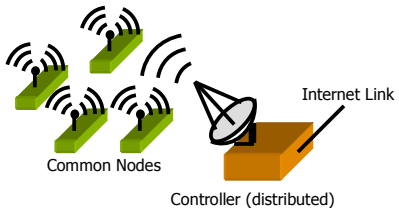
School of Computing
Master in
Computer Science

Dissertation

Optimizing the seed oppnet infrastructure.



Tsamados Panagiotis
ptsamados@gmail.com

Supervisor: Dr. Konstantinos Kokkinos

May 2010

Table of Contents

Abstract

We propose a new point-to-point approach for topology control that refers to opportunistic networks or oppnets [2]. This approach enables an integration of the diverse communication, computation, sensing, storage and other resources that they utilize the network infrastructure. The term "opportunistic" is used for the pervasive computing abilities that can formulate the networking of ad-hoc micro-sensors. The goal for oppnets is to exploit the large scale of resources and capabilities that can be aggregated from the individual capabilities of each sensor. This is often a set of resources that most times remains idle due to the fact that, the routing of communication is usually not distributed. Different devices and systems are either unable to speak/transmit to each other, or do not even try to communicate can be formulated into a combined system of communication once the necessary inter-communication infrastructure is implemented. Oppnets differ from traditional networks, in which the nodes of a single network are all deployed together, with the size of the network and locations of its nodes pre-designed. In oppnets, the initial seed oppnet grows into an expanded oppnet by taking in foreign nodes. In other words, diverse devices join the original set of seed nodes to help the oppnet realize its goals and are so called helpers. In fact, it might happen that the resources available in the seed oppnet are not sufficient to accomplish the task; in such situation, the network can try to scan the radio environment, detect the presence of other networks deployed for different tasks (e.g. WiFi hot spots, or computer networks in an office environment, or GSM/UMTS public networks) and address such helper networks trying to exploit their available resources. We will deal only with the formulation of a topology control that must be initiated prior to any transmission of real data. We propose a distributed algorithm for node identification in order to create the necessary topology.[20]

Keywords: Mobile Ad hoc Networks, Wireless Sensor Networks, Opportunistic networks , Oppnets , pervasive computing ,P2P mobile computing.

Chapter I

1. Introduction

Communication barriers exist among the numerous communications devices that surround us and it remains a great challenge for all researchers and developers working on ever more pervasive computing systems to break those barriers and make all these devices to "talk" to each other.

We propose a new paradigm based on the new technology of opportunistic networks or oppnets to enable integration of the diverse communication, computation, sensing, storage and other devices and resources that surround us more and more. This paradigm initiates an ad-hoc network based on nodes/sensors thrown randomly on a field of interest. Thus, it is our goal to discover the nodes which will finally participate on the network formation. We will consider which criteria and restrictions hold for this formation based on the node capabilities.

Lilien and Gupta, have been the first to propose and to investigate the oppnet paradigm and technology [1,2]. In Oppnets a set of nodes are deployed with the purpose of integrating systems in a diverse communication media under one umbrella. This integrated medium will be the fundamental layer, upon which numerous applications can be built such as rescue and relief, emergency preparedness and response, to mention a few.

Oppnets are a superset of Mobile Ad hoc Networks (MANETS) [8], which contain sensor networks. Traditional networks are deployed together with fixed, pre-defined network parameters, such as size, location, etc. We propose the deploying of sink based oppnets on the fly, where they will be self-configured into an ad hoc network. However, after this "set-up" the sink nodes are dedicated to detect the presence of new nodes or systems of different in general communication media, such as Bluetooth, Wired Internet, WiFi, Radio, RFID, Satellites, etc. These detected systems are then identified, classified and evaluated for their usefulness and reliability as candidates for joining the oppnet. Afterwards, potential candidates are invited to the oppnet. Candidates can accept or reject the invitation. On accepting the invitation, a candidate joining system is admitted into the oppnet under the conditions of following the rules. The resources of this helper system are then integrated with the oppnet and tasks can be offloaded to or distributed amongst these helpers. This makes the new system expandable as well as more powerful relatively to the additional services that it can support. A human interactive command

centre must also be present in order to provide the administration tool for the overall sequence of setting the services throughout the life of the oppnet. When the goals of the oppnet have been realized, it is imperative to release helpers and restore them to the state that is closest to the state that we found them in, minimizing the spending of their resources. Figure 1, depicts theses processes of the oppnet.

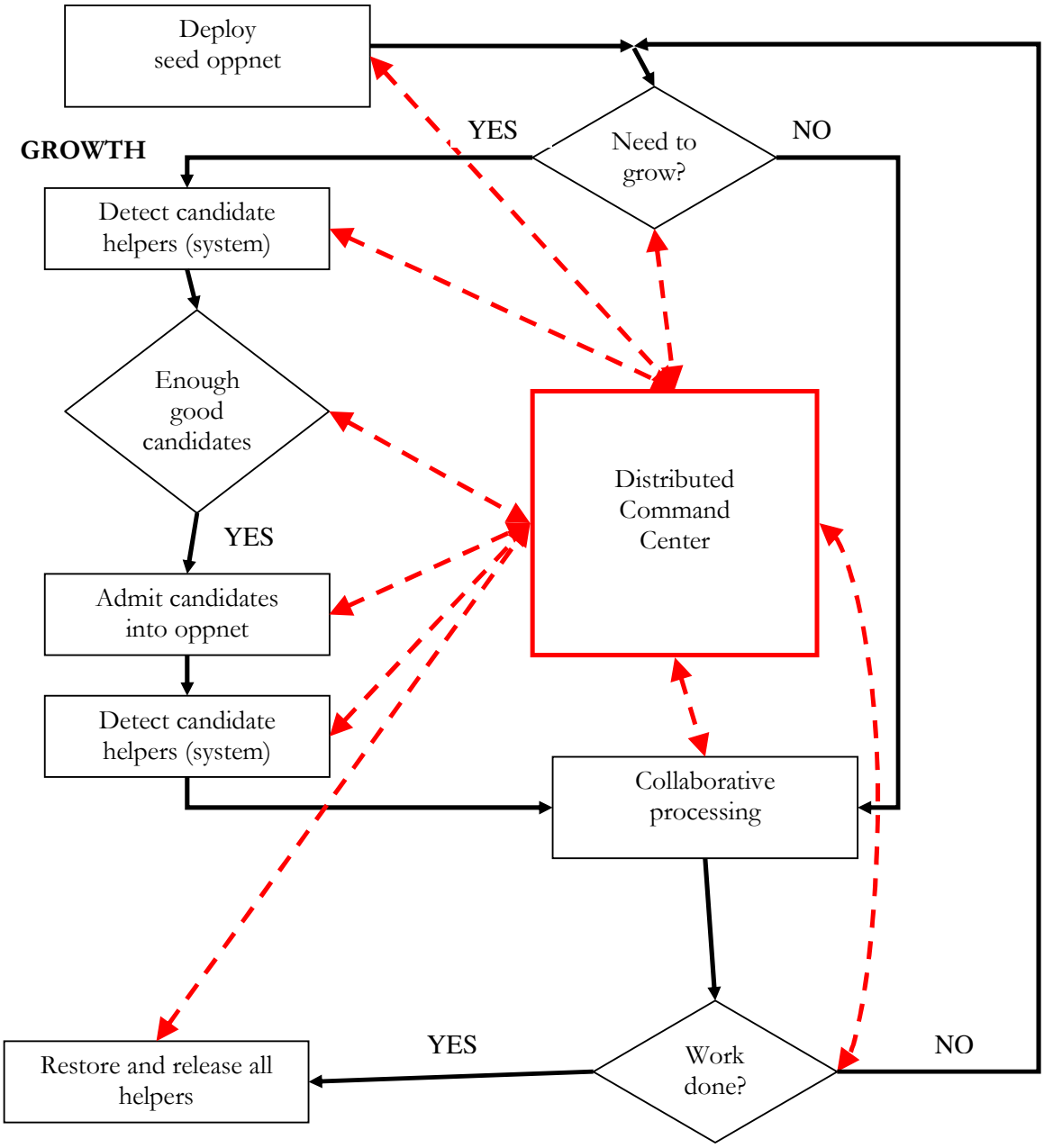


Fig. 1. Delineates and depicts the workings of an oppnet.

Similar architecture schema is depicted in [35].

With this approach, oppnets can be used as a bridge between disjoint communication media, in order to provide access to sensory data from diverse sensing systems that are already present in our environment.

Our goal is to optimize the seed oppnet infrastructure. We call seeds the sink nodes which gather sensory data from other participating nodes. We assume that all participating nodes have a location identification system installed (similar to a GPS). They are also equipped with a local resource controller to prevent battery outage. Communication between seeds is similar to P2P communication. Growth in oppnets looks like growth in Peer to Peer (P2P) systems, unstructured, decentralized P2P systems like Gnutella [3]. The expansion of oppnets is analogous to the spread of malicious instances of worms and viruses. Techniques for integration and management of heterogeneous nodes and devices in oppnets could trail after the techniques employed in grid-systems.

We believe that applications can benefit from using specialized ad hoc networks that can provide a natural basis for them, the basis more efficient and effective than what general-purpose ad hoc networks can offer. The practice of building specialized ad hoc networks is well established. (Mobile ad hoc networks, ad hoc wireless sensor networks, and ad hoc P2P systems are perhaps the best-known categories of specialized ad hoc networks. Yet, many proposed technical solutions are not generally applicable to all ad hoc networks and to systems based on them, being too inefficient for specific application classes or individual applications. This relative lack of application support (including the lack of sufficient quality of service or QoS guarantees) is perhaps the most neglected of today's crucial challenges facing ad hoc networks. Others—including variable topology, component heterogeneity, limited power supply and the need for effective energy-efficient designs—are investigated much more actively.

Specialized ad hoc networks [22,23] provide application-oriented divide-and-conquer approach to ad hoc networking and system research and development—rather than one-size-fits-all general approach, maybe unattainable. Oppnets are specialized ad hoc networks and enable building specialized ad hoc systems.

1.1 Introducing Wirelless Sensor Networks

Wireless sensor networks (WSNs for short) are a particular type of ad hoc network, in which the nodes are ‘smart sensors’, that is, small devices (approximately the size of a coin) equipped with advanced sensing functionalities (thermal, pressure, acoustic, and so on, are examples of such sensing abilities), a small processor, and a short-range wireless transceiver. In this type of network, the sensors exchange information on the environment in order to build a global view of the monitored region, which is made accessible to the external user through one or more gateway node(s).

Sensor networks are expected to bring a breakthrough in the way natural phenomena are observed: the accuracy of the observation will be considerably improved, leading to a better understanding and forecasting of such phenomena. The benefits to the community are considerable even in these early ages of this new network’s generation .

In addition to one or more sensors, each node in a WSN network is typically equipped with a radio transceiver or other wireless communications device, a small microcontroller, and an energy source, usually a battery. The size of a single sensor node can vary from shoebox-sized nodes down to devices with the size of grain of dust, although functioning 'motes' of genuine microscopic dimensions have yet to be created. The cost of sensor nodes is similarly variable, ranging from hundreds of dollars to a few cents, depending on the size of the sensor network and the complexity required for the individual sensors. Size and cost constraints on sensor nodes result in corresponding constraints on their available resources such as energy, memory, computational speed and bandwidth.[9]

A sensor network normally constitutes a wireless ad-hoc network, meaning that each sensor supports a multi-hop routing algorithm (several nodes may forward data packets to the base station).

The features of a typical Wireless Sensor Network are :

- Homogeneous network: A WSN is typically composed of nodes with the same features, especially for what concerns the communication apparatus. A partial exception to this rule is when different types of smart sensor nodes are used in the same network: for instance, a few ‘super nodes’ (with more memory and/or with a longer transmitting range) could be used in combination with standard sensor nodes to increase the network monitoring ability. However, also in this case the number of different device classes used in the network is very limited (2–3 at most).

- Stationary or quasistationary network: Nodes composing a WSN are typically stationary or at most slowly moving. Given the very wide range of WSN applications, exceptions to this rule are possible. This is the case, for instance, of a sensor network used to track animal movements.
- Relatively dispersed network: A wireless sensor network is typically formed by nodes that are dispersed in a relatively large geographical region, so that 1-hop communication between nodes is, in general, not possible.
- Large network size: Typically, the number of nodes composing a WSN is quite large, ranging from few tens to thousands of nodes.[21]

1.2 The Dissertation Problem

Oppnets due to their early nature birth create many great challenges for the researchers. One of them is the optimisation of the seed oppnet infrastructure. Measures and criteria for the optimization of the seed oppnets in their deployment environment need to be set. These criteria have to allow specifications for the amount of communication and also for their computational, sensing and energy resources. Optimal designing of the seed oppnets has to be defined. The minimal seed oppnet configuration that can assure a reliable execution of the oppnet tasks, that have to be carried out, has to be characterized. The localization problem has been shown to be NP-hard, even when distances between nodes are known exactly. So heuristic techniques for the optimization of the oppnets in order to achieve their goals need to be provided. These techniques might rely on probabilities especially when the seeds are not deployed with precise positioning.

These problems emerge the need for designing a new topology controller model. The new model has to define the optimal paths of the nodes of the oppnet taking into account the less energy consumption of the node and the reliability of the execution. Until now the previous approaches to that challenge used Minimum Spanning Tree approach or Minimum Spanning Tree Heuristics. We attempt to approach a solution for this problem by defining a distributed based algorithm for node inclusion in the network.

More specifically, following is a more rigorous definition of the core problems relatively to our dissertation:

Problem 1

Let G = (V , E) be a graph representing the potential oppnet by the seeds that can be formulated . More specifically, let V be the set of those nodes and E the set of possible links that can be possible candidates for any pair of nodes $(v_i , v_j) \in V$. Since we assume that every node V_i is equipped by a local system that identifies its position on a field and also a battery power measurement system, let (x_i , y_i , z_i , d_i) be a 4-tuple indicating the x,y,z coordinates of the node and the node transmission distance respectively. After the nodes are thrown randomly on a field, in a general case scenario, few of those nodes can be positioned in an (x,y,z) location making them unable to participate in the oppnet due to their transmission range restriction. We are looking for solving the following 2 sub-problems:

Sub-problem 1: Find a subset V' of V that can participate in the oppnet .

Sub-problem 2: Identify the set of minimum one-to-one paths for every pair $(v_i , v_j) \in V'$. Note that there is a linear relation between the distance of the path and the battery draining for the transmission of any communication. For this scenario we will also consider two possible variations :

a. Any path p_{ij} for the pair (v_i , v_j) must pass through a seed node.

b. The path p_{ij} does not necessarily pass through a seed node.

Problem 2

Given an oppnet G =(V,E) where V is the set of nodes/seeds that participate from sub-problem 1.1 and E the set of identified links from sub-problem 1.2, we are looking for the necessary and sufficient set of messages needed to be formulated in a communication protocol for other networks to participate in the oppnet. We identify few of those procedures/messages that need to be implemented and included in such a protocol:

1. Triggering a "call for help" extension of the network mechanism.
2. A procedure for authentication of "outsiders" that answer the previous call.
3. An acceptance to the oppnet-extension mechanism.
4. A procedure for advertising new nodes that joined the oppnet.

1.3 Dissertation organization

This dissertation describes the oppnets, the basic core seed of an oppnet and an early implementation framework for oppnet-based applications. In the first chapter of the dissertation we try to introduce the reader to the meaning and the philosophy of the oppnets by describing their basic working model. Also we introduce the reader to the Wireless Sensor Networks. Given that Wireless Sensor Networks are the means to implement an extended oppnet it is understandable that oppnet topology control algorithms and basic primitives will use a large amount of the know-how of the WSNs that already exist in order to bridge many different communication devices that cannot communicate among that due to "linguistic" barriers. In the last section of the first chapter we state the two problems that this dissertation will try to solve in order to contribute to the world research academic society. In the second we describe the modelling techniques of the Mobile Ad Hoc Networks and their energy autonomy capabilities. The topology control system with a location based approach of the MANETS and WSNs is also analysed in this chapter. More over a more thorough analysis of the oppnets characteristics and growth procedures of an oppnet are also stated at the last part of chapter two. The third chapter of the thesis approaches the two problems that we previously stated in chapter one. We prove that problem one is an NP-Complete problem and we propose a Heuristic algorithm and its source code. Approaching problem two we provide a set of call-commands for help to possible oppnet candidates and we will thoroughly describe the authentication procedure of an oppnet candidate.

CHAPTER	CONTENTS OF EACH CHAPTER
1.Introduction	The reader in this chapter is introduced into the understanding of the basic working model of oppnets and WSN networks. He also has the chance to read the 2 basic problems that we will solve in this dissertation.
2.Terminology and Literature review	Chapter II enters the reader to the Mobile Ad Hoc Networks modeling techniques and energy sufficiency. Topology control systems and network growth of the MANETs ,WSNs and Oppnet networks are also stated in this chapter.
3.Theoretical Proofs and Heuristic Algorithm Proposals for Connectivity	In this chapter we prove that problem 1 is NP-Complete and we propose a Heuristic algorithm. Approaching problem 2 we provide a set of call-commands for the authentication of possible oppnet candidates
4.Developed Simulation and Experimental Results	In Chapter IV we present the experimental results of the Steiner software that we developed and we do a comparison of our test results with the Geosteiner 3.1[52] software results.
5.Bibliography	References

Table 1. Table of the basic structure of the dissertation

Chapter II

2. Terminology and Literature review

Oppnets could be characterised as networks that lie within the intersection of ad hoc, P2P, and sensor networks. After necessary adjustments, they can use node localization and self organization techniques from ad hoc networks, growth-by joining approaches from P2P systems, and data aggregation algorithms from sensor nets. In this dissertation section we look at these more categories of related work on technologies potentially useful for oppnets. We also consider the following network scientific areas : 1. Interoperability research on highly heterogeneous wired and wireless types of communication media, networks, devices, and protocols. 2. Grid computing for resource integration and management. 3. Benevolent Trojans for helper search. There is a tremendous amount of knowledge and experience in all 3 areas that we can learn from but cannot employ 'as-is' in oppnets, due to their unique characteristics.

2.1 Topology Control Technology in Mobile Ad hoc networks

A Mobile Ad Hoc Network (MANET) [4] is a network architecture that can be rapidly deployed without relying on pre-existing fixed network infrastructure. The nodes in a MANET can dynamically join and leave the network, frequently, often without warning, and possibly without disruption to other nodes' communication. Finally, the nodes in the network can be highly mobile, thus rapidly changing the node constellation and the presence or absence of links. Examples of the use of the MANETs are:

- Tactical operation : for fast establishment of military communication during the deployment of forces in unknown and hostile terrain [4].
- Rescue missions : for communication in areas without adequate wireless coverage [4].
- National security : for communication in times of national crisis, where the existing communication infrastructure is non-operational due to a natural disaster or a global war [4].
- Law enforcement : for fast establishment of communication infrastructure during law enforcement operations [4].
- Commercial use : for setting up communication in exhibitions, conferences, or sales presentations [4].
- Education : for operation of wall-free (virtual) classrooms [4].

- Sensor networks : for communication between intelligent sensors (e.g., MEMS2)mounted on mobile platforms [4].

Nodes in the MANET exhibit nomadic behaviour by freely migrating within some area, dynamically creating and tearing down associations with other nodes. "Groups of nodes that have a common goal can create formations (clusters) and migrate together, similarly to military units on missions or to guided tours on excursions"[19]. Nodes can communicate with each other at any time and without restrictions, except for connectivity limitations and subject to security provisions. Examples of network nodes are pedestrians, soldiers, or unmanned robots. Examples of mobile platforms on which the network nodes might reside are cars, trucks, buses, tanks, trains, planes, helicopters or ships.

MANETs are intended to provide a data network that is immediately deployable in arbitrary communication environments and is responsive to changes in the underlying network topology. Because ad-hoc networks are intended to be deployable anywhere, existing infrastructure may not be present. The mobile nodes are thus likely to be the sole elements of the network. Differing mobility patterns and radio propagation conditions that vary with time and position can result in intermittent and sporadic connectivity between adjacent nodes. The result is a time-varying network topology [19]. MANETs are distinguished from other ad-hoc networks by rapidly changing network topologies, influenced by the network size and node mobility. Such networks typically have a large span and contain hundreds to thousands of nodes. The MANET nodes exist on top of diverse platforms that exhibit quite different mobility patterns. Within a MANET, there can be significant variations in nodal speed (from stationary nodes to high-speed aircraft), direction of movement, acceleration/deceleration or restrictions on paths (e.g., a car must drive on a road, but a tank does not). A pedestrian is restricted by built objects while airborne platforms can exist anywhere in some range of altitudes. In spite of such volatility, the MANET is expected to deliver diverse traffic types, ranging from pure voice to integrated voice and image, and even possibly some limited video.[4]

2.1.1 Mobile Ad Hoc Networks modelling

Nodes in ad hoc and sensor networks communicate through wireless transceivers. This is the reason why the most important building block of any model for ad hoc networks is the wireless channel model. The wireless channel model we will present below was presented by Rappaport in [24].

A radio channel between a transmitter unit u and a receiver unit v is established if and only if the power of the radio signal received by node v is above a certain threshold, called the sensitivity threshold. Formally, there exists a direct wireless link between u and v if $P_r \geq \beta$, where P_r is the power of the signal received by v, and β denotes the sensitivity threshold. The exact value of β depends on the features of the wireless transceiver and on the communication data rate: for a given radio, the higher the data rate, the higher the value of β, implying a stronger requirement on the received power. [24]

The received power P_r depends on the power P_t used by u to transmit the radio signal, and on the path loss, which models the radio signal degradation with distance. Denoting with PL(u, v) the path loss between units u and v, we have the equation :

$$P_r = \frac{P_t}{PL(u,v)} \qquad [21]$$

The occurrence of a radio channel between any two network nodes can be predicted if the path loss model is known.

Modeling path loss has always been one of the most difficult tasks of every wireless system designer. The mechanisms that regulate radio signal propagation in the environment can be grouped into three categories: reflection, diffraction and *scattering*. Reflection occurs when the electromagnetic wave hits the surface of an object that has very large dimensions when compared to the wavelength of the propagating signal. For instance, almost any radio signal is reflected by the surface of the earth and by large constructions such as skyscrapers and dams. Diffraction is caused by objects with very sharp edges that lie on the radio path between the transmitter and the receiver. Scattering occurs when several small objects are in between the transmitter and the receiver of the radio signal. Typical sources of scattering are the leaves of trees, traffic lights and so on. Given these mechanisms, it is clear that radio wave propagation is an extremely complex phenomenon, which is heavily influenced by environmental factors. Bellow we shortly describe the most common path loss models.

The most common path models that have been presented until today are:

1. The free space propagation model: This model is used to predict radio signal propagation when the path between the transmitter and the receiver is clear and unobstructed (line-of-sight, or LOS, path).
2. The two-ray ground model: It is very rare that the single direct path between the transmitter and the receiver is the only physical means of propagation of the radio signal and that's why the free space propagation model is often inaccurate. In order to improve accuracy, the two-ray ground model considers two propagation paths: the directed path and a ground reflected propagation between the transmitter and the receiver.
3. The long distance path model: The log-distance model has been derived combining analytical and empirical methods. Empirical methods are based on field measurements and reverse curve fitting on the experimental data. This model, which can be seen as a generalization of both the free space and the two-ray ground model, indicates that the average long-distance path loss is proportional to the separation distance d raised to a certain exponent α, which is called the path loss exponent, or distance-power gradient.
4. Large scale and small scale variations: The log-distance propagation model predicts the average received power at a certain distance. However, the intensity of the received signal can vary a lot from the average value. For this reason, probabilistic models have been used to account for the variability of the wireless channel. In a probabilistic propagation model, the radio coverage region is no longer a disk, since the occurrence of a wireless channel between two nodes is a random event. Probabilistic propagation models can be divided into two classes: Large-scale models: These models predict variations of the signal intensity over large distances.

   Small-scale models: These models predict variations of the signal intensity over very short distance. They are also called multipath fading (or simply fading) models.

   The most important large-scale model is the log-normal shadowing model, in which the path loss at distance d is modeled as a random variable with log-normal distribution centered about the mean value. The most important fading model is the Rayleigh model, which models small-scale variations of the radio signal intensity according to a random variable with Rayleigh distribution.[21]

2.1.2 Energy consumption Modelling

The primary concern of a designer while designing a MANET or a WSN is of course the issue of energy consumption. It is understandable that the energy features of nodes participating in a Mobile Ad hoc network differ from the nodes that participate in a Wireless Sensor network therefore we will discuss each network category separately.

2.1.2.1 Energy consumption in MANETs

In MANETs several case scenarios can be implemented and a MANET can be composed of nodes of the most diverse type such as: laptops, cellular phones, PDAs, smart appliances, and so on. Moreover, for many application scenarios (e.g. the 'ubiquitous network'), the network can be composed of heterogeneous devices. Given this node diversity, a typical approach in the literature is to focus attention on the energy consumption of the wireless transceiver only. This is also our choice, which is further motivated by the fact that topology control is primarily concerned with reducing the energy consumed to communicate.

Accordingly to the type of device, the amount of energy consumed by the transceiver varies from about 15% to about 35% of the total energy dissipated by the node. The former value refers to a laptop equipped with a IEEE 802.11 wireless card, while the latter is typical of a PDA device. Since the energy consumed by the wireless card is a significant portion of the total power dissipation in the node, optimizing the energy used to communicate is an important issue.

Several authors have measured the energy consumption of commercial 802.11 wireless cards. Typically, an IEEE 802.11 wireless card has four operational modes:

– *Idle*: The radio is turned on, but it is not used.

– *Transmit*: The radio is transmitting a data packet.

– *Receive*: The radio is receiving a data packet.

– *Sleep*: The radio is powered down.

	Power Idle (mA)	Power Tx (mA)	Power Rx (mA)
802.11 a	203	554	318
802.11 b	203	539	327
802.11 g	203	530	282
TxRange Indoor (m)		Tx Range Outdoor (m)	
802.11 a	13-50	30-300	
802.11 b/g	27-91	76-396	

Table 2. *Nominal power consumption and transmit range of the CISCO IEEE 802.11 a/b/g wireless card [25]*

In Table 2 we can clearly see that power consumption in sleep mode is not reported in the data sheets. The table also reports the nominal transmitting ranges when the card transmits at full power. As seen from the table, the nominal range depends on environmental factors (indoor or outdoor conditions) and on the data rate used to send the message.

We take into consideration that the data reported in Table 2 are nominal values, and may differ from the actual power consumption of the wireless card. For instance, if we consider the specifics of the CISCO Aironet 350 card as reported in the data sheets, the *sleep* : *idle* : *rx* : *tx* power ratios are 0.07:1:1.33:2.22 .[25] These values must be compared with the ratios computed with the measured power consumption, which are 0.04:1:1.20:1.73. All the measurements reported in the literature have outlined an important point, that is, that any radio state transition comes at a significant energy cost (and time latency). This is especially true when the radio transits from the sleep (power down) to the idle (power up) state.[21]

The total power consumption is measured by the drain current, expressed in mA. In the table, the minimum value of the nominal range refers to the maximum data rate (54 Mbps), and the maximum value to the 6 Mbps data rate message, power 1.*y* when the radio is transmitting a message at full power, and power 0.*z* when the radio is in sleep mode. On the other hand, topology control protocols are based on the ability of each wireless node to dynamically adjust its transmitting range. This feature is actually available on some commercial IEEE 802.11 cards,

such as those produced by CISCO. For instance, the CISCO Aironet IEEE 802.11 a/b/g card can use transmit powers ranging from 1 mW to 100 mW. However, this value refers to the power consumption of the RF amplifier, which is only a part of the total power consumed by the wireless card. In fact, the card consumes significant energy also to power up the other analog and digital circuitry. A standard method to model power

consumption when the radio is not transmitting at maximum power does not exist. Most of the approaches presented in the literature are concerned with the transmit power only.

2.1.2.2 Energy consumption in WSNs

For the Wireless Sensor Networks, the task of providing a simple but realistic energy model is relatively simpler, while compared to the case of ad hoc networks. In fact, sensor networks are typically composed of usually very simple and homogeneous devices. Since many sensor nodes have been designed in the research community, their features are very well known. As a result, several sets of energy consumption measurements of wireless sensor nodes have been reported in the literature. [27]

MCU Mode	Sensor Mode	Radio Mode	Total Power (mW)
On	On	Tx (power 36.3 mW)	1080.5
On	On	Tx (power 0.12 mW)	771.1
On	On	Rx	751.6
On	On	Idle	727.5
On	On	Sleep	416.3
On	On	Removed	383.3
Sleep	On	Removed	64.0

Table 3. *Measured power consumption of a Rockwell's WINS sensor node* [26]

In Table 3 we can see the power debauchery of a Rockwell's WINS sensor node [26]. The node is composed of three main components: the microcontroller unit (MCU), the sensing apparatus (sensor), and the wireless radio. If we consider the power consumption of the wireless radio only, we have the following *sleep* : *idle* : *rx* : *tx* ratios: 0.09:1:1.07:2.02. Note that these ratios are quite similar to the case of 802.11 wireless cards, except for a higher power consumption when the radio is transmitting at maximum power. When the transmit power is minimum (0.12 mW), the *idle* : *tx* ratio in the WINS sensor is 1.12. So, there is an almost twofold power consumption increase when varying the transmit power from the minimum to the maximum value. That makes clear to us that varying the transmit power level has a considerable effect on the node's energy consumption.

2.1.3 Mobility Models

The mobility of every node is an illustrious feature of ad hoc networks and, in some cases, also of WSNs. As a consequence, studying the performance of ad hoc/sensor networking protocols in the presence of mobility is a very crucial and fundamental stage of the design process. Since real implementations of ad hoc/sensor

networks are scarce, real-life movement patterns are very difficult to obtain, and the common approach is to use synthetic mobility models and simulation. Mobility models for ad hoc/sensor networks should:

– resemble real-life movements: Considering that the applications range of ad hoc and wireless sensor networks is quite wide, the movement patterns to consider are numerous: they range from university facilities movement of students to vehicle traffic control in highways, from movement of groups of tourists in a urban scenario to rescue squads motion in disaster areas, and from sensors carried around by strong hurricanes used in meteorological WSN applications. Providing a unique mobility model that resembles all these types of mobility is virtually impossible. However, a mobility model should be representative of at least one application scenario.[21]

– be simple enough for simulation/analysis: Since mobility models are used in the simulation of ad hoc networks, the model should be simple enough to be integrated in the simulator and to keep the simulation running time reasonable. The simpler mobility models we use the easiest the task of deriving meaningful analytical results concerning fundamental network parameters in presence of mobility is done. In turn, these results can be used to optimize the performance of ad hoc/sensor networking protocols.[21]

Clearly, the two goals above are conflicting: the more realistic the model is, the more the details that must be included in it, and the model complexity increases. Thus, a synthetic mobility model should be a good compromise between representativeness and simplicity, that is, it should consider the salient features of a certain movement pattern, while disregarding secondary details.

Below we will briefly describe the most important mobility models used in the simulation of ad hoc/sensor networks as described in [28].

Random waypoint model. This is by far the most commonly used mobility model for ad hoc networks. One of the reasons for its popularity is the fact that it is implemented in network simulations tools such as Ns2 [37] and GloMoSim [36] . The Random Waypoint (RWP) model has been introduced in [30] to study the performance of the DSR routing protocol. In RWP model, every single node chooses uniformly at random a destination point (the 'waypoint') within the deployment region *R*, and moves

toward it along a straight line. Node velocity is chosen uniformly at random in the interval [v min, v max], where v min and v max are the minimum and maximum node velocities. When the node arrives at destination, it remains stationary for a predefined pause time, and then starts moving again according to the same pattern.[28]

Random waypoint model is representative of an individual node movement, obstacle-free scenario: each node moves independent of each other (individual movement), and it can potentially move in any sub region of R (obstacle-free). Such a type of movement is usually done when various users move in a big empty warehouse, or in an outdoor environment.

Taking into account its popularity, RWP mobility has been deeply studied in the literature. In particular, it has been discovered that the long-term node spatial distribution of RWP mobile networks is concentrated in the center of the deployment region (border effect)[29], and that the average nodal speed, defined as the average of the node velocities at a given instant of time, decreases over time . These observations have brought to the attention of the community the fact that RWP mobile networks must be carefully simulated. In particular, network performance should be evaluated only after a certain 'warm-up' period, which must be long enough for the network to reach the node spatial and average velocity 'steady-state' distribution.

The RWP model has also been generalized to slightly more realistic, though still simple, models. In [29] the RWP model is extended by allowing nodes to choose pause times from an arbitrary probability distribution. Furthermore, a random fraction of the network nodes remains stationary for the entire simulation time.

Random direction model. This model is similar to the RWP model, the random direction model resembles individual, resembling obstacle-free movement. Random direction model was created to maintain a uniform node spatial distribution during the simulation time, thus avoiding the border effect typical of RWP mobility. In this model [31], any node chooses a direction uniformly at random in the interval [0, 2π], and a random velocity in the interval [v min, v max]. Then, it starts moving in the selected direction with the chosen velocity. In the time point that the node reaches the boundary of R, it chooses a new direction and velocity, and so on. Variants of this model have also been presented. In a first variant a node is 'bounced back' when it reaches the boundary of the deployment region. In another [29], the node moves for a random (exponentially distributed) time, and then it changes direction and velocity of movement.

Brownian-like motion. Opposite to the case of RWP and random direction mobility, which resemble intentional movement, the class of Brownian-like mobility models resembles non intentional motion. That's why these models are sometimes called drunkardlike models.

In Brownian-like motion, the position of a node at a given time step depends (in a certain, probabilistic, way) on the node position at the previous step. In particular, no explicit modeling of movement direction and velocity is used in this model.

A very realistic example of Brownian-like motion is the model used in [32]. Mobility is modeled using three parameters: p_{stat}, p_{move} and m. The first parameter represents the probability that a node remains stationary for the entire simulation time. Parameter p_{move} is the probability that a node moves at a given time step. Parameter m models, to some extent, velocity: if a node is moving at step i, its position at step i + 1 is chosen uniformly at random in the square or side 2m centered at the current node position.[32]

Map-based mobility. At the aggregation of all the models mentioned above, nodes are free to move within any sub-region of the deployment region *R*. However, in many realistic scenarios, nodes are constrained to move within specified paths. This is the case, for instance, of vehicles moving on a country road, or humans moving from sidewalk to sidewalk, and so on. Map-based models have been used to model these situations.

The basic primitive in the definition of map-based models is the map setup, that is, the definition of the paths within which nodes are allowed to move. Then, a certain number of nodes are randomly located on the paths, and they start moving according to scenario-specific rules.

A typical use case scenario of map-based mobility is the Freeway mobility model [33], used to mimic the movement of vehicles on freeways. In this model[33], several freeways are located in the deployment region. Each freeway is composed of a varying number of lanes in both directions. Nodes are randomly located on a freeway, and they move with a random velocity, which is temporally dependent on its previous velocity. If two nodes on the same lane are within a certain minimum distance (safety distance), the velocity of the following node cannot exceed the velocity of the preceding one.

Another instance of map-based mobility is the Manhattan mobility model [33], which is used to emulate urban and suburban movement scenarios. First, a Manhattan-like map, composed of horizontal and vertical streets, is generated. Nodes can move

along the streets in both directions. When a node arrives at an intersection, it randomly chooses whether to continue moving along the same direction, or to take a left or a right turn. Similar to the Freeway model, the velocity of a node at a certain instant of time depends on the node velocity at the previous time step.

Group-based mobility. All the models introduced so far resemble individual mobility. However, in many situations, nodes are expected to move in groups (for instance, groups of tourists moving in a city). Group-based mobility has been introduced to model these situations.

In group-based models, a small subset of the network nodes is defined as the set of group leaders. The remaining nodes are randomly assigned to one of the leaders, thus forming groups. Initially, the leaders are randomly distributed in the deployment region *R*, and the members of each group are randomly located in the neighborhood of the leader. Then, the group leader moves according to one of the previous mobility models, such as RWP or random direction. The other group members 'follow' the leader, having a speed and direction that are a random perturbation of those of the leader. When two groups cross, any group member can leave its group and join the other with a certain probability. Group-based mobility models have been used in [34].

2.2 Topology Control Terminology

Topology control is a term that is depended from many variables and arguments such as network capacity, energy conservation etc. Although that the term topology control is quite wide known in the research scientists community a clear definition of the term has not been introduced clearly. A general description of the term has been introduced in [21] by Paolo Santi and is formulated as "Topology control is the art of coordinating nodes' decision regarding their transmitting ranges, in order to generate a network with the desired properties (e.g. connectivity) while reducing node energy consumption and/or increasing network capacity"[21]. Despite the generality that is impressed through that term description we believe that that approach comes really close to the real definition of the term Topology control.

The description of the term mentioned above does not impose any constraint on the nature of the mechanism used to curb the network topology. So, both centralized and distributed techniques can be classified as topology control. Many scientists consider as topology control techniques also mechanisms used to super impose a network structure on an otherwise flat network organization. This is the case, for instance, of

clustering algorithms, which organize the network into a set of clusters, which are used to ease the task of routing messages between nodes and/or to better balance the energy consumption in the network. Clustering techniques are more often used in the context of wireless sensor networks since these networks are composed of a very large number of nodes and a hierarchical organization of the network units might prove extremely useful.

In a typically clustering protocol, a distributed leader election algorithm is executed in each cluster, and cluster nodes elect one of them as the clusterhead [21]. The election is based on criteria such as energy availability, communication quality etc. The routing of the messages is then performed on the basis of a two-level hierarchy: the message originating at a cluster node is destined to the clusterhead, which decides whether to forward the message to another clusterhead or to deliver the message directly to the destination.

Despite the fact that clustering protocols can be seen as a means of controlling the topology of the network by organizing its nodes into a multilevel hierarchy, a clustering algorithm cannot express in full the definition of the term expressed above because typically the transmit power of the nodes is not modified. A clustering algorithm is concerned with hierarchically organizing the network units assuming the nodes' transmitting range is fixed, while a topology control protocol is concerned with how to modify the nodes' transmitting ranges in such a way that a communication graph with certain properties is generated.

2.2.1 The Taxonomy of Topology Control

In the section above we show that within the spectrum of the general description of the term topology control, presented in [21] by Paolo Santi, many different techniques have the characteristics to be classified as topology control mechanisms. A very successful, in our opinion, attempt to categorise the topology control techniques was also presented in [21] by Paolo Santi and the categorization is the one presented in Figure 2 below.

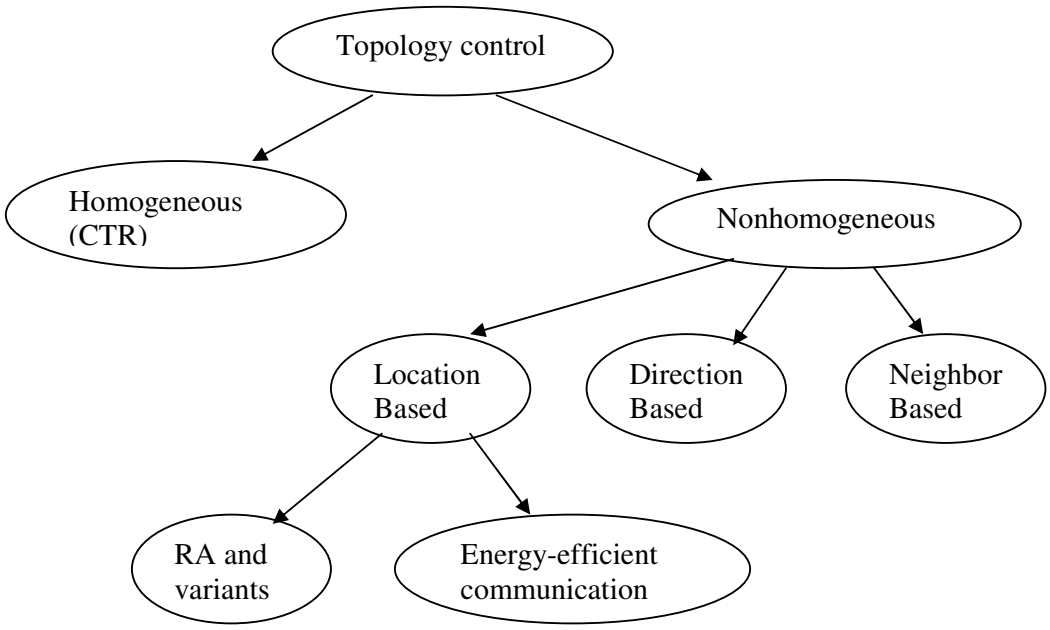


Figure 2. *A taxonomy of topology control techniques* (similar to the one presented by Paolo Santi in [21]).

Topology control can be firstly distinguished in two general categories, homogeneous CTR and Nonhomogeneous topology control. In homogeneous CRT topology control all the network nodes must use the same transmitting range r, and the topology control problem reduces to the simpler problem of determining the minimum value of r such that a certain networkwide property is satisfied. This value of r is known as the critical transmitting range (CTR), since using a range smaller than r would compromise the desired networkwide goal. On the other hand in Nonhomogeneous topology control nodes are allowed to choose different transmitting ranges (subject to the condition that the chosen range does not exceed the maximum range).

Below we will try to give a brief description of each category and subcategory shown in Figure 2.

Homogeneous Critical Transmitting Range topology. It is the more simple form of topology control. In CTR topology control, all the network nodes are assumed to have the same transmitting range r, and the problem is to identify the minimum value of r (the critical range) such that certain networkwide properties are fulfilled. The finding of the minimum value of r that guarantees certain properties is motivated by energy consumption and network capacity concerns. The commonly known version of the CTR problem in ad hoc and sensor networks is the characterization of the CTR for

connectivity, which is, identifying the minimum value of r such that the resulting communication graph is connected. The interest in characterizing the minimal conditions for connectivity lies in the fact that this is the most important network topological property [21].

The CTR for connectivity can be extended in a straightforward manner to deployment regions with arbitrary shape and size. The assumption that all the nodes use the same transmitting range reflects all those situations in which transceivers use the same technology and no transmit power control. That scenario takes flesh and bones for most of the 802.11 wireless cards on the market. Moreover, using the same transmitting range for all the nodes is a reasonable choice and the only way to reduce energy consumption and increase capacity is to reduce r as much as possible [38].

Nonhomogeneous topology control. Nonhomogeneous topology control according to the approach presented in Figure 2 is classified into three subcategories, depending on the type of information that is used in order to compute the topology.

Neighbour-based techniques rely on the nodes' ability to determine the number and identity of neighbors within the maximum transmitting range and to build an order on this neighbor set (based, for instance, on distance or on link quality). This is the minimum amount of information needed by the nodes to build the network topology: if a node is not able to identify its neighbors, it has no clue on how to set its transmit power level. The only possible way of setting the transmit power level in this no-deterministic-knowledge scenario would be to set it at random. In this case, the amount of knowledge available to the node is represented by the probability distribution used to set the transmit power level. One of the most widely known neighbor-based protocol is the KNeigh protocol introduced by Blough in [39] and is a distributed implementation of the computation of G_k based on distance estimation. In other words, it is assumed that when a node u receives a message from node v, u is able to estimate (possibly with a certain error) the distance to node v. This can be accomplished by using one of the many distance estimation techniques.

In direction-based techniques it is assumed that the nodes do not know their position but they can estimate the relative direction of their neighbors. This is relatively less accurate information than knowing exact node locations, as the former type of information can be determined if the latter is known, but not vice versa. This problem is known as the Angle-of-Arrival (AoA) problem, and it is typically solved by equipping nodes with more than one directional antenna [40]. So, in the case of directional

information also, some extra hardware on the nodes (with respect to the standard assumption of nodes equipped with a single, omnidirectional antenna) is needed in order to provide the requested information. An advantage of using AoA-based techniques instead of location-based techniques is that the AoA can be accurately estimated in indoor environments also. Despite the relatively less accurate information used, direction-based topology control protocols can produce almost as good topologies as in the case of location-based topology control. In particular, fully distributed, localized protocols that preserve worst-case connectivity can be designed in this setting also. The two most widely known direction based protocols are the CBTC (Cone-based Topology Control) protocol introduced in [41] and the DistRNG protocol presented in [42]. The Cone-Based Topology Control protocol [41] is based on the idea that a node must retain connections to at least one neighbor in 'every direction', where parameter ρ determines the granularity of what is meant by 'every direction'. On the other hand, the DistRNG protocol presented in [42] is a distributed implementation of the computation of the Relative Neighborhood Graph (RNG).

Location-based techniques are divided into two sub categories, the Range Assignment techniques and the Energy-efficient communication techniques. In location-based approaches, it is assumed that the most accurate information about node positions (the exact node location) is known. This information is either used by a centralized authority to compute a set of transmitting range assignments that optimizes a certain measure (this is the case of the Range Assignment problem and its variants), or it is exchanged between nodes and used to compute an 'almost optimal' topology in a fully distributed manner (this is the case of protocols for building energy-efficient topologies for unicast or broadcast communication). Typically, location-based approaches assume that network nodes, or at least a significant fraction of them, are equipped with GPS receivers. Location-based topology control techniques are thoroughly described in the following chapter.

2.3 Location Based Topology control

As mentioned in the previous section location-based approaches assume that network nodes, or at least a significant fraction of them, are equipped with GPS receivers. That assumption makes these techniques being approached as a more stable techniques. The location based topologies can be sub-divided into 2 categories the Range assignment problem topologies and the energy-efficient communication topologies.

2.3.1 The Range Assignment Problem

Given a set N of network nodes, a range assignment for N is a function RA that assigns to every $u \in N$ a transmitting range $RA(u)$, with $0 < RA(u) \leq r_{max}$, where r_{max} is the maximum transmitting range. Under the assumption that the path loss model is the same for all the network nodes, and that shadowing/fading effects are not considered, transmitting range, and transmit power level are equivalent concepts. Traditionally the function RA is defined in terms of range, instead of power.[21]

The Range assignment problem which was defined first in [43] is defined as:

Let N be a set of nodes in the d-dimensional space, with d = 1, 2, 3. Determine a range assignment function $\overline{\text{RA}}$ such that the corresponding communication graph is strongly connected, and $c(\overline{\text{RA}}) = \sum_{U \in n}\left(\overline{RA}(u)\right)^{\alpha}$ [21] is minimum over all connecting range assignment functions, where α is the distance-power gradient. [21]

The cost measure $c(RA)$ used in the definition of the RA problem is the sum of the transmit power levels used by all the nodes in the network. RA can be informally stated as the problem of finding a 'minimal' nodes' range assignment that generates a connected communication graph, where 'minimal' is intended as 'least energy cost'. Besides reducing energy consumption, a connecting range assignment with minimum energy cost is likely to increase network capacity also. In a certain sense, the RA problem can be seen as a generalization of the problem of determining the CTR for connectivity, where the constraint that all the nodes have the same transmitting range is dropped.

The optimal solution to the RA problem can be found in polynomial time in case of one dimensional networks. In [43] Kirousis presented the OPTIMAL1DRA algorithm which is demonstrated in Figure 3. The algorithm first identifies a set of optimal range assignments for connecting node u_1 to any other single node (step 1.2). Then, we have the recursive step, in which a set of optimal range assignments for connecting nodes in $\{u_1, \ldots, u_k\}$ with a receiver node u_l , with $k \leq l \leq n$, is calculated. After n recursive steps, any range assignment in $\text{Opt}((\{u_1, \ldots, u_n\}, \emptyset), u_n)$ is optimal for N. In general the

algorithm computes a set of transmitting range assignments of minimum energy cost that generates a strongly connected communication graph.

Algorithm OPTIMAL1DRA:
Initialization
Let RA_i be the range assignment such that $RA_i(u_1) = \delta(u_1, u_i)$,
and $RA_i(u_j) = 0$ otherwise
for $i = 2, \ldots, n$ do $Opt((\{u_1\}, \emptyset), u_i) = RA_i$
Step k:
Assume we know $Opt((\{u_1, \ldots, u_k\}, E_{i,k}), u_l)$, for any $1 \le i \le k$ and $k \le l \le n$
for any j, m such that $1 \le j \le k + 1 \le m \le n$
consider all possible values of $RA(u_{k+1})$ (there are $k + 2$ such values)
for each such value r, find an assignment RA in
$Opt((\{u_1, \ldots, u_k\}, E_{j,k+1}), u_{k+1}, (u_{k+1}, r))$
if RA has cost lower than that of the current range assignment for j, m,
store RA (new current minimum)
at the end of step k, we know a range assignment in
$Opt((\{u_1, \ldots, u_{k+1}\}, E_{i,k+1}), u_l)$, for any $1 \le i \le k + 1 \le l \le n$
after step n, an optimal assignment is one in $Opt((\{u_1, \ldots, u_n\}, \emptyset), u_n)$

Figure 3. The OPTIMAL1DRA algorithm for finding the optimal range assignment (as presented in [21] [43])

2.3.2 Energy-efficient Communication Topologies

Another main problem that has concerned the topology designers diachronically is the problem of identifying good topologies for energy-efficient communication among the network nodes. In other words, given the communication graph G obtained when all the nodes transmit at maximum power, and the goal is to identify a sparse subgraph G′ of G such that only energy-efficient links are retained in G′. The criterion used to determine whether a certain link in G is energy efficient depends on the communication pattern considered. Typically, the focus is on either end-to-end communication between arbitrary nodes (unicast) or on one-to-all communications (broadcast).

The most important factor in order to define an energy routing graph is the Power stretch factor which is a generalization of the concept of distance stretch factor, which is well known in computational geometry and is defined in [21] as:

Let G′ be an arbitrary sub-graph of the max power graph G = (N,E). The power stretch factor of G′ with respect to G, denoted as $\rho_{G'}$, is the maximum over all possible node pairs of the ratio between the power cost of the minimum power path in G′ and in G. Formally ,

$$\rho_{G'} = \max_{u,v \in N} \frac{pc\left(P_{uv}^{min,G'}\right)}{pc\left(P_{uv}^{min,G}\right)} \quad [21]$$

By convention, we define $\rho_{G'} = \infty$ if there exist nodes u, v that are connected in G, but are disconnected in G′.

The graphs that have been used to build rooting graphs, considering the power stretch factor, are called proximity graphs. They are called proximity graphs since the set of links incident in any node u of the computed graph can be calculated on the basis of the position of the neighbor nodes in the maxpower graph. Thus, proximity graphs can be constructed in a fully distributed and localized way.

One of the most famous proximity graphs is the Gabriel Graph . The Gabriel Graph has optimal power stretch factor and as we can see in Figure 4 an edge (u, v) ∈ G is included in the Gabriel Graph if and only if the disk with the same edge as diameter contains no node of G. The algorithm shown in Figure 5 was presented in [44] and is used to construct and compute the Gabriel Graph in a fully distributed and localized way. The algorithm relies on the assumption that every node in the network knows its position on the plane. This can be accomplished by equipping nodes with low-power GPS receivers, or by using other location estimation techniques. Considering that the maximum node degree in the maxpower graph can be as high as $n - 1$, it can be easily seen that the algorithm has $O(n^2)$ time complexity. The message complexity is $O(n)$, since every node in the graph transmits a single message.

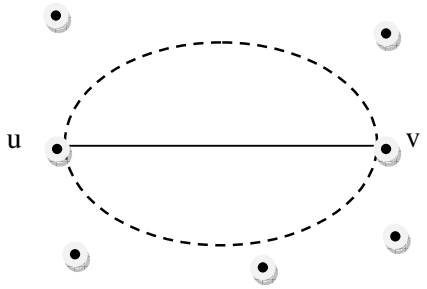


Figure 4. *Edge (u, v) ∈ G is included in the Gabriel Graph if and only if the disk with the same edge as diameter contains no node of G.* (taken from [21] [44])

Algorithm GabrielGraph:
(Algorithm for node u)

ID_u is the identifier of node u, and (x_u, y_u) is its location
$E_G(u)$ and $E_{GG}(u)$ are the set of links of the power graph and of the GG node u is aware of
disk(u, v) is the disk with edge (u, v) as diameter

1. Initialization
   Locally broadcast message $(ID_u,(x_u, y_u))$ at maximum power
   $E_G(u) = E_{GG}(u) = \emptyset$
2. Processing of incoming messages
   upon receiving message $(ID_v,(x_v, y_v))$ from node v
   add (u, v) to $E_G(u)$
   check whether exists edge (u,w) $\in E_G(u)$ such that w $\in$ disk(u, v)
   if no, then add (u, v) to $E_{GG}(u)$
   for each (u,w) $\in E_{GG}(u)$
   check whether v $\in$ disk(u,w)
   if yes, then remove (u,w) from $E_{GG}(u)$
3. Termination
   after processing all incoming messages, EGG(u) contains all the edges of GG incident in u

Figure 5. *The Gabriel Graph construction algorithm*. (as presented in [21])

2.3.3 Location Based distributed topology control protocols

After describing the two sub-categories of the location based topologies we will briefly present two protocols of that architecture, the R&M protocol and the LMST protocol. This two protocols rely on very accurate information, which is assumed to be somehow available to the nodes. The easiest way to satisfy this assumption is to equip every node in the network with low-power GPS receivers, which enable very accurate position estimation and loose synchronization with no message exchange between network nodes. This solution has a high cost in terms of hardware required on the nodes, but it entails no message overhead. These protocols are mainly used in outdoor applications, while their applicability in indoor environments is limited by the poor accuracy provided by location estimation techniques in this setting.

The R&M protocol was presented by Rodoplu and Meng in [45] and it builds a topology that is optimized for the all-to-one communication pattern, where one of the network nodes is designated as the master node, and all the other nodes send messages to the master. This traffic pattern is typical of Wireless Sensor Networks, where the deployed sensors must send the collected data to one (or more) base station(s).

Considering the fact that WSNs are typically deployed outdoor, the R&M protocol is well suited to build and maintain the topology in sensor networks. In order the R&M protocol to find the optimal solution to the MinEnergyAllToOne problem firstly every node in the network computes its enclosure and neighbor set, that is, its local view of the enclosure graph G_ε and then, a global cost distribution is initiated to determine the minimum-energy reverse spanning tree rooted at the master node, which is a sub-graph of G_ε. The protocol uses the EnclosureGraph algorithm shown in Figure 6 in order to construct the graph. Every node u broadcasts a beacon message containing its ID and position at maximum transmit power. As beacon messages of other nodes are received, u calculates the corresponding relay region, and keeps track of whether the newly found node is in the relay region of some previously found node. To accomplish this task, potential neighbors are marked dead or alive: a node is said to be dead if it lays in the relay region of some other neighbor of u, and it is said to be alive otherwise. After all the beacon messages from surrounding nodes are received, the set of alive nodes defines the neighbor set of node u, that is, u's local view of the enclosure graph. The FlipAllStatesDownChain is an auxiliary function shown in Figure 7 that is called *to* update the alive/dead state of the currently discovered nodes. After the construction of the enclosure graph the Bellman–Ford shortest path algorithm , presented by Lynch in [46], is used on the enclosure graph to compute the minimum-energy reverse spanning tree rooted at the master node.[21]

Algorithm EnclosureGraph:
(algorithm for node u)
AliveNodes is the current set of alive nodes
DeadNodes is the current set of dead nodes
N(u) is the neighbor set of node u
ε_u is the enclosure of node u
Initialization
AliveNodes = ∅
DeadNodes = ∅
N(u) = ∅
send beacon message (u, (x_u, y_u))
Upon receiving beacon message (v, (x_v, y_v))
compute relay region $RR_{u\rightarrow v}$
MarkDead(v)
FlipAllStatesDownChain(v)
Termination (after receiving all beacon messages)
N(u) = AliveNodes
$\varepsilon_u = \bigcap_{v \text{AliveNodes}} (RR^c{}_{u\rightarrow v} \cap R)$

Figure 6. *The EnclosureGraph algorithm* (as presented in [21]).

Function FlipAllStatesDownChain(v):
(algorithm for node u)

1. If v ∈ AliveNodes then
   MarkDead(v)
   for each w ∈ $RR_{u\rightarrow v}$ do FlipAllStatesDownChain(w)

else
  if (v ∉ $RR_{u\rightarrow w}$) ∀w ∈ AliveNodes then
        MarkAlive(v)
        for each w ∈ $RR_{u\rightarrow v}$ do FlipAllStatesDownChain(w)

Figure 7. *The auxiliary function FlipAllStatesDownChain used in EnclosureGraph algorithm* (as presented in [21].

The R&M protocol builds a topology that optimizes a certain energy-efficient topology problem, namely, MinEnergyAllToOne. In the MinEnergyAllToOne problem, the goal is to find the most efficient way of communicating with the master node, that is, the 'wireless advantage' is not useful in this case. Then, to solve the problem, it is sufficient to find the most energy-efficient path from any node to the master node, which can be done in polynomial time. The reverse version of MinEnergyAllToOne, that is, the energy-efficient broadcast problem, is NP-hard. In the energy-efficient broadcast problem the nodes use the so-called wireless advantage, that is, the fact that the message sent by a node can be received by all the nodes within its range, and computing the best way of using the 'wireless advantage' is NP-hard. Generally speaking the R&M protocol can be successfully used in many wireless sensor network application scenarios, where mostly stationary network nodes must send their data to a gateway node. In this case, the network topology is typically built at the beginning of the network lifetime, and constructing an energy-efficient topology is a primary design concern.

The LMST protocol was presented by Li in [47] and is composed of three phases: information exchange, topology construction, and determination of transmit power, and an optional optimization phase: construction of topology with bidirectional links. In the first phase of the protocol, each node sends its ID and location to all nodes in the visible neighborhood. This can be accomplished by sending a beacon message at maximum transmit power. Once all the beacon messages of the visible neighbors have been received, each node constructs its local MST by applying the classical Prim's algorithm [48]. After Prim's algorithm execution, every node u in the network knows the (unique) MST $T_u = (VN_u,E_u)$ connecting u to all its visible neighbors. The next step is to define

the set of neighbors of u in the final topology, that is, u's local view of the LMST topology. This is defined as:

Node v is a neighbor of node u, denoted as $u \rightarrow v$, if and only if v is a one-hop neighbor of u in its minimum spanning tree $T_u = (VN_u, E_u)$.

$$\text{Formally,} \quad u \rightarrow v \Leftrightarrow (u, v) \in E_u.$$

The neighbor set of node u, is defined as $N(u) = \{v \in VN_u \mid u \rightarrow v\}$.

The final step of the protocol is the determination of the transmit power needed to send a message to any neighbor node. This can be accomplished by measuring the received power of the beacon messages: when node u receives the beacon from a certain visible neighbor v, it can estimate the minimum power level needed to reach v by comparing the received power of the beacon with the maximum transmit power. The algorithm of the LMST protocol, which is presented in Figure 8, is proven in [47] that produces a topology that preserves connectivity in the worst case, has maximum logical node degree equal and can be computed in a fully distributed and localized fashion. Also Li in [47] proved that LMST protocol produces topologies with a smaller average logical node degree and average transmission radius with respect to those generated by R&M.

Algorithm LMST:
(algorithm for node u)
VN_u is the visible neighborhood of node u
N(u) is the neighbor set of node u
bp_u is the broadcast power of node u
(x_u, y_u) are the coordinates of node u

1. Information exchange
   send beacon $(u, (x_u, y_u))$ at maximum power
   upon receiving beacon $(v, (x_v, y_v))$, store the received power of this message in rp_v

2. Topology construction (after all beacons have been received)
   build the local MST on nodes in VNu using Prim's algorithm
   let $T_u = (VN_u, E_u)$ be this local MST
   $N(u) = \{v \in VN_u \mid (u, v) \in E_u\}$

3. Determination of transmit power
   for each $v \in N(u)$
   compute the minimum power mp_v needed to reach v based on rp_v
   $bp_u = \max_{v \in N(u)} mp_v$

4. (Optional) Topology with bidirectional links
   for each $v \in N(u)$
   send probe message (u, mp_v) to v using power mp_v
   upon receiving reply message (v, state) from v
   if state = uni then

notify v sending message (u, add) (technique 1)) using power mpv
or
delete v from N(u) (technique (2))
upon receiving probe message (v, mp_u)
if v ∈ N(u) then
send reply message (u, bi) using power mp_u
otherwise
send reply message (u, uni) using power mp_u
upon receiving notify message (v, add)
add node v to N(u), with associated power mp_v
if necessary, update the broadcast power bp_u

Figure 8. *The LMST protocol algorithm* (as presented in [47]).

2.4 State of Art in Seed Oppnets

Oppnets constitute the subcategory of ad hoc networks where diverse systems, not employed originally as nodes of an oppnet, join it dynamically in order to perform certain tasks they have been called to participate in. Oppnets have a significant potential to improve a variety of applications, and to create new application niches. Opportunistic networks are categorized as class 1 opportunistic networks that use opportunistically only communication resources, and class 2 opportunistic networks that use opportunistically all kinds of resources, including not only communication but also computation, sensing, actuation, storage, etc. Every opportunistic network grows from a seed oppnet, or simply a seed, which is a set of nodes employed together at the time of the initial oppnet deployment as shown in Figure 9.

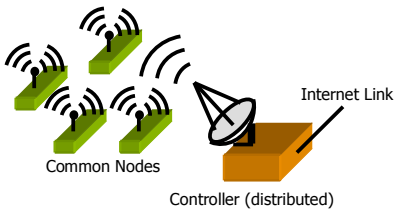


Fig. 9 *Seed oppnet—an example*

The seed is pre-designed. It can have just a few nodes, in the worst occasion even a single node. A seed can be wireless with nodes communicating via radio channels, and ad hoc with nodes not carefully pre-positioned but, for instance, thrown out of a plane or a moving car in the general disaster area. Seed nodes for emergency operation could be quite powerful, such as powerful mobile communication/computing/sensing hosts

mounted on heavy all-terrain trucks or amphibious vehicles, or in parachute-dropped containers. Once the seed becomes operational, its first task is to detect a set of "foreign" entities devices, node clusters, networks, and other systems which it recognizes as useful. The detected entities are candidates for becoming "helpers" for the oppnet. Each such candidate helper has a potential to provide the oppnet with some communication, computing, sensing, or other capabilities or resources. Candidate detection can be done by any means possible, both traditional and novel [6]. Oppnets can use satellite-based detection or radio-based detection (including detection of cell phone-equipped systems). It can search for systems in the disaster area using the range of Internet addresses, known as IP addresses, assigned to its own geographical area. It can even use artificial intelligence techniques for visual detection of systems and appliances with embedded chips. For example, it can visually detect a car within the surveillance area of a helper that already joined the oppnet, read its license plate, check if the car is equipped with the OnStar system [7], and attempt to contact its embedded network if it is. Candidates are evaluated by the oppnet, and the best ones are ordered or invited to join the oppnet. Invited candidates can either accept or refuse the invitation. However, in emergency life-or-death situations, candidates can be forced to join, and must agree to be conscripted in the spirit of citizens called to arms (or suffer the consequences of going AWOL absent without leave)[10]. A candidate ordered to join still needs to ask for permission to join, since the need of the oppnet for helpers is, in general, very dynamic. The oppnet admits the candidates that it still needs at the moment of its admission decision.

It is important to note that by admitting helpers an oppnet can leverage all kinds of resources it needs that are available in its environment. Oppnet growth is a mechanism to obtain a lot of help at a very low cost.

Foreign entities admitted into the oppnet become its full-fledged members, or helpers. By admitting candidates, the seed grows into an expanded oppnet. For example, the expanded oppnet in Figure 10 admitted the following candidates that became helpers: (a) a computer network, contacted via a wired Internet link; (b) a cell phone infrastructure (represented in Figure 10 by the Cell phone Tower), contacted via Bluetooth-enabled oppnet's cell phone peripheral, (c) a satellite, contacted via a direct link, (d) a home area network (HAN), contacted via an intelligent appliance (e.g., a refrigerator) with a wireless link, (e) a microwave network, contacted via a microwave relay. [10]

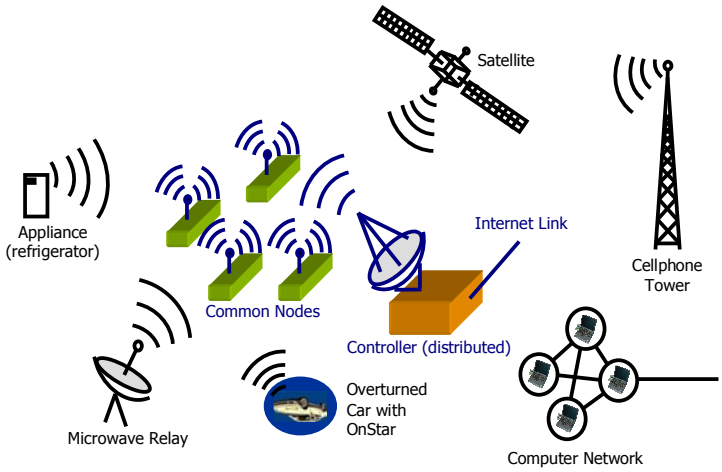


Fig. 10 *Expanded oppnet—an example*

All helpers collaborate on realization of its oppnet's goals. They can be deployed to execute all kinds of tasks even though, in general, they were not designed to become elements of an oppnet that invites them. A helper may be allowed by an oppnet manager to invite other systems. The more helpers are allowed to invite foreign nodes the faster oppnet grows.

Conclusively, oppnets differ from traditional networks, in which the nodes of a single network are all deployed together, with the network size and locations of its nodes pre-designed. In oppnets, the initial seed oppnet grows into an expanded oppnet by taking in foreign nodes. Thus, oppnets constitute the category of networks where diverse devices, not employed initially as its nodes, join the original set of seed nodes to become helpers assisting the oppnet in realization of its goals. When deployed, oppnets attempt to detect candidate helper systems existing in their relative vicinity ranging from sensing and monitoring, to computing and communication systems and integrate them under their own control. When such a candidate is detected by an oppnet, the oppnet evaluates the benefits that it could realize if the candidate joins it. If the evaluation is positive the oppnet invites the candidate to become its helper. In this manner, an oppnet can grow from a small seed into a large network with huge sensing, communication, and computation capabilities.[20]

Oppnets can facilitate many applications scenarios. As an example, they can help building an integrated network called for in various critical or emergency situations. Oppnets can be used to enable connectivity in an area where any existing communication or information infrastructure has been fractured or partially destroyed. Oppnets will

integrate various systems that were not designed to work together. The integration will enhance the flow of information that, for example, can assist in rescue and recovery efforts for destroyed areas, or can provide more data on phenomena that are just developing, such as wildfires or twister storms.

Chapter III

3. Theoretical Proofs and Heuristic Algorithm Proposals for Connectivity

We believe that every scientist has the duty to face unsolved problems and try to give solutions in order to advance the research made by other fellow scientists before him. That is the only way that science can advance forward and provide solutions for all mankind that will arm the humanity in order to face in a better way challenges that exist or will emerge in the future. Scientists have to try to provide these solutions not for their personal projection but for a contribution to all humanity.

In Chapter III we state once again, more thoroughly this time, the two main problems that we face in our dissertation and we provide the theoretic proofs and solutions for these problems.

3.1 Problem Statement

Oppnets due to their early nature birth create many great challenges for the researchers. One of them is the optimisation of the seed oppnet infrastructure. Measures and criteria for the optimization of the seed oppnets in their deployment environment need to be set. These criteria have to allow specifications for the amount of communication and also for their computational, sensing and energy resources. Optimal designing of the seed oppnets has to be defined. The minimal seed oppnet configuration that can assure a reliable execution of the oppnet tasks, that have to be carried out, has to be characterized. The localization problem has been shown to be NP-hard, even when distances between nodes are known exactly. So heuristic techniques for the optimization of the oppnets in order to achieve their goals need to be provided. These techniques might rely on probabilities especially when the seeds are not deployed with precise positioning.

These problems emerge the need for designing a new topology controller model. The new model has to define the optimal paths of the nodes of the oppnet taking into account the less energy consumption of the node and the reliability of
the execution. Until now the previous approaches to that challenge used Minimum Spanning Tree approach or Minimum Spanning Tree Heuristics. We approach a solution for this problem by defining a distributed based algorithm for node inclusion in the network.

More specifically, following is a more rigorous definition of the core problems that we face in our dissertation:

Problem 1

Let G = (V , E) be a graph representing the potential oppnet by the seeds that can be formulated . More specifically, let V be the set of those nodes and E the set of possible links that can be possible candidates for any pair of nodes $(v_i , v_j) \in V$. Since we assume that every node V_i is equipped by a local system that identifies its position on a field and also a battery power measurement system, let (x_i , y_i , z_i , d_i) be a 4-tuple indicating the x,y,z coordinates of the node and the node transmission distance respectively. After the nodes are thrown randomly on a field, in a general case scenario, few of those nodes can be positioned in an (x,y,z) location making them unable to participate in the oppnet due to their transmission range restriction. We are looking for solving the following 2 sub-problems:

Sub-problem 1: Find a subset V' of V that can participate in the oppnet .

Sub-problem 2: Identify the set of minimum one-to-one paths for every pair $(v_i , v_j) \in V'$. Note that there is a linear relation between the distance of the path and the battery draining for the transmission of any communication. For this scenario we will also consider two possible variations :

a. Any path p_{ij} for the pair (v_i , v_j) must pass through a seed node.

b. The path p_{ij} does not necessarily pass through a seed node.

Problem 2

Given an oppnet G =(V,E) where V is the set of nodes/seeds that participate from sub-problem 1.1 and E the set of identified links from sub-problem 1.2, we are looking for the necessary and sufficient set of messages needed to be formulated in a communication protocol for other networks to participate in the oppnet. We identify few of those procedures/messages that need to be implemented and included in such a protocol:

1. Triggering a "call for help" extension of the network mechanism.
2. A procedure for authentication of "outsiders" that answer the previous call.
3. An acceptance to the oppnet-extension mechanism.
4. A procedure for advertising new nodes that joined the oppnet.

3.2 Solution of Problem 1 sub-problem 1

In order to provide a realistic solution scenario to the sub-problem 1 of problem 1 we have to set some standards.

Restriction 1: The seeds of the oppnet are capable to move to different locations according to the commands they take from the Control Center in order to establish communication. That restriction is necessary given the fact that oppnets are designed to be deployed to emergency situations such as warzones that the standard communication infrastructure has been destroyed or natural disasters such as hurricane Katrina that destroyed almost every communication infrastructure that existed in New Orleans. The seeds can be remote controlled robotic all-terrain vehicles or remote controlled robotic bugs or remote controlled mini helicopters all equipped with a GPS receiver and a wireless communication 916/433 MHz sensor or a usb Bluetooth 2.4GHz dongle or a wireless Internet 802.11 b/g 2.4GHz adapter or all of them capable to send and receive messages M_i in a range r_i.

Restriction 2: All the seeds, along with the sensor nodes, are dropped in a scattered way by air from an airplane or helicopter in the disaster area or the warzone. We also have the capability to drop extra seeds if we have to in order to achieve the communication goal we target.

Our goal, is to find a subset V′ of V that can participate in the oppnet, such that to make it optimal in length. Therefore, we are looking for a minimum communication tree made by the virtual connections (links) in V′. Note that the set of seeds S can also participate in the process having the role of Steiner points since the minimum known communication tree is the Steiner tree. We set that V does not include set S of the seeds. In Figure 11 we provide a graphical representation of the graph G= (V,E) of the oppnet network.

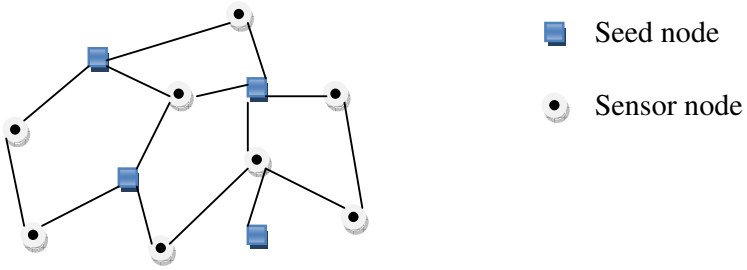


Figure 11. The graph G= (V,E) representing the oppnet network consisted by sensor nodes and seed nodes.

We now provide an algorithm we named UNION in order to create the set of nodes which will participate in the oppnet among the nodes in V. First of all the algorithm UNION creates and sends a broadcast message M_i for each node. At this point we have to assume that according to the specifications that we set in Restriction 1 each node can send and receive a message M_i in a range r_i . Then the algorithm creates a For loop in which each node $v_i \in V$ is in a state to receive messages of the form (broadcast, v_i, x_i, y_i, z_i). Within this loop each node forwards all messages to a seed assuming that $\exists$ has at least one seed. Then if $v_s \in S$ gather all messages forwarded by the nodes. The algorithm then creates an array A[1….n] assuming that $|V|$ equals n. A nested if that sets that if $\exists M_i$ (broadcast, V_i) then A[i] is set as true. Finally a For loop is created for (i← 1 to n) to check if A[i] is set as true then V' equals $V' \cup v_i$. The result of the algorithm is that V' contains all the nodes participating in the communication process.

Algorithm UNION

Create a broadcast message M_i for each node and send it

For each node $v_i \in V$ receive messages (M_i, v_i, x_i, y_i, z_i)

Forward all M_i to a seed

If $v_s \in S$ then

Gather all M_i forwarded by the nodes

Create an array A[1….n] (assuming that $|V| = n$)

If $\exists M_i$ (broadcast, V_i)

then A[i] = true

For (i← 1 to n)

If A[i] = true

$V' = V' \cup v_i$

Figure 12. The proposed pseudo-code algorithm UNION

3.3 Solution for Problem 1 sub-problem 2

Approaching the solution of Problem 1 sub-problem 2 we can clearly state that our goal in order to achieve a solution is to make the communication tree. The most obvious solution could be to calculate the Minimum Spanning Tree. A Minimum Spanning Tree is a spanning tree of a graph with weight less than or equal to the weight of every other spanning tree. Generally speaking, any undirected graph has a minimum spanning forest, which is a union of minimum spanning trees for its connected

components.[49] Although the Minimum Spanning Tree calculation seems to be the solution, it has been proven that Minimum Spanning Tree is not the optimal solution.

On the other hand the Minimum Steiner Tree solution has been proven to be the optimal one. Minimum Steiner Tree problem, which was presented by Jakob Steiner, is superficially similar to the minimum spanning tree problem: "given a set V of points (vertices), interconnect them by a network (graph) of shortest length, where the length is the sum of the lengths of all edges. The difference between the Steiner tree problem and the minimum spanning tree problem is that, in the Steiner tree problem, extra intermediate vertices and edges may be added to the graph in order to reduce the length of the spanning tree. These new vertices introduced to decrease the total length of connection are known as Steiner points or Steiner vertices. It has been proved that the resulting connection is a tree, known as the Steiner tree". [50]

The problem of establishing the Minimum Steiner tree belongs to the NP-Complete class of exponential exact solution computational problems. Therefore are the referred above citations to the Steiner tree formulation are all heuristics. We created our own heuristic algorithm for which we prove its computational complexity and we performed experimental results by our simulator to show the merit of our approach.

In order to provide a solution of the Minimum Steiner Tree problem in a more formal way we are obliged to set some propositions. The deployment field of the oppnet although it is formulated in a three dimensional space follows the primitives of a two dimensional space. As shown in Figure 13 the underlying grid

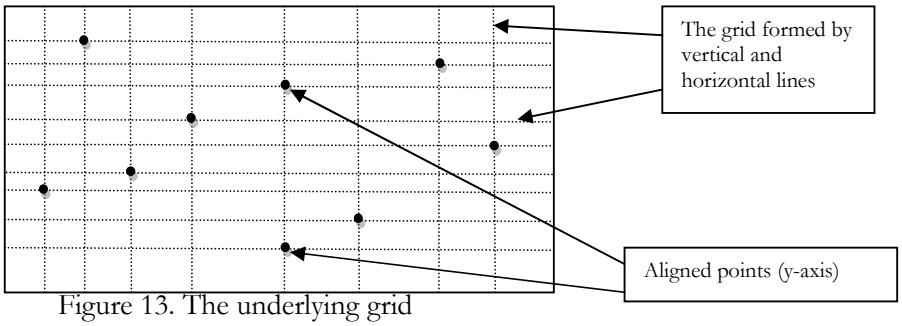


Figure 13. The underlying grid

of a point set P on the oriented plane is formed by horizontal and vertical lines passing through every point in P. The points in P are represented with their coordinates, i.e. $p_i \in$ P is represented with (x_i, y_i) The rectilinear distance between two points $p_i, p_j \in$ P is $|x_i-x_j|+|y_i-y_j|$. Let P be a set of n points on the grid and $X = \{x_1, x_2,\ldots,x_n\}$, $Y = \{y_1, y_2,\ldots, y_n\}$ the sets of x and y coordinates of all points in P. Let x_{min}, x_{max}, y_{min}, y_{max} represent the

minimum and maximum elements of the corresponding sets X and Y. An envelope for P is the periphery of the rectangle defined by the four points (x_{min}, y_{min}), (x_{min}, y_{max}), (x_{max}, y_{min}), and (x_{max}, y_{max}) . The whole rectangle is called a mesh for the set P.

Proposition 1: Let P be a set of n = 2k+1 points on the grid. Then, there exists a unique point (x,y) on the grid such that $\sum_{i=1}^{n}(|x_i - x| + |y_i - y|)$ is minimum.

Proof : Let X = $\{x_i | x_i$ is the x-coordinate of $p_i \in P\}$ and let Y = $\{y_i | y_i$ is the y-coordinates of $p_i \in P\}$. We assume, without loss of generality, that $x_i \leq x_{i+1}$ and $yi \leq y_{i+1}$; $1 \leq i \leq 2k$. As shown in Figure 14 the point $(x,y) = (x_{k+1}, y_{k+1})$ minimizes the distance : $D = \sum_{i=1}^{n}(|x_i - x_{k+1}| + |y_i - y_{k+1}|)$.

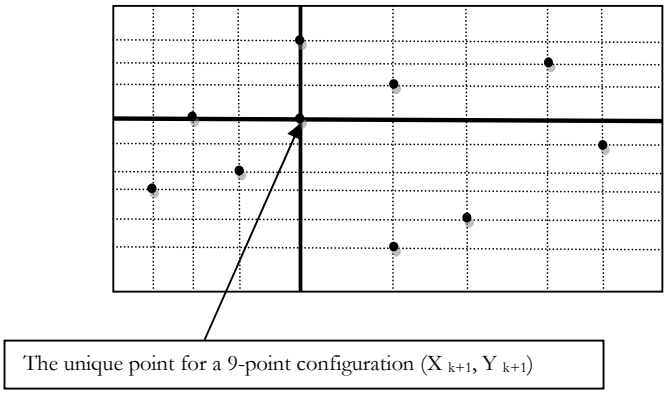


Figure 14. The (x_{k+1}, y_{k+1}) point minimizes the distance D

For every point $(x', y') = (x_{k+1}+d_1, y_{k+1}+d_2)$ on the grid, where $d_1, d_2 \in R$ and not both equal to zero, we will show that $D' > D$ with $D' = \sum_{i=1}^{n}(|x_i - x'| + |y_i - y'|)$. Let $d_1 \geq 0$, $d_2 \geq 0$ and $d_1+d_2 \neq 0$. Then,

$$D' = \sum_{i=1}^{2k+1}(|x_i - x_{k+1} - d_1| + |y_i - y_{k+1} - d_2|)$$
$$= \sum_{i=1}^{k}(|x_i - x_{k+1} - d_1| + |x_{k+1} - x_{k+1} - d_1| + \sum_{i=k+2}^{2k+1}(|x_i - x_{k+1} - d_1|)$$
$$+ \sum_{i=1}^{k}(|y_i - y_{k+1} - d_2|) + |y_{k+1} - y_{k+1} - d_2| + \sum_{i=k+2}^{2k+1}(|y_i - y_{k+1} - d_2|)$$

Since, $x_i \leq x_{k+1} + d_1$, $y_i \leq y_{k+1} + d_2$ for $1 \leq i \leq k$ we have

$$|x_i - x_{k+1} - d_1| = |x_i - x_{k+1}| + d_1,$$
$$|y_i - y_{k+1} - d_2| = |y_i - y_{k+1}| + d_2$$

Also since, $x_i \geq x_{k+1} + d_1$, $y_i \geq y_{k+1} + d_2$ for $k+2 \leq i \leq 2k+1$ we have

$$|x_i - x_{k+1} - d_1| \geq |x_i - x_{k+1}| - |d_1|,$$
$$|y_i - y_{k+1} - d_2| \geq |y_i - y_{k+1}| - |d_2|$$

Thus,

$$D' \geq \sum_{i=1}^{k}(|x_i - x_{k+1}|) + kd_1 + d_1 + \sum_{i=k+2}^{2k+1}(|x_i - x_{k+1}| - |kd_1|$$

$$+ \sum_{i=1}^{k}(|y_i - y_{k+1}| + kd_2 + d_2 + \sum_{i=k+2}^{2k+1}(|y_i - y_{k+1}| - |kd_2|$$

Since, $d_1 \geq 0$ and $d_2 \geq 0$ then $D' \geq D + d_1 + d_2$.

Since, d_1 and d_2 are not both zero it follows that $D' > D$.

Similar proofs for the other three cases:

$d_1 \geq 0$, $d_2 \leq 0$ and $|d_1| + |d_2| \neq 0$,

$d_1 \leq 0$, $d_2 \geq 0$ and $|d_1| + |d_2| \neq 0$,

$d_1 \leq 0$, $d_2 \leq 0$ and $|d_1| + |d_2| \neq 0$,

result in $D' > D$.

Therefore, the unique point that minimizes the summation of the distances from all points of P is the point (x_{k+1}, y_{k+1}).

Proposition 2: Let P be a set of n=2k points in the grid. Then, there exists a submesh of points on the grid such that $\sum_{i=1}^{n} |x_i - x| + |y_i - y|$ is minimum for every point (x,y) in the submesh.

Proof: Let X, Y be the x and y coordinates of all points in P respectively. Select the elements x_k, x_{k+1} to be the k^{th} and $(k+1)^{st}$ largest elements of X. Select similarly the y_k and the y_{k+1} elements of Y. Consider the submesh defined by the four corner points, (x_k, y_k), (x_k, y_{k+1}), (x_{k+1}, y_k) and (x_{k+1}, y_{k+1}) as shown in Figure 15. For any point (x,y) of the submesh the

$$D = \sum_{i=1}^{n} |x_i - x| + |y_i - y|$$

is minimum.

There are two possible moves of the point (x,y) by a distance d; one move inside the submesh (perimeter included) and the other move outside the submesh. In the first case, from the definition of the submesh, when (x,y) moves to the right by a distance d>0 we have:

$$D' = \sum_{i=1}^{k} |x_i - (x + d)| + \sum_{i=k+1}^{2k} |x_i - (x - d)| + \sum_{i=1}^{2k} |y_i - y|$$

Since, $|x_i - (x + d)| = (x + d) - x_i$ for $1 \leq i \leq k$ and $|x_i - (x - d)| = x_i - (x + d)$ for $k+1 \leq i \leq 2k$, we have $D' = D$.

When (x,y) moves to the right and outside the submesh by a distance d>0 we have:

$$D' = \sum_{i=1}^{k+1} |x_i - (x + d)| + \sum_{i=k+l+1}^{2k} |x_i - (x - d)| + \sum_{i=1}^{2k} |y_i - y|,$$

where l>k. Because k+1>2k-k-l-1 = k-l-1, there are more x-coordinates that increase by d than the ones that decrease by d. Therefore, D < D′.

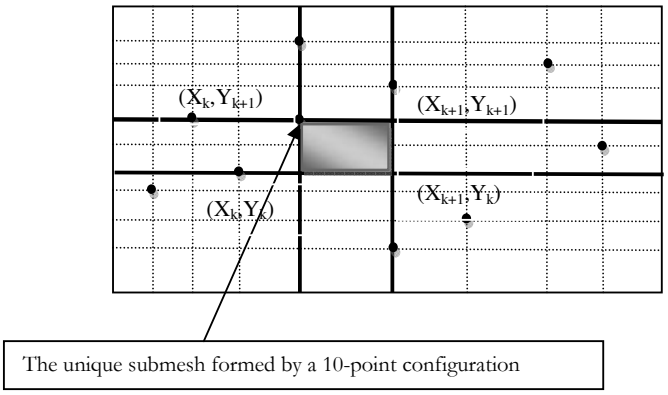


Figure 15. Creation of the submesh for an even number of points.

Arguing similarly for the remaining outside moves of (x, y) we conclude that it is always D < D′. The above two propositions appear in a different context in [51].

3.3.1 The ENVELOPE Algorithm

The propositions proved above are used to develop a polynomial algorithm that approximates the Minimum Steiner Tree of a finite number of points. The EVELOPE algorithm was first presented by Kountanis in [51] for the calculation of the Rectilinear Steiner Tree and we present a different version of the ENVELOPE algorithm which calculates the Euclidean Minimum Steiner Tree which is the Minimum Steiner Tree of an oppnet. Intuitively the algorithm works as follows:

Successive envelopes for P are obtained starting with the outmost envelope and moving inwards until an envelope is reached that contains only one point. If an envelope contains an odd number of points, then the unique defined by Proposition 1 is added to the points to be considered in successive envelopes. If an envelope n = 2k > 2 points then a submesh is defined by Proposition 2. When n=2, one of these points is selected, according to the following criteria: If only one of the two points belongs to the original set P then, this point has higher preference over the other. If both belong to the original set P then, the one with the maximum degree is selected. If the degree is the same for both points then, the one which is closer to the point with the maximum degree of the immediately next envelope is selected.

Note that the number of Potential Steiner points, which are added for further processing, is always less than the number of points in the corresponding envelope. Therefore, convergence of the algorithm is guaranteed.

At the beginning, every point in P has a Degree of 0 which means that all of them have the same preference to be chosen as starting points of local Minimum Steiner Tree's. The degree of added points increases by 1 over the max degree among all points in its corresponding envelope. A point with higher degree is preferable for starting point of local MST's over other points with lower degrees.

The construction of the MST starts with the point, possibly not in P, that has the maximum degree. We connect the points from each envelope to the existed MST, moving from inside out. In each envelope, only the points that belong to the original set P are connected to the MST. The points on the envelope are connected to the MST in order, according to their distances from the MST with shortest distance first. Points which are Potential Steiner points on each envelope, are used to break ties of equidistant point occurrences. The connection of the points on the envelope uses the following criteria listed in terms of priority:

1. Always connect first, the point with the shortest distance to the existed MST. If more than one points are equidistant from the MST then,
2. Choose the one that can be connected to the tree forming a path that has the maximum overlap with its corresponding envelope boundary. If there are more than such points then,
3. Select the point that is closer to the point with the maximum degree on the current envelope.

Note that Potential Steiner points may fall on a connection path of other points. This does not always mean that such points are going to be final Steiner points. These points simply direct the formation of each edge in the tree. The final Steiner points are the ones with degree of 3 or more.

The algorithm ends when all points in P are connected. Below in Figure 16 we can see the implemented algorithm.

Algorithm: ENVELOPE

Input: The set P={(x_i, y_i) where $1 \leq i \leq n$ are the given points}
The set X of the x-coordinates of the points in P
The set Y of the y-coordinates of the points in P
Output: A Minimum Steiner Tree

STEP 1
Let C be a stack of sets /* the stack holds the points of the envelopes */
C=NULL;
D=0; /*The degree of the preference of points*/
Let A={(x_i, y_i, D_i) where (x_i, y_i) belongs in P and D_i is its degree}

WHILE $|A| \geq 2$ DO
Find the x_{min}, x_{max} of x_i's where $(x_i, y_i, D_i) \in A$
Find the y_{min}, y_{max} of y_i's where $(x_i, y_i, D_i) \in A$
Let $B=\{(x_i, y_i, D_i) \in A$ where $x_i=x_{max}$ OR $x_i=x_{min}$
OR $y_i=y_{min}$ OR $y_i=y_{max}\}$ /* push the envelope B in the stack*/
A=A-B
IF $|B|=2k+1$ /* for some integer k*/ THEN
Select the Medians x_{med} and y_{med} of x_i's and y_i's from $(x_i,y_i,D_i) \in B$
Find D_{max}=maximum D_i where $(x_i, y_i, D_i) \in B$
$D_{max} = D_{max}+1$
Let $(x_s,y_s,D_s)=(x_{med},y_{med},D_{max})$ the unique point of Proposition 1
$A=A \cup \{(x_s,y_s,D_s)\}$ /* add the unique point */
ELSE /* the cardinality of B is even (Proposition 2) */
Select the k^{th} and the $(k+1)^{st}$ x_i's and y_i's from $(x_i,y_i,D_i) \in B$
Let R be the unique submesh defined by the corners
(x_k, y_k), (x_k, y_{k+1}), (x_{k+1}, y_k), (x_{k+1}, y_{k+1})
Select two corner points
Find D_{max}=maximum D_i where $(x_i,y_i,D_i) \in B$
$D_{max} = D_{max}+1$
Let (x_s,y_s,D_{max}) and (x_s',y_s',D_{max}) be the selected points
$A = A \cup \{(x_s,y_s,D_{max})), (x_s',y_s',D_{max}))\}$
ENDIF
ENDWHILE

STEP 2
Let (x_{ss}, y_{ss}, D_{ss}) be the remaining point in A
/* This is the starting point of the MST */
WHILE $C \neq$ NULL DO
POP(C,B) /* obtain the current envelope points from stack */
$B = P \cap \{\pi_1(B), \pi_2(B)\}$ /*(π_1, π_2) give the (x, y) from (x, y, D) */
WHILE there still points in B DO
Find a point in B which is the closest to the existed MST
IF there are more than one closest points from the point to MST THEN
Choose the one that overlaps the most with the current envelope boundary
IF There are still more than one such points consider THEN
Choose the closest to the maximum degree point of the current envelope
IF there are still more than one such points consider THEN
Choose one randomly
ENDIF
ENDIF
ENDIF
ENDWHILE
ENDWHILE

Figure 16. The Euclidean Steiner Tree ENVELOPE Algorithm

The ENVELOPE algorithm that is presented above consists of two steps: Step 1 consists of a WHILE loop. The maximum number of times that this loop will be executed equals to the number of envelopes generated. In the worst case this number will be (n – 1), where n is the number of points in P. The worst case will be when each one of the envelopes contains only two points among the points P and the added points. This implies that the number of envelopes in the worst case is O(n). Each of the two FIND operations inside the WHILE loop has complexity O(n). The MAX and MIN operations can be carried out in time O(n). The same is true for the A=A-B operation. The calculation of the Median of a set is O(n) which is also true for the UNION. Therefore, the order of complexity of step 1 is $O(n^2)$. Step 2 consists of two nested WHILE loops. The outside WHILE loop iterates r times, where r is the total number of envelopes constructed in step 1. On the other hand, the maximum number of envelopes that can be formed in step 1 is (n-1). Therefore, the maximum number of iterations due to the outside WHILE loop is O(n). Because, in the worst case, the number of points in each envelope is 2, the inside WHILE loop will iterate O(1) times. The POP operation as well as the INTERSECTION operation of the inside WHILE loop are O(n) operations. The operation of finding the closest distance of a point to the MST can be carried out also in time O(n). Each one of the IF statements in the inner WHILE loop has time complexity O(1). Therefore, the time complexity of step 2 is $O(n^2)$. Thus, the overall complexity of the algorithm is $O(n^2)$.

3.4 Solution of Problem 2

In order to provide the solution for problem 2 we have to set some standards for our oppnet deployment scenario. As shown in Figure 1 in order to create an oppnet we have to have at least one seed that has all the communication capabilities, mentioned in solution of Problem 1 sub-problem 1, such as a GPS receiver, a wireless communication 916/433MHz sensor, a usb Bluetooth 2.4GHz dongle and a wireless Internet 802.11 b/g 2.4GHz adapter . That, at least one, seed will play the role of the Distributed Command Center. The Distributed Command Center using all its communications capabilities will establish a remote VPN link to the Oppnet Command Control Center which may be very far away and will be used by the administrator of the Oppnet. The DCC is the one that has to try to scan for the rest of the seeds that were scattered in the drop zone and start to communicate with them in order to create the network tree of the oppnet. Also it is DCC's role to recognize which of the nodes are seeds or wireless sensors. The seeds due to their moving capabilities have a more energetic role in the oppnet on the contrary

with the wireless sensors that are not capable of moving and in general most of the times play the role of re-transmitters of the commands or messages. Seconds before dropped the seeds and the wireless sensors of the oppnet are enabled in order to start the procedure of scanning in all their communications mediums for near neighbors. After they scan for neighbors, they start an "advertising" procedure sending their exact geographical location to all their neighbors in order each node to "know" the location of its neighbors. The administrator of the oppnet from the OCCC sends the command to the DCC in order to initiate the oppnet. Then the DCC scans for its near neighbor nodes and orders all the nodes to report their geographical location back to it. After all the nodes have reported their geographical location back to the DCC the DCC runs the ENVELOPE algorithm in order to create the Minimum Steiner Tree which is the Minimum Communication Tree of the oppnet. An authentication procedure is started in order the nodes to be authorized to join the oppnet. After a node is authorized it enters the "listen", "check" and "forward" states in order to listen the commands from the DCC, check if the command is for it and forward the command or a message to the node it is destined if not for it. The DCC can drop a node from the oppnet if it decides that is not useful or the oppnet does not need to grow anymore and order a seed to move to a different geographical location in order to be able to achieve the communication goals of the oppnet.

Conclusively, we can state that the oppnet deployment scenario that we just presented reflects a multi-functional network that is dynamically growing or minimizing according to its communication goals. The human factor, which in our case is the administrator of the OCCC, still remains the decision maker of the communication goals of the network and takes full advantage of the mobility features of the oppnet.

Below in Tables 4 and 5 we can see the sets of the proposed message-commands that could be used in the oppnet to achieve its communication goals.

Command	**Function of the command**
DCC_go	Initiate the oppnet
DCC_stop	Shutdown the oppnet
DCC_order	Send command to the nodes

Table 4. *The set of commands send from the OCCC to the DCC.*

Command	Function of the command
Node_detect	Scan communication spectrum to detect active nodes.
Node_recon	Discover nodes with specific capabilities.
Node_listen	Receive and save commands or messages in a buffer.
Node_check	Verify if the received command or message is for it or not.
Node_forward	Process a message or command from the buffer.
Node_authendicate	Evaluate a node and admit it into the oppnet if needed.
Node_disengage	Drop a node from the oppnet if not authorized or not needed.
Node_report	Report information to the DCC.
Node_move	Order a seed to move to different geographical location.

Table 5. *The set of commands send from DCC to the nodes of the oppnet*.

Chapter IV

4. Developed Simulation and Experimental Results

In this Chapter we will present the software we developed for the construction of the Euclidean Minimum Steiner Tree of an oppnet and a comparison of our experimental results of our Steiner software with the Geosteiner 3.1 software developed by David Warme, Pawel Winter and Martin Zachariasen [52]. The seeds of our oppnet are simulated as Steiner points from a topology point of view. In order to develop the Steiner software we used strictly open source development tools that run in Open source Operating Systems. More specifically, we used GCC 4.4 (GNU Compiler Collection) which is a compiler system produced by the GNU Project [54] supporting various programming languages. GCC is a key component of the GNU tool chain. As well as being the official compiler of the unfinished GNU operating system, GCC has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including Linux Ubuntu 8.10. [53] The software and the experimental results were developed on a Laptop with Pentium 1,6 GHz processor, 1GB of RAM memory and Linux Ubuntu 8.10 Operating System.

4.1 The Steiner software

As mentioned before in Chapter III the Minimum Steiner Tree for our oppnet topology is the Euclidian Steiner Tree of seeds. That is the reason that motivated us to develop an application that constructs Euclidian Steiner Trees and represents them in graph form in postscript files. In order to develop the Steiner application we used C programming language which is the basic language of the GCC 4.4 compiler [54]. In general our application consists of a polynomial random point generator (archive rand_points.c) that generates random points of a given size, two archives (e_steiner.c and envelope.c) that compute and construct the Minimum Steiner Trees using the points generated by the random point generator, an archive fst2graph.c that transforms the structure of the trees to graphs and an archive plotfst.c that creates the postscripts files from the graphs. The source code of the files fst2graph.c and plotfst.c was not developed by us and is taken from the open source software Geosteiner 3.1 [52].

The random point generator takes as input an integer number of points, for example 30 points, and creates a random set of them with the above source code.

```
if (oflag NE 0) {
            if (use_new_generator) {
                  seed_to_string (sbuf, 16, "0x");
            }
            else {
                  seed_to_string (sbuf, 10, "");
            }
            fprintf ((oflag EQ 1) ? stdout : stderr,
                   "%% rand_points: seed = %s\n", sbuf);
      }
      for (i = 0; i < npoints; i++) {
            x = random_coordinate ();
            if (NOT use_new_generator) {
                  (void) __rand ();
            }
            y = random_coordinate ();
            (void) printf ("%5lu %5lu\n", x, y);
      }
      if (fflag NE 0) {
            if (use_new_generator) {
                  seed_to_string (sbuf, 16, "0x");
            }
            else {
                  seed_to_string (sbuf, 10, "");
            }
            fprintf ((oflag EQ 1) ? stdout : stderr,
                   "%% rand_points: final state = %s\n", sbuf);
      }
      exit (0);
```

After the random generator has produced the random points the file e_steiner.c which is the Euclidean generator reads that point set from the random point generator, generates the structure of the tree, and outputs them to standard output. Then the envelope.c file computes the Euclidean Minimum Steiner Tree.

4.2 Experimental results

We used the envelope version of our algorithm to develop the e_steiner tree for various topologies. The topologies were generated by the random generator functionality shown above. The data sets for our experiments include 30,40,50,60,70,80,90 and 100 points. For each data set category we generated 10 data collections. More specifically our random point generator for 30 points generated 10 sets of 30 points each. The same happened for all the other data set categories. For each category we found all the generated Minimum Steiner Trees. The time for the Steiner Tree generation was calculated based on the machine used for the implementation. We used the necessary clock ( ) functions in UNIX for the time calculations.

Using the above methodology we found for each data set category the average time spent for the tree calculation. The following subsections show various screenshots for the Steiner trees visualization. As a visualization tool we used the ps-visualizer which is

included in the Geosteiner 3.1 [52] software version in order to have better compatibility between that software and our software. More thoroughly we created:

Ten ps files of 30 points Minimal Steiner trees like the one shown in Figure 17.

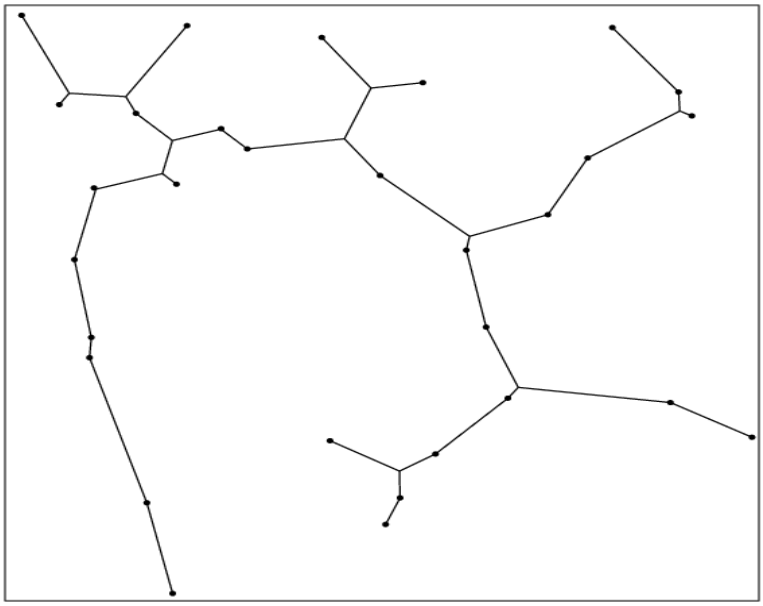


Figure 17. *A Steiner Minimum Tree of 30 points*

Ten ps files of 40 points Minimal Steiner trees like the one shown in Figure 18.

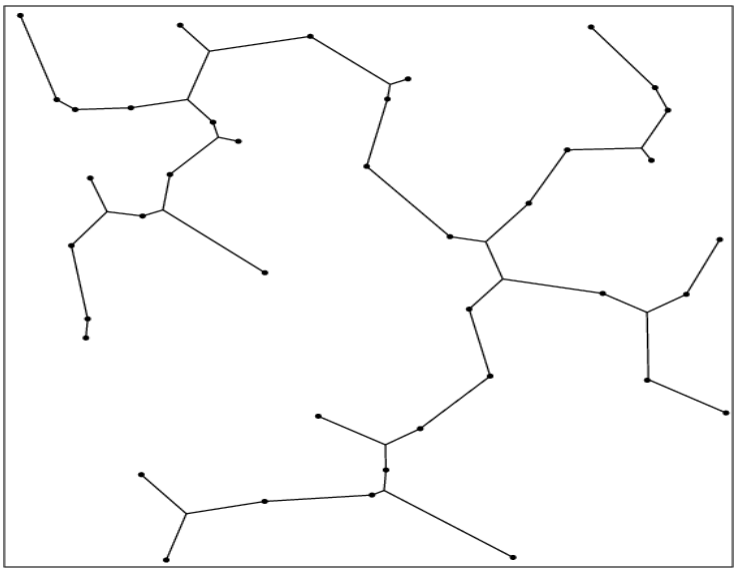


Figure 18. *A Steiner Minimum Tree of 40 points*

Ten ps files of 50 points Minimal Steiner trees like the one shown in Figure 19.

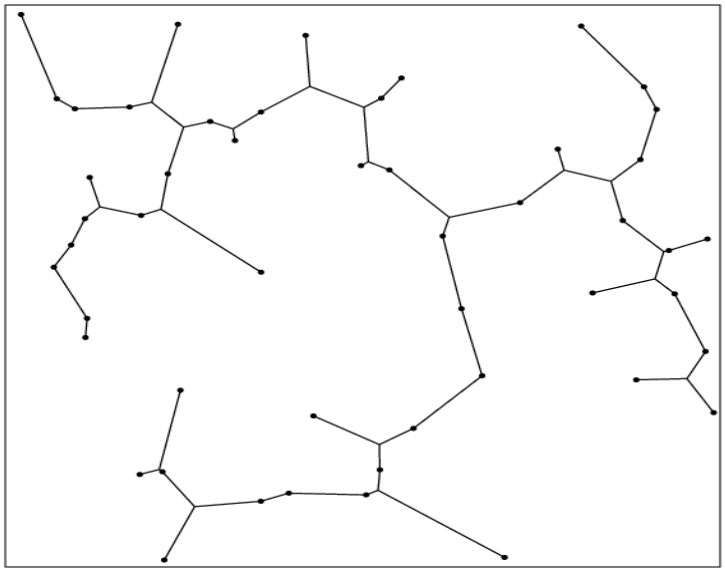


Figure 19. *A Steiner Minimum Tree of 50 points*

Ten ps files of 60 points Minimal Steiner trees like the one shown in Figure 20.

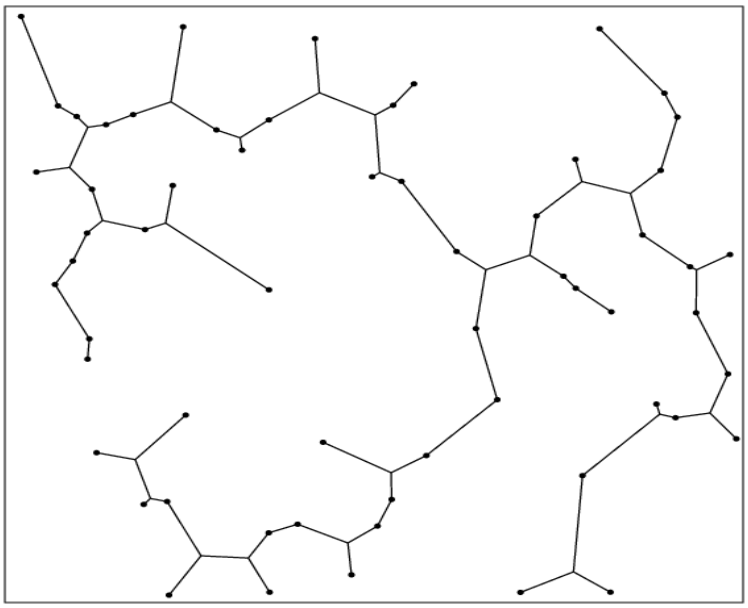


Figure 20. *A Steiner Minimum Tree of 60 points*

Ten ps files of 70 points Minimal Steiner trees like the one shown in Figure 21.

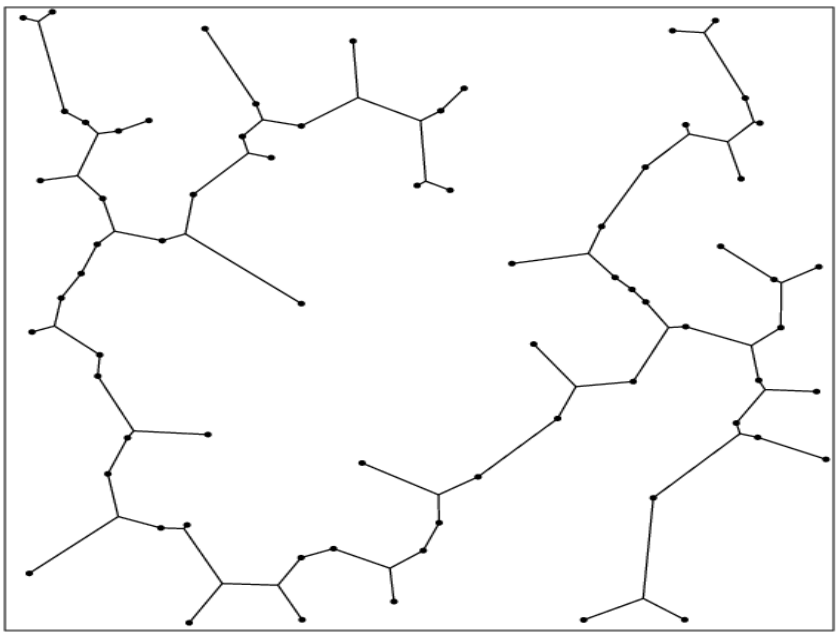


Figure 21. *A Steiner Minimum Tree of 70 points*

Ten ps files of 80 points Minimal Steiner trees like the one shown in Figure 22.

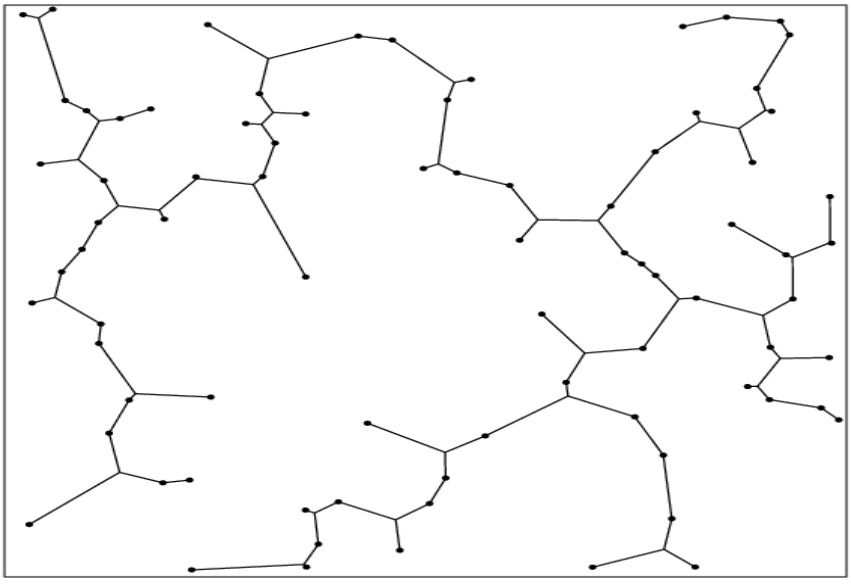


Figure 22. *A Steiner Minimum Tree of 80 points*

Ten ps files of 90 points Minimal Steiner trees like the one shown in Figure 23.

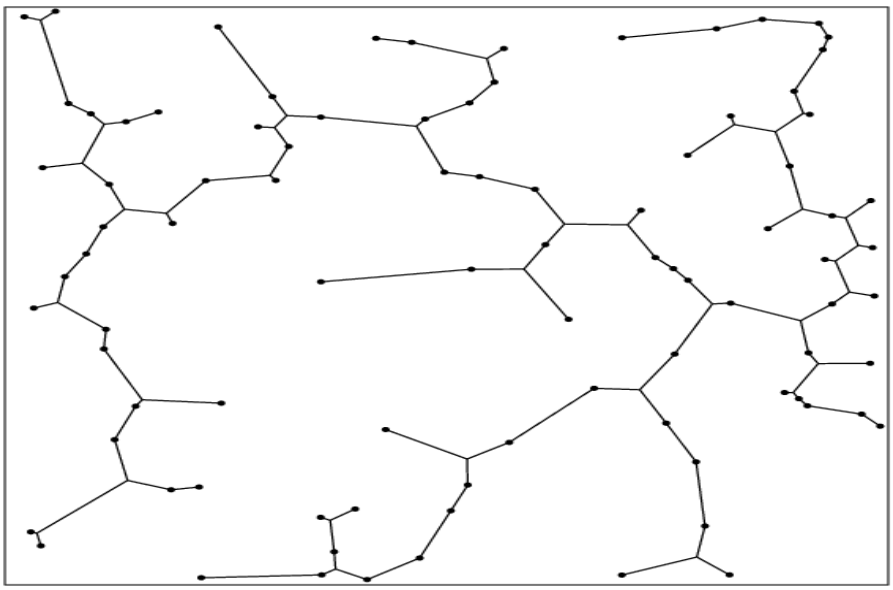


Figure 23. *A Steiner Minimum Tree of 90 points*

Ten ps files of 100 points Minimal Steiner trees like the one shown in Figure 24.

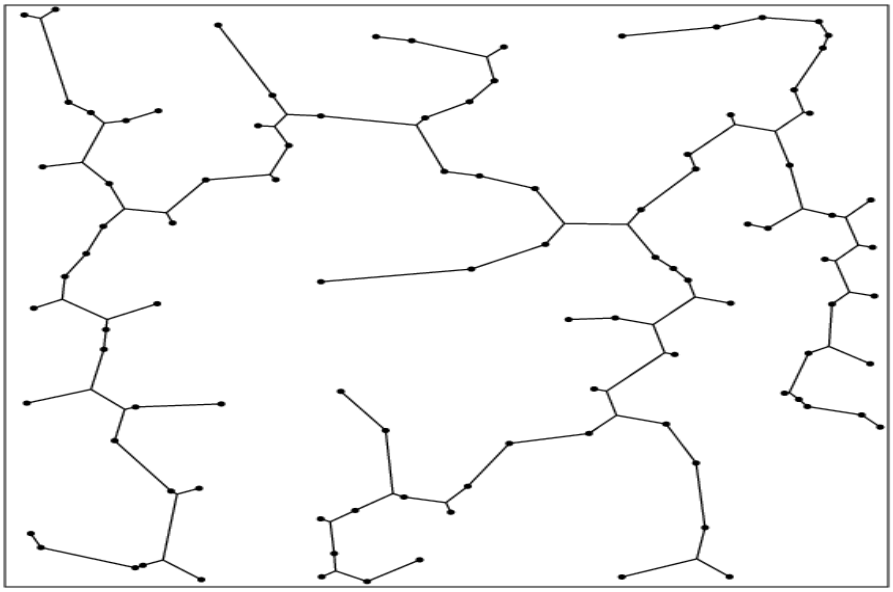


Figure 24. *A Steiner Minimum Tree of 100 points*

One ps file of 200 points Minimal Steiner trees like the one shown in Figure 25.

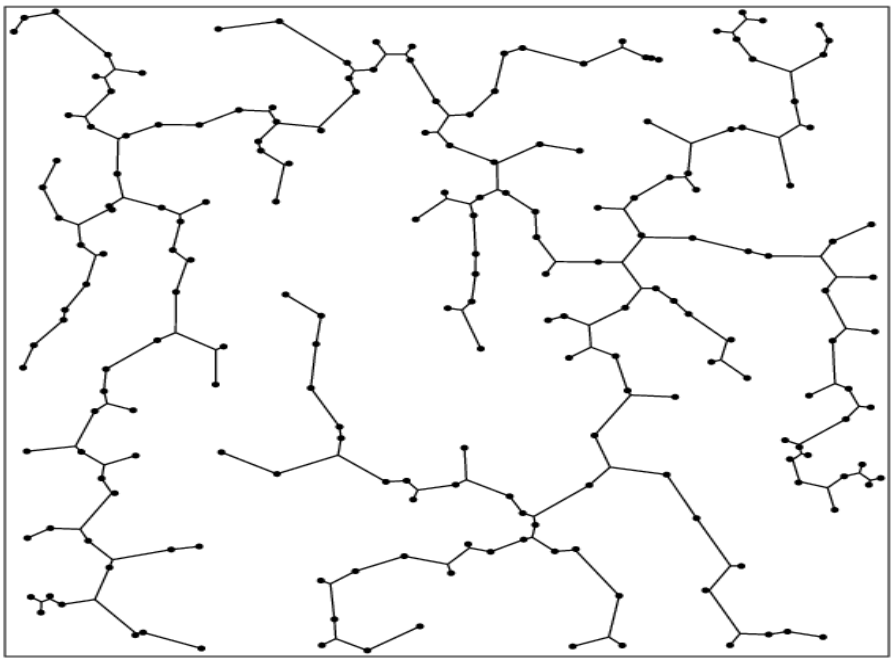


Figure 25. *A Steiner Minimum Tree of 200 points*

One ps file of 300 points Minimal Steiner trees like the one shown in Figure 26.

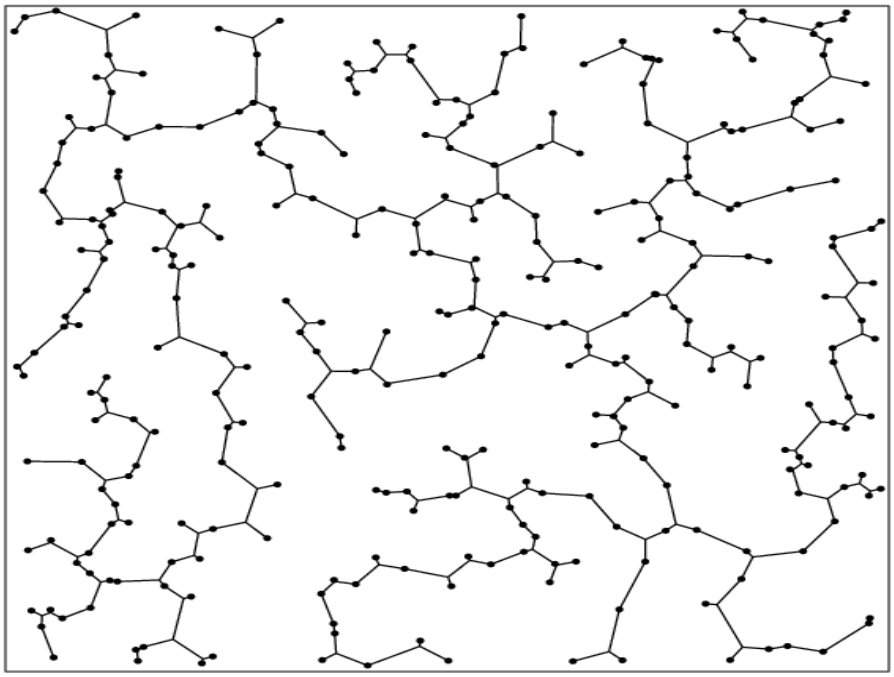


Figure 26. *A Steiner Minimum Tree of 300 points*

One ps file of 400 points Minimal Steiner trees like the one shown in Figure 27.

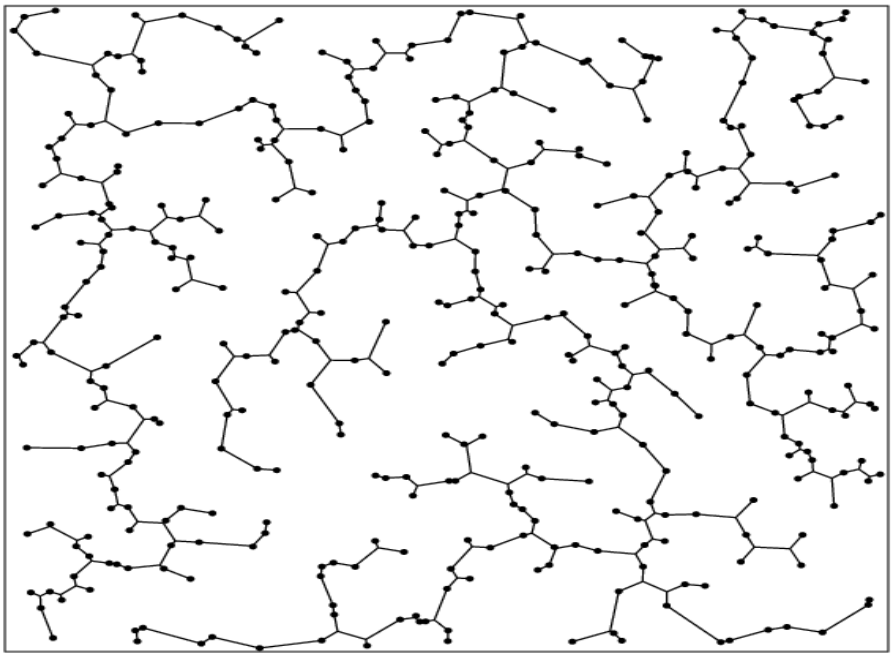


Figure 27. *A Steiner Minimum Tree of 400 points*

One ps file of 500 points Minimal Steiner trees like the one shown in Figure 28.

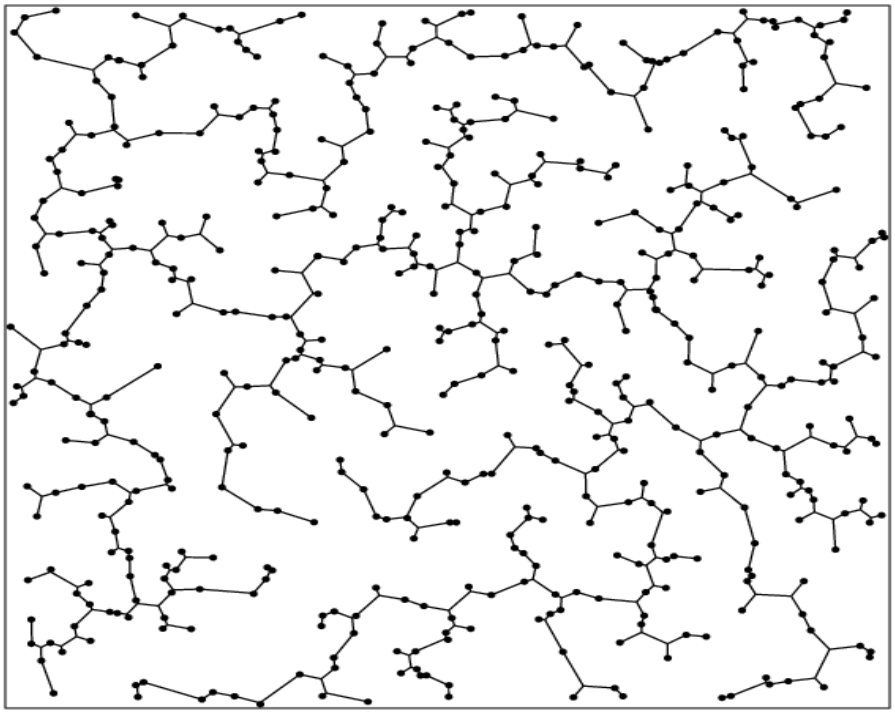


Figure 28. *A Steiner Minimum Tree of 500 points*

In Table 6 we can clearly see the experimental results relatively to the time needed for the Steiner tree calculation.

Number of points	Time calculation for Steiner software in seconds
30	0.22
40	0.52
50	0.83
60	1.24
70	1.97
80	2.67
90	3.16
100	3.87

Table 6. *The time calculation results of the Steiner application.*

We plotted the above table values as it is illustrated in Diagram 1. It is clear to see that the experimental results verify the pre-calculated time complexity of $O(n^2)$ of the ENVELOPE algorithm.

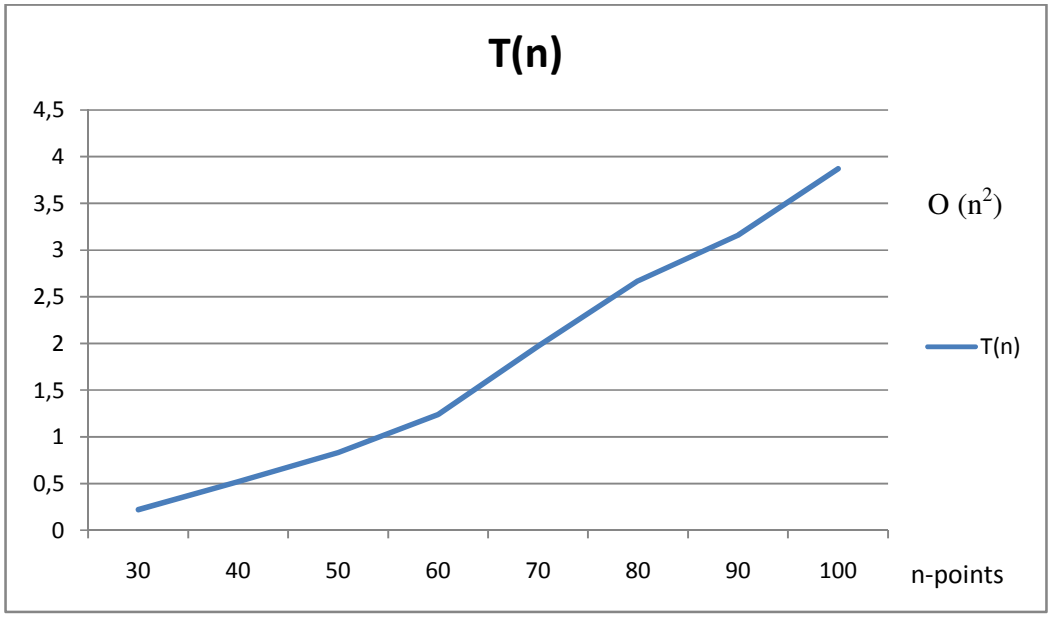


Diagram 1. *The experimental results of the Steiner application plotted.*

We also compared our generated data sets of points relatively to the length of the created Steiner Trees as opposed to the ones generated by the Geosteiner 3.1 [52] software. This computation is depicted below in Table 7.

Number of points	Envelope MST length	Geosteiner 3.1 MST length
30	37095,88	37092,44
40	45788,13	45789,54
50	48600,81	48599,86
60	54931,31	54929,12
70	59591,22	59593,28
80	62058,82	62056,46
90	65558,07	65557,03
100	69474,94	69473,88

Table 7. *The Steiner trees length computation comparison of the two algorithms.*

We can clearly observe that our algorithm is fully compatible with the Geosteiner 3.1 [52] algorithm since the lengths of the calculated Steiner trees are of the same order.

4.3 Conclusions and Future work

Conclusively, in this dissertation we tried our best to approach the two basic problems that concerned us as thoroughly as we could. For the Problem one sub-problem one we proposed the UNION algorithm which we believe that is an algorithm that can be implemented within a more general application software for the oppnets. Approaching Problem one sub-problem two we can say that the Euclidean Steiner Tree ENVELOPE algorithm we proposed is more than certain that can be a solution in the communication tree problem that oppnets face in field deployment conditions. In the future the ENVELOPE algorithm can be approached also from a more energy efficient communication point of view in order to be more realistic within the live oppnets applications scenarios. Since the oppnet's basic applications and infrastructure are not yet clearly standardized within the research community the proposed by us, in Problem two, set of messages-commands can be a starting point for future research and field experiments for the future oppnet applications. The experimental results of the Steiner software we presented prove in simulation conditions the functionality of the ENVELOPE algorithm and they generate to us many hopes that Steiner application can be an oppnet application by itself or a part of an oppnet Operational System of course after being tested in the field and live oppnet deployment conditions.

Chapter V

5. Bibliography

1. B.Bhargava, L.Lilien, A. Rosenthal, and M. Winslett, 'Pervasive Trust', IEEE Intelligent Systems vol.19(5), Sep/Oct.2004, pp.74-77.
2. L. Lilien, Z. H. Kamal, V. Bhuse, and A. Gupta, "Opportunistic Networks: The Concept and Research Challenges in Privacy and Security," *Proc. of the Intl. Workshop on Research Challenges in Security and Privacy for Mobile and Wireless Networks (WSPWN 06)*, Miami, March 2006, pp. 134-147.
3. R. Subramanian, and B. Goodman, "Peer-to-Peer Computing: The Evolution of a Disruptive Technology," Hershey, PA, USA: Idea Group Publishing, 2005.
4. *Wireless Ad Hoc Networks* Zygmunt J. Haas, Jing Deng, Ben Liang, Panagiotis Papadimitratos, and S. Sajama Cornell University School of Electrical and Computer Engineering.
5. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design Matei Ripeanu , Ian Foster ,Adriana Iamnitchi Computer Science Department, The University of Chicago.
6. A. Gupta, D. Agrawal, and A.E. Abbadi, "Distributed Resource Discovery in Large Scale Computing," *SAINT 200*5.
7. "On Star Explained," last accessed on November 26, 2005. Available at: http://www.onstar.com/us_english/jsp/explore/index.jsp.
8. Opportunistic Sensor Networks (Oppnets) Leszek Lilien, Zille H. Kamal, Ajay Gupta, Vijay Bhuse, and Zijiang Yang, *Members, IEEE*.
9. http://en.wikipedia.org/wiki/Wireless_sensor_network
10. Standard Implementation Framework for Opportunistic Networks in Emergency Preparedness and Response Applications Leszek Lilien, Ajay Gupta, , Zijiang Yang, Wireless Sensornet Laboratory (WiSe Lab), Department of Computer Science Western Michigan University, Kalamazoo.
11. Pervasive Trust Bharat Bhargava and Leszek Lilien, Purdue University Arnon Rosenthal, The MITRE Corp. Marianne Winslett, University of Illinois at Urbana- Champaign.
12. Wireless Ecosystems: Building the Next Generation of Pervasive Computing Systems Prof. Narayan Mandayam WINLAB, Rutgers University NJ.
13. From wireless to sensor networks and convergence: Capacity, architecture, and protocols P. R. Kumar G Baliga, V. Borkar, A. Giridhar, S. Graham, P. Gupta, V. Kawadia, S. Narayanaswamy, R. Rozovsky, V. Raghunathan, L-L. Xie) Dept. of Electrical and Computer Engineering, and Coordinated Science Lab University of Illinois, Urbana-Champaign.
14. Geographic Routing with Limited Information in Sensor Networks Sundar Subramanian and Sanjay Shakkottai.
15. Erasure-Coding Based Routing for Opportunistic Networks Yong Wang, Sushant Jain, Margaret Martonosi, Kevin Fall Princeton University, University of Washington, Intel Research Berkeley.
16. Analyzing Object Detection Quality Under Probabilistic Coverage in Sensor Networks Shansi Ren, Qun Li, Haining Wang, Xin Chen, and Xiaodong Zhang College of William and Mary Williamsburg.
17. S. Iyenger and R. Brooks, Distributed Sensor Networks, CRC Press, Inc. 2003.
18. J. Hightower, G. Borriello, "Location Systems for Ubiquitous Computing," IEEE Computer, August 2001.

19. Mobile ad hoc networking: imperatives and challenges Imrich Chlamtac , Marco Conti , Jennifer J.-N. Liu
20. Dissertation Proposal Optimizing the seed oppnet infrastructure Panagiotis Tsamados Msc Computer Science Staffordshire University 2008.
21. Topology Control in Wireless Ad Hoc and Sensor networks. Paolo Santi 2005 Willey publications.
22. Lilien L. A taxonomy of specialised ad hoc networks and systems for emergency applications. In: Procceedings of the 1st International workshop on mobile and ubiquitous context aware systems and applications (MUBICA 2007), Philadelfia, Pennsylvania,2007
23. The first international workshop on specialized ad hoc networks and systems (SAHNS 2007), Toronto 2007.
24. Rappaport T 2002 Wireless Communications: Principles and Practice, Second Edition. Prentice Hall, Upper Saddle River, NJ.
25. http://www.cisco.com/en/US/products/hw/wireless
26. The Rockwell wins project available at http://wins.rsc.rockwell.com
27. Raghunathan V, Schurgers C, Park S and Srivastava M 2002 Energy-aware wireless microsensor networks. *IEEE Signal Processing Magazine* 19(2), 40–50.
28. Bettstetter C 2001a Mobility modeling in wireless networks: categorization, smooth movement, and border effects. *ACM Mobile Computing and Communications Review* 5(3), 55–67.
29. Bettstetter C and Krause O 2001 On border effects in modeling and simulation of wireless ad hoc networks. *Proc. IEEE International Conference on Mobile and Wireless Communication Network (MWCN)*, Recife.
30. Johnson D and Maltz D 1996 Dynamic source routing in ad hoc wireless networks. *Mobile Computing*. Kluwer Academic Publishers, pp. 153–181.
31. Royer E, Melliar-Smith P and Moser L 2001 An analysis of the optimum node density for ad hoc mobile networks. *Proc. IEEE International Conference on Communications*, Helsinki, pp. 857–861.
32. Santi P and Blough D 2003 The critical transmitting range for connectivity in sparse wireless ad hoc networks. *IEEE Transactions on Mobile Computing* **2**(1), 25–39.
33. Bai F, Sadagopan N and Helmy A 2003 Important: a framework to systematically analyze the impact of mobility on performance of routing protocols for ad hoc networks. *Proc. IEEE Infocom*, San Francisco, CA, pp. 825–835.
34. Hong X, Gerla M, Pei G and Chiang C 1999 A group mobility model for ad hoc wireless networks. *Proc. ACM MSWiM*, Seattle, WA, pp. 53–60.
35. Kamal Z, Gupta A, Lilien L, The MicroOppnet Tool for Collaborative Computing Experiments with Class 2 Opportunistic Networks, Western Michigan University 2007.
36. Zeng X, Bagrodia R and Gerla M 1998 Glomosim: a library for parallel simulation of large-scale wireless networks. *Workshop on Parallel and Distributed Simulations (PADS)*, Banff, Alberta
37. Ns2 2002 The network simulator - ns-2 http://www.isi.edu/nsnam/ns/.
38. Narayanaswamy S, Kawadia V, Sreenivas R and Kumar P 2002 Power control in ad hoc networks: theory, architecture, algorithm and implementation of the compow protocol. *Proc. European Wireless 2002*, Florence, pp. 156–162
39. Blough D, Leoncini M, Resta G and Santi P 2003b The *k*-neighbors protocol for symmetric topology control in ad hoc networks. *Proc. ACM MobiHoc 03*, Annapolis, MD, pp. 141–152.

40. Krizman K, Biedka T and Rappaport T 1997 Wireless position location: fundamentals, implementation strategies, and source of error. *Proc. IEEE Vehicular Technology Conference (VTC)*, Phoenix, AZ, pp. 919–923.
41. Li L, Halpern J, Bahl P, Wang Y and Wattenhofer R 2001 Analysis of a cone-based distributed topology control algorithm for wireless multi-hop networks. *Proc. ACM PODC 01*, Newport, RI, pp. 264–273.
42. Borbash S and Jennings E 2002 Distributed topology control algorithm for multihop wireless networks. *Proc. IEEE International Joint Conference on Neural Networks*, Honolulu, HI, pp. 355–360.
43. Kirousis L, Kranakis E, Krizanc D and Pelc A 2000 Power consumption in packet radio networks. *Theoretical Computer Science* 243, 289–305
44. Song W, Wang Y, Li X and Frieder O 2004 Localized algorithms for energy efficient topology in wireless ad hoc networks. *Proc. ACM MobiHoc*, Tokyo, pp. 98–108.
45. Rodoplu V and Meng T 1999 Minimum energy mobile wireless networks. IEEE Journal Selected Areas in Communication 17(8), 1333–1344.
46. Lynch N 1996 *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA.
47. Li N, Hou J and Sha L 2003 Design and analysis of an mst-based topology control algorithm. *Proc. IEEE Infocom 03*, San Francisco, CA, pp. 1702–1712.
48. Prim R 1957 Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 1389–1401.
49. http://en.wikipedia.org/wiki/Minimum_spanning_tree
50. http://en.wikipedia.org/wiki/Steiner_minimum_tree
51. D. Kountanis and K. Kokkinos "A Balanced approach to the rectilinear Steiner Tree Problem" Congressus Numeratioum 108 pp.205-221 Winipeg Canada 1995
52. Geosteiner 3.1. David Warme, Emergent Information Technologies, Inc., Virginia, USA Pawel Winter, University of Copenhagen, Denmark Martin Zachariasen, University of Copenhagen, Denmark 2001.
53. http://en.wikipedia.org/wiki/GNU_Compiler_Collection
54. http://www.gnu.org/gnu/thegnuproject.html

APENDIX I Source code

In APENDIX I we present typical screenshots of the source code of Steiner application and Geosteiner application 3.1.[52]

rand_points.c

```
/*****************************************************************
      This program generates random point sets.
*/

#include "steiner.h"
#include <time.h>
int               main (int, char **);
#define     MAX_COORD   10000
static int16u           __rand (void);
static void       convert_seed (char *);
static int32u           cv_number (char *);
static void       decode_params (int, char **);
static void       generate_remainder_table (void);
static int32u           rand64 (int32u);
static int32u           random_coordinate (void);
static void       seed_to_string (char *, int32u, char *);
static void       usage (void);
static int        fflag;
static int32u           hi_seed;
static int32u           lo_seed;
static char *           me;
static int        npoints;
static int        oflag;
static bool       use_new_generator;

/*
 * This routine generates random point sets.
 */
      int main ( int argc, char ** argv )
{
int               i;
int32u                  x;
int32u                  y;
char              sbuf [32];
      setbuf (stdout, NULL);
      decode_params (argc, argv);
      if (oflag NE 0) {
            if (use_new_generator) {
                  seed_to_string (sbuf, 16, "0x");
            }
            else {
                  seed_to_string (sbuf, 10, "");
            }
            fprintf ((oflag EQ 1) ? stdout : stderr,
                   "%% rand_points: seed = %s\n",
                   sbuf);
      }
      for (i = 0; i < npoints; i++) {
            x = random_coordinate ();
            if (NOT use_new_generator) {
                  (void) __rand ();
            }
            y = random_coordinate ();
```

```
                (void) printf ("%5lu %5lu\n", x, y);
        }
        if (fflag NE 0) {
                if (use_new_generator) {
                        seed_to_string (sbuf, 16, "0x");
                }
                else {
                        seed_to_string (sbuf, 10, "");
                }
                fprintf ((oflag EQ 1) ? stdout : stderr,
                         "%% rand_points: final state = %s\n",
                         sbuf);
        }
        exit (0);
}
/*
 * Decoding of line arguments.
 */
static      void decode_params (int argc,char ** argv)
{
char *              ap;
char        c;
        --argc;
        me = *argv++;
        fflag = 0;
        oflag = 0;
        hi_seed = 1;
        lo_seed = 0;
        npoints = 10;
        while (argc > 0) {
                ap = *argv++;
                if (*ap NE '-') {
                        npoints = cv_number (ap);
                        break;
                }
                ++ap;
                while ((c = *ap++) NE '\0') {
                        switch (c) {
                        case 'e':
                                oflag = 2;
                                break;
                        case 'f':
                                fflag = 1;
                                break;
                        case 'o':
                                oflag = 1;
                                break;
                        case 'n':
                                use_new_generator = TRUE;
                                break;
                        case 's':
                                if (*ap EQ '\0') {
                                        if (argc <= 0) {
                                                usage ();
                                        }
                                        ap = *argv++;
                                        --argc;
                                }
                                convert_seed (ap);
                                ap = "";
                                break;
                        case 'r':
```

```
                        hi_seed = time (NULL);
                        lo_seed = 0;
                        break;
                  default:
                        usage ();
                        break;
                  }
            }
            --argc;
      }
}
/*
 * This routine prints out the proper usage and exits.
 */
static char *     arg_doc [] = {
      "",
      "\tGenerate N random points.  Default N is 10.",
      "",
      "\t-e\tWrite initial seed to stderr.",
      "\t-f\tWrite final seed to stderr (or stdout if -o given).",
      "\t-o\tWrite initial seed (and final seed if -f given) to
stdout.",
      "\t-n\tUse newer (better) random generator.",
      "\t-s K\tSet random seed to K.",
      "\t-r\tRandomize: use current time as seed.",
      NULL
};
      static      void usage (void)
{
char **           pp;
char *            p;
      (void) fprintf (stderr,
                  "\nUsage: %s [-efonr] [-s seed] [N]\n",
                  me);
      pp = &arg_doc [0];
      while ((p = *pp++) NE NULL) {
            (void) fprintf (stderr, "%s\n", p);
      }
      exit (1);
}
/*
 * Convertion the given string into a number.
 */
static      int32u cv_number ( char * num_string )
{
char *            s;
int         c;
int32u            val;
      s = num_string;
      val = 0;
      while ((c = *s++) NE '\0') {
            if (NOT isdigit (c)) {
                  (void) fprintf (stderr,
                              "%s: `%s' is not a decimal number.\n",
                              me, s);
                  exit (1);
            }
            val = (val * 10) + (c - '0');
      }
      return (val);
}
/*
```

```
 * Convertion of the given ASCII string into a seed.
 */
static      void convert_seed ( char * string) /* ASCII string to get
seed from. */
{
char *            p;
int         i;
int         c;
int32u            carry;
int32u            base;
int16u            digs [4];
      p = string;
      base = 10;
      if ((p [0] EQ '0') AND (tolower (p [1]) EQ 'x')) {
            base = 16;
            p += 2;
      }
      for (i = 0; i < 4; i++) {
            digs [i] = 0;
      }
      for (;;) {
            c = *p++;
            if (c EQ '\0') {
                  hi_seed = (((int32u) digs [2]) << 16) + digs [3];
                  lo_seed = (((int32u) digs [0]) << 16) + digs [1];
                  return;
            }
            if (c < '0') c = -1;
            else if (c <= '9') c -= '0';
            else if (c < 'A') c = -1;
            else if (c <= 'F') c -= ('A' - 10);
            else if (c < 'a') c = -1;
            else if (c <= 'f') c -= ('a' - 10);
            else c = -1;
            if ((c < 0) OR (c >= base)) {
                  fprintf (stderr,
                        "Warning: invalid seed: %s\n",
                        string);
                  hi_seed = 1;
                  lo_seed = 0;
                  return;
            }
            carry = c;
            for (i = 3; i >= 0; i--) {
                  carry += digs [i] * base;
                  digs [i] = carry;
                  carry >>= 16;
            }
      }
}
/*
 * Convertion of the current seed into a printable string
 */
static      void seed_to_string (char *   buf,int32u base,char *
prefix)
{
int         i;
int32u            rem;
char *            dstp;
int16u            digs [4];
char        tmp [32];
      while (*prefix NE '\0') {
```

```
                *buf++ = *prefix++;
        }
        if ((lo_seed EQ 0) AND (hi_seed EQ 0)) {
                *buf++ = '0';
                *buf++ = '\0';
                return;
        }
        digs [3] = (lo_seed >> 16);
        digs [2] = lo_seed;
        digs [1] = (hi_seed >> 16);
        digs [0] = hi_seed;
        dstp = &tmp [sizeof (tmp) / sizeof (tmp [0])];
        *--dstp = '\0';
        while ((digs [0] NE 0) OR (digs [1] NE 0) OR
               (digs [2] NE 0) OR (digs [3] NE 0)) {
                rem = 0;
                for (i = 3; i >= 0; i--) {
                        rem = (rem << 16) + digs [i];
                        digs [i] = rem / base;
                        rem %= base;
                }
                *--dstp = "0123456789ABCDEF" [rem];
        }
        (void) strcpy (buf, dstp);
}
/*
 * Computation of a new random coordinate.
 */
static      int32u random_coordinate (void)
{
int32u              n;
        if (use_new_generator) {
                return (rand64 (MAX_COORD));
        }
        n = (((int32u) __rand ()) >> 2) ^ (((int32u) __rand ()) << 3);
        return (n % MAX_COORD);
}
static      int16u__rand (void)
{
long        x;
        x = 1103515245L * ((long) hi_seed) + 12345L;
        hi_seed = x;
        return ((x >> 16) & 0x7FFF);
}
#define     GEN_POLY_HI 0x19B8AB86
#define     GEN_POLY_LO 0x8766D1B9
static int32u           remainders [512] = {
        0x00000000, 0x00000000, 0x9ac99dbb, 0x11ea8045,
        0x3b5e9804, 0x10a45787, 0xa19705bf, 0x014ed7c2,
        0x76bd3008, 0x2148af0e, 0xec74adb3, 0x30a22f4b,
        0x4de3a80c, 0x31ecf889, 0xd72a35b7, 0x200678cc,
        0xed7a6010, 0x42915e1c, 0x77b3fdab, 0x537bde59,
        0xd624f814, 0x5235099b, 0x4ced65af, 0x43df89de,
        0x9bc75018, 0x63d9f112, 0x010ecda3, 0x72337157,
        0xa099c81c, 0x737da695, 0x3a5055a7, 0x629726d0,
        0xd4396352, 0xb653eb35, 0x4ef0fee9, 0xa7b96b70,
        0xef67fb56, 0xa6f7bcb2, 0x75ae66ed, 0xb71d3cf7,
        0xa284535a, 0x971b443b, 0x384dcee1, 0x86f1c47e,
        0x99dacb5e, 0x87bf13bc, 0x031356e5, 0x965593f9,
        0x39430342, 0xf4c2b529, 0xa38a9ef9, 0xe528356c,
        0x021d9b46, 0xe466e2ae, 0x98d406fd, 0xf58c62eb,
        0x4ffe334a, 0xd58a1a27, 0xd537aef1, 0xc4609a62,
```

```
0x74a0ab4e, 0xc52e4da0, 0xee6936f5, 0xd4c4cde5,
0xa6bf65d7, 0x5fd68167, 0x3c76f86c, 0x4e3c0122,
0x9de1fdd3, 0x4f72d6e0, 0x07286068, 0x5e9856a5,
0xd00255df, 0x7e9e2e69, 0x4acbc864, 0x6f74ae2c,
0xeb5ccddb, 0x6e3a79ee, 0x71955060, 0x7fd0f9ab,
0x4bc505c7, 0x1d47df7b, 0xd10c987c, 0x0cad5f3e,
0x709b9dc3, 0x0de388fc, 0xea520078, 0x1c0908b9,
0x3d7835cf, 0x3c0f7075, 0xa7b1a874, 0x2de5f030,
0x0626adcb, 0x2cab27f2, 0x9cef3070, 0x3d41a7b7,
0x72860685, 0xe9856a52, 0xe84f9b3e, 0xf86fea17,
0x49d89e81, 0xf9213dd5, 0xd311033a, 0xe8cbbd90,
0x043b368d, 0xc8cdc55c, 0x9ef2ab36, 0xd9274519,
0x3f65ae89, 0xd86992db, 0xa5ac3332, 0xc983129e,
0x9ffc6695, 0xab14344e, 0x0535fb2e, 0xbafeb40b,
0xa4a2fe91, 0xbbb063c9, 0x3e6b632a, 0xaa5ae38c,
0xe941569d, 0x8a5c9b40, 0x7388cb26, 0x9bb61b05,
0xd21fce99, 0x9af8ccc7, 0x48d65322, 0x8b124c82,
0x43b368dc, 0x8cdc55c3, 0xd97af567, 0x9d36d586,
0x78edf0d8, 0x9c780244, 0xe2246d63, 0x8d928201,
0x350e58d4, 0xad94facd, 0xafc7c56f, 0xbc7e7a88,
0x0e50c0d0, 0xbd30ad4a, 0x94995d6b, 0xacda2d0f,
0xaec908cc, 0xce4d0bdf, 0x34009577, 0xdfa78b9a,
0x959790c8, 0xdee95c58, 0x0f5e0d73, 0xcf03dc1d,
0xd87438c4, 0xef05a4d1, 0x42bda57f, 0xfeef2494,
0xe32aa0c0, 0xffa1f356, 0x79e33d7b, 0xee4b7313,
0x978a0b8e, 0x3a8fbef6, 0x0d439635, 0x2b653eb3,
0xacd4938a, 0x2a2be971, 0x361d0e31, 0x3bc16934,
0xe1373b86, 0x1bc711f8, 0x7bfea63d, 0x0a2d91bd,
0xda69a382, 0x0b63467f, 0x40a03e39, 0x1a89c63a,
0x7af06b9e, 0x781ee0ea, 0xe039f625, 0x69f460af,
0x41aef39a, 0x68bab76d, 0xdb676e21, 0x79503728,
0x0c4d5b96, 0x59564fe4, 0x9684c62d, 0x48bccfa1,
0x3713c392, 0x49f21863, 0xadda5e29, 0x58189826,
0xe50c0d0b, 0xd30ad4a4, 0x7fc590b0, 0xc2e054e1,
0xde52950f, 0xc3ae8323, 0x449b08b4, 0xd2440366,
0x93b13d03, 0xf2427baa, 0x0978a0b8, 0xe3a8fbef,
0xa8efa507, 0xe2e62c2d, 0x322638bc, 0xf30cac68,
0x08766d1b, 0x919b8ab8, 0x92bff0a0, 0x80710afd,
0x3328f51f, 0x813fdd3f, 0xa9e168a4, 0x90d55d7a,
0x7ecb5d13, 0xb0d325b6, 0xe402c0a8, 0xa139a5f3,
0x4595c517, 0xa0777231, 0xdf5c58ac, 0xb19df274,
0x31356e59, 0x65593f91, 0xabfcf3e2, 0x74b3bfd4,
0x0a6bf65d, 0x75fd6816, 0x90a26be6, 0x6417e853,
0x47885e51, 0x4411909f, 0xdd41c3ea, 0x55fb10da,
0x7cd6c655, 0x54b5c718, 0xe61f5bee, 0x455f475d,
0xdc4f0e49, 0x27c8618d, 0x468693f2, 0x3622e1c8,
0xe711964d, 0x376c360a, 0x7dd80bf6, 0x2686b64f,
0xaaf23e41, 0x0680ce83, 0x303ba3fa, 0x176a4ec6,
0x91aca645, 0x16249904, 0x0b653bfe, 0x07ce1941,
0x8766d1b9, 0x19b8ab86, 0x1daf4c02, 0x08522bc3,
0xbc3849bd, 0x091cfc01, 0x26f1d406, 0x18f67c44,
0xf1dbe1b1, 0x38f00488, 0x6b127c0a, 0x291a84cd,
0xca8579b5, 0x2854530f, 0x504ce40e, 0x39bed34a,
0x6a1cb1a9, 0x5b29f59a, 0xf0d52c12, 0x4ac375df,
0x514229ad, 0x4b8da21d, 0xcb8bb416, 0x5a672258,
0x1ca181a1, 0x7a615a94, 0x86681c1a, 0x6b8bdad1,
0x27ff19a5, 0x6ac50d13, 0xbd36841e, 0x7b2f8d56,
0x535fb2eb, 0xafeb40b3, 0xc9962f50, 0xbe01c0f6,
0x68012aef, 0xbf4f1734, 0xf2c8b754, 0xaea59771,
0x25e282e3, 0x8ea3efbd, 0xbf2b1f58, 0x9f496ff8,
0x1ebc1ae7, 0x9e07b83a, 0x8475875c, 0x8fed387f,
0xbe25d2fb, 0xed7a1eaf, 0x24ec4f40, 0xfc909eea,
```

```
        0x857b4aff, 0xfdde4928, 0x1fb2d744, 0xec34c96d,
        0xc898e2f3, 0xcc32b1a1, 0x52517f48, 0xddd831e4,
        0xf3c67af7, 0xdc96e626, 0x690fe74c, 0xcd7c6663,
        0x21d9b46e, 0x466e2ae1, 0xbb1029d5, 0x5784aaa4,
        0x1a872c6a, 0x56ca7d66, 0x804eb1d1, 0x4720fd23,
        0x57648466, 0x672685ef, 0xcdad19dd, 0x76cc05aa,
        0x6c3a1c62, 0x7782d268, 0xf6f381d9, 0x6668522d,
        0xcca3d47e, 0x04ff74fd, 0x566a49c5, 0x1515f4b8,
        0xf7fd4c7a, 0x145b237a, 0x6d34d1c1, 0x05b1a33f,
        0xba1ee476, 0x25b7dbf3, 0x20d779cd, 0x345d5bb6,
        0x81407c72, 0x35138c74, 0x1b89e1c9, 0x24f90c31,
        0xf5e0d73c, 0xf03dc1d4, 0x6f294a87, 0xe1d74191,
        0xcebe4f38, 0xe0999653, 0x5477d283, 0xf1731616,
        0x835de734, 0xd1756eda, 0x19947a8f, 0xc09fee9f,
        0xb8037f30, 0xc1d1395d, 0x22cae28b, 0xd03bb918,
        0x189ab72c, 0xb2ac9fc8, 0x82532a97, 0xa3461f8d,
        0x23c42f28, 0xa208c84f, 0xb90db293, 0xb3e2480a,
        0x6e278724, 0x93e430c6, 0xf4ee1a9f, 0x820eb083,
        0x55791f20, 0x83406741, 0xcfb0829b, 0x92aae704,
        0xc4d5b965, 0x9564fe45, 0x5e1c24de, 0x848e7e00,
        0xff8b2161, 0x85c0a9c2, 0x6542bcda, 0x942a2987,
        0xb268896d, 0xb42c514b, 0x28a114d6, 0xa5c6d10e,
        0x89361169, 0xa48806cc, 0x13ff8cd2, 0xb5628689,
        0x29afd975, 0xd7f5a059, 0xb36644ce, 0xc61f201c,
        0x12f14171, 0xc751f7de, 0x8838dcca, 0xd6bb779b,
        0x5f12e97d, 0xf6bd0f57, 0xc5db74c6, 0xe7578f12,
        0x644c7179, 0xe61958d0, 0xfe85ecc2, 0xf7f3d895,
        0x10ecda37, 0x23371570, 0x8a25478c, 0x32dd9535,
        0x2bb24233, 0x339342f7, 0xb17bdf88, 0x2279c2b2,
        0x6651ea3f, 0x027fba7e, 0xfc987784, 0x13953a3b,
        0x5d0f723b, 0x12dbedf9, 0xc7c6ef80, 0x03316dbc,
        0xfd96ba27, 0x61a64b6c, 0x675f279c, 0x704ccb29,
        0xc6c82223, 0x71021ceb, 0x5c01bf98, 0x60e89cae,
        0x8b2b8a2f, 0x40eee462, 0x11e21794, 0x51046427,
        0xb075122b, 0x504ab3e5, 0x2abc8f90, 0x41a033a0,
        0x626adcb2, 0xcab27f22, 0xf8a34109, 0xdb58ff67,
        0x593444b6, 0xda1628a5, 0xc3fdd90d, 0xcbfca8e0,
        0x14d7ecba, 0xebfad02c, 0x8e1e7101, 0xfa105069,
        0x2f8974be, 0xfb5e87ab, 0xb540e905, 0xeab407ee,
        0x8f10bca2, 0x8823213e, 0x15d92119, 0x99c9a17b,
        0xb44e24a6, 0x988776b9, 0x2e87b91d, 0x896df6fc,
        0xf9ad8caa, 0xa96b8e30, 0x63641111, 0xb8810e75,
        0xc2f314ae, 0xb9cfd9b7, 0x583a8915, 0xa82559f2,
        0xb653bfe0, 0x7ce19417, 0x2c9a225b, 0x6d0b1452,
        0x8d0d27e4, 0x6c45c390, 0x17c4ba5f, 0x7daf43d5,
        0xc0ee8fe8, 0x5da93b19, 0x5a271253, 0x4c43bb5c,
        0xfbb017ec, 0x4d0d6c9e, 0x61798a57, 0x5ce7ecdb,
        0x5b29dff0, 0x3e70ca0b, 0xc1e0424b, 0x2f9a4a4e,
        0x607747f4, 0x2ed49d8c, 0xfabeda4f, 0x3f3e1dc9,
        0x2d94eff8, 0x1f386505, 0xb75d7243, 0x0ed2e540,
        0x16ca77fc, 0x0f9c3282, 0x8c03ea47, 0x1e76b2c7,
};
#if 0
static      void generate_remainder_table (void)
{
int         i;
int         j;
int32u            lo;
int32u            hi;
      for (i = 0; i < 256; i++) {
            lo = 0;
            hi = i;
```

```
            for (j = 0; j < 8; j++) {
                  if ((hi & 0x00000001) != 0) {
                        hi = (lo << 31) | (hi >> 1);
                        lo >>= 1;
                        hi ^= GEN_POLY_HI;
                        lo ^= GEN_POLY_LO;
                  }
                  else {
                        hi = (lo << 31) | (hi >> 1);
                        lo >>= 1;
                  }
            }
            printf ("\t0x%08lx,\t0x%08lx,", lo, hi);
            if ((i & 0x01) == 1) {
                  printf ("\n");
            }
      }
}
#endif

static      int32u rand64 (int32u modulus)
{
int32u            hi;
int32u            lo;
int32u *    p;
int         i;
int32u            rem;
#define     SHIFT8 \
      p = &remainders [2 * (hi & 0xFF)]; \
      hi = ((lo << 24) | (hi >> 8)) ^ p [1]; \
      lo = (lo >> 8) ^ p [0]
      hi = hi_seed;
      lo = lo_seed;
      SHIFT8;
      SHIFT8;
      SHIFT8;
      SHIFT8;
      hi_seed = hi;
      lo_seed = lo;
      if (modulus < 0x10000) {
            rem = lo % modulus;
            rem = (rem << 16) + (hi >> 16);
            rem %= modulus;
            rem = (rem << 16) + (hi & 0xFFFF);
            return (rem % modulus);
      }
      rem = lo % modulus;
      for (i = 0; i < 32; i++) {
            rem <<= 1;
            if ((hi & 0x80000000) NE 0) {
                  ++rem;
            }
            rem %= modulus;
            hi <<= 1;
      }
      return (rem);
}
```

e_steiner.c

```
/*****************************************************************
***
      The Euclidean tree generator.  It
      reads a point set from standard input, generates the tree,
      and outputs them to standard output.
*/

#include "bsd.h"
#include "e_steiner.h"
#include "efuncs.h"
#include "egmp.h"
#include "p1io.h"
#include "steiner.h"
int               main (int, char **);
int               Multiple_Precision = 0;
static void       add_zero_length_fsts (struct einfo *, int, int **);
static void       build_fst_list (struct einfo *);
static void       build_efst_graph (struct einfo *, struct point *,
struct eqp_t *,
                        struct point **, struct edge **, int *,
struct point *, int *);
static void       compute_efsts_for_unique_terminals (struct einfo
*);
static void       convert_delta_cpu_time (char *);
static void       decode_params (int, char **);
static int        generate_duplicate_terminal_groups (struct pset *,
int *, int ***);
static cpu_time_t get_delta_cpu_time (void);
static int *            heapsort_x (struct pset *);
static struct full_set ** put_trees_in_array (struct full_set *, int
*);
static struct pset *    remove_duplicates (struct pset * pts,int ndg,
int ** list,int ** fwd_map,
                                   int **    rev_map);
static void       renumber_terminals (struct einfo *, struct pset *,
int *);
static dist_t           test_and_save_fst (struct einfo *, struct
eqp_t *, struct eqp_t *);
static void       usage (void);
static char *           description;
static int        InitialEqPointsTerminal = 100;
static int        EpsilonFactor = 32;
static char *           me;
static int        MaxFSTSize = 0;
static int        output_version = CURRENT_P1IO_VERSION;
static bool       Use_Greedy_Heuristic = FALSE;
static bool       Print_Detailed_Timings = FALSE;
static cpu_time_t T0;
static cpu_time_t Tn;
#define UPDATE_PTR(p,old,new) ((new) + ((p) - (old)))
#define UPDATE_RECTANGLE_BOUNDS(p) \
      { *minx = MIN(*minx, p.x); *maxx = MAX(*maxx, p.x); \
        *miny = MIN(*miny, p.y); *maxy = MAX(*maxy, p.y); }
/*
 * The main "e_steiner" . It reads a point set from standard input,
 generates all of the trees,and outputs them to standard output in
our special "phase 1 I/O"
 * format.
 */
      int main ( int    argc, char ** argv)
```

```
{
int               i;
int               j;
int               k;
int               ndg;
int               ntrees;
int               count;
int               fpsave;
int **                  dup_grps;
int *             fwd_map;
int *             rev_map;
int *             ip1;
struct full_set * fsp;
struct pset *           terms;
struct einfo            einfo;
struct cinfo            cinfo;
struct pset *           pts;
struct pset *           pts2;
cpu_time_t        Trenum;
struct scale_info scale;
char              buf1 [32];
      fpsave = set_floating_point_double_precision ();
      setbuf (stdout, NULL);
      decode_params (argc, argv);
      pts = get_points (stdin, &scale);
      init_output_conversion (pts, EUCLIDEAN, &scale);
      T0 = get_cpu_time ();
      Tn = T0;
      einfo.x_order = heapsort_x (pts);
      if (Print_Detailed_Timings) {
            convert_delta_cpu_time (buf1);
            fprintf (stderr, "Sort X:                 %s\n", buf1);
      }
      /* Find all duplicate terminals in the input. */
      ndg = generate_duplicate_terminal_groups (pts, einfo.x_order,
&dup_grps);
      einfo.num_term_masks = BMAP_ELTS (pts -> n);
      /* Remove all but the first of each duplicate terminal. */
      pts2 = remove_duplicates (pts, ndg, dup_grps, &fwd_map,
&rev_map);
      j = 0;
      for (i = 0; i < pts -> n; i++) {
            k = einfo.x_order [i];
            if ((k < 0) OR (pts -> n < k)) {
                  fatal ("main: Bug 1.");
            }
            k = fwd_map [k];
            if (k < 0) continue;
            einfo.x_order [j++] = k;
      }
      if (Print_Detailed_Timings) {
            convert_delta_cpu_time (buf1);
            fprintf (stderr, "Remove Duplicates:      %s\n", buf1);
      }
      einfo.pts = pts2;
      compute_efsts_for_unique_terminals (&einfo);
      renumber_terminals (&einfo, pts, rev_map);
      build_fst_list (&einfo);
      if (ndg > 0) {
            add_zero_length_fsts (&einfo, ndg, dup_grps);
      }
      Trenum = get_delta_cpu_time ();
```

```
      if (Print_Detailed_Timings) {
            convert_cpu_time (Trenum, buf1);
            fprintf (stderr, "Renumber Terminals:     %s\n", buf1);
            convert_cpu_time (Tn - T0, buf1);
            fprintf (stderr, "Total:                  %s\n", buf1);
      }
      memset (&cinfo, 0, sizeof (cinfo));
      cinfo.num_edges               = einfo.ntrees;
      cinfo.num_verts               = einfo.pts -> n;
      cinfo.num_vert_masks          = BMAP_ELTS (cinfo.num_verts);
      cinfo.num_edge_masks          = BMAP_ELTS (cinfo.num_edges);
      cinfo.edge              = NEWA (einfo.ntrees + 1, int *);
      cinfo.edge_size               = NEWA (einfo.ntrees, int);
      cinfo.cost              = NEWA (einfo.ntrees, dist_t);
      cinfo.tflag             = NEWA (cinfo.num_verts, bool);
      cinfo.metric                  = EUCLIDEAN;
      cinfo.scale             = scale;
      cinfo.integrality_delta       = 0;
      cinfo.pts               = einfo.pts;
      cinfo.full_trees        = put_trees_in_array
(einfo.full_sets,&ntrees);
      cinfo.mst_length        = einfo.mst_length;
      cinfo.description       = gst_strdup (description);
      cinfo.p1time                  = Tn - T0;
      for (i = 0; i < cinfo.num_verts; i++) {
            cinfo.tflag [i] = TRUE;
      }
      count = 0;
      for (i = 0; i < einfo.ntrees; i++) {
            fsp = cinfo.full_trees [i];
            k = fsp -> terminals -> n;
            cinfo.edge_size [i]     = k;
            cinfo.cost [i]          = fsp -> tree_len;
            count += k;
      }
      ip1 = NEWA (count, int);
      for (i = 0; i < einfo.ntrees; i++) {
            cinfo.edge [i] = ip1;
            terms = cinfo.full_trees [i] -> terminals;
            k = terms -> n;
            for (j = 0; j < k; j++) {
                  *ip1++ = terms -> a [j].pnum;
            }
      }
      cinfo.edge [i] = ip1;
      /* Prints all of the data */
      print_phase_1_data (&cinfo, output_version);
      free ((char *) pts2);
#if 0
      free ((char *) pts);
#endif
      free_phase_1_data (&cinfo);
      free ((char *) rev_map);
      free ((char *) fwd_map);
      if (dup_grps NE NULL) {
            if (dup_grps [0] NE NULL) {
                  free ((char *) (dup_grps [0]));
            }
            free ((char *) dup_grps);
      }
      free ((char *) (einfo.x_order));
```

```
	restore_floating_point_precision (fpsave);

	exit (0);
}

static	void decode_params ( int argc,char ** argv )
{
char *		ap;
char		c;
int		v;
	--argc;
	me = *argv++;
	while (argc > 0) {
		ap = *argv++;
		if (*ap NE '-') {
			usage ();
		}
		++ap;
		while ((c = *ap++) NE '\0') {
			switch (c) {
			case 'd':
				if (*ap EQ '\0') {
					if (argc <= 0) {
						usage ();
					}
					ap = *argv++;
					--argc;
				}
				if (strlen (ap) >= 80) {
					fprintf (stderr,
						"Description must be less"
						" than 80 characters.\n");
					usage ();
				}
				description = ap;
				for (;;) {
					ap = strchr (ap, '\n');
					if (ap EQ NULL) break;
					*ap++ = ' ';
				}
				ap = "";
				break;
			case 'e':
				/* Number of eq-points per terminal */
				if (*ap EQ '\0') {
					if (argc <= 0) {
						usage ();
					}
					ap = *argv++;
					--argc;
				}
				InitialEqPointsTerminal = atoi (ap);
				ap = "";
				break;
			case 'f':
				if (*ap EQ '\0') {
					if (argc <= 0) {
						usage ();
					}
					ap = *argv++;
					--argc;
				}
```

```
                        EpsilonFactor = atoi (ap);
                        ap = "";
                        break;
                  case 'k':
                        if (*ap EQ '\0') {
                              if (argc <= 0) {
                                    usage ();
                              }
                              ap = *argv++;
                              --argc;
                        }
                        MaxFSTSize = atoi (ap);
                        ap = "";
                        break;
#ifdef HAVE_GMP
                  case 'm':
                        if (*ap EQ '\0') {
                              if (argc <= 0) {
                                    usage ();
                              }
                              ap = *argv++;
                              --argc;
                        }
                        if ((*ap < '0') OR (*ap > '9')) {
                              usage ();
                        }
                        Multiple_Precision = atoi (ap);
                        ap = "";
                        break;
#endif
                  case 't':
                        Print_Detailed_Timings = TRUE;
                        break;
                  case 'v':
                        if (*ap EQ '\0') {
                              if (argc <= 0) {
                                    usage ();
                              }
                              ap = *argv++;
                              --argc;
                        }
                        if ((*ap < '0') OR (*ap > '9')) {
                              usage ();
                        }
                        v = atoi (ap);
                        if ((v < 0) OR (v > CURRENT_P1IO_VERSION)) {
                              fprintf (stderr,
                                    "%s: Bad version `%s'."
                                    "  Valid versions range"
                                    " from 0 to %d.\n", me, ap,
CURRENT_P1IO_VERSION);
                              usage ();
                        }
                        output_version = v;
                        ap = "";
                        break;
                  default:
                        usage ();
                        break;
                  }
            }
            --argc;
```

```
	}
}
static char *	arg_doc [] = {
	"",
	"\t-d txt\tDescription of problem instance.",
	"\t-e K\tInitially allocate K eq-points per terminal",
	"\t\t(default: 100).",
	"\t-f E\tEpsilon multiplication factor for floating point
number",
	"\t\tcomparisons (default: 32).",
	"\t-k K\tOnly generate points spanning up to K terminals.",
#ifdef HAVE_GMP
	"\t-m N\tUse multiple precision.  Larger N use it more.",
	"\t\t Default is N=0 which disables multiple precision.",
#endif
	"\t-t\tPrint detailed timings on stderr.",
	"\t-v N\tGenerates version N output data format.",
	"",
	NULL
};
	static void usage (void){
char **		pp;
char *		p;
	(void) fprintf (stderr,
			"\nUsage: %s [-gt]"
			" [-d description]"
			" [-e K] [-f E] [-k K]"
#ifdef HAVE_GMP
			" [-m N]"
#endif
			" [-v N]"
			" <points-file\n",
			me);
	pp = &arg_doc [0];
	while ((p = *pp++) NE NULL) {
		(void) fprintf (stderr, "%s\n", p);
	}
	exit (1);
}
/*
 * Sorting of the given terminals in increasing order
 */
static	int * heapsort_x (struct pset *	pts) /* the terminals
to sort */
{
int		i, i1, i2, j, k, n;
struct point *		p1;
struct point *		p2;
int *		index;
	n = pts -> n;
	index = NEWA (n, int);
	for (i = 0; i < n; i++) {
		index [i] = i;
	}
	/* Construct the heap via sift-downs, in O(n^2) time. */
	for (k = n >> 1; k >= 0; k--) {
		j = k;
		for (;;) {
			i = (j << 1) + 1;
			if (i + 1 < n) {
				i1 = index [i];
				i2 = index [i + 1];
```

```
                        p1 = &(pts -> a [i1]);
                        p2 = &(pts -> a [i2]);
                        if ((p2 -> x > p1 -> x) OR
                            ((p2 -> x EQ p1 -> x) AND
                             ((p2 -> y > p1 -> y) OR
                              ((p2 -> y EQ p1 -> y) AND
                               (i2 > i1))))) {
                              ++i;
                        }
                  }
                  if (i >= n) {
                        break;
                  }
                  i1 = index [j];
                  i2 = index [i];
                  p1 = &(pts -> a [i1]);
                  p2 = &(pts -> a [i2]);
                  if ((p1 -> x > p2 -> x) OR
                      ((p1 -> x EQ p2 -> x) AND
                       ((p1 -> y > p2 -> y) OR
                        ((p1 -> y EQ p2 -> y) AND
                         (i1 > i2))))) {
                        /* if greatest child is smaller.  Sift-   */
                        /* down is done. */
                        break;
                  }
                  /* Sift down and continue. */
                  index [j] = i2;
                  index [i] = i1;
                  j = i;
            }
      }
      /* Exchange first/last and sift down. */
      while (n > 1) {
            --n;
            i = index [0];
            index [0] = index [n];
            index [n] = i;
            j = 0;
            for (;;) {
                  i = (j << 1) + 1;
                  if (i + 1 < n) {
                        i1 = index [i];
                        i2 = index [i + 1];
                        p1 = &(pts -> a [i1]);
                        p2 = &(pts -> a [i2]);
                        if ((p2 -> x > p1 -> x) OR
                            ((p2 -> x EQ p1 -> x) AND
                             ((p2 -> y > p1 -> y) OR
                              ((p2 -> y EQ p1 -> y) AND
                               (i2 > i1))))) {
                              ++i;
                        }
                  }
                  if (i >= n) {
                        break;
                  }
                  i1 = index [j];
                  i2 = index [i];
                  p1 = &(pts -> a [i1]);
                  p2 = &(pts -> a [i2]);
                  if ((p1 -> x > p2 -> x) OR
```

```
                        ((p1 -> x EQ p2 -> x) AND
                         ((p1 -> y > p2 -> y) OR
                          ((p1 -> y EQ p2 -> y) AND
                           (i1 > i2))))) {
                          break;
                    }
                    index [j] = i2;
                    index [i] = i1;
                    j = i;
              }
        }
        return (index);
}
/*
 * This routine finds all pairs of points whose coordinates are
exactly
 * identical.
 */
static      int generate_duplicate_terminal_groups ( struct pset *
pts,int * xorder,int *** grps)
{
int               i;
int               j;
int               n;
int               n_grps;
struct point *          p0;
struct point *          p1;
struct point *          p2;
int *             ip;
int **                  real_ptrs;
int *             real_terms;
int **                  ptrs;
int *             terms;
      n = pts -> n;
      n_grps = 0;
      ptrs  = NEWA (n + 1, int *);
      terms = NEWA (n, int);
      ip = &terms [0];
      for (i = 1; i < n; ) {
            p0 = &(pts -> a [xorder [i - 1]]);
            p1 = &(pts -> a [xorder [i]]);
            if ((p0 -> y NE p1 -> y) OR (p0 -> x NE p1 -> x)) {
                  /* Not identical. */
                  ++i;
                  continue;
            }
            /* Terminals xorder[i-1] and xorder[i] are identical. */
            for (j = i + 1; j < n; j++) {
                  p2 = &(pts -> a [xorder [j]]);
                  if (p0 -> y NE p2 -> y) break;
                  if (p0 -> x NE p2 -> x) break;
            }
            ptrs [n_grps++] = ip;
            *ip++ = xorder [i - 1];
            while (i < j) {
                  *ip++ = xorder [i++];
            }
      }
      ptrs [n_grps] = ip;
      if (n_grps <= 0) {
            *grps = NULL;
      }
```

```
      else {
            real_ptrs = NEWA (n_grps + 1, int *);
            real_terms = NEWA (ip - terms, int);
            (void) memcpy ((char *) real_terms,(char *) terms, (ip -
terms) * sizeof (int));
            for (i = 0; i <= n_grps; i++) {
                  real_ptrs [i] = &real_terms [ptrs [i] - ptrs [0]];
            }
            *grps = real_ptrs;
      }
      free ((char *) terms);
      free ((char *) ptrs);
      return (n_grps);
}
static      struct pset * remove_duplicates (
struct pset *           pts,
int               ndg,
int **                  list,
int **                  fwd_map_ptr,
int **                  rev_map_ptr
)
{
int         i;
int         j;
int         n;
int         kmasks;
int         numdel;
int         new_n;
int *       ip1;
int *       ip2;
int *       fwd;
int *       rev;
bitmap_t *  deleted;
struct pset *     newpts;
      n = pts -> n;
      kmasks = BMAP_ELTS (n);
      deleted = NEWA (kmasks, bitmap_t);
      for (i = 0; i < kmasks; i++) {
            deleted [i] = 0;
      }
      numdel = 0;
      for (i = 0; i < ndg; i++) {
            ip1 = list [i];
            ip2 = list [i + 1];
            while (++ip1 < ip2) {
                  if (BITON (deleted, *ip1)) {
                        fatal ("remove_duplicates: Bug 1.");
                  }
                  ++numdel;
                  SETBIT (deleted, *ip1);
            }
      }
      new_n = n - numdel;
      fwd = NEWA (n, int);
      rev = NEWA (new_n, int);
      newpts = NEW_PSET (new_n);
      ZERO_PSET (newpts, new_n);
      newpts -> n = new_n;
      j = 0;
      for (i = 0; i < n; i++) {
            if (BITON (deleted, i)) {
                  fwd [i] = -1;
```

```
            }
            else {
                  newpts -> a [j].x       = pts -> a [i].x;
                  newpts -> a [j].y       = pts -> a [i].y;
                  newpts -> a [j].pnum          = j;
                  rev [j] = i;
                  fwd [i] = j;
                  ++j;
            }
      }
      free ((char *) deleted);
      *fwd_map_ptr = fwd;
      *rev_map_ptr = rev;
      return (newpts);
}
static      bool solve_quadratic (double A,double B,double C,double *
root1,double * root2)
{
      double Q, D;
      if (A EQ 0.0) return FALSE;
      D = B*B - A*C;
      if (D < 0.0)  return FALSE;
      D = sqrt(D);
      if (B >= 0.0)
            Q = - B - D;
      else
            Q = - B + D;
      if (Q NE 0.0) {
            *root1 = Q / A;
            *root2 = C / Q;
      }
      else {
            *root1 = 0.0;
            *root2 = -2.0 * B / A;
      }
      return TRUE;
}
static      bool test_and_save_LP (struct einfo * eip,struct eqp_t *
eqpi,struct eqp_t * eqpj,
struct eqp_t *    eqpk,struct point *     CLP)
{
struct point NLP;
struct point PLP;
      if ((CLP -> x EQ eqpi -> E.x) AND (CLP -> y EQ eqpi -> E.y))
return TRUE;
      if (right_turn(&(eqpk -> E), &(eqpi -> E),  CLP))
return TRUE;
      if (left_turn (&(eqpk -> E), &(eqpk -> LP), CLP))
return TRUE;
      memset (&PLP, 0, sizeof (PLP));
      memset (&NLP, 0, sizeof (NLP));
      PLP.x = CLP -> x + (eqpj -> E.x - CLP -> x) * eip -> eps *
((double) eqpk -> S);
      PLP.y = CLP -> y + (eqpj -> E.y - CLP -> y) * eip -> eps *
((double) eqpk -> S);
      project_point(&(eqpk -> E), &(eqpk -> DC), &PLP, &NLP);
      if (left_turn (&(eqpk -> E), &(eqpk -> LP), &NLP))
return TRUE;
      if (right_turn(&(eqpk -> E), &(eqpk -> RP), &NLP))
return FALSE;
      eqpk -> LP = NLP;
      return TRUE;
```

```
}
static      bool test_and_save_RP (struct einfo * eip,struct eqp_t *
eqpi,struct eqp_t * eqpj,
struct eqp_t *    eqpk,struct point *     CRP)
{
struct point NRP;
struct point PRP;
      if ((CRP -> x EQ eqpj -> E.x) AND (CRP -> y EQ eqpj -> E.y))
return TRUE;
      if (left_turn (&(eqpk -> E), &(eqpj -> E),  CRP))
return TRUE;
      if (right_turn(&(eqpk -> E), &(eqpk -> RP), CRP))
return TRUE;
      memset (&PRP, 0, sizeof (PRP));
      memset (&NRP, 0, sizeof (NRP));
      PRP.x = CRP -> x + (eqpi -> E.x - CRP -> x) * eip -> eps *
((double) eqpk -> S);
      PRP.y = CRP -> y + (eqpi -> E.y - CRP -> y) * eip -> eps *
((double) eqpk -> S);
      project_point(&(eqpk -> E), &(eqpk -> DC), &PRP, &NRP);
      if (right_turn (&(eqpk -> E), &(eqpk -> RP), &NRP))
return TRUE;
      /* testing */
      if (left_turn(&(eqpk -> E), &(eqpk -> LP), &NRP))
return FALSE;
      eqpk -> RP = NRP;
      return TRUE;
}
static      void set_member_arr ( struct einfo * eip,struct eqp_t *
eqp,bool flag)
{
      eterm_t *p    = eqp -> Z;
      eterm_t *endp = p + eqp -> S;
      while (p < endp) {
            int t = *p++;
            eip -> MEMB [t] = flag;
      }
}
static      eterm_t * merge_terminal_lists (struct einfo * eip,struct
eqp_t * eqpi,
struct eqp_t * eqpj,struct eqp_t * eqpk)
{
      eterm_t *p1, *endp1, *p2, *endp2, *Zp, *eqpZ_old;
      int t1, t2;
      struct eqp_t *eqpt;
      eqpk -> Z = eip -> eqpZ_curr;
      if (eip -> eqpZ_curr + eqpi -> S + eqpj -> S
            >  eip -> eqpZ + eip -> eqpZ_size)  {
            if (Print_Detailed_Timings)
                  fprintf(stderr, "- doubling terminal list
array\n");
            eqpZ_old = eip -> eqpZ;
            eip -> eqpZ = NEWA ( eip -> eqpZ_size * 2, eterm_t );
            memcpy ( eip -> eqpZ, eqpZ_old, eip -> eqpZ_size *
sizeof(eterm_t) );
            eip -> eqpZ_size = 2 * eip -> eqpZ_size;
            eip -> eqpZ_curr = UPDATE_PTR( eip -> eqpZ_curr,
eqpZ_old, eip -> eqpZ );
            for (eqpt = eip -> eqp; eqpt <= eqpk; eqpt++)
                  eqpt -> Z = UPDATE_PTR( eqpt -> Z, eqpZ_old, eip ->
eqpZ );
            free( eqpZ_old );
```

```
	}
	p1 = eqpi -> Z;
	p2 = eqpj -> Z;
	endp1 = p1 + eqpi -> S;
	endp2 = p2 + eqpj -> S;
	t1 = *p1++;
	t2 = *p2++;
	Zp = eip -> eqpZ_curr;
	for (;;) {
		if (t1 < t2) {
			*Zp++ = t1;
			if (p1 >= endp1) goto finish2;
			t1 = *p1++;
		}
		else {
			*Zp++ = t2;
			if (p2 >= endp2) goto finish1;
			t2 = *p2++;
		}
	}
finish1:
	for (;;) {
		*Zp++ = t1;
		if (p1 >= endp1) break;
		t1 = *p1++;
	}
	return (Zp);
finish2:
	for (;;) {
		*Zp++ = t2;
		if (p2 >= endp2) break;
		t2 = *p2++;
	}
	return (Zp);
}
static	bool disjoint ( struct einfo * eip,struct eqp_t * eqp)
{
	eterm_t *p = eqp -> Z;
	eterm_t *endp = p + eqp -> S;
	while (p < endp) {
		int t = *p++;
		if (eip -> MEMB [t]) return FALSE;
	}
	return TRUE;
}
	static	int highest_terminal (struct eqp_t * eqp)
{
	return ( (eqp -> Z) [eqp -> S  -  1]);
}
static	dist_t closest_terminal (struct einfo * eip,struct point
* Q,struct eqp_t * eqpk)
{
	eterm_t *p, *endp;
	int t;
	dist_t min_dist2, dist2;
	int min_t;
	p = eqpk -> Z;
	endp = p + eqpk -> S;
	min_t = *p++;
	min_dist2 = sqr_dist(Q, &(eip -> eqp[min_t].E));
	while (p < endp) {
		t = *p++;
```

```
            dist2 = sqr_dist(Q, &(eip -> eqp[t].E));
            if (dist2 < min_dist2) {
                  min_dist2 = dist2;
                  min_t = t;
            }
      }
      return (sqrt (min_dist2));
}
/*
 * Get bottleneck Steiner distance between equilateral points i and j
 */
static      double getBSD (struct einfo * eip,struct eqp_t *
eqpi,struct eqp_t * eqpj)
{
      dist_t rbsd, lbsd;
      if (eqpi -> R EQ NULL) {
            if (eqpj -> R EQ NULL) return bsd(eip -> bsd, eqpi ->
E.pnum, eqpj -> E.pnum);
            rbsd = getBSD(eip, eqpi, eqpj -> R);
            lbsd = getBSD(eip, eqpi, eqpj -> L);
      }
      else {
            rbsd = getBSD(eip, eqpi -> R, eqpj);
            lbsd = getBSD(eip, eqpi -> L, eqpj);
      }
      if (rbsd < lbsd) return rbsd;
      return lbsd;
}
static      void eq_point_disp_vector ( struct einfo * eip,struct
eqp_t * eqpi,struct eqp_t * eqpj,
struct eqp_t * eqpk)
{
pnum_t            ti, tj;
double            dx, dy, rdx, rdy;
      ti = eqpi -> DV.pnum;
      tj = eqpj -> DV.pnum;
      dx = eip -> eqp [tj].E.x - eip -> eqp [ti].E.x;
      dy = eip -> eqp [tj].E.y - eip -> eqp [ti].E.y;
      dx -= eqpi -> DV.x;
      dy -= eqpi -> DV.y;
      dx += eqpj -> DV.x;
      dy += eqpj -> DV.y;
      rotate60(dx, dy, &rdx, &rdy);
      eqpk -> DV.x      = rdx + eqpi -> DV.x;
      eqpk -> DV.y      = rdy + eqpi -> DV.y;
      eqpk -> DV.pnum = eqpi -> DV.pnum;
}
/*
 * Computation of distance between eq-points.
 */
static      dist_t eq_point_dist (struct einfo * eip,struct eqp_t *
eqpi,struct eqp_t * eqpj)
{
pnum_t            ti, tj;
double            dx, dy, rdx, rdy;
      ti = eqpi -> DV.pnum;
      tj = eqpj -> DV.pnum;
      dx = eip -> eqp [tj].E.x - eip -> eqp [ti].E.x;
      dy = eip -> eqp [tj].E.y - eip -> eqp [ti].E.y;
      dx -= eqpi -> DV.x;
      dy -= eqpi -> DV.y;
      dx += eqpj -> DV.x;
```

```
	dy += eqpj -> DV.y;
	return hypot (dx, dy);
}
static	void eqpoint_terminals (struct einfo * eip,struct eqp_t *
eqpk,struct pset  * pts,bool append)
{
	eterm_t *p = eqpk -> Z;
	eterm_t *endp = p + eqpk -> S;
	struct point *pt;
	if (NOT append) pts -> n = 0;
	pt = pts -> a + pts -> n;
	while (p < endp)
		*(pt++) = eip -> eqp[ *p++ ].E;
	pts -> n += eqpk -> S;
}
/*
 * Initialize data structure for eq-point rectangles.
 */
static	void initialize_eqp_rectangles ( struct einfo * eip)
{
	int i;
	struct point * p;
	/* Allocating square data structure */
	eip -> eqp_squares = NEWA(eip -> pts -> n, struct eqp_square_t
*);
	/* Finding range of terminal coordinates */
	eip -> minx =  INF_DISTANCE;
	eip -> maxx = -INF_DISTANCE;
	eip -> miny =  INF_DISTANCE;
	eip -> maxy = -INF_DISTANCE;
	for (i = 0; i < eip -> pts -> n; i++) {
		p = &(eip -> pts -> a[i]);
		eip -> minx = MIN( eip -> minx, p -> x - eip -> mean.x);
		eip -> maxx = MAX( eip -> maxx, p -> x - eip -> mean.x);
		eip -> miny = MIN( eip -> miny, p -> y - eip -> mean.y);
		eip -> maxy = MAX( eip -> maxy, p -> y - eip -> mean.y);
		eip -> eqp_squares[i] = NULL;
	}
	/* Basic paramaters */
	eip -> eqp_square_size = eip -> max_mst_edge;
	eip -> srangex = floor( (eip -> maxx - eip -> minx) / eip ->
eqp_square_size) + 1;
	eip -> srangey = floor( (eip -> maxy - eip -> miny) / eip ->
eqp_square_size) + 1;
}
static	void destroy_eqp_rectangles (struct einfo * eip)
{
	int i;
	for (i = 0; i < eip -> pts -> n; i++)
	if (eip -> eqp_squares[i] NE NULL) {
		free( eip -> eqp_squares[i][0].eqp );
		free( eip -> eqp_squares[i]	);
	}
	free( eip -> eqp_squares );
}
/*
 * Compute boundary of rectangle in which the Steiner point joining
 * the current eq-point to another eq-point can be placed
 */
static	void compute_eqp_rectangle (
struct einfo * eip,  /* global  info */
struct eqp_t * eqpk, /* eq-point */
```

```
dist_t * minx,          /* minimum x-coordinate of boundary */
dist_t * maxx,          /* maximum x-coordinate of boundary */
dist_t * miny,          /* minimum y-coordinate of boundary */
dist_t * maxy)          /* maximum x-coordinate of boundary */

{
      struct point p;
      dist_t alpha;
      if (eqpk -> S EQ 1) {
            /* terminal */
            *minx = eqpk -> E.x - eip -> max_mst_edge;
            *maxx = eqpk -> E.x + eip -> max_mst_edge;
            *miny = eqpk -> E.y - eip -> max_mst_edge;
            *maxy = eqpk -> E.y + eip -> max_mst_edge;
      }
      else {
            /*  eq-point */
            *minx = eqpk -> LP.x; *maxx = eqpk -> LP.x;
            *miny = eqpk -> LP.y; *maxy = eqpk -> LP.y;
            UPDATE_RECTANGLE_BOUNDS(eqpk -> RP);
            alpha = 1.0 + eip -> max_mst_edge / EDIST(&(eqpk -> E),
&(eqpk -> LP));
            p.x   = eqpk -> E.x + alpha * (eqpk -> LP.x - eqpk ->
E.x);
            p.y   = eqpk -> E.y + alpha * (eqpk -> LP.y - eqpk ->
E.y);
            UPDATE_RECTANGLE_BOUNDS(p);
            alpha = 1.0 + eip -> max_mst_edge / EDIST(&(eqpk -> E),
&(eqpk -> RP));
            p.x   = eqpk -> E.x + alpha * (eqpk -> RP.x - eqpk ->
E.x);
            p.y   = eqpk -> E.y + alpha * (eqpk -> RP.y - eqpk ->
E.y);
            UPDATE_RECTANGLE_BOUNDS(p);
            p.x = eqpk -> DC.x + eqpk -> DR + eip -> max_mst_edge;
            p.y = eqpk -> DC.y;
            if (right_turn(&(eqpk -> E), &(eqpk -> LP), &p) AND
                left_turn(&(eqpk -> E), &(eqpk -> RP), &p))
                  UPDATE_RECTANGLE_BOUNDS(p);
            p.x = eqpk -> DC.x - eqpk -> DR - eip -> max_mst_edge;
            p.y = eqpk -> DC.y;
            if (right_turn(&(eqpk -> E), &(eqpk -> LP), &p) AND
                left_turn(&(eqpk -> E), &(eqpk -> RP), &p))
                  UPDATE_RECTANGLE_BOUNDS(p);
            p.x = eqpk -> DC.x;
            p.y = eqpk -> DC.y + eqpk -> DR + eip -> max_mst_edge;
            if (right_turn(&(eqpk -> E), &(eqpk -> LP), &p) AND
                left_turn(&(eqpk -> E), &(eqpk -> RP),
&p))UPDATE_RECTANGLE_BOUNDS(p);
            p.x = eqpk -> DC.x;
            p.y = eqpk -> DC.y - eqpk -> DR - eip -> max_mst_edge;
            if (right_turn(&(eqpk -> E), &(eqpk -> LP), &p) AND
                left_turn(&(eqpk -> E), &(eqpk -> RP), &p))
                  UPDATE_RECTANGLE_BOUNDS(p);
      }
      *minx -= (fabs(*minx) * eip -> eps * ((double) eqpk -> S));
      *miny -= (fabs(*miny) * eip -> eps * ((double) eqpk -> S));
      *maxx += (fabs(*maxx) * eip -> eps * ((double) eqpk -> S));
      *maxy += (fabs(*maxy) * eip -> eps * ((double) eqpk -> S));
}
static      void save_eqp_rectangles (struct einfo * eip,int
first_eqp,int last_eqp)
```

```
{
      int i, j, k, si, size;
      struct eqp_t * eqpk;
      dist_t minx, maxx, miny, maxy;
      int total_squares;
      int * curr_count;
      struct eqp_square_t* sqr;
      struct eqp_t ** eqp_alloc;
      if (first_eqp > last_eqp) return;
      /*Finds min/max square indices in each dimension and count the
total
       * number of squares that are overlapped by these eq-points.*/

      size = eip -> eqp[first_eqp].S;
      total_squares = 0;
      for (k = first_eqp; k <= last_eqp; k++) {
            eqpk = &(eip -> eqp[k]);
            /* Compute geometric range of rectangle for this eq-point
*/
            compute_eqp_rectangle(eip, eqpk, &minx, &maxx, &miny,
&maxy);
            /* Compute square index range for this eq-point */
            eqpk -> SMINX = MAX(0,floor( (minx - eip -> minx) / eip -
> eqp_square_size));
            eqpk -> SMAXX = MIN(eip -> srangex-1, floor( (maxx - eip
-> minx) /eip -> eqp_square_size));
            eqpk -> SMINY = MAX(0, floor( (miny - eip -> miny) / eip
-> eqp_square_size));
            eqpk -> SMAXY = MIN(eip -> srangey-1, floor( (maxy - eip
-> miny) / eip -> eqp_square_size));
            total_squares += (eqpk -> SMAXX - eqpk -> SMINX + 1) *
(eqpk -> SMAXY - eqpk -> SMINY + 1);
      }
      /* Allocate space for these rectangles */
      eip -> eqp_squares[size] = NEWA(eip -> srangex * eip ->
srangey, struct eqp_square_t);
      sqr = eip -> eqp_squares[size];
      /* Initialize eq-point counts */
      curr_count = NEWA(eip -> srangex * eip -> srangey, int);
      for (i = 0; i < eip -> srangex; i++)
            for (j = 0; j < eip -> srangey; j++) {
                  si = i * eip -> srangey + j;
                  sqr[si].n = 0; curr_count[si] = 0;
            }
      /* Find counts for each square */
      for (k = first_eqp; k <= last_eqp; k++) {
            eqpk = &(eip -> eqp[k]);
            for (i = eqpk -> SMINX; i <= eqpk -> SMAXX; i++)
                  for (j = eqpk -> SMINY; j <= eqpk -> SMAXY; j++)
                        sqr[ i * eip -> srangey + j ].n++;
      }
      eqp_alloc = NEWA(total_squares, struct eqp_t *);
      for (i = 0; i < eip -> srangex; i++)
            for (j = 0; j < eip -> srangey; j++) {
                  si = i * eip -> srangey + j;
                  sqr[si].eqp = eqp_alloc; eqp_alloc += sqr[si].n;
            }
      for (k = first_eqp; k <= last_eqp; k++) {
            eqpk = &(eip -> eqp[k]);
            for (i = eqpk -> SMINX; i <= eqpk -> SMAXX; i++)
                  for (j = eqpk -> SMINY; j <= eqpk -> SMAXY; j++) {
                        si = i * eip -> srangey + j;
```

```
                        sqr[si].eqp[ curr_count[si]++ ] = eqpk;
                  }
      }
      free(curr_count);
}
/*
 * Generate compatible eq-points
 */
static      void generate_compatible_eqp (struct einfo * eip,int
size,struct eqp_t * eqpi,struct eqp_t ** eqp_list)
{
      struct eqp_square_t * sqr;
      int i, j, l, si;
      struct eqp_t ** eqpp;
      struct eqp_t *    eqpj;
      sqr  = eip -> eqp_squares[size];
      eqpp = eqp_list;
      if (sqr) {
            for (i = eqpi -> SMINX; i <= eqpi -> SMAXX; i++)
                  for (j = eqpi -> SMINY; j <= eqpi -> SMAXY; j++) {
                        si = i * eip -> srangey + j;
                        for (l = 0; l < sqr[si].n; l++) {
                              eqpj = sqr[si].eqp[l];
                              if (NOT (eqpj -> CHOSEN)) {
                                    *(eqpp++) = eqpj;
                                    eqpj -> CHOSEN = TRUE;
                              }
                        }
                  }
      }
      *eqpp = NULL;
}
static      bool projection_test_case_I (struct einfo * eip,struct
eqp_t * eqpi,struct eqp_t * eqpj)
{
      if (eqpi -> R) {
            get_angle_vector(&(eqpi -> RP), &(eqpi -> E), &(eqpj ->
E),
                        &(eip -> dxi), &(eip -> dyi));
            if (angle_geq_120(eip -> dxi, eip -> dyi))
                  return FALSE;
      }
      if (eqpj -> R) {
            get_angle_vector(&(eqpi -> E), &(eqpj -> E), &(eqpj ->
LP),
                        &(eip -> dxj), &(eip -> dyj));
            if (angle_geq_120(eip -> dxj, eip -> dyj))
                  return FALSE;
      }
      return TRUE;
}
static      bool projection_test_cases_II_VI (struct einfo *
eip,struct eqp_t * eqpi,struct eqp_t * eqpj,
struct eqp_t * eqpk)
{
      struct point CLP, CRP;
      if (eqpi -> R) {
            if (angle_geq_60(eip -> dxi, eip -> dyi)) {
                  if (sqr_dist(&(eqpk -> DC), &(eqpi -> LP)) >= eqpk
-> DR2)
                        return FALSE;
```

```
                  project_point(&(eqpi -> E),  &(eqpk -> DC),&(eqpi -
> LP), &(eqpk -> LP));
                  project_point_perp(&(eqpi -> E),  &(eqpk ->
DC),&(eqpi -> DC), &(eqpk -> RP));
            }
            else {
                  if (sqr_dist(&(eqpi -> DC), &(eqpk -> LP)) <= eqpi
-> DR2)
                        return FALSE;
                  if (sqr_dist(&(eqpk -> DC), &(eqpi -> RP)) >= eqpk
-> DR2) {
                        if (sqr_dist(&(eqpk -> DC), &(eqpi -> LP)) >=
eqpk -> DR2)
                              return FALSE;
                        project_point_perp(&(eqpi -> E),  &(eqpk ->
DC),&(eqpi -> DC), &(eqpk -> RP));
                  }
                  else
                        project_point(&(eqpi -> E),  &(eqpk -> DC),
&(eqpi -> RP), &(eqpk -> RP));
                  if (right_turn(&(eqpi -> E), &(eqpk -> LP), &(eqpi
-> LP)))
                        project_point(&(eqpi -> E),  &(eqpk ->
DC),&(eqpi -> LP), &(eqpk -> LP));
            }
      }
      if (eqpj -> R) {
            memset (&CLP, 0, sizeof (CLP));
            memset (&CRP, 0, sizeof (CRP));
            if (angle_geq_60(eip -> dxj, eip -> dyj)) {
                  if (sqr_dist(&(eqpk -> DC), &(eqpj -> RP)) >= eqpk
-> DR2)
                        return FALSE;
                  project_point(&(eqpj -> E), &(eqpk -> DC),&(eqpj ->
RP), &CRP);
                  if (right_turn(&(eqpk -> E), &CRP, &(eqpk -> LP)))
                        return FALSE;
                  if (right_turn(&(eqpk -> E), &CRP, &(eqpk -> RP)))
                        eqpk -> RP = CRP;
                  project_point_perp(&(eqpj -> E),  &(eqpk ->
DC),&(eqpj -> DC), &CLP);
                  if (right_turn(&(eqpk -> E), &(eqpk -> RP), &CLP))
                        return FALSE;
                  if (right_turn(&(eqpk -> E), &(eqpk -> LP), &CLP))
                        eqpk -> LP = CLP;
            }
            else {
                  if (sqr_dist(&(eqpj -> DC), &(eqpk -> RP)) <= eqpj
-> DR2)
                        return FALSE;
                  if (sqr_dist(&(eqpk -> DC), &(eqpj -> LP)) >= eqpk
-> DR2)  {
                        if (sqr_dist(&(eqpk -> DC), &(eqpj -> RP)) >=
eqpk -> DR2)
                              return FALSE;
                        project_point_perp(&(eqpj -> E),  &(eqpk ->
DC), &(eqpj -> DC), &CLP);
                  }
                  else
                        project_point(&(eqpj -> E),  &(eqpk ->
DC),&(eqpj -> LP), &CLP);
                  if (right_turn(&(eqpk -> E), &(eqpk -> RP), &CLP))
```

```
                        return FALSE;
                  if (right_turn(&(eqpk -> E), &(eqpk -> LP), &CLP))
                        eqpk -> LP = CLP;
                  if (right_turn(&(eqpj -> E), &(eqpj -> RP), &(eqpk
-> RP))) {
                        project_point(&(eqpj ->E),   &(eqpk -> DC),
                                    &(eqpj -> RP), &CRP);
                        if (right_turn(&(eqpk -> E), &CRP, &(eqpk ->
LP)))
                              return FALSE;
                        if (right_turn(&(eqpk -> E), &CRP, &(eqpk ->
RP)))
                              eqpk -> RP = CRP;
                  }
            }
      }
      return TRUE;
}
static      bool bsd_test (struct einfo * eip,struct eqp_t *
eqpi,struct eqp_t * eqpj,struct eqp_t * eqpk)
{
      double bsbs, a, b, aa, sin1, sin2;
      struct point CLP, CRP, CLP1, CRP1, CLP2, CRP2, ai;
      eqpk -> BS = getBSD(eip, eqpi, eqpj);
      eqpk -> UB = eqpi -> UB + eqpj -> UB + eqpk -> BS;
      memset (&CLP, 0, sizeof (CLP));
      if (eqpi -> R EQ NULL) {
            rotate2(&(eqpi -> E), &(eqpk -> DC),
                  eqpk -> BS/(2.0*eqpk -> DR), 2.0, &CLP);
            if (NOT test_and_save_LP(eip, eqpi, eqpj, eqpk, &CLP))
return FALSE;
      }
      else {
            bsbs = eqpk -> BS * eqpk -> BS;
            project_point(&(eqpi -> E),  &(eqpi -> DC),&(eqpk -> LP),
&ai);
            aa   = sqr_dist(&ai, &(eqpk -> LP));
            if (bsbs < 0.999 * aa) {
                  a    = sqrt(aa);
                  b    = (a + 2.0 * (  EDIST(&(eqpj -> E), &(eqpk ->
LP))
                        + EDIST(&(eqpi -> R -> E), &ai))) / SQRT3;
                  if (solve_quadratic(aa + b*b, eqpk -> BS * b, bsbs
- aa, &sin1, &sin2)) {
                        memset (&CLP1, 0, sizeof (CLP1));
                        memset (&CLP2, 0, sizeof (CLP2));
                        rotate2(&(eqpk -> LP), &(eqpk -> DC),-sin1,
(b * sin1 + eqpk -> BS)/a, &CLP1);
                        rotate2(&(eqpk -> LP), &(eqpk -> DC),-sin2,
(b * sin2 + eqpk -> BS)/a, &CLP2);
                        if (right_turn(&(eqpk -> E), &CLP2, &CLP1))
                              CLP1 = CLP2;
                        if (NOT test_and_save_LP(eip, eqpi, eqpj,
eqpk, &CLP1)) return FALSE;
                  }
            }
      }
      memset (&CRP, 0, sizeof (CRP));
      if (eqpj -> R EQ NULL) {
            rotate2(&(eqpj -> E), &(eqpk -> DC),
                  -eqpk -> BS/(2.0*eqpk -> DR), 2.0, &CRP);
```

```
            if (NOT test_and_save_RP(eip, eqpi, eqpj, eqpk, &CRP))
return FALSE;
      }
      else {
            bsbs = eqpk -> BS * eqpk -> BS;
            project_point(&(eqpj -> E),  &(eqpj -> DC),&(eqpk -> RP),
&ai);
            aa   = sqr_dist(&ai, &(eqpk -> RP));
            if (bsbs < 0.999 * aa) {
                  a    = sqrt(aa);
                  b    = (a + 2.0 * (  EDIST(&(eqpi -> E), &(eqpk ->
RP))
                        + EDIST(&(eqpj -> L -> E), &ai))) / SQRT3;
                  if (solve_quadratic(aa + b*b, eqpk -> BS * b, bsbs
- aa, &sin1, &sin2)) {
                        memset (&CRP1, 0, sizeof (CRP1));
                        memset (&CRP2, 0, sizeof (CRP2));
                        rotate2(&(eqpk -> RP), &(eqpk -> DC),
                              sin1, (b * sin1 + eqpk -> BS)/a,
&CRP1);
                        rotate2(&(eqpk -> RP), &(eqpk -> DC),
                              sin2, (b * sin2 + eqpk -> BS)/a,
&CRP2);
                        if (right_turn(&(eqpk -> E), &CRP1, &CRP2))
                              CRP1 = CRP2;
                        if (NOT test_and_save_RP(eip, eqpi, eqpj,
eqpk, &CRP1)) return FALSE;
                  }
            }
      }
      return TRUE;
}
static      bool lune_test (struct einfo * eip,struct eqp_t *
eqpi,struct eqp_t * eqpj,struct eqp_t * eqpk)
{
      int r;
      struct point CJ, AI, CLP, CLPP, CRP, CRPP;
      dist_t a, aa, b, bb, c, d, e, f, ff, g, gg, h, hh;
      dist_t dist_qicj, dist_oacr, dist_opqr, dist_pqi, dist_piai,
dist_qpi;
      dist_t sin1, sin2;
      memset (&CLPP, 0, sizeof (CLPP));
      memset (&CRPP, 0, sizeof (CRPP));
      if (eqpi -> R) {
            project_point(&(eqpi -> E),  &(eqpi -> DC), &(eqpk ->
LP), &CJ);
            aa      = sqr_dist(&CJ, &(eqpk -> LP));
            dist_qicj = 0.999 * aa;
            for (r = 0; r < eip -> pts -> n; r++) {
                  if (sqr_dist(&(eip -> eqp[r].E), &CJ)           >=
dist_qicj) continue;
                  if (sqr_dist(&(eip -> eqp[r].E), &(eqpk -> LP)) >=
dist_qicj) continue;
                  CLP = eqpi -> E;
                  a   = sqrt(aa);
                  b   = (a + 2.0 * (  EDIST(&(eqpj -> E), &(eqpk ->
LP))
                        + EDIST(&(eqpi -> R -> E), &CJ))) / SQRT3;
                  bb  = b*b;
                  dist_oacr = EDIST(&(eip -> eqp[r].E), &(eqpi ->
DC));
                  c = dist_oacr*dist_oacr + eqpi -> DR2;
```

```
                    d = 2.0*((CJ.x - eqpi -> DC.x) * (eqpi -> DC.x -
eip -> eqp[r].E.x) +
                             (CJ.y - eqpi -> DC.y) * (eqpi -> DC.y - eip
-> eqp[r].E.y));
                    e = 2.0*((CJ.y - eqpi -> DC.y) * (eqpi -> DC.x -
eip -> eqp[r].E.x) -
                             (CJ.x - eqpi -> DC.x) * (eqpi -> DC.y - eip
-> eqp[r].E.y));
                    f = (aa+bb)/2.0-c; ff = f*f;
                    g = -e-a*b;    gg = g*g;
                    h = d-(aa-bb)/2.0; hh = h*h;
                    if (solve_quadratic(gg + hh, f*g, ff - hh, &sin1,
&sin2)) {
                         if ((fabs(sin1) < eip -> eps) OR (fabs(sin2)
< eip -> eps)) continue;
                         rotate(&(eqpk -> LP), &(eqpk -> DC), -sin1,
(f+g*sin1)/h, &CLPP);
                         if (right_turn(&(eqpk -> E), &(eqpk -> LP),
&CLPP) AND
                             left_turn (&(eqpk -> E), &CLP, &CLPP))
                               CLP = CLPP;
                         rotate(&(eqpk -> LP), &(eqpk -> DC), -sin2,
(f+g*sin2)/h, &CLPP);
                         if (right_turn(&(eqpk -> E), &(eqpk -> LP),
&CLPP) AND
                             left_turn (&(eqpk -> E), &CLP, &CLPP))
                               CLP = CLPP;
                    }
                    dist_opqr = EDIST(&(eip -> eqp[r].E), &(eqpk ->
DC));
                    c = dist_opqr*dist_opqr + eqpk -> DR2;
                    d = 2.0*((eqpk -> LP.x - eqpk -> DC.x) * (eqpk ->
DC.x - eip -> eqp[r].E.x) +
                             (eqpk -> LP.y - eqpk -> DC.y) * (eqpk ->
DC.y - eip -> eqp[r].E.y));
                    e = 2.0*((eqpk -> LP.y - eqpk -> DC.y) * (eqpk ->
DC.x - eip -> eqp[r].E.x) -
                             (eqpk -> LP.x - eqpk -> DC.x) * (eqpk ->
DC.y - eip -> eqp[r].E.y));
                    f = (aa+bb)/2.0-c; ff = f*f;
                    g = -e-a*b;    gg = g*g;
                    h = d-(aa-bb)/2.0; hh = h*h;
                    if (solve_quadratic(gg + hh, f*g, ff - hh, &sin1,
&sin2)) {
                         if ((fabs(sin1) < eip -> eps) OR (fabs(sin2)
< eip -> eps)) continue;
                         rotate(&(eqpk -> LP), &(eqpk -> DC), -sin1,
(f+g*sin1)/h, &CLPP);
                         if (right_turn(&(eqpk -> E), &(eqpk -> LP),
&CLPP) AND
                             left_turn (&(eqpk -> E), &CLP, &CLPP))
                               CLP = CLPP;
                         rotate(&(eqpk -> LP), &(eqpk -> DC), -sin2,
(f+g*sin2)/h, &CLPP);
                         if (right_turn(&(eqpk -> E), &(eqpk -> LP),
&CLPP) AND
                             left_turn (&(eqpk -> E), &CLP, &CLPP))
                               CLP = CLPP;
                    }
                    if (NOT test_and_save_LP(eip, eqpi, eqpj, eqpk,
&CLP)) return FALSE;
```

```
                  project_point(&(eqpi -> E),  &(eqpi -> DC),&(eqpk -
> LP), &CJ);
                  aa      = sqr_dist(&CJ, &(eqpk -> LP));
                  dist_qicj = 0.999 * aa;
            }
      }
      else {
            aa     = sqr_dist(&(eqpi -> E), &(eqpk -> LP));
            dist_pqi = 0.999 * aa;
            for (r = 0; r < eip -> pts -> n; r++) {
                  if (r EQ eqpi -> E.pnum) continue;
                  if (sqr_dist(&(eip -> eqp[r].E), &(eqpi -> E))  >=
dist_pqi) continue;
                  if (sqr_dist(&(eip -> eqp[r].E), &(eqpk -> LP)) >=
dist_pqi) continue;
                  rotate2(&(eqpi -> E), &(eqpk -> DC),
                        EDIST(&(eip -> eqp[r].E), &(eqpi ->
E))/(2.0*eqpk -> DR), 2.0, &CLP);
                  a  = sqrt(aa);
                  b  = (a + 2.0 * EDIST(&(eqpj -> E), &(eqpk -> LP)))
/ SQRT3;
                  bb = b*b;
                  dist_opqr = EDIST(&(eip -> eqp[r].E), &(eqpk ->
DC));
                  c = dist_opqr*dist_opqr + eqpk -> DR2;
                  d = 2.0*((eqpk -> LP.x - eqpk -> DC.x) * (eqpk ->
DC.x - eip -> eqp[r].E.x) +
                           (eqpk -> LP.y - eqpk -> DC.y) * (eqpk ->
DC.y - eip -> eqp[r].E.y));
                  e = 2.0*((eqpk -> LP.y - eqpk -> DC.y) * (eqpk ->
DC.x - eip -> eqp[r].E.x) -
                           (eqpk -> LP.x - eqpk -> DC.x) * (eqpk ->
DC.y - eip -> eqp[r].E.y));
                  f = (aa+bb)/2.0-c; ff = f*f;
                  g = -e-a*b;    gg = g*g;
                  h = d-(aa-bb)/2.0; hh = h*h;
                  if (solve_quadratic(gg + hh, f*g, ff - hh, &sin1,
&sin2)) {
                        if ((fabs(sin1) < eip -> eps) OR (fabs(sin2)
< eip -> eps)) continue;
                        rotate(&(eqpk -> LP), &(eqpk -> DC), -sin1,
(f+g*sin1)/h, &CLPP);
                        if (right_turn(&(eqpk -> E), &(eqpk -> LP),
&CLPP) AND
                            left_turn (&(eqpk -> E), &CLP, &CLPP))
                              CLP = CLPP;
                        rotate(&(eqpk -> LP), &(eqpk -> DC), -sin2,
(f+g*sin2)/h, &CLPP);
                        if (right_turn(&(eqpk -> E), &(eqpk -> LP),
&CLPP) AND
                            left_turn (&(eqpk -> E), &CLP, &CLPP))
                              CLP = CLPP;
                  }
                  if (NOT test_and_save_LP(eip, eqpi, eqpj, eqpk,
&CLP)) return FALSE;
                  aa     = sqr_dist(&(eqpi -> E), &(eqpk -> LP));
                  dist_pqi = 0.999 * aa;
            }
      }
      if (eqpj -> R) {
            project_point(&(eqpj -> E),  &(eqpj -> DC),&(eqpk -> RP),
&AI);
```

```
            aa      = sqr_dist(&AI, &(eqpk -> RP));
            dist_piai = 0.999 * aa;
            for (r = 0; r < eip -> pts -> n; r++) {
                  if (sqr_dist(&(eip -> eqp[r].E), &AI)           >=
dist_piai) continue;
                  if (sqr_dist(&(eip -> eqp[r].E), &(eqpk -> RP)) >=
dist_piai) continue;
                  CRP = eqpj -> E;
                  a   = sqrt(aa);
                  b   = (a + 2.0 * (  EDIST(&(eqpi -> E), &(eqpk ->
RP))
                        + EDIST(&(eqpj -> L -> E), &AI))) / SQRT3;
                  bb  = b*b;
                  dist_oacr = EDIST(&(eip -> eqp[r].E), &(eqpj ->
DC));
                  c = dist_oacr*dist_oacr + eqpj -> DR2;
                  d = 2.0*((AI.x - eqpj -> DC.x) * (eqpj -> DC.x -
eip -> eqp[r].E.x) +
                        (AI.y - eqpj -> DC.y) * (eqpj -> DC.y - eip
-> eqp[r].E.y));
                  e = 2.0*((AI.y - eqpj -> DC.y) * (eqpj -> DC.x -
eip -> eqp[r].E.x) -
                        (AI.x - eqpj -> DC.x) * (eqpj -> DC.y - eip
-> eqp[r].E.y));
                  f = (aa+bb)/2.0-c; ff = f*f;
                  g = e-a*b;     gg = g*g;
                  h = d-(aa-bb)/2.0; hh = h*h;
                  if (solve_quadratic(gg + hh, f*g, ff - hh, &sin1,
&sin2)) {
                        if ((fabs(sin1) < eip -> eps) OR (fabs(sin2)
< eip -> eps)) continue;
                        rotate(&(eqpk -> RP), &(eqpk -> DC), sin1,
(f+g*sin1)/h, &CRPP);
                        if (left_turn(&(eqpk -> E), &(eqpk -> RP),
&CRPP) AND
                            right_turn(&(eqpk -> E), &CRP, &CRPP))
                              CRP = CRPP;
                        rotate(&(eqpk -> RP), &(eqpk -> DC), sin2,
(f+g*sin2)/h, &CRPP);
                        if (left_turn(&(eqpk -> E), &(eqpk -> RP),
&CRPP) AND
                            right_turn(&(eqpk -> E), &CRP, &CRPP))
                              CRP = CRPP;
                  }
                  dist_opqr = EDIST(&(eip -> eqp[r].E), &(eqpk ->
DC));
                  c = dist_opqr*dist_opqr + eqpk -> DR2;
                  d = 2.0*((eqpk -> RP.x - eqpk -> DC.x) * (eqpk ->
DC.x - eip -> eqp[r].E.x) +
                        (eqpk -> RP.y - eqpk -> DC.y) * (eqpk ->
DC.y - eip -> eqp[r].E.y));
                  e = 2.0*((eqpk -> RP.y - eqpk -> DC.y) * (eqpk ->
DC.x - eip -> eqp[r].E.x) -
                        (eqpk -> RP.x - eqpk -> DC.x) * (eqpk ->
DC.y - eip -> eqp[r].E.y));
                  f = (aa+bb)/2.0-c; ff = f*f;
                  g = e-a*b;     gg = g*g;
                  h = d-(aa-bb)/2.0; hh = h*h;
                  if (solve_quadratic(gg + hh, f*g, ff - hh, &sin1,
&sin2)) {
                        if ((fabs(sin1) < eip -> eps) OR (fabs(sin2)
< eip -> eps)) continue;
```

```
                        rotate(&(eqpk -> RP), &(eqpk -> DC), sin1,
(f+g*sin1)/h, &CRPP);
                        if (left_turn(&(eqpk -> E), &(eqpk -> RP),
&CRPP) AND
                            right_turn(&(eqpk -> E), &CRP, &CRPP))
                              CRP = CRPP;
                        rotate(&(eqpk -> RP), &(eqpk -> DC), sin2,
(f+g*sin2)/h, &CRPP);
                        if (left_turn(&(eqpk -> E), &(eqpk -> RP),
&CRPP) AND
                            right_turn(&(eqpk -> E), &CRP, &CRPP))
                              CRP = CRPP;
                  }
                  if (NOT test_and_save_RP(eip, eqpi, eqpj, eqpk,
&CRP)) return FALSE;
                  project_point(&(eqpj -> E),  &(eqpj -> DC), &(eqpk
-> RP), &AI);
                  aa      = sqr_dist(&AI, &(eqpk -> RP));
                  dist_piai = 0.999 * aa;
            }
      }
      else {
            aa     = sqr_dist(&(eqpj -> E), &(eqpk -> RP));
            dist_qpi = 0.999 * aa;
            for (r = 0; r < eip -> pts -> n; r++) {
                  if (r EQ eqpj -> E.pnum) continue;
                  if (sqr_dist(&(eip -> eqp[r].E), &(eqpj -> E))  >=
dist_qpi) continue;
                  if (sqr_dist(&(eip -> eqp[r].E), &(eqpk -> RP)) >=
dist_qpi) continue;
                  rotate2(&(eqpj -> E), &(eqpk -> DC),
                        -EDIST(&(eip -> eqp[r].E), &(eqpj ->
E))/(2.0*eqpk -> DR), 2.0, &CRP);
                  a  = sqrt(aa);
                  b  = (a + 2.0 * EDIST(&(eqpi -> E), &(eqpk -> RP)))
/ SQRT3;
                  bb = b*b;
                  dist_opqr = EDIST(&(eip -> eqp[r].E), &(eqpk ->
DC));
                  c = dist_opqr*dist_opqr + eqpk -> DR2;
                  d = 2.0*((eqpk -> RP.x - eqpk -> DC.x) * (eqpk ->
DC.x - eip -> eqp[r].E.x) +
                           (eqpk -> RP.y - eqpk -> DC.y) * (eqpk ->
DC.y - eip -> eqp[r].E.y));
                  e = 2.0*((eqpk -> RP.y - eqpk -> DC.y) * (eqpk ->
DC.x - eip -> eqp[r].E.x) -
                           (eqpk -> RP.x - eqpk -> DC.x) * (eqpk ->
DC.y - eip -> eqp[r].E.y));
                  f = (aa+bb)/2.0-c; ff = f*f;
                  g = e-a*b;     gg = g*g;
                  h = d-(aa-bb)/2.0; hh = h*h;
                  if (solve_quadratic(gg + hh, f*g, ff - hh, &sin1,
&sin2)) {
                        if ((fabs(sin1) < eip -> eps) OR (fabs(sin2)
< eip -> eps)) continue;
                        rotate(&(eqpk -> RP), &(eqpk -> DC), sin1,
(f+g*sin1)/h, &CRPP);
                        if (left_turn(&(eqpk -> E), &(eqpk -> RP),
&CRPP) AND
                            right_turn(&(eqpk -> E), &CRP, &CRPP))
                              CRP = CRPP;
```

```
                        rotate(&(eqpk -> RP), &(eqpk -> DC), sin2,
(f+g*sin2)/h, &CRPP);
                        if (left_turn(&(eqpk -> E), &(eqpk -> RP),
&CRPP) AND
                            right_turn(&(eqpk -> E), &CRP, &CRPP))
                              CRP = CRPP;
                  }
                  if (NOT test_and_save_RP(eip, eqpi, eqpj, eqpk,
&CRP)) return FALSE;
                  aa     = sqr_dist(&(eqpj -> E), &(eqpk -> RP));
                  dist_qpi = 0.999 * aa;
            }
      }
      return TRUE;
}
static      bool upper_bound_test (struct einfo * eip,struct eqp_t *
eqpi,struct eqp_t * eqpj,struct eqp_t * eqpk)
{
      dist_t      low_arc_length, rlb, llb, lower_bound, upper_bound,
dist, distr, distl,     rsmt, lsmt, dx, dy, l,
            ub_ri = INF_DISTANCE, ub_lj = INF_DISTANCE, ub_rk =
INF_DISTANCE, ub_lk = INF_DISTANCE,
            distrj, distli, cosp, sinp, a, aa, b, bb, sin1, sin2;
      struct point P, CRP, CRPP, CLP, CLPP, M;
      if (EDIST(&(eqpk -> RP), &(eqpk -> LP)) < eip -> eps) return
FALSE;
      rlb = EDIST(&(eqpk -> RP), &(eqpk -> E)) * (1.0 - eip -> eps *
((double) eqpk -> S));
      llb = EDIST(&(eqpk -> LP), &(eqpk -> E)) * (1.0 - eip -> eps *
((double) eqpk -> S));
      lower_bound = MIN(rlb, llb);
      M.x = (eqpk -> RP.x + eqpk -> LP.x) / 2.0;
      M.y = (eqpk -> RP.y + eqpk -> LP.y) / 2.0;
      P.x = eqpk -> DC.x + (eqpk -> DR / EDIST(&(eqpk -> DC), &M))*
(M.x - eqpk -> DC.x);
      P.y = eqpk -> DC.y + (eqpk -> DR / EDIST(&(eqpk -> DC), &M))*
(M.y - eqpk -> DC.y);
      low_arc_length = 2.0*EDIST(&P, &(eqpk -> RP));
      distr = closest_terminal(eip, &(eqpk -> RP), eqpi);
      distl = closest_terminal(eip, &(eqpk -> LP), eqpj);
      upper_bound = (eqpi -> UB + distr) + (eqpj -> UB + distl) +
low_arc_length;
      if (upper_bound    < lower_bound) return FALSE;
      rsmt = upper_bound;
      lsmt = upper_bound;
      dist = distl;
      if (distr < dist) dist = distr;
      upper_bound = eqpi -> UB + eqpj -> UB + dist + EDIST(&(eqpk ->
RP), &(eqpk -> LP)) + eqpk -> BS;
      if (upper_bound    < lower_bound) return FALSE;
      if (rsmt > upper_bound) rsmt = upper_bound;
      if (lsmt > upper_bound) lsmt = upper_bound;
      if (eqpi -> R EQ NULL)
            ub_ri = EDIST(&(eqpk -> RP), &(eqpi -> E));
      else {
            eip -> termlist -> a[0] = eqpk -> RP;
            eip -> termlist -> n    = 1;
            eqpoint_terminals(eip, eqpi, eip -> termlist, TRUE);
            ub_ri = upper_bound_heuristic(eip -> termlist, eip ->
bsd);
      }
      upper_bound = ub_ri + (eqpj -> UB + distl) + low_arc_length;
```

```
      if (upper_bound < lower_bound) return FALSE;
      if (rsmt > upper_bound) rsmt = upper_bound;
      if (lsmt > upper_bound) lsmt = upper_bound;
      if (eqpj -> R EQ NULL)
            ub_lj = EDIST(&(eqpk -> LP), &(eqpj -> E));
      else {
            eip -> termlist -> a[0] = eqpk -> LP;
            eip -> termlist -> n    = 1;
            eqpoint_terminals(eip, eqpj, eip -> termlist, TRUE);
            ub_lj = upper_bound_heuristic(eip -> termlist, eip ->
bsd);
      }
      upper_bound = eqpi -> UB + distr;
      if (ub_ri < upper_bound) upper_bound = ub_ri;
      upper_bound += ub_lj + low_arc_length;
      if (upper_bound < lower_bound) return FALSE;
      if (rsmt > upper_bound) rsmt = upper_bound;
      if (lsmt > upper_bound) lsmt = upper_bound;
      upper_bound = 0.0;
      eip -> termlist -> a[0] = eqpk -> RP;
      eip -> termlist -> a[1] = eqpk -> LP;
      eip -> termlist -> n    = 2;
      if (eqpi -> R EQ NULL)
            upper_bound += distr;
      else
            eqpoint_terminals(eip, eqpi, eip -> termlist, TRUE);
      if (eqpj -> R EQ NULL)
            upper_bound += distl;
      else
            eqpoint_terminals(eip, eqpj, eip -> termlist, TRUE);
      if (eip -> termlist -> n EQ 2)
            upper_bound += eqpk -> BS + low_arc_length/2.0;
      else
            upper_bound += upper_bound_heuristic(eip -> termlist, eip
-> bsd) + low_arc_length/2.0;
      if (upper_bound < lower_bound) return FALSE;
      if (rsmt > upper_bound) rsmt = upper_bound;
      if (lsmt > upper_bound) lsmt = upper_bound;
      if (eqpi -> R) {
            eip -> termlist -> a[0] = eqpk -> RP;
            eip -> termlist -> n    = 1;
            eqpoint_terminals(eip, eqpk, eip -> termlist, TRUE);
            ub_rk = upper_bound_heuristic(eip -> termlist, eip ->
bsd);
            upper_bound = ub_rk + low_arc_length;
            if (upper_bound < lower_bound) return FALSE;
            if (rsmt > upper_bound) rsmt = upper_bound;
      }
      if (eqpj -> R) {
            eip -> termlist -> a[0] = eqpk -> LP;
            eip -> termlist -> n    = 1;
            eqpoint_terminals(eip, eqpk, eip -> termlist, TRUE);
            ub_lk = upper_bound_heuristic(eip -> termlist, eip ->
bsd);
            upper_bound = ub_lk + low_arc_length;
            if (upper_bound < lower_bound) return FALSE;
            if (lsmt > upper_bound) lsmt = upper_bound;
      }
      upper_bound = eqpj -> UB + low_arc_length + eqpk -> BS;
      if (eqpi -> R EQ NULL)
            upper_bound += EDIST(&(eqpk -> RP), &(eqpi -> E));
      else
```

```
            upper_bound += ub_ri;
      if (upper_bound < lower_bound) return FALSE;
      if (rsmt > upper_bound) rsmt = upper_bound;
      upper_bound = eqpi -> UB + low_arc_length + eqpk -> BS;
      if (eqpj -> R EQ NULL)
            upper_bound += EDIST(&(eqpk -> LP), &(eqpj -> E));
      else
            upper_bound += ub_lj;
      if (upper_bound < lower_bound) return FALSE;
      if (lsmt > upper_bound) lsmt = upper_bound;
      distrj = closest_terminal(eip, &(eqpk -> RP), eqpj);
      upper_bound = eqpi -> UB + eqpj -> UB + distr + distrj;
      if (rsmt > upper_bound) rsmt = upper_bound;
      distli = closest_terminal(eip, &(eqpk -> LP), eqpi);
      upper_bound = eqpi -> UB + eqpj -> UB + distl + distli;
      if (lsmt > upper_bound) lsmt = upper_bound;
      if (eqpi -> R NE NULL)
            upper_bound = ub_rk;
      else {
            if (eqpj -> R EQ NULL)
                  upper_bound = EDIST(&(eqpj -> E), &(eqpk -> RP)) +
EDIST(&(eqpi -> E), &(eqpk -> RP));
            else {
                  eip -> termlist -> a[0] = eqpk -> RP;
                  eip -> termlist -> n    = 1;
                  eqpoint_terminals(eip, eqpj, eip -> termlist,
TRUE);
                  upper_bound = upper_bound_heuristic(eip ->
termlist, eip -> bsd) +
                                    EDIST(&(eqpi -> E), &(eqpk -> RP));
            }
      }
      if (upper_bound < rsmt) rsmt = upper_bound;
      if (rsmt < 0.999 * rlb) {
            CRP = eqpj -> E;
            memset (&CRPP, 0, sizeof (CRPP));
            get_angle_vector(&(eqpi -> E), &(eqpk -> E), &(eqpk ->
RP), &dx, &dy);
            l = sqrt(dx*dx + dy*dy);
            cosp = dx/l; sinp = dy/l;
            a = eqpk -> DR*(SQRT3*cosp+sinp);     aa = a*a;
            b = eqpk -> DR*(SQRT3*sinp-cosp+2.0); bb = b*b;
            if (solve_quadratic(aa + bb, rsmt*b, rsmt*rsmt - aa,
&sin1, &sin2)) {
                  if ((fabs(sin1) < eip -> eps) OR (fabs(sin2) < eip
-> eps)) return TRUE;
                  rotate(&(eqpk -> RP), &(eqpk -> DC), sin1,
(b*sin1+rsmt)/a, &CRPP);
                  if (left_turn(&(eqpk -> E), &(eqpk -> RP), &CRPP)
AND
                      right_turn(&(eqpk -> E), &CRP, &CRPP))
                        CRP = CRPP;
                  rotate(&(eqpk -> RP), &(eqpk -> DC), sin2,
(b*sin2+rsmt)/a, &CRPP);
                  if (left_turn(&(eqpk -> E), &(eqpk -> RP), &CRPP)
AND
                      right_turn(&(eqpk -> E), &CRP, &CRPP))
                        CRP = CRPP;
                  if (NOT test_and_save_RP(eip, eqpi, eqpj, eqpk,
&CRP)) return FALSE;
            }
      }
```

```
      if (eqpj -> R)
            upper_bound = ub_lk;
      else {
            if (eqpi -> R EQ NULL)
                  upper_bound = EDIST(&(eqpi -> E), &(eqpk -> LP)) +
                              EDIST(&(eqpj -> E), &(eqpk -> LP));
            else {
                  eip -> termlist -> a[0] = eqpk -> LP;
                  eip -> termlist -> n    = 1;
                  eqpoint_terminals(eip, eqpi, eip -> termlist,
TRUE);
                  upper_bound = upper_bound_heuristic(eip ->
termlist, eip -> bsd) +
                              EDIST(&(eqpj -> E), &(eqpk -> LP));
            }
      }
      if (upper_bound < lsmt) lsmt = upper_bound;
      if (lsmt < 0.999 * llb) {
            CLP = eqpi -> E;
            memset (&CLPP, 0, sizeof (CLPP));
            get_angle_vector(&(eqpk -> LP), &(eqpk -> E), &(eqpj ->
E), &dx, &dy);
            l = sqrt(dx*dx + dy*dy);
            cosp = dx/l; sinp = dy/l;
            a = eqpk -> DR*(SQRT3*cosp+sinp);     aa=a*a;
            b = eqpk -> DR*(SQRT3*sinp-cosp+2.0); bb=b*b;
            if (solve_quadratic(aa + bb, lsmt*b, lsmt*lsmt - aa,
&sin1, &sin2)) {
                  if ((fabs(sin1) < eip -> eps) OR (fabs(sin2) < eip
-> eps)) return TRUE;
                  rotate(&(eqpk -> LP), &(eqpk -> DC), -sin1,
(b*sin1+lsmt)/a, &CLPP);
                  if (right_turn(&(eqpk -> E), &(eqpk -> LP), &CLPP)
AND
                      left_turn(&(eqpk -> E), &CLP, &CLPP))
                        CLP = CLPP;
                  rotate(&(eqpk -> LP), &(eqpk -> DC), -sin2,
(b*sin2+lsmt)/a, &CLPP);
                  if (right_turn(&(eqpk -> E), &(eqpk -> LP), &CLPP)
AND
                      left_turn(&(eqpk -> E), &CLP, &CLPP))
                        CLP = CLPP;
                  if (NOT test_and_save_LP(eip, eqpi, eqpj, eqpk,
&CLP)) return FALSE;
            }
      }
      return TRUE;
}
static      bool wedge_test (struct einfo * eip,struct eqp_t *
eqpi,struct eqp_t * eqpj,struct eqp_t * eqpk)
{
      int r, t;
      int right_counter = 0;
      int middle_counter = 0;
      int left_counter = 0;
      int top = highest_terminal(eqpk);
      bool flag;
      dist_t dist, dist1, dist2, bsdi, bsdj;
      struct point SP;
      struct eqp_t * eqpt;
      struct eqp_t * other_eqp;
      other_eqp = eqpj;
```

```
      if (NOT disjoint(eip, other_eqp)) other_eqp = eqpi;
      set_member_arr(eip, other_eqp, TRUE);
#ifdef HAVE_GMP
      if (Multiple_Precision > 0) {
            update_eqpoint_and_displacement (eip, eqpk);
      }
#endif
      for (t = 0; t < eip -> pts -> n; t++) {
            if (eip -> MEMB[t]) continue;
            eqpt = &(eip -> eqp[t]);
            if ((left_turn(&(eqpi -> E), &(eqpk -> LP), &(eqpt ->
E))) OR
                (right_turn(&(eqpj -> E), &(eqpk -> RP), &(eqpt -
>E)))) continue;
            if (left_turn(&(eqpk -> E), &(eqpk -> LP), &(eqpt -> E)))
{
                  left_counter++; continue;
            }
            if (right_turn(&(eqpk -> E), &(eqpk -> RP), &(eqpt ->
E))) {
                  right_counter++; continue;
            }
            if (sqr_dist(&(eqpk -> DC), &(eqpt -> E)) > eqpk -> DR2)
{
                  middle_counter++;
                  if (t <= top) continue;
                  project_point(&(eqpk -> E), &(eqpk -> DC),&(eqpt ->
E), &SP);
                  dist = EDIST(&(eqpt -> E), &SP) * (1.0 - eip -> eps
* ((double) eqpk -> S));
                  if (dist >= getBSD(eip, eqpt, eqpk)) continue;
                  flag = TRUE;
                  for (r = 0; r < eip -> pts -> n; r++) {
                        if (r EQ t) continue;
                        if ((EDIST(&SP, &(eip -> eqp[r].E))      <
dist) AND
                            (EDIST(&(eqpt -> E), &(eip -> eqp[r].E))
< dist)) {
                              flag = FALSE; break;
                        }
                  }
                  if (NOT flag) continue;
                  eip -> termlist -> a[0] = eqpt -> E;
                  eip -> termlist -> n    = 1;
                  eqpoint_terminals(eip, eqpk, eip -> termlist,
TRUE);
                  if (eqpk -> S > 2) {
                        dist1 = upper_bound_heuristic(eip ->
termlist, eip -> bsd);
                        dist2 = eq_point_dist(eip, eqpt, eqpk) *
                              (1.0 - eip -> eps * ((double) eqpk ->
S));
                        if (dist1 < dist2) {
                              flag = FALSE;
                        }
                        else {
                              bsdi = getBSD(eip, eqpi, eqpt);
                              bsdj = getBSD(eip, eqpj, eqpt);
                              if ((bsdi + eqpk -> UB < dist2) OR
                                  (bsdj + eqpk -> UB < dist2) OR
                                  (bsdi + bsdj + eqpi -> UB + eqpj ->
UB < dist2))
```

```
                                    flag = FALSE;
                              }
                        }
                        if (NOT flag) continue;
                        test_and_save_fst(eip, eqpt, eqpk);
                  }
            }
            set_member_arr(eip, other_eqp, FALSE);
            flag = FALSE;
            if (middle_counter >= 1) {
                  flag = TRUE;
                  if ((middle_counter EQ 1) AND (left_counter +
right_counter EQ 0)) return FALSE;
            }
            else {
                  if ((left_counter >= 1) AND (right_counter >= 1)) flag =
TRUE;
            }
            return(flag);
}
static      void compute_efsts_for_unique_terminals (struct einfo *
            eip)
{
int               n;
int               nedges;
struct pset *           pts;
struct edge *           ep;
struct edge *           mst_edges;
dist_t                  mst_len;
char              buf1 [32];
eterm_t                 *new_Zp;
int               i, j, k, si, l, size, iter, sz, starti, endi;
dist_t                  upper_bound;
struct eqp_t            *eqpi, *eqpj, *eqpk, *eqpt, *eqp_old;
struct eqp_t            **eqp_list, **eqpp, **eqppp;
struct elist            *rp;
            pts = eip -> pts;
            n = pts -> n;
            mst_edges = NEWA (n - 1, struct edge);
            nedges = euclidean_mst (pts, mst_edges);
            if (nedges NE n - 1) {
                  fatal ("compute_points_for_unique_terminals: Bug 1.");
            }
            mst_len = 0.0;
            eip -> max_mst_edge = 0.0;
            ep = mst_edges;
            for (i = 0; i < nedges; ep++, i++) {
                  mst_len += ep -> len;
                  if (eip -> max_mst_edge < ep -> len) eip -> max_mst_edge
= ep -> len;
            }
            eip -> mst_length = mst_len;
            if (Print_Detailed_Timings) {
                  convert_delta_cpu_time (buf1);
                  fprintf (stderr, "Compute MST:            %s\n", buf1);
            }
            eip -> eps   = ((double) EpsilonFactor) * DBL_EPSILON;

            /* Compute bottleneck Steiner distances */
            eip -> bsd = compute_bsd (nedges, mst_edges, 0);
            free ((char *) mst_edges);
            if (Print_Detailed_Timings) {
```

```
            convert_delta_cpu_time (buf1);
            fprintf (stderr, "Compute BSD:            %s\n", buf1);
      }
      eip -> term_check = NEWA (n, bool);
      eip -> hash       = NEWA (n, struct elist *);
      for (i = 0; i < n; i++) {
        eip -> term_check [i] = FALSE;
        eip -> hash [i] = NULL;
      }
      eip -> list.forw  = &(eip -> list);
      eip -> list.back  = &(eip -> list);
      eip -> mean.x = 0.0;
      eip -> mean.y = 0.0;
      for (k = 0; k < n; k++) {
            eip -> mean.x += pts -> a[k].x;
            eip -> mean.y += pts -> a[k].y;
      }
      eip -> mean.x = floor( eip -> mean.x / (double) n);
      eip -> mean.y = floor( eip -> mean.y / (double) n);
      eip -> eqp_size         = InitialEqPointsTerminal * n;
      eip -> eqp        = NEWA (eip -> eqp_size, struct eqp_t);
      eip -> size_start = NEWA (n, int);
      eip -> eqpZ_size  = 10 * eip -> eqp_size;
      eip -> eqpZ       = NEWA (eip -> eqpZ_size, eterm_t);
      eip -> eqpZ_curr  = eip -> eqpZ;
      eip -> MEMB       = NEWA (n, bool);
      initialize_eqp_rectangles(eip);
      eip -> fsts_checked = 0;
#ifdef HAVE_GMP
      if (Multiple_Precision > 0) {
            qr3_init (&(eip -> cur_eqp.x));
            qr3_init (&(eip -> cur_eqp.y));
      }
#endif
      for (k = 0; k < n; k++) {
            eqpk = &(eip -> eqp[k]);
            memset (&(eqpk -> E), 0, sizeof (eqpk -> E));
            memset (&(eqpk -> DV), 0, sizeof (eqpk -> DV));
            eqpk -> E.x = pts -> a[k].x - eip -> mean.x;
            eqpk -> E.y = pts -> a[k].y - eip -> mean.y;
            eqpk -> E.pnum    = pts -> a[k].pnum;
            eqpk -> DV.x      = 0.0;
            eqpk -> DV.y      = 0.0;
            eqpk -> DV.pnum = k;
            eqpk -> R   = NULL;
            eqpk -> L   = NULL;
            eqpk -> S   = 1;
            eqpk -> UB  = 0.0;
            eqpk -> CHOSEN    = FALSE;
            eqpk -> Z   = eip -> eqpZ_curr++;
            *(eqpk -> Z)      = k;
            eip -> MEMB[k]    = FALSE;
      }
      save_eqp_rectangles(eip, 0, n-2);
      eip -> size_start[1] = 0;
      k = n;
      eqpk = &(eip -> eqp[k]);
      eip -> termlist = NEW_PSET(n+2);
      eqp_list = NEWA( eip -> eqp_size, struct eqp_t *);
      if (MaxFSTSize EQ 0) MaxFSTSize = n;
      for (size = 2; size <= MaxFSTSize-1; size++) {
            starti = eip -> size_start[(size-1)/2 + 1];
```

```
            endi   = k;
            if (size EQ 2) endi = n-1;
            eip -> size_start[size] = k;
            if (Print_Detailed_Timings) {
                  fprintf (stderr,
                        "- starting eq-point size %d"
                        " (total number eq-points is %d)\n",
                        size, k);
            }
            for (i = starti; i < endi; i++) {
                  eqpi = &(eip -> eqp[i]);
                  set_member_arr(eip, eqpi, TRUE);
                  generate_compatible_eqp(eip, size - eqpi -> S,
eqpi, eqp_list);
                  eqpp = eqp_list;
                  while (*eqpp) {
                        eqpj = *(eqpp++);
                        eqpj -> CHOSEN = FALSE;
                        j = eqpj -> E.pnum;
                        if (j > i)                        continue;
                        if (NOT disjoint(eip,eqpj))         continue;
                        for (iter = 1; iter <= 3; iter++) {
                              if (iter >= 2) {
                                    struct eqp_t * eqptmp = eqpi;
                                    eqpi = eqpj; eqpj = eqptmp;
                                    if (iter >= 3) break;
                              }
                              if (NOT projection_test_case_I(eip,
eqpi, eqpj)) continue;
                              eq_point_disp_vector(eip, eqpi, eqpj,
eqpk);
                              eqpk -> E = eip -> eqp [ eqpk ->
DV.pnum ].E;
                              eqpk -> E.x += eqpk -> DV.x;
                              eqpk -> E.y += eqpk -> DV.y;
                              eqpk -> E.pnum = k;
                              eqpk -> R  = eqpi;
                              eqpk -> L  = eqpj;
                              eqpk -> S  = eqpi -> S + eqpj -> S;
                              eqpk -> RP = eqpi -> E; eqpk -> RP.pnum
= -1;
                              eqpk -> LP = eqpj -> E; eqpk -> LP.pnum
= -1;
                              eqpk -> CHOSEN = FALSE;
                              eq_circle_center(&(eqpi -> E), &(eqpj
-> E), &(eqpk -> E), &(eqpk -> DC));
                              eqpk -> DR2 = sqr_dist(&(eqpk -> DC),
&(eqpi -> E));
                              if (NOT
projection_test_cases_II_VI(eip, eqpi, eqpj, eqpk)) continue;
                              eqpk -> DR = sqrt(eqpk -> DR2);
                              if (NOT bsd_test(eip, eqpi, eqpj,
eqpk))                    continue;
                              if (NOT lune_test(eip, eqpi, eqpj,
eqpk))                    continue;
                              new_Zp = merge_terminal_lists(eip,
eqpi, eqpj, eqpk);
                              if (NOT upper_bound_test(eip, eqpi,
eqpj, eqpk))              continue;
                              if (NOT wedge_test(eip, eqpi, eqpj,
eqpk))                    continue;
                              eip -> eqpZ_curr = new_Zp;
```

```
                                if (eqpk -> S > 2) {
                                      eqpoint_terminals(eip, eqpk, eip
-> termlist, FALSE);
                                      upper_bound =
upper_bound_heuristic(eip -> termlist, eip -> bsd);
                                      if (eqpk -> UB > upper_bound)
eqpk -> UB = upper_bound;
                                }
                                k++;
                                if (k >= eip -> eqp_size) {

                                      if (Print_Detailed_Timings)
                                            fprintf(stderr, "- doubling
eq-point array\n");
                                      eqp_old = eip -> eqp;
                                      eip -> eqp = NEWA ( eip ->
eqp_size * 2, struct eqp_t );
                                      memcpy ( eip -> eqp, eqp_old, eip
-> eqp_size * sizeof(struct eqp_t) );
                                      eqpi = UPDATE_PTR( eqpi, eqp_old,
eip -> eqp );
                                      eqpj = UPDATE_PTR( eqpj, eqp_old,
eip -> eqp );
                                      eqpk = UPDATE_PTR( eqpk, eqp_old,
eip -> eqp );
                                      eqppp = eqp_list;
                                      while (*eqppp) {
                                            *eqppp = UPDATE_PTR(
*eqppp, eqp_old, eip -> eqp ); eqppp++;
                                      }
                                      eqppp = eqp_list;
                                      eqp_list = NEWA( eip -> eqp_size
* 2, struct eqp_t *);
                                      memcpy ( eqp_list, eqppp, eip ->
eqp_size * sizeof(struct eqp_t *) );
                                      eqpp = UPDATE_PTR( eqpp, eqppp,
eqp_list );
                                      free( eqppp );
                                      for (eqpt = &(eip -> eqp[n]);
eqpt <= eqpk; eqpt++) {
                                            eqpt -> L = UPDATE_PTR(
eqpt -> L, eqp_old, eip -> eqp );
                                            eqpt -> R = UPDATE_PTR(
eqpt -> R, eqp_old, eip -> eqp );
                                      }
                                      for (sz = 1; sz < size; sz++)
                                       if (eip -> eqp_squares[sz] NE
NULL)
                                        for (si = 0; si < eip ->
srangex * eip -> srangey; si++)
                                          for (l = 0; l < eip ->
eqp_squares[sz][si].n; l++)
                                            eip ->
eqp_squares[sz][si].eqp[l] =
                                              UPDATE_PTR( eip ->
eqp_squares[sz][si].eqp[l], eqp_old, eip -> eqp );
                                      free( eqp_old );
                                      eip -> eqp_size = 2 * eip ->
eqp_size;
                                }
                                eqpk++;
                          }
```

```
                }
                set_member_arr(eip, eqpi, FALSE);
            }
            save_eqp_rectangles(eip, eip -> size_start[size], k-1);
        }
        if (Print_Detailed_Timings)
            fprintf(stderr, "%d eq-points generated.\n", k);
        ep = &(eip -> bsd -> mst_edges [1]);
        for (i = 1; i < n; i++) {
#ifdef HAVE_GMP
            if (Multiple_Precision > 0) {
                update_eqpoint_and_displacement (eip,&(eip -> eqp
[ep -> p2]));
            }
#endif
            eip -> termlist -> a[0] = pts -> a[ ep -> p1 ];
            eip -> termlist -> a[1] = pts -> a[ ep -> p2 ];
            eip -> termlist -> n    = 2;
            test_and_save_fst (eip,&(eip -> eqp[ ep -> p1 ]),&(eip ->
eqp[ ep -> p2 ]));
            ++ep;
        }
        if (Print_Detailed_Timings) {
            convert_delta_cpu_time (buf1);
            fprintf (stderr, "Generating eq-points:   %s\n", buf1);
        }
#ifdef HAVE_GMP
        if (Multiple_Precision > 0) {
            qr3_clear (&(eip -> cur_eqp.y));
            qr3_clear (&(eip -> cur_eqp.x));
        }
#endif
        free( eqp_list );
        free( eip -> termlist );
        destroy_eqp_rectangles(eip);
        free( eip -> MEMB );
        free( eip -> eqpZ );
        free( eip -> size_start );
        free( eip -> eqp );
        for (rp = eip -> list.forw;
             rp NE &(eip -> list);
             rp = rp -> forw) {
              rp -> next = NULL;
        }
        free ( eip -> hash );
        free ( eip -> term_check );
        shutdown_bsd (eip -> bsd);
}
static      dist_t test_and_save_fst (struct einfo * eip,struct eqp_t
*     eqpt,struct eqp_t *     eqpk)
{
int               i, j, k;
int               nedges;
int size, spidx, termidx;
dist_t length;
struct edge *           ep;
struct point *          sp;
struct point            nsp;
struct point *          tp;
struct pset *           pts;
struct point *          p1;
struct elist *          rp;
```

```
struct elist **         hookp;
struct elist *          rp1;
struct elist *          rp2;
struct pset *           new_terms;
struct pset *           new_steiners;
struct full_set * fsp;
struct edge *           edges;

      ++(eip -> fsts_checked);
      pts   = eip -> pts;
      size  = eip -> termlist -> n;
#ifdef HAVE_GMP
      if (Multiple_Precision > 0) {
            length      = compute_EFST_length (eip, eqpt);
      }
      else {
            length      = eq_point_dist (eip, eqpt, eqpk);
      }
#else
      length      = eq_point_dist (eip, eqpt, eqpk);
#endif
      k = 0;
      for (i = 0; i < size; i++) {
            j = eip -> termlist -> a[i].pnum;
            eip -> term_check [j] = TRUE;
            k += j;
      }
      k %= pts -> n;
      hookp = &(eip -> hash [k]);
      for (;;) {
            rp = *hookp;
            if (rp EQ NULL) break;
            if (rp -> size < size) { rp = NULL; break; }
            if (rp -> size EQ size) {
                  p1 = &(rp -> fst -> terminals -> a [0]);
                  for (i = 0; ; i++, p1++) {
                        if (i >= size) goto found_efst;
                        if (NOT eip -> term_check [p1 -> pnum])
break;
                  }
            }
            hookp = &(rp -> next);
      }

found_efst:
      for (i = 0; i < size; i++) {
            eip -> term_check [eip -> termlist ->a[i].pnum ] = FALSE;
      }
      if (rp NE NULL) {
            fsp = rp -> fst;
            if (fsp -> tree_len <= length) {
                  return (fsp -> tree_len);
            }
            *hookp = rp -> next;
            rp2 = rp -> forw;
            rp1 = rp -> back;
            rp2 -> back = rp1;
            rp1 -> forw = rp2;
            free ((char *) (fsp -> terminals));
            free ((char *) (fsp -> steiners));
            free ((char *) (fsp -> edges));
            free ((char *) fsp);
```

```
            free ((char *) rp);
      }
      new_terms = NEW_PSET (size);
      tp = &(new_terms -> a [0]);
      new_terms -> n = size;
      if (size <= 2) {
            new_steiners = NULL;
            nedges = 1;
            edges = NEWA (1, struct edge);
            tp [0] = eip -> termlist -> a [0];
            tp [1] = eip -> termlist -> a [1];
            edges -> p1   = 0;
            edges -> p2   = 1;
            edges -> len  = length;
      }
      else {
            new_steiners = NEW_PSET (size - 2);
            new_steiners -> n = size - 2;
            nedges = 2 * size - 3;
            edges = NEWA (nedges, struct edge);
            tp [0].x    = eip -> termlist -> a [0].x + eip -> mean.x;
            tp [0].y    = eip -> termlist -> a [0].y + eip -> mean.y;
            tp [0].pnum = eip -> termlist -> a [0].pnum;
            sp        = new_steiners -> a;
            ep        = edges;
            spidx     = size;
            termidx         = 0;
            project_point(&(eqpk -> E), &(eqpk -> DC), &(eqpt -> E),
&nsp);
            nsp.pnum    = spidx++;
            sp -> x         = nsp.x + eip -> mean.x;
            sp -> y         = nsp.y + eip -> mean.y;
            sp -> pnum  = nsp.pnum;
            ep -> p1    = termidx++;
            ep -> p2    = nsp.pnum;
            ep -> len   = EDIST(&(eqpt -> E), &nsp);
            sp++;
            ep++;
            build_efst_graph (eip, &nsp, eqpk -> R, &sp, &ep, &spidx,
tp, &termidx);
            build_efst_graph (eip, &nsp, eqpk -> L, &sp, &ep, &spidx,
tp, &termidx);
      }
      fsp = NEW (struct full_set);
      fsp -> next       = NULL;
      fsp -> tree_num         = 0;
      fsp -> tree_len         = length;
      fsp -> terminals  = new_terms;
      fsp -> steiners         = new_steiners;
      fsp -> nedges           = nedges;
      fsp -> edges            = edges;
      rp = NEW (struct elist);
      rp2 = &(eip -> list);
      rp1 = rp2 -> back;
      rp -> back  = rp1;
      rp -> forw  = rp2;
      rp -> next  = eip -> hash [k];
      rp -> size  = size;
      rp -> fst   = fsp;
      rp1 -> forw = rp;
      rp2 -> back = rp;
      eip -> hash [k] = rp;
```

```
      return (length);
}
/*
 * Construction of the graph of the e_steiner and filling in the
Steiner points.
 */
static      void build_efst_graph (
struct einfo *          eip,
struct point *          nsp,
struct eqp_t *          eqpk,
struct point **         sp,
struct edge **          ep,
int *             spidx,
struct point *          tp,
int *             termidx
)
{
int               idx;
int               pnum;
struct point            nnsp;
      if (eqpk -> R EQ NULL) {
            pnum = eqpk - eip -> eqp;
            idx = (*termidx)++;
            tp [idx] = eip -> pts -> a [pnum];

            (*ep) -> p1  = nsp -> pnum;
            (*ep) -> p2  = idx;
            (*ep) -> len = EDIST (nsp, &(eqpk -> E));
            (*ep)++;
      }
      else {
            project_point (&(eqpk -> E), &(eqpk -> DC), nsp, &nnsp);
            idx = (*spidx)++;
            nnsp.pnum     = idx;
            (*sp) -> x    = nnsp.x + eip -> mean.x;
            (*sp) -> y    = nnsp.y + eip -> mean.y;
            (*sp) -> pnum = idx;
            (*ep) -> p1   = nsp  -> pnum;
            (*ep) -> p2   = idx;
            (*ep) -> len  = EDIST (nsp, &nnsp);
            (*sp)++;
            (*ep)++;
            build_efst_graph (eip, &nnsp, eqpk -> R, sp, ep, spidx,
tp, termidx);
            build_efst_graph (eip, &nnsp, eqpk -> L, sp, ep, spidx,
tp, termidx);
      }
}
static      void renumber_terminals (struct einfo *   eip, struct
pset * to_pts,int * rev_map)
{
int               i;
int               j;
int               from_n;
int               to_n;
int               kmasks;
struct pset *           from_pts;
struct elist *          rp1;
struct elist *          rp2;
struct full_set * fsp;
struct pset *           terms;
struct point *          p1;
```

```
	from_pts    = eip -> pts;
	from_n            = from_pts -> n;
	to_n        = to_pts -> n;
	kmasks = eip -> num_term_masks;
	eip -> pts = to_pts;
	rp2 = &(eip -> list);
	for (rp1 = rp2 -> forw; rp1 NE rp2; rp1 = rp1 -> forw) {
		fsp = rp1 -> fst;
		terms = fsp -> terminals;
		p1 = &(terms -> a [0]);
		for (i = 0; i < terms -> n; i++, p1++) {
			j = p1 -> pnum;
			if ((j < 0) OR (j >= from_n)) {
				fatal ("renumber_terminals: Bug 1.");
			}
			j = rev_map [j];
			if ((j < 0) OR (j >= to_n)) {
				fatal ("renumber_terminals: Bug 2.");
			}
			p1 -> pnum = j;
		}
	}
}
static	void build_fst_list (struct einfo * eip)
{
int		i;
struct elist *		rp1;
struct elist *		rp2;
struct elist *		rp3;
struct full_set * fsp;
struct full_set **	hookp;
	hookp = &(eip -> full_sets);
	rp2 = &(eip -> list);
	i = 0;
	for (rp1 = rp2 -> forw; rp1 NE rp2; ) {
		fsp = rp1 -> fst;
		fsp -> tree_num = i++;
		*hookp = fsp;
		hookp = &(fsp -> next);
		rp3 = rp1;
		rp1 = rp1 -> forw;
		free ((char *) rp3);
	}
	*hookp = NULL;
	eip -> list.forw = rp2;
	eip -> list.back = rp2;
	eip -> ntrees	= i;
	eip -> hookp	= hookp;
}
static	void add_zero_length_fsts (struct einfo * eip,int ndg,int
** list)
{
int		i;
int		t;
int		u;
int		kmasks;
int *		ip1;
int *		ip2;
struct pset *		pts;
struct pset *		terms;
struct edge *		edges;
struct full_set * fsp;
```

```
	pts   = eip -> pts;
	kmasks      = eip -> num_term_masks;
	for (i = 0; i < ndg; i++) {
		ip1 = list [i];
		ip2 = list [i + 1];
		if (ip1 + 2 > ip2) {
			fatal ("add_zero_length_fsts: Bug 1.");
		}
		t = *ip1++;
		while (ip1 < ip2) {
			u = *ip1++;
			terms = NEW_PSET (2);
			terms -> n = 2;
			terms -> a [0]    = pts -> a [t];
			terms -> a [1]    = pts -> a [u];
			edges = NEW (struct edge);
			edges -> p1 = 0;
			edges -> p2 = 1;
			edges -> len      = 0.0;
			fsp = NEW (struct full_set);
			fsp -> next  = NULL;
			fsp -> tree_num    = (eip -> ntrees)++;
			fsp -> tree_len    = 0.0;
			fsp -> terminals = terms;
			fsp -> steiners    = NULL;
			fsp -> nedges      = 1;
			fsp -> edges       = edges;
			*(eip -> hookp) = fsp;
			eip -> hookp      = &(fsp -> next);
		}
	}
}
static	struct full_set ** put_trees_in_array (struct full_set *
fsp,int * acount)
{
struct full_set **	ap;
struct full_set * p;
int		count;
int		num;
struct full_set **	array;
	count = 0;
	for (p = fsp; p NE NULL; p = p -> next) {
		++count;
	}
	array = NEWA (count, struct full_set *);
	num = 0;
	ap = array;
	for (p = fsp; p NE NULL; p = p -> next) {
		*ap++ = p;
	}
	*acount = count;
	return (array);
}
static	cpu_time_t get_delta_cpu_time (void)
{
cpu_time_t	now;
cpu_time_t	delta;
	now = get_cpu_time ();
	delta = now - Tn;
	Tn = now;
	return (delta);
}
```

```
static      void convert_delta_cpu_time ( char * buf )
{
cpu_time_t  delta;
      delta = get_delta_cpu_time ();
      convert_cpu_time (delta, buf);
}
```

envelope.c

```
/*****************************************************************
***
      Routines to compute Euclidean Minimum Steiner Trees using the
ENVELOPE.
*/

#include "steiner.h"
#define ANSI_DECLARATORS
#define REAL coord_t
int               euclidean_mst (struct pset *, struct edge *);
dist_t                  euclidean_mst_length (struct pset *);
static int        build_euclidean_edges (struct pset *,struct edge
**);
static int *            heapsort_x (struct pset *);
dist_t euclidean_mst_length (struct pset *      pts)
{
int         i;
int         nedges;
dist_t            total;
struct edge *     ep;
struct edge *     edges;
      edges = NEWA (pts -> n - 1, struct edge);
      nedges = euclidean_mst (pts, &edges [0]);
      if (nedges NE pts -> n - 1) {
            fatal ("euclidean_mst_length: Bug 1.");
      }
      total = 0;
      ep = &edges [0];
      for (i = 0; i < nedges; i++) {
            total += ep -> len;
            ++ep;
      }
      free ((char *) edges);
      return (total);
}
int euclidean_mst (struct pset * pts,struct edge * edges)
{
int         nedges;
int         mst_edge_count;
struct edge *     edge_array;
      nedges = build_euclidean_edges (pts, &edge_array);
      mst_edge_count = mst_edge_list (pts -> n,nedges,&edge_array
[0],edges);
      free ((char *) edge_array);
      return (mst_edge_count);
}
static      int build_euclidean_edges (struct pset * pts,struct edge
**    edges_out)
{
int               i, i1;
```

```
int               j, j1;
int               k;
int               n;
int               nedges;
int               ndup;
struct edge *           edges;
struct point *          p1;
struct point *          p2;
int *             order;
int *             orig_vnum;
bool *                  dflags;
struct edge *           zedges;
struct triangulateio    in, out, vorout;
      n = pts -> n;
      if (n < 10) {
            /* Build of the graph*/
            nedges = n * (n - 1) / 2;
            edges = NEWA (nedges, struct edge);
            *edges_out = edges;
            p1 = &(pts -> a [0]);
            for (i = 0; i < n; i++, p1++) {
                  p2 = p1 + 1;
                  for (j = i + 1; j < n; j++, p2++) {
                        edges -> len      = EDIST (p1, p2);
                        edges -> p1 = ((pnum_t) i);
                        edges -> p2 = ((pnum_t) j);
                        ++edges;
                  }
            }
            return (nedges);
      }
            order = heapsort_x (pts);
      dflags = NEWA (n, bool);
      memset (dflags, FALSE, n * sizeof (dflags [0]));
      zedges = NEWA (n, struct edge);
      memset (zedges, 0, n * sizeof (zedges));
      ndup = 0;
      for (i = 0; i < n - 1; ) {
            i1 = order [i];
            p1 = &(pts -> a [i1]);
            for (j = i; ; ) {
                  ++j;
                  if (j >= n) break;
                  j1 = order [j];
                  p2 = &(pts -> a [j1]);
                  if (p1 -> x NE p2 -> x) break;
                  if (p1 -> y NE p2 -> y) break;
                  dflags [j1] = TRUE;
                  zedges [ndup].len = 0.0;
                  zedges [ndup].p1  = i1;
                  zedges [ndup].p2  = j1;
                  ++ndup;
            }
            i = j;
      }
      free (order);
      orig_vnum = NEWA (n, int);
      in.pointlist        = NEWA (2 * n, coord_t);
      j = 0;
      for (i = 0; i < n; i++) {
            if (dflags [i]) continue;
            in.pointlist [2*j    ] = pts -> a [i].x;
```

```
            in.pointlist [2*j + 1] = pts -> a [i].y;
            orig_vnum [j] = i;
            ++j;
      }
      in.numberofpoints = j;
      free (dflags);
      in.numberofpointattributes    = 0;
      in.pointattributelist         = 0;
      in.pointmarkerlist            = 0;
      in.numberofsegments           = 0;
      in.numberofholes        = 0;
      in.numberofregions            = 0;
      in.regionlist                 = 0;
      out.pointlist                 = 0;
      out.edgelist                  = 0;
      out.trianglelist        = 0;
      out.neighborlist        = 0;
      triangulate ("zeNBQ", &in, &out, &vorout);
      free (in.pointlist);
      free (out.pointlist);
      free (out.trianglelist);
      free (out.neighborlist);
      nedges = out.numberofedges;
      edges = NEWA (nedges + ndup, struct edge);
      *edges_out = edges;
      for (k = 0; k < ndup; k++) {
            *edges++ = zedges [k];
      }
      free (zedges);
      for (k = 0; k < nedges; k++) {
            i  = orig_vnum [out.edgelist [2*k    ]];
            j  = orig_vnum [out.edgelist [2*k + 1]];
            p1 = &(pts -> a [i]);
            p2 = &(pts -> a [j]);
            edges -> len      = EDIST (p1, p2);
            edges -> p1 = ((pnum_t) i);
            edges -> p2 = ((pnum_t) j);
            ++edges;
      }
      free (out.edgelist);
      free (orig_vnum);
      return (nedges + ndup);
}
static      int * heapsort_x (struct pset * pts)
{
int               i, i1, i2, j, k, n;
struct point *          p1;
struct point *          p2;
int *             index;
      n = pts -> n;
      index = NEWA (n, int);
      for (i = 0; i < n; i++) {
            index [i] = i;
      }
      for (k = n >> 1; k >= 0; k--) {
            j = k;
            for (;;) {
                  i = (j << 1) + 1;
                  if (i + 1 < n) {
                        i1 = index [i];
                        i2 = index [i + 1];
                        p1 = &(pts -> a [i1]);
```

```
                        p2 = &(pts -> a [i2]);
                        if ((p2 -> x > p1 -> x) OR
                            ((p2 -> x EQ p1 -> x) AND
                             ((p2 -> y > p1 -> y) OR
                              ((p2 -> y EQ p1 -> y) AND
                               (i2 > i1))))) {
                              ++i;
                        }
                  }
                  if (i >= n) {
                        break;
                  }
                  i1 = index [j];
                  i2 = index [i];
                  p1 = &(pts -> a [i1]);
                  p2 = &(pts -> a [i2]);
                  if ((p1 -> x > p2 -> x) OR
                      ((p1 -> x EQ p2 -> x) AND
                       ((p1 -> y > p2 -> y) OR
                        ((p1 -> y EQ p2 -> y) AND
                         (i1 > i2))))) {
                              break;
                  }
                  index [j] = i2;
                  index [i] = i1;
                  j = i;
            }
      }
      while (n > 1) {
            --n;
            i = index [0];
            index [0] = index [n];
            index [n] = i;
            j = 0;
            for (;;) {
                  i = (j << 1) + 1;
                  if (i + 1 < n) {
                        i1 = index [i];
                        i2 = index [i + 1];
                        p1 = &(pts -> a [i1]);
                        p2 = &(pts -> a [i2]);
                        if ((p2 -> x > p1 -> x) OR
                            ((p2 -> x EQ p1 -> x) AND
                             ((p2 -> y > p1 -> y) OR
                              ((p2 -> y EQ p1 -> y) AND
                               (i2 > i1))))) {
                              ++i;
                        }
                  }
                  if (i >= n) {
                        break;
                  }
                  i1 = index [j];
                  i2 = index [i];
                  p1 = &(pts -> a [i1]);
                  p2 = &(pts -> a [i2]);
                  if ((p1 -> x > p2 -> x) OR
                      ((p1 -> x EQ p2 -> x) AND
                       ((p1 -> y > p2 -> y) OR
                        ((p1 -> y EQ p2 -> y) AND
                         (i1 > i2))))) {
                              break;
```

```
                }
                index [j] = i2;
                index [i] = i1;
                j = i;
            }
        }
        return (index);
}
```

plotfst.c

```
/*****************************************************************
      Main routine for a utility to plot the trees in
      various ways.
*/
#include "genps.h"
#include "steiner.h"
int               main (int, char **);
bool              Print_Grouped_Full_Sets       = FALSE;
bool              Print_Overlaid_Full_Sets      = FALSE;
static void       decode_params (int, char **);
static void       usage (void);
static char *           me;
static bool       Print_Full_Sets = FALSE;
static bool       Print_Points = FALSE;
int main (int argc,char **    argv)
{
int               i;
int               nedges;
int               nmasks;
int               fpsave;
bitmap_t *        edge_mask;
bitmap_t *        all_fsets_mask;
bitmap_t *        no_fsets_mask;
int               count;
char              tbuf [20];
char              title [128];
struct cinfo            cinfo;
      fpsave = set_floating_point_double_precision ();
      setbuf (stdout, NULL);
      decode_params (argc, argv);
      init_tables ();
      read_phase_1_data (&cinfo);
      edge_mask   = cinfo.initial_edge_mask;
      convert_cpu_time (cinfo.p1time, tbuf);
      printf (" %% Phase 1: %s seconds\n", tbuf);

      /* Prepare for plotting all terminals. */
      define_Plot_Terminals (cinfo.pts, &cinfo.scale);
      nedges = cinfo.num_edges;
      nmasks = cinfo.num_edge_masks;
      all_fsets_mask = NEWA (nmasks, bitmap_t);
      no_fsets_mask = NEWA (nmasks, bitmap_t);
      for (i = 0; i < nmasks; i++) {
            all_fsets_mask [i] = 0;
            no_fsets_mask [i] = 0;
      }
      for (i = 0; i < nedges; i++) {
            SETBIT (all_fsets_mask, i);
      }
      if (Print_Points) {
            if ((cinfo.description NE NULL) AND
```

```
                (cinfo.description [0] NE '\0')) {
                  strcpy (title, cinfo.description);
            }
            else {
                  sprintf (title, "%lu points", (int32u)
cinfo.num_verts);
            }
            overlay_plot_subset (title, no_fsets_mask, &cinfo,
BIG_PLOT);
      }
      if (Print_Full_Sets) {
            plot_full_sets (all_fsets_mask, &cinfo, SMALL_PLOT);
      }
      if (Print_Grouped_Full_Sets) {
            plot_full_sets_grouped (all_fsets_mask, &cinfo,
SMALL_PLOT);
      }
      if (Print_Overlaid_Full_Sets) {
            sprintf (title,
                  "All FSTs:  %lu points,  %s seconds",
                  (int32u) cinfo.num_verts, tbuf);
            overlay_plot_subset (title, edge_mask, &cinfo, BIG_PLOT);
      }
      restore_floating_point_precision (fpsave);
      exit (0);
}
/*
 * This routine decodes the various command-line arguments.
 */
static      void decode_params (int argc,char ** argv)
{
char *            ap;
char        c;
      --argc;
      me = *argv++;
      while (argc > 0) {
            ap = *argv++;
            if (*ap NE '-') {
                  usage ();
            }
            ++ap;
            while ((c = *ap++) NE '\0') {
                  switch (c) {
                  case 'f':
                        Print_Full_Sets = TRUE;
                        break;
                  case 'g':
                        Print_Grouped_Full_Sets = TRUE;
                        break;
                  case 'o':
                        Print_Overlaid_Full_Sets = TRUE;
                        break;
                  case 'p':
                        Print_Points = TRUE;
                        break;
                  default:
                        usage ();
                        break;
                  }
            }
            --argc;
      }
```

```
}
/*
 * This routine prints out the proper usage and exits.
 */
static char *     arg_doc [] = {
      "",
      "\t-f\tPrints all full-sets in \"fly specks\" fashion.",
      "\t-g\tPrints full sets in \"grouped fly specks\" fashion.",
      "\t-o\tPrints all full-sets in overlaid fashion.",
      "\t-p\tPrints the point set.",
      "",
      NULL
};
static      void usage (void)
{
char **           pp;
char *            p;
      (void) fprintf (stderr,
                  "\nUsage: %s [-fgop]\n",
                  me);
      pp = &arg_doc [0];
      while ((p = *pp++) NE NULL) {
            (void) fprintf (stderr, "%s\n", p);
      }
      exit (1);
}
```