is simple, but it achieves only an $O(\sqrt{n})$ approximation for points in the plane. To obtain a constant-factor approximation, we will use a hierarchical algorithm that proceeds in a number of rounds. At each round we apply the basic algorithm to the *clusterheads* produced by the previous round, using a larger covering range. Suppose that $\delta_i = 2^i / \log n$, for $i > 0$. Initially, set P_0 to be P, the input set of points. At the i^{th} step, for $1 \leq i < \log\log n$, we apply the basic algorithm using squares with side length δ_i to the set P_{i-1} and let P_i be the output. The final output of the algorithm is $P' = P_{\log\log n - 1}$ [2]. We claim that:

Lemma 3.3.1. *P' is a 1-cover of P.*

[2]To make our analysis fully rigorous, we would need to use $\lfloor \log n \rfloor$ and $\lfloor \log\log n \rfloor$ instead of $\log n$ and $\log\log n$; however, in the interest of readability, we will omit the floor functions from this chapter.

Proof: We actually prove a stronger statement: P_i is a $\frac{2^{i+1}}{\log n}$-cover of P. We proceed by induction. The assertion is clearly true when $i = 0$. Suppose that it is true for i, i.e., every point $p \in P$ can be covered by a size $2^{i+1}/\log n$ square centered at a point $q \in P_i$. If q is also in P_{i+1}, then p is covered. Otherwise, there must be a q' so that q nominates q' at the $(i+1)^{\text{th}}$ step. Thus, p is covered by q' with a square with side length $2^{i+1}/\log n + \delta_{i+1} = 2^{i+2}/\log n$. That is, P_{i+1} is a $(2^{i+2}/\log n)$-cover of P. □

In the following, we bound the approximation factor for P'. To explain the intuition, we first consider the situation when P admits a single clusterhead, i.e., there is a unit square that covers all the points in P. Recall that $\alpha(x)$ denotes the number of clusterheads of an optimal covering of P by using squares with side length x. First, we observe that

Lemma 3.3.2. $\alpha(x) \leq 4/x^2$.

Proof: We uniformly divide the unit square into $4/x^2$ small squares of side length $x/2$. We then pick one point from each non-empty small square, which gives an x-cover with at most $4/x^2$ clusterheads. □

According to Theorem 3.2.5, the expected size of P_{i+1} is at most $c\sqrt{|P_i|}\alpha(\delta_{i+1})$, for some constant $c > 0$. Denote by n_i the size of P_i. We have the following recursive relation:

$$n_0 = n\,, \quad n_{i+1} \leq c\sqrt{n_i}\alpha(\delta_{i+1}) \leq c\sqrt{n_i}\frac{4\log^2 n}{2^{2i+2}}\,.$$

By induction, it is easy to verify that

$$n_i \leq \frac{(c^2\log^4 n)n^{\frac{1}{2^i}}}{4^{2i-4}}\,.$$

Thus $|P'| = n_{\log\log n - 1} \leq c^2 2^{14} = O(1)$.

We cannot apply this argument directly to the general case because $\alpha(x)$ can be as large as $\Theta(n)$. In order to establish a similar recursive relation, we consider points restricted to lie in squares of a certain size. For any square S with side length δ_i, let $m_i(S)$ denote the expected value of $|P_i \cap S|$. Further, let m_i denote the maximum of

$m_i(S)$ over all the squares S with size δ_i. We then have the following relation between m_i's.

Lemma 3.3.3. *$m_{i+1} \leq c\sqrt{m_i}$, for some constant $c > 0$ and any $0 \leq i < \log\log n - 1$.*

Proof: Consider a square S of side length δ_{i+1}. Its visible region L, with respect to side length δ_{i+1}, is a square with side length $2\delta_{i+1} = 4\delta_i$. Thus L can be covered by $4^2 = 16$ squares with side length δ_i. That is, $|P_i \cap L| \leq 16m_i$ in expectation. By Lemma 3.2.4, we know that the expected number of points inside S that survive after the $(i+1)^{\text{th}}$ step of the algorithm is $O(\sqrt{|P_i \cap L|}) = O(\sqrt{m_i})$. Thus, we have $m_{i+1} \leq c\sqrt{m_i}$, for some constant $c > 0$. □

Now, we can prove that

Theorem 3.3.4. *P' is a constant approximation to the optimal covering of P with unit squares in expectation.*

Proof: Clearly $m_0 \leq n$. Solving the recursive relation in Lemma 3.3.3, we find that $m_i \leq O(c^2 n^{1/2^i})$. Setting $i = \log\log n - 1$, we have $m_{\log\log n - 1} = O(1)$, i.e., for a square S with side length $1/2$, the expected number of points of P' inside S is $O(1)$.

Now, suppose that an optimal cover uses k unit squares. We can then cover all the points by $4k$ squares with side length $1/2$. Since each of these squares contains $O(1)$ points in P' in expectation, the total number of points in P' is bounded by $O(k)$. □

In addition, we have:

Corollary 3.3.5. *For a modified version of the hierarchical algorithm, i.e., we stop the clusterhead election process as soon as m_i drops below $\log n$, then the number of clusterheads generated is an $O(\log^3 n)$ approximation to the optimal cover of P, with probability $1 - o(1)$.*

Proof: From Lemma 3.2.4, at round i, $m_{i+1} \leq 8\sqrt{m_i}\ln m_i + 1$, with probability $1 - O(1/m_i^{\ln m_i})$. So $m_{i+1} \leq c' m_i^{1/(2-\delta)}$ for some constant c' and $0 < \delta < 1$. In this

corollary, we change the base of the log function from 2 to $2-\delta$, so $m_i \leq c'^{\frac{2-\delta}{1-\delta}} n^{\frac{1}{(2-\delta)^i}}$. To obtain a $O(\log^3 n)$ approximation, we could stop the clusterhead election process as soon as m_i drops below $\log n$. For a square of side length 1, the total number of clusterheads inside is $O(\log^3 n)$, because the size of squares at level i is at least $1/\log n$. We achieve this bound with probability bigger than $(1-O(1/(\log n)^{\ln \log n}))^{\log \log n - 1} \geq 1 - o(1)$. □

3.4 Kinetic discrete clustering

To kinetize the algorithm, we place a half-size square centered over each point. If two such squares intersect, we know the corresponding points are mutually visible. In this section when we say "squares", we refer to these half-size squares.

3.4.1 Standard KDS implementation

The intersection relation between two squares can change only at discrete times. If two squares of the same size intersect with each other, one square must have a corner inside the other square. Therefore, we can maintain the left and right extrema of squares in x-sorted order and the top and bottom extrema of squares in y-sorted order. The certificates of the KDS are the ordering certificates for the x- and y-sorted lists of square extrema. We maintain the lists containing the extrema of active squares for each level of the hierarchy. An event is a certificate failure. When an event happens, we first check whether it is a "real" event, i.e., whether it causes two squares to start/stop intersecting. When two squares S_1, S_2 start intersecting, we will need to check the square with the lower rank, say S_1, to see if its nomination has a lower rank than S_2. If so, we need to let S_1 nominate S_2. If S_1, S_2 stop intersecting, we need to check if S_1 nominated S_2. If so, we need to find another overlapping square of S_1 with the highest rank. To answer this query efficiently, we maintain a standard range search tree [128] for the n points. For our purpose, the internal nodes of the second-level binary trees in the range tree are augmented with the maximum index of the points stored at the descendants of each node. This will let us find

the points within a query square that are larger than some query index in $O(\log^2 n)$ time. To maintain the range search trees kinetically, we keep sorted lists of the x- and y-coordinates of the points themselves, in addition to the sorted lists containing the extrema of the squares on each level. A range tree can be updated by deleting a point and re-inserting it in the right place [30].

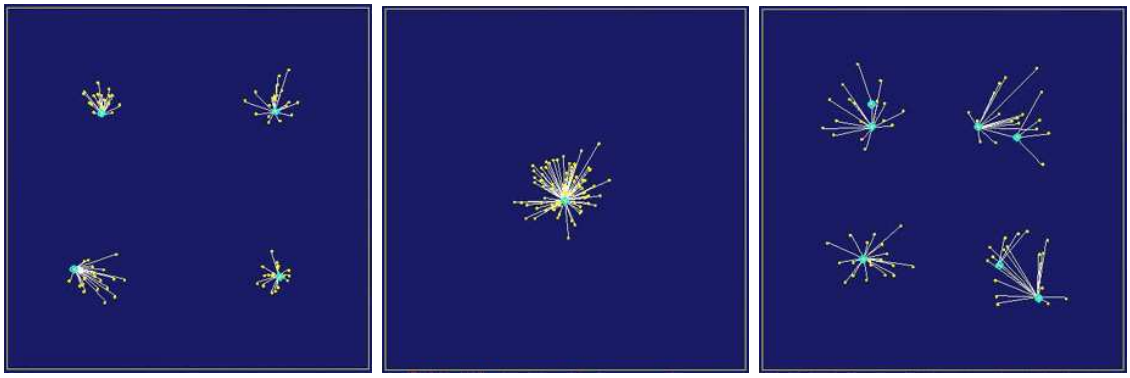


Figure 3.6. The clusterheads evolve along with motion.

For the hierarchical algorithm, we need to maintain these structures for each level. In addition, we also need to insert or delete a point to or from a level, as a consequence of an event happening at a lower level. This requires the sorted lists and range search trees used in the basic algorithm above to be dynamic. These requirements can easily be satisfied by maintaining balanced binary search trees and dynamic range search trees.

3.4.2 Kinetic properties

This kinetic data structure has most of the properties of a good KDS [28]. We assume the points have bounded-degree algebraic motion in the following arguments.

To analyze the efficiency, i.e., the number of events, of our algorithms, we first give some lower bound constructions. It turns out that the optimal cover is too rigid and has to change many times in the worst case.

Lemma 3.4.1. *The number of changes of the optimal cover for n points in motion is $\Theta(n^3)$ in the worst case.*

Proof: Consider the graph G in which each vertex represents a point and each edge joins a visible pair of points. Clearly, the minimum discrete covering of the points is exactly the same as the minimum dominating set of the graph. The graph can change only when two points become or cease to be visible to each other. For bounded degree algebraic motions, this can happen only $O(n^2)$ times. For each such event, the change to the minimum covering is at most $O(n)$. Thus, in the worst case, the number of changes is $O(n^3)$.

We now construct an example in which any optimal cover must change $\Theta(n^3)$ times. The construction uses $6m + 6$ static points along the perimeter of a rectangle $[0, R] \times [0, 1.6]$, where $R = 0.4 \times (3m + 1)$. The left and right sides of the rectangle have three points apiece, located at $(0, 0.4i)$ and $(R, 0.4i)$ for $i = 1, 2, 3$. The top and bottom sides of the rectangle have $3m$ points apiece, located at $(0.4i, 0)$ and $(0.4i, 1.6)$, for $i = 1, \ldots, 3m$. We label the points counterclockwise from 0 to $6m+5$ as shown in Figure 3.7. In this configuration, each point i can see the points $i - 1, i + 1$ (modulo $6m+6$) and no other points. Thus, an optimal cover contains $2m+2$ clusterheads and can be realized in one of three ways by using points $3i$, $3i + 1$, or $3i + 2$, respectively, which we call types $0, 1$, and 2, respectively. To change from one type to another, we need to make $\Theta(m)$ changes to the cover.

Now consider what happens when a single point p moves linearly along the x-axis. For any i, suppose that q_j is the middle point between the pair $3i + j, 3i + j + 1$, for $0 \leq j \leq 2$. When p is located at q_j, the only points p can see are $3i+j$ and $3i+j+1$. Thus, an optimal cover has to use either $3i + j$ or $3i + j + 1$ as a clusterhead. In other words, an optimal cover has to be of type j or $j + 1$. It is easily verified that when p moves from q_0 to q_2, an optimal cover has to change its type. Therefore, an optimal cover undergoes $\Theta(m)$ changes when p moves from q_0 to q_2. When p moves from $(0, 0)$ to $(R, 0)$, the number of changes is $\Theta(m^2)$. We repeat this procedure by sending m points along the x-axis, passing through the interval $[0, R]$ one at a time. This causes a total of $\Theta(m^3)$ changes to optimal covers. The total number of points is $n = 7m + 6$, so the total number of clusterhead changes is $\Theta(n^3)$. □

While the optimal cover in this construction changes $\Omega(n^3)$ times, a 2-approximate cover does not change at all—we can simply use an optimal cover for the static points

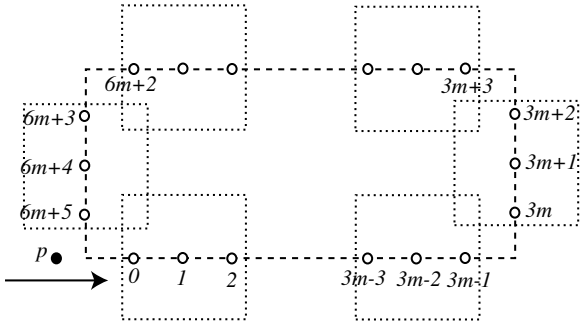


Figure 3.7. Lower bound for optimal coverings

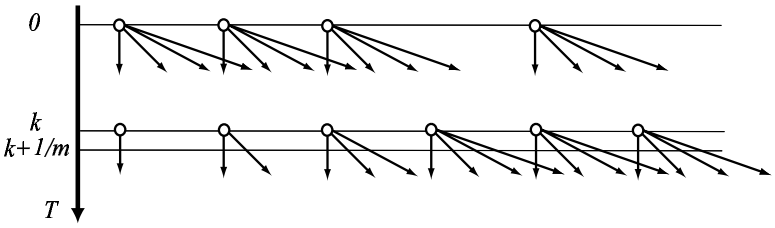


Figure 3.8. Lower bound approximate coverings

and assign each moving point to be a clusterhead. However, in the following, we will show that for any constant c, there is a set of moving points that forces any c-approximate cover to change $\Omega(n^2/c^2)$ times.

Theorem 3.4.2. *For any constant $c > 1$, there exists a configuration of n points moving linearly on the real line so that any c-approximate cover undergoes $\Omega(n^2/c^2)$ changes.*

Proof: In the following, we assume that c is an integer and $n = 2cm$, where $m > 2c$ is an integer. We group n points into m groups, each containing $2c$ points. We label each point by (i, j) where $0 \leq i < m$ is the group number, and $0 \leq j < 2c$ is the numbering within each group. Initially, all the points in the i^{th} group are located at $i \cdot 2m$, and the speed of the point (i, j) is $j \cdot 2m$. To summarize, we consider points $p(i, j, t)$ defined as $p(i, j, t) = (i + jt) \cdot 2m$, for $0 \leq i < m$, $0 \leq j < 2c$, and $t \geq 0$.

Whenever $t = k + 1/m$, for some integer $k < m$, $p(i, j, t) = (i + jk + j/m) \cdot 2m = 2(i + jk)m + 2j$. For any two distinct points (i, j) and (i', j'), if $i + jk \neq i' + j'k$, then $|p(i, j, t) - p(i', j', t)| > 2m - 4c \geq 2$; if $i + jk = i' + j'k$, since (i, j) and (i', j') are

distinct, $j' \neq j$ and $|p(i,j,t) - p(i',j',t)| \geq 2$. Thus, at time t, no two points are within distance 1. In other words, any covering has to have n clusterheads (Figure 3.8).

On the other hand, at time $t = k$ for an integer $k < m$, since $p(i,j,k) = (i+jk)\cdot 2m$ where $0 \leq i < m$, $0 \leq j < 2c$, and $k < m$, each point has position $2sm$ for some $0 \leq s < m + 2ck$. That is, at $t = k$, the minimum covering has at most $m + 2ck$ clusterheads (Figure 3.8). Thus, a c-approximate cover may have at most $c(m + 2ck)$ clusterheads. Therefore, between times k and $k + 1/m$, there are at least $n - c(m + 2ck) = n/2 - 2c^2k$ changes to any c-approximate covering. In total, for all $0 \leq t < K$, the number of changes is at least $\sum_{0 \leq k < K}(n/2 - 2c^2k) > Kn/2 - c^2K^2$. Setting $K = \frac{n}{4c^2} < m$, we have established that the total number of changes is $\Omega(n^2/c^2)$. □

Lemma 3.4.3. *The number of events in our basic algorithm is $O(n^2)$.*

Proof: An event is the failure of an ordering certificate in an x- or y-sorted list of square side coordinates or point coordinates. Since the points have bounded-degree algebraic motion, each pair of points can cause $O(1)$ certificate failures. □

Theorem 3.4.4. *The number of events processed by our hierarchical KDS is at most $O(n^2 \log\log n)$, and hence the KDS is efficient.*

Proof: We maintain x- and y-ordering certificates on each of the $\log\log n$ levels. As in Lemma 3.4.3, each pair of points can cause $O(1)$ certificate failures on each level. In addition, in the hierarchical KDS, we need to consider the events for maintaining the range search tree. Those events can happen when two points swap their x- or y-ordering. Such an exchange requires possible updates of the range trees on all levels where the exchanging pair is present. Again, there are $O(n^2)$ exchange events at each level. □

We now proceed to examine the cost of processing the kinetic events.

Theorem 3.4.5. *The expected update cost for one event is $O(\log^{3.6} n)$. Hence the KDS is responsive in an expected sense.*

Proof: When two points exchange in x- or y-order, only the relevant range search trees need to be updated. We need $O(\log^2 n)$ time to update each of the $\log \log n$ range trees.

When two points p_i, p_j start/stop being mutually visible at any level of the hierarchy, we can update the clusterheads involved with p_i, p_j in $O(\log^2 n)$ time, since we may need to search for a replacement clusterhead in the range tree. One new clusterhead may appear and one old clusterhead may disappear; these changes bubble up the hierarchy.

On hierarchy levels above the bottom, we divide the changes into two kinds, those caused by the motion of the points in that level and those caused by insertion or deletion of points bubbled up from lower levels. The number of changes of the first kind per event is a constant.

Let us consider the insertion of point p. The only points that may change their clusterheads are those in p's visible range S. We divide S into four quadrants S_i, each with k_i $(i = 1, 2, 3, 4)$ points. If there is some point in S_i that nominates p to be its clusterhead, the index of p must be bigger than the indices of all the k_i points. The probability of this occurring is $\frac{1}{k_i+1}$. Therefore, the expected number of client-clusterhead changes caused by the appearance of p is at most

$$\frac{k_1}{k_1+1} + \frac{k_2}{k_2+1} + \frac{k_3}{k_3+1} + \frac{k_4}{k_4+1} + 1 \leq 5.$$

Assuming that p becomes a clusterhead, how many clusterheads does it replace? For a given quadrant S_i, suppose the number of clusterheads its points nominate is $m_i \leq k_i$. At most one of these clusterheads is inside S_i. If m' points are outside S_i, the probability that p replaces j of them is at most $1/(m'+1)$. Hence the expected number of clusterheads replaced in a single quadrant is upper bounded by either

$$\frac{1}{k_i+1}\left(1 + \frac{1 + \cdots + m_i - 1}{m_i}\right) = \frac{m_i+1}{2(k_i+1)} \leq \frac{1}{2},$$

if one of the clusterheads is inside S_i, or by

$$\frac{1}{k_i+1}\left(\frac{1+\cdots+m_i}{m_i+1}\right) = \frac{m_i}{2(k_i+1)} \leq \frac{1}{2}$$

if none of the clusterheads is inside S_i. Each replaced clusterhead may stop being a clusterhead at this level of the hierarchy, if it is nominated by no points outside S. Thus the expected number of clusterheads created/destroyed in this level (inserted/deleted at higher levels) due to the appearance of p is at most $4 \times \frac{1}{2} + 1 = 3$.

We can make a similar argument for the disappearance of a point. So the expected total number of point-clusterhead changes at all levels of the hierarchy per event is at most

$$5 \times (3^{\log\log n} + 3^{\log\log n - 1} + \cdots + 1)$$

which is $O(3^{\log\log n}) = O(\log^{1.6} n)$.

Since insertion or deletion in a range search tree costs $O(\log^2 n)$, the total expected update cost is $O(\log^{3.6} n)$. □

Theorem 3.4.6. *The kinetic data structure uses $O(n \log n \log\log n)$ storage. Each point participates in at most $O(\log\log n)$ ordering certificates. Therefore, the KDS is compact and local.*

Proof: Range trees take $O(n \log n)$ space per level. All other data structures use less space. Each point participates in at most $O(1)$ ordering certificates in each level. □

3.4.3 Distributed implementation

The KDS implementation requires a central node to collect all the information and perform the hierarchical clustering algorithm. For wireless mobile *ad hoc* networks [153], all the wireless nodes are homogeneous and a central node is not available most of the time. Furthermore, the cost of communicating all the data to such a node can be prohibitive. Our hierarchical algorithm can be implemented in a distributed manner, making it appropriate for mobile networking scenarios.

To implement the hierarchical clustering algorithm, we first describe how to obtain range information about internode distances. Each node broadcasts a "who is there"

message and waits for replies. Each node that hears the request responds. The hierarchy can be implemented by having nodes broadcast with different power for each level or by other local positioning mechanisms. When the nodes move around, each node broadcasts these "Hello" beacons periodically, with a time interval dependent on its moving speed. Therefore, each point keeps track of its neighborhood within different size ranges. When a neighbor enters or leaves any of the $\log \log n$ ranges of a node, each node involved checks whether it needs to update its clusterhead. When it nominates a clusterhead that is not nominated by any other node, the clusterhead will also be added to a higher level and may cause updates in that level. If a node ceases to be pointed to by any node, then it also has to be deleted from higher levels in the hierarchy. Clearly, all of these operations can be done locally without centralized control.

Notice that by the power-attenuation model, the energy consumed at each node is kept low, since each node transmits only within a small neighborhood. We emphasize here that range information (inter-node distances) is sufficient for each node to select a clusterhead for each level. This information can often be obtained using the node radios themselves or acoustic range-finding ultrasonic devices [157]. No global positioning information is needed to implement our clustering algorithm.

This contrasts with many algorithms proposed for mobile *ad hoc* networks [87, 82, 107], which require that the exact location of each wireless node be known. Obtaining global position information is expensive—a GPS receiver per node is a costly addition. In addition, GPS does not work for indoor situations. Multiple localization methods have been proposed by using only a modest number of nodes with global coordination obtained by GPS or explicit configuration [136]. But obtaining accurate global position information still remains an open problem in ad hoc networking area. Furthermore, the algorithms quickly breaks down when most nodes are moving, as the convergence of the methods is slow.

In the distributed implementation described above, the total storage needed is $O(sn)$, where s is the maximum number of nodes inside a node's range. In the worst case, this can be $\Theta(n^2)$, but in practice, s is often small. Furthermore, we can restrict the storage of each node to be n^ε, where $0 < \varepsilon < 1$, and still get a

constant approximation. If we let each node keep up to $O(n^\varepsilon)$ neighbors and select a clusterhead among them, then we have the following:

Lemma 3.4.7. *In 2-dimensional space, the number of clusterheads nominated inside a unit-size square S by the space-restricted one-level algorithm is $O(n^{\max(1-\varepsilon,1/2)})$ in expectation.*

Proof: We use the same notation as Lemma 3.2.4. Suppose L is the visible range of S. L can be divided into 9 sub-squares of size 1/2. Consider any such sub-square S', and suppose $m = |L|$, $x = |S|$, $y = |S'|$. For points in S' such that the number of neighbors is $\leq n^\varepsilon$, from Lemma 3.2.4, the expected number of clusterheads in S that are nominated by those nodes in S' is bounded by $\sqrt{m} \leq \sqrt{n}$. The rest of the points in S' store only n^ε neighbors. A point p in S can be nominated by those points only if p has the largest index among n^ε points. So the probability is at most $1/n^\varepsilon$. The expected number of clusterheads in S nominated by such points in S' is no more than $x/n^\varepsilon \leq n^{1-\varepsilon}$. So the total number of clusterheads in expectation is $O(n^{\max(1-\varepsilon,1/2)})$. □

Theorem 3.4.8. *The expected number of clusterheads generated by the space-restricted hierarchical algorithm is a constant-factor approximation to the optimum.*

Proof: Recall the notations in Section 3.3. From the previous lemma we established the recurrence $m'_{i+1} \leq cm_i^{c_\varepsilon}$, where c_ε denote $\max(1-\varepsilon, 1/2)$. So $m_i \leq c^{\frac{1}{1-c_\varepsilon}} n^{c_\varepsilon^i}$. We simply change the base of the log function from 2 to $1/c_\varepsilon$ in the proof of Theorem 3.3.4. After $\log\log n - 1$ rounds, we again obtain a constant approximation. □

3.4.4 Extensions

The randomized hierarchical algorithm can be extended to the case in which the ranges are any congruent convex shape and to higher dimensions, specifically, the ranges are disks in 2-D or balls in 3-D. Most of the analysis for the 2-D case works for any dimension d, except that the constant approximation factor depends exponentially on d. Our algorithms also support efficient insertion or deletion of nodes.

However, the KDS maintenance of the hierarchical algorithm exploits the fact that the ranges are aligned congruent squares. If the ranges are congruent disks, the KDS in this chapter doesn't work anymore. The central problem is to maintain the nodes inside each one's communication range. We will describe in Chapter 8 an efficient KDS that maintains the proximity information, and therefore can be used to maintain the discrete centers of a set of points with congruent disk ranges.

Chapter 4

Geometric Spanners for Routing

4.1 Introduction

Geographical routing is considered as a scalable routing scheme when the geographical locations of the wireless nodes are available. However, geographic forwarding methods suffer from the so called *local minimum phenomenon*, in which a packet gets stuck at a node that does not have a closer neighbor to the destination, even though the source and destination are connected in the network. Face routing (also called perimeter routing) has been used to help the packets get out of the local minima [36, 87]: the packet is routed around a face of a planar graph until either the destination is reached or we can do greedy forwarding again. Two planar subgraphs, the *relative neighborhood graph* (RNG) and the *Gabriel graph* (GG), have been used in the face routing process. They are both based on local geometric conditions and can be computed efficiently.

However, one drawback of the GG and RNG is that they are not good spanners: nodes that can be reached via a path with few hops might become far apart in the GG or RNG [52]. We use the *stretch factor* to capture this aspect of path quality. Roughly speaking, the stretch factor of a subgraph G' of a graph G measures the worst-case ratio between the length of a shortest path in G' to the length of the shortest path with the same endpoints in G. Both GG and RNG have unbounded stretch factor in the worst case. This fact limits the quality of paths even if we use

globally optimum routing methods on these subgraphs. In this chapter we introduce a new spanner graph, called the *restricted Delaunay graph* (RDG) that has nice theoretical guarantees on the quality of the routing paths. In particular, the RDG has paths with Euclidean or topological length only a constant factor longer than the length of the optimal path in the original communication graph. Our routing graph can be maintained efficiently in a distributed fashion under node motion. In addition to presenting rigorous theoretical analysis, we also demonstrate that experimentally, GPSR on our routing graph improves the routing path quality compared to the path quality using other graphs, such as the GG or RNG used in GPSR, under both uniform and multi-modal distributions of the points.

4.1.1 Overview

The construction of our routing graph consists of two phases. For a set of points S in the plane, we first make use of a clustering algorithm, for example the one in Chapter 3, to select a small subset of S, called *clusterheads*, so that each node in S can communicate directly to some clusterheads. Each non-clusterhead node in S (called a *client*) is assigned to a unique clusterhead visible to it. We also identify those pairs of clusterheads that may communicate to each other via their clients. For each such pair, we pick one pair of clients, called *gateways*, that enable such communication. In the second phase, we form a planar routing graph on the clusterheads and gateways by applying a local rule, called the *restricted Delaunay edge* rule. The graph produced this way is called the *restricted Delaunay graph* (RDG). Routing between clusterheads and gateways is then done on the RDG.

For a node u to send a packet to a non-neighbor node v, u first forwards the packet to its clusterhead, and the packet is then forwarded in the restricted Delaunay graph until it reaches some clusterhead or gateway that is visible to v. Therefore, our final routing graph R is the union of RDG and the edges that connect clients to clusterheads. Our routing graph has the following properties:

- RDG is a planar graph (no two edges cross each other in the graph).
- R has constant stretch factor, in both topological and Euclidean senses. That

is, if there exists a path in $\mathcal{G}$ with length ℓ between two nodes, then there is a path in R with length $C \times \ell$ for some constant $C > 0$, where the length can be either topological or Euclidean.

- R can be efficiently computed and maintained in the distributed setting, when the wireless nodes are inserted/deleted or move around.

We use a clustering algorithm to guarantee that each clusterhead/gateway has only a constant number of neighbors [58]. This simplifies forwarding during routing. In [87], the greedy geographic forwarding is done by examining all the neighboring nodes in order to skip short edges in the graph. This process is expensive when the nodes are densely distributed. In our routing graph, since we cluster nodes in the first stage, we can perform the greedy geographic forwarding by considering only the adjacent nodes in the *routing graph*, and this reduces the complexity significantly. The clustering algorithm also improves the behavior of GPSR, since we have only essential points in the graph. In the GG or RNG, GPSR may traverse a short boundary that consists of a dense sequence of points; but boundaries in the RDG have only constant density. We also investigate the trade-off between scaling and the spanning property, and the efficiency of clusterhead changes.

The rest of the chapter is organized as follows: Section 4.2 gives a detailed description of the RDG and proves the spanning property, Section 4.3 deals with the distributed implementation of the RDG, Section 4.4 proves theoretical bounds on the length of the actual routing paths under certain circumstances, Section 4.5 compares the simulation results of GPSR on the RDG vs. GPSR on the RNG and discusses various aspects of the RDG.

4.2 Routing graph with constant stretch factor

In this section, we will explain the two phases in constructing the routing graph: the clustering and the restricted Delaunay graph.

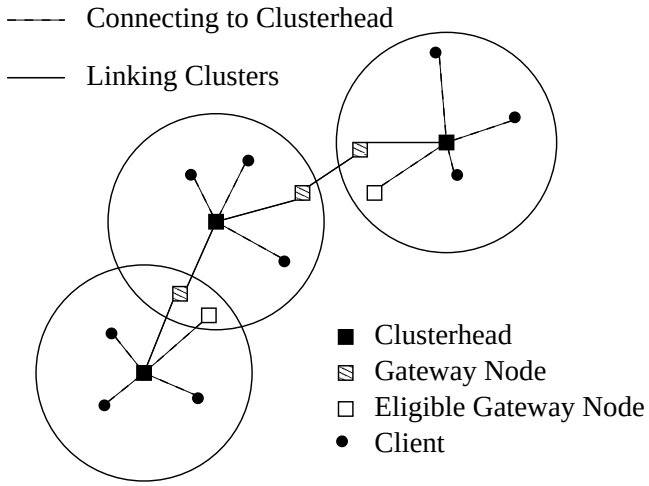


Figure 4.1. Example of linked cluster organization of a mobile network.

4.2.1 Clusterheads and gateways

The goal of clustering is to select a subset of nodes as the *clusterheads* such that the rest of the nodes are visible to at least one of the clusterheads. While any clustering algorithm can be used in the first stage, the algorithm developed in Chapter 3 is used because we need some special properties of the clustering algorithm to achieve good properties on the routing graph. As shown in Chapter 3, the set of clusterheads is a constant approximation of the optimal solution. Since there are at most 6 centers inside any unit disk in the optimal solution due to a packing argument, we have,

Corollary 4.2.1. *The number of clusterheads in any unit disk is $O(1)$ in expectation.*

If we define the density of a set of nodes to be the maximum number of nodes inside any unit disk, then the clusterheads have constant density in expectation.

To enable different clusters to communicate with each other, we introduce *gateways* [51], to link nearby clusters. For each clusterhead p, define the cluster $C(p)$ centered at node p to be the set of nodes that nominated p and p itself [1]. A node x's clusterhead is denoted by c_x. For a pair of clusterheads (c_1, c_2), if there exists a pair of nodes $p_1 \in C(c_1)$, $p_2 \in C(c_2)$ such that p_1 and p_2 are visible to each other,

[1]Note that one node can participate in two clusters, if it nominates another node as its clusterhead, and at the same time it is nominated by others to be a clusterhead.

we define p_1 and p_2 to be *gateway* nodes. Note that p_1 and p_2 might be clusterheads already, in which case they remain clusterheads. Between each pair of overlapping or adjacent clusters, only one pair of gateway nodes is maintained at any time. From Theorem 4.2.1, there are at most a constant number of clusterheads within 3 hops of any given clusterhead, therefore we can derive the following fact.

Corollary 4.2.2. *The number of clusterheads and gateways in any unit disk in the plane is $O(1)$ in expectation.*

The hierarchical algorithm provides a theoretical bound that holds for *any* distribution of points in space. In reality, it's very rare to have distributions that cause bad clustering quality. In our simulation, we only use the one-level clustering algorithm described above.

4.2.2 Restricted Delaunay graph

The clusterheads and gateways provide a summarization of the wireless network without losing connectivity. The routing of a packet from u to v (if v is not directly reachable) is realized by first sending the packet to u's clusterhead, then forwarding the packet among clusterheads and gateways until it reaches a node visible to v, which forwards the packet to v. In this subsection, we propose a planar graph on the clusterheads and gateways as a routing backbone. The routing graph R includes the backbone and the edges that connect each client to its clusterhead.

We design a *restricted Delaunay graph* (RDG) for connecting clusterheads and gateways, and use it to replace the GG or RNG in the perimeter routing [87]. The advantage of RDG is that it provides theoretical guarantees on the Euclidean and topological stretch factors[2], while the GG and RNG do not.

Restricted Delaunay graph

The Delaunay triangulation is known to be a good spanner of the complete graph [49, 88]. However, the Delaunay triangulation may have very long edges, while in the

[2]The stretch factor of RDG is with respect to the unit disk graph on the clusterheads and gateways.

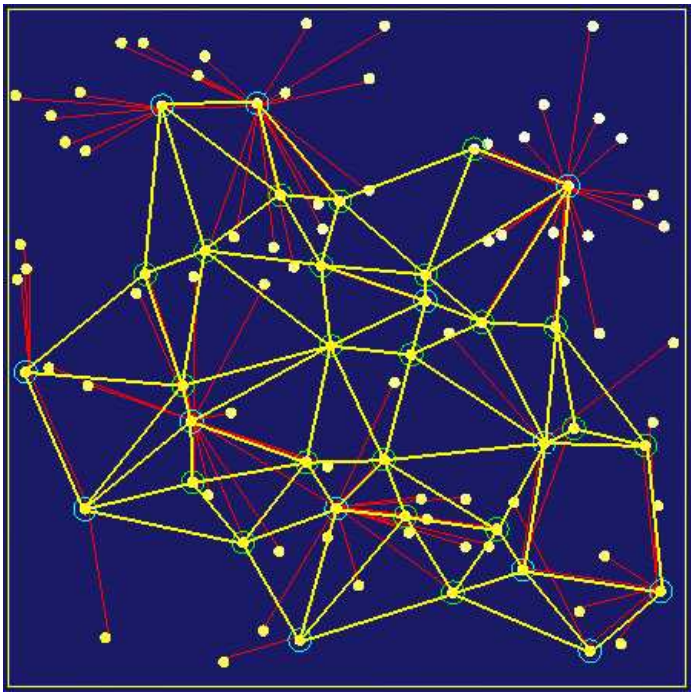


Figure 4.2. The edges in the restricted delaunay graph are drawn in yellow. Red edges connect nodes to their clusterheads. Nodes with blue annuli are clusterheads. Nodes with green annuli are gateways.

wireless network setting we are only allowed to connect points within distance 1. Furthermore, the empty-circle rule is a global rule that is not suitable for local computation. To deal with those two problems, we define the *restricted Delaunay graph* (RDG) and show that it has good spanning properties and is easy to maintain locally.

Definition 4.2.3. *A restricted Delaunay graph of a set of points in the plane is a planar graph and contains all the Delaunay edges with length* ≤ 1 *(called short D-edges).*

Notice that the restricted Delaunay graph always exists and is not necessarily unique. Figure 4.2 shows one example of a restricted delaunay graph. By its planarity, we also know that RDG is sparse, i.e., it has linearly many edges in terms of the number of nodes. In the following, we will first show that any RDG has nice spanning properties.

Spanning properties of R

In this subsection we study the spanning properties of the restricted Delaunay graph with the unit disk graph, under both Euclidean and topological measure. Under Euclidean measure we study the weighted graphs, where an edge has a weight of its Euclidean distance. Under topological measure we assume each edge has a weight 1. The Euclidean (topological) stretch factor represents the worst case ratio of the shortest path length in the restricted Delaunay graph versus that of the unit disk graph.

The Euclidean spanning property of the Delaunay graph was first proved in [49] and later improved in [88]. We extend the proof in [49] to show that the graph with only short D-edges has a constant Euclidean stretch factor, with respect to the communication graph with all the edges shorter than 1. Then the RDG graph is also an Euclidean distance spanner, since it contains all the short D-edges.

Lemma 4.2.4. *Given a set of points S in the Euclidean plane, we define $SD(S)$ to be the graph with only the Delaunay edges with length no more than 1, and $I(S)$ to be the unit disk graph on S. Then, for any $u, v \in S$, $\pi_{DS}(u,v) < C_1 \cdot \pi_I(u,v)$, where $\pi_G(u,v)$ is the Euclidean shortest path distance in G, $C_1 = \frac{1+\sqrt{5}}{2}\pi \approx 5.08$. That is, $DS(S)$ is a Euclidean spanner graph with stretch factor of at most 5.08.*

Proof: It suffices to prove that for any two nodes $u, v \in S$ with Euclidean distance $\ell \leq 1$, there exists a path in $DS(S)$ connecting them whose total Euclidean length is at most $C_1 \cdot \ell$. We can use the following spanning property proven for regular Delaunay triangulations by Dobkin et al. [49]: for any two nodes u, v, there exists a path in the Delaunay triangulation that lies entirely inside the circle with uv as the diameter, and the path length is no more than $\frac{1+\sqrt{5}}{2}\pi \cdot \ell$. For any two points in the circle with uv as the diameter, their distance is at most $\ell \leq 1$. Therefore, all the edges in the path constructed in [49] are short D-edges, which all exist in $DS(S)$. □

While the above lemma shows that RDGs are good Euclidean spanners, a RDG is not necessarily a good topological spanner. We can construct an example where n points are aligned uniformly in a unit interval. In the RDG the shortest path

may have $\Theta(n)$ hops. However, if the nodes are distributed with constant density, i.e., there are $O(1)$ nodes in any unit disk in the plane, then we can also show the topological stretch factor is bounded. Fortunately, the clusterheads and gateways have constant density by Corollary 4.2.2.

Lemma 4.2.5. *For a set of nodes S with constant density, a RDG is a topological spanner graph with constant stretch factor. That is, for any two nodes u, v in S, $\tau_{RDG}(u, v) \leq C_2 \cdot \tau_I(u, v)$ for some constant $C_2 > 0$.*

Proof: Since S has constant density, in the proof of Lemma 4.2.4, there are at most $O(1)$ points in the disk with uv as the diameter. Thus, the path in the RDG has a constant number of intermediate nodes. That is, the RDG is a topological spanner graph with constant stretch factor. □

In addition, the routing graph R that includes the RDG and the edges from clients to their clusterheads, is an Euclidean and topological spanner.

Theorem 4.2.6. *The routing graph R is an Euclidean and topological spanner graph with constant stretch factor.*

Proof: We denote by I_{CG} the unit disk graph and G_{CG} the RDG on the clusterheads and gateways. Suppose that the topological shortest path between u and v is $P = u_1 u_2 \dots u_{k+1}$, where $u_1 = u$, $u_{k+1} = v$. Suppose that the clusterhead of u_i is c_i. Since

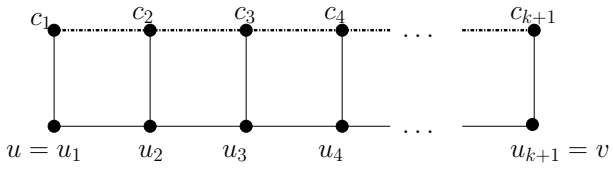


Figure 4.3. Spanner property of the routing graph R.

node u_i and u_{i+1} are visible to each other, there must exist a pair of gateway nodes between clusterheads c_i and c_{i+1}, i.e., $\tau_{I_{CG}}(c_i, c_{i+1}) \leq 3$ (Figure 4.3). From lemma 4.2.5, there exists a path P_i in the RDG whose length is at most $C_2 \cdot \tau_{I_{CG}}(c_i, c_{i+1})$.

We define the path P' to be the union of P_i and the edges u_1c_1 and $u_{k+1}c_{k+1}$. Then $\tau_R(u,v) \leq 2 + \sum_{i=1}^{k} C_2 \cdot \tau_{I_{CG}}(c_i, c_{i+1}) \leq 3C_2 \cdot k + 2$.

The Euclidean spanner property follows from the constant density of the clusterheads and gateways. Basically all the c_i lie in a region whose area is linear to the Euclidean length of P. Due to the constant density argument, the number of clusterheads and gateways inside the region is also linear to the length of P. So the length of the path P' is only a constant times the length of P. □

4.3 Maintaining the routing graph

In this section we discuss the maintenance of the routing graph in the distributed setting when the nodes move around. The challenge here is that each node only has a fixed communication range and only performs local computation. We aim to design an algorithm that enables each node to efficiently and consistently maintain the relevant part of the routing graph, with only the knowledge of the neighbors. The maintenance of clusterheads is covered in chapter 3. Here we focus on the maintenance of restricted Delaunay graphs and gateway nodes.

4.3.1 Maintaining RDG

According to Lemma 4.2.5, any RDG has the desired spanning property. We will describe a distributed algorithm which maintains an RDG at any instance of time. At the end of the algorithm, each node u maintains a set of edges $E(u)$ incident to u, and those edges satisfy that (1) each edge in $E(u)$ is short, i.e. of length ≤ 1; (2) the edges are consistent, i.e. the edge uv is in $E(u)$ if and only if it is in $E(v)$; (3) the graph obtained is planar, i.e. no two edges cross; and (4) all the short Delaunay edges are guaranteed to be in the union of $E(u)$'s.

The algorithm works as follows. First, each node u acquires the position of its neighbors $N(u)$ and computes the Delaunay triangulation, denoted by $T(u)$, on $N(u)$. Since $T(u)$ is computed only on $N(u)$, the edges we obtained is a superset of the short Delaunay edges, and some of them might be non-Delaunay edges. Furthermore, the

```
E(u) := {uv|uv ∈ T(u)}
For each edge uv in E(u)
    For each w in N(u)
        If (u, v ∈ N(w) and uv ∉ T(w)) then
            delete uv from E(u)
```

Figure 4.4. Pseudo-code for resolving inconsistency.

local graphs at different nodes might be inconsistent, i.e. an edge uv is in u's local graph but not in v's. Due to the inconsistency, the union of local graphs might not be planar although they are planar individually. To resolve these problems, we perform one-hop information propagation. Each node u sends $T(u)$ to all of its neighboring nodes. Then node u deletes an edge uv from $E(u)$ if u was told by one of its neighbors w that uv is not present in w's local Delaunay triangulation. A pseudo-code executed at each node u is shown in Figure 4.4.

Now, we will argue that after the one-hop information propagation, all the $E(u)$ satisfy the stated properties. The invariant the above pseudo-code achieves is that for each visible pair u and v, the edge uv belongs to $E(u)$ if and only if $uv \in T(w)$ for all $w \in N(u) \cap N(v)$ (notice that $u, v \in N(u) \cap N(v)$ since u, v are mutually visible). If an edge uv is a short Delaunay edge, it has to present in all the local graph $T(w)$ for $w \in N(u) \cap N(v)$. Therefore, the properties (1),(2), and (4) hold. The following simple geometric fact shows that property (3) holds as well.

Lemma 4.3.1. *For two visible pairs uv and wx, if the edges uv and wx cross, then one of the four nodes sees all of the other three.*

Proof: Assume that uv (of length ≤ 1) and wx (of length ≤ 1) intersect at point p. By triangular inequality, $|wp| + |up| \geq |uw|$ and $|vp| + |xp| \geq |vx|$. Summing these two equations, we have that $|uv| + |wx| \geq |uw| + |vx|$. Therefore, either uw or vx has length ≤ 1. Similarly, either ux or vw has length ≤ 1. No matter in which case, the endpoint shared by two short edges sees all three other points. □

By Lemma 4.3.1, we now argue that no crossing exists in the final graph. Suppose that the edge $uv \in E(u)$ intersects the edge $wx \in E(w)$. Then by the lemma, one of

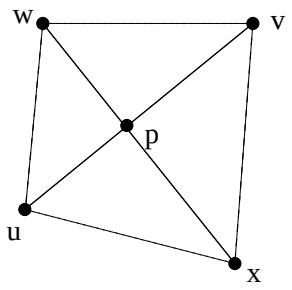


Figure 4.5. Property of a pair of crossing edges.

u, v, w, x, say u, sees all the four nodes. Therefore, w must have received $T(u)$ when computing $E(w)$. According to Figure 4.4, both uv and wx must present in $T(u)$, contradicting that $T(u)$ is a planar graph. The above procedure could be expensive if $N(u)$ contains many nodes. Fortunately, it is not the case in our setting because we apply this algorithm on clusterheads and gateways. According to Corollary 4.2.2, those nodes have constant density. Therefore, $E(u)$ can be computed in constant time for each u.

When the nodes move around, we can maintain the restricted Delaunay graph, provided that we can maintain the clusterheads and the gateway nodes (Section 4.3.2). Basically, we maintain the local Delaunay triangulation for each node. When a node u enters or leaves another node v's communication range, u and v update their own local Delaunay triangulations and inform their neighbors of this update. Their neighbors then update their edge sets accordingly. Notice that such update only happens within one hop of nodes u and v and won't propagate to other nodes. So the restricted Delaunay graph can be maintained efficiently.

4.3.2 Maintaining gateway nodes

As the wireless nodes move around, we intend to maintain the gateway nodes. We present here an algorithm to let clusterheads select gateways. Note that changes to clusterheads and gateways occur only if two nodes become visible or invisible to each other. We show that when such an event happens, only the nearby nodes of u and v need proper update, and the update time is constant per node.

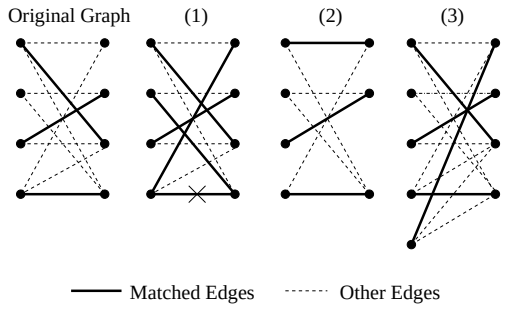


Figure 4.6. A maximal matching in bipartite graph $B(c_1, c_2)$. Left: original graph. (1) A pair of nodes become invisible. (2) A node leaves the cluster. (3) A new node joins the cluster.

For two clusterheads c_1 and c_2, we define a bipartite graph $B(c_1, c_2)$ with vertices $C(c_1) \cup C(c_2)$. The edge pq is in $B(c_1, c_2)$ if $p \in C(c_1)$, $q \in C(c_2)$, and p is visible to q. The edges in the bipartite graph $B(c_1, c_2)$ represent all eligible gateway pairs between c_1 and c_2. A naive method is to let c_1 and c_2 maintain this bipartite graph, which might be of size $O(|C(c_1)| \cdot |C(c_2)|)$. Here we propose a memory efficient method to avoid storing all the edges of $B(c_1, c_2)$. We only maintain a maximal matching $M(c_1, c_2)$ at c_1. A matching is a subset of edges in $B(c_1, c_2)$ such that no two edges share the same end points. A maximal matching is a matching such that no edge can be added. A maximal matching has size $O(|C(c_1)| + |C(c_2)|)$. Figure 4.6 shows an example. If pq is an edge in the matching, we call p is matched to q, or to c_2 ($q \in C(c_2)$), or simply, p is matched. By maintaining maximal matchings, we can save storage and reduce update time to $O(1)$ per node.

The property of maximal matching guarantees that if there is at least one edge in the bipartite graph, i.e. clusterheads c_1 and c_2 can be connected via gateways, all maximal matchings have to contain at least one edge too. To maintain the maximal matching record, a clusterhead c_1 maintains the pair (p, c_2), where p is visible to c_1, p is matched to q, and $q \in C(c_2)$. For each matched node p (which may or may not be chosen as a gateway node), p maintains the pair (q, c_2), where p is matched to q, and $q \in C(c_2)$. At the beginning, after proper rounds of information propagation, each clusterhead pair would properly select a maximal matching from the bipartite graph. (to make the matching consistent on both sides, we let the clusterhead with higher ID selects the matching and informs the other clusterhead). We let the clusterhead with

c_1	c_2	Nodes in $C(c_1)$ and matched to c_2
		Nodes in $C(c_1)$ and *not* matched to c_2
	c_3	Nodes in $C(c_1)$ and matched to c_3
		Nodes in $C(c_1)$ and *not* matched to c_3
	⋮	⋮
c_2	c_1	Nodes in $C(c_2)$ and matched to c_1
		Nodes in $C(c_2)$ and *not* matched to c_1
	⋮	⋮
⋮	⋮	⋮

Figure 4.7. Organizing neighbors.

higher ID select a gateway pair out of the available matching. As points move around, if the previous selected gateways are no longer valid as indicated by the matching, the clusterhead would select another gateway pair out of the current matching.

Now we show how to maintain a maximal matching. We first describe how the neighborhood information is organized inside a node u. To be more specific, each node v would propagate information by broadcasting an update entry of the following form | ID | c | c' | c_1 | c_2 | ... | , where the ID uniquely identifies v, c and c' are the ID's of the clusterheads of the clusters that v belongs to (recall that a node may belong to two clusters), $c_1, c_2, \ldots$ are the ID's of the clusterheads v is matched to. Note that since clusterheads have constant density, each such entry is of constant size. Then u would organize its neighbors' entries into a table that is indexed by a pair of clusterhead ID's (Figure 4.7). The first index is the ID of a clusterhead that a neighbor belongs to, and the second index is the ID of a clusterhead that a neighbor is matched (not matched) to. This table enables $O(1)$ lookup time to find a neighbor, whose clusterhead is c_i and is currently matched (not matched) to c_j. For u, the indices of the table don't have to include all pairs of clusterheads, rather only clusterheads that are nearby. Also a neighbor v might appear in the table several times. But by the constant density argument, at each node the total number of pairs of indices in the table is a constant, each neighbor only appears a constant number of times. So the total storage at each node is still linear to the neighborhood size. The table and the maximal matching record are updated upon receiving any update

entry from neighbors.

Changes of the maximal matching can only happen when two nodes begin or stop seeing each other, this may also cause one client in $C(c_i)$ to change clusterhead. We will discuss these situations separately.

1. When two nodes p and q begin to see each other, $p \in C(c_1)$, $q \in C(c_2)$. The change takes place if both p, q are not matched with respect to clusterheads c_2 and c_1. They become matched in $B(c_1, c_2)$ by adding c_2 and c_1 to the update entry respectively. Once the update entry is modified, a node broadcasts the entry to its neighbors. If two nodes p, q stop seeing each other and they are matched before in $B(c_1, c_2)$, we need to find out if they can be matched with other nodes in the same bipartite graph. To do this, p and q look into their neighbor set and find unmatched nodes in $C(c_2)$ and $C(c_1)$ respectively (Figure 4.6(1)). For example, p looks for a neighbor q' with c_2 as one of the clusterheads and q' is not matched to c_1.

2. When one node changes its clusterhead. This involves p disappearing in the original cluster and appearing in the new cluster. When p disappears in $C(c_1)$, if p is not matched at all, nothing needs to be done. If not then p needs to broadcast the clusterhead change to its neighbors. Notice that because of the constant density of clusterheads, p participates in at most a constant number of matchings in total. So a clusterhead change would only affect a constant number of nodes in the graph. Suppose p was matched to some node q in some $C(c_2)$, once q receives message from p about clusterhead changes, q needs to search its neighbors for potential matchings (Figure 4.6(2)). When p appears in $C(c_1)$, p needs to find among its neighbors that belong to some cluster $C(c_2)$ and are currently not matched to c_1. This can be done in a similar way as described in the previous situation(Figure 4.6(3)).

4.4 Quality analysis of routing graphs

The restricted Delaunay graph is shown to be a planar graph with constant stretch factor. Therefore it can be used in the perimeter routing stage of a geographical routing scheme to help a packet get out of the local minima. In the current geographical routing scheme, for example, the GPSR protocol [87], two planar graphs are used. Relative Neighborhood Graph (RNG) is defined such that an edge (u, v) exists if there is no other node w whose distances to u and v are less or equal to the distance between u and v. Gabriel Graph (GG) is defined such that an edge (u, v) exists if no other node w is inside the circle with the diameter uv (Figure 4.8).

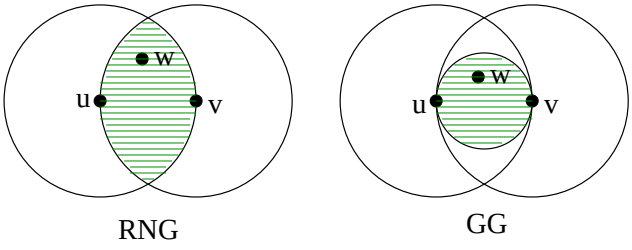


Figure 4.8. RNG and GG.

As planar graphs, RNG and GG can effectively help on the local minima phenomena. But they are not spanners. Bose et al. [35] proved that the Euclidean stretch factor of GG and RNG are $\Theta(\sqrt{n})$ and $\Theta(n)$, respectively, where n is the number of points. The same construction also implies that even for constant density point set, the topological stretch factors can be $\Omega(\sqrt{n})$ for GG and $\Theta(n)$ for RNG. Thus they impose limits on the quality of routing paths discovered by GPSR. In other words, the routing paths used by GPSR might be far longer than the shortest path possible, because in RNG or GG, a path with length comparable with the shortest path might not even exist!

The geographical routing on a restricted Delaunay graph produces routing paths with better quality, since a RDG has both constant Euclidean and topological stretch factor. When a packet gets stuck at a node, it will travel along a face of the RDG until either the destination is reached or greedy forwarding can be performed again. The RDG shortens such travel. In the RDG, routing is done in a much smaller graph on

clusterheads and gateways only. Specifically, the case when a packet follows a series of short edges in RNG or GG is not going to happen in a RDG. Since the clusterheads and gateways have constant density, the routing paths contains only relatively long paths. A routing path on RDG make substantial progress towards its goal. This is also observed through simulation, as will be described in the next section.

Although in the worst case, the routing path produced by the geographic routing with RDG can still be much longer than the shortest path. We can show the quality of routing paths is bounded in some special cases. The first case is when the greedy forwarding never gets stuck at a local minima. As discovered in [87], this is the typical case.

Theorem 4.4.1. *If a packet can be greedily forwarded from u to v, i.e. no local minimum is reached during the forwarding, then the routing path length is bounded by $O(\ell^2)$ if the shortest path between u, v has length ℓ.*

Proof: Recall that by greedy forwarding, each time we check all the visible neighbors and forward the packet to the one closest to the destination. Let the path be: P_{uv} : $u_1 = u, u_2, \ldots, u_m = v$. Note that the distance between u_i and v is decreasing when i increases. Since the optimum path is of length ℓ, the distance between u and v is at most ℓ. Thus all u_i's lie in a circle of radius ℓ centered at v. Also, we know that the points u_i and u_{i+k}, for $k \geq 2$, cannot see each other because otherwise we would have chosen u_{i+k} instead of u_{i+1} as the successor of u_i in the path. Therefore, the points $u_1, u_3, \ldots, u_{2\lceil m/2 \rceil - 1}$ are mutually invisible. According to a simple packing lemma, we know that there can be at most $O(\ell^2)$ such points in a disk with radius ℓ. □

Note that the above bound is tight. Figure 4.4 (a) illustrates a situation where a path with $\Theta(\ell^2)$ nodes is discovered by greedy geographic forwarding while the optimum path has length ℓ.

Similarly, we can also prove the quality of perimeter routing on our graph, again in a special case. We call that a perimeter routing follows right-hand (left-hand) rule, if we always traverse a face in a counter-clockwise (clockwise) direction. We call a path connecting u, v right-sided (left-sided) if the path lies entirely on the right (left) side of the line passing through u, v. Then, we have that

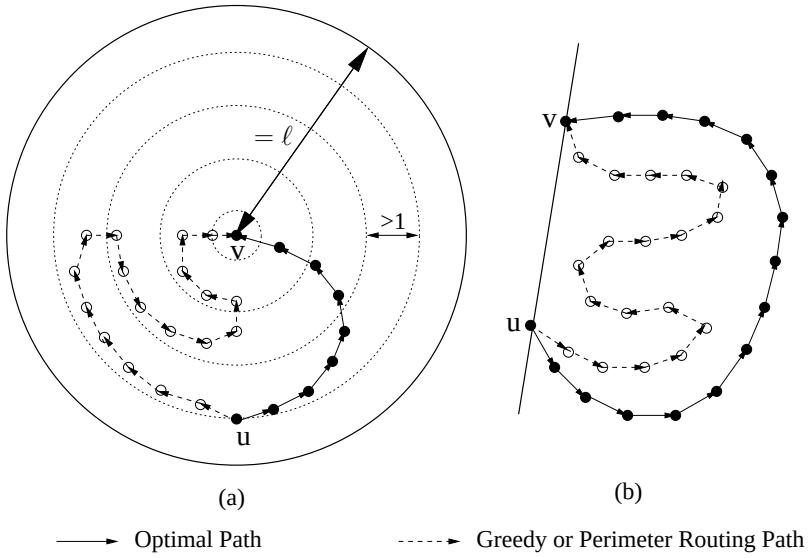


Figure 4.9. Examples of greedy forwarding and one-sided perimeter routing.

Theorem 4.4.2. *If the shortest path is right-sided (left-sided) and has length ℓ, then the path discovered by the perimeter routing following right (left) hand rule has length at most $O(\ell^2)$.*

Proof: Suppose that the optimum path is to the right of line uv. If the perimeter routing follows the right-hand rule, then all the points traversed lie entirely inside the region bounded by the line segment uv and the optimum path from u to v (Figure 4.4 (b)). The area of that region is $O(\ell^2)$. By the constant density property, the number of nodes in that region is at most $O(\ell^2)$. Therefore, the length of the path is at most $O(\ell^2)$ as well. □

The above theorem does not specify a way to figure out which side the shortest path lies. This is in general a difficult question in perimeter routing — by following a wrong direction, we may have to traverse a very long path while a short path exists by following the other direction. We do not know of any good local rule to resolve this problem. However, one trick one may use is to try both directions. Specifically, we can forward the packet to the right t hops, and then come back to u and forward the packet to the left t hops. Then we double t's value and repeat the process until either we reach a point where greedy forwarding is available, or we enter another face,

or we come back to the starting edge, which means there is no path between u, v. We may obtain competitive bounds about this method, but the details are omitted in this chapter.

Another advantage of using RDG as the routing graph is the saving of the cost of routing decisions at each node. In GPSR, at the greedy forwarding stage, the next hop is selected as the one who is closest to the destination. Such a decision is made after the examination of all the neighbors. Since a node can have a large number of neighbors, the cost of routing decision can be high. In contrast, the RDG is built on clusterheads and gateways and one node has only a constant number of neighbors in RDG. Thus the routing decision can be made after one examines a constant number of nodes.

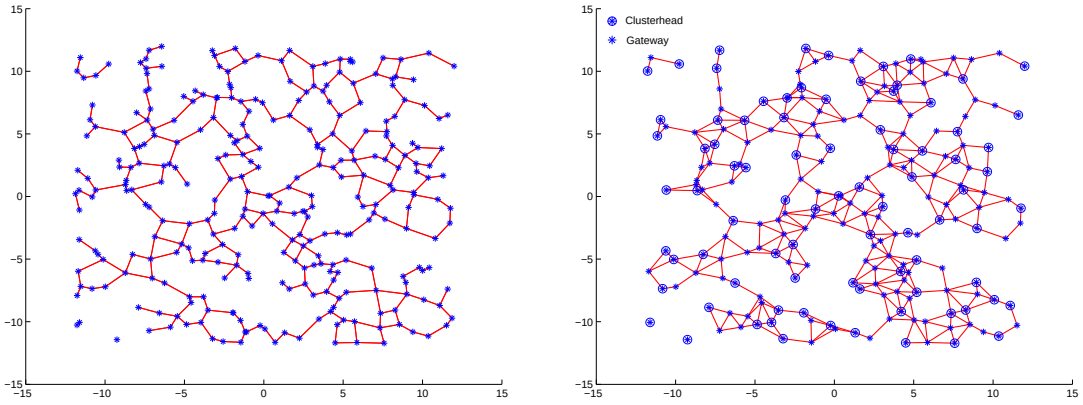
4.5 Simulations

In the previous sections, the analysis are mostly theoretical and help us to understand the quality of the algorithm in the extreme cases. To demonstrate the quality of our algorithm, we have also performed simulations on nodes under both uniform and non-uniform distributions.

Uniform distribution

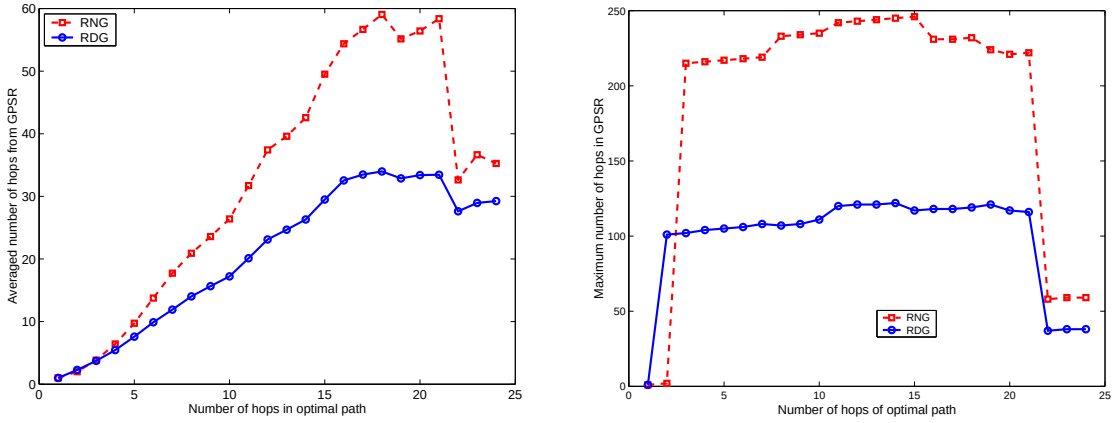
In this simulation, we used 300 nodes randomly distributed in a square of side length 24. Each node can see all the nodes in a disk of radius 2 around itself. The density of nodes is about 8. We use the one-level clustering algorithm to select the clusterheads. We evaluate the quality of the paths found by GPSR on the RNG and RDG. The RDG on clusterheads and gateways is shown in Figure 4.10(b). The RNG is shown in Figure 4.10(a).

Compared with the RNG, the RDG is a sparser backbone with fewer nodes. Therefore, when we do perimeter routing along a face in RDG, the number of hops is much smaller than in the RNG. This is also shown by simulation. Figure 4.11 shows the comparison of the performance of GPSR in the RNG and RDG. For all pairs of reachable nodes, we compute the number of hops of the optimal path, and the paths we get



(a) (b)

Figure 4.10. (a) RNG (b) RDG on a set of nodes with uniform distribution.



(a) small(b)

Figure 4.11. (a) Averaged length using GPSR vs. optimal length (b) Maximal length using GPSR vs. optimal length.

using GPSR on the RNG and RDG. For pairs with the same optimal length, we take the average length of the paths obtained by GPSR. The RDG outperforms the RNG in terms of the routing path quality. Figure 4.11 (a) is the comparison of the average path length of GPSR on the RDG and the RNG. Figure 4.11(b) shows the maximal number of hops by GPSR on RNG and RDG. If we examine all the unreachable pairs, on average 67 hops are travelled in RDG and 139 hops are travelled in RNG.

We also do experiments for moving nodes. We assume that every node moves with a constant velocity in a random direction at a random speed between 0 and 1. Nodes are constrained to move within the square (of side length 24), so a node bounces back once hitting the virtual boundary[3]. In this study we are interested in how the path quality between 2 fixed nodes changes over time. We track the topology of the network under motion over 1000 frames at 1 frame per second. We then compute the path length between these two specific nodes per each frame. On average RDG outperforms RNG by more than 37% (23 hops vs. 37 hops).

Non-uniform distribution

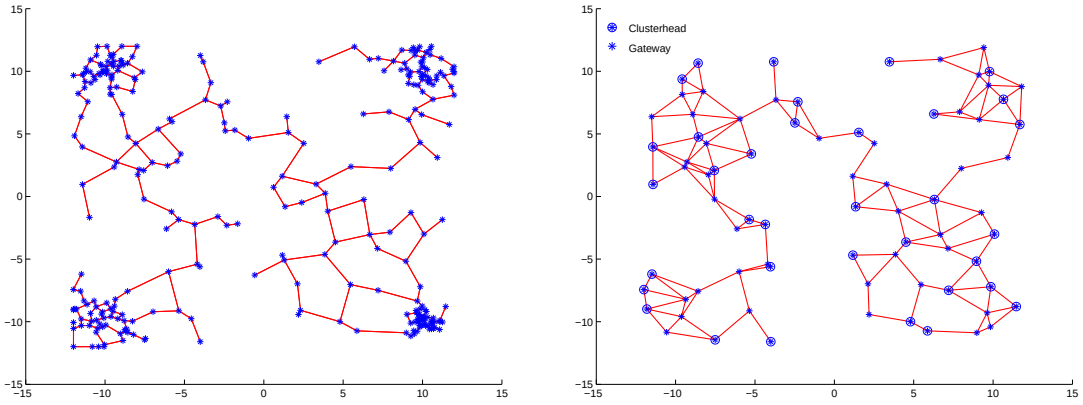
In the real world, the wireless nodes are usually far from uniformly distributed. In this case, the advantage of the RDG over the RNG is shown more obviously by the simulation. Here we show a simulation with 300 nodes, 100 nodes are randomly distributed, and another 200 nodes are clustered in four groups. The size of a node's visible range is a disk of radius 3.5. The RNG and RDG are shown in Figure 4.12.

The comparison of path length in RNG and RDG is shown as Figure 4.13. We can see from the figures that most of the packets follow a shorter path in RDG, compared to RNG. The advantages are clearer when the length of the optimal path gets longer.

4.6 Discussion

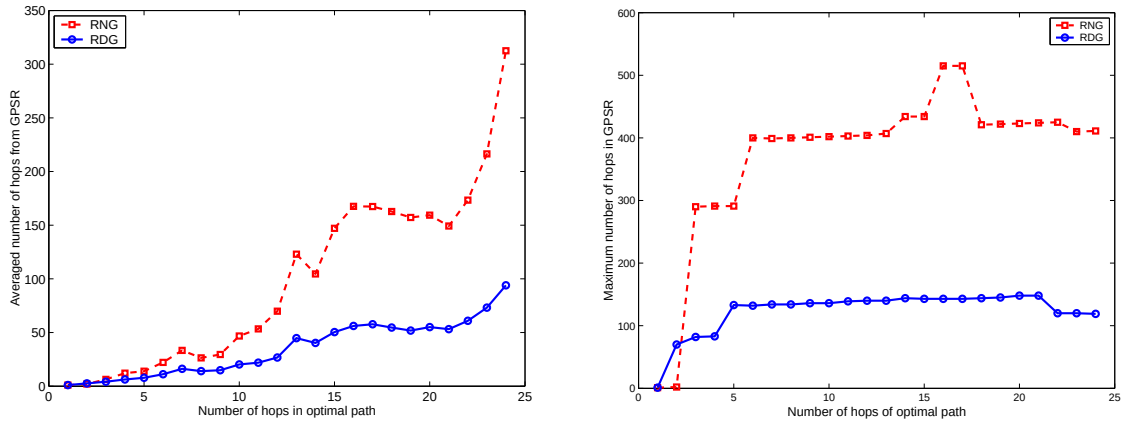
We discuss some other practical issues in implementing and measuring the method in this chapter. We also give the results on how to find a short path in wireless *ad*

[3]this models the situation that one point moves into and one point moves out of the specified region.



(a) (b)

Figure 4.12. (a) RNG (b) RDG on a non-unform distribution.



(a) (b)

Figure 4.13. (a) Averaged length using GPSR vs. optimal length; (b) Maximal length using GPSR vs. optimal length.

hoc networks, to make the problem more complete.

4.6.1 Scaling vs. spanner property

Besides the spanner property, another desirable property of the routing graph is that every node has small degree, so no node will be overloaded. However, there is a trade-off between the constant degree and the spanner property. If we allow constant degree of the routing graph, the spanner property can't be achieved. Consider the situation in which n nodes are close to each other such that every node can see all the other nodes. If we let each node's degree in the routing graph be at most C, then one node x can reach at most C nodes in one hop, and C^2 nodes in two hops, and C^k nodes in k hops. Then there must exist a node that x can reach in at least $\log n$ hops.

4.6.2 Efficiency of clusterheads

One common issue in using clusterheads in routing protocols is that, frequent clusterhead changes may adversely affect the performance of the routing protocols since nodes are busy in clusterhead selection rather than packet forwarding. For example, consider the Clusterhead Gateway Switch Routing (CGSR) protocol proposed by Chiang et al. [45]. Each node keeps a cluster member table, where it stores the destination clusterhead for each mobile node in the network. So when a node changes its clusterhead, the updated information must be broadcast to every node in the network, which causes a lot of traffic. In addition, each node keeps a routing table that is used to determine the next hop in order to reach the destination. Changes of the clusterheads also cause a lot of changes in the routing table.

However, the above is not a problem in our routing graph, GPSR doesn't require any routing tables. The routing graph changes locally and needs not be broadcast over the whole network. Changes to clusterheads and gateways occur only when the underlying communication graph changes. As shown in Chapter 3, if all the nodes follow bounded-degree algebraic motion, the number of changes of our clustering is at most $O(n^2 \log\log n)$, which is near optimal. Any routing graph such as the RNG

or GG needs to be updated according to the network topology as well. On the other hand, under certain conditions RDG doesn't change, while both GG and RNG suffer from a lot of changes. Consider nodes moving on a line with the same speed, except a special node moves faster. Since the probability that the fast node has the ID high enough to be a clusterhead is small, most of the time the RDG doesn't change. But the RNG or GG could change $\Omega(n)$ times.

4.6.3 Finding a short path

This chapter proposed a spanner for the *ad hoc* wireless networks which *contains* a short path. To make the problem complete, the natural follow-up question is how to find this path.

If we only have local information, i.e., each node only knows the nodes within a constant number of hops, then there is a lower bound construction showing that any online algorithm finds a path of length $\Omega(k^2)$ if the shortest path has length k [99], as shown in Figure 4.14. Kuhn *et al.* [99] also proposed an localized geometric routing algorithm that achieves the $O(k^2)$ bound.

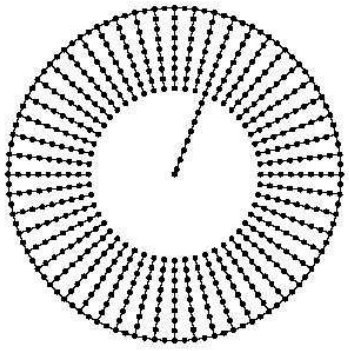


Figure 4.14. The lower bound construction for online localized routing problem, from [99]. There are multiple spikes on a circle pointing inside. Only one of them connects to the destination which lies at the center of the circle. Any local algorithm has no idea which spike will lead to the destination and has to try all of them in the worst case.

On the other hand, if the topology of the whole network is available, Chapter 7 proposes an algorithm that preprocess the unit-disk graph into a structure of size

$O(n \log n/\varepsilon^4)$ such that a $(1+\varepsilon)$-approximate shortest distance (path) query is answered in $O(1)$ ($O(k)$) time, for any $\varepsilon > 0$.

Part III

Load Balanced Routing

Chapter 5

Load Balanced Short Path Routing

5.1 Introduction

In a wireless ad hoc network, the nodes are usually energy constrained. Therefore it's very important to design energy efficient routing schemes. The shortest path routing scheme, not only minimize the routing delay, but also minimize the total energy assumption, since the energy needed to deliver a packet is correlated to the length of the routing path. However, the shortest path routing ignores "fairness" on the nodes — it may use the same set of nodes to relay packets for the same source and destination pair. This will heavily load those nodes on the path even when there exist other feasible paths. An uneven use of the nodes may cause some nodes die much earlier, thus creating holes in the network, or worse, leaving the network disconnected. In addition, unbalanced use of the nodes may discourage them to participate in the routing. In order to prolong the lifetime of a wireless network and avoid uneven energy consumption of the nodes, we hope to design load balanced routing schemes that minimize the maximum energy consumption of any node. The energy consumption of a wireless node is dominated by the energy used to transmit packets. So we measure its energy consumption by the total size of packets relayed by the node. Then the load balanced routing can be thought as to minimize the maximum load on the nodes in the network.

We hope to find an ideal routing algorithm that minimizes the latency and the

maximum load simultaneously. However, these two goals are conflicting to some extent: the shortest path routing restricts the resources that can be used, while load-balanced routing aims to use all the available resources to even the load. Indeed, one can easily construct an example to show that these two goals are conflicting, i.e., a routing algorithm limited to using paths with lengths at most c times those of the shortest paths will necessarily cause some node to have $\Omega(n/c)$ loads while the optimal load-balancing algorithm only create $O(1)$ load on every node. See Figure 5.1. In this

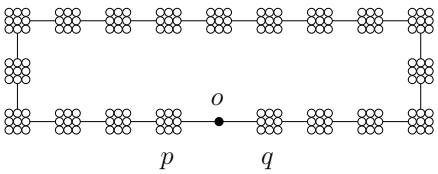


Figure 5.1. The packets from spot p to q either go through node o, thus causing o to be heavily loaded, or route along a long path, thus having large latency.

chapter, we show that for a realistic distribution of the nodes, we can find algorithms that achieve good performance in terms of minimizing latency and maximum load simultaneously.

The case we study is when the nodes are located in a "narrow" strip with width at most $\sqrt{3}/2 \approx 0.86$ times the communication radius of each node. This model captures the situation when the nodes are on highways or streets, for examples, the wireless networks on vehicles [74] or people walking on streets. In such cases, routing can be done in two phases. In the broad phase, the nodes figure out the "meta-path" needed to route a packet with the aid of the position information and the underlying transportation network map. In the second phase, the routing is done on the nodes along the meta-path. The second problem reduces to routing on nodes located in a narrow strip.

What makes routing in a narrow strip easier is that, if there exists a path from the source to the destination, then a greedy scheme that always forwards the nodes to the correct direction guarantees to reach the destination. This also opens the door to local routing algorithms. For example, the short path routing can be implemented by a greedy scheme, where the packet is sent to the furthest reachable node in the

right direction. As we mentioned earlier, this may create heavily loaded nodes. So we revise the greedy strategy to take into account the current load of the nodes and achieve load balancing simultaneously.

The basic idea of our methods is that we maintain, for each node, a set of edges, called bridges, that are guaranteed to make substantial progress. A node chooses the "lightest" bridge to relay a packet. We show that this simple algorithm has good performance in terms of both path length and maximum load. In addition, we show that the bridges can be dynamically maintained by using only local information. Specifically, we can guarantee the following properties of our algorithm.

1. It uses only short paths: the number of hops of the path used is at most four times that of the shortest path;

2. It balances the load: the maximum total size of packets relayed by any node is at most three times as much as the optimum load balancing algorithm;

3. It is local and scales to large networks: each node only needs information in its local neighborhood to make routing decisions; and as a consequence, our algorithm handles dynamic change and mobility efficiently since only a node's neighborhood is affected;

4. It is online and "globally oblivious": the routing decision of a packet depends only on the previously routed packets, i.e., the current state of the network. It doesn't need to know the packets in the future.

In addition to providing rigorous analysis, we also implement the algorithm and show that the good performance of our algorithm is supported by the simulation results as well.

The Chapter is organized as follows. In Section 5.2, we introduce some definitions and notations. In Section 5.3 and 5.4, we describe the algorithm for the nodes that are aligned on a line and its efficient implementation. Then, we show that the similar technique can be extended for nodes that are inside a narrow strip in Section 5.5. In Section 5.6, we show the simulation results of our algorithm.

5.2 Notations and definitions

Given a set of n nodes S in the plane, the *communication graph* of S is an unweighted unit-disk graph $I(S) = (S, E)$, where $(p, q) \in E$ if the Euclidean distance between $p, q \in S$ is at most 1. The *length* of a path P, denoted by $|P|$, is the number of nodes on the path. A path P connecting $p, q \in S$ is called *α-short* if the length of P is at most α times that of the shortest path between p and q. α is also called the *stretch factor* of P.

A routing request is represented by $r = (s, t, \ell)$ where s, t, ℓ represent the source, destination, and packet size, respectively. To satisfy a request r, a path P_r between s and t is used to relay the packet. For a set of requests R, a path set $\mathcal{P}$ satisfies R, denote by $\mathcal{P} \models R$, if $\mathcal{P} = \{P_r \,|\, r \in R\}$ where P_r satisfies r. Similarly, the *stretch factor* of $\mathcal{P}$ is defined to be the maximum stretch factor of the paths in $\mathcal{P}$. $\mathcal{P}$ is called α-short if every path in $\mathcal{P}$ is α-short. For example, the shortest path routing algorithm always produces 1-short paths.

For a set of requests R satisfied by $\mathcal{P}$, the *load* $\ell(v)$ incurred to $v \in S$ is the total size of the packets that pass v, i.e. $\ell(v) = \sum_{v \in P_r} \ell_r$. The maximum load $\ell(\mathcal{P})$ of $\mathcal{P}$ is then defined to be $\max_{v \in S} \ell(v)$. Denote by $\ell^*(R)$ the maximum load of the optimum load balanced routing, i.e. $\ell^*(R) = \min_{\mathcal{P} \models R} \ell(\mathcal{P})$. The *load-balancing ratio* of $\mathcal{P}$ is defined as $\ell(\mathcal{P})/\ell^*(R)$. An algorithm is said *$\beta$-balanced* if for any set of requests R, the load-balancing ratio is at most β. In this Chapter, our goal is to design wireless routing algorithms with both small stretch factor and small load-balancing ratio.

5.3 Load balanced routing on a line

In this section, we focus on the special case when all the nodes are aligned on a line. We first show that the optimal load balanced routing in this simplest case is still a hard problem. We then describe an algorithm that achieves both constant stretch factor and constant load-balancing ratio without worrying about the algorithmic issue. An efficient distributed implementation is presented in the next section.

5.3.1 Hardness of load balanced routing

The optimum load balancing is difficult even for a simple network, as shown in Figure 5.2 (i). Suppose that each node x_i wants to send a packet with size ℓ_i to the node y_i. They have to choose, from z_1, z_2, a node to relay the packet. The optimum solution distributes the packets evenly on z_1, z_2. This is exactly the knapsack problem, a well-known NP-hard problem. In fact, if there are m nodes $z_1, \cdots, z_m$, inside the intersection of the communication ranges of x_i's and y_i's, then minimizing the maximum load on the m nodes becomes the on-line load balancing problem on m identical machines. Even obtaining an approximation within a ratio of 1.852 has been proven NP-hard [10].

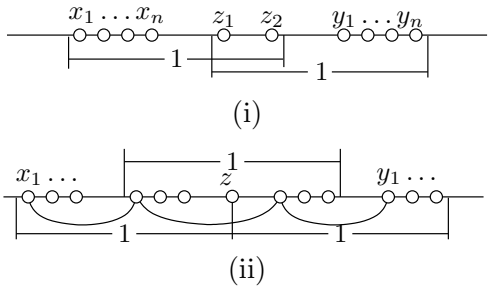


Figure 5.2. Load-balanced routing is hard. The problem in (i) is equivalent to the knapsack problem. In (ii), the shortest path from x_i to y_i all pass through z, but one can evenly distribute the load by using the path as shown in the figure.

Next, we show that it is impossible to optimize the stretch factor and the load-balancing ratio simultaneously. In Figure 5.2 (ii), when x_i sends a packet to y_i, if we use the shortest path, then all the packets have to pass the node z while we may evenly distributed the packets as shown in the figure.

5.3.2 Requests with unit packet size

We start with the case when all the requests have the same packet size and propose a greedy algorithm that is 2-short and 2-balanced. Suppose that all the nodes lie on the real line. For each node $p \in S$, denote by x_p the coordinate of p. Then the communication range of p is the interval $I(p) = [x_p - 1, x_p + 1]$. Define the left (right) communication range of p as $I_l(p) = [x_p - 1, x_p)$ ($I_r(p) = (x_p, x_p + 1]$).

The algorithm GREEDY1 works as follows: Each node p_i keeps track of $\ell(p_i)$, the total number of packets it has relayed so far, and also the maximum load in its left and right communication range (p_i exclusive), denoted by $\ell_l(p_i)$ and $\ell_r(p_i)$, respectively. When a node p_i receives a new request with destination t, it checks if t is within its communication range. If it is, then p_i simply sends the request to t. Otherwise, assume t is to the right of p_i, then p_i sends the request to the furthest node in its right communication range whose load is strictly smaller than $\ell_r(p_i)$, the maximum load in $I_r(p)$. In other words, p_i chooses the next hop to be as far as possible without increasing the maximum load in its right communication range. If all the nodes in $I_r(p_i)$ have the same load, i.e., the maximum load $\ell_r(p_i)$, then p_i sends the request to the furthest node in $I_r(p_i)$. In this case, $\ell_r(p_i)$ is increased by 1.

GREEDY2 is obtained by adding one look-ahead to GREEDY1: When a receives a request, it finds the next hop b according to GREEDY1 and then asks b to find the next hop c. If c is in a's communication range, then a shortcuts b and sends the packet directly to c.

Theorem 5.3.1. *GREEDY2 is 2-short.*

Proof: Suppose that P_r is a left to right path produced by GREEDY1. Take any four adjacent nodes, say a, b, c, d from left to right, along P_r. We claim that a and d are not visible to each other. Suppose otherwise, then a, b, c, d are all mutually visible,

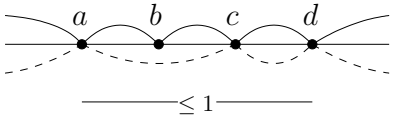


Figure 5.3. For any four adjacent nodes a, b, c, d along a path generated by GREEDY1, a and d are not visible to each other.

as shown in Figure 5.3. Since c, d are both in $I_r(b)$ and b chooses c instead of d to be the next hop, we must have that $\ell(c) < \ell(d)$ by the greedy forwarding strategy. Therefore $\ell(c) < \ell(d) \leq \ell_r(a)$. That is, c is a node further away from a than b and c doesn't have the highest load in a's right communication range. By GREEDY1, a

should have chosen c or a node to the right of c to be the next node, contradicting with the fact that b is the next hop of a on the path.

Since GREEDY2 does one-hop look-ahead based on GREEDY1, a direct consequence of the above fact is that for any two non-adjacent nodes a, b on a path produced by GREEDY2, they are not visible to each other. This also explains why one shortcut is sufficient in GREEDY2. Therefore, the total number of nodes used by GREEDY2 is at most twice that of the shortest path routing, since any unit interval has at most two nodes on a path produced by GREEDY2 and at least one node on the shortest path. This proves that the stretch factor of GREEDY2 is no more than 2. □

Theorem 5.3.2. *GREEDY2 is 2-balanced.*

Proof: We consider the first time when the maximum load on all the nodes reaches $\ell(\mathcal{P})$. Suppose that node i has the maximum load $\ell(\mathcal{P})$ after i relays the request r. That is, no other node has load equal or more than $\ell(\mathcal{P})$, and $\ell(i) = \ell(\mathcal{P})$. Assume that the forwarding direction of r is from left to right. Since i relays the request to

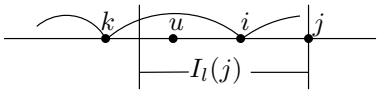


Figure 5.4. Load-balancing competitive ratio of GREEDY2.

a node to its right, then there must be at least one node in i's right communication range. Suppose that j is the node to the immediate right of i in S, and k is the node to the left of i on the path P_r (Figure 5.4). According to GREEDY2, the reason that k chooses i as the next hop and therefore increases the maximum load in k's right range, i.e., $\ell_r(k)$, is because i is the furthest node in $I_r(k)$ and all the other nodes in $I_r(k)$ have the same load $\ell(\mathcal{P}) - 1$. Then k is outside $I_l(j)$, i.e., k and j cannot be visible to each other — otherwise k would have chosen j, instead of i to relay the packet. Therefore all the nodes in j's left communication range $I_l(j)$ must be inside k's right communication range $I_r(k)$.

Assume there are m (≥ 1) nodes inside j's left communication range $I_l(j)$. Then every node $u \in I_l(j)$, except for i, must have load exactly $\ell(\mathcal{P}) - 1$, as shown earlier. Therefore the total load summed over all the nodes in $I_l(j)$ is $(m-1)(\ell(\mathcal{P})-1)+\ell(\mathcal{P}) = m\ell(\mathcal{P})-m+1$. Since a path generated by GREEDY2 has at most two nodes inside any unit interval, according to Theorem 5.3.1, each request r with packet size ℓ contributes at most 2ℓ to the total load over all the nodes in $I_l(j)$. Therefore the number of requests that pass the interval $I_l(j)$ is at least $(m\ell(\mathcal{P}) - m + 1)/2$. In contrast, each request must use at least one node in $I_l(j)$. Therefore the best way that the optimum load-balanced routing algorithm does, is to distribute the $(m\ell(\mathcal{P})-m+1)/2$ requests evenly on the nodes in $I_l(j)$. So we have that the optimum maximum load, $\ell^*(R)$, is at least $(m\ell(\mathcal{P}) - m + 1)/(2m)$. This proves that $\ell(\mathcal{P}) \leq 2\ell^*(R) + 1$. □

5.3.3 Requests with variable packet sizes

For requests with variable size, GREEDY2 can not guarantee a good load balancing ratio. For example, with variable sized packets, we can force GREEDY2 to alternate between two nodes while it is possible to distribute the loads among nearby nodes (See Appendix A). Hence, we use a different greedy strategy with stretch factor of 2 and the load-balancing ratio of 3. This idea will also be used for routing inside strips.

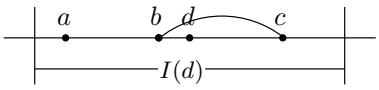


Figure 5.5. The bridge bc over d, b is to the left of d, c is to the right of d and b,c are visible to each other.

For each node $d \in S$, a pair of nodes b and c form a *bridge* over d if $b \in I_l(d)$, $c \in I_r(d)$, and b, c are visible to each other (Figure 5.5). The load of a bridge $\ell(bc)$ is defined as the maximum load $\max(\ell(b), \ell(c))$. The lightest bridge among all the bridges over d is denoted as $B(d)$. A bridge guarantees that we can make substantial progress within one hop. The algorithm GREEDY3 works as follows. Whenever a node a receives a request, say, from its left, we first check if the destination is within

a's communication range. If not, a asks the furthest node d in its right communication range and use the lightest bridge $B(d) = bc$ to route the request. The node c makes the next routing decision if the destination is not reached yet.

From the definition of a bridge, we know that a cannot see the node c. Therefore, for any four adjacent nodes a, b, c, d on the path produced by GREEDY3, a and d are not visible to each other since they must be separated by a bridge. Using the same technique as in the previous subsection, we can add one look-ahead to GREEDY3 to shortcut the path if two non-adjacent node can see each other in the path. Therefore, the stretch factor of GREEDY3 is 2. We now argue that GREEDY3 has a load balancing ratio of 3.

Theorem 5.3.3. *GREEDY3 is 3-balanced, i.e.* $\ell(\mathcal{P}) \leq 3\ell^*(R)$.

Proof: The proof is by induction. Denote by R_t the set of the first t requests. The claim is clearly true when $t = 1$. Suppose that after the t-th request is delivered, we have that $\ell(R_t) \leq 3\ell^*(R_t)$. We now argue that after we deliver the $t + 1$-th request, $\ell(R_{t+1}) \leq 3\ell^*(R_{t+1})$.

We prove this by contradiction. Suppose that $\ell(R_{t+1}) > 3\ell^*(R_{t+1})$. Consider the first time when the condition is violated when we route the $t + 1$-th request r_{t+1}. Suppose that it is when a receives r_{t+1} and routes it through bc, the lightest bridge over d. Let ℓ_{t+1} denote the packet size of request r_{t+1}. Then, $\ell_t(bc)+\ell_{t+1} > 3\ell^*(R_{t+1})$. For every bridge B over d, $\ell_t(B) \geq \ell_t(bc)$ since bc is the lightest bridge over d. A node u is *heavy* if there exists a bridge $B = uv$ or $B = vu$ over d such that $\ell_t(B) = \ell_t(u)$. Denote by D the set of heavy nodes in d's visible range $I(d)$. All the nodes in D have load at least $\ell_t(bc)$, since bc is the lightest bridge. Assuming that $|D| = m$, the total load on D for GREEDY3 is at least $\sum_{u \in D} \ell(u) \geq m \cdot \ell_t(bc)$.

On the other hand, any routing algorithm that delivers a request has to use one bridge to jump over d. Each bridge passes at least one heavy node and at most two heavy nodes. So for GREEDY3, a request contributes at most twice its load to the total load on D. Since the total load generated by GREEDY3 is $m \cdot \ell_t(bc)$, the total packet size of the requests that were routed on at least one node in D is at least $m \cdot \ell_t(bc)/2$. On the other hand, for the optimal routing algorithm, these requests use

at least one heavy node. So the total load on D for the optimal routing algorithm is at least $m \cdot \ell_t(bc)/2$. Therefore, the best that the optimal routing algorithm can do is to distribute the requests as evenly as possible on D. Therefore the maximum load of the nodes in D produced by the optimal algorithm is at least $\ell_t(bc)/2$. That is, $\ell^*(R_t) \geq \ell_t(bc)/2 > (3\ell^*(R_{t+1}) - \ell_{t+1})/2$. Since $\ell_{t+1} \leq \ell^*(R_{t+1})$, the above formula implies that $\ell^*(R_t) > \ell^*(R_{t+1})$, i.e., the maximum load of the optimal load balanced routing algorithm decreases after r_{t+1} is routed. This can not happen. Thus the claim $\ell(R_{t+1}) \leq 3\ell^*(R_{t+1})$ holds for $t+1$. □

5.4 Distributed implementation

In this section, we present an efficient implementation of the above algorithms. We assume that each node knows its location by either GPS or some localization methods [76, 137, 138, 157]. We also assume that the rough location of the destination is known such that the source node knows whether it should send the packet to its left or right. Denote by $h_1(p)$ the number of 1-hop neighbors and by $h_2(p)$ the number of 2-hop neighbors. Our implementation has the following properties:

- A wireless node makes the routing decision by using only local information.
- Each node only stores $O(\log h_1(p))$ bytes.
- A node p makes the routing decision in $O(\log h_1(p))$ ($O(\log^2 h_2(p))$) time, for the case of unit (variable) packet size.
- Any dynamic update, including changing load on p, adding or deleting a node p, takes $O(\log h_1(p))$ time.

5.4.1 Requests with unit packet sizes

When all the requests have the same size, each node p needs to compute p^*, the furthest node in p's right (or left) communication range whose load is not maximum over all the nodes in $I_r(p)$. In the following part of this subsection, we focus on how

to find p^* by a memory-efficient mechanism. Once p^* is found, the packet is delivered to p^*. This process is repeated until the destination is reached.

First, if we build a balanced binary search tree on all the nodes in $I_r(p)$, we can clearly compute p^* in time $O(\log |I_r(p)|)$. By this simple implementation p stores $O(|I_r(p)|)$ bits of information. Here we propose a more efficient implementation which actually distributes the storage and computation to each node instead of using a central node. Then each node only needs poly-logarithmic storage. To achieve this, we pay some price for extra communication. A node p finds out the next hop p^* by asking its neighbors to do some computation. We assume that in the procedure of finding the next hop p^*, the size of the control information transferred is very small and thus can be omitted. If this is not the case, i.e., the control information is also taken into account in loading the wireless nodes, we should use the first scheme where a node keeps the locations of all its 1-hop neighbors.

We construct a virtual forest $\mathcal{F}$. Imagine a bottom-up binary grouping process where every two adjacent and mutually visible nodes are grouped. Pick one of two nodes as a (first-level) leader. We then group adjacent and visible first-level leaders to create second-level leaders and so on. If a node cannot find a same level leader within its communication range to group, the process simply stops at that node (Figure 5.6). At the end of the process, each node has a rank which is the highest level it is on.

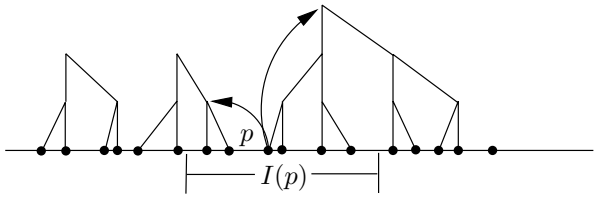


Figure 5.6. A binary forest.

The virtual forest $\mathcal{F}$ is stored distributedly on the nodes in S. If a node u is grouped with a node v on level $i-1$ and u is selected as the level i leader, we call the node v the child of u. The virtual forest F is stored implicitly such that each node stores the ID's of its parent and children. Now, we consider the communication range $I(p)$ of a node p. There are at most three trees in $\mathcal{F}$ that contain nodes in $I(p)$, which are denoted by a set $\mathcal{T}(p)$. We store at p the set $V(p)$ that contains the highest level

node inside $I(p)$, for each tree $T \in \mathcal{T}(p)$. For a node u in a tree $T \in \mathcal{F}$, we also store the maximum load of its subtree. The storage at each node is $O(\log h_1)$ in total.

To compute p^* for p, p asks the nodes in $V(p)$ which node should be p^*. Each of the node in $V(p)$ does a binary search top-down and recursively asks its children to compute p^*. The next hop p^*, once found, is returned to the node p and the packet is delivered from p to p^*. Since each tree in $\mathcal{F}$ is balanced, the computation and any dynamic change of load can be done in time $O(\log h_1)$ as well.

5.4.2 Requests with variable packet sizes

The algorithm for requests with variable packet size is more complicated. The major task is to find the lightest bridge over a node p. For each node p, define a function $f_p(x) = \min_{x_p \le x_u \le x_p + x} \ell(u)$ and $g_p(x) = \min_{x_p + x - 1 \le x_u \le x_p} \ell(u)$. Clearly, both $f_p(x), g_p(x)$ are stair-case functions, where f is decreasing, and g is increasing. Since $f_p(0) = g_p(1) = \ell(p)$, there must exist an x^* so that $f_p(x)$ and $g_p(x)$ intersect. We take b^* and c^* to be the nodes such that $f_p(x^*) = \ell(c^*)$ and $g_p(x^*) = \ell(b^*)$, see Figure 5.7. Then we claim that,

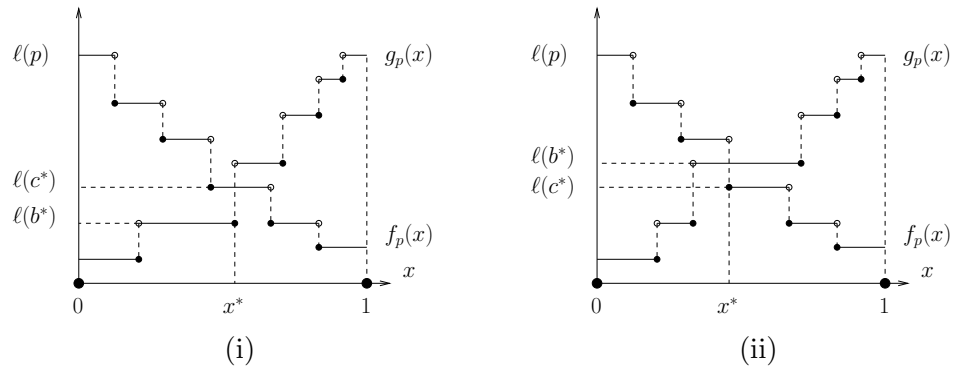


Figure 5.7. (i) $g_p(x^*) = \ell(b^*) \le f_p(x^*) = \ell(c^*)$; (ii) $g_p(x^*) = \ell(b^*) > f_p(x^*) = \ell(c^*)$.

Lemma 5.4.1. *b^*c^* is the lightest bridge over p.*

Proof: First since $g_p(x^*) = \ell(b^*)$, $f_p(x^*) = \ell(c^*)$, then $b^* \in [x_p + x^* - 1, x_p]$, $c^* \in [x_p, x_p + x^*]$. So b^*c^* is a valid bridge.

Now we assume that there is another bridge bc with weight smaller than b^*c^*. If $\ell(b^*) \leq \ell(c^*)$, as shown in Figure 5.7 (i), then we know that $\ell(c) < \ell(c^*)$. For the location of c we must have x_c is greater than $x_p + x^*$. Then b's location x_b is inside interval $[x_c - 1, x_p]$. So $b \geq x_c - 1 > x_p + x^* - 1$. The stair-case property of $f_p(x)$ and $g_p(x)$ implies that for all $x > x^*$, $g_p(x) > f_p(x^*) = \ell(c^*)$. So $\ell(b) > \ell(c^*)$. Then the weight of bc is greater than the weight of b^*c^*. This causes contradiction. The case when $\ell(b^*) > \ell(c^*)$ can be proved in a similar way. □

Computing x^* can be done by using binary search. Again, we can use the method for unit packet size to solve the problem but with one more log factor due to the binary search in the computation and update time.

5.4.3 Handling dynamic changes

The virtual forest needs to be updated when there are two types changes. One is when two nodes start or stop to become visible to each other due to either insertion/deletion of nodes or mobility. The other is when the load on a node changes. In both cases, event only a constant number of trees in the forest are affected. If we use a dynamic balanced binary tree, then each dynamic change can be done in $O(\log h_1(p))$ time.

5.5 Load-balanced routing for nodes in a narrow strip

The load-balanced routing algorithm can be extended to a strip with width $w \leq \sqrt{3}/2 \approx 0.86$, if the communication radius of each node is 1. In what follows, we assume that a strip is bounded by two horizontal parallel lines where the width is the vertical distance between the two lines. The restriction on the width will become clear later.

We say a node b is to the left (right) of a, if the x-coordinate of b is smaller (larger) than that of a. Then, for each vertex p, bc is a right (left) bridge if b is inside the communication range of p, c is outside and to the right (left) of p, and

b, c are visible to each other (Figure 5.8(i)). The load of a bridge is then defined as $\max(\ell(b), \ell(c))$. The right (left) lightest bridge is the bridge with the smallest load among all the right (left) bridges. The algorithm GREEDY4 works in a way similar to GREEDY3: when p receives a packet, it first checks if the destination of the packet inside its communication range. If it is, then the packet is forwarded to the destination. Otherwise, according to which side the destination lies, p chooses the right or left lightest bridge, say bc, sends the packet on the path pb and bc. When the packet arrives on b, it again checks if the destination is in b's communication range, and if it is, forwards the packet to the destination. Then, repeat the above process at the node c until the destination is reached.

To show that the above algorithm works correctly, we make the following observation, which also explains the restriction on the strip width.

Lemma 5.5.1. *Suppose that u, v, w, from left to right, are three nodes in a strip with width at most $\sqrt{3}/2$ and u, w are mutually visible. Then either u or w is visible to v.*

Proof: If neither u nor w is visible to v, then both nodes are outside the communication range of v. In addition, u is to the left of v, and w to the right. It is easy to see that if the width is at most $\sqrt{3}/2$, then u and w cannot be visible to each other (Figure 5.8(ii)). □

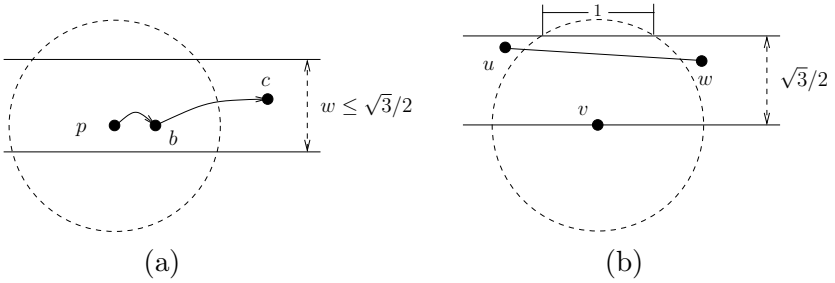


Figure 5.8. (i) bc is a right bridge of p. (ii) u, w cannot see each other if they are on different sides of v and outside of v's communication range.

We now claim that

Theorem 5.5.2. *The stretch factor of GREEDY4 is* 4*, and the load-balancing ratio is* 3*.*

Proof: We first show that the algorithm always succeeds. Suppose that there is a path from a node s to t where t is to the right of s. We argue that GREEDY4 will reach the destination t. Otherwise, assume that the algorithm is stuck at a node p, and p is not visible to t. If p is to the left of t, we argue that there must be a bridge that p can routes to. Since s is to the left of t, any path from s to t has to cross the right boundary of p's communication range. The edge that crosses the boundary then must be a right bridge of p, contradicting with the assumption (Figure 5.9 (i)).

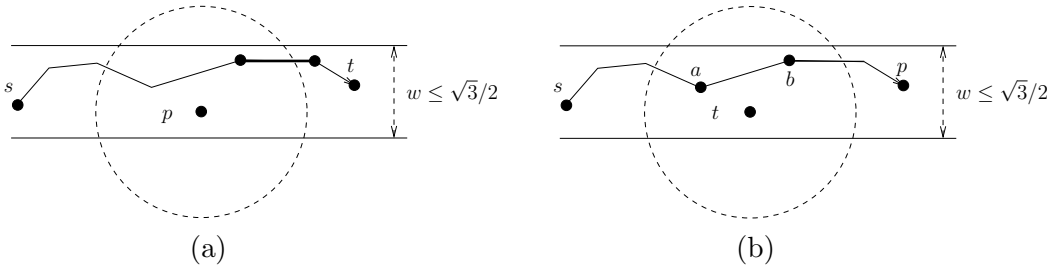


Figure 5.9. (i) The path from s to t has to cross the right boundary of p's communication range. The thickened edge is a right bridge of p. (ii) The path from s to p has an edge ab that sandwich t.

Now, suppose that p is to the right of t. Consider the pair of adjacent nodes, say a, b, on the path from s to p that sandwich t, i.e., a is to the left of t, b is to the right of t, see Figure 5.9 (ii). By Lemma 5.5.1, t is visible to either a or b. Thus, the packet must have been delivered to t by either a or b. Therefore, the algorithm always delivers a packet if there exists a path.

Next we will bound the stretch factor of the algorithm. For a right bridge bc of p, since c is outside the communication range of p, we have that $x_c \geq x_p + 1/2$, where x_c and x_p are the x-coordinates of c and p, respectively. By taking two hops, GREEDY4 is guaranteed to progress at least $1/2$ along the x-coordinate. Clearly, for a packet from s to t, it has to take at least $x_t - x_s$ hops, while GREEDY4 takes at most $2(x_t - x_s)/(1/2) = 4(x_t - x_s)$ hops. Thus, the stretch factor is at most 4.

To bound the load-balancing ratio we use the same techniques as in Theorem 5.3.3. For a node p that GREEDY4 picked, every path from the source to the destination has to use one right bridge of p, and therefore at least one and at most two heavy nodes. So by the same argument the load-balancing ratio for GREEDY4 is at most 3. □

Furthermore we show that the upper bound $\sqrt{3}/2$ on the width of the strip is indeed necessary. If otherwise, then there could be two paths P_1, P_2 from the source s such that they are not visible to each other except at the source and destination, see Figure 5.10. So any algorithm based on local information will not be able to find out at the source node whether P_1 or P_2 is more heavily loaded.

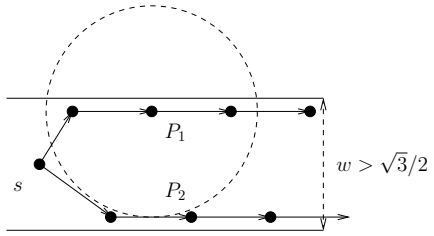


Figure 5.10. A strip of width $w > \sqrt{3}/2$, P_1, P_2 are two paths from s.

Clearly, the above algorithm can be implemented such that the cost for both routing decision and update is linear to the number of nodes in the 2-hop neighborhood. The reason that it does not admit an efficient algorithm as in the line case is that the two dimensional dynamic disk range search is much more difficult than the one dimensional case, which can be done by binary search. There are techniques in computational geometry which yield better theoretical bounds. But the number of nodes in a neighborhood is typically small, it is unnecessary to use those heavy machinery.

5.6 Simulations

We evaluate the load balanced routing algorithm under various wireless nodes distributions and traffic patterns. The shortest path routing algorithm minimizes the total energy the network consumed but does not balance the loads on different nodes. So

a node can be overloaded under the shortest path routing. We compare GREEDY3 with the shortest path routing algorithm.

In our experiments, 1000 wireless nodes are distributed randomly in the interval $[0, 100]$. In one set of experiments, we assume that the nodes can handle as many traffic as possible, and we evaluate the maximum size of packets that one node relays. In the other set, we put a limit on the maximum energy of a node and evaluate the number of packets the network delivers before any node dies.

For the traffic patterns, we consider two cases: random traffic pattern and aligned traffic pattern. In the random traffic pattern, we generate a packet by choosing the source and destination of a packet uniformly randomly among all the nodes. In the aligned traffic pattern, a packet is originated from a random node inside the communication range of the leftmost node and destined at a random node inside the communication range of the rightmost node. In both cases, our experimental results suggest that the load-balanced routing works much better than the shortest path routing in terms of balancing the load. In addition, we measure the length of the paths produced by our algorithm and compare it with the shortest path routing. From time to time, our algorithm produces path noticeably longer than the shortest path. However, on average, the path length is only slightly longer. This indicates that the load-balancing is achieved at the price of increasing the path length but only by a small fraction.

Unlimited energy and random traffic We generate 1000 random packets, each with size randomly chosen between 1 and 10. In Figure 5.11, we plot, for both the load balancing routing and the shortest path routing, the maximum load in terms of the total size of the packets delivered by any node, if the communication range has radius 5. According to the data, the ratio of the maximum load of the shortest path routing to that of the load-balanced routing is about 5. We also compare the length (the number of hops) of the paths produced by our algorithm to the shortest path length, both in the worst case and on average. Figure 5.12 shows the worst case and average ratio of the length of the paths produced by our algorithm to the shortest path length, as well as the ratio of the maximum load under shortest path routing

and load-balanced routing under different communication ranges. The observation from Figure 5.12 is that we achieve substantially in terms of load balancing by paying a modest price on the maximum and average delay.

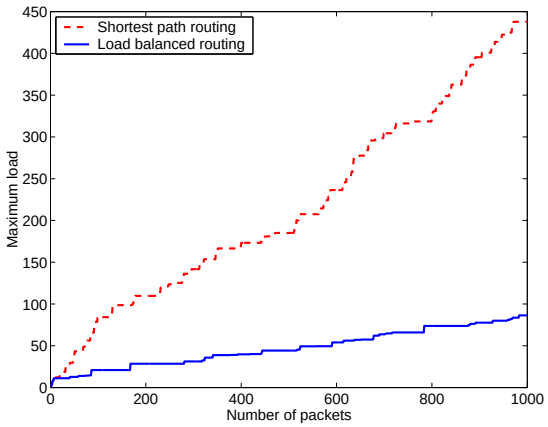


Figure 5.11. The maximum node load in terms of the number of packets delivered under the random traffic pattern. The dashed line curve is for the shortest path routing, and the solid line curve for the load-balanced routing. The communication range has radius 5.

Unlimited energy and aligned traffic Under the aligned traffic pattern, a packet is originated from a random node in the interval $[0, 10]$ and destined to a random node in $[90, 100]$. Each with size randomly chosen between 1 and 10. Figure 5.13 shows the experimental result. The data shows that the ratio of the maximum load of the shortest path routing to that of the load-balanced routing is about 10.3, much higher than the random traffic pattern, if the communication range has radius 5.

Limited energy and random traffic In this case we assume the nodes have a maximum energy, measured by the maximum total size of the packets it can relay. A node dies if the total size of the packets it relays exceed the given limit. We vary the maximum energy from 0 to 90. Correspondingly we show the number of packets delivered before any node dies in Figure 5.15. Compared to the shortest path routing, the load-balanced routing algorithm delivers about twice as many packets before any node dies for all the energy levels.

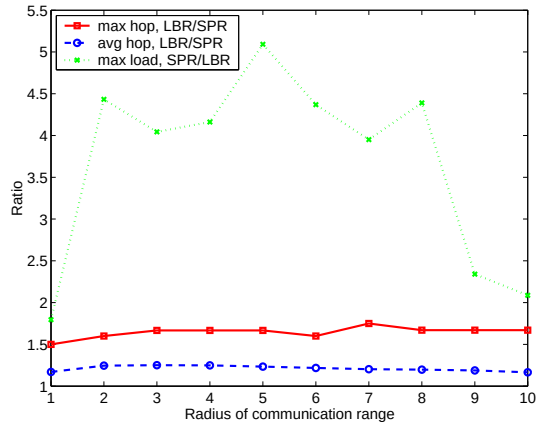


Figure 5.12. The worst case and average ratio of the length of the paths produced by our algorithm to the shortest path length under different communication ranges, under the random traffic pattern. We also show the ratio of the maximum load under the shortest path routing to that under the load-balanced routing for different communication ranges.

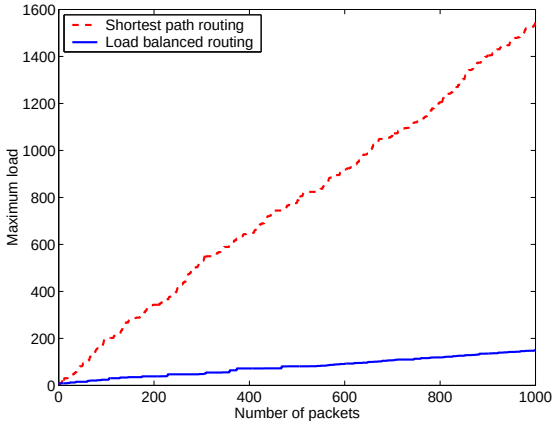


Figure 5.13. The maximum node load in terms of the number of packets delivered under the aligned traffic pattern. The communication range has radius 5.

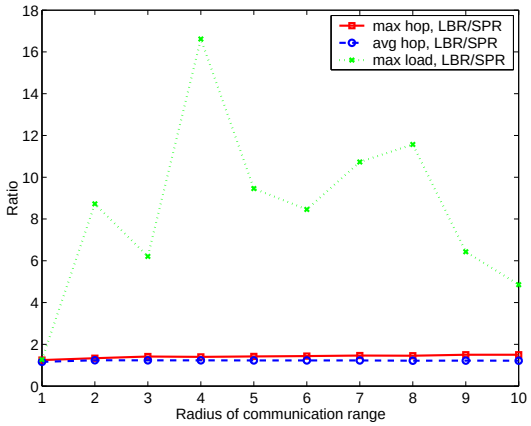


Figure 5.14. The worst case and average ratio of the length of the paths produced by our algorithm to the shortest path length under different communication ranges, under the aligned traffic pattern. We also show the ratio of the maximum load under the shortest path routing to that under the load-balanced routing for different communication ranges.

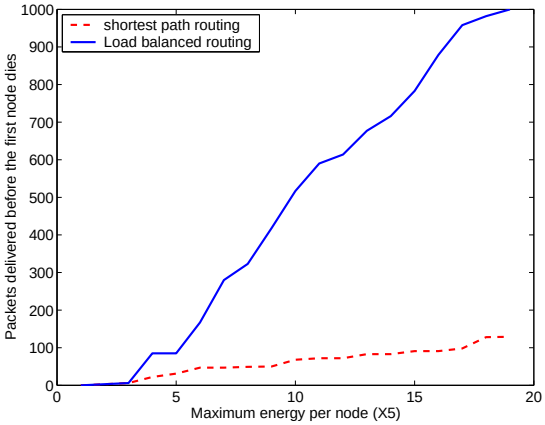


Figure 5.15. The number of packets delivered when the first node dies in terms of the maximum energy of each node under the random traffic pattern. Again, dashed line curve is the shortest path routing, and sold line curve the load-balanced routing.

Limited energy and aligned traffic We also try aligned traffic under the constrained energy model. We vary the maximum energy from 0 to 150. As shown in Figure 5.16, the result for load-balanced routing is better than the case for random traffic.

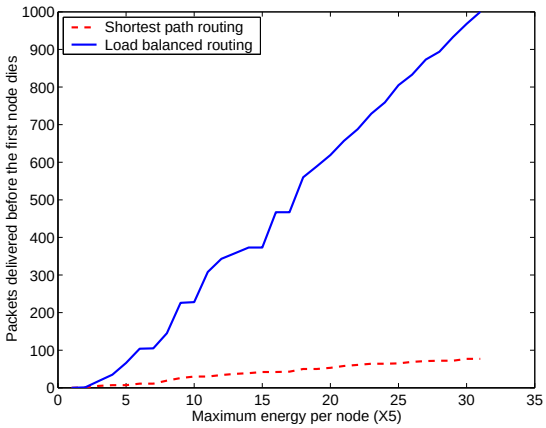


Figure 5.16. The number of packets delivered when the first node dies in terms of the maximum energy of each node under the aligned traffic pattern.

In summary, we find through simulation that our load-balanced routing algorithms perform consistently better than the shortest path routing in terms of the maximum node load and only slightly worse in terms of the path length. Furthermore, the implementation of the load-balanced routing is fairly simple. We would expect the algorithm to find practical use in real applications.

5.7 Extension

Firstly, the load balanced routing algorithms assume that all the nodes start with the same energy level. But the algorithm GREEDY3 and GREEDY4 can also be used in the case when the nodes start with different energy level. Under this setup, the algorithms try to prolong the network lifetime, defined as the first time when some node dies.

Secondly, we describe our algorithms in a per-packet basis. Its main purpose is to provide a rigorous analysis. In practice, the control overhead can be high if we try

balance load on a per-packet basis. Instead, we maintain for each node the level of its energy use and ran our algorithm on the levels. This way, a route can be cached, and a load update is only needed when the energy level of a node changes.

Chapter 6

Short Paths vs. Load Balancing

6.1 Introduction

In the previous Chapter, we showed that for a set of wireless nodes in a narrow strip, we can find a routing algorithm that achieves both a constant stretch factor and a constant load balancing ratio. We also showed that this is impossible when the nodes are distributed in the plane, for example, in the case as Figure 6.1. If we study this

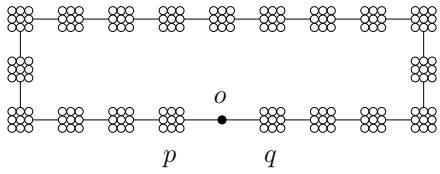


Figure 6.1. The packets from spot p to q either go through node o, thus causing o to be heavily loaded, or route along a long path, thus having large latency.

example carefully, we find that it requires highly crowded nodes. In practice, the wireless nodes usually are distributed nicely. In this chapter, we show that when the nodes are nicely distributed, there exists a better tradeoff between the stretch factor and the load balancing ratio of a routing algorithm.

For a point set, we define the maximum density as the maximum number of points covered by any unit disk in the plane, and the average density as the average number of points covered by the unit disks centered at the points in the set. We

show that there are non-trivial tradeoffs involving the density of the wireless nodes. For example, if the maximum density of a point set is ρ, then a c-short path routing can achieve a load balancing ratio of $O(\min(\sqrt{\rho n/c}, n/c))$. In particular, if we use shortest path routing, i.e., when $c = 1$, the load balancing ratio is $O(\sqrt{\rho n})$. When ρ is constant (this includes the graphs such as meshes), the bound is then $O(\sqrt{n})$. Furthermore, all those bounds are tight asymptotically in the worst case. When points are not evenly distributed, the average density is a more appropriate measure. We also obtain similar tradeoffs involving the average rather than the maximum density. It's natural to take density into account appears when analyzing wireless networks, even in a dense network. Techniques such as clustering are often used to reduce the routing complexity by reducing the problem on the original point set to the routing on a smaller set of "backbone" nodes. The "backbone" nodes usually have small density [58, 59].

6.2 Preliminaries

We follow the same definitions as in Chapter 5. Let S be a set of n points in the plane, the *density* $\rho(S)$ of S is defined to be the maximum number of points in S covered by any unit disk. For each $p \in S$, denote by $\rho(p)$ the number of points inside the unit disk centered at p. Define the *average density* $\bar{\rho}(S)$ of S to be $\sum_{p \in S} \rho(p)/n$. Clearly, $\bar{\rho}(S) \leq \rho(S)$. For a geometric shape B, we abuse the notation a little bit and also use B to represent the set of nodes inside. We can prove the following results which will be used later.

Lemma 1. *For any disk B with radius $r \geq 1$,*

1. $|B \cap S| = O(\rho(S)r^2)$;
2. $|B \cap S| = O(r\sqrt{n\bar{\rho}(S)})$.

Proof: Since B can be covered by $O(r^2)$ unit disks, we have $|B \cap S| = O(\rho(S)r^2)$. For the second claim, suppose that there are x points in $B \cap S$. We can partition $B \cap S$ into $O(r^2)$ disjoint subsets such that all the points in one subset are mutually visible[1].

[1]It's possible that two points in different subsets are visible.

Suppose that those sets are $S_1, \ldots, S_m$, and let $n_i = |S_i|$. Therefore, $\sum_i n_i^2 \leq n\bar{\rho}(S)$. By Cauchy-Schwartz inequality, we have that $x^2 = (\sum_i n_i)^2 \leq m(\sum_i n_i^2) \leq mn\bar{\rho}(S)$. Since $m = O(r^2)$, $x = O(r\sqrt{n\bar{\rho}(S)})$. □

The length $|P|$ of a path P in the graph $U(S)$ is the number of points on the path. For any two points $p, q \in S$, denote by $\tau(p, q)$ the number of hops in the shortest path between p and q. For any path P between p, q, the *stretch factor* $\omega(P)$ of P is defined to be $|P|/\tau(p, q)$. P is called *c-short* if $\omega(P) \leq c$. For a set of requests R, a set of paths $\mathcal{P}$ satisfy R, denoted $\mathcal{P} \models R$, if $\mathcal{P} = \{P_r \,|\, r \in R\}$ where P_r is a path between s_r and t_r. We define the stretch factor $\omega(\mathcal{P})$ of $\mathcal{P}$ to be $\max_{r\in R} \omega(P_r)$. A routing algorithm is called a *c-short-path* (or *c*-short) routing if it only uses paths with stretch factor at most c.

For a set of requests R and paths $\mathcal{P}$ that satisfy R, the *load* $\ell(v)$ incurred to v is the total size of the packets that pass v, i.e. $\ell(v) = \sum_{v\in P_r} \ell_r$. The *load* $\ell(\mathcal{P})$ *of* $\mathcal{P}$ is then defined to be $\max_{v\in S} \ell(v)$. Define $\ell^*(R) = \min_{\mathcal{P}\models R} \ell(\mathcal{P})$ to be the optimal load for satisfying R and $\ell_c^*(R) = \min_{\mathcal{P}\models R \wedge \omega(\mathcal{P})\leq c} \ell(\mathcal{P})$ the optimal load by any c-short routing algorithm. Specifically, we use $\ell_1(R)$ to represent the maximum load created by a shortest path routing algorithm[2]. For a routing algorithm $\mathcal{A}$, denote by $\mathcal{A}(R)$ the set of paths produced by $\mathcal{A}$ to satisfy R. Then $\mathcal{A}$'s approximation ratio (if $\mathcal{A}$ is off-line) or competitive ratio (if $\mathcal{A}$ is on-line) is defined to be $\max_R \ell(\mathcal{A}(R))/\ell^*(R)$. We generally call it a *load-balancing ratio*. In this chapter, our goal is to study the tradeoff between the stretch factor and the load-balancing ratio of routing algorithms for wireless networks.

6.3 Tradeoffs based on the maximum density

Our main result for the maximum density is as follows:

Theorem 2. *For any n nodes with the maximum density ρ and any set of requests R, $\ell_c^*(R)/\ell^*(R) = O(\min(\sqrt{\rho n/c}, n/c))$. This bound is tight in the worst case.*

[2]We assume that the shortest path routing generates unique routing paths, by perturbation.

As a special case of the above theorem, when the set of nodes has constant bounded density, then the load-balancing ratio of the optimal c-short path routing is bounded by $O(\sqrt{n/c})$. In another special case, $c = 1$, the load-balancing ratio for shortest path routing is $O(\sqrt{\rho n})$. So shortest path routing on nodes with constant density achieves a load balancing ratio of $O(\sqrt{n})$. We first prove the above theorem for the case of shortest path routing and extend the technique to prove Theorem 2.

Theorem 3. *For any n nodes with the maximum density ρ and any set of requests R, $\ell_1(R)/\ell^*(R) = O(\sqrt{\rho n})$.*

Proof: Suppose that p is the node with the maximum load if we use shortest path routing. Without loss of generality, we can assume that all the requests in R are routed through p by shortest path routing, because otherwise we can safely delete those requests that do not — this does not change the maximum load by shortest path routing but can only decrease the maximum load of the optimal routing algorithm. Suppose that the set of requests is $R = \{r_1, \ldots, r_m\}$ where $r_i = (s_i, t_i, \ell_i)$ is a request from s_i to t_i with packet size ℓ_i. We denote by ℓ^* the maximum load of the optimal load balanced routing algorithm $\ell^*(R)$. Since all the requests in R pass through p in shortest path routing scheme, the maximum load of shortest path routing, $\ell_1(R) = \ell = \sum_{i=1}^{m} \ell_i$. We now wish to upper-bound $\alpha = \ell/\ell^*$.

The intuition of the proof is that shortest path routing is optimal in the sense of the total loads it creates. If the load on p is high, the total load a shortest path routing creates is also necessarily high. This causes the optimal algorithm to create high total loads as well. The average load therefore cannot be too low, even if those loads can be evenly distributed. This intuition is made concrete by the following lemma.

We first give some notations. For each point $q \in S$, denote by $R(q)$ all the requests that originate at q and by $\ell(q)$ the total size of those packets, i.e. $\ell(q) = \sum_{r_i \in R(q)} \ell_i$. Write $\beta(q) = \ell(q)/\ell$, where $\ell = \sum_{i=1}^{m} \ell_i$. Clearly $\sum_q \beta(q) = 1$.

Lemma 4. *Suppose that D_τ is the disk with radius $\tau \geq 1$ centered at p, then $\sum_{q \in D_\tau} \beta(q) \leq c_0 \rho \tau/\alpha$, for some constant $c_0 > 0$.*

Proof: We partition D_τ into a set of $\log \tau$ disjoint sets B_k, $0 \leq k \leq \log \tau$, where B_0 is the unit disk centered at p and for $k \geq 1$, B_k is an annulus with an inner radius of 2^{k-1} and an outer radius of 2^k. See Figure 6.2. Consider the set R_k of the

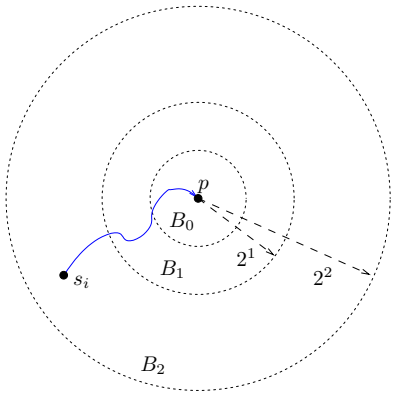


Figure 6.2. Division of D_τ into a set of annuli B_i. All the traffic pass through the center p by shortest path routing.

requests originating at some point in B_k and a request $r_j = (s_j, t_j, \ell_j) \in R_k$. Since the shortest path between s_j and t_j passes the point p, the length of the shortest path between s_j and t_j is at least the shortest path length between p and s_j, i.e., $\tau(s_j, t_j) \geq \tau(p, s_j) \geq |ps_j| \geq 2^{k-1}$. Now, suppose that P_j is the path from s_j to t_j produced by the optimal load-balanced routing algorithm. The number of points on P_j is at least 2^{k-1}. Let A_j be the first 2^{k-1} points on P_j. Denote by S_k the union of all the A_j, i.e., $S_k = \bigcup_{r_j \in R_k} A_j$. We study the total load produced by the optimal load balanced routing algorithm on the nodes inside S_k. Firstly we have

$$\sum_{v \in S_k} \ell(v) \geq \sum_{r_j \in R_k} \ell_j |A_j| = 2^{k-1} \sum_{r_j \in R_k} \ell_j \,. \tag{6.1}$$

On the other hand, for any point $a \in A_j$, $|pa| \leq |ps_j| + |as_j| \leq 2^k + 2^{k-1} = 3 \cdot 2^{k-1}$. That is, all the points in A_j are inside a disk with radius $3 \cdot 2^{k-1}$ centered at p. Since the nodes have maximum density ρ, $|S_k| = O(\rho(3 \cdot 2^{k-1})^2)$. Since each node has load at most $\ell^* = \ell/\alpha$, we have that

$$\sum_{v \in S_k} \ell(v) \leq |S_k| \ell^* \leq c_0 \rho (2^{k-1})^2 \ell/\alpha \,, \tag{6.2}$$

for some constant $c_0 > 0$. Combining (6.1) and (6.2), we have that

$$\sum_{r_j \in R_k} \ell_j \leq c_0 \rho 2^{k-1} \ell / \alpha \,.$$

Thus $\sum_{r_j \in R_k} \beta_j = \sum_{r_j \in R_k} \ell_j / \ell \leq c_0 \rho 2^{k-1} / \alpha$, for $1 \leq k \leq \log \tau$. For the unit disk B_0, we have that

$$\begin{aligned} \textstyle\sum_{q \in B_0} \beta(q) &= \textstyle\sum_{q \in B_0} \ell(q)/\ell \leq |B_0| \ell^* / \ell \\ &\leq \rho \ell^* / \ell = \rho / \alpha \,. \end{aligned}$$

By summing up over all the k's, we have that

$$\begin{aligned} \textstyle\sum_{q \in D_\tau} \beta(q) &= \textstyle\sum_{q \in B_0} \beta(q) + \sum_{k=1}^{\log \tau} \sum_{r_j \in R_k} \beta_j \\ &\leq \rho/\alpha + \textstyle\sum_{k=1}^{\log \tau} c_0 \rho 2^{k-1} / \alpha \\ &\leq c_0 \rho \tau / \alpha \,. \end{aligned}$$

□

Now we proceed to prove Theorem 3. We can assume that for any $q \in S$, $\beta(q) \leq 1/3$; otherwise $\ell^* \geq \ell(q) > \ell/3$, i.e. $\alpha < 3$. Now, consider the smallest disk D centered at p such that

$$\sum_{q \in D} \beta(q) \geq 1/2 \,.$$

We assume that there is only one node on the boundary of D — otherwise we can perturb (conceptually) the nodes so that the assumption is valid. Since $\beta(q) \leq 1/3$ for any q, we have that

$$\sum_{q \notin D} \beta(q) \geq 1/6 \,.$$

Let τ^* denote the radius of D. Then, by Lemma 4,

$$c_0 \rho \tau^* / \alpha \geq \sum_{q \in D} \beta(q) \geq 1/2$$

i.e.

$$\alpha \leq 2 c_0 \rho \tau^* \,. \tag{6.3}$$

On the other hand, for any point $q \notin D$, $|pq| \geq \tau^*$. By the same argument used in the proof of Lemma 4, for any algorithm, the loads incurred by those requests originating at q are at least $\ell(q)\tau^*$. Therefore, the total loads caused by such requests are at least

$$\sum_{q \notin D} \ell(q)\tau^* = \sum_{q \notin D} \beta(q)\ell\tau^* \geq \ell\tau^*/6\,.$$

The last inequality is due to that $\sum_{q \notin D} \beta(q) \geq 1/6$. Hence, the optimal load balancing routing algorithm can do no better than distributing these loads evenly on the n nodes. That is, $\ell^* \geq \ell\tau^*/6n$, i.e.

$$\alpha = \ell/\ell^* \leq 6n/\tau^*\,. \tag{6.4}$$

By combining (6.3) and (6.4), we have that

$$\alpha \leq \min(2c_0\rho\tau^*, 6n/\tau^*) \leq c_1\sqrt{\rho n}\,,$$

for $c_1 = \sqrt{12c_0}$. This proves Theorem 3. □

Now, we extend the result to c-short routing.

Proof of Theorem 2. We show that, for any set of requests R, we can construct a set of c-short paths that achieve the claimed upper bound. Consider the optimal routing that minimizes the maximum load. We divide R into two subsets R_1 and R_2, where R_1 contains the requests that are routed by c-short paths in the optimal algorithm, and R_2 contains those requests routed by non-c-short paths. We construct a set of paths $\mathcal{P}$ as follows. We include in $\mathcal{P}$ the paths that the optimum algorithm produced for requests in R_1. For each request in R_2, we add to $\mathcal{P}$ (any) shortest path between the source and the destination of that request. Clearly, all the paths in $\mathcal{P}$ are c-short. We now show that the maximum load caused by $\mathcal{P}$ is at most $O(\min(\sqrt{\rho n/c}, n/c)\ell^*(R))$.

For each point $q \in S$, denote by $\ell_1^*(q), \ell_2^*(q)$, the loads on q caused by, respectively, routing R_1 and R_2 by the optimal algorithm. Let $\ell_1^* = \max_q \ell_1^*(q)$ and $\ell_2^* = \max_q \ell_2^*(q)$. Clearly, $\ell^* \geq \max(\ell_1^*, \ell_2^*) \geq (\ell_1^* + \ell_2^*)/2$. For each point $q \in S$, denote by $\ell_2(q)$ the loads on q caused by routing R_2 by using shortest path routing. Let $\ell_2(R) = \max_q \ell_2(q)$. Clearly, $\ell_c^*(R) \leq \ell(\mathcal{P}) \leq \ell_1^* + \ell_2(R)$.

We now bound $\ell_2(R)/\ell_2^*$ by using almost the same argument as in the proof of Theorem 3. The only difference is that all the paths used to route requests in R_2 by the optimal algorithm are not c-short. Therefore, all the requests originating at nodes outside the disk D generate a total load of $\sum_{q \notin D} \ell(q) \cdot c\tau^*$, which is equal or greater than $\ell c\tau^*/6$. Then we can replace (6.4) with the following inequality

$$\ell_2(R)/\ell_2^* \leq 6n/(c\tau^*)\,.$$

Since (6.3) is still valid, we have that

$$\begin{aligned} \ell_2(R)/\ell_2^* &= \min(2c_0\rho\tau^*, 6n/(c\tau^*)) \\ &= O(\min(\sqrt{\rho n/c}, n/c))\,. \end{aligned}$$

Therefore,

$$\begin{aligned} \ell_c^*(R) &\leq \ell_1^* + \ell_2(R) = O(\min(\sqrt{\rho n/c}, n/c))(\ell_1^* + \ell_2^*) \\ &= O(\min(\sqrt{\rho n/c}, n/c)) \cdot \ell^*\,. \end{aligned}$$

This proves the upper bound in Theorem 2.

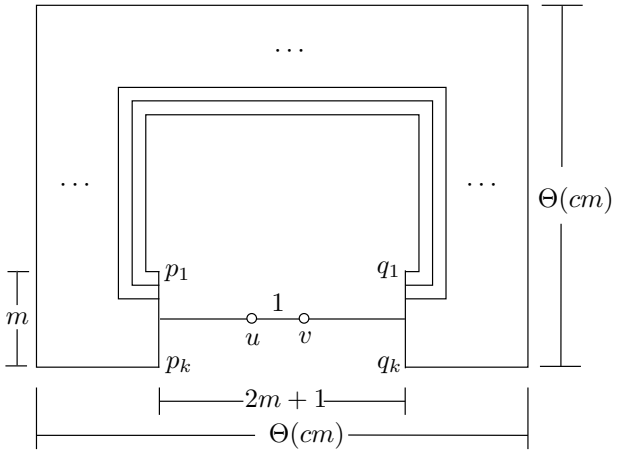


Figure 6.3. Lower bound of the load-balancing ratio for the optimal c-short routing with maximum density ρ.

In the following, we show a lower bound construction. We only describe the lower bound construction for $\rho c \leq n$, i.e. $\sqrt{\rho n/c} \leq n/c$. The other case is similar. Consider the example illustrated in Figure 6.3. The distance between u, v is 1. Take

a parameter $m > 1$ which will be determined later, we place $k = \rho m$ points $p_1, \ldots, p_k$ on a vertical line segment with length m and distance m away from u. Similarly, we create $q_1, \ldots, q_k$ with respect to v. On the horizontal line segment through u, v, we place about $2m$ points evenly. In addition, there is a path between every pair of p_i and q_i as drawn in Figure 6.3. Each path is about $4cm$ long and has $4cm$ points on it. Clearly, the maximum density of the point set is $O(\rho)$. The shortest path between p_i and q_i goes through u, v and has length at most $3m$. On the other hand, any other path connecting u, v has to go through the outside loop with length $4cm$. So all the c-short paths connecting p_i, q_i have to pass u and v. Therefore, if we request to send a unit packet from p_i to q_i, for $1 \leq i \leq k$, then the c-short path routing causes load $k = \rho m$ on u, v. On the other hand, we can use the outer path to route each packet, creating load 1 on each point. Thus, the load-balancing ratio of any c-short path routing of this example is $\Omega(\rho m)$. The total number of points in the example is about $\Theta((\rho m) \cdot (cm)) = \Theta(\rho c m^2)$. Setting $m = \sqrt{n/(c\rho)}$, we obtain the desired lower bound. □

We should emphasize that in the proof of Theorem 3, we do not restrict which shortest path to use when there are more than one shortest paths. That is, the bound holds no matter which shortest paths are used when there exist multiple shortest paths. However, the proof of Theorem 2 does use a set of c-short paths produced by the optimal algorithm. Therefore, the bound does not hold for arbitrary c-short paths. Actually, if we choose bad c-short paths, we may end up with a bound even worse than that of the shortest path routing. In section 6.6, we will present an algorithm to discover a set of c-short paths with the maximum load within $O(\log n)$ factor of the optimum load using c-short paths.

6.4 Tradeoffs based on average density

We now show the tradeoff based on the average density of the point set. The benefit of considering average density is clear — it is applicable to a wider family of point sets, in particular to the point sets with uneven distribution.

Theorem 5. *Given a set of n nodes S in the plane with average density $\bar{\rho}$, for any set of requests R,*

$$\ell_1(R)/\ell^* = O(\min(\sqrt{\bar{\rho}n}\log n, n)) .$$

In addition, there exists example such that

$$\ell_c^*(R)/\ell^* = \Omega(\sqrt{\bar{\rho}n/\max(1,\log c)}) .$$

Proof: The proof for the upper bound is similar to the proof of Theorem 3. We use the notation in the proof of Lemma 4. We take τ to be the diameter of the communication graph. $\tau = O(n)$. So D_τ, a disk with radius τ centered at p, covers all the n nodes. Then we partition D_τ into a set of $\log\tau$ disjoint sets B_k, $0 \le k \le \log\tau$, where B_0 is the unit disk centered at p and for $k \ge 1$, B_k is an annulus with an inner radius of 2^{k-1} and an outer radius of 2^k. The only difference is that with the average density, by Lemma 1, we can only bound $|\cup_{r_j\in R_k} A_j| = O(\sqrt{\bar{\rho}n}2^k)$, for $1 \le k \le \log\tau$. Since each node has load at most ℓ^*, we have that

$$2^{k-1}\sum_{r_j\in R_k}\ell_j \le c_0\sqrt{\bar{\rho}n}2^{k-1}\ell^* ,$$

for some constant $c_0 > 0$. Thus $\sum_{r_j\in R_k}\ell_j \le c_0\sqrt{\bar{\rho}n}\ell^*$, for $1 \le k \le \log\tau$. We also know that $\sum_{r_j\in R_0}\ell_j \le \bar{\rho}\ell^* \le \sqrt{\bar{\rho}n}\ell^*$, since $\bar{\rho} \le n$. By summing up for all the k's, we have that

$$\ell_1(R) = \ell = \sum_{k=0}^{\log\tau}\sum_{r_j\in R_k}\ell_j \le c_1\sqrt{\bar{\rho}n}\ell^*\log n ,$$

for some constant c_1.

As for the lower bound, consider the example shown in Figure 6.4. In the figure, the distance between u, v is 1. There are c vertical bars with length $1, 2, \ldots, c$ and with distance $0.5, 1.0, 1.5, \ldots$ away from u. We place k nodes on each of the line segments evenly with k determined later. Symmetrically, we place nodes with respect to the node v. Label those nodes on the outside bars $p_1, \ldots, p_k$ and $q_1, \ldots, q_k$, respectively, and those nodes on the bar closest to u, v to be $u_1, \ldots, u_k$, and $v_1, \ldots, v_k$, respectively. Again, we place nodes to connect every pair p_i, q_i as shown in the figure. The length

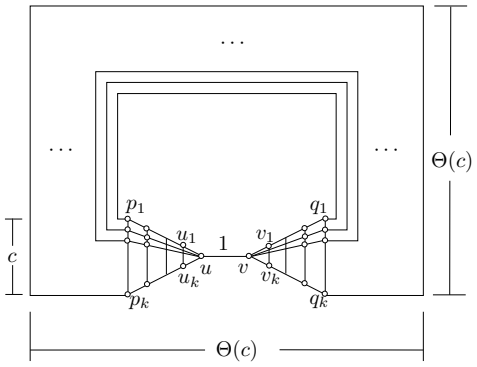


Figure 6.4. Lower bound of the load-balancing ratio for the optimal c-short routing with average density $\bar{\rho}$.

of those paths is $\Theta(c)$. Now, we request to send a packet from p_i to q_i, for $1 \leq i \leq k$. Again, each c-short path routing has to use the path through the nodes u, v, causing a load of k on u, v. On the other hand, the optimal algorithm can route the requests through the outside paths and create only load 1 to each node. Thus, the load-balancing ratio of any c-short routing algorithm is $\Omega(k)$. The total number of nodes in the figure is bounded by $O(c \cdot k)$. To bound the average density of the point set, we consider two types of points. For a point x on a vertical bar with length h, the number of points it sees is about $\Theta(k/h)$. Thus,

$$\sum_x \rho(x) = \Theta(\sum_{h=1}^{c} k^2/h) = \Theta(\max(1, \log c) \cdot k^2)\,.$$

For a point y on the outside path, $\rho(y) = \Theta(k/c)$. Therefore, the average density $\bar{\rho}$ is

$$\Theta((\max(1, \log c) \cdot k^2 + ck(k/c))/n) = \Theta(\max(1, \log c) \cdot k^2/n)\,.$$

That is, $k = \Theta(\sqrt{\bar{\rho}n/\max(1, \log c)})$, and the load-balancing ratio is $\Omega(\sqrt{\bar{\rho}n/\max(1, \log c)})$. □

6.5 Extensions

The above results naturally extend to other routing problems and to a larger family of growth-restricted graphs.

6.5.1 VLSI routing

In VLSI routing, the task is to connect some given pairs of nodes by paths on a mesh. One important goal is to reduce the line width, i.e. the maximum number of paths that pass the same edge. Such a problem has been studied extensively [132, 159, 101, 23]. A mesh can be realized as a unit disk graph of a set of points with constant bounded density. Thus, we have the following extension of our result to bound the line width in VLSI routing.

Corollary 6. *If we are restricted to use c-short paths to route wires in a mesh, then the line width is within $O(\sqrt{n/c})$ factor of the optimum solution. In particular, if we use (any) shortest paths, the approximation factor is $O(\sqrt{n})$.*

6.5.2 Unit ball graphs in higher dimensions

Similar results hold for unit ball graphs in higher dimensions. The definitions in Section 6.2 extend naturally to points in higher dimensions. We can apply the same technique to obtain the following.

Theorem 7. *For n point in $\mathbb{R}^k$ with maximum density ρ, the load-balancing ratio of the optimal c-short routing is $O((n/c)^{1-1/k}\rho^{1/k})$. In particular, the load-balancing ratio of (any) shortest path routing is $O(n^{1-1/k}\rho^{1/k})$.*

6.5.3 Growth restricted graphs

If we examine the proof of Theorem 2, we can see that the only property we needed for the proof is that there are $O(\rho r^2)$ nodes inside any disk with radius r. Thus, the result extends immediately to graphs with small growth rate. Recall that a graph has *density* ρ and *growth rate* k (or growth dimension k) if for any vertex v and any $r > 1$, $|B_r(v)| \leq \rho r^k$, where $B_r(v) = \{u|\tau(u,v) \leq r\}$, the ball with radius r centered at v. By using exactly the same argument, we have that

Theorem 8. *For a graph with density ρ and growth rate k, the load-balancing ratio of the optimal c-short routing is $O((n/c)^{1-1/k}\rho^{1/k})$. In particular, the load-balancing*

ratio of (any) shortest path routing for a graph with constant density and growth rate k is $O(n^{1-1/k})$.

6.6 An algorithm for short path load balancing routing

In the previous section, we showed a combinatorial bound on the load balancing ratio for the optimal c-short routing algorithm. However, it is NP-hard to compute the set of c-short paths (actually even the shortest paths) that minimizes the maximum load. Here, we describe an algorithm that computes c-short paths with maximum load within an $O(\log n)$ factor of the optimum.

One general approach for approximation is by the randomized rounding technique [132, 131]. But that technique cannot be directly applied to our case because of the restriction on the stretch factor — it will make the size of the linear programming problem exponentially large. But we show that the on-line virtual circuit routing algorithm by Aspnes *et al.* [21] applies to our problem to obtain an $O(\log n)$ approximation ratio.

Theorem 9. *There is a polynomial time on-line c-short routing algorithm with load balancing competitive ratio $O(\log n)$ when compared to the optimal off-line c-short routing algorithm. The competitive ratio is tight in the worst case.*

Proof: We apply the method in [21] with slight modification. In the algorithm in [21], a weight is assigned to each edge (or vertex in our case) according to the current load on the edge and the size of the request. Then for any new request, the lightest path[3] with respect to this weighting function is used to satisfy the request. Similarly, for c-short routing, we use the lightest path among all the c-short paths. We just need to show that this modification can be done in polynomial time, and it does find us an $O(\log n)$ approximation.

To see the former, we can use dynamic programming: given a pair of nodes (s, t), we iteratively compute, for every node u in the graph, the lightest path from s to u

[3] we call it the lightest path, to be distinguished from the shortest path in the graph.

with length exactly L (this may include non-simple paths) for $L = 1, 2, \ldots, c \cdot \tau(s,t)$, where $\tau(s,t)$ denotes the shortest distance between s, t. This will give us the lightest c-short path connecting s and t in polynomial time.

The proof of the $O(\log n)$ competitive ratio follows from the argument in the proof of Theorem 5.2 in [21]. By a close examination of that proof, we can see that it still holds even if we associate each request r with a subset of paths P_r such that only a path in P_r can be used to satisfy r. Therefore, restricting all the paths to be c-short is just a special case.

The lower bound construction in [25] can be used to show that even in a mesh, any on-line c-short routing algorithm is $\Omega(\log n)$ competitive compared to the optimal c-short routing algorithm. Actually, we can show a stronger result where any on-line algorithm is $\Omega(\log n)$ competitive even when compared against the optimal off-line algorithm that only uses the shortest paths. The details can be found in Appendix B. □

6.7 Load-balancing ratio of routing on spanners

In wireless network, one important method to reduce the complexity of routing is to construct a sparse spanner graph and route on the spanner graph [87, 59, 106]. A sub-graph G of a unit-disk graph $I(S)$ is a c-spanner if the shortest path between any two points in G is c-short compared with $I(S)$. Since a spanner graph has fewer edges than the unit-disk graph, the load balancing ratio on a spanner graph might be high. The following theorem provides a worst case tight bound.

Theorem 10. *Suppose S is a set of n points in the plane with density ρ, and G is a c-spanner of $I(S)$, for any requests R, $\ell^*_G(R)/\ell^* = O(\rho c^2)$, where $\ell^*_G(R)$ (ℓ^*) is the maximum load resulted by the optimal load-balancing routing algorithm on G ($I(S)$). The bound is tight in the worst case.*

Proof: For a set of requests R, consider the optimal solution $\mathcal{P}^*$ on the unit-disk graph $I(S)$. We now construct a routing scheme on G from $\mathcal{P}^*$. For an edge uv on a path in $\mathcal{P}^*$, if it is not in G, then there must exist a path P with length c in G

because G is a c-spanner. We can then reroute the packet on that path P. Clearly, this way we obtain a set of paths $\mathcal{P}'$ in G that satisfy R. Now, consider a point $p \in S$. A packet can be rerouted to it only if it is routed in the optimal solution through a point u which is at most distance c away from p. Or, u is in the disk with radius c and centered at p. There are $O(\rho c^2)$ such points. Therefore, the load on p is $O(\rho c^2 \ell^*)$.

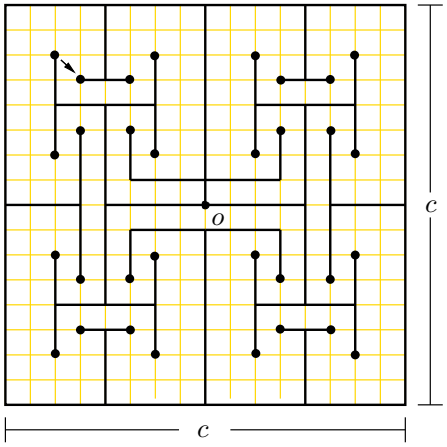


Figure 6.5. Lower bound $\Omega(c^2)$ on the competitive ratio on c-spanners.

As for the lower bound, we use the classic H-tree construction [113]. We only show the construction for constant density. The extension to arbitrary density is easy – we just put ρ copies on each node. Consider $\Theta(c^2)$ points positioned on a grid shown in Figure 6.5. Each little square of the grid has side length $1/2$. The spanner G is composed of an H-tree and a "complement" skeleton joined by a single edge at the center of the grid o. So any path from a node on the H-tree to a node in the complement H-tree has to go through o. Clearly, G is a $\Theta(c)$-spanner graph. Now we make a request from each leaf point of the H-tree to its nearby point on the complement part of the H-tree, as shown by the little arrow in Figure 6.5. The optimal solution can send the requests directly. However, in G, all the requests have to be routed through the point o. Therefore, the load-balancing ratio of the routing on this c-spanner is $\Omega(c^2)$. □

Part IV

Clustering the Pairwise Distances

Chapter 7

Well-Separated Pair Decomposition

7.1 Introduction

The notion of a well-separated pair decomposition was initially introduced by Callahan and Kosaraju [40] for points in Euclidean space. The notion of a well-separated pair decomposition can be extended to general graph metrics. For a metric space (S, π), two non-empty subsets $S_1, S_2 \subseteq S$ are called *c-well-separated* if $\pi(S_1, S_2) \geq c \cdot \max(D_\pi(S_1), D_\pi(S_2))$, for example in Figure 7.1. For any two sets A and B, a set

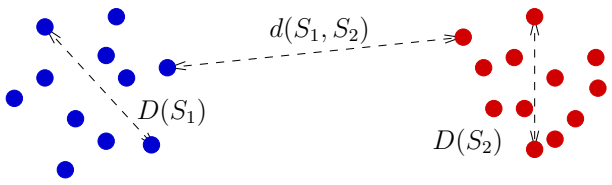


Figure 7.1. An example of a c-well-separated pair (S_1, S_2).

of pairs $\mathcal{P} = \{P_1, P_2, \cdots, P_m\}$, where $P_i = (A_i, B_i)$, is called a *pair decomposition* of (A, B) (or of A if $A = B$) if

- for all the i's, $A_i \subseteq A$, and $B_i \subseteq B$;
- $A_i \cap B_i = \emptyset$;

- for any two elements $a \in A$ and $b \in B$, there exists a unique i such that $a \in A_i$, and $b \in B_i$. We say (a, b) is *covered* by the pair (A_i, B_i).

If in addition, every pair in $\mathcal{P}$ is c-well-separated, $\mathcal{P}$ is called a *c-well-separated pair decomposition* (or c-WSPD in short). Clearly, any metric space admits a c-WSPD with quadratic size by using the trivial family that contains all the pairwise elements.

Well separated pair decompositions have found numerous applications in solving proximity problems for points in Euclidean space [38, 40, 39, 19, 15, 116, 103, 68, 54]. In [40], Callahan and Kosaraju showed that for any point set in an Euclidean space and for any constant $c \geq 1$, there always exists a c-well-separated pair decomposition with linearly many pairs. This fact has been very useful in obtaining nearly linear time algorithms for many problems such as computing k-nearest neighbors, N-body potential fields, geometric spanners and approximate minimum spanning trees. Well-separated pair decomposition is also shown to be very useful in obtaining efficient dynamic, parallel, and external memory algorithms [38, 39, 40, 37, 67].

Curiously enough however, there has been no work for an extension to more general metric spaces. One reason is probably that a general metric space may not admit a well-separated pair decomposition with a sub-quadratic size. Indeed, even for the metric induced by a star tree with unit weight on each edge, any well-separated pair decomposition requires quadratically many pairs. This makes the well-separated pair decomposition useless for such a metric. In this Chapter, we will show that for unit-disk graph metric, there do exist well-separated pair decompositions with almost linear size, and therefore many proximity problems under that metric can be solved efficiently.

We call the well-separated pair decomposition in the Euclidean space the *geometric well separated pair decomposition*, to be distinguished from the decomposition in graph metrics. For the metric induced by the unit-disk graph on n points and for any constant $c \geq 1$, there exists a c-well-separated pair decomposition with $O(n \log n)$ pairs, and such a decomposition can be computed in $O(n \log n)$ time. We also show that the bounds can be extended to higher dimensions: for $k \geq 3$, there always exists a c-well-separated pair decomposition with size $O(n^{2-2/k})$ for the unit-ball graph metric on n points, and the bound is tight in the worst case. The construction time

is $O(n^{4/3} \operatorname{polylog} n)$ for $k = 3$ and $O(n^{2-2/k})$ for $k \geq 4$.

The difficulty in obtaining a well-separated pair decomposition for unit-disk graph metric is that two points that are close in the space are not necessarily close under the graph metric. We first prove the bound for the point set with constant-bounded density, i.e. a point set where any unit disk covers only a constant number of points in the set, by using a packing argument similar to the one in [70]. For a point set with unbounded density, we apply the clustering technique similar to the one used in [59] to the point set and obtain a set of "clusterheads" with a bounded density. We then apply the result for bounded density point set on those clusterheads and show that the bound holds for any point sets.

For a pair of well-separated sets, the distance between any two points from different sets can be approximated by the "distance" between the two sets or the distance between any pair of points in different sets. In other words, a well-separated pair decomposition can be thought as a compressed representation to approximate the $\Theta(n^2)$ pairwise distances. Many problems that require to check the pairwise shortest distances can therefore be approximately solved by examining those distances between the well-separated pairs of sets. When the size of the well-separated pair decomposition is sub-quadratic, it often gives us more efficient algorithms than examining all pairs distances. Indeed, this is the intuition behind many applications of the geometric well-separated pair decomposition. By using the same intuition, we show the application of well-separated pair decomposition in several proximity problems under the unit-disk graph metric. Specifically, we consider the following natural proximity problems. Assume that $S_1 \subseteq S$.

- **Furthest neighbor, diameter, center.** The furthest neighbor of $p \in S_1$ is the point in S_1 with the maximum distance to p. Related problems include computing the *diameter*, the maximum pairwise shortest distance for points in S_1, and the *center*, the point that minimizes the maximum distance to all the other points.

- **Nearest neighbor, closest pair.** The nearest neighbor of $p \in S_1$ is the point in S_1 with the minimum distance to p. Related problems include computing

the *closest pair*, the pair with minimum shortest distance, and the *bichromatic closest pair*, the pair that minimizes distance between points from two different sets.

- **Median.** The median of S is the point in S that minimizes the average (or total) distance to all the other points.

- **Stretch factor.** For a graph G defined on S, its stretch factor with respect to the unit-disk graph metric is defined to be the maximum ratio $\pi_G(p,q)/\pi(p,q)$ where π_G, π are the distances induced by G and by the unit-disk graph, respectively.

- **k-center.** For a metric (S,π), the k-center is a set $K \subseteq S$, $|K| = k$, such that $\max_{p\in S} \min_{q\in K} \pi(p,q)$ is minimized.

- **Minimum spanning/Steiner tree.** For a metric space (S,π), the minimum spanning tree on a subset $S' \subseteq S$ is a tree on S' such that the total weight of the edges in the tree is minimized. If points other than S' can be used, called Steiner points, the minimum weight tree is called the minimum Steiner tree.

All the above problems can be solved or approximated efficiently for points in the Euclidean space. However, for the metric induced by a graph, even for planar graphs, very little is known other than solving the expensive all-pairs shortest path problem. By using the powerful tool of the well-separated pair decomposition, we are able to obtain, for all the above problems, nearly linear time algorithms for computing 2.42-approximation[1] and $O(n\sqrt{n\log n}/\varepsilon^3)$ time algorithms for computing $(1+\varepsilon)$-approximation for any $\varepsilon > 0$. In addition, the well-separated pair decomposition can be used to obtain an $O(n\log n/\varepsilon^4)$ space distance oracle so that any $(1+\varepsilon)$ distance query in the unit-disk graph can be answered in $O(1)$ time.

[1]For a minimization problem, a quantity $\hat{\ell}$ is a c-approximation of ℓ if $\ell \le \hat{\ell} \le c\ell$. An object $\hat{O}$ is a c-approximation of O with respect to a cost function f if $f(O) \le f(\hat{O}) \le cf(O)$. For a maximization problem, $\hat{\ell}$ is a c-approximation of ℓ if $\ell/c \le \hat{\ell} \le \ell$, and $\hat{O}$ is a c-approximation of O if $f(O)/c \le f(\hat{O}) \le f(O)$.

While the existence of an almost linear size well-separated pair decomposition has reduced the number of pairs we needed to examine, we still need good approximation of the distances between those pairs. Our construction algorithm produces well separated pair decompositions without knowing an accurate approximation of the distances. For approximation algorithms, we need accurate estimation of shortest distances between $O(n \log n)$ pairs of points in the unit-disk graph. Indeed, the approximation ratio and the running time of our algorithms are dominated by the efficiency of such algorithms. Once the distance estimation has been made, the rest of computation only takes almost linear time.

For a general graph, it is unknown whether $O(n \log n)$ pairs shortest path distances can be computed significantly faster than all pairs shortest path distances. For the planar graph, one can compute $O(n \log n)$ pairs shortest path distance in $O(n\sqrt{n \log n})$ time by using graph separators with $O(\sqrt{n})$ size [14]. This method extends to the unit-disk graph with constant bounded density since such graphs enjoy similar separator property as the planar graphs [115, 142]. As for approximation, Thorup [150] recently discovered an algorithm for planar graphs that can answer any $(1+\varepsilon)$-shortest distance query in $O(1/\varepsilon)$ time after almost linear time preprocessing. Unfortunately, Thorup's algorithm uses balanced shortest-path separators in planar graphs which do not obviously extend to the unit-disk graphs. On the other hand, it is known that there does exist a planar 2.42-spanner for a unit-disk graph [106]. By applying Thorup's algorithm to that planar spanner, we can compute the 2.42-approximate shortest path distance for $O(n \log n)$ pairs in almost linear time.

Another application of a well-separated pair decomposition is that we are able to obtain an almost linear size data structure to answer $(1 + \varepsilon)$-approximate shortest path query in $O(1)$ time. Approximate distance oracles have been studied where the emphasis is often on the size of the oracles (for a survey, see [163]). For general graphs, it has been shown that it is possible to construct a $(2k - 1)$-approximate distance oracle with size $O(kn^{1+1/k})$ [151]. It is also shown in [151] that this bound is tight for some small k's and is conjectured to be tight for all the k's. For planar graphs, Thorup [150] and Klein [91] have shown that there exists $(1+\varepsilon)$-approximate distance oracle by using almost linear space for any $\varepsilon > 0$. As mentioned before, their results

do not extend to the unit-disk graph. In addition, the query time of their algorithm is $O(1/\varepsilon)$. Recently, Gudmundsson *et al.* showed that when a geometric graph is an Euclidean spanner, there does exist an almost linear time (and therefore almost linear space) method to construct a $(1+\varepsilon)$-approximate and $O(1)$ query time distance oracles [68]. Again, a unit-disk graph is not necessarily an Euclidean spanner with bounded stretch factor, and their technique does not extend.

7.2 WSPD for unit-disk graph metric

We start with a point set with constant bounded density. Then, by combining with a geometric well-separated pair decomposition, we show the extension of the result to arbitrary point sets. We will focus our discussion on points in the plane, but most results extend to higher dimensions, resulting sub-quadratic size well-separated pair decomposition. We also show that our bounds in $\mathbb{R}^k$ for $k \geq 3$ are tight.

7.2.1 Point sets with constant bounded density

The density ρ of a point set S is defined to be the maximum number of points in S covered by a unit disk. S has constant bounded density if its density is $O(1)$. We assume that the unit-disk graph on S is connected; otherwise, we can consider each connected component separately.

To construct a well-separated pair decomposition, we first compute the unit-disk graph $I(S)$ of S and then a spanning tree T of $I(S)$ where the maximum degree of T is 6. This can be done by computing the relative neighborhood graph of S [154] and keeping those edges with length at most 1. Let G be the resulted graph. It can be shown that G is connected, and the degree of G is at most 6 (The details are in Appendix C.). We then compute a spanning tree of G. This step takes $O(n \log n)$ time [148]. It is also known that any n-vertex tree with maximum degree $\beta - 1$ can be divided into two parts by removing a single edge so that each subtree contains at least n/β vertices[2]. We now recursively apply the balanced partitioning to obtain a

[2]We take a root r of the tree and consider the children of r, one of them must have a subtree

balanced hierarchical decomposition of T (see Figure 7.2). The decomposition can

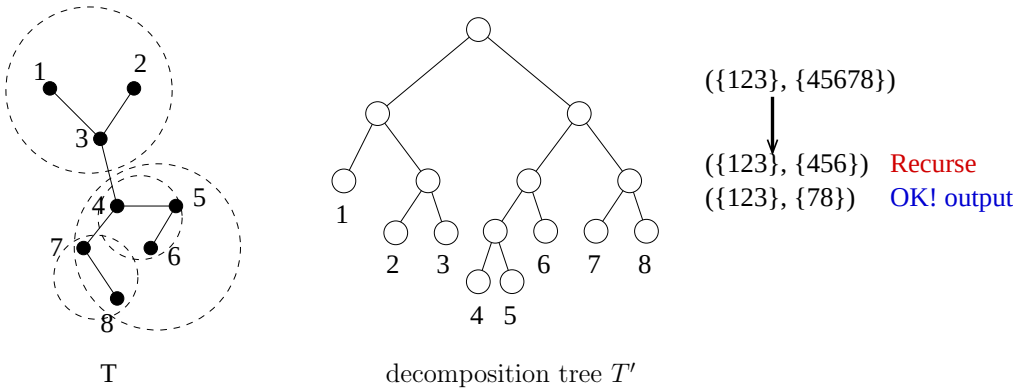


Figure 7.2. An example of the decomposition tree.

be represented as a rooted binary tree T' where each node $v \in T'$ corresponds to a (connected) subtree $T(v)$ of T. The root of T' corresponds to T, and for a node $v \in T'$, v's two children v_1, v_2 represent the two connected subtrees $T(v_1)$ and $T(v_2)$ obtained by removing an edge from $T(v)$. We denote by $S(v)$ the set of points in the subtree in $T(v)$. For a node $v \in T'$, denote by $P(v)$ the parent node of v in T'. We also use $P(S(u))$ to denote $S(P(u))$. Since the decomposition of T is balanced, the height of the tree T' is obviously $O(\log n)$.

Now, we describe a procedure to produce a c-WSPD of S. For each node $v \in T'$, we pick an arbitrary point from $S(v)$ as a representative of $S(v)$ and denote it by $\sigma(S(v))$ (or $\sigma(v)$). We place in a queue the pair $(S(r), S(r))$ where r is the root of T'. We run the following process until the queue becomes empty: repeatedly remove the first element $(S(v_1), S(v_2))$ from the queue. There are two cases:

- $|\sigma(v_1)\sigma(v_2)| \geq (c+2) \cdot \max(|S(v_1)|-1, |S(v_2)|-1)$. In this case, we include the pair to $\mathcal{P}$.

- $|\sigma(v_1)\sigma(v_2)| < (c+2) \cdot \max(|S(v_1)|-1, |S(v_2)|-1)$. If $|S(v_1)| = |S(v_2)| = 1$, then it must be the case that $S(v_1)$ and $S(v_2)$ contain the same point. In this case, we simply discard the pair. Otherwise, suppose that $|S(v_1)| \geq |S(v_2)|$ and

with at least $(n-1)/(\beta-1) \geq n/\beta$ nodes.

that u_1, u_2 are two children of v_1. We add to the queue two pairs $(S(u_1), S(v_2))$ and $(S(u_2), S(v_2))$.

The above process is very similar to the collision detection algorithm in [70] except that here a pair is produced when they are c-well-separated. We now make the following claims.

Lemma 7.2.1. *$\mathcal{P}$ is a c-WSPD of S. Furthermore, each ordered pair of distinct points (p, q) is covered by exactly one pair in $\mathcal{P}$.*

Proof: By the construction, a pair $(S(v_1), S(v_2))$ is included in $\mathcal{P}$ only if $|\sigma(v_1)\sigma(v_2)| \geq (c+2) \cdot \max(|S(v_1)| - 1, |S(v_2)| - 1)$. Since for any $v \in T'$, $S(v)$ is connected, $D_\pi(S(v)) \leq |S(v)| - 1$. In addition, $\pi(p, q) \geq |pq|$. Thus, we have that

$$\begin{aligned}
& \pi(S(v_1), S(v_2)) \\
\geq\ & \pi(\sigma(v_1), \sigma(v_2)) - (D_\pi(S(v_1)) + D_\pi(S(v_2)) \\
\geq\ & |\sigma(v_1)\sigma(v_2)| - 2\max(|S(v_1)| - 1, |S(v_2)| - 1) \\
\geq\ & c \cdot \max(|S(v_1)| - 1, |S(v_2) - 1|) \\
\geq\ & c \cdot \max(D_\pi(S(v_1)), D_\pi(S(v_2)))\,.
\end{aligned}$$

That is, every pair in $\mathcal{P}$ is a c-well-separated pair. To argue that $\mathcal{P}$ covers all the pairs of distinct points, we observe that we begin with the pair $(S(r), S(r))$ that covers all the pairs, and each time when we split a node, the union of the pairs covered remains the same. The pairs we discard are of the form $(\{p\}, \{p\})$. Thus, all the ordered pairs of distinct points are covered by $\mathcal{P}$. Since the splitting produces two disjoint sets, each ordered pair is covered exactly once. □

The following lemma shows that the sizes of two sets in the same pair do not differ too much.

Lemma 7.2.2. *Each pair (A, B) that ever appears in the queue satisfies $1/\beta \leq |A|/|B| \leq \beta$.*

Proof: The proof is done by induction. Clearly, it is true for the pair $(S(r), S(r))$. Now, consider the splitting that generates the pair (A, B). Without loss of generality,

assume that we split $P(B)$, the parent node of B. By the splitting rule, we have that $|A| \leq |P(B)|$. By induction hypothesis, $|A| \geq |P(B)|/\beta \geq |B|/\beta$. Since the splitting is balanced, $|B| \geq |P(B)|/\beta \geq |A|/\beta$. Therefore $1/\beta \leq |A|/|B| \leq \beta$. □

Now, we bound the size of $\mathcal{P}$.

Lemma 7.2.3. *If $(A, B_i) \in \mathcal{P}$, $i = 1, \cdots m(A)$, then $B_i \cap B_j = \emptyset$, and $m(A) = O(c^2|A|)$.*

Proof: By Lemma 7.2.1, each pair of points can only be covered once, thus $B_i \cap B_j = \emptyset$ if both (A, B_i) and (A, B_j) are in $\mathcal{P}$.

If $(A, B_i) \in \mathcal{P}$, then $\big(P(A), P(B_i)\big)$ is not in $\mathcal{P}$. So we have that $|\sigma(P(A))\sigma(P(B_i))| < (c+2) \cdot \max(|P(A)| - 1, |P(B_i)| - 1)$. Set $R = \beta|P(A)| \leq \beta^2|A|$. If we split $P(B_i)$ to get the pair (A, B_i), then $(A, P(B_i))$ appeared in the queue, by Lemma 7.2.2, we have $|P(B_i)| \leq \beta|A| \leq \beta|P(A)| = R$. If we split $P(A)$ to get the pair (A, B_i), then $|B_i| \leq |P(A)|$, so $|P(B_i)| \leq \beta|B_i| \leq \beta|P(A)| = R$. Then,

$$|\sigma(P(A))\sigma(P(B_i))| < (c+2)R, D_\pi(P(B_i)) \leq R\,.$$

Then all the points in B_i must be inside a disk of radius $(c+3)R$ centered at $\sigma(P(A))$. Therefore we have that $|\cup_{i=1}^{m(A)} B_i| = O\big((c+3)^2R^2\big)$ because S has constant bounded density. By Lemma 7.2.2, we know that $|B_i| \geq |A|/\beta \geq |P(A)|/\beta^2$. Thus, $|B_i| \geq R/\beta^3$. Then, we have that $m(A) = O((c+3)^2R^2/(R/\beta^3)) = O(c^2R) = O(c^2|A|)$. □

Lemma 7.2.4. $|\mathcal{P}| = O(c^2 n \log n)$.

Proof: Define $V_i = \{v \in T' \,|\, 2^i \leq |S(v)| < 2^{i+1}\}$, for $0 \leq i \leq \log n$. Clearly, $|V_i| = O(n/2^i)$. Define $\Phi_i = \{(S(v), B) \in \mathcal{P} \,|\, v \in V_i\}$. Denote by $m(S(v))$ the total number of pairs in which $S(v)$ is involved. By Lemma 7.2.3, we have that

$$\begin{aligned} |\Phi_i| &= \textstyle\sum_{v \in V_i} m\big(S(v)\big) = \sum_{v \in V_i} O(c^2|S(v)|) \\ &= O(c^2 2^{i+1} \cdot n/2^i) = O(c^2 n)\,. \end{aligned}$$

Thus, $|\mathcal{P}| = \sum_{i=0}^{\log n} |\Phi_i| = O(c^2 n \log n)$. □

Combining the above result, we now have that

Theorem 11. *For any n points with constant-bounded density in the plane and any $c \geq 1$, there exists a c-WSPD with $O(c^2 n \log n)$ pairs, which can be computed in $O(c^2 n \log n)$ time.*

Proof: Clearly, the time needed is proportional to the number of pairs that ever appear in the queue. We can represent the construction as a tree: each pair corresponds to a node in the tree, and when a pair is split, we treat those two resulting pairs as the children of the pair. Clearly, the leave of the tree correspond to those pairs included in $\mathcal{P}$ and the pairs discarded. All the discarded pairs have the form $(\{p\}, \{p\})$, and there are $O(n)$ such pairs. Thus, the total number of nodes in the tree is bounded by $O(|\mathcal{P}|) = O(c^2 n \log n)$. Each split costs $O(1)$. Therefore, the total computation cost is $O(c^2 n \log n)$. □

The result can be easily extended to the point set with maximum density ρ.

Corollary 7.2.5. *For a point set with maximum density ρ, for any $c \geq 1$, a c-WSPD with $O(\rho c^2 n \log n)$ pairs can be constructed in $O(\rho c^2 n \log n)$ time.*

Proof: If the point set has maximum density ρ, Lemma 7.2.3 still holds if we change $m(A)$ to $O(\rho c^2 |A|)$. Plug it in Lemma 7.2.4, we have that $|\mathcal{P}| = O(\rho c^2 n \log n)$. The claim then follows from Theorem 11. □

By a similar argument, we can extend the result to higher dimensions.

Theorem 7.2.6. *Given a point set in $\mathbb{R}^k$, where $k \geq 3$, with constant bounded density and any constant $c \geq 1$, there exist a c-WSPD with $O(n^{2-\frac{2}{k}})$ pairs for the unit-ball graph metric. This bound is tight in the worst case. And the decomposition can be computed in $O(n^{2-\frac{2}{k}})$ time.*

Proof: We first compute a spanning tree of S with constant maximum degree β_k, a constant dependent on k only. This can be done by using the technique in [15]. We then follow the same process as we described above. Lemma 7.2.3 can be changed so

that the number of pairs associated with a node A is $m(A) = O(|A|^{k-1})$. In addition, by Lemma 7.2.2, for any pair $(A, B) \in P$, $1/\beta_k \leq |A|/|B| \leq \beta_k$. Thus, $m(A) = O(n/|A|)$. Define V_i as in Lemma 7.2.4, $|V_i| = O(n/2^i)$. When $0 \leq i \leq \frac{1}{k}\log n$, $|\Phi_i| = \sum_{v\in V_i} m\big(S(v)\big) = O(\sum_{v\in V_i} |S(v)|^{k-1}) = O(2^{i(k-1)} \cdot n/2^i) = O(n2^{i(k-2)})$. When $i > \frac{1}{k}\log n$, $|\Phi_i| = \sum_{v\in V_i} m\big(S(v)\big) = O(\sum_{v\in V_i} n/|S(v)|) = O\big((n/2^i)^2\big)$. Therefore,

$$\begin{aligned} |\mathcal{P}| &= \sum_{0\leq i\leq \frac{1}{k}\log n} n2^{i(k-2)} + \sum_{\frac{1}{k}\log n < i \leq \log n} O(n^2/2^{2i}) \\ &= O(n^{2-2/k})\,. \end{aligned}$$

As for the lower bound, consider the points on the k-dimensional grid $[0, n^{1/k}) \times \ldots \times [0, n^{1/k})$. Define a graph G with edges between the pairs of points $(x_1, \ldots, x_i, \ldots, x_k)$ and $(x_1, \ldots, x_i+1, \ldots, x_k)$ for $i = 1$, or $x_1 = 0$ and $i \geq 2$. A point $(n^{1/k}-1, x_2, \ldots, x_k)$ for $0 \leq x_i < n^{1/k}$, $i \geq 2$, is called a *tip point*. Intuitively, G can be thought as a graph where the tip points dangle down from a $k-1$ dimensional mesh, see Figure 7.3. Clearly, we can perturb the point set so that its unit-ball graph has the same

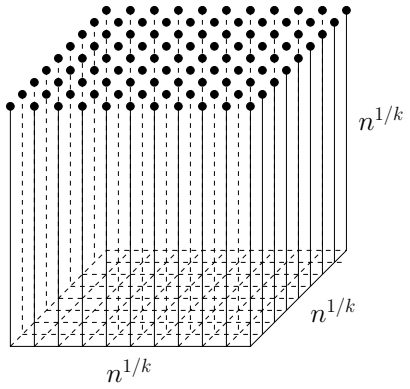


Figure 7.3. A lower bound example of the well separated pair decomposition for points in high dimensions.

topology as G. The metric defined by G has the following property: (i) the diameter of G is $kn^{1/k}$; (ii) the distance between any two tip points is at least $2n^{1/k}$. Therefore, when $c > k/2$, the diameters of the sets in a c-well-separated pair must be less than $2n^{1/k}$. This says that a c-WSPD cannot have two tip points in the same set of a pair. Since there are $\Theta(n^{1-1/k})$ tip points, we need $\Omega(n^{2-\frac{2}{k}})$ pairs only to separate those

tip points.

By the same argument as in Theorem 11, it is easily seen that the c-WSPD can be computed in $O(n^{2-2/k})$ time. □

7.2.2 Arbitrary point sets

The packing argument fails for the unit-disk graph of point sets with unbounded density. However, we can reduce the problem to the constant density case. We first cluster the points and then consider those crowded points separately by using geometric well-separated pair decompositions.

For $0 \leq \delta \leq 1$, a point p is δ-covered (or simply covered) by a point s if $|sp| \leq \delta$. Denote by $U(s)$ the set of points δ-covered by s. A subset $X \subseteq S$ is called a δ-cover of S if any point in S is δ-covered by some point in X. We call the points in a δ-cover X *clusterheads*. For each point in S, we assign it to the nearest clusterhead. Thus X induces a partitioning of S into sets $C(s) = S \cap \mathrm{Vor}(s)$, where $\mathrm{Vor}(s)$ denotes the Voronoi cell of s in X. We also call the points in $C(s)$ the *clients* of s. Clearly, for any $p \in C(s)$, $|sp| \leq \delta$, i.e. $C(s) \subseteq U(s)$. A δ-cover is called *minimal* if no two points in X are within distance δ to each other. For any set $A \subseteq X$, denote by $\hat{A}$ the set $\hat{A} = \cup_{s \in A} C(s)$.

To deal with an arbitrary point set S, we first compute a minimal cover X of S with an appropriately chosen δ. We then apply our results on constant-bounded density point sets to X. Note that we can not use the unit-disk graph on X because it may not have the same connectivity as the unit-disk graph on S. We have to use gateway nodes to connect the clusterheads as in chapter 4. For any two points s_1, s_2 in X, they are *neighbors* if $|s_1 s_2| > 1$, and there exist two points $p_1 \in C(S_1)$, and $p_2 \in C(s_2)$ such that $|p_1 p_2| \leq 1$. We call the pair (p_1, p_2) a pair of *gateways* between s_1 and s_2. For each neighboring pair, we only pick one pair of gateways arbitrarily. Let Y denote the set of all gateways points. Consider the point set $Z = X \cup Y$. Let π' denote the unit-disk graph metric on the set Z. Now, we claim that

Lemma 7.2.7. *X has $O(1/\delta^2)$-density. Z can be computed in $O(n \log n/\delta^2)$ time.*

Proof: For any two points s_1, s_2 in a minimal cover X, their distance is at least δ.

Therefore, there are $O(1/\delta^2)$ points of X inside any unit disk. So X has $O(1/\delta^2)$ density.

To compute X, we can use a greedy algorithm with the assistance of a dynamic point location data structure of unit disks [46]. The algorithm runs in $O(n \log n)$ time. To compute all the neighboring pairs, we can enumerate all the pairs (s_1, s_2) where s_2 is inside the square centered at s_1 and with side-length $2(1 + 2\delta)$. There are $O(n/\delta^2)$ such pairs according to Lemma 7.2.7 and can be computed in $O(n \log n/\delta^2)$ time by using a standard rectangular range searching data structure. Call such pairs *candidate pairs*. Clearly, only a candidate pair can possibly be a neighboring pair. To find a pair of gateways between two clusterheads s_1, s_2 of a candidate pair, we can compute the bichromatic closest pair between their clients $C(s_1), C(s_2)$. This can be done in $O(|U(s_1) \cup U(s_2)| \log n)$ time [47]. Since we only need to examine each clusterhead against $O(1/\delta^2)$ clusterheads, the total computation time is bounded by $O(n \log n/\delta^2)$. □

Now, we show that π' approximates π well on the set X.

Lemma 7.2.8. *For any two points $p, q \in X$,*

$$\pi(p, q) \leq \pi'(p, q) \leq (1 + 12\delta)\pi(p, q) + 12\delta\,.$$

Proof: Since $Z \subseteq S$, $\pi(p, q) \leq \pi'(p, q)$. On the other hand, assume that $p_0 p_1 \cdots p_m$, where $p_0 = p$ and $p_m = q$, is the shortest path between p and q in the unit-disk graph of S. For $0 \leq i \leq m$, suppose that s_i is the clusterhead that covers p_i. Note that $s_0 = p$ and $s_m = q$ since $p, q \in X$.

Consider two consecutive points p_i, p_{i+1}. If $s_i = s_{i+1}$, then we have that $|p_i p_{i+1}| \leq 2\delta$. Otherwise, suppose that $s_i \neq s_{i+1}$. If $|s_i s_{i+1}| \leq 1$, then $\pi'(s_i, s_{i+1}) = |s_i s_{i+1}| \leq |p_i p_{i+1}| + 2\delta$. If $|s_i s_{i+1}| > 1$, then s_i, s_{i+1} must be a neighboring pair since $|p_i p_{i+1}| \leq 1$. In this case, we can verify that $\pi'(s_i, s_{i+1}) \leq |p_i p_{i+1}| + 6\delta$. Thus,

$$\begin{aligned} \pi'(p, q) &\leq \textstyle\sum_{i=0}^{m-1} \pi'(s_i, s_{i+1}) \\ &\leq \textstyle\sum_{i=0}^{m-1} |p_i p_{i+1}| + 6m\delta \leq \pi(p, q) + 6m\delta\,. \end{aligned}$$

Since $p_0p_1\cdots p_m$ is the shortest path, $|p_ip_{i+2}| \geq 1$ for any $0 \leq i \leq m-2$ because otherwise the path could be shortened due to triangular inequality. Therefore, $\pi(p,q) \geq \lfloor m/2 \rfloor > m/2 - 1$, i.e. $m < 2(\pi(p,q)+1)$. Thus we have that $\pi'(p,q) \leq (1+12\delta)\pi(p,q) + 12\delta$. □

Before we describe the construction of c-WSPD for S, we need a straight-forward extension of geometric well-separated pair decomposition in [40] to two separable point sets.

Lemma 12. *Suppose that A and B are two point sets that can be separated by a line and have n points in total. For any constant $c \geq 1$, there exists a geometric c-well-separated pair decomposition of (A,B) with $O(n)$ pairs.*

Proof: This can be done by modifying the algorithm in [40] so that the first split of the point set of $A \cup B$ is by the line that separates A and B. □

Now, we describe a process that produces a c-WSPD of S for any $c \geq 1$. Set $\delta = 1/(2c+4)$, and $c' = 9(c+14)$. We first construct a minimal δ-cover X and the set Z as described above. Next we compute a c'-well-separated pair decomposition of the clusterheads X in the unit-disk graph metric of point set Z. Specifically, we give weight 1 to points in X and 0 to gateways. We find the spanning tree T of the unit-disk graph $I(Z)$. T has total weight $|X|$. We then recursively find balanced weighted decomposition of T: by removing an edge, each subtree has weight at least $1/\beta$ times the weight of the original tree. Since X has a bounded density $O(1/\delta^2)$, the packing argument is still valid and we can compute a c'-well-separated pair decomposition for X. Suppose the decomposition obtained is $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ where $P_i = (A_i, B_i)$, $A_i \subseteq X$, $B_i \subseteq X$. We now create a set of pairs $\mathcal{P}' = \mathcal{P}'_1 \cup \mathcal{P}'_2 \cup \mathcal{P}'_3$ as follows.

1. For each $P_i \in \mathcal{P}$, if $|A_i| > 1$ or $|B_i| > 1$, we include in $\mathcal{P}'_1$ the pair $P'_i = (\hat{A}_i, \hat{B}_i)$. Recall that $\hat{A} = \cup_{s\in A} C(s)$.

2. If $|A_i| = |B_i| = 1$, suppose that $A_i = \{a\}$ and $B_i = \{b\}$. If $|ab| \geq (2c+2)\delta$, we then include in $\mathcal{P}'_1$ the pair $P'_i = (\hat{A}_i, \hat{B}_i)$. Otherwise, any pair of points in $\hat{A}_i \bigcup \hat{B}_i$ are within distance $(2c+2)\delta + 2\delta = 1$. Since $\hat{A}_i \subset \mathrm{Vor}(a)$, and

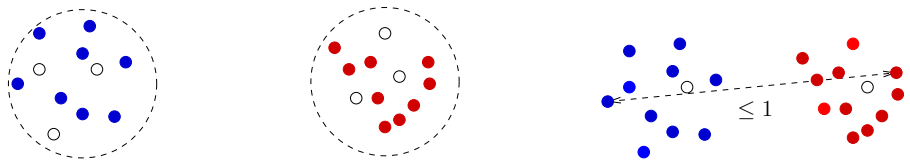


Figure 7.4. (i) For far-away pairs, we add the clients to the well-separated pairs of their clusterheads; (ii) For close-by pairs that every pair is within distance 1, we use a geometric well-separated pair decomposition.

$\hat{B}_i \subset \mathrm{Vor}(b)$, $\hat{A}_i$ and $\hat{B}_i$ are separable by a line. By Lemma 12, we can compute a geometric c-WSPD of $(\hat{A}_i, \hat{B}_i)$ and include into $\mathcal{P}'_2$ all the pairs produced this way.

3. For every $s \in X$, we compute a geometric c-WSPD of $C(s)$ and include into $\mathcal{P}'_3$ all the pairs produced.

Now, we claim that

Lemma 7.2.9. *$\mathcal{P}'$ is a c-WSPD of S.*

Proof: We first argue that $\mathcal{P}'$ is a pair decomposition of S. For any pair of points $s_1, s_2 \in S$, suppose that the clusterheads covering them are s'_1 and s'_2, respectively. If $s'_1 \neq s'_2$, then (s_1, s_2) is covered by a pair in $\mathcal{P}'_1 \cup \mathcal{P}'_2$. Otherwise, it is covered by a pair in $\mathcal{P}'_3$. It is also easily verified that each ordered pair is covered exactly once.

Now, we show that all the pairs in $\mathcal{P}'$ are c-well-separated with respect to the unit-disk graph metric. Since $\delta = 1/(2c+4)$, for all the pairs in $\mathcal{P}'_2$, the Euclidean distance between any two points in $\hat{A}_i \cup \hat{B}_i$ is at most $(2c+4)\delta = 1$. Therefore, the unit-disk graph on the subset $\hat{A}_i \cup \hat{B}_i$ is a complete graph, i.e. every pair in $\mathcal{P}'_2$ is c-well-separated under the unit-disk graph metric. The same argument applies to $\mathcal{P}'_3$ as the distance between two points in $C(s)$ is at most $2\delta \leq 1$.

Now, consider a pair $(\hat{A}_i, \hat{B}_i) \in \mathcal{P}'_1$. We have two cases.

(1). When $|A_i| = |B_i| = 1$. Then we must have $\pi(A_i, B_i) \geq (2c+2)\delta$ according

to the construction rule, and thus

$$\pi(\hat{A}_i, \hat{B}_i) \geq \pi(A_i, B_i) - 2\delta \geq 2c\delta = c/(c+2)\,.$$

On the other hand, $D_\pi(\hat{A}_i), D_\pi(\hat{B}_i) \leq 2\delta = 1/(c+2)$. Therefore, $(\hat{A}_i, \hat{B}_i)$ is c-well-separated.

(2). When $|A_i| > 1$ or $|B_i| > 1$. Clearly,

$$\pi(\hat{A}_i, \hat{B}_i) \geq \pi(A_i, B_i) - 2\delta\,, \text{ and } D_\pi(\hat{A}) \leq D_\pi(A) + 2\delta\,.$$

Since one of A_i and B_i contains at lease two clusterheads, it must be true that $\max(D_\pi(A_i), D_\pi(B_i)) \geq \delta$ as the distance between two clusterheads is at least δ. So, $\max(D_\pi(\hat{A}_i), D_\pi(\hat{B}_i)) \geq \delta$, and $\max(D_\pi(\hat{A}_i), D_\pi(\hat{B}_i)) \leq \max(D_\pi(A_i), D_\pi(B_i)) + 2\delta \leq 3\max(D_\pi(A_i), D_\pi(B_i))$.

As A_i, B_i are c'-well-separated under π', $\pi'(A_i, B_i) \geq c' \cdot \max(D_{\pi'}(A_i), D_{\pi'}(B_i))$. Therefore,

$$\begin{aligned}
\pi(\hat{A}_i, \hat{B}_i) &\geq \pi(A_i, B_i) - 2\delta \\
&\geq (\pi'(A_i, B_i) - 12\delta)/(1 + 12\delta) - 2\delta \\
&\quad \text{by Lemma 7.2.8} \\
&\geq c'/(1+12\delta) \cdot \max(D_{\pi'}(A_i), D_{\pi'}(B_i)) - 14\delta \\
&\geq c'/(1+12\delta) \cdot \max(D_{\pi}(A_i), D_{\pi}(B_i)) - 14\delta \\
&\geq (c'/(3(1+12\delta)) - 14) \cdot \max(D_\pi(\hat{A}_i), D_\pi(\hat{B}_i)) \\
&\geq c \cdot \max(D_\pi(\hat{A}_i), D_\pi(\hat{B}_i))\,. \\
&\quad \text{by } c \geq 1, \delta = 1/(2c+4), \text{ and } c' = 9(c+14).
\end{aligned}$$

In both cases, $\hat{A}_i, \hat{B}_i$ are c-well-separated, i.e. all the pairs in $\mathcal{P}'_1$ are c-well-separated. □

Now, we are ready to claim that

Theorem 7.2.10. *For any set S of n points in the plane and any $c \geq 1$, there exists a c-WSPD $\mathcal{P}$ of S under the unit-disk graph metric where $\mathcal{P}$ contains $O(c^4 n \log n)$*

pairs and can be computed in $O(c^4 n \log n)$ *time.*

Proof: By combining Corollary 7.2.5 and Lemma 7.2.7, we have that $|\mathcal{P}'_1| \leq |\mathcal{P}| = O(c^2 n \log n/\delta^2) = O(c^4 n \log n)$. If $|A_i| = 1$, then the number of pairs $(A_i, B_i) \in \mathcal{P}'_2$ where $|B_i| = 1$ is bounded by $O(1/\delta^2) = O(c^2)$. Since the size of the geometric well-separated pair decomposition is linear in terms of the number of points [40], $|\mathcal{P}'_2| = O(c^2 n)$. Clearly, $|\mathcal{P}'_3| = O(n)$. Sum them up, we have that $|\mathcal{P}'| = O(c^4 n \log n)$.

By Theorem 11 and Lemma 7.2.7, it is easy to see that the total time needed is $O(c^4 n \log n)$. □

Similarly, in higher dimensions, we have that

Corollary 7.2.11. *For any set* S *of* n *points in* $\mathbb{R}^k$*, for* $k \geq 3$*, and for any constant* $c \geq 1$*, there exist a c-WSPD* $\mathcal{P}$ *of* S *under the unit-ball graph metric where* $\mathcal{P}$ *contains* $O(n^{2-2/k})$ *pairs and can be constructed in* $O(n^{4/3} \operatorname{polylog} n)$ *time for* $k = 3$ *and in* $O(n^{2-2/k})$ *time for* $k \geq 4$*.*

Proof: For simplicity of computation, we use axis-aligned boxes instead of balls to find clusterheads with constant bounded density. A point p is covered by a point s if p is inside the box with size 2δ centered at s. Finding the minimal cover can be done by using a dynamic rectilinear range search tree in k-dimension [46]. The running time is $O(n \operatorname{polylog} n)$. Notice that every point can be covered by at most a constant number of clusterheads, thus we can find the nearest clusterhead for every point in linear time in total. To find a pair of gateways between two clusterheads s_1, s_2, we compute the bichromatic closest pair between two sets $C(s_1), C(s_2)$. Let $m_1 = |C(s_1)|$ and $m_2 = |C(s_2)|$. According to [4], when $k = 3$, the bichromatic closest pair computation takes $O((m_1 m_2)^{2/3} \operatorname{polylog} n)$ time; and when $k = 4$, the computation time is

$$\begin{aligned} & O((m_1 m_2)^{1-1/(\lceil k/2 \rceil + 1)+\varepsilon} + m_1 \log m_2 + m_2 \log m_1) \\ = \; & O((m_1 m_2)^{1-1/k} + m_1 \log m_2 + m_2 \log m_1)\,. \end{aligned}$$

Since each set is only involved in $O(1)$ bichromatic closest pair computation, the total time is $O(n^{4/3} \operatorname{polylog} n)$ when $k = 3$ and $O(n^{2-2/k})$ for $k \geq 4$. Computing the WSPD on the clusterheads takes $O(n^{2-2/k})$ time, according to Theorem 11. □

7.2.3 Estimating distance between pairs

We have shown how to construct a well separated pair decomposition for a unit-disk/unit-ball graph. To apply a WSPD in solving proximity problems in the unit-disk graphs, we first need to estimate the shortest path distances between $O(n \log n)$ pairs of the WSPD. Note that in our construction for the point sets with constant bounded density, we use Euclidean distance as a lower bound for the unit-disk graph distance and the size of the point set as an upper bound for the diameter. While these approximations are sufficient for bounding the size of a WSPD, it is too coarse for obtaining good approximation. Recall that $\sigma(A)$ is an (arbitrary) point in set A. For a c-well-separated pair (A, B), we can use the estimated distance $\hat{\pi}(\sigma(A), \sigma(B))$ to approximate all the pairwise distances between points in A and points in B. In this section, we show several tradeoffs for measuring the distances between m pairs of points in a unit-disk graph.

Denote by $t(n, c, m)$ the time needed to compute m-pairs c-approximate distance in a unit disk graph. In what follows, we set $c_0 = 2.42 > \frac{4\sqrt{3}}{9}\pi$ and c_1 a number slightly smaller than c_0 but greater than $\frac{4\sqrt{3}}{9}\pi$. We have that:

Lemma 7.2.12. *1.* $t(n, c_1, m) = O(n \log^3 n + m)$.

2. $t(n, 1+\varepsilon, m) = O(n^2/(\varepsilon r) + mr/\varepsilon)$, *for any* $1 \leq r \leq n$.

Proof: 1. We first construct a planar $\frac{4\sqrt{3}}{9}\pi$-spanner of the unit disk graph in $O(n \log n)$ time [106]. Now, we apply Thorup's construction of $(1+\varepsilon)$-approximate distance oracle [150] to that planar spanner, for a sufficiently small constant $\varepsilon > 0$. The bound follows immediately from the preprocessing and query time bounds of Thorup's algorithm.

2. We again cluster the points and consider the set of clusterheads, X. Suppose that we have constructed a $(1 + \varepsilon/2)$-approximate shortest distance oracle for X. For two query points q_1, q_2, if $|q_1 q_2| \leq 1$, we return $|q_1 q_2|$. Otherwise, we find the clusterheads s_1, s_2 that cover q_1 and q_2, respectively, and return $\hat{\pi}(q_1, q_2) = \pi'(s_1, s_2) + 2\delta$ as an approximation of $\pi(q_1, q_2)$.

We first verify that $\hat{\pi}(q_1, q_2)$ is a $(1+\varepsilon)$-approximation for $\delta = O(\varepsilon)$. By Lemma 7.2.8, $\hat{\pi}(q_1, q_2) \leq (1 + 2\delta)\pi(s_1, s_2) + 14\delta \leq (1 + 2\delta)(\pi(q_1, q_2) + 2\delta) + 14\delta$. Since

$\pi(q_1, q_2) > 1$, we have that $\hat{\pi}(q_1, q_2) \leq \big((1+2\delta)^2 + 14\delta\big)\pi(q_1, q_2)$. If we take $\delta = \Theta(\varepsilon)$ and is sufficiently small, we have $\hat{\pi}(q_1, q_2) \leq (1+\varepsilon)\pi(q_1, q_2)$.

Now we show how to compute $\pi'(s_1, s_2)$. The density of X is $O(1/\delta^2) = O(1/\varepsilon^2)$. The graph formed by connecting neighboring pairs in X is an $O(1/\varepsilon^2)$-overlap graph as defined in [115] and therefore admits a balanced separator with size $O(\sqrt{n}/\varepsilon)$. Furthermore, a balanced separator can be computed in linear time by the deterministic method in [53]. Now, we extend the shortest distance algorithm for planar graphs in [14] to the above geometric graph on X. By using the same technique, we can obtain a tradeoff with $O(n^2/(\varepsilon r))$ preprocessing time and $O(r/\varepsilon)$ query time to answer a shortest path length query, for any $1 \leq r \leq \sqrt{n}$. The total running time for answering $O(m)$ queries is $O(n^2/(\varepsilon r) + rm/\varepsilon)$. □

7.3 Approximate distances via WSPD

In this section, we show the application of the well separated pair decomposition in obtaining efficient algorithms for approximating the furthest neighbor (diameter, center), nearest neighbor (closest pair), median, and stretch factor, all under the unit-disk graph metric. Since the running time of the algorithms for computing c_0-approximate and $(1+\varepsilon)$-approximate distance are different, we will describe the bounds for both approximations (recall that $c_0 = 2.42$). Roughly speaking, our algorithms for computing c_0-approximation is about linear and for computing $(1+\varepsilon)$-approximation is about $O(n\sqrt{n})$, dominated by the distance estimation.

We should note that for the problems of computing diameter and center, there is a simple linear time method to achieve a 2-approximation[3]. It is therefore not interesting to present algorithms to obtain c_0-approximation for those problems. For the other problems, it is still interesting as we are not aware of any algorithms that achieve comparable approximation ratio in sub-quadratic time, even for planar graphs.

We first describe the well-separated pair decomposition we will use. In what

[3]We compute the shortest path tree at an arbitrary node r, the maximum length ℓ from the root to a leaf is a 2-approximation of the diameter of the graph. This is because for the pair of nodes u, v whose distance equals to the diameter, $\pi(u, v) \leq \pi(u, r) + \pi(r, v) \leq 2\ell$.

follows, we also include the time for measuring the distances between the well separated pairs into the construction time. For c_0-approximation, we construct a c-well-separated pair decomposition $\mathcal{P}_1$ for sufficiently large constant c and, for each pair (A, B) in the WSPD, compute c_1-approximate distance $\hat{\pi}_1(A, B)$ between $\sigma(A)$ and $\sigma(B)$ according to Lemma 7.2.12.1. For $(1+\varepsilon)$-approximation, we compute a c-well-separated pair decomposition $\mathcal{P}_2$ for $c = O(1/\varepsilon)$ and, for each pair (A, B), compute the $(1+\varepsilon/2)$-approximate distance $\hat{\pi}_2(A, B)$ between $(\sigma(A), \sigma(B))$ by Lemma 7.2.12.2 with $r = \varepsilon^2\sqrt{n/\log n}$. The following result then follows.

Lemma 7.3.1. *$\mathcal{P}_1$ contains $O(n \log n)$ pairs and can be computed in $O(n \log^3 n)$ time. $\mathcal{P}_2$ contains $O(n \log n/\varepsilon^4)$ pairs and can be computed in $O(n\sqrt{n \log n}/\varepsilon^3)$ time. For any pair of points (p, q), suppose that its covering pair in $\mathcal{P}_1(\mathcal{P}_2)$ is (A, B), then $\hat{\pi}_1(A, B)$ ($\hat{\pi}_2(A, B)$) is a c_0-approximation ($(1+\varepsilon)$-approximation) of $\pi(p, q)$.*

In the process of producing a well-separated pair decomposition, we constructed several trees, the balanced hierarchical decomposition tree for constant bounded density points and the fair split trees for geometric well-separated pair decomposition [40]. For presentation simplicity, we treat them as a single tree T_1' and T_2', for $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, by joining the trees created in the geometric well-separated pair decomposition to the clusterheads appropriately. In what follows, $\mathcal{P}, T', \hat{\pi}$ mean that they could be either case.

7.3.1 $(1+\varepsilon)$-distance oracle

While the computation for $\mathcal{P}_2$ takes time $O(n\sqrt{n \log n}/\varepsilon^3)$, the space needed is only $O(n \log n/\varepsilon^4)$. We can use it to answer $(1+\varepsilon)$-approximate distance query between any two points (p_1, p_2) by first locating the pair (A, B) that covers (p, q) and returning $\hat{\pi}(A, B)$. The query time is the time needed to discover a well separated pair in $\mathcal{P}_2$ that covers the query pair (p, q). We show this can be done in $O(1)$ time by using the properties of WSPD.

Corollary 13. *For a unit-disk graph on n points and for any $\varepsilon > 0$, we can preprocess it into a data structure with $O(n \log n/\varepsilon^4)$ size so that for any query pair, a $(1+\varepsilon)$-approximate distance can be answered in $O(1)$ time.*

Proof: It suffices to prove it for constant-bounded density point sets. We store all the pairs in $\mathcal{P}$ in a hash table indexed by the pairs. We will show that for each query pair (p, q), we can find $O(1)$ candidate pairs that are guaranteed to contain the pair in $\mathcal{P}$ that covers (p, q). Then, we simply query the hash table using those candidate pairs and discover the one that does cover (p, q).

We modify our construction in Section 7.2.1 so that we are more careful on deciding when to include a pair in $\mathcal{P}$. We use a c_1-approximate distance oracle as constructed in Lemma 7.2.12.1. When producing $\mathcal{P}$, we include a pair in $\mathcal{P}$ if $\hat{\pi}(A, B) > (cc_1 + 2c_1) \max(|A| - 1, |B| - 1)$, where $\hat{\pi}(A, B)$ is c_1-approximate distance between $\sigma(A)$ and $\sigma(B)$, $\pi(\sigma(A), \sigma(B)) \leq \hat{\pi}(A, B) \leq c_1\pi(\sigma(A), \sigma(B))$. We define $s = \max(|A| - 1, |B| - 1)$. Therefore $\pi(A, B) \geq \pi(\sigma(A), \sigma(B)) - 2s \geq \hat{\pi}(A, B)/c_1 - 2s \geq cs$. This shows that $\mathcal{P}$ is a valid well-separated pair decomposition.

On the other hand, assume that (A, B) was obtained by splitting $(P(A), B)$. We know that $\hat{\pi}(P(A), B) < (cc_1+2c_1) \max(|P(A)|-1, |B|-1) \leq \beta(cc_1+2c_1)s$. Therefore, $\pi(A, B) \leq \pi(P(A), B) + |P(A)| \leq \hat{\pi}(P(A), B) + |P(A)| \leq \beta(cc_1 + 2c_1)s + \beta \cdot s = \beta(cc_1+2c_1+1)s$. Define $c_2 = \beta(3c_1+1)$. Then for any $c \geq 2$ and any pair $(A, B) \in \mathcal{P}$, $cs \leq \pi(A, B) \leq cc_2s$, where $s = \max(|A| - 1, |B| - 1)$.

Now, to answer a query (p, q), we first use the c_1-approximate distance oracle to compute an approximation ℓ of $\pi(p, q)$, i.e. $\pi(p, q) \leq \ell \leq c_1\pi(p, q)$. Suppose that $(A, B) \in \mathcal{P}$ is the pair that covers (p, q). Then, we have

$$s \leq \pi(A, B)/c \leq \pi(p, q)/c \leq \ell/c\,.$$

On the other hand $s \geq \pi(A, B)/(cc_2) \geq (\pi(p, q)-2s)/(cc_2)$. That is, $s \geq \pi(p, q)/(cc_2+2) \geq \ell/(c_1(cc_2 + 2)) \geq \frac{\ell/c}{c_1c_2+2c_1}$.

Set $\hat{\ell} = \ell/c$. Assume without loss of generality that $|A| \geq |B|$, so $s = |A| - 1$. Then, for (A, B) to cover (p, q), A has to be an ancestor of p in T', and the size of A is sandwiched by $\hat{\ell}/(c_1c_2 + 2c_1) + 1$ and $\hat{\ell} + 1$. Notice that c_1, c_2 are constants independent of c. There are only $O(1)$ such nodes in T'. Since $|A|/\beta \leq |B| \leq |A|$ for a constant β, there are only $O(1)$ such B's as well. We now simply form the $O(1)$ candidate pairs by joining every possible A and B. □

7.3.2 Furthest neighbor

Suppose that $S_1 \subseteq S$. For any p, define the (relative) furthest neighbor of p to be $\xi(p) = \arg\max_{q \in S_1} \pi(p, q)$ in S_1. Then the diameter of S_1 is $D(S_1) = \max_{p \in S_1} \pi(p, \xi(p))$. The center of S_1 is the point that minimizes the maximum distance to the other points, i.e. $\arg\min_{p \in S_1} \pi(p, \xi(p))$. Therefore, once we have compute approximate furthest neighbors for all the p's, we also obtain approximate diameter and center.

Consider any WSPD. To compute the furthest neighbor of S_1, we traverse the balanced hierarchical decomposition tree T' and mark all the nodes $v \in T'$ where $S(v) \cap S_1 \neq \emptyset$. This can be done in $O(n)$ time in a post-order visit of the tree. A pair $P = (S(u), S(v))$ is called *marked* if both u and v are marked. Let

$$R_1(u) = \max\{\hat{\pi}(S(u), B) | (S(u), B) \text{ is marked}\},$$

and 0 if there is no such pair. With each node u, we also record $\ell(u)$ such that $\big(S(u), S(\ell(u))\big)$ achieves $R_1(u)$.

For any $p \in S_1$, consider the path P in T' from p to the root. Suppose that u is the node that maximizes $R_1(u)$ among all the nodes on P. Now, we pick any point, say q, from $S(\ell(u)) \cap S_1$ (since $\ell(u)$ is marked, $S(\ell(u)) \cap S_1 \neq \emptyset$) and claim that it is an approximate furthest neighbor with the approximation ratio 2.42, if the above process is applied to $\mathcal{P}_1$, or $1 + \varepsilon$, if applied to $\mathcal{P}_2$. For the correctness, consider the (marked) pair in $\mathcal{P}$ that covers $(p, \xi(p))$. Suppose it is $(S(u), S(v))$. Then $R_1(u) \geq \hat{\pi}(S(u), S(v))$. Since the pairs are well-separated, q is an approximate furthest neighbor of p. After we have computed the approximate furthest neighbor, it is simple to compute the diameter and the center. Therefore, we have that

Corollary 14. *For any set S of n points in the plane and any $S_1 \subseteq S$, we can compute*

- *c_0-approximate furthest neighbor for all the points in S_1 in $O(n \log^3 n)$ time; and*
- *$(1 + \varepsilon)$-approximation, for any $\varepsilon > 0$, of the furthest neighbor, the diameter of S_1, and the center of S_1 in $O(n\sqrt{n \log n}/\varepsilon^3)$ time.*

7.3.3 Nearest neighbor, closest pair

Computing the nearest neighbor or closest pair in S under the unit-disk graph metric is trivial — it is the same as under the Euclidean metric as long as the graph is connected. However, the problem becomes harder if we restrict our attention to a subset $S_1 \subseteq S$, i.e. computing the nearest neighbor in S_1 for each point in S_1 or computing the closest pair between points in S_1. For any two sets S_1, S_2, we can also define the bichromatic closest pair to be $\arg\min_{p \in S_1, q \in S_2} \pi(S_1, S_2)$.

By using the same technique in the previous section, we are able to show:

Corollary 15. *For any set S of n points in the plane, and any $S_1, S_2 \subseteq S$, we can compute*

- *c_0-approximation for the nearest neighbor for all the points in S_1, the closest pair in S_1, the bichromatic closest pair of S_1, S_2, in time $O(n \log^3 n)$; and*
- *$(1+\varepsilon)$-approximation for the same problems in time $O(n\sqrt{n \log n}/\varepsilon^3)$.*

Remark. We should note that for the Euclidean metric on a set of points, we can actually enumerate $O(n)$ pairs of points that is guaranteed to include the closest pair, by applying the geometric well-separated pair decomposition in [40]. However, unlike in the geometric case, we can only compute approximate closest pair, since our distance oracle is approximate.

7.3.4 Median

Similar to the definition of center, the median is defined to be the point that minimizes the average (or total) distance to all the other points. Let $\rho(p) = \sum_{q \in S_1} \pi(p, q)$. Then the median of S_1 is the point that minimizes $\rho(p)$.

By using similar technique, we can show that

Corollary 16. *For any planar point set S with n points and $S_1 \subseteq S$, a c_0-approximate median of S_1 can be computed in $O(n \log^3 n)$ time, and for any $\varepsilon > 0$, a $(1+\varepsilon)$-approximation can be computed in $O(n\sqrt{n \log n}/\varepsilon^3)$ time.*

Proof: Computing an approximate median is similar to computing an approximate furthest neighbor. The only difference is that instead of computing $R_1(u)$, we compute

$$R_2(u) = \sum_{(S(u),B)\in\mathcal{P}} \hat{\pi}(S(u), B) \cdot |B|\,,$$

and then for each point p and the path P from p to the root, compute $\hat{\rho}(p) = \sum_{u\in P} R_2(u)/(n-1)$, as an approximation of $\rho(p)$. The correctness is guaranteed by the property of pair decomposition that every pair of points is covered by a unique pair in the decomposition. Again, we pick the point with the minimum $\hat{\rho}(p)$ to be the approximate median. The approximation ratio and running time bounds follow immediately. □

7.3.5 k-center

For a metric (S, π), the k-center is a set $K \subseteq S$, $|K| = k$, such that $\max_{p\in S} \min_{q\in K} \pi(p, q)$ is minimized. Gonzalez [66] proposed a furthest point algorithm to compute a 2-approximate solution for discrete k-center on a general metric. The algorithm can be directly applied to the unit-disk graph metric to get a 2-approximate k-center. However, the algorithm requires computing the pair-wise distances. We show a careful implementation by using the well-separated pair decomposition in time $O(n\sqrt{n \log n}/\varepsilon^3)$ (or $O(n \log^3 n)$) to find a $(2+\varepsilon)$ $(2c_0)$-approximate k-center, for $c_0 = 2.42$.

Gonzalez's algorithm maintains a subset $R \subseteq S$ and adds to R the point $p \in S \setminus R$ whose distance to R is maximized. The algorithm ends when R contains k points. R is a 2-approximate k-center. Here we show how to compute R by using a c-well-separated pair decomposition, for $c = O(1/\varepsilon)$ but sufficiently large. Basically, we want to find a point p whose distance to R is a $(1+\varepsilon)$-approximation of the maximum possible. We denote a pair $(A, B) \in \mathcal{P}$ as marked if $A \bigcap R \neq \emptyset$ and $B \bigcap (S \setminus R) \neq \emptyset$. Notice that since every pair of points (p, q) is covered by at least one pair in $\mathcal{P}$, all the edges connecting R and $S \setminus R$ must be covered by all the marked pairs. We also maintain a priority queue Q for all the marked pairs so that the pair with the furthest distance is stored at the root.

To start, R contains one arbitrary point from S. We find the marked pair (A, B) in Q so that the distance between (A, B) is the furthest among all the marked ones. We take an arbitrary point q from B and add q to R. Certain pairs from $\mathcal{P}$ need to be marked or unmarked. We keep doing this until R contains k nodes. To check whether a pair should be marked, we check the two subsets respectively. Specifically, for each node u in the decomposition tree T, we check if it intersects with R and if it is fully contained in R. Update on those properties is done in a bottom-up fashion. Once a point p is added to R, only the nodes on the path from the leaf p to the root of the tree T can change their properties. The newly marked pairs are inserted to the priority queue Q. The pairs that are unmarked are deleted from the queue.

What remains is to bound the running time. A node A is marked if and only if $A \bigcap R \neq \emptyset$. Since R is keep increasing, once A is marked, it won't be unmarked. Similarly, once B is unmarked, it is not going to be marked any more. This implies that a pair $(A, B) \in \mathcal{P}$ can be marked and unmarked at most once respectively. Insertion and deletion of the priority queue takes $O(\log n)$ in the worst-case. So the total running time is $O(\log n)$ times the total number of the well-separated pairs. By the result in Section 7.3, we can get a $(2 + \varepsilon)$ $(2c_0)$ approximate k-center in time $O(\sqrt{n \log n}/\varepsilon^3)$ (or $O(n \log^3 n)$), for $c_0 = 2.42$.

Remark. The k-center problem for general metric is hard to approximate within factor $2 - \varepsilon$ for any $\varepsilon > 0$ [78, 126] unless $P = NP$. For the geometric k-center problem, where the vertices lie in the plane and the geometric metric is used, Feder and Greene [55] gave a 2-approximation algorithm that runs in time $O(n \log k)$. They also show that obtaining an approximation within 1.822 is NP-hard. This also gives a lower bound on the approximation factor of the k-center problem in weighted unit-disk graphs. It is still not clear what is the tight hardness result for k-center on unit-disk graphs.

7.3.6 Stretch factor

For a graph G defined on S, the *stretch factor* of G with respect to π is defined as $\max_{p,q \in S} \pi_G(p,q)/\pi(p,q)$. Narasimhan and Smid [116] gave an algorithm to approximate the stretch factor of a geometric graph to the Euclidean metric using the geometric well-separated pair decomposition. By following the same argument we can approximate the stretch factor of an arbitrary graph G with respect to the unit-disk graph metric. Again, we consider the well-separated pair decomposition $\mathcal{P}$. For each pair $(A,B) \in \mathcal{P}$, we pick any two points (p,q) where $p \in A$ and $q \in B$ and compute the approximate shortest path $\hat{\pi}_G(p,q)$ in G and $\hat{\pi}(p,q)$ in I. The maximum ratio of $\hat{\pi}_G(p,q)/\hat{\pi}(p,q)$ over all pairs in $\mathcal{P}$ is an approximation to the stretch factor by the same argument in [116].

Corollary 17. *For any graph G on S, we can compute an $O(1)$-approximate stretch factor of G in time $O(\tau_1'(n \log n))$ where $\tau_1'(m)$ is the time to compute m $O(1)$-approximate shortest path queries in G. In particular, if G is a subgraph of I, an $O(1)$-approximate can be computed in time $O(n \log^3 n)$. Similarly, we can compute for any $\varepsilon > 0$, a $(1+\varepsilon)$-approximate stretch factor of G in time $O(\tau_2'(n \log n/\varepsilon^4) + n\sqrt{n \log n}/\varepsilon^3)$, where $\tau_2'(m)$ is the time to compute m $(1+\varepsilon)$-approximate shortest path queries in G. When G is a subgraph of I, this can be done in $O(n\sqrt{n \log n}/\varepsilon^3)$ time.*

7.4 Approximate paths via WSPD

In the previous section we have shown that the well-separated pair decomposition can be used to approximate the shortest path distances between any two points p, q, by using the (approximate) shortest path distances between p', q', if (p,q) and (p',q') belong to the same well-separated pair. Here we show that not only the approximate shortest distance, but the actual paths between any two points can be obtained.

The algorithms of obtaining the approximate shortest distances are of two kinds: an almost linear time algorithm of constructing constant-approximate shortest distances and an $O(n^{3/2}\sqrt{\log n})$ algorithm of constructing $(1+\varepsilon)$-approximate shortest

paths with arbitrary $\varepsilon > 0$. We discuss them separately. We assume that the nodes have constant bounded density, since the case of arbitrary density can be turned into constant bounded density by a clustering technique.

To obtain the constant-approximate shortest paths, we first planarize the unit-disk graph by using the Restricted Delaunay Graph which is a $\frac{4\sqrt{3}}{9}\pi$-spanner. Then we apply Thorup's distance oracle [150] on the spanner. Thorup's distance oracle [150] can also be used to compute the actual approximate shortest paths.

For $(1+\varepsilon)$-approximate shortest paths, we first show a compact way of storing the approximate shortest paths in $O(n \log^5 n/\varepsilon^4)$ space totally. In computing the c-well-separated pair decomposition, we replace the geometric distance between two pairs by the c_1-approximate shortest distances as the upper bound, as in the proof of Corollary 13. Therefore for each c-well-separated pair $(A, B) \in P$, $cs \leq \pi(A, B) \leq cc_2 s$, where $s = \max(|A| - 1, |B| - 1)$, c_1, c_2 are constants. We also choose $c = O(\log n/\varepsilon)$ so that the size of the WSPD is $O(n \log^5 n/\varepsilon^4)$. For a c-well-separated pair (A, B), we have $\pi(p', q') \leq (1 + \varepsilon')\pi(p, q)$, if $(p, q), (p', q')$ are pairs of points in (A, B), $\varepsilon' = \ln(1 + \varepsilon)/\log n = O(\varepsilon/\log n)$.

For each well-separated pair (A, B), we first construct the approximate shortest path for a pair of points (p, q) with $p \in A$, $q \in B$. The approximate shortest path between other pairs (p', q'), $p' \in A$, $q' \in B$, can be obtained by concatenating the approximate shortest paths between (p', p), (p, q) and (q, q'). We take the decomposition tree T' and evaluate the approximate shortest paths between well-separated pairs in an increasing order of the approximate shortest distances. For a singleton pair (p, q), it is easy to see that the shortest distance between p, q is at most a constant, which can be computed in $O(1)$ time by doing breadth-first search starting at p for a constant number of hops, since the nodes have constant-bounded density. For a pair (p, q) such that $p \in A$, $q \in B$ and $(A, B) \in \mathcal{P}$, we chop the shortest path $P(p, q)$ into $O(c_2)$ segments so that each segment has length less than $c \cdot s$, $s = \max(|A| - 1, |B| - 1)$. This implies that each segments $p_i p_{i+1}$ is covered by a pair which has already been computed earlier. Therefore the approximate shortest path can be obtained by concatenating the approximate shortest paths of each segment. In the following we show how to find such sub-paths. We define a graph G' whose vertices are the nodes in T',

we add an edge (A, B) in G' if $(A, B) \in \mathcal{P}$. We do a breadth-first search in graph G' starting from node A for at most $O(c_2)$ hops which must cover node B. All the nodes X visited during such a breadth-first search must have size $|X| = O(|A|)$. By similar argument as in Lemma 7.2.3, we know that the total number of nodes visited during the breadth-first search is bounded by $\min(O(n/|A|), O(|A|)) = O(\sqrt{n})$ pairs. That is, the breadth-first search takes time $O(\sqrt{n})$. The approximate shortest path is represented compactly by only remembering the $O(1)$ hops in graph G'. The approximation error accumulated during induction is $(1+\varepsilon')^{\log n} = (1+\varepsilon)$. The total memory needed is at most $O(n \log^5 n/\varepsilon^4)$. On a query of an approximate shortest path, we need to concatenate the approximate shortest paths for every hop recursively. This takes time $O(k)$, where k is the length of the approximate shortest path.

7.4.1 Minimum spanning/steiner tree

Given a subset $R \subseteq S$. We apply the Prim's Algorithm [129] to compute the Minimum Spanning Tree U. Prim's Algorithm involves adding one point p at a time to the partial spanning tree U' so that the distance from p to U' is minimized. The algorithm works in exactly the same way as the approximate k-center algorithm, by finding the closest point instead of the furthest point to a subset of points. A $(1 + \varepsilon)$-approximate MST can be constructed in time $O(n\sqrt{n \log n}/\varepsilon^3 + n \log^5 n/\varepsilon^4)$. Also a c_0-approximate MST can be constructed in time $O(n \log^5 n)$. Furthermore since the minimum spanning tree of R is a 2-approximation of the minimum Steiner tree of R. This immediately gives a $(2 + \varepsilon)$-approximate minimum Steiner tree for a subset of nodes in the unit-disk graph, which could be used to do multi-cast routing in wireless *ad hoc* networks.

The online Steiner Tree problem takes the input of a series of requests $v_1, v_2, \cdots, v_n$, at each request a vertex v_i needs to be connected by a Steiner Tree T_i so that $T_{i-1} \subseteq T_i$. This is also called the Dynamic Steiner Tree problem. Imase and Waxman [80] proved that the greedy algorithm, i.e, using the shortest path to connect v_i to T_{i-1}, is a $O(\log n)$ approximation to the optimum offline solution. They also showed that this bound is tight in the worst case. This greedy algorithm, also fits perfectly in our

framework and therefore enjoys a near-linear running time.

Goel and Munagala [63] proposed an online algorithm that computes delay sensitive Steiner trees for a set of requests coming online. The algorithm is based on the greedy algorithm of Imase and Waxman [80] and gives a Steiner tree with weight at most $O(\log n)$ times the off-line optimum and height at most $O(1)$ times the height of the shortest path tree. The basic idea is to apply the greedy algorithm and detect when the tree gets too tall and reroute the troublesome nodes using shortest paths to the root. Since every node only get rerouted at most once, the running time is bounded in the same way as above.

Computing the Minimum Spanning Tree in an off-line fashion can be done in $O(|E| + |V| \log |V|)$ by the algorithm of Mehlhorn [114], where $|E|$ is the number of edges and $|V|$ is the number of vertices. So by an ε-clustering of the points the ε-approximate MST can be constructed in time $O(n \log n)$. The algorithm by Mehlhorn, however, doesn't work for the online computation. So our algorithm gives a general method to construct the MST (and therefore Minimum Steiner Tree) in both offline and online settings.

7.5 Extensions

There are several direct extensions of our techniques. Here, we outline the extension to the intersection graph of disks with bounded radii ratio and the unweighted unit-disk graph.

7.5.1 Disks with bounded ratio of radii

When the size of the disks are not uniform, it is generally not possible to obtain sub-quadratic well-separated pair decomposition of the metric induced by the intersection graph. This can be shown by the example where there is a big disk and $n-1$ pairwise disjoint small disks intersecting it. Indeed, the intersection graph of this example is a tree with one internal node and $n-1$ leaves.

However, if the ratio between the radii of any two disks (or balls) is upper bounded

by a constant, then the packing property (Lemma 7.2.3) still holds. We can obtain similar results for the intersection graph of disks (or balls in high dimensions) with bounded radii ratio.

7.5.2 Unweighted unit-disk graphs

There are applications in which people are interested in the unweighted unit-disk graph. The results for point set with constant bounded density can be directly extended to unweighted unit-disk graphs. If the density is unbounded, then it is impossible to obtain a sub-quadratic size well-separated pair decomposition as shown by the example of the unweighted complete graph. But again we can apply the clustering technique to reduce it to the problem for point sets with constant unbounded density. The clustering will increases the approximation ratio by a multiplicative factor of 3. This gives us almost linear time algorithms to find $3c_0$-approximate solutions to the following problems: the furthest neighbor, nearest neighbor, closest pair, bichromatic closest pair, median, and stretch factor, all with respect to the unweighted unit-disk graph metric. Again, we did not list the problems of computing diameter and center because there are trivial 2-approximate algorithms.

For the unweighted unit-disk graph $I(S)$ on point set S, we first cluster the points by centers X with radius 1, i.e, any two centers $c_1, c_2 \in X$ must be distance at least 1 away, any point is covered by at least one center. Furthermore we assign a unique center $c(p)$ to every node p in S. For any two centers within distance 3, if there exists a path with no more than three hops to connect them, we pick up such two nodes as gateways. Define Z to be the union of centers X and gateways Y. The shortest path metric in the unweighted unit-disk graph $I(Z)$ is denoted by π', to be distinguished by the metric π in $I(S)$. It is easy to see that Z has constant bounded density. So we build the c-well-separated pair decomposition $\mathcal{P}'$ on Z. For each pair $(A', B') \in \mathcal{P}'$, we build a pair (A, B) where $A = \bigcup_{c(p)\in A'} p$, $B = \bigcup_{c(q)\in B'} q$.

Lemma 7.5.1. *For $p, q \in S$,*

1. $\pi(p, q) \leq \pi'(p, q)$;

2. $\pi'(p,q) \leq 3\pi(p,q) + 2$*, if* p, q *are centers, then* $\pi'(p,q) \leq 3\pi(p,q)$*.*

Proof: The first claim is easy. The second one is proved in chapter 4. □

Set $c = 6/\varepsilon$, we have,

Theorem 7.5.2. *For any two pairs* $(p_1, q_1), (p_2, q_2) \in (A, B)$*, where* $(A, B) \in \mathcal{P}$*,* $\pi(p_1, q_1) \leq (1+\varepsilon)\pi(p_2, q_2) + (4 + 2\varepsilon)$*.*

Proof: Take the centers of p_2, q_2, $c(p_2) \in A'$, $c(q_2) \in B'$. We can see that $\pi(p_1, c(p_2)) \leq 1 + \pi(c(p_1), c(p_2)) \leq 1 + \pi'(c(p_1), c(p_2)) \leq 1 + D'(A')$, where D' denote the diameter in metric π'. Since (A', B') is c-well-separated, we have $\pi'(A', B') \geq c \cdot \max(D'(A'), D'(B'))$. So $\pi'(c(p_2), c(q_2)) \geq c \cdot \max(D'(A'), D'(B'))$. $\pi(c(p_2), c(q_2)) \leq \pi(p_2, q_2) + 2$. Combining all these, we have,

$$\begin{aligned}
\pi(p_1, q_1) &\leq \pi(p_1, c(p_2)) + \pi(c(p_2), c(q_2)) + \pi(c(q_2), q_1)) \\
&\leq 1 + D'(A') + \pi(c(p_2), c(q_2)) + 1 + D'(B') \\
&\leq \pi(c(p_2), c(q_2)) + 2 + D'(A') + D'(B') \\
&\leq \pi(c(p_2), c(q_2)) + 2 + (2/c)\pi'(c(p_2), c(q_2)) \\
&\leq (1 + 6/c)\pi(c(p_2), c(q_2)) + 2 \\
&\leq (1 + 6/c)\pi(p_2, q_2) + (4 + 12/c) \\
&= (1+\varepsilon)\pi(p_2, q_2) + (4 + 2\varepsilon).
\end{aligned}$$

□

The above theorem implies that, the distances between two pairs covered by the same well-separated pair A, B are $(1+\varepsilon, 4+2\varepsilon)$-approximation of each other[4].

Remark. For the k-center problem on unweighted unit-disk graph (each edge has a weight 1), obtaining a $2-\varepsilon$ approximation factor is NP-hard, by reduction to the minimum dominating set problem in unit-disk graph, which is known to be NP-hard [57]. For the unweighted unit-disk graph on point sets with unbounded density, the clustering technique will give us 3-approximate shortest distance [59]. So the

[4]Define $\hat{f}$ to be (ρ, β)-approximate of f if $\hat{f} \leq \rho f + \beta$. ρ is called the multiplicative error and β is called the additive error.

algorithm in the previous section gives a set S' so that the maximum distance from S to S' is at most $6+\varepsilon$ times the minimum possible. If we allow additive error, by the results above, we can get a $(2+\varepsilon, 4+\varepsilon)$-approximate k-center. The running time is $O(\sqrt{n \log n}/\varepsilon^3)$. Similarly we can get $O(1)$-approximate k-center in time $O(n \log^3 n)$.

7.6 Open problems

The most notable open problem is the gap between $\Omega(n)$ and $O(n \log n)$ on the number of pairs needed in the plane. Also, the time bound for $(1+\varepsilon)$-approximation is still about $O(n\sqrt{n})$ due to the lack of efficient method for computing $(1+\varepsilon)$-approximate shortest distance between $O(n)$ pairs of points. Any improvement to the algorithm for that problem will immediately lead to the improvement to all the $(1+\varepsilon)$-approximate algorithms presented in this Chapter.

Chapter 8

Kinetic Spanners

8.1 Introduction

A graph G' is a *spanner* of a graph G if G' is a subgraph of G and $\pi_{G'}(p,q) \leq s \cdot \pi_G(p,q)$ for some constant s and for all pairs of nodes p and q, where $\pi_G(p,q)$ is the shortest path distance between p and q in the graph G. The factor s is called the *stretch factor* of G'. G' is also called an s-spanner of G. If G is the complete graph of a set of n points S in a metric space $(S, |\cdot|)$ with $\pi_G(p,q) = |pq|$, we call G' an s-spanner of the metric $(S, |\cdot|)$ — in our case we will be focusing on collections of points in $\mathbb{R}^d$. A spanner provides an efficient encoding of distance information in a graph-theoretic setting, or of proximity information in a geometric setting.

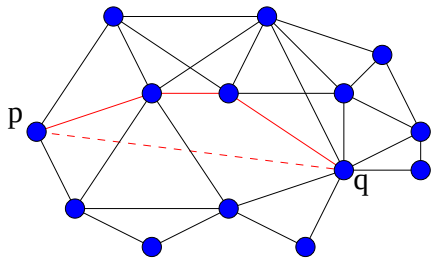


Figure 8.1. A spanner on a set of points. The shortest path length between p and q is at most s times the Euclidean distance $|pq|$.

There is a vast literature on spanners [15, 52, 122]. Extant spanner constructions

are all static, based on sequential centralized algorithms. Our interest is in devising spanner data structures for points in a Euclidean space that can be maintained efficiently under dynamic insertion/deletion and continuous motion of the point set. Maintaining proximity information is crucial in many physical simulations, as most forces in nature are short range — things interact when they are near. This is true across all scales, from smoothed particle hydrodynamics in astronomy to molecular dynamics in biology. Our spanner structure, which we call a DefSpanner (or deformable spanner), is built on realistic point sets, i.e., point sets with bounded aspect ratio, i.e., ones where the ratio of the maximum pairwise distance and minimum pairwise distance is polynomially bounded by the number of points.

We propose a new deformable $(1+\varepsilon)$-spanner (given any $\varepsilon > 0$) for a set of n points in $\mathbb{R}^d$ under the Euclidean metric. We study the properties and applications of such a spanner. Our spanner has $O(n/\varepsilon^d)$ edges. If the point set has bounded aspect ratio, our spanner will have low degree and low weight, i.e., the maximum number of spanner edges incident to any point is $O(\log \alpha/\varepsilon^d)$. Furthermore, the DefSpanner enjoys the additional advantage that it can be updated efficiently under both dynamic and kinetic situations. Most previously proposed algorithms to compute $(1+\varepsilon)$-spanners are all sequential and efficient updates are difficult. To be specific, in the DefSpanner, insertion or deletion of any point can be done in time $O(\log \alpha/\varepsilon^d)$ in the worst case. When the points move continuously, we study the kinetic properties of our spanner in the Kinetic Data Structure (KDS for short) framework [28, 71]. The kinetic spanner changes only at discrete times and has all the properties of a good KDS: efficiency, locality, responsiveness and compactness. To our knowledge, this is the the first kinetic spanner data structure. Under the assumption of bounded aspect ratio, $\log \alpha$ can be replaced by $\log n$ in all the above bounds.

It turns out that our DefSpanner construction only depends on a packing property of Euclidean metrics: a ball with radius r can be covered by at most a constant number of balls of radius $r/2$. Therefore, the spanner, as well as the applications on all the proximity problems, can be directly extended to the metrics with such properties, which were defined as *metrics with constant doubling dimension* [13]. Independently,

Krauthgamer and Lee [98] proposed a quite similar hierarchical structure for proximity search in such metrics. They use the hierarchical structure to answer $(1+\varepsilon)$ nearest neighbor search in $O(\log \alpha + (1/\varepsilon)^{O(1)})$ time. Their data structure can be maintained so that each insertion and deletion takes $O(\log \alpha \operatorname{loglog} \alpha)$ time. That work, however, does not address any of the difficult maintenance under motion issues that form the focus of our work.

In addition to basic proximity maintenance, our DefSpanner can be used to give efficient kinetic algorithms for a lot of related problems. For example, we can maintain the closest pair of points and thus have a collision detection mechanism. We can maintain the near neighbors of all points (to within a specified distance), and perform approximate nearest neighbor searches (aka get the functionality of approximate Voronoi diagrams). We also get the first kinetic algorithms for maintaining well-separated pair decompositions and approximate k-centers of our point set. So this one simple combinatorial structure provides a "one-stop shopping" mechanism for a wide variety of proximity problems and queries on moving points.

We now discuss in greater detail the specific problems we address and the contributions that the DefSpanner data structure makes.

Closest pair, collision detection. The crucial insight here is that before a pair of nodes can collide, any spanner must put an edge between them. Otherwise the bounded spanning ratio condition would be violated. Note also that the closest pair of elements must have an edge in any $(1+\varepsilon)$-spanner, if $\varepsilon < 1$. Thus the DefSpanner naturally contains the information we need for closest pair maintenance and collision detection. Compared to other collision detection structures, the DefSpanner is much lighter weight. It is a purely combinatorial structure (edges, specified by pairs of points) of size $O(n/\varepsilon^d)$ that allows collision detection in $O(n)$ time.

All near neighbors search. The all near neighbors search problem is to find all the pairs of points with distance less than a given value r, i.e., for each point, we must return the list of points inside the ball with radius r. With the DefSpanner, to find all the points within a certain distance r from a point p, we start from p and

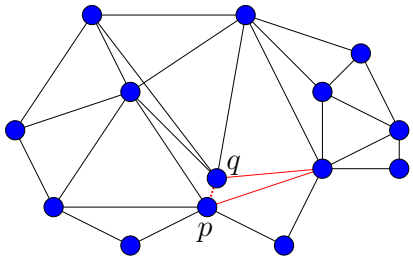


Figure 8.2. Before a collision happens, a spanner edge must connect the colliding elements.

follow the spanner edges until the total length is greater than $s \cdot r$. We then filter the points thus collected and keep only those that are within distance r of p. We can show that the cost of this is $O(n + k)$, where k is the number pairs in the answer set — thus the method is output sensitive and the cost of filtering does not dominate.

Well-separated pair decompositions. In fact, many of the spanner constructions for points in Euclidean space use the well-separated pair decomposition [19, 15, 116, 103] as a tool. The basic idea is this: the graph defined by taking an arbitrary edge connecting each s-well-separated pair must be a spanner [37]. The spanning ratio can be made arbitrarily close to 1 as long as we choose a large enough s. Here we show that the other direction is also true: the DefSpanner we build can be used to generate an s-well-separated pair decomposition, for any positive s — it suffices to take $\varepsilon = 4/s$ in the spanner construction. The size of the WSPD is linear, which matches the bound by Callahan and Kosaraju [40]. Since the spanner can be maintained in dynamic and kinetic settings, the well-separated pair decomposition can also be maintained efficiently for a set of moving points.

$(1+\varepsilon)$-nearest neighbor query/approximate Voronoi diagram. The $(1+\varepsilon)$-spanner we propose can be used to output the approximate nearest neighbor of any point $p \in \mathbb{R}^d$ with respect to the point set S, in time $O(\log n/\varepsilon^d)$. There has been a lot of work on data structures to answer approximate nearest neighbor queries quickly [81, 16, 18, 17]. However, they all try to minimize the storage or query cost and do not consider points in motion.

k-centers. The usual algorithms to compute approximate k-center are of a greedy nature [55, 66], and thus not easy to kinetize. Here we show how to compute an 8-approximate k-center by using the DefSpanner. Furthermore, we are the first to give a kinetic approximate k-center as the points move.

Eulerian vs. Lagrangian formulations. As we noted above, voxel grids are commonly used in molecular dynamics to maintain neighbor lists. Similarly, classical n-body codes use oct-trees to derive the well-separated pair decompositions over which aggregate (fast multipole) potentials are computed. These are both Eulerian approaches in the sense of computational mechanics, i.e. data structures built on partitions of the ambient space of the problem into elements. In contrast, the spanner is a Lagrangian data structure built on a particle formulation and affixed to the object itself, as opposed to its ambient space. The Lagrangian approach has clear advantages when a body experiences large scale motion but modest internal deformation, as a structure affixed to such a body undergoes only small changes. In contrast, Eulerian approaches need to spend a lot of time moving the simulation particles across element boundaries.

8.1.1 Summary

We summarize the results below. All the algorithms/data structures are deterministic and we consider the worst-case behavior. For n points in $\mathbb{R}^d$, we have,

- A $(1+\varepsilon)$-spanner with $O(n/\varepsilon^d)$ edges;
- A linear-size structure for finding all near neighbors in time $(k+n)$, where k is the size of the output;
- A linear-size $(1/\varepsilon)$-well-separated pair decomposition;
- A linear-size structure for $(1+\varepsilon)$-nearest neighbor queries in $O(\log n/\varepsilon^d)$ time;
- A linear-size data structure for closest pair and collision detection;
- An 8-approximate k-center, for any $0 < k \leq n$.

Furthermore, if the point set also has bounded aspect ratio, we have efficient kinetic and dynamic maintenance for the spanner so that each operation takes $O(\log n/\varepsilon^d)$ time. The kinetic data structures for maintaining the $(1+\varepsilon)$-spanner, the $(1/\varepsilon)$-well-separated pair decomposition, the 8-approximate k-center, have the four desirable properties of efficiency, compactness, locality, and responsiveness.

8.2 The deformable spanner

In this paper we focus on a set S of points in the Euclidean space $\mathbb{R}^d$. The *aspect ratio* of S, defined by the ratio of the maximum pairwise distance and minimum pairwise distance, is denoted as α. Without loss of generality, we assume that the closest pair of points has distance 1, so the furthest pair of S has distance α.

8.2.1 Spanner definition

The *discrete centers* with radius r for point set S is defined as a maximal subset $S' \subseteq S$ such that any two centers are of a distance at least r away, and such that the balls with radius r centered at the discrete centers contain all the points of S. Notice that the set of discrete centers is not unique.

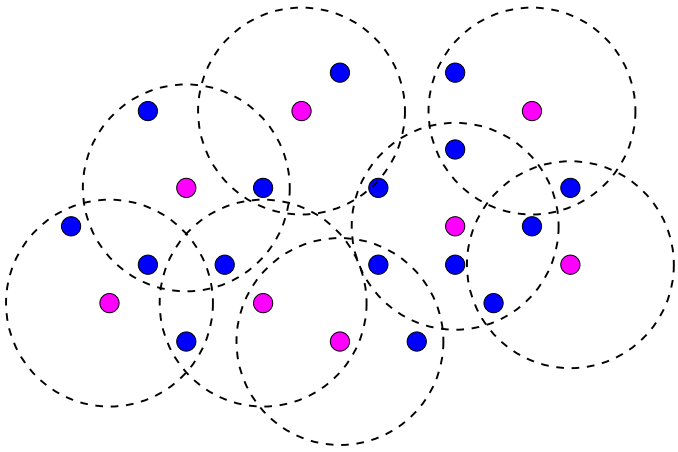


Figure 8.3. A set of discrete centers is drawn in purple.

We construct a hierarchy of discrete centers such that S_0 is the original point set

S and S_i is a set of discrete centers of S_{i-1} with radius 2^i, for $i > 0$. Intuitively the hierarchical discrete centers are samplings of the point set at different spatial scales.

The DefSpanner G on S is constructed as follows: we first construct the hierarchy of discrete centers S_i then add edges of length no more than $c \cdot 2^i$ between points in S_i to the graph G, where $c = 4 + 16/\varepsilon$. These edges connect each center to other centers in the same level whoses distances are comparable to the radius at that level. As pointed out later in Lemma 8.2.1(3), the edges also connect each center to the points (or centers) it covers in the lower level.

We use the following notations throughout the paper. Since a point p may appear in many levels in the hierarchy, when it is not clear, we use $p^{(i)}$ to denote the point p in level S_i. A center q in S_i is said to *cover* a point p in S_{i-1} if $|pq| \leq 2^i$. A point p in S_{i-1} may be covered by many centers in S_i. We denote $P(p)$ one of those centers and call it the parent of p. The choice of $P(p)$ is arbitrary but fixed. We also call p a child of $P(p)$. We recursively define $P^{j-i}(p)$ as the ancestor in level S_j of p by $P^0(p) = p, P^{j-i}(p) = P(P^{j-i-1}(p))$, and call $C_{i-1}(p) = \{q \in S_{i-1} | P(q) = p\}$, the set of children of p in S_{i-1}. We denote $N_i(p) = \{q \in S_i \mid |pq| \leq c \cdot 2^i\}$, the set of neighbors of p in S_i.

8.2.2 Spanner property

We first prove some properties about the discrete center hierarchy and the spanner.

Lemma 8.2.1. *1. $S_i \subseteq S_{i-1}$.*

2. *For any two points $p, q \in S_i$, $|pq| \geq 2^i$.*

3. *If $q \in C_i(p)$ and $q \neq p$, then $q \in N_i(p)$, i.e. there is an edge from each point q to its parent.*

4. *The hierarchy has at most $\lceil \log_2 \alpha \rceil$ levels.*

5. *For any point $p \in S_0$, its ancestor $P^i(p) \in S_i$ is of a distance at most 2^{i+1} away from p.*

Proof: The first three claims are obvious. For the fourth claim, an i-th level center $p \in S_i$ has radius 2^i. So if $2^i \geq \alpha$, the ball centered at p contains all the points in S. Therefore the height of the hierarchy h is at most $\lceil \log_2 \alpha \rceil$. The last claim is because there is a path from $p \in S_0$ to $P^i(p)$ with total length of at most $2 + 2^2 + \cdots + 2^i$. □

We are now ready to prove that G is a spanner.

Theorem 8.2.2. *G is a $(1+\varepsilon)$-spanner.*

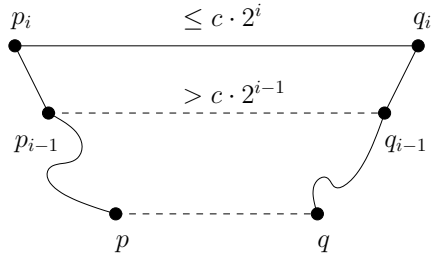


Figure 8.4. There exists a path in G between any two points p and q with length at most $(1+\varepsilon)|pq|$.

Proof: For a pair of points $p, q \in S_0$ we find the smallest level i so that there is an edge between their i-th parents $P^i(p)$ and $P^i(q)$. Denote $p_i = P^i(p), q_i = P^i(q), p_{i-1} = P^{i-1}(p), q_{i-1} = P^{i-1}(q)$. To prove that G is a spanner, we show that the path connecting p, q via p_i, q_i has length at most $(1+\varepsilon)|pq|$.

First, we have that $|p_i q_i| \leq c \cdot 2^i$ and $|p_{i-1} q_{i-1}| > c \cdot 2^{i-1}$. By Lemma 8.2.1, $|pp_{i-1}| \leq 2^i$, $|qq_{i-1}| \leq 2^i$. So $|pq| \geq |p_{i-1}q_{i-1}| - |pp_{i-1}| - |qq_{i-1}| > (c-4) \cdot 2^{i-1}$. Also the length of the path that connect p, q via p_i, q_i is at most $2^{i+1} + |p_i q_i| + 2^{i+1} \leq 8 \cdot 2^i + |pq| \leq (1 + 16/(c-4))|pq| = (1+\varepsilon)|pq|$. This proves that G is a $(1+\varepsilon)$-spanner. □

8.2.3 Size of the spanner

By a standard packing argument, we have,

Lemma 8.2.3. *Each point in S_i covers at most 5^d points in S_{i-1}.*

Lemma 8.2.4. *The number of edges any point $p \in S_i$ has with other points of S_i is at most $(1+2c)^d - 1$.*

Note that the bound in Lemma 8.2.3 could be improved. A more careful analysis, see Sullivan [147], shows that in $\mathbb{R}^2$, the maximum number of children is 19, and in $\mathbb{R}^3$, 87. In higher dimension, the number of children is at most $O(2.641^d)$.

Lemma 8.2.5. *The maximum degree of G is $(1+2c)^d \lceil \log_2 \alpha \rceil$.*

Proof: It follows from Lemma 8.2.4 and Lemma 8.2.1 (4). □

Lemma 8.2.6. *The total number of edges in G is less than $2(1+2c)^d \cdot n$.*

Proof: Note that if G is a DefSpanner and p is a point in G that does not have any children, then removing p and all edges incident on p from G gives us another DefSpanner G' with one less vertex. The lemma follows if we can show that we can always find a childless point p in G that is incident to at most $2(1+2c)^d$ edges.

Let p and q be the closest pair of points in G and let $k = \lfloor \log_2 |pq| \rfloor$. As p and q cannot be both in S_{k+1}, we assume further that p is not in S_{k+1}. Since p is at least 2^k apart from all points, it does not have any children in level $k-1$ or below, and thus it is childless. By a packing argument similar to that in Lemma 8.2.4, p is incident to at most $(1+2c/2^j)^d - 1$ edges in S_{k-j}, for all $0 \leq j \leq \log_2 c$. The total number of edges incident on p is thus at most $\sum_{j=0}^{\log_2 c} \left((1+2c/2^j)^d - 1\right) \leq 2(1+2c)^d$. □

To summarize, we have,

Theorem 8.2.7. *For a set of n points in $\mathbb{R}^d$ with aspect ratio α, we can construct a $(1+\varepsilon)$-spanner G so that the total number of edges is $O(n/\varepsilon^d)$ and the maximum degree of G is $O(\log_2 \alpha/\varepsilon^d)$.*

Remark. Notice that the hierarchy has at most $\lceil \log_2 \alpha \rceil$ levels. We can replace the base with any number greater than 1. Specifically if we choose $\beta > 1$, we can build the hierarchy so that for any two points p, q in S_i, $|pq| \geq \beta^i$. The hierarchy has at most $\lceil \log_\beta \alpha \rceil$ levels. Similar to Theorem 8.2.2, we can show that the graph constructed is a $(1+\epsilon)$ spanner when $c = \max\left(\beta, \frac{4\beta^2}{(\beta-1)\varepsilon} + \frac{2\beta}{\beta-1}\right)$.

8.3 Construction and dynamic maintenance

The previous section defines a set of properties such that a graph with those properties is a spanner. In this section we show that we can efficiently construct the spanner in $O(n \log_2 \alpha)$ time, where n is the number of points and α is the aspect ratio of the point set. We also show that we can dynamically insert or remove a point from our hierarchy, at a cost of $O(\log_2 \alpha)$ for each operation. In practical settings where α is a polynomial function of n, the construction of the hierarchy is $O(n \log_2 n)$, and dynamic update operations are done in $O(\log_2 n)$ time each.

To describe the dynamic maintenance of the spanner, we adopt a different setting. We assume that the aspect ratio α is always bounded by a polynomial of the number of points. However, as the points are inserted and deleted, the minimum separation of the point set may change. Therefore we keep virtually a set of points S_i, $-\infty < i < \infty$, such that S_i is a set of discrete centers of S_{i-1} with radius 2^i. Since the aspect ratio is bounded, there exist m and M such that there are spanner edges only on S_i, $m \leq i \leq M$, $M - m = O(\log n)$. We refer to S_m and S_M the bottom and the top of the hierarchy respectively. For each point p other than the root of the hierarchy, we store the maximum number M_p such that level S_{M_p} contains p, and store its parent $P(p^{(M_p)})$. We also store the minimum number m_p such that p has a neighbor in S_{m_p}, non-empty lists of neighbors of p in each of the levels between S_{m_p} and S_{M_p}, and non-empty lists of children of p in all levels below S_{M_p}. We also store the value of m and M for the hierarchy. Notice that we can always scale the point set so that the minimum separation is 1 and thus return to the previous setup.

We begin with the following simple yet crucial observation which is used repeatedly in this section. Basically if there is an edge between two nodes then there is an edge between their parents:

Lemma 8.3.1. *If $q \in N_i(p)$ then $P(q) \in N_{i+1}(P(p))$.*

Proof: If $q \in N_i(p)$, $|pq| \leq c \cdot 2^i$. Thus $|P(p)P(q)| \leq |P(p)p| + |pq| + |qP(q)| \leq (c+4) \cdot 2^i \leq c \cdot 2^{i+1}$. □

8.3.1 Construction

We construct the hierarchy incrementally by inserting points one by one. Suppose that we already have a hierarchy of $n-1$ points. The n-th point p is inserted as following. We first pretend that p appears in all levels of the hierarchy and insert p into the hierarchy from the top level to the bottom level. $p^{(i)}$'s parent is $p^{(i+1)}$. From Lemma 8.3.1, p only connects to nodes on level i whose parents on level $i+1$ have already been connected to p. So we compute $N_i(p)$ in each level i in the hierarchy by checking the distance from p to all its "cousins", i.e., the children of the neighbors of its parent. We stop if p does not have any neighbor. If p has a neighbor in the bottom level, we check whether p has any neighbors in even lower levels and if necessary, decrease m and extend the hierarchy downward. Intuitively, in this step we do point location from top down by using Lemma 8.3.1 and connect edges no longer than $c \cdot 2^i$ to p on each level i.

We then traverse the hierarchy from the bottom up and clean up the hierarchy of discrete centers. We find the highest level S_i and the point $q \in N_i(p)$ such that $|pq| < 2^i$. We set the parent of p in S_{i-1} to be q, and remove p from all levels S_i and above. If p still remains in the top level, we increase M and extend the hierarchy upward.

Note that we are making two passes through the hierarchy. In each level in each pass, the work is at most $5^d(1+2c)^d = O(1/\varepsilon^d)$, and thus the cost of one insertion is $O(h/\varepsilon^d)$, and the total cost of the construction is $O(nh/\varepsilon^d)$, where $h = O(\log_2 \alpha)$ is the height of the hierarchy.

8.3.2 Dynamic updates

From the previous subsection, it is clear that we can insert points into the hierarchy at the cost of $O(\log_2 \alpha/\varepsilon^d)$ each. In this section, we show that points can be removed from the hierarchy, again at the cost of $O(\log_2 \alpha/\varepsilon^d)$ each.

To remove a point p from the hierarchy, we remove p from bottom up. If p has no children (except itself), we can simply remove p and all edges incident on p in each level. If p has children, its children would become *orphans*, and we need to find new

parents for them before we can remove p. We assume $q^{(i)}$ is a child of $p^{(i+1)}$, $q \neq p$. From (3) in Lemma 8.2.1, $q^{(i)}$ is a neighbor of $p^{(i)}$ on level i.

From Lemma 8.3.1, we know the parent of $q^{(i)}$ must be a neighbor of the parent of $p^{(i)}$, i.e., $p^{(i+1)}$. If there is a neighbor w of $p^{(i+1)}$ that covers $q^{(i)}$, we set $q^{(i)}$'s parent to be w and we are done. If not $q^{(i)}$ is not covered by any centers on level $i+1$, and thus it must be inserted into level $i+1$. Notice that $q^{(i+1)}$ is a neighbor of $p^{(i+1)}$, we can recursively either find a parent for $q^{(i+1)}$ or promote q further up. The neighbors of q can then be computed from top down in a way similar to point insertion in previous subsection.

Note that the cost of raising a child of p up one level is $O(1/\varepsilon^d)$, and as the child may end up in the top level, the cost of fixing a child of p is $O(\log_2 \alpha/\varepsilon^d)$. Since p could appear in $O(\log_2 \alpha)$ levels and has $O(\log_2 \alpha)$ children, removing p may cost $O(\log^2 \alpha/\varepsilon^d)$. However notice that for any level S_i, all children of p on or below the level are inside a disk of radius $2 \cdot 2^i$, and the minimum separation in S_i is 2^i. By a packing argument, at most $O(1)$ of the children can end up being in S_i. The total cost of removing a point is thus $O(\log_2 \alpha/\varepsilon^d)$.

Theorem 8.3.2. *Dynamic insertion and deletion of points in the spanner takes $O(\log_2 \alpha/\varepsilon^d)$ each, α is the aspect ratio. The spanner can be constructed in time $O(n \log_2 \alpha/\varepsilon^d)$.*

8.4 KDS maintenance

To maintain the spanner G in the KDS framework, we need to maintain the discrete centers hierarchy and the edges between them. First, we keep the neighborhood information of each node p. We have three kinds of certificates for this purpose. A *parent-child certificate* guarantees that a child p is within distance 2^{i+1} from its parent in level $i+1$. An *edge certificate* guarantees that a neighbor q of p at level i is within distance $c \cdot 2^i$. A *separation certificate* guarantees that a neighbor q of p at level i is of distance 2^i away. These three certificates guarantee the validity of the discrete centers hierarchy and also detect when the near neighbors move further away. However, the

more difficult part is to detect when two far away points move close to each other for the first time. The key observation on the spanner hierarchy is that before two points can become neighbors at some level i, their parents are already neighbors at level $i+1$, as shown in Lemma 8.3.1. Therefore we only need to keep track of the *potential neighbors* of a point p, which are the "cousins" of p, i.e., the centers that are not neighbors of p but their parents are $P(p)$'s neighbors. A fourth certificate, *potential neighbor certificate*, guarantees that a potential neighbor of p at level i is of distance $c \cdot 2^i$ further away. All certificates are simple distance comparisons among pairs of points. To summarize, the four kinds of certificates make sure that for each center p in level i, the values $P(p)$ (if $P(p) \neq p$), $N_i(p)$, and $C_{i-1}(p)$ we maintain are valid. The failure of four types of certificates generates four types of events, which are discussed separately as follows. Let p be a point in level i:

1. **Addition of a spanner edge.** When a potential neighbor certificate fails, i.e., a potential neighbor q of p comes within a distance $c \cdot 2^i$ of p, we add an edge between p and q, making q a neighbor of p. We also update the list of potential neighbors of the children of p and q.

2. **Deletion of a spanner edge.** When an edge certificate fails, i.e., a neighbor q of p moves such that it is further than $c \cdot 2^i$ from p, we drop the edge between p and q, making them potential neighbors. We also update the list of potential neighbors of the children of p and q.

3. **Promotion of a node.** When a parent-child certificate fails, i.e., $q = P(p)$ no longer covers p, $|pq| > 2^{i+1}$. p becomes an orphan, and we need to find a new parent for p or promote it into higher levels. We deal with orphans the same way as in dynamic updates. We then update the potential neighbors of p in level i and above.

4. **Demotion of a node.** When a separation certificate fails, i.e., a neighbor q of p comes within a distance of 2^i, we need to remove one of the two points from level i. Assume without lost of generality that p is not in level $i+1$. We *demote*

p, i.e., removing p from level i. Each former child t of p in level $i-1$ becomes an orphan, and we deal with each of them as in the previous event.

The number of certificates for a point in any level it participates is $O(1/\varepsilon^d)$, and thus the total number of certificates is $O(n/\varepsilon^d)$, and the number of certificates associated with any point is $O(\log_2 \alpha/\varepsilon^d)$. Assuming that the motion preserves the bounded aspect ratio α, the total number of events in maintaining the spanner under pseudo-algebraic motion is bounded by $O(n^2 \log_2 \alpha)$ since an event only happens when the distance between two points becomes either 2^i or $c \cdot 2^i$ for $i = 0, \cdots, \log_2 \alpha$.

Note that both the dynamic and kinetic maintenance can also be done exactly in the same way for spanners with hierarchy expansion ratio $\beta > 1$ and $c > \max(\beta, 2\beta/(\beta-1))$.

8.4.1 Quality of the kinetic spanner

As pointed out in the previous subsection, our kinetic spanner has linear number of certificates, and that each point participates in at most $O(\log_2 n/\varepsilon^d)$ certificates. It is also easy to see that the cost to repair the KDS when a certificate fails is either $O(1)$ or $O(\log_2 n/\varepsilon^d)$. Our spanner KDS is thus compact, responsive, and local. As for the efficiency, we first have the following result:

Lemma 8.4.1. *There exists a set of n points so that any linear-size c-spanner has to change $\Omega(n^2/c^2)$ times.*

Proof: We consider a necklace of balls in the plane. The necklace consists of 3 segments. For the two segments close to the ends, each contains n/c bumps, where each bump has height c and the distance along the necklace between adjacent bumps is $2c$. The two segments are connected by a bent segment with $2n$ balls. The total number of balls in the necklace is $10n$. The top bumped segment is moving linearly towards the left; the bottom segment remains static. The balls on the middle segment moves accordingly to keep the necklace connected. Figure 8.5 shows the configuration of the necklace at the starting and ending point.

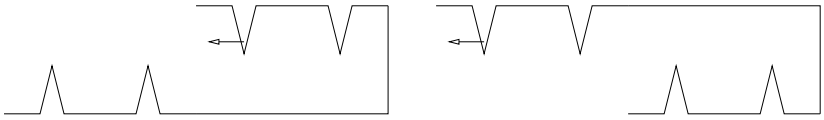


Figure 8.5. Motion of the points.

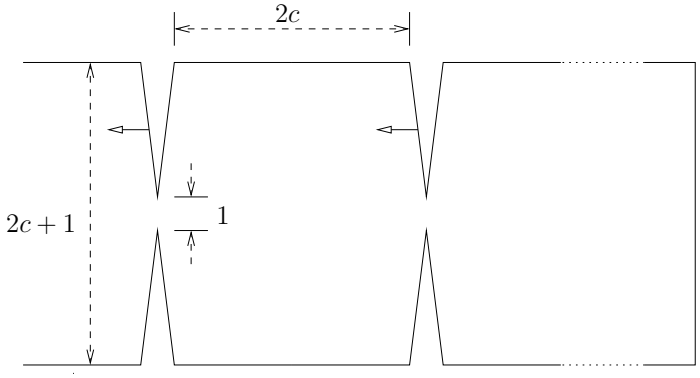


Figure 8.6. Lower bound $\Omega(n^2/c^2)$ for the changes of any linear-size spanner.

Consider the time when a top bump is directly on top of a bottom bump, so that the distance between their peaks is 1. See Figure 8.6. For any c-spanner, there must be a path connecting the two peaks with length no more than c. Therefore there must be an edge between some point in the top bump and some point in the bottom bump, since otherwise any path between the two peaks will be longer than c. So the total number of edges in the c-spanner that ever appear during the motion is at least $\Omega(n^2/c^2)$. Since the spanner starts with $O(n)$ edges. there must be at least $\Omega(n^2/c^2)$ changes of any linear-size c-spanner. □

As the number of events that our spanner KDS has to handle is $O(n^2 \log_2 \alpha) = O(n^2 \log_2 n)$, we have thus obtained:

Theorem 8.4.2. *The kinetic spanner is efficient, responsive, local and compact. Specifically, the total number of events in maintaining G is $O(n^2 \log_2 n)$ under pseudo-algebraic motion. Each event can be updated in $O(\log_2 n/\varepsilon^d)$ time. A flight-plan change can be handled in $O(\log_2 n/\varepsilon^d)$ time. Each point is involved in at most $O(\log_2 n/\varepsilon^d)$ certificates.*

8.5 Applications

8.5.1 Spanners and well-separated pair decompositions

We show by the following theorem that the spanner implies a linear size well-separated pair decomposition.

Lemma 8.5.1. *The spanner can be turned into an s-well-separated pair decomposition, so that $s = c/4 - 1 = 4/\varepsilon$. The size of the WSPD is $O(n/\varepsilon^d)$.*

Proof: For each node p_i in the spanner, we denote P_i be the set of all decedents of p_i and p_i itself. Now we consider the set C of pairs (P_i, Q_i) where p_i and q_i are not connected by an edge in some level i, but their parents in level $i+1$ are connected by an edge. $|p_i q_i| > c \cdot 2^i$. Note that all nodes in P_i (or Q_i) are within a distance of 2^{i+1} from p_i (or q_i), and thus, the distance between P_i and Q_i is at least $(c-4)2^i$. The diameter of P_i (or Q_i) is at most 2^{i+2}. Thus P_i and Q_i is s-separated, where $s = (c-4)/4$. Note that each pair of points in the hierarchy is covered by one and exactly one pair (P_i, Q_i) in C. It follows that C is an s-well-separated pair decomposition. The number of pairs in C is $O(n/\varepsilon^d)$. □

Theorem 8.5.2. *The s-well-separated pair decomposition can be maintained by a KDS which is efficient, responsive, local and compact.*

Proof: We construct and maintain the spanner. By using the spanner as a supporting data structure, we maintain the well-separated pair decomposition implicitly by marking the pairs (P_i, Q_i) where p_i and q_i are not connected by an edge in some level i, but their parents in level $i+1$ are connected by an edge. They only change when the edges are inserted/deleted. So the total number of events is $O(n^2 \log n)$, as per Theorem 8.4.2. Upon request, a well-separated pair can be output in time proportional to the number of points it covers.

On the other hand, there exists a set of n points such that any linear-size c-well-separated pair decomposition has to change $\Omega(n^2/c^2)$ times. For the setting in Figure 8.6, there must be a well-separated pair that contains only the points of the upper

bump and lower bump. The total number of such pairs is $\Omega(n^2/c^2)$, so is the total number of changes. □

8.5.2 All near neighbors query

The near neighbors query for a set of points, i.e., for each point p, returning all the points within distance ℓ from p, has been studied extensively in computational geometry. A number of papers use spanners and their variants to answer near neighbors query in almost linear time [20, 48, 102, 135]. Specifically, on a spanner, we do a breadth-first search starting at p until the graph distance to p is greater than $s \cdot \ell$, where s is the stretch factor. Due to the spanning property, this guarantees that we find all the points within distance ℓ from p. Furthermore, we only check the pairs with distance at most $s \cdot \ell$. Notice that unlike the previous papers that focus only on static points, the DefSpanner can be maintained under motion, so the near neighbors query can be answer at any time during the movement of the points.

Before we bound the query cost of the algorithm, we first show that the number of pairs within distance $s \cdot \ell$ will not differ significantly with the number of pairs within distance ℓ. A similar result has been proved in [135]. The following theorem is more general with slightly better results and the proof is much simpler.

Theorem 8.5.3. *For a set S of points in $\mathbb{R}^d$, denote by $\chi(\ell)$ as the number of ordered pairs (p, q), $p, q \in S$ such that $|pq| \leq \ell$, then $\chi(s \cdot \ell) \leq 2(2s+3)^d \chi(\ell) + n(2s+3)^d/2$.*

Proof: We first select a set of discrete centers $S_{\ell/2}$ with radius $\ell/2$ from points S. We then assign a point q to a center p if $|pq| \leq \ell/2$. A point can be within distance $\ell/2$ of more than 1 centers. In this case, we assign it to one of them arbitrarily. Any point is assigned to one and only one discrete center. We say q is covered by p if p is the assigned center for q. The set of points covered by $p \in S_{\ell/2}$ is denoted by $C(p)$, and $N(p) = |C(p)|$. Note that any pair of points within the same $C(p)$ for any p are within a distance of ℓ of each other.

We consider the pairs of sets $C(p)$ and $C(q)$ such that the distance between $C(p)$ and $C(q)$ is at most $s \cdot \ell$. Clearly $|pq| \leq (s+1) \cdot \ell$, and thus each center participates

in at most $(2s+3)^d$ such pairs. Since all the pairs (p', q') with $\ell < |p'q'| \leq s \cdot \ell$ are covered by at least one pair. We try to charge the long 'inter-distances' to the short 'intra-distances'.

From the inequality $ab \leq a(a-1)+b(b-1)+1$ for all real values a and b, summing over all pairs of p and q such that the distance between $C(p)$ and $C(q)$ is at most $s \cdot \ell$, we obtain $\sum_{p,q} N(p)N(q) \leq 2(2s+3)^d \sum_p N(p)(N(p)-1)+n(2s+3)^d$, and thus $2\chi(s \cdot \ell) \leq 4(2s+3)^{2d}\chi(\ell) + n(2s+3)^d$. □

Theorem 8.5.4. *For a set S of points in $\mathbb{R}^d$, we organize the points into a structure of size $O(n)$ so that we can perform the near neighbors query, i.e., for each point p, find all the points within distance ℓ of p, in time $O(k+n)$, if the size of the output is k.*

Proof: As we described before, we traverse the s-spanner G by a breadth-first search and collect the pairs with distance at most $s \cdot \ell$ that include all pairs with distance no more than ℓ. We then filter out unnecessary pairs and only keep the pairs within distance ℓ. ¿From Theorem 8.5.3, $\chi(s\ell) \leq 2(2s+3)^d k + n(2s+3)^d/2$, where k is the size of the output. We choose s as a constant, the size of the spanner G is $O(n)$, from Theorem 4.2.6. □

We remark that this output sensitivity is not valid on a per point basis. Figure 8.7 shows an example situation where for point p the number of neighbors within distance $s \cdot \ell$ is not proportional to those within distance ℓ.

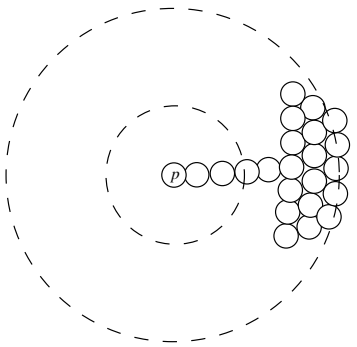


Figure 8.7. The number of neighbors of point p increases abruptly.

8.5.3 $(1+\varepsilon)$-nearest neighbor

An s-approximate nearest neighbor of a point $p \in \mathbb{R}^d$ with respect to a point set S is a point $q \in S$ such that $|pq| \leq s \cdot |pq^*|$, where q^* is the nearest neighbor of p. We first show that a $(1+\varepsilon)$ nearest neighbor is embedded in a $(1+\varepsilon)$ DefSpanner.

Lemma 8.5.5. *For a $(1+\varepsilon)$ DefSpanner on a set S of points in $\mathbb{R}^d$, we can perform the $(1+\varepsilon)$-nearest neighbors query in $O(\log \alpha/\varepsilon^d)$ time, i.e., given a point $p \in \mathbb{R}^d$, find a point q in S such that $|pq| \leq (1+\varepsilon)|pq^*|$, where q^* is the nearest neighbor of p.*

Proof: We construct the $(1+\varepsilon)$ DefSpanner G as before. Firstly, we do a fake insertion of p. Assume q is the direct neighbor of p in the spanner with the closest distance, $|pq| = x$, and q^* is the nearest neighbor of p. From the spanner property we know that $\pi_G(p, q^*) \leq (1+\varepsilon) \cdot |pq^*|$. On the other hand, since pq is the shortest edge attached with p in the graph G, then we must have $\pi_G(p, q^*) \geq |pq|$. This implies that $|pq| \leq (1+\varepsilon) \cdot |pq^*|$.

We find such a q, i.e., the closest neighbor of p on the spanner, as follows. Take the lowest level i where edge pq appears, $c \cdot 2^{i-1} < |pq| \leq c \cdot 2^i$. Then for the level $j \leq i-1$, there is no edge attached with point p. Otherwise that edge would have shorter distance than pq. Therefore for each point $p \in S$, we take the lowest level i where p has an edge in the spanner. We take the shortest edge pq among all the level i edges. q is the $(1+\varepsilon)$-approximate neighbor of p. The theorem then follows from Theorems 4.2.6 and 8.3.2. □

Furthermore, if we keep for each point its shortest edge in the spanner, we can get the $(1+\varepsilon)$-nearest neighbor of any point $p \in S$ by a single lookup. The maintenance of the spanner implies the maintenance of the $(1+\varepsilon)$-nearest neighbor information as well. So we have,

Theorem 8.5.6. *For a set S of points in $\mathbb{R}^d$, we can maintain a kinetic data structure of size $O(n/\varepsilon^d)$ that keeps the $(1+\varepsilon)$-nearest neighbor in S of any node $p \in S$. The structure is efficient, responsive, local and compact.*

Proof: All we need to prove is the efficiency of the KDS. The example in Lemma 8.4.1 shows that any linear-size structure maintaining the $(1+\varepsilon)$-approximate neighbor has to change $(n^2\varepsilon^2)$ times. □

So far we build a $(1+\varepsilon)$ DefSpanner to answer and maintain the $(1+\varepsilon)$-approximate nearest neighbor query for a specific ε. In fact, to answer the $(1+\varepsilon)$-approximate nearest neighbor query, we can decouple the dependency of the DefSpanner on the parameter ε by using a $O(1)$ DefSpanner as an auxiliary structure.

Theorem 8.5.7. *For a set S of points in $\mathbb{R}^d$, we can organize the n points into a structure of size $O(n)$ so that we can perform the $(1+\varepsilon)$-nearest neighbor query in $O(\log\alpha/\varepsilon^d)$ time, i.e., given a point $p\in\mathbb{R}^d$, find a point q in S such that $|pq|\leq(1+\varepsilon)|pq^*|$, where q^* is the nearest neighbor of p.*

Proof: Fix a constant $c>4$ and construct a DefSpanner using that constant. Given an $\varepsilon>0$ and a query point p, we let $t=2+4/\varepsilon$. To answer the $(1+\varepsilon)$ approximate nearest neighbor of p, we traverse the DefSpanner top down and keep track of the set $K_i=\{q\mid q\in S_i, |pq|<t\cdot 2^i\}$ as the level i decreases.

First of all, we notice that $|K_i|=O(t^d)$, for any i, since the points in S_i are at least distance 2^i apart. Secondly, we observe that for a point $q\in S$, if $|pq|<t\cdot 2^i$, then $|pP(q)|<t\cdot 2^{i+1}$. This is due to the triangular inequality: $|pP(q)|\leq|pq|+|qP(q)|<t\cdot 2^i+2^{i+1}\leq t\cdot 2^{i+1}$. Therefore K_i must be included in the set of the children of K_{i+1}. So we can construct K_i from K_{i+1} in $O(t^d)$ time. The total running time of such a traversal is bounded by $O(t^d\log\alpha)=O(\log\alpha/\varepsilon^d)$.

At the end of the traversal of the DefSpanner, let q be the point closest to p in K_0. We will show that q is a $(1+\varepsilon)$-nearest neighbor of p. Let $q^*\in S_0$ be the closest point to p among all points in the spanner. If q^* is in K_0, then clearly $|pq|=|pq^*|$, and we are done. If not, let j be such that $P^{j-1}(q^*)\notin K_{j-1}$ and $P^j(q^*)\in K_j$. By definition of q and Lemma 8.2.1, $|pq|\leq|pP^j(q^*)|\leq|pq^*|+2^{j+1}$. Since $|pq^*|\geq|pP^{j-1}(q^*)|-2^j>(t-2)\cdot 2^{j-1}$, $|pq|<(1+4/(t-2))|pq^*|=(1+\epsilon)|pq^*|$. The theorem is proved. □

We note that while we need $c>4$ in order to construct and maintain the DefSpanner, if we are only interested in static nearest neighbor queries, a DefSpanner

with $c > 2$ would be suffice, even though a DefSpanner may not be a spanner when $c \leq 4$.

8.5.4 Closest pair and collision detection

Theorem 8.5.8. *For a set S of points in $\mathbb{R}^d$, we have a structure of size $O(n)$ to output the closest pair of the point set.*

Proof: We construct the $(1+\varepsilon)$-spanner G as before. If we take $\varepsilon < 1$, then the closest pair pq must have an edge in G. Otherwise, since the shortest path between p, q in G contains at least 2 edges, each of them is longer than $|pq|$, $\pi_G(p,q) > 2|pq|$. This contradicts with the spanner property. To maintain the closest pair, we simply use a kinetic priority queue [71] to keep the shortest edge among all the spanner edges. □

8.5.5 k-center

For a set S of points in $\mathbb{R}^d$, we choose a set K of points, $K \subseteq S$, $|K| = k$, we assign all the points in S to the closest point in K. The k-center problem is to find a K such that the maximum radius of the k-center, $\max_{p \in S} \min_{q \in K} |pq|$, is minimized.

We take the lowest level i such that $|S_i| \leq k$. If $|S_i| = k$, then we take $K = S_i$. If $|S_i| < k$, we also add some (arbitrary) children of S_i to K so that $|K| = k$.

Lemma 8.5.9. *K is an 8-approximation of the optimal k-center.*

Proof: On level $i-1$ we have more than k points with distance at least 2^{i-1} pairwise apart. So in the optimal solution, at least two points in S_{i-1} are assigned to the same center. Thus the optimal k-center has maximum radius at least 2^{i-2}. But K has radius at most 2^{i+1}. So K is an 8-approximation. □

Theorem 8.5.10. *For a set S of points in $\mathbb{R}^d$, we can maintain an 8-approximate k-center. The KDS is efficient, responsive, local and compact.*

Proof: We first maintain a constant-spanner under motion. When the nodes in S_i changes, we update the approximate k-center as well. If a node $p \in S_i$ is deleted from level i, we add a children of S_i to K to keep $|K| = k$.

The only problem that needs to be clarified is when the number of centers at level $i-1$ becomes k or less. However, since $S_i \subseteq S_{i-1}$ and we take K to be S_i and some children of S_i, we thus smoothly switch from level i to level $i-1$. The other event is when $|S_i| = k+1$, we simply take S_{i+1}. Notice that $S_{i+1} \subseteq S_i \subseteq S_{i-1}$ so the update to K is $O(1)$ per event. And K is changed at most $O(n \log_2 \alpha)$ times.

In addition, there exists a situation and a certain k where the optimal k-center undergoes $\Omega(n^3)$ changes, as the example in [58]. In fact, that example shows any approximate k-center with approximation ratio < 1.5 has to change $\Omega(n^3)$ times. For general approximation factor c, an example shows an $\Omega(n^2)$ bound. The details are omitted. So the KDS to maintain 8-approximate k-center is efficient.

□

Remark Notice that the spanner actually gives the approximate solutions to a set of k-center problems with different k simultaneously. We can maintain t subsets $K_1, K_2, \cdots, K_t$ such that K_i is an 8-approximation of the optimal k_i-center, where $0 \leq k_i \leq n$. The update cost per event is $O(t + \log_2 n)$.

Chapter 9

Conclusion

As the state of the art, how to apply the ideas of *ad hoc* wireless networks on real-world applications still remains a big challenge. Besides the applications that ad hoc networks were proposed for, for example, environment monitoring and digital battlefields, the recent emergence of intelligent radio devices are likely to lead to the wide-spread emergence of *ad hoc* networks in the civilian world. Although this dissertation focuses mostly on the data structures and algorithms in *ad hoc* networks, we should note that there are many other issues involved in constructing such a network. Proper charging/pricing schemes need to be designed to motivate selfish parties to participate in the multi-hop routing [130, 50, 100, 12]. Security is another important and difficult problem in *ad hoc* networks, due to high dynamics, wireless link vulnerability, and the requirement of complete decentralization [118, 119].

As the ending words of this dissertation, the structures and algorithms we proposed are especially interesting in a philosophical point of view. These algorithms are perfect examples to show how globally optimal behaviors and structures "emerge" from simple local interactions – the exact philosophy of many other complex systems like the biological system or the social system. Controlled by efficient local rules, the structures evolve and adapt by themselves. The relationship between local individual interactions and the global behaviors of a complex network has been studied in many different disciplines, such as physics, sociology, cell biology and economics. So far the only way to study the autonomous systems in nature is to passively observe

— we can hardly control their development and evolvement. Ad hoc networks, as complex networks which people build and have fully control on, provide us a perfect test bed for distributed algorithms. We believe that the study of data structures and distributed algorithms along this line, is important and will find applications in many other complex systems as well.

Appendix A

A lower bound example

We show by an example that the algorithm GREEDY2 doesn't guarantee a constant load balancing ratio if the packets have variable sizes. Assume we have $3n$ nodes distributed on a line. The nodes are organized in three groups, $\{x_1, \cdots, x_n\}$, $\{y_1, \cdots, y_n\}$, $\{z_1, \cdots, z_n\}$. All the x_i's are visible to all the y_i's. All the y_i's are visible to z_i's. But no x_i is visible to z_i, as shown in Figure A.1. We have a set of n requests. The first request is $r_n = (x_n, z_n, 1)$, the next $n-1$ requests are $r_i = (x_i, z_i, 2)$, where $i = 1, 2, \cdots, n-1$. The optimum load balanced routing algorithm routes r_i though y_i. So the maximum load is 2. Now let's see how the GREEDY2 works here. The first

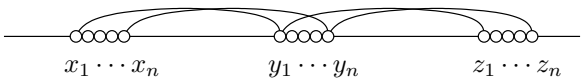


Figure A.1. The optimum load balanced routing algorithm. The maximum load is 2.

request $(x_n, z_n, 1)$ is going to be routed on y_n since y_n is the furthest node and all the current load of y_i are all 0. The next request $(x_1, z_1, 2)$ will be routed on y_{n-1} since y_{n-1} is the furthest node whose load is strictly smaller than the maximum load among all the y_i's. The third request $(x_2, z_2, 2)$ will be routed on y_n since now the maximum load is 2 and the load of y_n is 1 and is strictly smaller than the maximum load in x_2's right communication range. So the load on y_n is 3. So the next request $(x_3, z_3, 2)$ will be routed on y_{n-1}. After this point, the requests will be routed alternatively on y_n

and y_{n-1}. Thus the maximum load produced by GREEDY2 is $\lceil \frac{n-1}{2} \rceil \times 2 + 1$. The load balancing ratio of GREEDY2 in this case is $\Omega(n)$.

Appendix B

A lower bound example

We give an lower bound construction that shows any on-line c-short routing algorithm on a mesh is $\Omega(\log n)$ competitive compared to the optimal c-short routing algorithm. The example is similar to the one in [25].

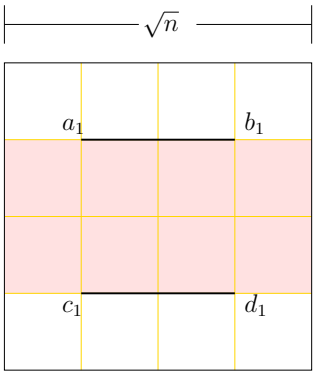


Figure B.1. Lower bound $\Omega(\log n)$ on the competitive ratio.

We assume a set of n nodes on a grid of size $\sqrt{n} \times \sqrt{n}$. Divide the square uniformly into 16 sub-squares. The adversary makes $\log n$ rounds of requests. The first set of requests with size 1 are from the nodes on the line segment a_1b_1 to the vertically corresponding nodes on c_1d_1. It's clear that any routing algorithm must use the nodes in the pink region. The adversary randomly chooses a pink sub-square S_2 and recurses.

For the optimal routing, according to how the random sub-square is selected, we

can route the first round requests by using nodes outside the sub-square S_2. Therefore, every node has load at most 1.

On the other hand, the average load on the pink sub-square S_2 is at least 1/2. The online algorithm has no idea which sub-square will be selected for the next round of requests. After $\log_2 n/4$ rounds, the node in the sub-square $S_{\log_2 n/4}$ has average load at least $\log_2 n/8$, i.e., one node will have load $\Omega(\log n)$ while the optimal routing can always spread out the load.

Appendix C

Relative neighborhood graph

Given n point set S, the relative neighborhood graph is defined as a graph G such that there is an edge between u, v if and only if there are no points inside the "cone", defined as the intersection of the two disks centered at u, v with radius $|uv|$.

Theorem C.0.11. *The relative neighborhood graph has degree at most* 6.

Proof: For any node u, we sort its edges counterclockwise. Then for any two adjacent edges uv and uw, we claim that the angle $\angle vuw$ is least $\pi/3$. Otherwise, assume that $\angle vuw < \pi/3$. We take the longer edge among uv and uw. Then w must be inside the cone defined by the intersection of two disks with radius $|uv|$ on points u, v, as shown in Figure C.1. This contradicts with the definition of the relative neighborhood

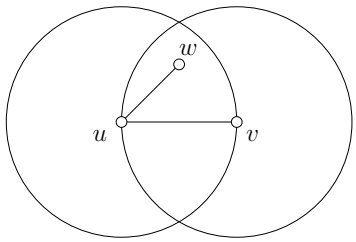


Figure C.1. For any two adjacent edges uv and uw, the angle $\angle vuw$ is least $\pi/3$.

graph. Thus the claim is proved. □

For a relative neighborhood graph G, we keep only the edges with length no longer than 1 and denote the graph G'. Now we claim that G' has the same connectivity with the unit disk graph $I(S)$.

Theorem C.0.12. *G' has the same connectivity with the unit disk graph $I(S)$.*

Proof: We assume that $I(S)$ is connected, otherwise we can study each connected component separately. First we show that the closest pair p, q must be connected in G'. This is due to the construction of the relative neighborhood graph and its restricted version G'.

Now we prove by induction. Assume that all the pairs of nodes with distance no more than ℓ are connected in G'. Now we take the shortest edge uv among those with length more than ℓ. If uv is in G', we are done. Otherwise if an edge uv in $I(S)$ is not in G', there must exist a node w inside the cone of node u, v. $|uw| \leq |uv|$, $|wv| \leq |uv|$. By induction hypothesis, we know that u, w and w, v must be connected by some paths in G'. This proves that u, v is connected in G'. The theorem follows. □

Bibliography

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. Annu. ACM Sympos. Principles Database Syst.*, 2000. 175–186.

[2] P. K. Agarwal, J. Basch, M. de Berg, L. J. Guibas, and J. Hershberger. Lower bounds for kinetic planar subdivisions. In *Proc. ACM Symposium on Computational Geometry*, pages 247–254, 1999.

[3] P. K. Agarwal, J. Basch, L. Guibas, J. Hershberger, and L. Zhang. Deformable free space tilings for kinetic collision detection. *International Journal of Robotics Research*, 21(3):179–197, 2002.

[4] P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete Comput. Geom.*, 6(5):407–422, 1991.

[5] P. K. Agarwal, D. Eppstein, L. J. Guibas, and M. Henzinger. Parametric and kinetic minimum spanning trees. In *Proc. 39th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 596–605, 1998.

[6] P. K. Agarwal, J. Erickson, and L. J. Guibas. Kinetic binary space partitions for intersecting segments and disjoint triangles. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 107–116, 1998.

[7] P. K. Agarwal, J. Gao, and L. J. Guibas. Kinetic medians and *kd*-trees. In *Proc. 10th Annu. European Sympos. Algorithms*, pages 5–16, 2002.

[8] P. K. Agarwal, L. J. Guibas, J. Hershberger, and E. Veach. Maintaining the extent of a moving point set. In *Proc. 5th Workshop Algorithms Data Struct.*, volume 1272 of *Lecture Notes Comput. Sci.*, pages 31–44. Springer-Verlag, 1997.

[9] P. K. Agarwal, L. J. Guibas, T. M. Murali, and J. S. Vitter. Cylindrical static and kinetic binary space partitions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 39–48, 1997.

[10] S. Albers. Better bounds for online scheduling. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 130–139, 1997.

[11] A. D. Amis, R. Prakash, T. H. P. Vuong, and D. T. Huynh. Max-Min D-cluster formation in wireless ad hoc networks. In *19th IEEE INFOCOM*, March 1999.

[12] L. Anderegg and S. Eidenbenz. Ad hoc-VCG: A truthful and cost-efficient routing protocol for mobile ad hoc networks with selfish agents. In *Proc. 9th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2003.

[13] R. K. Anupam Gupta and J. R. Lee. Bounded geometries, fractals, and low-distortion embeddings. In *Proc. 44th Symposium on Foundations of Computer Science (FOCS '03)*, pages 534–543, 2003.

[14] S. R. Arikati, D. Z. Chen, L. P. Chew, G. Das, M. H. M. Smid, and C. D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In J. Díaz and M. Serna, editors, *Proc. of 4th Annual European Symposium on Algorithms*, pages 514–528, 1996.

[15] S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: short, thin, and lanky. In *Proc. 27th ACM Symposium on Theory Computing*, pages 489–498, 1995.

[16] S. Arya and T. Malamatos. Linear-size approximate voronoi diagrams. In *Proc. of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 147–155, 2002.

[17] S. Arya, T. Malamatos, and D. M. Mount. Space-efficient approximate Voronoi diagrams. In *Proc. of the 34th ACM Symposium on Theory of Computing*, pages 721–730, 2002.

[18] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[19] S. Arya, D. M. Mount, and M. Smid. Randomized and deterministic algorithms for geometric spanners of small diameter. In *Proc. 35th IEEE Symposium on Foundations of Computer Science*, pages 703–712, 1994.

[20] S. Arya and M. Smid. Efficient construction of a bounded-degree spanner with low weight. *Algorithmica*, 17:33–54, 1997.

[21] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line machine scheduling with applications to load balancing and virtual circuit routing. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 623–631, 1993.

[22] Y. Azar. On-line load balancing. In A. Fiat and G. Woeginger, editors, *On-line Algorithms: The State of the Art*, pages 178–195. LNCS 1442, Springer, 1998.

[23] H. Badr and S. Podar. An optimal shortest-path routing policy for network computers with regular mesh-connected topologies. *IEEE Transactions on Computers*, 38(10):1362–1371, 1989.

[24] N. Bambos. Toward power–sensitive network architectures in wireless communications: Concepts, issues, and design aspects. *IEEE Personal Communications*, 5:50–59, 1998.

[25] Y. Bartal and S. Leonardi. On-line routing in all-optical networks. *Theoretical Computer Science*, 221(1–2):19–39, 1999.

[26] S. Basagni. Distributed clustering for ad hoc networks. In *Proc. 99' International Symp. on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, pages 310–315, June 1999.

[27] J. Basch. *Kinetic Data Structures*. PhD thesis, Stanford University, Stanford, CA, June 1999.

[28] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. *J. Alg.*, 31(1):1–28, 1999.

[29] J. Basch, L. J. Guibas, C. Silverstein, and L. Zhang. A practical evaluation of kinetic data structures. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 388–390, 1997.

[30] J. Basch, L. J. Guibas, and L. Zhang. Proximity problems on moving points. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 344–351, 1997.

[31] P. Basu, N. Khan, , and T. D. Little. A mobility based metric for clustering in mobile ad hoc networks. In *Proc. of IEEE ICDCS 2001 Workshop on Wireless Networks and Mobile Computing*, April 2001.

[32] C. Bettstetter and R. Krausser. Scenario-based stability analysis of the distributed mobility-adaptive clustering (DMAC) algorithm. In *Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 232–241. ACM Press, 2001.

[33] L. Blazević, L. Buttyán, S. Capkun, S. Giordano, H. Hubaux, and J. L. Boudec. Self-organization in mobile ad hoc networks: the approach of terminodes. *IEEE Communications Magazine*, pages 166–175, 2001.

[34] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[35] P. Bose, L. Devroye, W. Evans, and D. Kirkpatrick. On the spanning ratio of gabriel graphs and β-skeletons. In *Proceedings of the Latin American Theoretical INformatics (LATIN)*, pages 479–493, 2002.

[36] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *3rd Int. Workshop on Discrete Algorithms and methods for mobile computing and communications (DialM '99)*, pages 48–55, 1999.

[37] Callahan and Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, 1993.

[38] P. B. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *Proc. 34th IEEE Symposium on Foundations of Computer Science*, pages 332–340, 1993.

[39] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest-pair and n-body potential fields. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 263–272, 1995.

[40] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42:67–90, 1995.

[41] J.-H. Chang and L. Tassiulas. Energy conserving routing in wireless ad-hoc networks. In *Proc. INFOCOM*, pages 22–31, 2000.

[42] J.-H. Chang and L. Tassiulas. Fast approximation algorithms for maximum lifetime routing in wireless ad-hoc networks. In *Networking, LNCS 1815*, pages 702–713, 2000.

[43] M. Chatterjee, S. K. Das, and D. Turgut. WCA: A weighted clustering algorithm for mobile ad hoc networks. *Cluster Computing*, 5(2):193–204, 2002.

[44] G. Chen and I. Stojmenovic. Clustering and routing in mobile wireless networks. Technical Report TR-99-05, SITE, June 1999.

[45] C.-C. Chiang, H.-K. Wu, W. Liu, and M. Gerla. Routing in clustered multihop, mobile wireless networks with fading channel. In *Proc. IEEE Singapore International Conference on Networks (SICON'97)*, pages 197–211, April 1997.

[46] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE, Special Issue on Computational Geometry*, 80(9):1412–1434, September 1992.

[47] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[48] M. T. Dickerson, R. L. Drysdale, and J. R. Sack. Simple algorithms for enumerating interpoint distances and finding k nearest neighbors. *Internat. J. Comput. Geom. Appl.*, 2(3):221–239, 1992.

[49] D. P. Dobkin, S. J. Friedman, and K. J. Supowit. Delaunay graphs are almost as good as complete graphs. In *Proc. 28th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 20–26, 1987.

[50] S. Eidenbenz, V. S. A. Kumar, and S. Zust. Equilibria in topology control games for ad hoc networks. In *Proceedings of the 2003 joint workshop on Foundations of mobile computing*, pages 2–11. ACM Press, 2003.

[51] A. Ephremides, J. E. Wieselthier, and D. J. Baker. A design concept for reliable mobile radio networks with frequency hopping signaling. *Proc. of IEEE*, 75(1):56–73, Jan 1987.

[52] D. Eppstein. Spanning trees and spanners. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 425–461. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[53] D. Eppstein, G. L. Miller, and S.-H. Teng. A deterministic linear time algorithm for geometric separators and its applications. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 99–108, 1993.

[54] J. Erickson. Dense point sets have sparse Delaunay triangulations. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 125–134, 2002.

[55] T. Feder and D. H. Greene. Optimal algorithms for approximate clustering. In *Proc. 20th ACM Symposium on Theory of Computing*, pages 434–444, 1988.

[56] G. G. Finn. Routing and addressing problems in large metropolitan-scale internetworks. Technical Report ISU/RR-87-180, ISI, March 1987.

[57] R. J. Fowler, M. S. Paterson, and S. L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Inform. Process. Lett.*, 12(3):133–137, 1981.

[58] J. Gao, L. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Discrete mobile centers. *Discrete and Computational Geometry*, 30(1):45–65, 2003.

[59] J. Gao, L. J. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Geometric spanner for routing in mobile networks. In *Proceedings of the 2nd ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'01)*, pages 45–55, 2001.

[60] M. Gerla and J. Tsai. Multicluster, mobile, multimedia radio network. *ACM-Baltzer Journal of Wireless Networks*, 1(3):255–265, 1995.

[61] S. Giordano and I. Stojmenovic. *Position Based Ad Hoc Routes in Ad Hoc Networks*, chapter 16, pages 1–14. CRC Press, 2003.

[62] S. Giordano, I. Stojmenovic, and L. Blazevie. Position based routing algorithms for ad hoc networks: a taxonomy. *Ad Hoc Wireless Networking*, 2003.

[63] A. Goel and K. Munagala. Extending greedy multicast routing to delay sensitive applications. *Algorithmica*, 33(3):335–352, 2002.

[64] A. Goldsmith and S. Wicker, editors. *Special Issue: Energy-aware ad hoc wireless networks*, volume 9. IEEE Wireless Comm., Auguest 2002.

[65] J. Gomez, A. T. Campbell, M. Naghshineh, and C. Bisdikian. PARO: Supporting dynamic power controlled routing in wireless ad hoc networks. *ACM/Kluwer Journal on Wireless Networks (WINET)*, 9(5):443–460, September 2003.

[66] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoret. Comput. Sci.*, 38:293–306, 1985.

[67] S. Govindarajan, T. Lukovszki, A. Maheshwari, and N. Zeh. I/O efficient well-separated pair decomposition and its applications. In *Proc. 8th European Symposium on Algorithms*, pages 220–231, 2000.

[68] J. Gudmundsson, C. Levcopoulos, G. Narasimhan, and M. Smid. Approximate distance oracles for geometric graphs. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 828–837, 2002.

[69] L. Guibas, J. Hershberger, S. Suri, and L. Zhang. Kinetic connectivity for unit disks. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 331–340, 2000.

[70] L. Guibas, A. Nguyen, D. Russel, and L. Zhang. Collision detection for deforming necklaces. In *Proc. 18th ACM Symposium on Computational Geometry*, pages 33–42, 2002.

[71] L. J. Guibas. Kinetic data structures — a state of the art report. In P. K. Agarwal, L. E. Kavraki, and M. Mason, editors, *Proc. Workshop Algorithmic Found. Robot.*, pages 191–209. A. K. Peters, Wellesley, MA, 1998.

[72] N. Gupta and S. R. Das. Energy-aware on-demand routing for mobile ad hoc networks. In *Proc. IWDC*, pages 164–173, 2002.

[73] S. Har-Peled. Clustering motion. In *Proc. 42nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 84–93, 2001.

[74] H. Hartenstein, B. Bochow, A. Ebner, M. Lott, M. Radimirsch, and D. Vollmer. Position-aware ad hoc wireless networks for inter-vehicle communications: the fleetnet project. In *Proceedings of the 2nd ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 01')*, pages 259–262, 2001.

[75] J. Hershberger. Smooth kinetic maintenance of clusters. In *Proc. ACM Symposium on Computational Geometry*, pages 48–57, 2003.

[76] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *IEEE Computer*, 34(8):57–667, August 2001.

[77] S. Hornus and C. Puech. A simple kinetic visibility polygon. In *Proc. 18th European Workshop on Computational Geometry*, pages 27–30, 2002.

[78] W. L. Hsu and G. L. Nemhauser. Easy and hard bottleneck location problems. *Discrete Appl. Math.*, 1:209–215, 1979.

[79] H. B. Hunt, H. Marathe, V. Radhakrishnan, S. Ravi, D. Rosenkrantz, and R. Stearns. NC-approximation schemes for NP- and PSPACE-hard problems for geometric graphs. *Journal of Algorithms*, 26(2), 1998.

[80] M. Imase and B. Waxman. Dynamic steiner tree problem. *SIAM J. Discrete Math*, 4:369–384, 1991.

[81] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 604–613, 1998.

[82] R. Jain, A. Puri, and R. Sengupta. Geographical routing using partial information for wireless ad hoc networks. *IEEE Personal Communications*, 8(1):48–57, Feb. 2001.

[83] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353, pages 153–181. Kluwer Academic Publishers, 1996.

[84] C. E. Jones, K. M. Sivalingam, P. Agrawal, and J.-C. Chen. A survey of energy efficient network protocols for wireless networks. *Wireless Networks*, 7(4):343–358, 2001.

[85] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proc. Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 271–278, 1999.

[86] D. Karger and M. Ruhl. Find nearest neighbors in growth-restricted metrics. In *Proc. ACM Symposium on Theory of Computing*, pages 741–750, 2002.

[87] B. Karp and H. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proc. of the ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 243–254, 2000.

[88] J. M. Keil and C. A. Gutwin. The Delaunay triangulation closely approximates the complete Euclidean graph. In *Proc. International Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes Comput. Sci.*, pages 47–56, 1989.

[89] D. Kim, L. J. Guibas, and S. Shin. Fast collision detection among multiple moving spheres. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 373–375, 1997.

[90] D. Kirkpatrick, J. Snoeyink, and B. Speckmann. Kinetic collision detection for simple polygons. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 322–330, 2000.

[91] P. Klein. Preprocessing an undirected planar network to enable fast approximate distance queries. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 820–827, 2002.

[92] J. Kleinberg. *Approximation Algorithms for Disjoint Paths Problems*. PhD thesis, Dept. of EECS, MIT, 1996.

[93] J. Kleinberg and E. Tardos. Disjoint paths in densely embedded graphs. In *Proc. 36th IEEE Symposium on Foundations of Computer Science*, pages 52–61, 1995.

[94] J. M. Kleinberg. Single-source unsplittable flow. In *IEEE Symposium on Foundations of Computer Science*, pages 68–77, 1996.

[95] S. G. Kolliopoulos and C. Stein. Improved approximation algorithms for unsplittable flow problems. In *IEEE Symposium on Foundations of Computer Science*, pages 426–435, 1997.

[96] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proc. 11th Canadian Conference on Computational Geometry*, pages 51–54, 1999.

[97] R. Krauthgamer and J. R. Lee. The intrinsic dimensionality of graphs. In *Proceedings of the 35th ACM symposium on Theory of computing*, pages 438–447. ACM Press, 2003.

[98] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 798–807. Society for Industrial and Applied Mathematics, 2004.

[99] F. Kuhn, R. Wattenhofer, and A. Zollinger. Asymptotically optimal geometric mobile ad-hoc routing. In *Proc. of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 24–33, 2002.

[100] B. Lamparter, K. Paul, and D. Westhoff. Charging support for ad hoc stub networks. *Computer Communications*, 26(13):1504–1514, August 2003.

[101] T. Leighton. Methods for message routing in parallel machines. *Theoretical Computer Science*, 128(1–2):31–62, 1994.

[102] H. P. Lenhof and M. Smid. Sequential and parallel algorithms for the k closest pairs problem. *Internat. J. Comput. Geom. Appl.*, 5:273–288, 1995.

[103] C. Levcopoulos, G. Narasimhan, and M. H. M. Smid. Improved algorithms for constructing fault-tolerant spanners. *Algorithmica*, 32(1):144–156, 2002.

[104] J. Li, J. Jannotti, D. Decouto, D. Karger, and R. Morris. A scalable location service for geographic ad-hoc routing. In *Proceedings of 6th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 120–130, 2000.

[105] Q. Li, J. Aslam, and D. Rus. Online power-aware routing in wireless ad-hoc networks. In *Proc. ACM MobiCom*, pages 97–107, 2001.

[106] X.-Y. Li, G. Calinescu, and P.-J. Wan. Distributed construction of planar spanner and routing for ad hoc networks. In *IEEE INFOCOM*, pages 1268 – 1277, 2002.

[107] W.-H. Liao, J.-P. Sheu, and Y.-C. Tseng. GRID: A fully location-aware routing protocol for mobile ad hoc networks. *Telecommunication Systems*, 18(1-3):37–60, 2001.

[108] C. R. Lin and M. Gerla. Adaptive clustering for mobile wireless networks. *IEEE Journal of Selected Areas in Communications*, 15(7):1265–1275, 1997.

[109] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance calculation. In *IEEE International Conference on Robotics and Automation*, pages 1008–1014, Apr. 1991.

[110] I. Lotan, F. Schwarzer, D. Halperin, and J.-C. Latombe. Efficient maintenance and self-collision testing for kinematic chains. In *Proceedings of the eighteenth annual symposium on Computational geometry*, pages 43–52. ACM Press, 2002.

[111] J. D. Lundquist, D. R. Cayan, and M. D. Dettinger. Meterorology and hydrology in yosemite national park: A sensor network application. In *Proc. 2nd International Workshop on Information Processing in Sensor Networks*, pages 518–528, 2003.

[112] S. Marti, T. J. Giuli, K. Lai, and M. Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proc. 6th Annual International Conference on Mobile Computing and Networking*, pages 255–265, 2000.

[113] C. Mead and L. Conway. *Introduction to VLSI systems*. Addison-Wesley, 1980.

[114] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27(3):125–128, Mar. 1988.

[115] G. L. Miller, S.-H. Teng, and S. A. Vavasis. An unified geometric approach to graph separators. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 538–547, 1991.

[116] G. Narasimhan and M. Smid. Approximating the stretch factor of Euclidean graphs. *SIAM J. Comput.*, 30:978–989, 2000.

[117] E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *Proc. IEEE INFOCOM*, pages 170–179, 2002.

[118] P. Papadimitratos and Z. Haas. *Securing Mobile Ad Hoc Networks*. CRC Press, 2002.

[119] P. Papadimitratos and Z. Haas. Secure message transmission in mobile ad hoc networks. *Elsevier Ad Hoc Networks Journal*, 1(1), July 2003.

[120] A. Parekh. Selecting routers in ad-hoc wireless networks. In *Proc. of the SBT/IEEE International Telecommunications Symposium*, August 1994.

[121] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proc. IEEE Infocom*, pages 1405–1413, 1997.

[122] D. Peleg. *Distributed Computing: A Locality Sensitive Approach*. Monographs on Discrete Mathematics and Applications. SIAM, 2000.

[123] C. E. Perkins and E. M. Royer. Ad hoc on-demand distance vector routing. In *Proc. of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1999.

[124] C. Petrioli, R. Rao, and J. Redi, editors. *Special Issue: Energy Conserving Protocols*, volume 6. Mobile Networks and Applications, June 2001.

[125] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[126] J. Plesník. On the computational complexity of centers locating in a graph. *Aplikace Matematiky*, 25:445–452, 1980.

[127] S. A. Plotkin. Competitive routing of virtual circuits in ATM networks. *IEEE Journal of Selected Areas in Communications*, 13(6):1128–1136, 1995.

[128] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[129] R. C. Prim. Shortest connection networks and some generalizations. *Bell Syst. Tech. J.*, 36:1389–1401, 1957.

[130] B. Raghavan and A. C. Snoeren. Priority forwarding in ad hoc networks with self-interested parties. In *Proc. Workshop on Economics of Peer-to-Peer Systems (P2PEcon '03)*, June 2003.

[131] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *J. Comp. and System Sciences*, pages 130–143, 1988.

[132] P. Raghavan and C. D. Thompson. Provably good routing in graphs: regular arrays. In *Proceedings of the 17th annual ACM Symposium on Theory of Computing*, pages 79–87, 1985.

[133] T. Rappaport. *Wireless Communications: Principles and Practice*. Prentice-Hall, 1996.

[134] E. M. Royer and C.-K. Toh. A review of current routing protocols for ad-hoc mobile wireless networks. *IEEE Personal Communications*, 6(2):46–55, Apr 1999.

[135] J. S. Salowe. Enumerating interdistances in space. *Internat. J. Comput. Geom. Appl.*, 2:49–59, 1992.

[136] A. Savvides, C.-C. Han, and M. B. Strivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Proceedings of 7th ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 166–179, 2001.

[137] A. Savvides, C.-C. Han, and M. B. Strivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Proc. MobiCom*, pages 166–179, 2001.

[138] A. Savvides and M. B. Strivastava. Distributed fine-grained localization in ad-hoc networks. *IEEE Transactions of Mobile Computing*, 2003.

[139] R. C. Shah and J. M. Rabaey. Energy aware routing for low energy ad hoc sensor networks. In *Proc. IEEE WCNC'02*, March 2002.

[140] J. Sharony. An architecture for mobile radio networks with dynamically changing topology using virtual subnets. *ACM-Baltzer Mobile Networks and Applications Journal*, 1(1):75–86, 1996.

[141] S. Singh, M. Woo, and C. S. Raghavendra. Power-aware routing in mobile ad hoc networks. In *Proc. ACM/IEEE MobiCom*, pages 181–190, 1998.

[142] W. D. Smith and N. C. Wormald. Geometric separator theorems and applications. In *Proc. 39th IEEE Symposium on Foundations of Computer Science*, pages 232–243, 1998.

[143] V. Srinivasan, C. F. Chiasserini, P. Nuggehalli, and R. R. Rao. Optimal rate allocation and traffic splits for energy efficient routing in ad hoc networks. In *IEEE INFOCOM*, pages 950–957, 2002.

[144] I. Stojmenovic. Position based routing in ad hoc networks. *IEEE Communications Magazine*, 40(7):128–134, July 2002.

[145] I. Stojmenovic and X. Lin. Loop-free hybrid single-path/flooding routing algorithms with guaranteed delivery for wireless networks. *IEEE Trans. Parallel Dist. Sys.*, 12(10):1023–1032, 2001.

[146] I. Stojmenovic and X. Lin. Power-aware localized routing in wireless networks. *IEEE Trans. Parallel Dist. Sys.*, 12(11):1122–1133, 2001.

[147] J. M. Sullivan. Sphere packings give an explicit bound for the besicovitch covering theorem. *The Journal of Geometric Analysis*, 2(2):219–230, 1994.

[148] K. J. Supowit. The relative neighborhood graph with an application to minimum spanning trees. *J. ACM*, 30(3):428–448, 1983.

[149] H. Takagi and L. Kleinrock. Optimal transmission ranges for randomly distributed packet radio terminals. *IEEE Trans. Commun.*, 32(3):246–257, 1984.

[150] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science*, pages 242–251, 2001.

[151] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. ACM Symposium on Theory of Computing*, pages 183–192, 2001.

[152] C.-K. Toh. Maximum battery life routing to support ubiquitous mobile computing in wireless ad hoc networks. *IEEE Communications Magazine*, pages 138–147, June 2001.

[153] C.-K. Toh. *Ad Hoc Mobile Wireless Networks: Protocols and Systems*. Prentice Hall, 2002.

[154] G. Toussaint. The relative neighborhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.

[155] D. Turgut, S. K. Das, R. Elmasri, and B. Turgut. Optimizing clustering algorithm in mobile ad hoc networks using genetic algorithmic approach. In *Proc. Global Telecommunications Conference (GlobeCom'02)*, volume 1, pages 62–66, 2002.

[156] J. H. van Lint and R. M. Wilson. *A Course in Combinatorics*. Cambridge Press, 1992.

[157] A. Ward, A. Jones, and A. Hopper. A new location technique for the active office. *IEEE Personnel Communications*, 4(5):42–47, October 1997.

[158] Y. Xu, J. S. Heidemann, and D. Estrin. Geography-informed energy conservation for ad hoc routing. In *Proc. ACM MOBICOM*, pages 70–84, 2001.

[159] T.-H. Yeh, C.-M. Kuo, C.-L. Lei, and H.-C. Yen. Competitive source routing on tori and meshes. In *ISAAC: 8th International Symposium on Algorithms and Computation*, pages 82–91, 1997.

[160] W. Yu and J. Lee. DSR-based energy-aware routing protocols in ad hoc networks. In *Proc. of the 2002 International Conference on Wireless Networks*, 2002.

[161] Y. Yu, R. Govindan, and D. Estrin. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical report tr-01-0023, University of California, Los Angeles, Department of Computer Science, 2001.

[162] G. Zussman and A. Segall. Energy efficient routing in ad hoc disaster recovery networks. In *IEEE INFOCOM*, 2003.

[163] U. Zwick. Exact and approximate distances in graphs - a survey. In *Proc. of 9th European Symposium on Algorithms*, pages 33–48, 2001.