Embedding Unit Disk Graphs

Internship Report
ETH Zürich, January 30th, 2006 - April 28th, 2006

Jan-Maarten Verbree

Supervisors ETH Zürich
Regina O'Dell
Prof. Roger Wattenhofer

Distributed Computing Group
ETH Zürich

Gloriastrasse 35
CH-8092 Zürich
Switzerland

Supervisors University of Twente
Dr. Johann Hurink
Dr. Tim Nieberg

Chair of Discrete Mathematics and
Mathematical Programming
University of Twente

Drienerlolaan 5
7522 NB Enschede
The Netherlands

Preface

For the period of January 30th, 2006 through April 28th, 2006 I have been working on my internship project at the ETH Zürich. During this time I was mainly supervised by Regina O'Dell at the ETH, but my initial contact was with Professor Roger Wattenhofer. My supervisors at the University of Twente were Dr. Tim Nieberg and Dr. Johann Hurink.

An internship is a compulsory element of the study in Applied Mathematics at the University of Twente. The main focus of this practical training period lies on learning to use mathematical models in several real world problems and experiencing working in a company or at a university. During my mathematical education I have seen a lot of different parts of mathematics. I started with physical engineering and ended up with discrete mathematics with several courses in computer science, logic and telematics networks. With the help from one of my supervisors of the University of Twente, Tim Nieberg, I found this internship placement at the ETH Zurich, where they had several projects running that fitted well with my mathematical background.

Although this was my first time doing theoretical mathematical research myself, I enjoyed it and got used to it within a couple of weeks. Within four days I already decided which project I wanted to work on. My choice was to work on the embedding of a Unit Disk Graph, what was called a high-risk topic. The reason for this descriptive term was that the problem was hard, not much research was done on the problem and there was a high risk that I would not get any results. But on the flipside, these reasons made the topic very interesting and challenging (but sometimes also frustrating). The problem was also very intuitive and was strongly connected to discrete mathematics, so I could maybe use some of the mathematics I learned.

Gradually I understood more and more aspects of the problem and discovered that the problem was really hard. We have tried different approaches and ideas to investigate the problem and to get more insights. Unfortunately no big results were found, only algorithms that do not work and counterexamples for other algorithms, with some mathematical and intuitive explanation. Therefore, this report describes several faces of the problem and is not a systematical constructed proof towards a final theorem or statement.

I want to thank all my supervisors for giving me the opportunity for this internship. Special thanks to Tim Nieberg for the initial contact with the ETH and a lot of information about traveling, placement and funds for my internship. Thanks to Regina O'Dell for the many useful discussions and comments on my writings and presentation. To finish this preface, I wish the reader to have a pleasant time reading this report.

Jan-Maarten Verbree

Abstract

The many applications of wireless sensor networks make these networks very attractive and interesting for research. One of the goals of research is to create a network that has a long life-time, which means that all procedures should be very efficient because of the battery capacity of the sensor nodes. The networks become even more interesting if it is possible to let the nodes build their own network without intervention of a programmer. These ad hoc networks need algorithms to discover the network and to route messages. Geographic routing seems to be very useful for such networks, because it is simple, easy to implement and efficient. The only information needed to run this algorithm on a network is the position information of all the nodes.

In this report we look at methods to define positions and focus on virtual positions. Although it is strongly connected to routing in sensor networks it is more a mathematical problem. The mathematical description of the problem is: find an embedding for a Unit Disk Graph, given only its connectivity information. The goal is to find an intuitive algorithm that creates a reasonable embedding and to get more insight into the problems, as it is already known that the problem itself is NP-hard. By giving examples, the difficulties are shown and some core problems are discussed more extensively, like the problem of detecting the shape of a graph. This report also summarizes several (heuristic) algorithms and discusses the quality of these algorithms. There seems to be a general counterexample for these algorithms, but unfortunately it is also possible to show that the counterexample is not general enough for all possible algorithms. Although no proofs or new statements are presented in this report, it gives a better understanding of the difficulties of the problem of embedding a UDG.

Contents

Preface **2**

Abstract **3**

Contents **4**

1 Introduction **6**

1.1 Wireless sensor networks 6

1.2 Modeling a sensor network 7

1.3 Content 7

2 Problem **8**

2.1 Preliminary definitions 8

2.2 Problem statement 8

2.3 Related work to the problem 9

3 Intuitive Ideas on Embedding **10**

3.1 Grids 10

3.2 Cycles 10

3.3 Comparing cycles with trees 14

3.4 Conclusion 15

4 Detecting the Shape **16**

4.1 Introduction 16

4.2 The Shape of an Enclosed Spiral 16
4.3 A problematic graph 18
4.4 Inside or outside? 19
4.5 Extension to three spirals 21
4.6 Conclusion 23

5 Existing Algorithms and their Counterexamples **24**
5.1 Characterization 24
5.2 Algorithms using more than connectivity 25
5.3 Algorithms using only connectivity 27
5.4 The quality of the algorithms 29
5.5 Generality of the General Counterexample 34
5.6 Conclusion 34

6 Conclusion and Further Research **36**

Bibliography **37**

List of Abbreviations **39**

List of Symbols and Terms **40**

Chapter 1

Introduction

1.1 Wireless sensor networks

A wireless sensor network is nothing more than some sensor nodes that can communicate with each other by wireless signals. The sensors could measure temperature, light intensity, sound intensity et cetera, which can be used for many applications, such as monitoring environments. To make wireless sensor networks broadly applicable some restrictions on the sensor nodes have to be done. The devices have to be small and consume little energy. For some applications we even would like to admit mobility in the network without every time installing network nodes again. This means that the nodes have to detect by themselves that they can connect to a network and that no installing work by an outsider has to be done. Such networks are called ad hoc networks.

In many situations it is needed to know the positions of the sensor nodes to route messages between the nodes in a fast an efficient way. A Global Positioning System could be used for detecting the positions, but there are some disadvantages like the costs and the size of a GPS-system. The major disadvantage of GPS is that it is only applicable outside and not inside a building, as GPS-receivers need a straight connection with a GPS-satellite. Other solutions to the positioning problem use often anchor nodes, which are nodes that know their positions (e.g. by setting manually the positions for some nodes). The remaining nodes have to discover their positions by using mechanisms like Received Signal Strength Indicator (RSSI) or Time of Arrival (ToA). Both of these techniques compute their distances to other nodes by sending messages to the network nodes. Advantages of these mechanisms are the low costs and easiness of implementation. However, one of the major problems is the distortion in the signal due to interference, walls and other objects. This can lead to big errors in the computation of the positions. Another technique that is proposed to determine the positions is Angle of Arrival (AoA), which uses the angles of incoming signals to determine the place of a node in the area. The same problems as mentioned for RSSI and ToA also occur when this technique is used.

Although estimations of distances by using signal strength are not reliable, connectivity information is. If two nodes can receive each others signals, then they are connected, if not, they are obviously disconnected. Therefore, one would like an algorithm based on connectivity information only to compute the positions of the sensor nodes. It is straightforward to see that it is not possible to give absolute position of the nodes without using anchor nodes, but in many applications absolute positions are not necessary for routing. This way of thinking resulted in the goal to define virtual positions that are sufficient for routing messages over the network.

1.2 Modeling a sensor network

One of the simplest models for describing a sensor network is a Unit Disk Graph. This model is fully based on the connectivity between two nodes. It is assumed that all nodes have the same transmitting range and if two nodes are within each others range they are connected. If the transmitting range is scaled to 1, we get a Unit Disk Graph. The formal definition is given below and is taken from [12]. In Figure 1.1 an example is shown.

Definition 1.2.1 (Unit Disk Graph). *Let $V \subset \mathbb{R}^2$ be a set of nodes in the 2-dimensional plane. The Euclidean graph $G = (V, E)$ is called unit disk graph if any two nodes are adjacent if and only if their Euclidean distance is at most 1. That is, for arbitrary $u, v \in V$, it holds that $u, v \in E \Leftrightarrow |u, v| \leq 1$.*

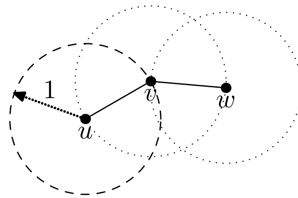


Figure 1.1: Example of a UDG. Node u is connected with node v, but not with node w.

Although this model is too idealistic, it is a clear model and therefore very useful for theoretical research. In this report we will also work with this model, the problem statement is even strongly connected to the model.

1.3 Content

This short chapter is only the introduction to the rest of the report. In chapter 2 we will state the problem and give mathematical background to the problem. Subsequently we will discuss several heuristic algorithms that have been proposed to solve the problem. Also counterexamples to these algorithms will be presented in that chapter. We conclude the official part of the paper with conclusions and advice for further research.

Definitions and symbols will be introduced when needed, but one can look up all used definitions and symbols in the summary that is given on page 40. The full meaning of abbreviations can be found on page 39. In the discussion of several heuristic algorithms original symbols and definitions ared used, which resulted in multiple definitions and notations for the same element. Normally one would like to avoid this, but I decided to keep the original notations, for it is more convenient if one would like to read the discussed paper.

Chapter 2

Problem

2.1 Preliminary definitions

Defining virtual positions is a possible solution to the routing problem as described in section 1.1. Once the positions are known we could use simply a geographic routing algorithm for sending messages over the network. Such an algorithm tries to forward messages in a greedy way by using the locations of the several sensor devices as their addresses. One of the characteristics of geo-routing is that its implementation is scalable, so it can also be used for networks with many nodes. In this report we discuss the possibilities to find coordinates for networks that can be modeled by Unit Disk Graphs. To make the problem clearer, we first introduce a formal definition for an embedding and for the quality of an embedding of a Unit Disk Graph.

Definition 2.1.1 (Embedding)**.** *An embedding of a graph $G = (V, E)$ in the Euclidean plane is a mapping $f : V \to \mathbb{R}^2$. In other words: every node v is mapped to a point (x, y) in the plane.*

Definition 2.1.2 (Quality of an embedding)**.** *Let f(G) be an embedding of UDG G=(V,E) in the plane. Let $d(u, v)$ denote the Euclidean distance between nodes u and v in f(G). The quality of the embedding f(G) is then defined as:*

$$q(f(G)) = \frac{\max_{(u,v)\in E} d(u,v)}{\min_{(u',v')\notin E} d(u',v')} \tag{2.1}$$

If we can embed a graph in such a way that it satisfies Definition 1.2.1, the quality of the embedding is less than one. An embedding that does not resemble a Unit Disk Graph will have a quality value bigger than one.

2.2 Problem statement

It is straightforward that if a graph is given, connectivity information is also available. This statement is of course also true for UDGs. The question is whether it also possible to create a UDG given only the connectivity information. This is the essence of the problem statement, which can be formulated formally by the following sentence:

Problem 2.2.1. *Find an algorithm to embed Unit Disk Graphs in the Euclidean plane and define the quality of the algorithm by giving theoretical bounds for the quality ratio as defined in Definition 2.1.2.*

The problem statement is clear and understandable, but is also very broad and undirected. To give some guidance through out my working on the problem, several goals have been formulated. First of all there is a huge need to an intuitive algorithm with a provable upper bound. In chapter 5 some algorithms that already exist are discussed, but all of them are not provable or not intuitive. This makes it hard to understand the problem better and to improve the algorithms. For first steps it can be useful to restrict the input of graphs or to assume more information available than only connectivity information. It will probably make the problem easier and more understandable. To reach this goal, a smaller goal or a sub-goal can be added: getting more insights into the difficulty of the problem. Even if an algorithm cannot be found, it is maybe possible to come up with small lemmas or theorems about the problem. The ultimate and final goal is of course finding an algorithm that can embed general UDGs given only connectivity information. Accomplishing this goal, but the other goals as well, means also giving proofs for the results and giving an understandable explanation.

2.3 Related work to the problem

Since Breu and Kirkpatrick [2] have proven that it is NP-hard to recognize whether a given graph is UDG, a lot of research followed to find the essence of the NP-hardness of the problem. Aspnes et al. [1] proved that the problem is still NP-hard when the $O(1)$hop distances of a UDG are given. The problem of embedding a UDG with known $O(1)$-hop angles is also NP-hard [3]. A more general UDG NP-hardness result was given by Kuhn et al. [6] by proving that finding an approximation of the constraints to within a factor of $\sqrt{3/2}$ is NP-hard. Finding a representation of a UDG can be solved in polynomial time if all the distances between the vertices, all the angles or the $O(1)$-hop angles and the $O(1)$-hop distances are known [3].

Although the problem is NP-hard, one would like to have an algorithm that gives at least a good approximation of a UDG. In [8] a provable algorithm for finding an approximation of a UDG is presented. All the other algorithms that have been proposed for approximating a UDG are not provable. In their work the researchers use heuristics and defend their algorithm with examples on random graphs. More on these algorithms can be found in chapter 5, in which the algorithms are explained and discussed.

Chapter 3

Intuitive Ideas on Embedding

Coming up with a new provable algorithm is not an easy task. Drawing graphs, thinking about proofs and sometimes implementing an idea are time-consuming and sometimes frustrating activities, but necessary to understand the problem well. In this chapter some ideas are presented to issue the difficulties of the problem. Only simple algorithms are given, to keep it intuitive and clear.

In this section some new terms are being used. An n-cycle, is a cycle with n nodes. The term $cycle - graph$ is defined by the following definition.

Definition 3.0.1 (Cycle-graph)**.** *A* $cycle - graph$ *is a graph* $G(V, E)$ *in which every node* $v \in V$ *is part of at least one cycle.*

3.1 Grids

The easiest thought is to embed nodes arbitrarily on a grid. Such an embedding is intuitive, but gives poor results. This can be understood very simply. Let us assume to have a UDG with n nodes and place them on a grid with sides of length $\sqrt{n}$. If we search for the largest distance between two grid nodes, we have to look at the distance between two opposite corner nodes, which is equal to $\sqrt{2n}$. These corner nodes can be connected by an edge, as we placed the nodes arbitrarily. Therefore, the quality of the embedding is according to definition 2.1.2 $\sqrt{2n}/1 = \Theta(\sqrt{n})$. For a hexagonal grid a similar computation holds.

3.2 Cycles

While working on the problem I soon noticed that $cycle - graphs$ seem to give a lot more information than trees. Therefore one would think that it is much easier to proof certain properties for $cycle - graphs$. This subsections consists some of the ideas that have been worked out.

3.2.1 Merging cycles

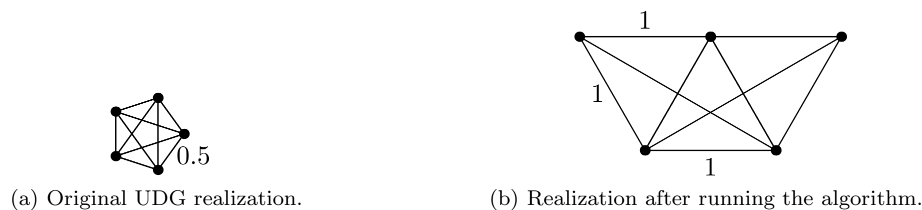
The algorithm-idea

In the first algorithm, all cycles have to be listed. This can be done in polynomial time (see [7] for an overview of several algorithms). Subsequently, all cycles are embedded in such a way that the nodes are placed on a circle with a certain radius, beginning with the 3-cycles and going up to the biggest cycles. The edges have a fixed length of 1. Nodes that are already positioned by the algorithm may not be changed. Every new cycle that has to be embedded is being fixed to the already embedded graph in a planar way, cycles may not overlap each other in the constructing phase. If merging the remaining nodes (the non-fixed nodes) is not possible, the best linear transformation is used to place these nodes on the Euclidean plane.

Problems of the algorithm

As the algorithm is very simplistic, we can find easily examples for which the quality of the embedding is poor. We hopefully get also more insight into the problem by these specific graphs. The counterexamples will be enumerated first and subsequently we will discuss the main problems.

The opening counterexample is a clique, a graph in which all nodes have a mutual distance less than one. In Figure 3.1a, the original graph is depicted and in Figure 3.1b the result after applying the algorithm. It can easily be seen that the edges have been stretched, which decreases the quality of the embedding. An even more problematic graph is a clique with more that six nodes, for our intuitive algorithm places the cycles next to each other. This will increase the size of the graph depending on the number of nodes. The result is comparable to placing nodes on a hexagonal grid, but now based on a certain idea and not in a random way. Next we show in Figure 3.2 a less interesting, but existing problem, when two big cycles that are connected by one edge. If all the nodes have to be placed in a circle, two unconnected nodes will be embedded very close to each other, namely the nodes that are next to the connected edge. In Figure 3.2 the nodes u and v represent one pair of these unconnected nodes. The bigger the cycles are, the shorter the distance between the two unconnected nodes. The last example (Fig. 3.3) gives more insight into the difficulty and shows a shortcoming of the algorithm: how does the algorithm know where to place the cycles? Both embeddings follow the procedure exactly, but the qualities of the embeddings are really different, for in Figure 3.3a one cycle overlaps the other and in Figure 3.3b the cycles are place at different sides of the bigger cycle. Therefore, the algorithm should have a decision system to decide on which side the cycles have to be placed.



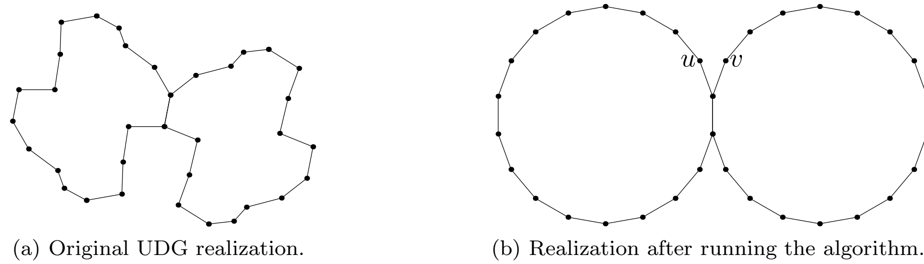
(a) Original UDG realization.

(b) Realization after running the algorithm.

Figure 3.1: Counterexample for merging cycles: a clique.

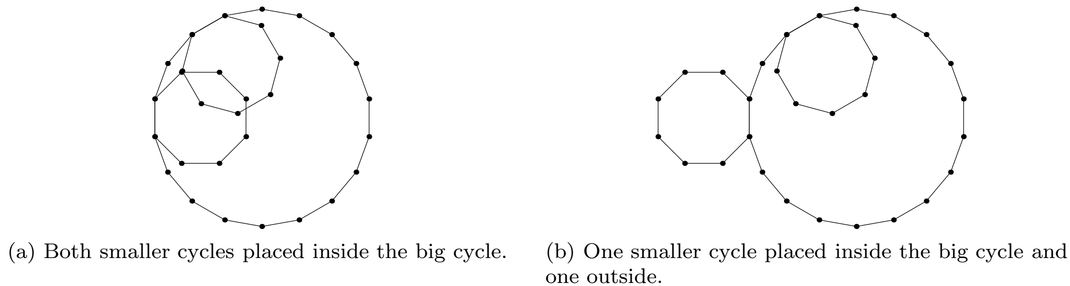
Analysis of the problems

Probably, there are beside these counterexamples many other graphs that will have a poor embedding after running the algorithm. A major reason for the poor quality of the algorithm is that it merges several maps and connects the triangles to each other. This is because of the rule that embedded nodes have fixed positions. The incremental way of connecting maps increments also the errors. In the example of the clique this is shown for a small graph: non-planar graphs are transformed into planar graphs if we only look at edges with a length of



(a) Original UDG realization. (b) Realization after running the algorithm.

Figure 3.2: Counterexample for merging cycles: two big cycles.



(a) Both smaller cycles placed inside the big cycle. (b) One smaller cycle placed inside the big cycle and one outside.

Figure 3.3: Shortcoming for merging cycles: where to place the cycles?

one or less. To handle the difficulty of non-planarity and to decide whether a cycle has to be embedded inside an other cycle it seems necessary to know more about the global structure. Cliques could possibly be detected, and also perimeter nodes or center nodes. Moreover it is needed to detect in which way the cycles are connected to each other. This information should be used to choose better positions for the nodes, instead of placing them always on a circle.

We could take all these improvements into account and compute the best fitting as an initial embedding. Optimization could even improve the embedding further. But by doing this, the problem becomes far less intuitive and even tends to be heuristic. Finding proofs for this algorithm is very difficult and no upper bound was found by me.

3.2.2 Distance levels

The algorithm-idea

A special kind of $cycle - graphs$ is a clique, in which every three nodes form a 3-cycle. This makes cliques important structures and therefore we want at least that cliques are properly embedded. The simplest way to do this, is to place all the nodes on one spot. However, if we have a cycle-graph that is not a clique, but may have several cliques as subgraphs, this method is not sufficient and too simplistic. Look for example at a cycle-graph with only 3-cycles, as shown in Figure 3.4. As every 3-cycle is a clique, all nodes in a 3-cycle will be placed on the same spot. Doing this for every 3-cycle leads to an embedding with all nodes of the graph on one spot, which gives a quality ratio of infinity. Although embedding cliques on one spot seems to be not very good for every graph, we would like to use the idea behind the algorithm: for every clique you want the nodes to be close to each other. In terms of cycles: if a node u and a node v share a lot of cycles, then one could say that they are somehow strongly connected and the nodes have to be placed close to each other. To get an algorithm we need to be more precize. Let $S(u, v)$ denote the number of cycles that vertices u and v share and $d(u, v) = g(S(u, v))$ the distance between those vertices. The function g should be chosen in a way that

it is a decreasing function in S, for example $g(S(u,v)) = \frac{1}{S(u,v)}$. This way of setting the distances will result in distance levels. Strongly connected vertices are on a low level and weakly connected vertices on a high level (Fig. 3.5). On each level k the algorithm can be used to calculate the places that approximate the distance levels on level k the most. Weakly connected nodes can be spread out more than strongly connected nodes. To get an embedding we start with a random node u. This node u counts for each of his neighbors the number of cycles his neighbors shares with it (node u) and computes the distances to his neighbors with the function $g(S(u,v))$. Subsequently the nodes that are on the same distance level will be placed such that they spread out on the circle with radius $g(S(u,v))$ and center u. After this step a new node has to be selected (for example one of the nodes with the biggest distance) and it has to run the same procedure as node u already has done. Nodes that are already placed are fixed. If all nodes are fixed the algorithm has computed an embedding. In Section 3.2.1 we already mentioned that listing all cycles is possible in polynomial time. This algorithm will therefore also have computation in polynomial time.

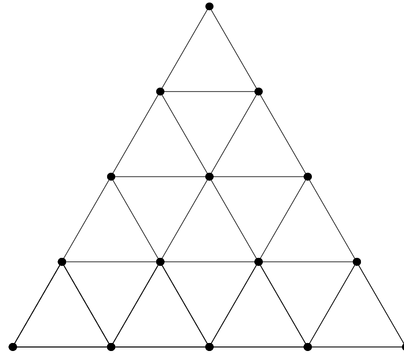


Figure 3.4: A graph with only 3-cycles.

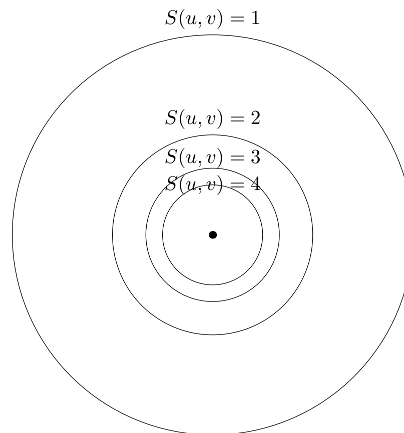


Figure 3.5: Distance levels, $d(u,v) = \frac{1}{S(u,v)}$.

Problems of the algorithm

The best way to name the first counterexample is an ‘almost’-clique, which is the graph after removing one edge of a clique. To implement this in a UDG we have to place two nodes a bit further from each other to let the mutual distance be larger than one. This is equivalent with a clique with two nodes less and on the left and right side one node connected to all the nodes in the clique. In Figure 3.6 an example is given. Let us assume that the algorithm is run by node u. If we add new vertices to this graph in the middle of the clique, nodes v_1 and v_2 will share more cycles with node u. The distance function was a decreasing function in S and therefore the vertices v_1 and v_2 will be deployed closer to each other as more vertices are added, depending on the function $g(S(u,v))$. According to Definition 2.1.2 the embedding ratio will grow, as the minimum length

of the non-edge between v_1 and v_2 decreases. The exact value of the ratio depends on the function and the number of (added) nodes in the clique. It is easy to see that if there are n nodes in the clique the ratio for our example with $g(S(u,v)) = \frac{1}{S(u,v)}$ will be $q = \Theta(n)$. The second counter example is simpler and shows a shortcoming. If we look at a huge cycle, then the algorithm gives reasonable edge lengths, but does not have a clue about the angles. An additional algorithm is needed to give realistic angles.

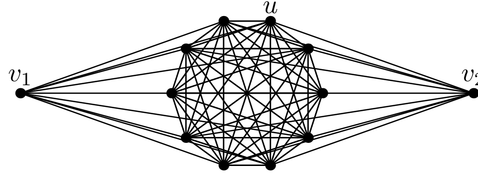


Figure 3.6: An 'almost'-clique.

Analysis of the problems

Similar to the previous one, the algorithm based on distance levels will only work well on certain graphs, but it has not enough information to give a good embedding for a random graph. Although it uses more information about the structure, e.g. it detects how strong nodes are connected, it still has a big lack of information about the global structure. An improvement could be to implement a sub-algorithm that uses the edge lengths and the number of nodes to compute pairwise angles for all connected edges in order to embed big cycles properly. But like the algorithm that merged cycles, this algorithm is also incremental. It deploys each node subsequently and not all nodes at the same time. An algorithm that places all nodes at one time is harder to understand and less intuitive, but the only way to avoid incrementing errors.

3.3 Comparing cycles with trees

The previous subsection gave some insights into the problems and we can draw the conclusion that finding a good embeddings of random $cycle-graphs$ is not as easy as it seemed to be. Embedding trees also seems to be difficult. Therefore, it is meaningful to look for a connection between those types of graph.

For some specific graphs a connection is very obvious. Look for example at Figure 3.7. If we place in each cycle a vertex and connect vertices if their surrounding cycles are connected by one or more nodes, we will get a tree. If we can a find an embedding of the corresponding tree, we have a good basis for embedding our $cycle-graphs$ and vice versa.

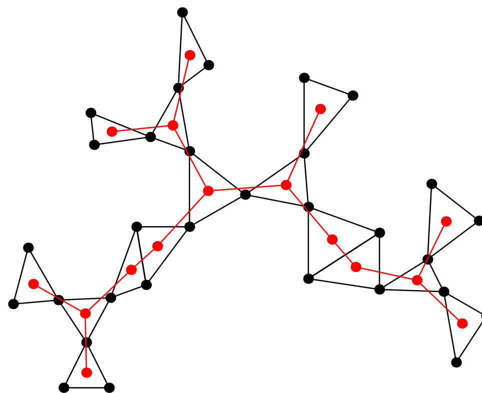


Figure 3.7: A 3-cycle graph with corresponding tree.

Although the idea is nice, it will probably not help much, as embedding a tree seems to be even more difficult than embedding a $cycle - graph$. Nevertheless, the connection between cycles and trees can give us more understanding about the $cycle - graphs$. If it can be proven that finding an embedding for a tree is hard, you can probably also prove that finding an embedding for the corresponding 3- and 4-cycles is hard. Until we have more information about embedding those cycle graphs or trees, the result is not usable, as it only gives an embedding if one of the two corresponding graphs can be embedded.

3.4 Conclusion

The two most important intuitive ideas that I have looked at have been presented in this chapter. These two examples show the same behaviour and difficulties as the other ideas that are not presented. Basically all my intuitive ideas worked on an incremental way, which increments also errors as said before, which can be seen as a first conclusion. A second conclusion is that the algorithms lacked information about the global structure. More knowledge is needed before embedding. For this reason we will look closer to detecting the shape of a graph in the next chapter. An interesting insight was found by comparing trees with $cycle - graphs$. Unfortunately, this insight could not be used for a purpose, for there are still no proofs for trees or $cycle - graphs$ concerning embedding.

If we look at all the examples and problems, then we can say the main conclusion of this capter is that embedding is not that easy at all. Although the problem statement is very intuitive, good intuitive embedding algoritms are not so easy to find.

Chapter 4

Detecting the Shape

4.1 Introduction

In Chapter 3 we concluded that it is needed to know more about the global structure of the graph. Detecting several characteristics like the center and perimeter nodes could give more information about the graph we want to embed. The algorithms presented in [9] and [10] contain such procedures to detect the structure. The first one selects perimeter nodes and the second one tries to find four corner nodes and a center node. In Chapter 5 we will discuss those algorithms and also some counterexamples.

A major problem of the algorithms is the use of hop distance. It is obvious that there is some difference between Euclidean distance and hop distance. In this section some more support on this idea is given and it is shown that the difference can grow quadratically. Next we will investigate the possibilities to detect the structure of a graph by measuring only hop distances.

4.2 The Shape of an Enclosed Spiral

In this section we will point out the problem of using hop distance to estimate Euclidean distance and to detect the shape. To make the problem clear we will create a special graph, an enclosed spiral. The basic idea which we know from circle packing theory is that placing circles on a hexagonal grid is the most efficient way to position circles on a plane without overlapping. We create our enclosed spiral, which we can us as a building block for creating more graphs with the same properties, in the following way. Let us have a hexagonal grid in a hexagonal shape with sides of l nodes and let the distance between the grid points be 1.

First we pick the nodes on the outer hexagon. Then we select one corner node of this hexagonal cycle as the starting point for the spiral and create the spiral. This can be done by choosing the first node of the line from the corner node to center and then going clockwise following the hexagon. But instead of making a closed cycle the path will lead towards the center just before it reaches the beginning node of the second hexagon. By following this procedure the final graph will be like the one that is depicted in Figure 4.1. For easiness in further computations we will shorten the last edge of the spiral, otherwise the last node would have been connected to several other nodes.

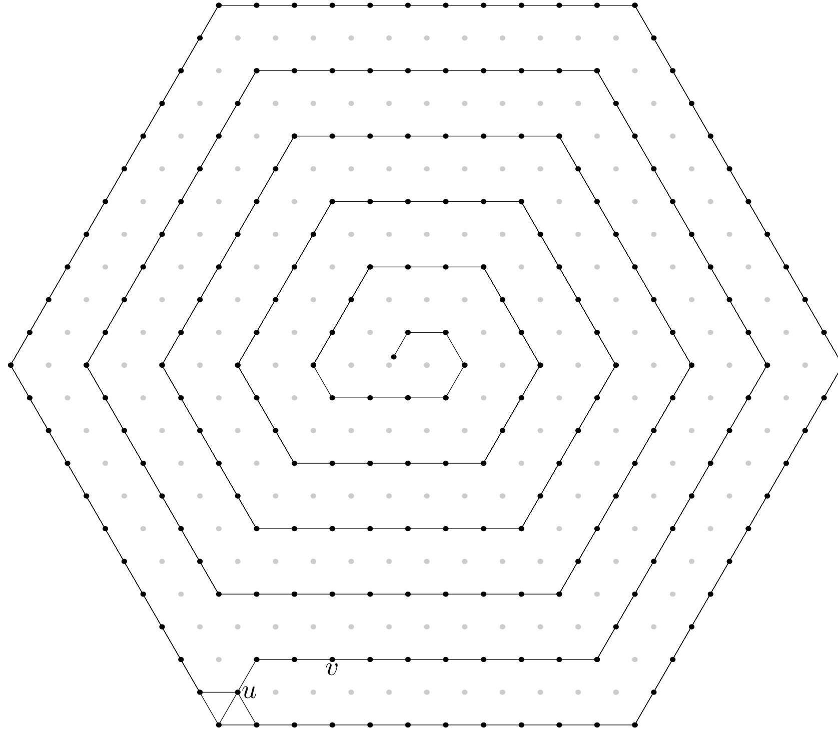


Figure 4.1: An enclosed spiral.

4.2.1 Lower bound calculations

The smallest number of edges that connects node u with node v is called the hop distance between u and v. We will denote this by $h(u, v)$. The hop distance between the nodes u and v in Figure 4.1 is $h(u, v) = 3$. We want to calculate the number of nodes of the spiral to say something about the hop distances. The total number of nodes can be computed by summing the number of edges of half of the hexagons. We do not have closed hexagons, but the two edges that we use to lead the spiral closer to the center can be used for each open hexagon to compensate the missing edges.

The number of nodes of the spiral $N_s(l)$ will therefore be:

$$N_s(l) = \sum_{k=1}^{l/2} 6(l - 2k) = \Theta(l^2) \tag{4.1}$$

Consequently the hop distance between the beginning and the end of the spiral will be $N_s(l)$ for a graph with an outside cycle edge of length l. The Euclidean distance is however at most l, as the nodes on the cycle are located on level l from the center. We can simplify the calculation of the minimum hop distance and subtract the Euclidean distance, to get a lower bound L_{dif} for the difference in distance measurements for nodes on the cycle and the node on the end of the spiral. The increasing difference function is as follows:

$$L_{dif}(l) = \left(\sum_{k=1}^{l/2} 6(l - 2k) \right) - l = \frac{3}{2}l^2 - 4l = \Theta(l^2) \tag{4.2}$$

As one can see in the example figure there is some more space for placing extra nodes in the spiral. By doing

this, the difference will grow more and more. In the further writing we assume the spiral to have the maximal number of nodes inside the cycle. Although we will not use this numerical result in further sections, we want to emphasize that the difference between Euclidean distances and hop distances can grow quadratically for certain graphs.

4.3 A problematic graph

With the building block, i.e. the enclosed spiral, we can construct several graphs that will have a big difference between the hop distances and Euclidean distances. We only have to be sure that in the realization the spiral has to be placed inside the outer cycle, to use the difference between the Euclidean and the hop distance. If we look at the building block again, we can see that there are two possibilities to get the spiral out of the cycle. The spiral has to be flipped around his starting point, or the cycle has to be flipped.

A simple method to prevent this is connecting two building blocks and adjust them a little to let the beginnings of the spirals be part of two cycles: the big outercycle and a small cycle that is attached to this bigger outercycle and which we will call a cage. With the beginning of the spiral we mean the part of the spiral that is connected to the cycle. For example, node u is in Figure 4.1 the starting point of the spiral. An example of the graph that we created by these adjustments is shown in Figure 4.2. Assume we want to have a realization with the left spiral outside the cycle. Flipping the spiral is not possible for the cage is too small. But flipping outer cycles is also impossible, because the two cycles have the same sizes and the result would be that unconnected nodes (those of the spiral and the cycle) would have a distance less than 1. Because of the assumption of the presence of the biggest possible spiral in the cycle we can not reduce the size of the right cycle.

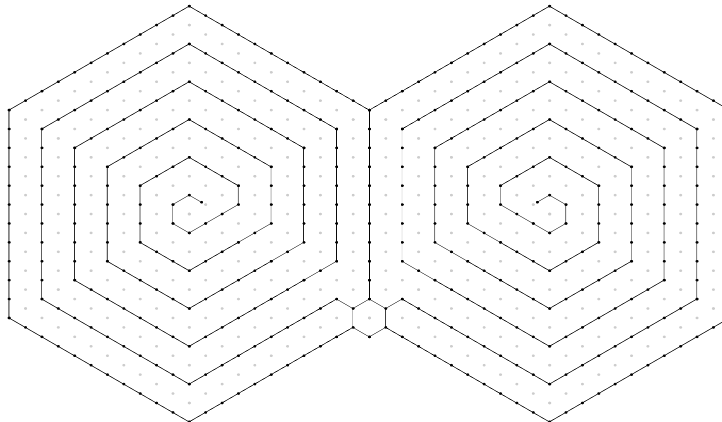


Figure 4.2: Two hexagons with enclosed spirals.

By using the building blocks and the above mentioned method we can create a lot of different graphs. One of the graphs is shown in Figure 4.4. The cage is replaced by a wheel graph, but this does not result in possibilities for embedding a spiral outside a cycle. Most algorithms for detecting the structure of a graph search for the center node and the perimeter nodes. Although several methods can be used to define the graph center, there is a classical definition for the graph center. This center is the set of central points, which are the nodes that have the lowest eccentricity. The eccentricity $\epsilon(u)$ of a node u in a connected graph $G(V, E)$ is the maximum hop distance between u and any other node $v \in V$. In Figure 4.3 an example of eccentricity values is given; the graph center is indicated by the big black node. By using this definition the center of the graph in Figure 4.4 will be the center of the little wheel. A common method for finding perimeter nodes is looking at the hop distances of all the nodes and subsequently selecting those nodes whose distances to the center are the biggest. In our graph this will result in selecting the nodes at the end of the spirals. By selecting the incorrect perimeter nodes wrong information will be used for embedding a graph. If one for example wants to position all nodes insides the hull of the selected perimeter nodes, the spiral has to cross the outercycle somewhere or the end of the spiral has to be at the same distance from the center as the node of the outercycle that is the farthest away

from the center. By this, the shape of the graph will not be preserved. Therefore the question arises whether it is possible to discover if the spiral of given UDG is positioned inside or outside a cycle by using only hop distances and we will discuss this in the next paragraph.

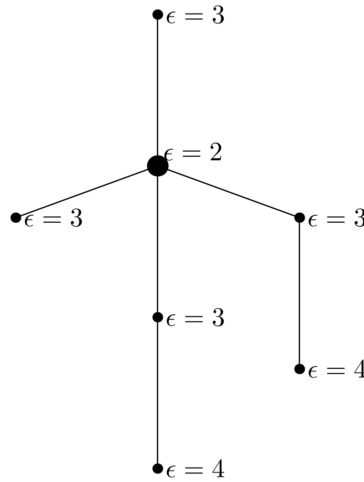


Figure 4.3: An example for eccentricity values.

4.4 Inside or outside?

The term *free* spiral is used in this and the following sections to indicate that there is a perfect UDG embedding for the graph in which the spiral is not enclosed by any cycle. This means that there exists an embedding in which the nodes of the spiral are not positioned between nodes of a cycle. By this definition it is clear that the spirals in Figure 4.2 are not *free*, but that the spiral in Figure 4.1 is *free* as the perfect embedding as shown in Figure 4.5 exists.

In order to get a clear vision of what the problem is, we create a graph that depicts the problem. If we take the graph from Figure 4.2 and remove one of the outer cycles around a spiral, we are almost done with making a good example of the problem. We only make the cage a bit smaller, in order to have it exactly four edges. The result is depicted in Figure 4.6. For the two connected hexagons we have proven that the spirals are fixed inside the cycles. By the adjustments to the graph we have some other properties. It is easy to see that it is not possible to get the spirals inside the little square, but the spirals are not totally fixed. The left spiral can be swapped by the right spiral and vice versa. Consequently the hop distances between the nodes of the graph do not change. Therefore we can conclude that it is not possible for a node to detect whether he is enclosed by a cycle by using only hop distances, but we can of course also conclude that one spiral is *free* and one not.

Because both of the potential realizations are UDGs, it does not matter which spiral will be embedded inside a cycle. But a decision has to be made to point out the spiral that will be embedded inside the cycle. The rest of the structure is of course of great importance for this decision, but in our case we do not have more useful information and we could decide possibly in a random way. Though, the algorithm has to know if a spiral is inside or outside a cycle, as the embeddings of these paths (spirals) can be very different. Inside the cycle the path has to be embedded compact, but outside the cycle this could lead to a poor quality if there are a lot of nodes around the square at the outside of the cycle (which is by the way not the case in our depicted graph). This problem is basically the essence of the NP-hardness proof in [2].

Of course it is possible to detect cycles in a graph by looking at the connections, which is essentially nothing else than using hop distances. But most of the algorithms that start with an initial embedding based on the size of a graph and use only the hop distances of the node they want to embed. To detect the structure it is needed to first discover if the graph contains cycles. Then there has to be computed if it is inevitable that a

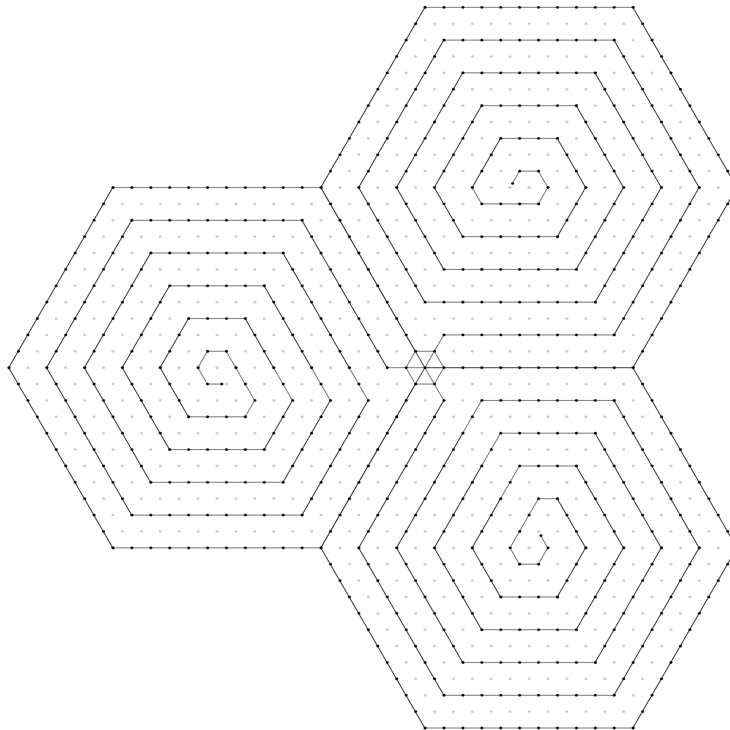


Figure 4.4: Three hexagons with enclosed spirals.

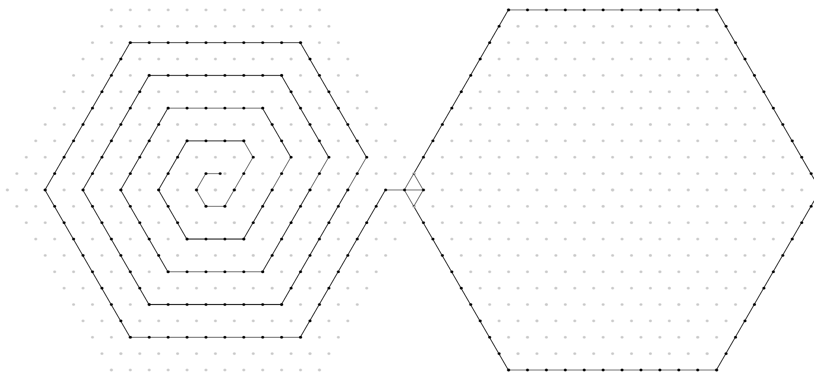


Figure 4.5: A perfect UDG embedding of the graph in Figure 4.1 with the spiral not enclosed by the cycle.

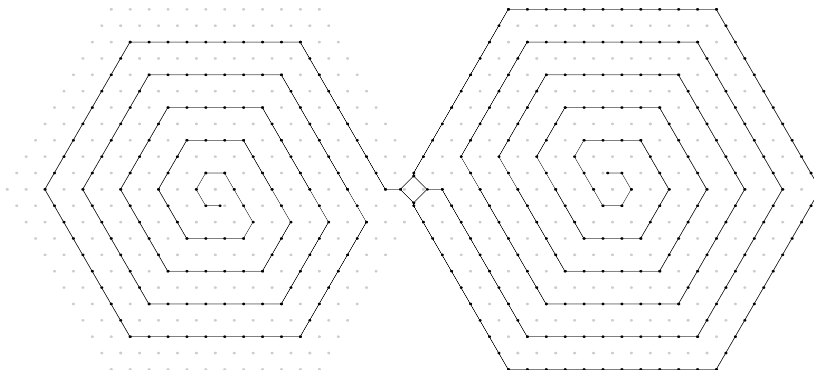


Figure 4.6: A graph with a *free* spiral and an enclosed spiral.

path has to be embedded in a cycle, whereupon can be decided which path has to be embedded inside the cycle.

4.5 Extension to three spirals

To see if the problem can occur in similar ways for other graphs, we modify the graph in Figure 4.4. Obviously we first have to remove an outer cycle. To be sure that only one spiral is outside a cycle, we also remove the center node that is mentioned in Section 4.3. The resulting graph after applying these adjustments is shown in Figure 4.7. For the two connected hexagons we have already proven that the spirals are fixed inside the cycles. It is easy to see that the graph consists of only one $free$ spiral, which can not be placed inside a cycle by definition. Therefore the structure of the graph is fixed, except for the directions of the rotations of the spirals.

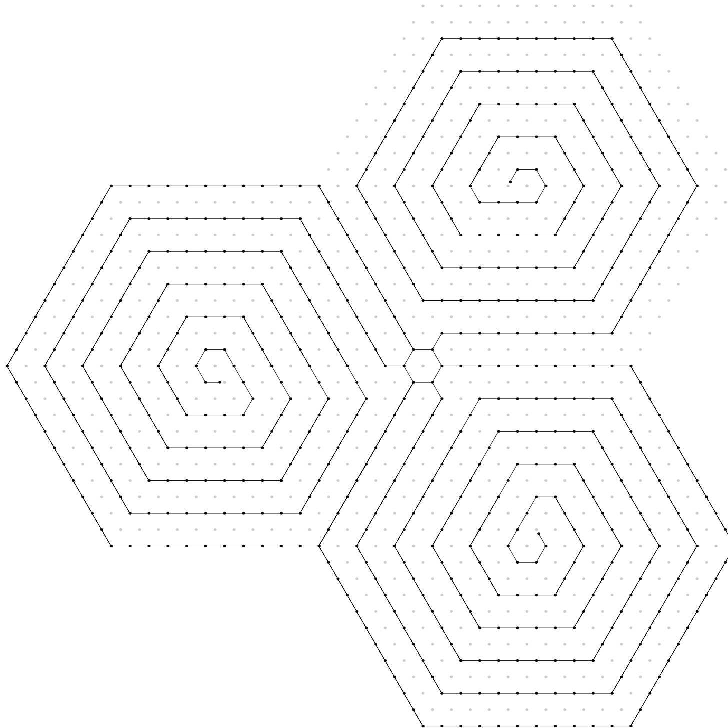


Figure 4.7: A graph with two enclosed spirals and one $free$ spiral.

Let us number the spirals and the nodes on the spiral in the following way. If we have a graph with several spirals S_i, we can number the spirals in a clockwise manner. With the notation $S_{i,j}$ we mean the node from spiral i that is j hops removed from the connected part of the spiral, i.e. the starting point of the spiral, so with $S_{1,3}$ we indicate node v. In Figure 4.7 we choose the $free$ spiral to be S_1 and the other spirals are numbered S_2 and S_3. The remaining nodes, i.e. the nodes of the cycles, can also be numbered in order to give every node an identifying number. We will denote them by R_i. The node that is connected to the beginnings of the spirals S_1 and S_2 will be called R_0 and we if we go along the cycle around spiral S_2 (clockwise) we can number every node by counting up and will reach finally the node that is connected to the starting points of S_2 and S_3. Now we have to number the nodes around S_1 and we do this on the same way, clockwise around spiral S_3, but ignore the nodes that are already numbered. This method of numbering will give every node a unique identification number. Then we can derive the following equations, for the three spirals are all connected to the hexagon with side length 1.

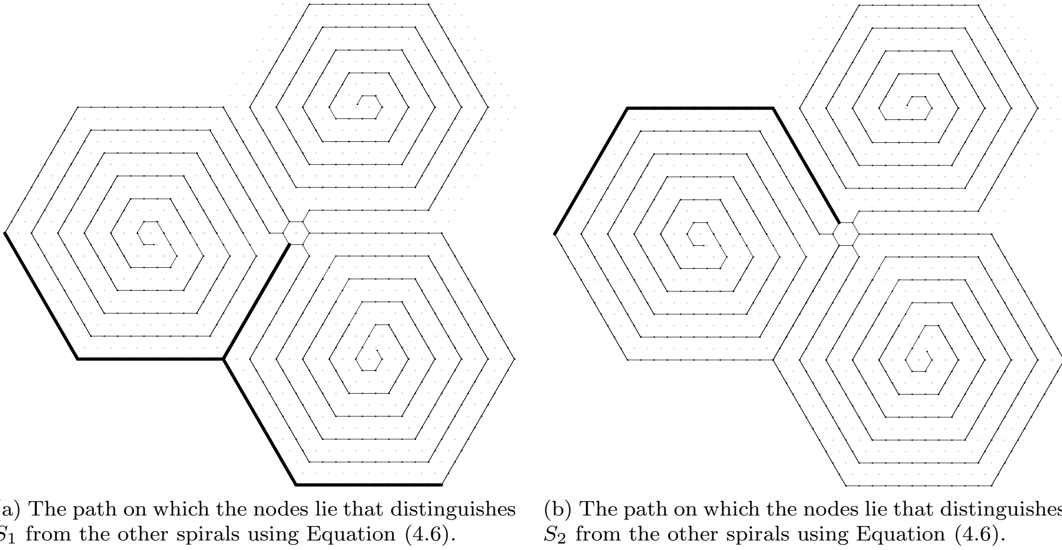
$$h(S_{s_1,i}, S_{s_2,j}) = h(S_{s_1,j}, S_{s_2,i}) \tag{4.3}$$

$$h(S_{1,i}, S_{2,j}) = h(S_{3,i}, S_{2,j}) = h(S_{1,i}, S_{3,j}) \tag{4.4}$$

$$h(S_{1,i}, S_{1,j}) = h(S_{2,i}, S_{2,j}) = h(S_{3,i}, S_{3,j}) \tag{4.5}$$

$$|h(S_{s_1,i}, R_k) - h(S_{s_2,i}, R_k)| \leq 2 \tag{4.6}$$

Equation (4.3) can easily be understood if we look at a path with fixed distance. Because of the symmetry of the two spirals in one of the nodes of the smallest hexagon, we can flip the path in that node with preserving the distance. In the next equation (Eq. (4.4)) the idea is formalized that for every node in a specific spiral there are two nodes, each in one of the remaining spirals, at the same distance. The fact that the spirals are identical is made clear by noting that all hop distances in one spiral also occur in the other two spirals. This is done by Equation (4.5). We can simply observe that all paths with one endpoint at the spirals and the other endpoint at the cycles will go via the little cage in the middle. As the nodes on the cycle can only be reached via the cage and the maximum distance difference between a node in the cage and the starting points of the several spirals is 2, we can state that the distances to one of the remaining nodes from a spiral node $S_{i,j}$ can differ mostly 2 over all i. This is formally done by Equation (4.6).



(a) The path on which the nodes lie that distinguishes S_1 from the other spirals using Equation (4.6).

(b) The path on which the nodes lie that distinguishes S_2 from the other spirals using Equation (4.6).

Figure 4.8: Two graphs that point out which information can be used to detect which of the spirals is the *free* spiral.

A global view at these equations shows us that we have to use Equation (4.6) to detect the *free* spiral S_1, as the other equations state that there is an equivalent spiral to S_1 that gives exactly the same hop distances. We know of course that there has to be some differences in the hop distances, otherwise we could swap two spirals. We have to focus on the nodes that distinguish S_1 from S_2 and S_3 and therefore we look at Figure 4.8a, in which the path with the nodes that will give $h(S_{1,i}, R_k) - h(S_{2,i}, R_k) = h(S_{1,i}, R_k) - h(S_{3,i}, R_k) = 2$ is printed boldly. Although this gives information to distinguish S_1 from the other spirals, the question is if we also can detect that S_1 is a *free* spiral. To explain this we have to look at Figure 4.8b, which shows the path on which the nodes have $h(S_{2,i}, R_k) - h(S_{1,i}, R_k) = h(S_{2,i}, R_k) - h(S_{3,i}, R_k) = 2$. This figure makes it clear that every spiral has nodes to distinguish it from the others and we see that S_1 has almost twice as much of these nodes as S_2. But for every hop distance between a node $S_{1,j}$ and one of the nodes that distinguishes S_1 we can find a node that distinguishes S_2 and has the same hop distance to $S_{2,j}$. That all hop distances can be found in both

situations is because of the 'symmetry' of the nodes that distinguish S_1. As for both spirals S_1 and S_2 they have similar hop distances it probably makes it harder to detect the $free$ spiral by using only hop distance from a node of the spiral. I could not yet find a proof to formalize the idea that it is not possible, but the examples make it clear that there are reasons to support this idea.

4.6 Conclusion

In this chapter some examples were given to illustrate the problem of detecting the structure in two ways. In the first part a lower bound for the difference between the Euclidean distance and the hop distance for an enclosed spiral was presented. The increasing difference function formalizes the idea that hop distance can not be used as good estimators for the real Euclidean distance. But this is not a new insight, only a new example of a graph that gives this result.

With this analysis we created a building block to show the problem of finding the structure. Commonly used ideas to find perimeter nodes can not handle the building block in a good manner and lead to wrongly detected perimeter nodes. For the cycle that includes a spiral ensures the problems, we had to take a closer look at the possibilities to find out if a spiral is enclosed by a cycle. In a simple way it could be shown that a spiral node on itself could not decide by his hop distance if he was enclosed. It seems that only an extensive search for cycles before detecting the structure is needed.

The last part gave a short look in an extended graph. No real proofs were found, but some more insights into the difficulty of the problem.

All these insights lead again to the same conclusion that detecting the structure of the graph seems to be very difficult by using only hop distances. In Chapter 3 we stressed that it is needed to know more about the global structure and the shape to give even a reasonable approximation of a UDG embedding. Combining these two statements we have to conclude that approximating a UDG embedding is probaby not doable on an intuitive way. However, these chapters and this research did not provide any upper or lower bounds for the ratio of quality of an approximation, which is of course needed. In this chapter I made just a little start by giving some understanding of the core of the problem.

Chapter 5

Existing Algorithms and their Counterexamples

As it seemed to be difficult to find a provable intuitive algorithm, a better look into heuristic algorithms could be helpful. The authors of these algorithms claim that their heuristics give good results and support this with statistical information gathered from many uniformly distributed random graphs. But knowing that these still are heuristic algorithms, there are probably some counterexamples for which the result will be very bad. In this section we want to discover those counterexample and use the knowledge from the previous chapter. First, several algorithms will be summarized and subsequently we give counterexamples for some of the algorithms. We finish this section with conclusions.

5.1 Characterization

The algorithms used for giving positions to sensor nodes can be characterized in many ways. Bulusu et al. [4] gave a kind of characterization, but only for algorithms based on reference nodes. The two main categories in their paper are fine-grained localization and coarse-grained localization. In the first category algorithms use several time and signal strength methods to compute the distances between nodes and in the second category algorithms use only proximity to a given reference point. A more global characterization was given by Pryantha et al. [9]. They came up with two characteristics of algorithms: anchor-based or anchor-free and incremental or concurrent.

The characterization I have chosen to work with is one from a higher level: I distinct the type of algorithm that works with connectivity only and the type that uses more information. To find an embedding with only connectivity information is obviously more complicated than to find an embedding with given edge-length information (although the latter case is still complicated). Per type of algorithm one could use the characterizations of Pryantha and Bulusu to characterize the algorithms further.

Most of the algorithm that are discussed are not specifically written for UDGs, but for general graphs including UDGs. Only the papers [3], [10] and [8] model the network as a UDG.

5.2 Algorithms using more than connectivity

The UDG problem is strongly connected to wireless sensor applications. Therefore, a lot of algorithms are created with this background in mind. The result of this is that those algorithms use more information than only connectivity information, for example estimated distances or estimated pairwise angles. In Section 1.1 we already discussed the problems of gathering this information and it could be concluded that finding a UDG with such information is easier, but less reliable in the real world, because of the distortion. Nevertheless, the algorithms can have some ideas that are also applicable when only connectivity information is used, so it is worth looking at those algorithms. The algorithms of this category that are discussed in this section are listed below.

1. Locationing in Distributed Ad-Hoc Networks [11]

2. GPS-less Low Cost Outdoor Localization for very small devices [4]

3. Localization and Routing in Sensor Networks by local angle information [3]

4. Anchor-Free Distributed Localization in Sensor Networks [9]

5. Distributed Graph Layout for Sensor Networks [5]

5.2.1 Locationing in Distributed Ad-Hoc Networks

In the paper it is assumed that some anchor nodes are available with enough power to reach all the nodes. Two algorithms are presented: ABC and TERRAIN. ABC is very simple and intuitive: estimate the distances between a node and the anchor nodes by RSSI and use a triangulation method to calculate the positions. Its implementation is very straightforward. Pick a random node, call it n_0 and place it on the coordinate (0,0). The second node n_1 has to be placed at $(0, r)$ with r the distance between n_0 and n_1. For the third node the assumption is made that its y-coordinate is positive. With this assumption the coordinate can be calculated, by using the distances from n_0 to n_2 and from n_1 to n_2. All the other nodes are completely determined by these three nodes. TERRAIN is slightly more complicated, but is similar to ABC. At every node ABC is initiated and the exact positions are calculated in a later stadium. Measurement errors will be less propagated than by normal ABC.

Both algorithms depend on the RSSI . If there is a range error of 5% the average estimated position error, i.e. the actual offset of the node position from the true position, is close to 40% for TERRAIN and close to 60% for ABC.

5.2.2 GPS-less Low Cost Outdoor Localization for very small devices

In the title of the paper the authors mention the term 'outdoor localization' because they use anchor nodes, which is not very well applicable within buildings. The anchor nodes are placed on the four corners of a square full of nodes and have a certain transmission range. Every node can hear some anchor nodes and derives his position by simply taking the average of the x-coordinates and y-coordinates of these anchor nodes. No RSSI is used, but only connectivity with the anchor nodes. The average estimated position error for hundred nodes on a grid and four anchor nodes is almost 20%.

5.2.3 Localization and Routing in Sensor Networks by local angle information

In contrast to other algorithms that use distance information, this algorithm is based on angle information. The main part of the algorithm is a LP with constraints for the edge-lengths, cycles, non-adjacent node pairs and crossing edges. The heuristic objective function is chosen to be maximizing the minimum length of all edges. To reduce the number of constraints and variables, some extra work is done. For a rigid subgraph, i.e. a graph with only cycles of three edges, all the edge-lengths can be expressed by one variable. In special cases it is even possible to use only one variable for connected rigid subgraphs.

5.2.4 Anchor-Free Distributed Localization in Sensor Networks

The Anchor-Free Localization algorithm (AFL) proceeds in two phases. In the first phase an initial fold-free graph embedding is created and in the second phase a mass-spring based optimization corrects the errors in this initial embedding. It is assumed that mechanisms are available for a node to detect neighbors and distances to his neighbors, for example by using RSSI-measurements. The distances to neighboring nodes is only used in the second phase of AFL. So the first phase falls into the category "connectivity only". But as the whole algorithms uses more than only connectivity, this algorithm is discussed in this section.

To create an initial embedding four corners are detected. A random node n_0 is picked and node n_1 is determined by searching for the node with the largest distance to n_0 The distance is measured by hop-counts between nodes, which is denoted by $h(u, v)$ for the number of hops between nodes u and v. In case of a tie, the node with the lowest ID-number will be n_1. For n_2 a similar method is used, but now the distance between n_1 and n_2 has to be maximized. Reference node n_3 has to be chosen to minimize $|h(n_1, n_3) - h(n_2, n_3)|$. The tie-breaking rule is to select the one that maximizes $h(n_1, n_3) + h(n_2, n_3)$. The first step for selecting n_4 is similar to the one for n_3; the tie-breaking rule however differs. Now one has to pick the node which is the farthest away from n_3. The center node n_5 has to minimize $|h(n_1, n_5) - h(n_2, n_5)|$. From all the possible n_5-nodes the node that also minimizes $|h(n_3, n_5) - h(n_4, n_5)|$ is chosen to be n_5. Hop-counts are also used to give positions to all other nodes by using polar coordinates. In the case of a UDG where the maximum radio range is one, the formulas are the following:

$$\rho_i = h(n_5, n_i) \tag{5.1}$$

$$\theta_i = \tan^{-1}\left(\frac{h(n_1, n_i) - h(n_2, n_i)}{h(n_3, n_i) - h(n_4, n_i)}\right) \tag{5.2}$$

After this initialization step, the mass spring optimization is run concurrently at each node. The force of a node n_i on n_j is formulated as $\overrightarrow{F}_{i,j} = \hat{v}_{i,j}(\hat{d}_{i,j} - r_{i,j})$, in which $\hat{v}_{i,j}$ represents the (estimated) unit vector in the direction from n_i to n_j, $\hat{d}_{i,j}$ denotes the estimated Euclidean distance and $r_{i,j}$ denotes the measured distance (by for example RSSI). The resultant force on a node n_i can simply be calculated by taking the sum over all the forces that are working on node n_i. The squared difference between the estimated distance and the measured distance is taken as the energy of an edge and has to become smaller and smaller. If the total energy is less than a certain amount, the iterative process is stopped. Each node moves by the amount of $|\overrightarrow{F}_i|/(2m_i)$ with m_i the number of neighbors of node n_i. This value was selected empirically.

5.2.5 Distributed Graph Layout for Sensor Networks

The authors of the paper claim that the algorithm they present is the first fully distributed protocol. Like AFL it uses an initialization algorithm and a refinement algorithm. In the problem statement they assume that the

edge-lengths (l_{ij}) and connectivity information are known. The variable w_{ij} is introduced to measure in some way the similarity of adjacent sensors i and j. In their experiment they used $w_{ij} = \exp(-l_{ij})$. The goal for the initialization phase is to minimize the energy $E(n)$, defined as follows:

$$E(n) = \frac{\sum_{\langle i,j\rangle \in E} w_{ij} \|n_i - n_j\|^2}{\sum_{i<j} \|n_i - n_j\|^2} \tag{5.3}$$

By minimizing $E(n)$ adjacent nodes are located close to each other and non-connected nodes are separated from each other, what is exactly what one wants for a UDG embedding. With some matrix theory it is possible to show that the eigenvectors of $D^{-1}W$ minimize the equation. W denotes the matrix with all the elements w_{ij} and D is the diagonal matrix whose i'th diagonal entry the sum of the i'th row of W is. Coordinates of a node i can then be derived in a distributed way by the following two formulas:

$$x_i \leftarrow a\left(x_i + \frac{\sum_{\langle i,j\rangle \in E} w_{ij} x_j}{\sum_{\langle i,j\rangle \in E} w_{ij}}\right) \tag{5.4}$$

$$y_i \leftarrow \frac{y_i - min_i y_i}{max_i y_i - min_i y_i} - \frac{1}{2} \tag{5.5}$$

The formula for y_i is different from x_i, because the power iteration used in x_i will detect only the second eigenvector (the first eigenvector isn't used, because it is the constant eigenvector $(1, 1, ..., 1)$). For the technical background of deriving the third eigenvector I refer the reader to [5] . The scales of the x and y coordinates can be very different and therefore it is needed to normalize them in a certain way. An easy balancing constraint that is applicable in distributive protocols is $\max_i x_i - \min_i x_i = \max_i y_i - \min_i y_i$. The derivation method of the y coordinates ensure us that the right hand side of the equation is equal to one. If $x_i \leftarrow \frac{x_i}{\max_i x_i - \min_i x_i}$ is performed for every i, the balancing constraint is satisfied.

The second phase describes how to optimize the localized stress energy. Where in [9] is chosen for a gradient descent method, they prefer a different method: majorization. Two reasons found their preference. The first is that it is insensitive to the original scale and the second that no step size (the amount of movement) needs to be fixed. The task of the majorization algorithm is to minimize the stress function that is defined by:

$$Stress(x, y) = \sum_{\langle i,j\rangle \in E} (d_{ij(x,y)} - l_{ij})^2 \tag{5.6}$$

The basic idea behind the majorization method is that the stress on an edge between nodes x and y can be bounded from above by using the Cauchy-Schwartz inequality. For more insight in how this method is used one should read the paper [5].

5.3 Algorithms using only connectivity

As said before, the information that is very reliable in a network is the connectivity information. If nodes are connected, then we know for sure that the nodes are located within each other ranges, but we do not know precisely the distance between them. An unconnected pair of nodes is modeled by placing the nodes at a distance larger than the ranges of the nodes. For the algorithms that use only connectivity have less information, it is harder to compute an approximation of a UDG. Only a few algorithms are based only on the connectivity of the graph and I want to summarize three of them in this subsection, as listed below.

1. Geographic Routing without Location Information [10]
2. Improved MDS-based localization [13]
3. Virtual Coordinates for Ad Hoc and Sensor Networks [8]

5.3.1 Geographic Routing without Location Information

From the three scenarios that are presented in the paper, the last one is the most interesting. In this scenario nodes don't know their location, or whether they are on the perimeter, in contrast to the other scenarios where the nodes know one or both things.

The first step is to detect the perimeter nodes. A bootstrapping beacon (which can be selected in several ways) broadcasts messages to all the nodes in order to let the nodes discover their distance to the beacon. To decide whether they are on the perimeter, nodes look at their two-hop neighbors. If no node is farther away from the beacon than the 'looking' node, it will decide that it is on the perimeter.

When all the perimeter nodes are elected, they broadcast a message to the entire network. By this every perimeter node knows the distance between him and all the other perimeter nodes, what is called a perimeter vector. This vector is also broadcasted to the entire network, so that it is possible for every node to compute the coordinates of the other nodes in such a way that the Euclidean distance approximates the hop distances. A least square method takes care for the best fitting. As there are more solutions possible (by translating, rotating and flipping) another bootstrapping beacon and the center of gravity of the perimeters are used to canonicalize the computation. The final step for the perimeter nodes is projecting all the nodes on the circle with origin at the center of gravity of the perimeter nodes and radius equal to the average distance from the perimeter nodes to the center of gravity.

After the perimeter nodes have been fixed, the other nodes have to be placed on good spots. By passing the perimeter vectors the nodes already have some information about their position. Triangulation and a least square method result in an initial placement for every node. After these phases each node will determine iteratively its new position, by simply calculating the average x- and y-coordinates from is neighbors, what will be his new coordinate.

5.3.2 Improved MDS-based localization

MDS-MAP(P) is presented as an improvement of the MDS-MAP algorithm. The new algorithm does not use MDS on all the nodes for computing the global map, but works on small patterns and merges them to a global map. The local map is the map of the R_{lm}-hop neighborhood. R_{lm} is a variable that effects the computational time and quality; in the paper $R_{lm} = 2$ is chosen.

Every node computes a shortest path distance matrix (based on hop-counts) for all pairs of nodes in his R_{lm}-hop neighborhood. MDS is applied to this distance matrix to compute coordinates for the nodes within the local map. The MDS-algorithm tries to minimize the Stress-function, which is a function of the difference between the Euclidean and the hop-count distances. A refinement protocol can be applied to the result of MDS to correct some errors. In the paper a weighted least squared method is used to minimize the difference between the hop and Euclidean distances further.

The next step is to merge the local maps. First of all a core map needs to be selected. This is done by picking randomly a local map and appointing it to be the core map, denoted by M. By merging local maps to the core map, the embedded map will grow and finally give a representation of the whole network. Every local map is based on the neigborhood of a node p that can be used as the identifier of the local map. The selecting procedure to point out which local map has to merge is straightforward: select the map M_p with the largest number of

common nodes with M. The actual merging of two maps uses a function T which is a linear translation function that has to be found in such a way that the sum of squared errors between M and $T(M_p)$ for the non-fixed nodes is minimized. A node p is fixed if its local map M_p has merged to M. After this stage an embedding of the global graph is available for the next step. Likewise the procedure for local maps, there is also a refinement step that could be applied to the computed global embedding. The method is exactly the same as used by the refinement of the local maps.

The algorithm can also use distance information, if it is available. This information will only be used for the refinements of the local and the global maps.

5.3.3 Virtual Coordinates for Ad Hoc and Sensor Networks

As mentioned before, this is the only approximation algorithm with a provable upper bound. The algorithm is not based on heuristics and differs for that reason a lot from the already discussed algorithms. Four stages are used to compute a UDG embedding: solving linear constraints, finding a volume-respecting embedding, projecting the nodes randomly and running a final embedding algorithm.

The linear constraints use edge lengths as variables and include a triangle inequality constraint, maximum edge-length constraint and a spreading constraint for independent sets, for independent nodes have to be spread out. No objective function is added to the constraint, so it is not a real linear program, but nevertheless it gives pairwise distances. For this metric a volume respecting embedding, introduced by Feige, can be computed. This method projects the coordinates into a high-dimensional space and tries to keep non-edges far apart. Random projection is used to project the nodes from the high-dimensional space to the Euclidean space. It is proven that the nodes spread very well and the edge lengths will be bounded. If the plane is partitioned into a grid with a cell-width of $1/\sqrt{n}$ it can be shown that at most $O(\log^4 n \log\log n)$ independent points will lie in a cell. The final step is to give a location to the nodes in a cell. The nodes that are also elements of the Maximum Independent Set are placed on a refined grid arbitrarily. Remaining nodes are assigned to one of the nodes of the Maximum Independent Set, namely the one with whom they have a connection. These nodes are subsequently placed on a circle around the MIS-node. Neighbored nodes may get a place on the same spot. Non-neighbored nodes are spread out equally on the circle.

In the paper it is shown that the approximation factor of the algorithm is $O(\log^{3.5} n\sqrt{\log\log n})$.

5.4 The quality of the algorithms

The main goal of discussing several algorithms for approximating a UDG embedding is getting more insight in the problem. Hopefully this insight will lead to an algorithm that is applicable to general graphs. To get a better understanding of the quality of the heuristic algorithm we will search for counterexamples. If we can find such graphs that lead to a poor result, we might also find solutions to it. These solutions could then be used to construct a new algorithm that is applicable to all possible graphs. In Chapter 4 we used a building block to construct graphs that showed a huge difference between hop distances and Euclidean distances. We will make use of this knowledge and show that the building block as shown in Figure 4.1 is almost a general counterexample for the discussed algorithms.

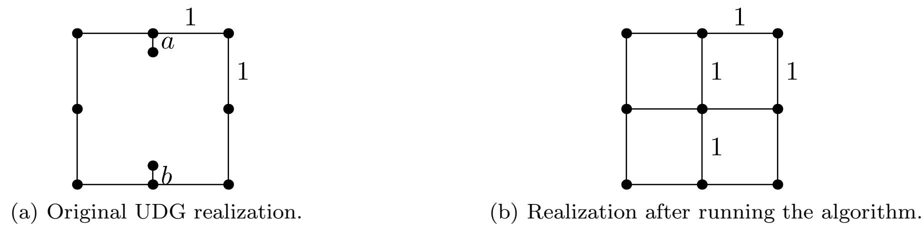
Due to the dependence of external information it is very simple to think of situations when the first type of algorithms will not function well. Walls, interference and measurement errors will decrease the quality of the computed embedding. But looking more precisely at the algorithms and finding counterexamples with obeying the assumptions can help us further in the search for a good approximation algorithm.

In this section I will not discuss the first two discussed papers ([4], [11]) for they make too many assumptions to be usable for a general algorithm. The performances of these algorithms are also very poor when distance errors

occur. For a totally different reason, the provable algorithm [8] will not be discussed. As the upper bound is proven, no counterexamples can be found that give worse quality of the embedding than the upper bound.

5.4.1 Localization and Routing in Sensor Networks by local angle information [3]

The main assumption in the algorithm of Bruck et al. [3] is that the angles between connected edges are given. By solving a linear program with constraints based on the known angles an embedding is computed. The problem of this algorithm is that it works with an objective function that tries to maximize the minimum length of all edges. For a simple graph as shown in Figure 5.1a the objective function is disastrous, as it tries to maximize the edge-lengths of edges a and b and keeps angles of the original realization. The final embedding is shown in Figure 5.1b. Of course, there are graphs where such situations will not be encountered, but with this example it is shown that there are also graphs that lead to a quality ratio of infinity, as two unconnected nodes are placed on the same spot. Implementing other objective functions could be an option, but we know for sure that minimizing edge-lengths will lead also to bad result (e.g. a star topology). The objective function has to be a function of the variables used in the constraints; otherwise it would be useless to solve the constraints. But as minimizing and maximizing edge lengths both lead to poor embeddings, a more sophisticated objective function is needed. To use the graph in Figure 4.1 also as a counterexample for this algorithm, we have to change the graph a bit. The last edge of the spiral can be divided into two edges, as shown in Figure 5.2. Then the embedding of the graph will have the same weakness as mentioned before: two nodes, u and v, will be placed on the same spot. The reasons for this are the use of angle information (so the shape is preserved and the spiral will not be embedded outside the cycle) and the maximizing of those two edges.



(a) Original UDG realization. (b) Realization after running the algorithm.

Figure 5.1: Counterexample for the angle-based LP algorithm.

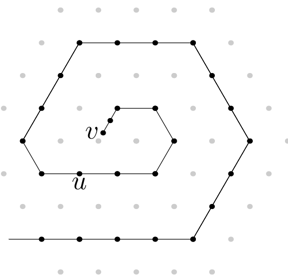


Figure 5.2: The changed end of the spiral.

5.4.2 Anchor-Free Distributed Localization in Sensor Networks [9]

The most logical method of finding a counterexample is looking at graphs that do not follow the idea of the algorithm. From the two phases of the Anchor-Free Distributed Localization algorithm [9] the first one is the most understandable in the way that you can predict what will happen. In this case we can find a graph that does not have four corners. Look for example at a simple tree graph with all edge-lengths equal to one, as

shown in Figure 5.3a. The algorithm picks a random node n_0, and detects the reference nodes by running the described protocol. The result is that the nodes n_3, n_4 and n_5 are placed on the same spot. Looking at the two formulas for calculating all the positions we notice that we have to divide by zero. In the paper they do not mention this problem and let us therefore assume that we have to take the limit. Consequently, the angle θ_i has to be the same for all i, what means that all the nodes are projected on a line, at a distance that is equal to the hop distance to reference node n_5. Several nodes having the same hop distance to n_5 will be placed on one spot, even if they are no neighbors.

Like one would expect, the final embedding is as bad as the initial embedding. The force to move each node to a better embedding is equal to zero, because the real (measured) distance between a node and his neighbor is the same as in the initial embedding. The initial embedding will therefore be also the final embedding, which has a quality ratio of infinity. Note that we did obey the assumptions, but just constructed a special graph. If we would have varied the edge-lengths in such a way that they would not be equal anymore, the nodes would have encountered a force. The final embedding would be probably better, but still very bad, for the nodes are on the same line and the opposite forces will also be on one line.

Applying the algorithm to or general counterexample gives also a poor example. In Figure 5.3b it is shown which nodes will be selected as center and corner nodes. Similar to the previous example for this algorithm, this will give a poor embedding with a quality of infinity, for several unconnected nodes will be placed on exactly the same spot in the final embedding. To be more specific, all nodes that have the same hop distance to n_5 will be placed on the same position, so the two paths from the beginning of the spiral to n_1 will get identical embeddings.

(a) The selected corner and center nodes of a specific counterexample.

(b) The selected corner and center nodes of our building block.

Figure 5.3: Counterexamples for AFL.

5.4.3 Distributed Graph Layout for Sensor Networks [5]

The presented algorithm is hard to understand, which makes it even harder to find counterexamples. Even though I could not find a provable counterexample, I have ideas where to look for such an example. The approach of the algorithm is similar to AFL, but it uses different methods to compute the embedding. It is therefore assumable to find a counterexample based on the initialization. The heuristic elements in this phase are the choices for the function to calculate w_{ij} and the constant a that controls the growth of $\|x\|$. The algorithm uses too much mathematical theory that is unfamiliar to me and besides that it is not clear how the initial positions are chosen. If we look at Equation (5.4) we can see that the x-coordinates are determined iteratively. Normally one could assume all nodes to be on the same spot, after which the nodes can detect their true positions by running the iterative process. But if we test this for the case where we assume all nodes to be at the origin, the positions of the nodes will not change, as we multiply by zero. This outcome makes it even more difficult to understand the idea of the algorithm. For these reasons I decided to skip this algorithm.

5.4.4 Geographic Routing without Location Information [10]

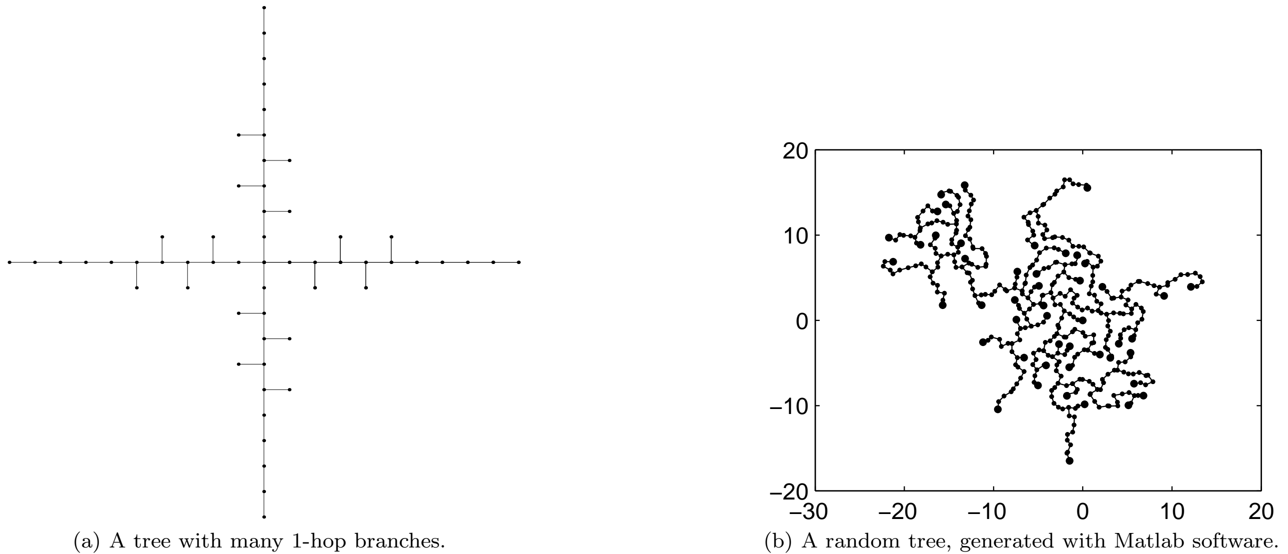
The algorithm for finding virtual coordinates for geographic routing as described in [10] is like other embedding algorithms based on an initialization and an optimizing phase. The goal of the initialization phase is to find the perimeter nodes, which is done by looking at 2-hop neighborhoods of every node. Because the initial embedding is very important for avoiding local minima, we will search for a graph that will lead to a poor initial embedding.

One of the simplest counterexamples for this algorithm is a cycle with $2 \cdot s$ nodes. If node n_1 is the designated beacon node and the nodes are numbered in a clockwise manner, node n_{s+1} will be the only node depicted as a perimeter. Triangulation can't be applied and therefore the node will get the coordinate $(0, 0)$. The spring embedding algorithm that is used to get the final embedding will place all the other nodes initially on the origin. The problem is that they will stay there during the iterative procedure and will not get a better position. For a cycle with an odd number of nodes the initial embedding will not be much better. Only two perimeter nodes will be detected and the other nodes will be placed by triangulation on both sides of the perimeter nodes, except for the beacon node. The procedure to determine the final x- and y-coordinate will lead to an embedding where all the non-perimeter nodes are placed between the two perimeter nodes, because of the symmetry and the placement of node n_1 on the y-axis.

According to the paper it is not a big problem if some nodes decide wrongly that they are a perimeter node, for the triangulation algorithm places the nodes inside the other perimeter-nodes. But if we look at a tree with a lot of branches (e.g. Figure 5.4a or Figure 5.4b) we can see that many nodes become a perimeter node, even if they are near the 'centre' of the graph. Intuitively one would like to have only the border nodes of a graph to be perimeter nodes, so that the perimeters form a convex hull. While the idea of the counterexample is simple, the proof of the counterexample seems to be hard. The problem is that the triangulation algorithm will replace the nodes, by minimizing the error between the calculated hop distance and the Euclidean distance of the coordinates. It is not easy to tell what the outcome will be of this algorithm for a certain graph, but it is clear that a lot of perimeter nodes will be inside the convex hull formed by the four corner perimeter-nodes. What will be the effect of this on the placement of the other nodes is hard to say and therefore this is only a counterexample-idea, not a real proven counterexample.

An other counterexample-idea is based on the problem of the difference between the hop distances and the Euclidean distances. In the triangulation algorithm the hop distances are used to calculate the positions of the several perimeter nodes. The difference between hop distance and Euclidean distances in a graph can be made as large as wanted. I am convinced that this will give some problems for certain graphs, but the least square approximation for coordinates is too difficult to predict. Several experiments with mathematical software support my ideas, though the question is whether the result of solving the least square problem is correct, it can give also local minima instead of global minima.

To make the algorithm suitable for mobility in the network, projection of the perimeter nodes on a circle was



(a) A tree with many 1-hop branches.

(b) A random tree, generated with Matlab software.

Figure 5.4: Counterexample ideas: Trees with many perimeter nodes.

introduced. Counterexamples for this last step can be found easily, but it gives no insights in placing virtual coordinates. The projection on the circle was meant for routing, the main focus of the paper, and not for finding a good embedding of a graph.

If we test the algorithm with our general counterexample as input, we also have to choose a beacon node to detect the perimeter nodes. Choosing the node that is on the cycle but the farthest away from the beginning of the spiral (so, the opposite node if we only look at the cycle), we will detect only one perimeter node. The end of the spiral is the only node that in his 2-hop neighborhood has the largest hop distance to the beacon node. It is obvious that the same result holds but vice versa if we select the node at the end of the spiral as beacon node. This case with one perimeter node is similar to the first counterexample and leads to a ratio of infinity. All other nodes that will be selected as a beacon node will give result to two perimeter nodes, what can be seen fairly easily. Consequently we have only two perimeter nodes at most for the building block and all the other nodes will be finally placed on the line between those nodes, because of the averaging procedure which is run by every non-perimeter node. The projection of the nodes on this line will result in a embedding with constant ratio. To see this, we point to Figure 5.5 in which the initial positions before the beginning of the averaging process are shown. By triangulation there are always two possible best fittings when two reference nodes are used and therefore we have chosen to take only the nodes with positive y-coordinate. The averaging procedure will try to place the nodes on the path $A - B - C$ on the line between the nodes A and D. Therefore, nodes will be placed close to each other. In the best case $A - D$ consists of $2l - 1$ nodes, with l the number of nodes on the side of the hexagon in Figure 4.1, and $A - B - C$ consists of $4l - 3$ nodes. The minimal distance between two unconnected nodes will consequently be $\frac{2l-2}{4l-4} = \frac{2(l-1)}{4(l-1)} = \frac{1}{2}$. For this result shows some shortcomings of our counterexample, we will discuss the generality of the counterexample further in Section 5.5.

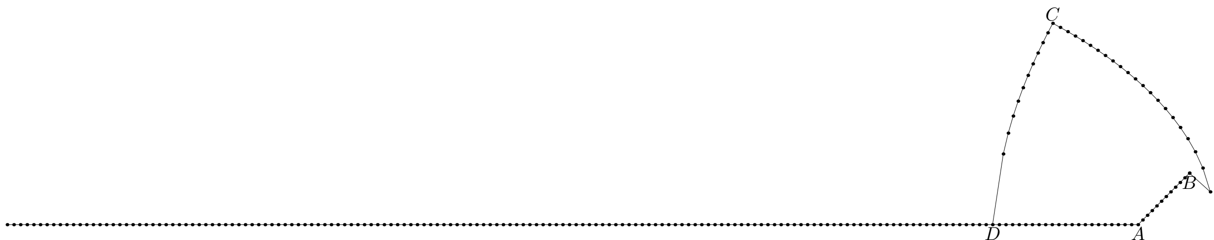


Figure 5.5: The initial placement after triangulation, but before the averaging procedure.

5.4.5 Improved MDS-based localization [13]

MDS-MAP(P) is an incremental algorithm in the way that it starts with a small embedding and extends it till the whole graph is embedded. The main problem of this kind of algorithms is that they also increase embedding errors. A counterexample shows this in a clear way. Assume we have a big cycle with s nodes, labeled n_1 to n_s. For easiness we assume also that the random node that has to be picked will be n_1. One of the neighbors of n_1 will merge his map with the initial map, suppose it will be n_2. When MDS is applied to the maps, both of them will have a straight line as representation. The intersection of these maps consist of $2 \cdot R_{lm}$ nodes and has to be transformed in such a way that the transformed map and the core map differ as less as possible. The straightforward transformation that puts the two lines on each other in order to place the same nodes on the same spot is the best transformation possible. This procedure will be run for every node and will have the result that the core map will extend to a long straight line. Finally a node n_i will have a R_{lm}-neighborhood that consist also a node of the local map of n_1. The only thing the algorithm can and will do is to find a transformation for the map $M(n_i)$ that will try to make the distance between n_1 and the $M(n_i)$ smaller. But this will just have a small effect on the embedding of the graph. A lower bound for the ratio of quality for this specific example can be given. The straight line will have a length of $s - (2 \cdot R_{lm} + 1)$ before a local map has to be embedded that has common nodes with the local map of n_1. At that moment only $2 \cdot R_{lm}$ nodes have to be fixed. The lower bound is therefore at least $\frac{s-(2 \cdot R_{lm}+1)}{2(2 \cdot R_{lm})} = \Theta(n)$, as we know from the already fixed nodes that the minimal length of a non-edge is at most 2. Connecting a spiral to the cycle does obviously not change this result. Therefore we can say that the general counterexample also holds for this algorithm.

5.5 Generality of the General Counterexample

Noticing that only one specific graph works as counterexample for some discussed graphs arises the question whether the graph gives for all algorithms poor embeddings. Obvious not, for the algorithm that detects perimeter nodes gives a constant ratio of quality (see Section 5.4.4). To see the shortcomings of the counterexample we will look again at the building block as shown in Figure 4.1. Of course we can embed the spiral outside the cycle, but for general use we assume that the building block will be used in a construction where this is not possible. Embedding the spiral inside the cycle will give the best ratio, but also needs a sophisticated algorithm to detect this. Simple algorithms will just use the hop distance and embed the spiral as a long straight line that crosses an edge of the cycle. An example of this is given in Figure 5.6. As one can see, this embedding is worse than the original, but has only a constant quality ratio, i.e. the ratio will not grow if the graph grows. This leads to the conclusion that our building block is not a very good counterexample in general.

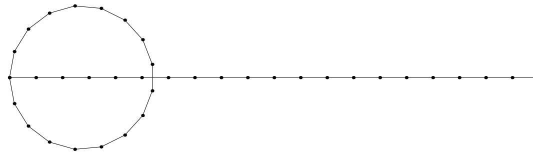


Figure 5.6: A possible embedding of a spiral 'enclosed' by a cycle. Crossing a cycle-edge gives quality of a constant ratio, adding more nodes is not helpful.

5.6 Conclusion

After looking at several algorithms and their counterexamples some general observations can be done. First of all, we can conclude that for the discussed heuristic algorithms counterexamples can be found for which the quality is very poor. This emphasizes the hardness of the problem and the need for provable algorithms. Two major problems for algorithms can be noticed. The main tool to estimate distances is to count the hops between two nodes. This is the only way of measuring distances, but it is a bad estimator, for hop-count distances do not

have to resemble Euclidean distances, look for example to a C-shaped graph. This is the first major problem; the second problem is the ignoring of the non-edges. I call it that way, because almost every algorithm, except for the provable algorithm [8], does not use the non-edges. Their focus on edges only leaves the possibility open to place non-neighboring nodes at a small distance from each other. By giving rules or constraints for non-edges this problem can be solved, like done in [8].

Unfortunately, our building block is not a general counter example, though it gives a better understanding of the problem. Extending the graph by attaching more nodes to it does not seem to be very promising. The long spiral can cross every edge, also the edges that are used to enlarge the graph, without increasing the ratio of quality. This insight gives us no further insight where to look for a new counterexample, which is even more disappointing.

Among all algorithms for positioning, there is one that is the most interesting and promising for further research, namely the algorithm of Moscibroda et al [8]. This algorithm is provable and by this it distinguishes itself from the other heuristic algorithms. But not only by the provability, also the method used in the algorithm is very different form the previously discussed algorithms. It contains a linear program for edge distances and uses besides the edge lengths also the lengths of non-edges. There is also a disadvantage of the algorithm, which also has to be mentioned. The algorithm is far from intuitive. The severe and sophisticated mathematical operations are hard to understand and therefore hard to improve. Some improvements could be done by adding constraints to get a better metric, but proving the effects of these added constraints on the final embedding and the upper bound is too difficult, because of the projections to other dimensions. Nevertheless, I will present my ideas for improvement. The first step of the algorithm computes a metric, but as it is not possible to describe a Euclidean metric linearly, the metric differs from the Euclidean metric. To develop a better metric, we need linear equations that are true for the Euclidean metric and thus can be used in the linear program. The triangle inequality is the most important and gives a lot of information, also about the diagonals in a quadrilateral. But the inequality only gives constraints for edges that are connected and it gives no constraint for the sum of two diagonals (except of the sum of two triangle inequality for the diagonals). Here some improvement can be done, as there are linear constraints that are stricter than the summed triangle inequalities. In a paper [15] of J. Töröcsik there are given three extra constraints, which are:

Theorem 5.6.1. *The sum of the diagonals of a quadrilateral is not greater than the sum of its three largest sides.*

Theorem 5.6.2. *The sum of the diagonals of a quadrilateral is not greater than the sum of its two largest sides and its smallest side.*

Theorem 5.6.3. *The sum of the diagonals of a quadrilateral is not greater than the sum of its largest side, the side opposite to it, and* $2\sqrt{2} - 2 \approx .83$ *times the average of the two other sides.*

As these constraints have to hold for every quadrilateral, this will result in a lot more constraints that have to be solved, depending on the number of nodes in the graph. But with this huge number of added constraints, the linear program will presumably give a far better metric. However, proofs are missing to confirm this assumption.

Chapter 6

Conclusion and Further Research

Embedding a Unit Disk Graph appears to be one of the hardest mathematical problems and it reveals his secrets not easily. In this internship report some understanding is given to various problems, mainly concentrated on detecting the shape of a graph. Proposed heuristic algorithms have been investigated and counterexamples have been given. All this information leads to several conclusions.

First of all, the problem of embedding a UDG on an intuitive way is very difficult. Several reasons can be given for the difficulty. One of the reasons is that placing all the nodes at the same time is hard to imagine, but it is needed to avoid an incremental algorithm. Incrementing a core map will also increment errors and will therefore lead to poor embeddings. Secondly, we noticed that the intuitive ideas of detecting the shape of a graph can result in selecting wrong perimeter nodes. There are also a lot of algorithms that use best fitting methods, which will give of course a best fitting, but are hard to predict. One would like to use some of these procedures to improve the algorithm, but it makes an algorithm less intuitive.

We can also conclude that the heuristic algorithms are not that good as they are presented. For many randomized graphs they give good result, but for simple counterexamples they give very poor embeddings. The initialization step is often the phase in which the errors are made and these mistakes cannot be corrected in the optimization step. If one wants to improve those algorithms more research has to be done on the initial embedding.

The question whether there is any hope that the problem of giving good embeddings for a UDG will have a solution is hard to answer, but it is a question that has to be asked. My expectation is that an algorithm can be found, but not an intuitive one. The advantages of a Linear Program should be used, in order to place all nodes on the same time. As there is one provable algorithm that uses already a Linear Program, there is some hope that it can be improved and give new insights. Perhaps this will lead to an algorithm with better embeddings that can be used in practice.

The main challenge for further research is to bring the findings and ideas to formal proofs. The basic idea that hop distances do not resemble Euclidean distances seems to be an idea that could be transformed to a proof, maybe for special graphs. A general counterexample is also needed, as the proposed counterexample does not give poor results for every algorithm. Once an example has been found, it can be used to give more understanding about the problems with heuristic algorithms. Other interesting topic of research is finding out how non-edges can be used in more intuitive algorithms. A first attempt could be a spring embedding algorithm for edges and non-edges, even though the computation costs are very high. There is some existing work on this topic, like the paper from Shavitt and Tankel [14]. This paper does not focus on UDGs, so there may be some improvements possible if one uses the characteristics of a UDG. Implementing and computing the effects of the improvements as described in section 5.6 could maybe lead to first steps to make the algorithm better. As the use of Independent Sets gives a reasonable metric, there may be some possibilities to use Independent Sets in more ways. Perhaps Independent Sets can also be used in intuitive algorithms that do no use a Linear Program.

The final conclusion after this global view on embedding algorithms is that there is a lot of work that still has to be done to find a good embedding for a UDG, for the heuristic algorithms are not that good and there is up to now only one provable algorithm.

Bibliography

[1] J. Aspnes, D. Goldenberg, and Y.R. Yang. On the computational complexity of sensor network localization. In *The 1st Int. Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS)*, pages 32–44, 2004.

[2] H. Breu and D.G. Kirkpatrick. Unit disk graph recognition is np-hard. *Comput. Geom. Theory Appl.*, 9(1-2):3–24, 1998.

[3] J. Bruck, J. Gao, and A. Jiang. Localization and routing in sensor networks by local angle information. In *Proc. of the 6th ACM Interational Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc 2005*, pages 181–192, 2005.

[4] N. Bulusu, J. Heidemann, and D. Estrin. Gps-less low-cost outdoor localization for very small devices. *IEEE Personal Communications Magazine*, 7(5):28–34, 2000.

[5] C. Gotsman and Y. Koren. Distributed graph layout for sensor networks. *Graph Drawing*, pages 273–284, 2004.

[6] F. Kuhn, T. Moscibroda, and R. Wattenhofer. Unit disk graph approximation. In *Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M)*, 2004.

[7] P. Mateti and N. Deo. On algorithms for enumerating all circuits of a graph. *SIAM Journal of Computing*, 5(1):90–99, 1976.

[8] T. Moscibroda, R. O'Dell, M. Wattenhofer, and R. Wattenhofer. Virtual coordinates for ad hoc sensor networks. In *Proc. 2004 joint workshop on Foundation of Mobile Computing*, pages 8–16, 2004.

[9] N.B. Priyantha, H. Balakrishnan, E. Demaine, and S. Teller. Anchor-free distributed localization in sensor networks. *SenSys*, pages 340–341, 2003.

[10] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proc. Mobicom'03*, pages 96–108, 2003.

[11] C. Savarese, J.M. Rabaey, and J. Beutel. Locationing in distributed ad-hoc wireless sensor networks. In *Proc. IEEE ICASSP*, pages 2037–2040, 2001.

[12] S. Schmid and R. Wattenhofer. Algorithmic models for sensor networks. In *The 14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2006.

[13] Y. Shang and W. Ruml. Improved mds-based localization. In *Proc. of the 23rd Conference of the IEEE Communicatons Society (Infocom 2004)*, 2004.

[14] Y. Shavit and T. Tankel. Big-bang simulation for embedding network distances in euclidean space. In *Proc. of IEEE Infocom*, 2003.

[15] J. Töröcsik. $n+1$ segments beat n. *SIAM Journal on Discrete Mathematics*, 6(3):491–500, 1993.

List of Abbreviations

Abbreviation	Description
AFL	Anchor Free Localization algorithm, as presented in [9]
AoA	Angle of Arrival
MDS	Multidimensional scaling
MIS	Maximum Independent Set
NP-hard	Non-deterministic Polynomial-time hard.
RSSI	Received Signal Strength Indicator
ToA	Time of Arrival
UDG	Unit Disk Graph

List of Symbols and Terms

Symbol	Description
$\epsilon(u)$	The eccentricity of a node u
ρ_i	The Euclian distance from node n_i to the origin, as used for polar coordinates in [9]
θ_i	The angle between node n_i and the positive x-axis, as used for polar coordinates in [9]
A, B, C, D	Specific nodes of the graph, explained in the text
$cycle - graph$	A graph in which every node is part of at least one cycle, see Definition 3.0.1
$d(u, v)$	Euclidean distance between nodes u and v
$\hat{d}_{i,j}$	The estimated Euclidean distance between nodes n_i and n_j, as used in [9]
D	The diagonal matrix whose i'th diagonal entry is the sum of the i'th row of W, as used in [5]
$E(n)$	The energy function for all nodes of a graph, as used in [5]
$f(G)$	The embedding of a graph G
$f : V \to \mathbb{R}^2$	The function that embeds nodes of a graph $G(V, E)$ to the Euclidean plane
$free$ spiral	A spiral that can be embedded outside a cycle without harming UDG constraints
$\overrightarrow{F}_{i,j} = \hat{v}_{i,j}(\hat{d}_{i,j} - r_{i,j})$	The force of a node n_i on n_j , as used in [9]
$g(Sh(u, v))$	A decreasing function in Sh
$G(V, E)$	A graph G with nodes V and edges E
$h(u, v)$	The hop distance between node u and v, i.e. the number of edges of the shortest path between u and v
k	The distance level, used in the embedding idea in Section 3.2.2
l	The length of the side of the hexagonal outercycle of the building block
l_{ij}	Length of the edge between nodes i and j, as used in [5]
$L_{dif}(l)$	The difference function between the hop distance and Euclidean distance from the beginning of the spiral to the end of the spiral
m_i	The number of neighbors of a node n_i, as used in [9]
M	The core map, as used in [13]
M_p	The local map of node p, which is the R_{lm}-hop neigborhood of node p, as used in [13]
$n_i, n_0, n_1, n_2, n_3, n_4, n_5$	Specific nodes of the graph, explained in the text
n-cycle	A cycle with n nodes
$N_s(l)$	Number of nodes of a spiral, depending on the length of the side of the hexagonal outercycle
p	Specific node of the graph, explained in the text
$q(f(G))$	The ratio of an embedding of a graph G

Symbol	Description
(x, y)	Coordinates of a node
r	The term that the ABC algorithm uses for the distance between two nodes n_0 and n_1 in [11]
$r_{i,j}$	The measured distance between nodes n_i and n_j, as used in [9]
R_i	A term to identify the remaining nodes of a graph, i.e. the nodes that not yet have an identifying number)
R_{lm}-hop neighborhood	The neigborhood of a node, which includes all nodes that are at a distance of less than R_{lm} hops from the specific node
s	Variable used once for the size of a cycle in Section 5.4.4
$S(u, v)$	The number of cycles nodes u and v share, i.e. the number of cycles in which nodes u and v are both elements of the cycle
S_i	Spiral i
$S_{i,j}$	The node from spiral i that is j hops removed from the connected part of the spiral, i.e. the starting point of the spiral
$Stress(x, y)$	A stress function used for majorization in [5]
T	Linear transformation function, as used in [13]
u	Specific node of the graph, explained in the text
v, v_1, v_2	Specific nodes of the graph, explained in the text
w	Specific node of the graph, explained in the text
$\hat{v}_{i,j}$	The estimated unit vector in the direction from n_i to n_j, as used in [9]
w_{ij}	A term to give a value of similairy of adjacent nodes i and j, as used in [5]
W	The matrix with all the elements w_{ij}, as used in [5]