Planarity Based Algorithms for Minor Embedding in Grid Graphs

by

Sahba Etezad

B.Sc., Ferdowsi University of Mashhad, 2014

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2016

Approval

Name:	**Sahba Etezad**
Degree:	**Master of Science (Computing Science)**
Title:	***Planarity Based Algorithms for Minor Embedding in Grid Graphs***
Examining Committee:	**Chair:** Eugenia Ternovska Associate Professor

David Mitchell
Senior Supervisor
Associate Professor _____

Pavol Hell
Supervisor
Professor _____

Andrei Bulatov
External Examiner
Professor
School of Computing Science
Simon Fraser University _____

Date Defended: July 20th, 2016

Abstract

We present heuristic algorithms for minor embedding planar graphs in grids. Our work is motivated by the development of quantum computing hardware that performs quantum annealing. This hardware can be used to solve hard combinatorial problems, but requires the graph of each problem instance to be minor embedded in a graph that models the hardware. Hence, there is a need for practical minor embedding algorithms. We restrict our attention to planar graphs, and thus are able to make use of existing graph drawing methods. We present two algorithms for minor embedding planar graphs in grid graphs, and provide an experimental evaluation of one.

Keywords: Simon Fraser University; Planar embedding, Minor-embedding, D-Wave, Adiabatic quantum annealing, Planar graphs, Grid graphs

Dedication

This thesis is dedicated to my parents whose unfailing support has helped me all these years, and my husband and best friend, Ehsan, who has been there for me in every turn of this journey.

Acknowledgements

I would like to express my sincere gratitude to my senior supervisor, Dr. David Mitchell, for his continuous support, vast expertise, guidance, and patience. I am grateful for his peaceful presence and insight which got us through chaotic and stressful times in the best way possible.

I would like to thank Dr. Pavol Hell, my supervisor, and Dr. Andrei Bulatov, my examiner, for their helpful comments. I would also like to thank Dr. Eugenia Ternovska for serving as the chair of my examining committee.

My deepest appreciation goes to my family and friends for their support and encouragement. Special thanks go to my uncle, Hamid Akbari, and his family whose kindness and support has helped me throughout my studies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the recent approaches toward solving NP-hard problems is *adiabatic quantum computing*. D-Wave Systems Inc. is the first company to produce a commercially available computing device which uses *quantum annealing*, a type of adiabatic quantum computing. This device, named *D-Wave Two*, solves NP-hard problems and can achieve good quality solutions, as good as the other known conventional devices such as CPLEX, but much faster. On the other hand, before using this device, we need to overcome a few challenges. The main challenge is the restricted formatting for feeding the problem instance to the device. We have to minor embed a graph representing the problem instance in a graph representing the hardware. The minor embedding of two given graphs is a NP-hard problem itself.
This thesis studies the slightly restricted problem of minor embedding an *input* planar graph in a *target* grid graph. Without any knowledge or information regarding the structure and attributes of the input graph, the probability of success in obtaining a feasible minor embedding of good quality can be very low. Therefore, we approached this challenge by exploiting the existing graph drawing techniques. These techniques can provide valuable insight and information for the minor embedding process.
The material in Chapter 4 except subsection 4.4.2 is based on a joint work with another master student, Ehsan Tavakoli, with approximately equal collaboration. His thesis can be found in [52].
The upcoming section explains the motivation behind this research which is followed by a short description of our contributions. Subsequently, we review the literature on planarity based algorithms for drawing and minor embedding graphs. The last section provides an outline of the next chapters.

1.1 Motivation

Various real life problems, such as scheduling, financial analysis, machine learning, and so forth can be modeled as non-deterministic polynomial time (NP) problems. The classic

computing is trying hard to keep up with the complexity of recent problems and the ongoing growth of data. In the past decades, scientists have begun investigating alternative approaches toward computing. Among them, *quantum computing*, processing data based on quantum mechanics, has been through a great theoretical and practical progress in the past years. The quantum computers have the potential to solve problems which would take years to solve, much more faster than the classic computers.

Consider a classic computer with a memory of n binary bits. A quantum computer has a sequence of *qubits*, each equivalent to a bit with quantum information, which can be 0, 1, or a superposition of these two states which makes this computer similar to probabilistic computers. The sum over all states should be 1 for both types of computers. On the other hand, the difference between two consecutive states conveys a message, if it occurs for a qubit, whereas this difference is meaningless in a probabilistic computer.

While the idea of quantum computers is highly intriguing, they are still going through improvements. In 2011, D-Wave Systems introduced a commercially available device, designed to exploit quantum computations. This device is a step forward which requires minor embedding algorithms to fulfill its potentials.

We can obtain a minor of a given graph G by eliminating edges and vertices, and contracting edges. Graph minors and their interesting properties were first noticed and explored by Wagner [54], and thoroughly discussed by Robertson and Seymour [39]. Graph minors have many applications, but there may have been few concrete applications of the problem: Given a graph G, find a minor embedding of G in H where H is from some special family. This lack of applications might have been a disadvantage in attracting researchers in the past years. But with emergence of the D-Wave machine, minor embedding algorithms are essential to take advantage of its vast computing powers. Given a graph G representing the problem we are trying to solve, G needs to be minor embedded on the graph structure modeling D-Wave hardware.

Before explaining how that mapping works, consider a general optimization problem, in which there are some constraints and some variables. The goal is to either minimize or maximize a certain value that can be achieved by changing the variables. On the other hand, the constraints limit the changes we are allowed to make to the variables. Now the D-Wave hardware solves an optimization problem by first modeling it as quadratic optimization, called *Ising models*, with real coefficients, and binary variables representing qubits [8]. A quadratic optimization refers to an optimization problem with a quadratic function of several variables where the constraints on these variables are linear. Assuming $h \in \mathbb{R}^n$ and $J \in \mathbb{R}^{n \times n}$, then the goal is to minimize the following energy function:

$$E(z) = h^T z + z^T J z$$

where $z \in [-1, 1]^n$. The system starts from an initial state, and undergoes various states until it reaches the final state, which is the solution to the input problem. Being restricted to Ising models, the D-Wave device is not a universal quantum computer, but it can compete with the best of the existing classical algorithms [8]. There is also further limitation on Ising models due to engineering conditions. Processors in the D-Wave machine can vary in the number of functional qubits: some of the low quality qubits are not engaged in the computing process, and hence, we refer to them as *missing qubits*. Now consider an Ising model (h, J) as a graph where variables $z_1, z_2, ..., z_n$ represent the vertices, and for any $J_{ij} \neq 0$, z_i and z_j are adjacent. We need to map this graph to the D-Wave hardware. Variable z_i, referred to as *logical qubit*, is represented by a connected subgraph of one or more qubits in the hardware, referred to as *physical qubits*. For instance, suppose z_1 is mapped to q_1, q_2, and q_3. These three physical qubits should be all connected, setting $J_{i,j} = -\inf$ *such that* $i \neq j$ and $i, j \leq 3$ ($J_{i,j} = -1$ in practice), so that they take the same value in the hardware Ising model. This way, for each pair of z_i and z_j with non-zero J_{ij}, the corresponding subgraphs of (physical) qubits should be connected. Therefore, the only way a problem can be solved by the D-Wave hardware is that its corresponding Ising model graph is a minor of the hardware's graph. In other words, D-Wave's machine can solve a problem instance if we can answer the following question, "given an input graph H and a *target* graph G, can we find a minor of G isomorphic to H?". The desired output would be a set S of subgraphs of G representing the vertices of H such that the subgraphs corresponding to adjacent vertices of H are connected in G. This problem is NP-hard. An algorithm which solves this problem can be viewed as a translator that enables us to solve any NP problems.

Assuming H is fixed, there is a polynomial time algorithm that decides whether G contains a minor of H [32]. This assumption corresponds to fixing the problem instance we are trying to solve, hence, the mentioned algorithm is not of use here. The currently best algorithm, for the NP-hard version of the problem in which G and H are both input, has exponential running time in branchwidth of G [2]. Consider the graph representing the D-Wave hardware which is called Chimera$^{\text{TM}}$. This graph has $8K^2$ vertices, made by replacing each node in a $K \times K$ grid graph by a complete bipartite graph $K_{4,4}$. These $K_{4,4}$s are connected to each other so that the inner vertices have degree 6, and the vertices on the border have degree 5 [16, 37]. Now, Chimera is a sparse graph which makes it possible to find shortest paths in linear time. On the other hand, its large treewidth makes it unsuitable for exact algorithms. Based on these attributes of Chimera, and its large automorphism group, we should aim for heuristic algorithms [8].

It is noteworthy that even though the Chimera is the ultimate target graph, it is neither considered as a fixed input for the minor embedding problem, nor a concept it can be based on. Due to the inconsistent pattern of the hardware, described earlier, and possible chances of different structures for future versions, the practical minor embedding algorithm should

Figure 1.1: A Chimera graph [37]

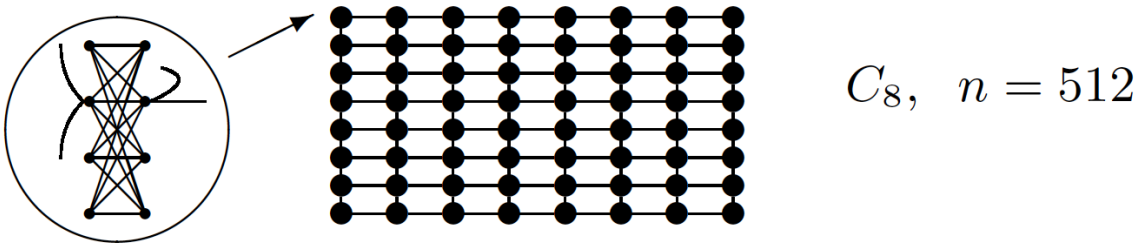
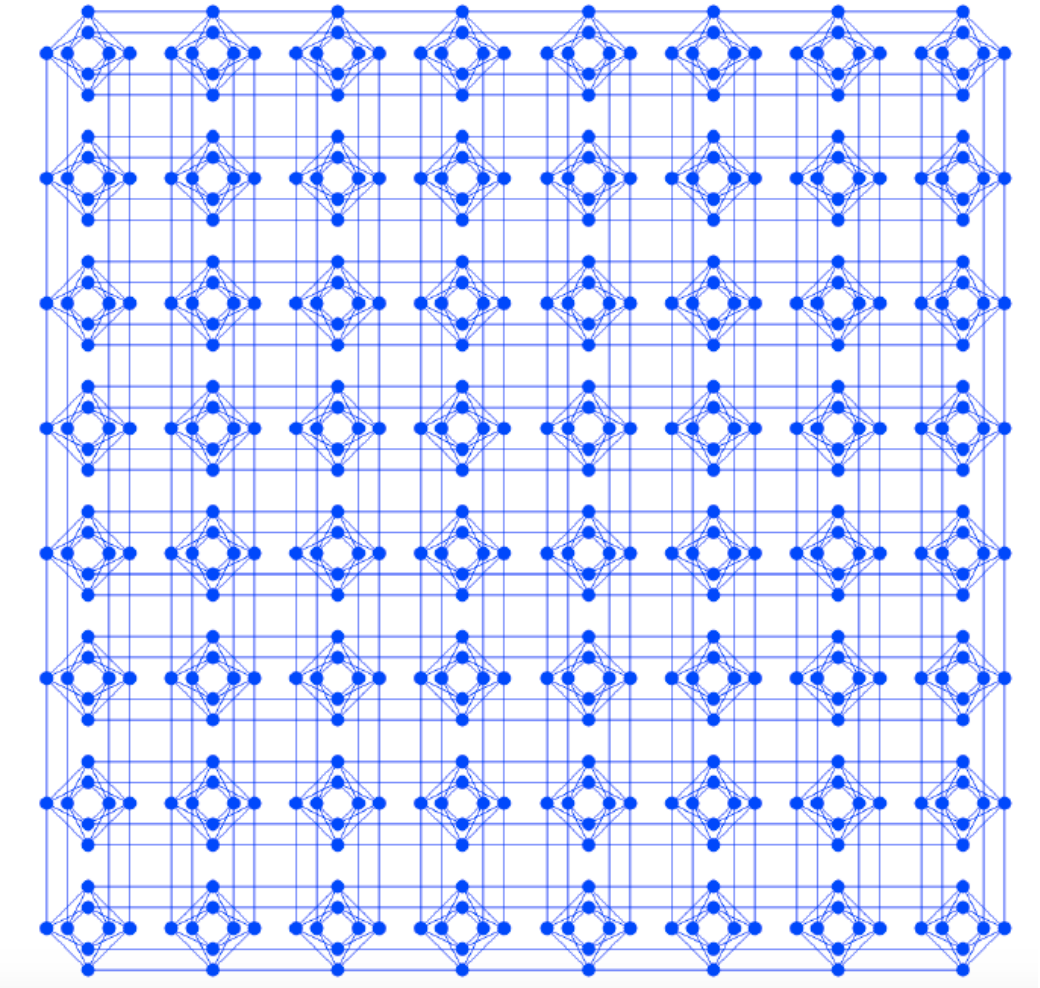


Figure 1.2: Another view of a Chimera graph [8]



not be dependent on the target graph's exact structure.

In this thesis, we have explored a restricted version of the original problem. If H represents the question we are trying to solve with a quantum computer, then we have set a condition on the input graph H to be planar. On the other hand, since the target graph G which H

will be mapped onto is the Chimera^{TM}graph and can be considered as several grid graphs on top of each other, we redefine G to be a grid graph. The problem is still NP-hard, and can be formally stated as follows: "Given an input planar graph H and a target grid graph G, is there a minor of G isomorphic to H?". As before, the desired output is a set of vertex models representing the vertices in H in G such that adjacent vertices in H correspond to distinct vertex models connected by an edge in G. This work can be extended in future to include non-planar input graphs H by breaking them into planar components and minor embedding those in different layers of Chimera. Then, the remaining edges can connect those components together.

1.2 Contributions

Based on the recent progress in quantum physics and quantum processors, there is a need for efficient minor embedding algorithms to bridge the gap between the existing hardware and the solutions we are looking for. As mentioned in [57], to realize the full potential of the adiabatic quantum computers we need to design application-specific algorithms. Here, we have limited the problem to planar input graphs and grid graphs as targets. With the restriction on planarity, we can employ existing planarity algorithms.

We present two minor embedding approaches for planar graphs. The first algorithm is inspired by the Chrobak and Payne [14] straight line drawing algorithm. Our algorithm places the vertices on the grid according to the output of the straight line drawing algorithm. Then, based on the canonical ordering, in every step of the algorithm, edges incident to each vertex and its neighbors below are found by a path finding algorithm and embedded in the grid. The output minor embedding consists of long paths, and hence, is inefficient in terms of space. The process of forming the vertex models is dependent on the grid structure which limits its application to a specific type of target graphs (grids). On the other hand, this process and the coordinates of vertices provided invaluable insight and perspective for us which helped toward developing the next algorithm. Given the triangulated structure of the straight line drawing which is preserved during the minor embedding process, optimizing the output or modifying the vertex placements is not trivial. Nevertheless, an optimization technique is presented that attempts to reduce the size of embedding after a feasible solution is achieved. The core idea of this optimization is simple, but it can effectively improve the worst cases when combined with a good heuristic. Given the challenges we faced with this algorithm, we chose to pursue another approach.

The second algorithm is founded on visibility representation of the input graph. Given the visibility representation, it can minor embed all planar graphs successfully and efficiently. The major component influencing the minor embedding quality is the employed technique for managing the vertex models. By deferring the decisions in the process, the algorithm can make better and more informed choices. This technique is not dependent on the target

graph, and hence, can be deployed in other minor embedding algorithms as well. We also present an optimization method, able to shrink the minor embedding successfully. The range of improvement by the optimization method depends on the structure of the input graph. The second algorithm addresses the shortcomings of the first algorithm such as dependency on the target grid graph, lack of success in minor embedding all planar graphs, and inefficient outputs.
The experiments were mostly conducted on the second algorithm and an existing algorithm in [8]. These two algorithms were evaluated on graphs of different families and sizes. The results are presented here and demonstrate the consistent success of the second algorithm in minor embedding, whereas the existing algorithm fails on most cases as the size or density grows.

1.3 Related Work

Minor embedding for planar graphs is a kind of graph drawing, and graph drawings can be evaluated in different categories based on diverse factors including size, aesthetics, readability, angular resolution, and so forth. There are different types of graph drawing for different applications:

- In *straight line drawing*, edges are represented by straight-line segments.
- An *orthogonal drawing* is a drawing comprised of only vertical and horizontal line segments as edges.
- In *polyline drawings*, each edge is represented by a set of line segments for which there are no restrictions on being straight, vertical, or horizontal. The drawing may include *bend points*, meaning the edges can bend with angles between 0° and 180°.

When vertices and bend points are mapped to integer coordinates, then a *grid drawing* is produced. On this grid, the area of the smallest rectangle that contains the drawing and is aligned to the axes, is *the area of grid drawing.*
Following the contributions of Wagner [54] in proving the existence of straight line drawings for planar graphs, Tutte [53] was the first to introduce an algorithm for constructing such drawing. The need for precise arithmetic operations made that algorithm and the ones presented by Chiba *et. al.* [10] and [11] too complex. Rosenstiehl and Tarjan [40] proposed the idea of drawing in a grid to avoid the complicated arithmetic operations. On the other hand, considering a grid as the output framework makes it possible to measure the area used for the drawing.
De Fraysseix *et. al.* [22] was the first to present a polynomial time algorithm for drawing in the grid based on an incremental approach using a vertex ordering, called the *canonical ordering*, and without the previous high-precision operations. The algorithm was defined

Table 1.1: Straight Line Drawing Algorithms

Authors	Time	Area of Grid Drawing
de Fraysseix, Pach, and Pollack[22]	$O(nlogn)$	$(2n-4)(n-2)$
Chrobak and Payne[14]	$O(n)$	$(2n-4)(n-2)$
Schynder [41]	$O(n)$	$(n-2)(n-2)$
Kant[31]	$O(n)$	$(2n-4)(n-2)$

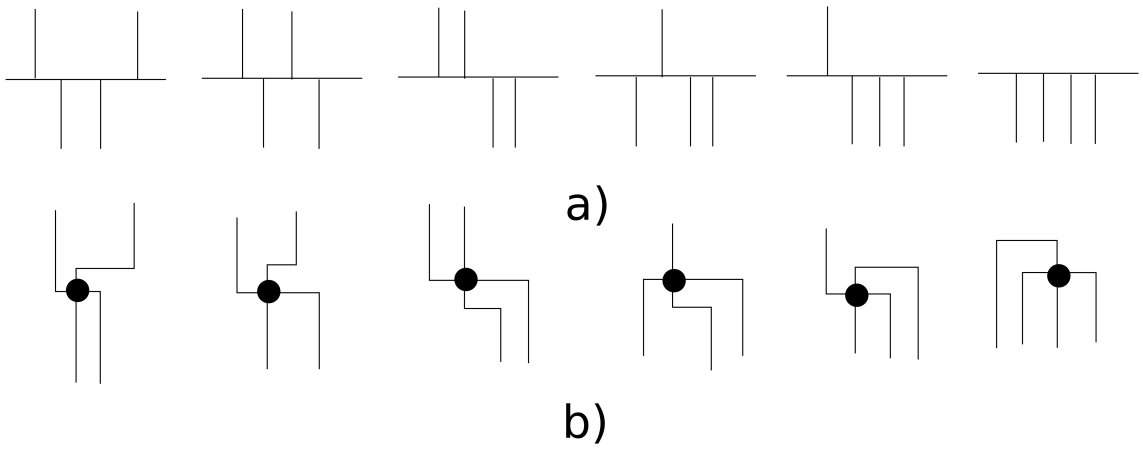
for maximal planar graphs, but was later improved by Kant [31] to process tri-connected graphs. Later, Gutwenger and Mutzel [26] made it possible to include bi-connected graphs. In 1995, Chrobak and Payne [14] presented a linear time algorithm for drawing a planar graph of n vertices in a grid. The size of the grid enclosing the drawing is $(2n-4) \times (n-2)$ for both the algorithms in [22] and [14], but the running time has improved from $O(nlogn)$ to $O(n)$. The latter algorithm employs two main components: the canonical ordering and *shifting sets*. The shifting sets are more of an algorithmic or technical component to enable vertices to move dynamically on the grid during the construction of the drawing. Table 1.1 provides a summary of running times and size of the grid enclosing the drawing for some of the straight line drawing algorithms mentioned here.

Later, Schnyder [41] presented a more intricate algorithm that also works on triangulated graphs in the same way as [14], but draws the graph in a smaller grid of $(n-2) \times (n-2)$. His algorithm uses relative positions of vertices, and then, three barycentric coordinates represent each vertex's actual position.

Different approaches have been proposed for orthogonal drawings. The key features of 90 degree angle between edges incident to one vertex, and the intersection-free edges are interesting for various applications such as VLSI circuit design. The earlier orthogonal drawing methods were focused on decreasing the number of bends but did this by sacrificing size and efficiency. Tamassia and Tollis [50] have proved the existence of orthogonal drawings for 4-planar graphs (planar graphs with maximum degree 4); however, recent improvements such as rectangular regions have addressed the issue of limited vertex degree. Figure 1.3 depicts six possible cases of embedding vertices of degree 4 in an orthogonal drawing. Tamassia [48] also proved an orthogonal drawing of minimum number of bends from a given 4-planar graph can be obtained using a network flow algorithm. The introduced algorithm had a time complexity of $O(n^2 \log n)$, but was improved by Garg and Tamassia [23] to $O(n^{\frac{7}{4}}\sqrt{\log n})$ which recently, was reduced to $O(n^{\frac{3}{2}} \log n)$ by Cornelsen and Karrenbauer [15] .

Later on, visibility representations revolutionized these algorithms by decreasing the running time down to linear while also reducing the size. The first algorithms to compute a visibility representation in linear time were introduced in [50, 40]. Boyer introduced a more efficient method of computing a visibility representation in [6] which we have employed in

Figure 1.3: a) Six possible cases for a vertex of degree 4 b) Their corresponding orthogonal drawing



our experiments.

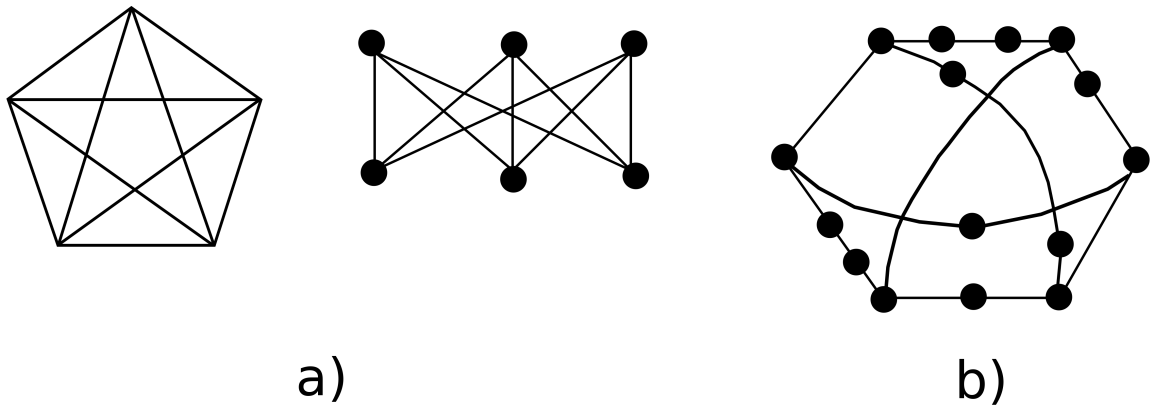
In orthogonal drawing, we are restricted to graphs of maximum degree four. This limitation can be addressed in polyline drawings, where the focus is mainly on angles between edges not degrees of vertices. The most common components among these drawing methods are exit port assignments and some sort of vertex ordering. As an example, Duncan and Kobouro [20] presented a one-bend algorithm, meaning each edge is allowed to bend at most once, by employing the canonical ordering and shifting sets. The vertices and bend points are on a 2D grid with each dimension linear in the number of vertices.

Regarding the number of bends, network flow algorithms can guarantee the lowest number of bends in a very small area by using up almost quadratic time [20].

As for polyline drawing, Gutwenger and Mutzel [26] introduced an algorithm, called the *mixed-model algorithm*, mainly for bi-connected graphs. Their proposed ordering assigns two points to each edge, an *inpoint* and an *outpoint*, *e.g.*, for $e = (v, w) : e_{in} = (x_{in}, y_{in}) \quad e_{out} = (x_{out}, y_{out})$. Using this method, the line connecting v and w is actually made of the following smaller components: from v to e_{out}, from e_{out} to (x_{out}, y_{in}) vertically, from (x_{out}, y_{in}) to e_{in} horizontally, and then finally to w. This method generates more aesthetical drawings.

All the mentioned drawing algorithms admit only planar graphs, and report failure if the planarity condition is not met. Some algorithms actually output the subgraphs obstructing planarity. These algorithms are sometimes referred to as *planarity testing* algorithms. According to West [55], a graph is planar if it admits a planar drawing. On the other hand, if it has a K_5, or $K_{3,3}$ minor, it is non-planar, see Figure 1.4. A subdivision of graph H can be obtained by subdividing its edges successively. The edges have been replaced by disjoint paths, see definition 2.1.3. Therefore, an arbitrary graph either admits a planar drawing, or one of its Kuratowski subgraphs are reported by the planarity testing algorithm.

Figure 1.4: a) Kuratowski subgraphs (K_5 and $K_{3,3}$) b) A subdivision of $K_{3,3}$



Hopcroft and Tarjan [29] introduced the first linear time planarity testing algorithm in 1974. The linear algorithm runs a depth-first-search on the input graph, and then assigns weights to edges incident to each vertex. Then, it runs another depth-first-search on the weighted graph to generate an embedding. The algorithm, sometimes referred to as *path addition*, is complex, and there are a few expository articles such as [56] offering alternative explanations.

Another approach toward planarity testing is the use of *st-numbering*. Given a graph $G = (V, E)$ such that $|V| = n$ and $s, t \in V$, an *st-numbering* of G is a function $\pi : V \to \{1, ..., n\}$ where $\pi(s) = 1$, $\pi(t) = n$, and $\forall v \neq s, t \in V : \pi(v) = j$, $1 < j < n$ such that v is adjacent to two other vertices numbered $j + 1$ and $j - 1$. The first algorithm using this approach was introduced by Lempel *et. al.* [35] and runs quadratic in time. The following contributions lead to linear algorithms based on s-t numbering. Even and Tarjan [21] presented a linear time algorithm for obtaining an st-numbering, and Booth and Lueker [3] introduced a planarity testing algorithm based on PQ-trees which keep required information for flipping and altering vertices before and after embedding. Since the algorithm verifies planarity by checking whether adding a new vertex to the current partial embedding violates planarity or not, it is referred to as the *vertex addition* algorithm.

Another interesting data structure used in planar embedding algorithm is PC-tree which is rather generalized version of the PQ-trees. This data structure enabled Shih and Hsu [43, 44] to introduce the first vertex addition planarity algorithm based on the bottom-up view of depth-first-search. The algorithm tests for planarity conditions on P-nodes and C-nodes instead of processing the complex PQ-tree templates.

Depth-first-search and its "backtracking" attribute can provide valuable information regarding a graph's structure. Tarjan [51] originally discovered this concept and presented several linear time graph algorithms using it. Later, Boyer and Myrvold [5] proved the planarity

testing algorithm from [3] can proceed using the information regarding cut vertices and bi-connected components, extracted by depth-first-search, rather than the PQ-trees.
Another contribution of Boyer and Myrvold is the introduction of the linear *edge addition* technique in [7], inspired by PQ-tree implementation of [3] of the vertex addition algorithm [35]. This algorithm considers an edge as a unit of addition and tests planarity conditions by making localized decisions. It produces a planar embedding if the input graph is planar; otherwise, it outputs a Kuratowski subgraph. According to [7], some of the presented techniques can be employed to optimize PC-trees.
The graph minor theorem, presented by Robertson and Seymour [39], is one of the most interesting and complicated concepts in graph theory. The recent employment of this theorem in quantum computers, invented by D-Wave Systems, has attracted a new wave of researchers. The quantum computer is expected to solve NP-hard problems in seconds in comparison to years for classic computers. In this era, this processor can be the solution to Big Data. Companies like Google and NASA have already started taking advantage of this powerful processor to solve problems in machine learning, artificial intelligence, astronomy, and so forth.
Choi [12] presented how an NP-hard quadratic optimization problem represented as a graph G can be reduced to finding a minor of G in the hardware graph U of a quantum computer. In intuitive terms, D-Wave's processor maps an optimization problem to a landscape, then searches for the point with the minimum value. This process consists of two steps: finding the minor embedding, and setting the parameters. The latter is the subject Choi explored in [12]. Based on the mentioned reduction, the quantum computer can solve a problem if its representing graph can be minor embedded in the hardware graph. Therefore, the hardware graph should be designed in a way that given the physical limitations already known to us, it still includes minors of various families of sparse graphs. Choi studied the problem of designing such a hardware graph in [13], and it is still an open problem. The current hardware suffers from missing qubits. This requires the minor embedding algorithms to be able to produce feasible solutions without using these specific qubits. For instance, Klymko *et. al.* presented two algorithms in [34] that are resilient to hardware's missing qubits. The first algorithm attempts to minor embed the input graph, and if it fails, it tries again using a smaller set of functional qubits. The second algorithm uses matching within qubits on the diagonals to find a minor embedding with the minimum number of logical qubits that their representations in the hardware include missing (physical) qubits.
In 2014, Cai *et. al.* [8] introduced the first heuristic probabilistic approach with running time $O(n.n'.m(m' + n'logn'))$ where the number of vertices and edges of H are represented by n and m, and for G by n' and m', respectively. Given graphs H and G as inputs, the algorithm either succeeds in minor embedding H in G, or reports a failure. While independent of the input graphs' structures, it requires H to be much smaller than G [8]. All in all, this randomized algorithm, implemented on D-Wave's machine, minor embeds large sparse

graphs successfully.

In 2016, Zaribafiyan *et. al.* [57] introduced another heuristic algorithm that explores the structure of both input and target graphs. They formulate the problem instances as *quadratic unconstrained binary optimization* (QUBO) problems, and then they represent them as Cartesian products of two complete graphs. According to [57], there is a regularity in QUBO problem formulation of most families of NP-hard optimization problems. Hence, in each family, we can reuse the embedding of one graph for the rest of them. By Cartesian product of two graphs C and G we mean the graph $C\square G$ with the vertex set $V(C) \times V(G)$ and (v_1, v_2) and (u_1, u_2) are adjacent if i)$v_2 = u_2$ and u_1 is adjacent to v_1 in C, or ii) $v_1 = u_1$ and u_2 is adjacent to v_2 in G [55].

Considering the problem of Cartesian product of complete graphs (CPCG), their embedding strategy divides the minor embedding problem into two subproblems: first, choosing an appropriate complete graph K_m (in $K_m\square K_n$), and second, placing n copies of K_m and then connecting them together. The introduced algorithm was compared to the heuristic algorithm by Cai *et. al.* [8] and also another algorithm introduced by Boothby *et. al.* [4] specialized to minor embed cliques in Chimera. Given a fixed size ideal Chimera as target graph and limited running time, the Cartesian based product outperformed the other two. The required gap between the sizes of input and target graphs for Cai's algorithm in [8] puts it at a disadvantage in compare to other algorithms. Here, given a hardware graph $C_{N,N,4}$, the algorithm introduced by Zaribafiyan *et. al.* [57] can embed graphs up to $K_8 \times K_{N-1}$. This size goes than to $K_8 \times K_{N/2}$ for Boothby's algorithm [4]. The performance of Cai's algorithm [8] is not predictable due its probabilistic behavior, but in these experiments stands between the two aforementioned algorithms.

1.4 Outline of the Thesis

The next chapter reviews essential concepts of this thesis. Subsequently, Chapter 3 introduces the minor embedding algorithm based on straight line drawing and explains the challenges we face in minor embedding a planar graph on a grid. Then, Chapter 4 presents a minor embedding algorithm based on visibility representation, along with an efficient technique to better handle and optimize the output. Chapter 5 discusses the conducted experiments to evaluate the performance of the introduced and existing algorithms. All is concluded in Chapter 6, and a few ideas are suggested for future work.

Chapter 2

Review of Basic Concepts

This chapter provides a brief overview of the fundamental concepts applied in our algorithms. After defining a few basic terms, each section explores a different concept in details.

Definition 2.0.1. A graph G is defined with a set of vertices V and a set of edges E such that $E \subseteq V^2$, meaning each edge $e \in E$ is a 2-element subset of V [18].

Two vertices are *adjacent*, or *neighbors*, if they are the endpoints of an edge. An edge is a *loop* if its endpoints are identical. A graph G is *simple* if it has no loops.
One way to represent a graph is by using its *adjacency matrix*. Given simple graph $G = (V, E)$ such that $|V| = n$, the adjacency matrix of G, written $A(G)$, is an n-by-n boolean matrix which the entry $a_{i,j}$ equal 1 if there is an edge between v_i and $v_j \in V$. The *degree* of a vertex v is the number of edges incident to it.
A graph $G = (V, E)$ is *directed* if its edges have directions, otherwise it is *undirected*.

Definition 2.0.2. A *cut vertex* of a graph G is a vertex v such that removal of v from G increases the number of its connected components. A *subgraph* $G' = (V', E')$ of $G = (V, E)$ is a graph for which $V' \subseteq V$ and $E' \subseteq E$. G' is an *induced subgraph* if $E' = \{(u, v) \in E | u, v \in V'\}$.

A simple graph G with n vertices is a *clique* or a *complete graph*, written K_n, if and only if all of its vertices are of degree $n - 1$. Consider a graph G composed of two disjoint sets of vertices V and U such that no two vertices in one set are adjacent. If every pair of vertices (v, u) such that $v \in V, u \in U$ are neighbors, then G is a *complete bipartite* graph [55].

2.1 Planarity

Definition 2.1.1. (Drawing): Consider a function f that maps each vertex $v \in V$ to a distinct point $f(v)$ on the plane $\mathbb{R}^2$, and maps each edge $e = (u, v) \in E$ to one and only

one curve in the plane with $f(u)$ and $f(v)$ as endpoints. This function is a *drawing* of G. If $e \cap e' = \emptyset$ but $f(e) \cap f(e') \neq \emptyset$ then a point p in $f(e) \cap f(e')$ is a *crossing* [55].

Definition 2.1.2. (Planar drawing): A drawing is *planar* if no two images of the edges intersect, except possibly at their endpoints. A graph that admits a planar drawing is *planar*. A *planar embedding* is a planar drawing, where the drawing preserves the adjacency and non-adjacency of vertices, with a certain clockwise ordering for edges incident to all vertices [55].

Theorem 2.1.1. (Euler's Theorem): Let $G = (V, E)$ be a simple planar graph such that $|V| = n$ and $n \geq 3$. Then, $|E| \leq 3n - 6$.

Theorem 2.1.1 shows that the so-called *Kuratowski subgraphs*, $K_{3,3}$ and K_5, are nonplanar.

Definition 2.1.3. Given a graph $G = (V, E)$, if we replace each edge $(u, v) \in E$ with disjoint paths between its endpoints u and v, the new graph is an *G-subdivision* or a subdivision of G [18].

Theorem 2.1.2. (Kuratowski Theorem): A graph is planar if and only if it does not contain a subdivision of $K_{3,3}$ and K_5.

A proof of this theorem can be found in [55].

Definition 2.1.4. (Region): Consider a set $U \subseteq \mathbb{R}^2$. U is an *open set* if, for any point $x \in U$, there exists a real number $\epsilon > 0$ such every point $y \in \mathbb{R}^2$ whose Euclidean distance from x is smaller than ϵ is also in U.
A *region* in the plane is an open set U such that for every pair of $(u, v) \in U$, U contains a curve with u and v as its endpoints. The *faces* of an embedding of a planar graph are the maximal regions of the plane that do not contain any point of the embedding [55].

A planar embedding of a planar graph can have many faces, but it can only have one unbounded face, called the *outer face*, assuming the embedding of the graph is bounded (meaning a sufficiently large circle C can enclose it). Therefore, the face containing $\mathbb{R}^2$ minus the surface of C is the outer face. The other faces are called *inner faces*.

Definition 2.1.5. (Isomorphism): Assume $G = (V, E)$ and $G' = (V', E')$ are two graphs. If and only if there exists a bijection $\phi : V \to V'$ such that $\forall u, v \in V$, if $(u, v) \in E$ then $(\phi(u), \phi(v)) \in E'$, then ϕ is called an *isomorphism*, and G and G' are *isomorphic*, written as $G \simeq G'$.

A simple graph G is *k-connected* ($k > 0$) if after removing any $t < k$ vertices, it is still connected. In the special case of $k = 2$, the k-connected graph is called *bi-connected*.
A *maximal planar* graph is a graph that will be non-planar if we add one more edge to it.
A *k-planar graph* is a planar graph whose maximal degree is k.

2.2 Grid Graphs

In graph theory, we can generate new graphs by applying a product operation on two graphs G and H. The product operation is called the *Cartesian product*, and written as $G\square H$. The vertex set of $G\square H$ is $V(G) \times V(H)$, the Cartesian product of the vertex sets of G and H, and its edge set is formed in the following way: (x, y) and (x', y') are adjacent in $G\square H$ if and only if one of these two conditions hold [30]:

- $x = x'$, and y and y' are adjacent in H
- $y = y'$, and x and x' are adjacent in G.

The edge set can also be defined as the following set:

$$E(G\square H) = (E(G) \times V(H)) \cup (V(G) \times E(H))$$

Consider a tree of n vertices such that 2 vertices are of degree 1, and the other $n-2$ are of degree 2. Such a graph is called a *path graph*, written as P_n, and can be drawn on a straight line on the plane. Now let us define a *grid graph* [25].

Definition 2.2.1. A $m \times n$ grid graph, or a *complete grid graph*, is the Cartesian product of two path graphs $P_m\square P_n$ [30].

A $m \times n$ grid is a graph on vertices $\{v_1, ..., v_{mn}\}$ with edge set $\{(u, v)(u', v') : |u - u'| + |v - v'| = 1\}$ [18].
Grid graphs have unique characteristics. For instance, every planar graph is a minor of some grid [18]. Another example is their particular planar representations. The $m \times n$ grid has a specific planar representation, consisting of perpendicular straight lines (m rows and n columns), see Figure 2.1. If the planar representation can be viewed as a set of square tiles, then it is called a *square grid*.

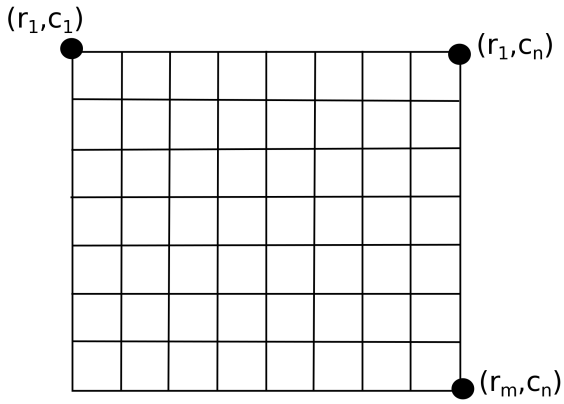
2.3 Graph Minors

Given a graph $G = (V, E)$, we can perform three types of operations: *edge deletion*, *vertex deletion*, and *edge contraction*. Edge deletion simply refers to removing an edge from E. Vertex deletion refers to the removal of a vertex v from V and all edges incident to v from E. Edge contraction is defined below.

Definition 2.3.1. Let $e = (u, v)$ be an edge in a graph $G = (V, E)$. If we *contract* e, u and v will be substituted by a new vertex v_e, and we will have a new graph $G' = (V', E')$ in which v_e is adjacent to all the vertices previously adjacent to u and v. Formally,

- $V' = (V - \{u, v\}) \cup v_e$ *s. t.* $v_e \notin V \cup E$, and

Figure 2.1: Planar representation of a grid graph $G_{m,n}$ generated by Cartesian product of two path graphs P_m and P_n



- $E' = \{(x, y) \in E | \{x, y\} \cap \{u, v\} = \emptyset\} \cup \{(v_e, x) | (u, x) \in E - \{e\} \text{ or } (v, x) \in E - \{e\}\}$ [18].

Based on these operations, we define graph minors.

Definition 2.3.2. G' is a minor of G, written $G' \preceq G$, if it can be obtained from G by means of a sequence of vertex and edge deletions, as well as edge contractions [18].

Any subgraph of G, including G itself, is a minor.
Informally, based on the structure we preserve, we can *embed* a graph in another graph in different ways. An embedding of H in G is an one-to-one map $\phi : V(G) \to V(H)$. If ϕ preserves the adjacency of vertices, then H is embedded as a subgraph. If ϕ preserves both the adjacency and non-adjacency of vertices, then H is embedded as an induced subgraph. Further, if ϕ maps vertices of H to disjoint connected vertex sets in G, and in addition maps the edges (u, v) to independent paths in G such that $\phi(u)$ and $\phi(v)$ are its ends, then H is embedded as a minor.

2.4 Path Finding Algorithms: Dijkstra and A*

Consider an arbitrary graph $G = (V, E)$ with non-negative weights (distances) assigned to all edges, and the problem of finding the shortest path between two given nodes (vertices) s and g . Let $d(s, g)$ be the length of the shortest path from s to g.
The *Dijkstra's algorithm* finds the shortest-path distances of s to all other nodes in the graph, including g as follows. The algorithm is initialized by setting $d(s, u)$ to zero for $u = s$ and to inf for all other nodes. Then, it marks s as the *current* node, and all other vertices as *unvisited*. At each step, the algorithm evaluates the costs of the shortest paths between the current node and its unvisited neighbors, and updates a tentative distance if

the new value is smaller. After processing all the neighbors, the current node is marked *visited* and will never be processed again. The algorithm then moves on to the neighbor with minimum cost, unless one of the following stopping criteria has been met: there is no unvisited node left, we had set a goal, and it has been already processed, or the smallest cost of getting to the next unvisited node is infinity, meaning the part of graph containing s is disconnected from the part containing the unvisited nodes [33].
Dijkstra's correctness can be proved by induction and its running time is $O(|E|+|V|log(|V|))$ [33].

A-star (A)* is also a path finding algorithm which traverses a graph from a given start node to a given goal. It is considered as an extension of Dijkstra's algorithm. It iteratively evaluates all the shortest paths possible and chooses the one with minimum cost. Its popularity is due to use of heuristics to lead the algorithm toward the cheapest path faster by prioritizing the ones which seem to lead to the desired path.
In any iteration, the path from s to g is comprised of two parts: the part from s to some discovered node u, and the part from u to g. By then, the algorithm has already computed the cost of the shortest $s-u$ path, but not the $u-g$ path. Therefore, the cost of the shortest path at each step is the sum of the actual cost of the $s-u$ path plus an estimation of the cost for the $u-g$ path. This computation is presented in the following function: $f(u) = g(u) + h(u)$, where u is the current node, $g(u)$ the cost of the partial path already chosen, and $h(u)$ estimation of the cost to traverse from u to g based on the heuristic [27]. The criterion for choosing a vertex among undiscovered vertices is similar to Dijkstra: any un-visited vertex minimizing the function f is chosen to be processed. If each node keeps track of its previously chosen neighbor, then the whole path can be retrieved in the end.
Given a graph with finite branching factor and positive weights, the algorithm is optimal and complete if its heuristic is admissible, meaning it does not overestimate the costs. This algorithm is linear in the size the set of vertices in the best case scenario, and in the worst case scenario, it is exponential in l which is the length of the shortest path. Let b be the average number of choices the algorithm can make at each node. Then, in the worst case scenario, meaning the desired shortest path between two points is the last possible choice of the algorithm, the running time will be $O(b^l)$ [28].

2.5 Algorithm D: Existing Heuristic

In 2014, Cai, Macready, and Roy [8] introduced a heuristic algorithm for minor embedding a given graph H in another graph G. The algorithm, referred to as *Algorithm D* henceforth, starts processing the vertices in a random order, and places them close to their already embedded neighbors, if any. It either generates a feasible minor embedding, by which we mean a minor embedding without crossings, or reports a failure.

Definition 2.5.1. A model of $H = (V_H, E_H)$ in $G = (V_G, E_G)$ is a $\phi : V_H \to V_G$ such that

- for $u, v \in V_H$ where $u \neq v$, $\phi(v)$ and $\phi(u)$ are disjoint,
- for each $v \in V_H$, there exists a subtree T of G with $V_T = \phi(v)$,
- and for each edge $(u, v) \in E_H$ incident to v, there is an edge $(x, y) \in E_G$ with $x \in \phi(v)$ and $y \in \phi(u)$.

Algorithm D has two phases. In the first phase, it embeds all the vertices. Sometimes, a part of G is used in more than one vertex model. Such occurrence is considered as a crossing. While crossings are penalized, they are not prohibited. In the next phase, the algorithm attempts to resolve all the crossings by re-embedding the vertices in an attempt to generate a feasible embedding. If the algorithm fails to generate a feasible embedding, it attempts to decrease the total size of vertex models, and failing that, it tries to decrease the size of the largest vertex model.

The final stopping criteria is achieving vertex models of maximum size 1 (smallest vertex models possible), or having no improvement for a certain number of consecutive iterations (set to 10 according to [8]).

Let $\phi(x)$ be the vertex model for vertex $x \in V(H)$. $\phi(x)$ is a desirable vertex model if i) it is representing x exclusively, meaning it does not intersect with any other vertex model, and ii) it is connected to all its neighbors while using as few vertices of the target graph as possible.

The main principles of Algorithm D are explained below.

Placing a vertex close to its already embedded neighbors: Let u be the vertex we are trying to place, and $y_1, y_2, ..., y_k$ are its already embedded neighbors as $\phi(y_1), \phi(y_2), ..., \phi(y_k)$. At this step, the shortest path distance between any unused vertex $g \in V(G)$ and $\phi(y_j)$, $1 \leq j \leq k$, should be computed. Then $c(g, j) = \sum_{t \in \phi(y_j)}^{|\phi(y_j)|} c(g, t)$. The root of the new vertex model $g*$ will be the one with the minimum $\sum_j c(g, j) = \sum_j (\sum_{t \in \phi(y_j)}^{|\phi(y_j)|} c(g, t))$. Then, the union of the shortest paths connecting $g*$ to its neighbors forms the new vertex model $\phi(u)$. The process of computing the distances between every single grid node and all vertices in every vertex models is not very efficient. Therefore, when computing the distance of a vertex g to a vertex model $\phi(y_j)$, we add a dummy vertex $\hat{y_j}$ adjacent to all vertices in $\phi(y_j)$. Then, we compute the cost of the shortest path between g and $\hat{y_j}$.

Representing one vertex exclusively: There are not always unused vertices available to provide the closest location to neighbors of a vertex being processed. Therefore, Algorithm D allows more than one vertex model to contain a certain node of the target graph. On the other hand, to discourage the algorithm from doing so, we assign a penalty to vertices being used more than once:

$$wt(g) = D^{|\{j : g \in \phi(y_j)\}|}$$

where $g \in V(G)$ and D is the diameter of the target graph.
This way of assigning weights to vertices can be an issue for path finding algorithms, due to them being able to process weighted edges. This issue can be resolved by considering a directed graph with the weight of edge equal to the weight of its head.

Assigning shortest paths to vertex models: Upon choosing a vertex $g*$ as the root of the current vertex model $\phi(u)$, along with the shortest path connecting it to its neighbors, we have to decide which parts of the discovered paths should be assigned to $\phi(u)$ or the neighbor it is connecting to $(\phi(y_j))$. Algorithm D settles this decision based on the number of vertex models using that specific node of the target graph. If the node is included in more than one shortest path, emanating from u, then it is assigned to $\phi(u)$. But if it is only included in the shortest path to $\phi(y_j)$, then it is added to $\phi(y_j)$. The reason is if u uses that particular node to connect to more than one of its neighbors, then adding that vertex to $\phi(u)$ instead of more than one vertex models of its neighbors serves in minimizing the size of vertex models. On the other hand, in case that node is only used to connect u and some neighbor y_j, assigning it to $\phi(y_j)$ means giving a chance to $\phi(y_j)$ to use it in future for connecting to neighbors of y_j.

Random choice of roots: To avoid getting stuck in local optima, once in a while, Algorithm D chooses a random node of the target graph as $g*$ with the probability proportional to $\exp(cost(g))$, instead of choosing the one with minimum distance to the neighbors.

Considering the weights based on inclusion: Evaluating the weight of the shortest path between a node g and a vertex model $\phi(y_j)$ depends on whether $g \in \phi(y_j)$ or not. If $g \notin \phi(y_j)$, then there is a path $(v_1 = g, v_2, ..., v_{k-1}, v_k \in \phi(y_j))$ where $v_k \in \phi(y_j)$, and the weight of v_k should not be included in $c(g, j)$. This way, assuming g is chosen as the root, then only $(v_1 = g, v_2, ..., v_{k-1})$ belong to the new vertex model. On the other hand, if $g \in \phi(y_j)$, then the shortest path between the new vertex model and $\phi(y_j)$ would be $(v_k = g)$, and hence, its weight should be included in $c(g, j) = c(v_k, j)$. The node g will be a part of both the new vertex model and $\phi(y_j)$.

To improve the running time, the authors [8] introduced a *localized version* of the algorithm. The most consuming part of Algorithm D is the shortest path computation at each iteration between every node of the target graph and every vertex model. To fix this issue, two variations were proposed:

- *Multisource Dijkstra*: According to [8], the choice of roots at each iteration do not differ from the previous iteration. Therefore, the series of shortest path computations to every vertex model is substituted by one simultaneous computation of shortest path to each vertex model, which is referred to as *multisource Dijkstra*. The Dijkstra's algorithm has been explained in Section 2.4. Here, multiple trees are formed, starting from the sources, and grow every iteration until there is a node $v*$ that has been

reached by all trees. While minimizing the maximum weight of shortest paths, this process is searching only a subset of the entire target graph.

- $A*$: Dijkstra is replace by the A* algorithm, described in Section 2.4. The heuristic is set to the unweighted distance between any two vertices which can be precomputed in advance. The target for vertex y_j is the root of the vertex model $\phi(y_j)$. Hence, there is no target in the first iteration.

The impact of using multisource A* in the second phase of Algorithm D is evident when the choice of the new root from the target graph nodes has failed the condition on reducing the number of represented vertices (or the number of vertex models including a certain node). At this point, Algorithm D will quickly place the new root close to the previous one. But if there was a chance to improve the embedding by choosing a node representing fewer vertices, then the process would be carried out in approximately the same duration.
The other impact is on the success rate of the algorithm which is made worse by using A*. Dijkstra is able to minimize the overlap among vertex models by minimizing the sum of the shortest paths, whereas A* minimizes the maximum shortest path. In other words, Dijkstra considers all the overlaps, not just the worst case, and that is why it is more time consuming than A*.
The running time of the original algorithm is bounded to $2e_H$ times of running Dijkstra with running time $O(e_G + v_G \log v_G)$ for each iteration. In the worst case, it would take $n_H n_G$ iterations to obtain a feasible embedding, coming to a running time of $O(n_G n_H e_H(e_G + v_G \log v_G))$. Algorithm D was studied for grid graphs, random cubic graphs, and complete graphs.
The success rate of Algorithm D is similar to a sigmoid function. All in all, it could be said that the algorithm's behavior is more erratic for complete graphs, which might be due to their complex nature.

Chapter 3

Minor Embedding Algorithm Based on Straight Line Drawing

In this chapter, a minor embedding algorithm is introduced based on a straight line drawing of the input graph, obtained from the algorithm presented in [14]. A straight line drawing is a type of planar embedding which represents edges with non-intersecting straight lines. In this approach, the output of the drawing algorithm helps us determine "where" to embed each vertex so that a path finding algorithm can find a way from each vertex to its neighbors without introducing crossings. Either the path finding algorithm succeeds, or it reports a failure.

We begin this chapter by discussing the straight line drawing algorithm in detail and defining its components. Later, we explain how the minor embedding process is carried out given the output of straight line drawing algorithm. In the last section, we analyze the introduced approach and evaluate its strong and weak points.

3.1 Straight Line Drawing

In 1990, de Fraysseix *et. al.* [22] proposed an incremental algorithm for drawing a graph in a grid based on the canonical ordering with running time $O(n \log n)$. Later, their method was improved and simplified by Chrobak and Payne [14], and its running time reduced to $O(n)$. The drawing algorithm and its proof are both based on the canonical ordering of vertices of a planar graph [17]. The output of this algorithm can be embedded in a grid with dimensions $(2n - 4)$ by $(n - 2)$ [17].

These algorithms are only applicable to maximal planar graphs. Later, this restriction was loosened to some extend to include tri-connected and bi-connected graphs by [26] and [31]. Nevertheless, we chose the Chrobak and Payne algorithm due to the availability of implementation and ease of use. The implementation deployed in this thesis is taken from

[24]. We must carry out the extra step to make the input graph maximal planar, in case it is not. Since there exist linear-time algorithms to add extra edges to a given planar graph to the point of maximal planarity [14], this process will not impact the overall time complexity. On the other hand, the extra line segments, representing the added edges, can be eliminated from the output of the straight line drawing.

Before further discussion of canonical ordering, we repeat the definition of maximal planar graphs given in Section 2.1. A planar graph which loses its planarity if we add one more edge to it, without making changes to the set of vertices, is called a maximal planar or a triangulated graph. At this point, based on Theorem 2.1.1, if the graph has $n \geq 3$ vertices, it also has $3n-6$ edges and $2n-4$ faces, each of which is surrounded by exactly three edges [42].

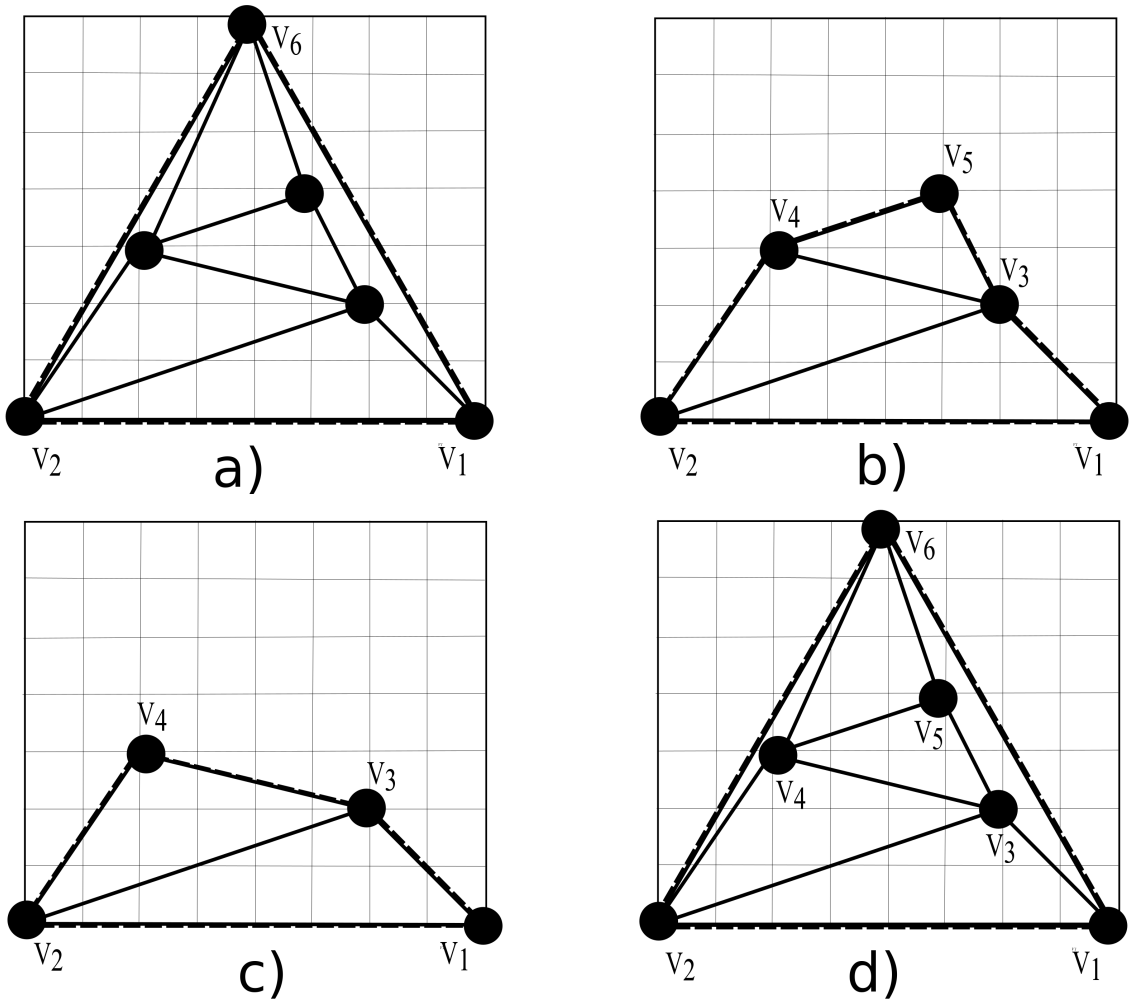
Here, we will briefly explain the straight line drawing algorithm in [14], and then, we provide formal definitions for its vital components. The algorithm uses a certain ordering of vertices V_G, called the *canonical ordering*, and embeds the vertices in that order by connecting them to their already embedded neighbors. Readjusting the positions of a subset of vertices is an essential technique to embed edges as straight line segments. The embedding process starts with the first three vertices in the order v_1, v_2, v_3 taking place on the grid at the following coordinates: $(0,0), (2,0)$, and $(1,1)$. If at the current stage, a vertex v is the next vertex in the sequence to be added to the outer face of the embedded subgraph, its neighbors that have already been processed form a simple path on the cycle bounding the outer face of the embedded subgraph. During the minor embedding process, it is important to keep track of a list of vertices to be relocated if the coordinates of an already embedded vertex are altered. The change of coordinates happens when a vertex needs to move further from its neighbors to preserve the property that edges are straight lines. Any vertex influenced by this movement has a set of dependent vertices, so the algorithm moves all the necessary vertices to right by 1 or 2 points in the grid, and eventually connects the new vertex to each of its neighbors using straight lines. Based on this explanation, the two main components of the algorithm are the vertex ordering, called the canonical ordering and the sets of locationally-dependent vertices, called the *shifting sets*.

The canonical ordering can be defined as follows [14, 9, 20]:

Definition 3.1.1. Assume $G = (V, E)$ with $|V| = n$ is a maximal planar graph, and $\mu = (v_1, v_2, v_3, \ldots, v_n)$ is an ordering of vertices. Let G_i ($1 \leq i \leq n$) be a plane subgraph of G, induced on the vertices $v_1, v_2, \ldots, v_i$, and C_i be the cycle enclosing the outer face of G_i such that $C_i = (v_1 = w_1, w_2, \ldots, w_j = v_2)$. μ is a *canonical ordering* if

- the counter-clock wise order of v_1, v_2, and v_n forms the external face of G
- $\forall i$, $2 < i < n$, G_i is a bi-connected triangulated subgraph of G, and
- $\forall i$, $2 < i < n$, v_i is included in C_i and adjacent to at least one vertex in $G - G_i$.

Figure 3.1: The reverse canonical ordering is generated by eliminating vertices one by one a) The vertex v_6 is about to be removed from the outer cycle of the graph. b) Elimination of the vertex v_5. c) Elimination of the vertex v_4. d) The final ordering of the vertices.



Theorem 3.1.1. Every maximal planar graph has a canonical ordering that can be computed in linear time and space.

Theorem 3.1.1 was proven in [22, 14]. Here is a brief explanation; Consider a triangulated graph G. Assume we are processing G_{i+1} with C_{i+1} as the current outer cycle. If we eliminate a vertex v_{i+1}, adjacent to at most two other vertices on the external face, we will create the new subgraph G_i, and its corresponding external face C_i which includes all the neighbors of v_{i+1}. This repetitive process can produce the reversed canonical ordering in linear time, see Figure 3.1.

Let μ be a canonical ordering. As described before, the drawing starts with v_1, v_2, v_3 forming a triangle, and then iteratively new vertices v_{i+1} are inserted along with the edges connecting them to their neighbors on the external face C_i, forming the new face C_{i+1},.

The neighbors of v_{i+1} can be labeled $w_x, w_{x+1}, ..., w_y$, being a sequence on the external face, where in this sequence, w_x is the leftmost and w_y the rightmost neighbor.
The iterative nature of the algorithm calls for adjustable variables such as dynamic vertex coordinates. Therefore, when inserting a new vertex v_{i+1}, it might be necessary to shift the leftmost and rightmost neighbors and increase the space between them. However, because those vertices are connected to other vertices themselves, we need to track dependent vertices to avoid intersection among edges. This sort of adjustment can be revised based on different applications. The promise of keeping straight line edges results in more space usage, essential to algorithms like Chrobak and Payne's [14] but inessential to minor embedding.

Definition 3.1.2. Given a canonical ordering μ, for each vertex w_k on the external face of C_i, $3 \leq i \leq n$, the *shifting set* $M_i(w_k) \subseteq V$ is defined as follows [49, 20]. $M_3(v_3) = \{v_3\}$, $M_3(v_2) = M_3(v_3) \cup \{v_2\}$, $M_3(v_1) = M_3(v_2) \cup \{v_1\}$. Let w_y and w_x be the rightmost and leftmost neighbors of v_{i+1} such that $3 \leq i < n$. Then the following states hold:

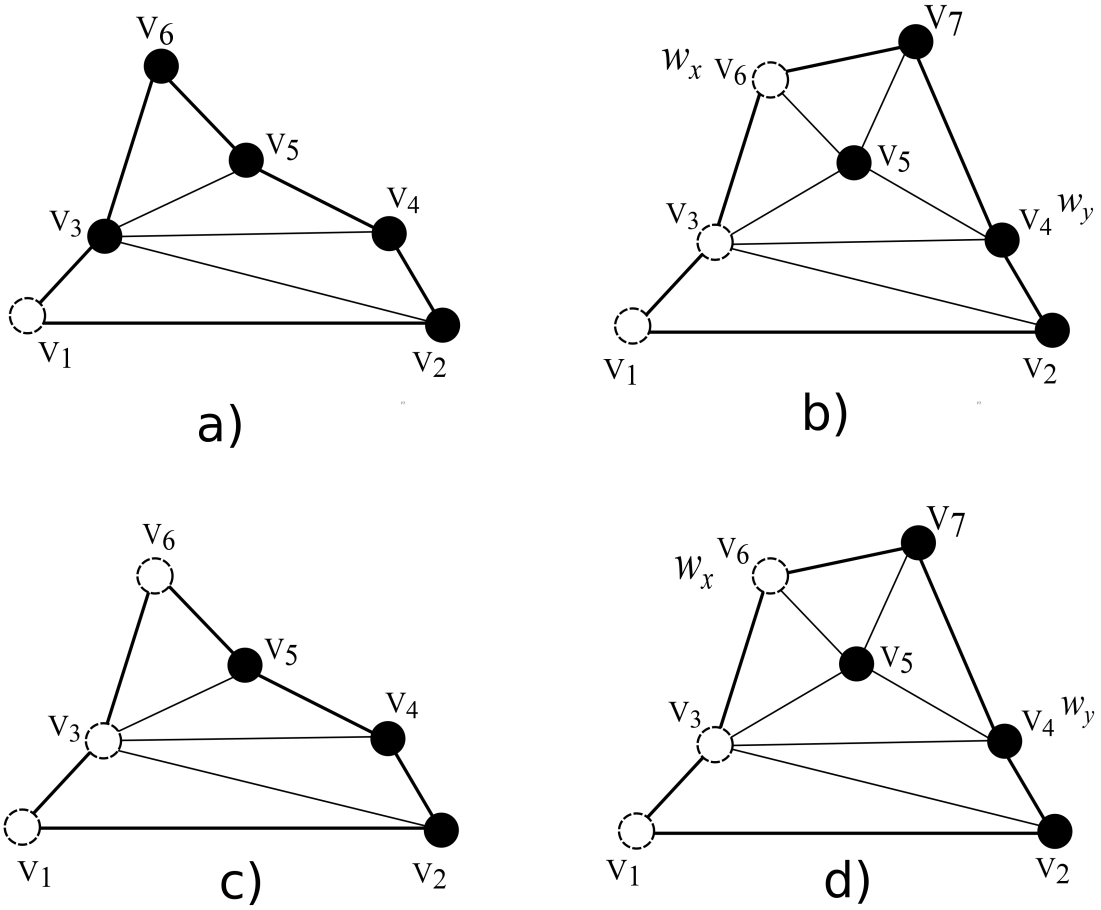
- $\forall k \leq x, M_{i+1}(w_k) = M_i(w_k) \cup \{v_{i+1}\}$,
- $\forall j \geq y, M_{i+1}(w_j) = M_i(w_j)$, and
- $M_{i+1}(v_{i+1}) = M_i(w_{x+1}) \cup \{v_{i+1}\}$.

No shifting sets are defined for $x < k < y$ because those vertices are *covered*, by which we mean they are no longer on the external face. Figure 3.2 illustrates an example of how shifting sets are updated through iterations.
More details regarding the proof of correctness and the implementation of the algorithm can be found in [14]. However, the following points are important to note regarding the output straight line drawing.

- All vertices are embedded on the grid points in the plane and have integer coordinates.
- All edges are embedded as straight lines.
- The first two vertices in the canonical ordering are always connected to each other with a horizontal line of length $(2n - 4)$ units of grid-edge length.
- The last vertex to be embedded has $y = (n - 2)$ which effectively bounds the area of the grid to $(2n - 4)(n - 2)$.

Figure 3.3 depicts a straight line drawing of a graph, known as the *wheel graph* with 5 vertices.
Our approach to the minor embedding problem was to use the coordinates of the vertices in a straight line drawing of the input graph $H = (V_H, E_H)$. Based on these coordinates, we generate new coordinates to make a minor embedding of H in a grid graph $G = (V_G, E_G)$.

Figure 3.2: Construction of the shifting sets as the vertices are embedded in the grid. The highlighted vertices represent the shifting sets. a) The shifting set $M_6(v_3)$. b) The shifting set $M_7(v_3)$. v_7 is inserted and this changes the outer cycle. c) The shifting set $M_6(v_5)$. d) The shifting set $M_7(v_7)$. v_7 is inserted and this changes the outer cycle. $M_7(v_5)$ is the union of $M_6(v_5)$ and v_7 because $w_{x+1} = v_5$ as mentioned in the third state.

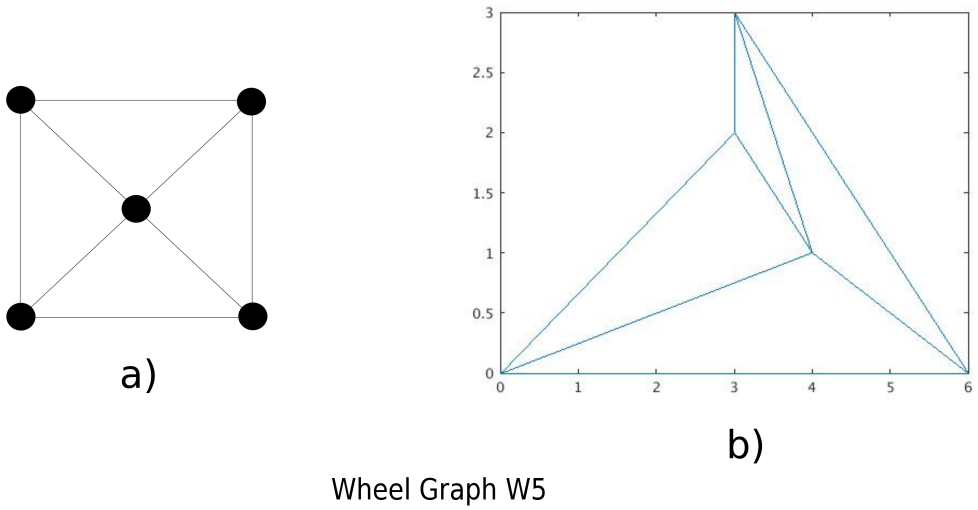


In a nutshell, the root of the vertex model for each vertex $v \in V_H$ is a vertex of G, and each edge $e \in E_H$ is mapped to a path $p \subseteq E_G$. There must exist at least one way of assigning the vertices of G, the ones forming the shortest paths, to vertex models of V_H, in a way that the resulting assignment is a feasible minor embedding of H in G.

The first phase of the minor embedding algorithm, called the *SLM Algorithm* from hereinafter, includes the following linear-time operations: triangulating the graph, obtaining the straight line drawing, and removing the extra edges to retrieve the original set of edges. Linear MATLAB implementations of all these operations can be found in [24] and have been deployed in our experiments.

In the straight line drawing algorithms discussed above, including Chrobak and Payne's [14], the only obligation the grid forces on the algorithm is the integer coordinates for ver-

Figure 3.3: Wheel graph with 5 vertices (W5). a) A planar representation of W5 b) A straight line drawing of W5



tices while the edges are embedded on the plane $\mathbb{R}^2$. But for minor embedding on a grid graph G, in addition to integer coordinates for vertices, the edges are to embed on the grid, meaning for a minor graph H, $E_H \subseteq E_G$. This situation is similar to the conditions for orthogonal drawing algorithms. Each embedded edge on the grid is a path between two vertices $u, v \in V_H$, consisting of some number of horizontal and vertical grid edges. Parts of this path may be used again for connecting one of the two vertices u, v to one or more neighbors. This process will be explained in more details, in the following section.

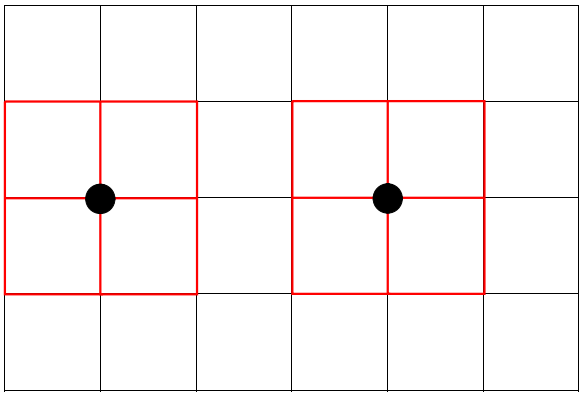
3.2 Minor Embedding Based on a Straight Line Drawing

3.2.1 The minor embedding technique

In our algorithm, the process of minor embedding a graph $H = (V_H, E_H)$ in a grid graph $G = (V_G, E_G)$ consists of the two following phases: placing the root vertices (the roots of vertex models) for each $v \in V_H$ on V_G, and embedding the edges E_H using a path finding algorithm (which includes generating the vertex models). Each phase consists of smaller steps which will be discussed shortly.

As mentioned earlier, the purpose of applying the straight line drawing algorithm is to obtain tentative coordinates for placing vertices on the grid. But using the same exact coordinates of the straight line drawing may cause complications and crossings due to dealing with a

Figure 3.4: Two vertices on the grid, and their perimeters highlighted. The perimeters of vertices with Manhattan distance less than three will overlap.



fixed angle between edges when minor embedding or orthogonal drawing versus straight line drawing. Therefore, these coordinates must be altered to meet the new needs of our application.

For any vertex $v \in V_H$, assume $\phi(v)$ is the vertex model, and r_v the root of this vertex model. Then, let P_i be the coordinates of the root of the i^{th} vertex $v \in V_H$ in the canonical ordering. For each $r_{v \in V_H}$, there are at most four *gateways* in G. Each vertex v, its immediate neighbors are the gateways. Each path connecting r_v to any other vertex in G passes exactly one gateway of v. Generally, if earlier discovered paths or roots of other vertex models are blocking all paths from r_v to one of its neighbors, v's status is stated as *trapped*. It is trivial that assigning a gateway of vertex v to represent the root of another vertex model will increase chances of trapping v, and we need to prevent this from happening.

A trivial way to avoid getting trapped is to simply multiply the drawing coordinates of vertices by three. Since the distance between any immediate pair of vertices on the same grid row or column is always an integer multiplied by the length of a grid edge, it is evident in Figure 3.4 that embedding roots of vertex models at least three grid edges far from each other will guarantee that no two roots will be placed on the gateways of each other, or share their gateways. This approach will effectively make the grid area needed for minor embedding 9 times bigger than that of the straight line drawing. However, at the final stage of the minor embedding, this area will be reduced. Therefore, in the first phase, for any vertex $v \in V_H$,

- if (x_i, y_i) is the coordinates of v in the output straight line drawing, then the coordinates of the root vertex r_v is set to $P_i = (3x_i, 3y_i)$, and
- any vertex $g \in V_G$, adjacent to r_v that satisfies $|y_g - 3y_i| + |x_g - 3x_i| = 1$ is labeled as a gateway and added to $\phi(v)$.

In the second phase, we iterate over the vertices of H in the canonical ordering, and for each vertex v use the A* algorithm to find a shortest path between v and each of its neighbors

prior to v in the canonical ordering. The correctness of the final solution is guaranteed by the weights of edges in G which are passed to A*. All edges in E_G originally have the default unit weight. The only exception to the default weights are a set of 12 edges for each vertex model. Those 12 edges are the four edges connecting a root to its gateways, and the eight edges connecting the gateways together. We call this set of edges, the *perimeter* of each vertex, illustrated in Figure 3.4.

For a vertex $g \in V_G$, we can *make* g *unavailable* or inaccessible by setting the weights to infinity for all or some of the edges incident to g. Similarly, for an edge $e \in E_G$, we can *make* e *unavailable* and prohibit it from being included in a shortest path by simply setting its weight to infinity. In order to make g, *available* again, we need to reverse the changes, made to make the vertex unavailable in the first place. Depending on the desired status for g, available or not available, the weights associated to some or all of the edges incident to g should be modified. For instance, in the initialization phase of the SLM algorithm, for all vertices in V_H, we make the roots and perimeters unavailable.

Generally, in our algorithm, when a vertex is being connected to its neighbors below, the vertex models are formed or updated for the neighbors. The potential vertex model of each neighbor u can be pictured as a subtree, rooted in r_u, which grows its branches upward. Suppose we are processing a vertex v adjacent to u and placed above it. In order to connect these two vertices, v can reach down to the subtree of u which is a temporary version of the vertex model $\phi(u)$. So the subtree includes the vertices that have been used once, and the SLM algorithm has a preference for using them again, and hence, called the *available branches* and written AB. We have implemented this concept by a data structure which allocates D slots to each vertex, where D is the maximum degree of the input graph H. Each slot holds one branch that connects u to one of its neighbors. The details will be explained shortly through an example.

Consider the process of connecting the vertex $v \in V_H$ to its neighbors. Let N_v be the set of vertices adjacent to v and embedded below r_v in the grid. In other words, it is the set of vertices adjacent to v which precede it in the canonical ordering. To connect v to a neighbor $u \in N_v$, we start by making the perimeters of r_v available, and then, we check whether there is a grid vertex $g \in AB_u$ in available branches of u closer than r_u to r_v. Such a grid vertex can replace r_u as the destination of the A* search. By setting the source and destination of the path finding algorithm to vertices other than roots, we can partially merge edges incident to each vertex and hence, take better advantage of the minor embedding concept. Therefore, the SLM algorithm computes the *Manhattan distance* between r_v and each vertex in AB_u. The vertex g with the minimum Manhattan distance from r_v will be the new destination of A*. It is crucial to make sure that in making a vertex available we do not make the available branches or perimeters of other vertices accessible. Assume the existence of g sets a flag called *alternate goal* to true, and it is false, otherwise. We will explain the different actions the SLM algorithm takes based on the status of this

flag.

Assume the alternate goal is true. Then, g is the destination for A* because it is closer to v than r_u, and having smaller paths is one of our criteria toward a better minor embedding. Hence, we will make sure g is accessible. This way, A* can find the shortest path between r_v and g. Consider the following scenario: a path p connecting a pair of vertices s and t. The grid vertices on p are available to both vertices. Then, if another vertex r adjacent to s connects to a point $m \in p$, then it splits the path in two: the part between r_s and m, and the rest of the path up to r_t. The first part belongs to s but the second part should be inaccessible to s. Therefore, here, AB_u is split in two, and like cutting a branch, we eliminate the vertices that come after g in that specific slot. In other words, we eliminate the grid vertices that should no longer be used by u. This helps us maintain the correctness of the minor embedding as we build it, avoiding any mix up among vertex models.

Now lets consider the case when the alternate goal is false. Then, r_u is the destination for A* and should be made available. After A* finds the shortest path, the grid nodes forming that path are added to a free slot of u, except for the ones in the perimeters of u and v.

Before moving on to the next neighbor of v in N_v, the available branches and the perimeter of the current neighbor r_u are made unavailable. Once all members of N_v are processed, same procedure occurs for r_v. We also make sure that each grid node in each AB_u, where $u \in N_v$, has only been used once.

So far, we mentioned running A* search for path finding between vertices. Here, we discuss the details on how to modify the path finding step to decrease both the running time and the possibility of entrapping other vertices. Based on the description of A* algorithm in Section 2.4, in the worst case, it covers the whole search space. Therefore, by reducing the size of the search space, A* will find the shortest path faster. Further, the methodical modification of the search space does not prevent A* from finding a feasible solution. Consider finding a shortest path between any two vertices $u, v \in V_G$. We define the search space to be the smallest rectangular area C in G, containing the perimeters of both vertices, and we make the rest of the grid outside this area inaccessible. In order to do so, we require four parameters to set the corners of this rectangle: min_x, max_x, min_y, and max_y. Assume the bottom-left vertex in the grid is the origin of the xy-plane with coordinates $(0, 0)$, and each edge in the grid have the same unit length. In the Cartesian coordinate system, min_x is the minimum value of x among all vertices in the perimeters of u and v. Other three values are defined, similarly. This way, two vertices in G with coordinates (min_x, min_y) and (max_x, max_y) are the two opposing corners of C. By setting the weights of all edges incident to vertices outside of C to ∞, we limit A*'s search to inside C's boundaries. In a way, we are forcing the SLM algorithm to transform a straight line, representing a particular edge, to a path consisting of vertical and horizontal grid edges which fits in the same rectangle as the straight line.

To prevent crossings, the following actions are taken during an iteration embedding a vertex $v \in V_H$:

- when finding a shortest path between v and a neighbor u, making the perimeters of other vertices (including other neighbors of v) unavailable will guarantee that no vertex from other vertex models will be added to the path.
- the vertices forming all the discovered paths, connecting v to its neighbors, are made unavailable at the very end of this iteration. All the new members of available branches are also made unavailable before moving on to the next vertex in the order.

It might seem that these precautions encourage the SLM algorithm to employ more and more new grid vertices for embedding and hence, increases the size of vertex models. But on the other hand, during an iteration in which a vertex is being connected to its lower neighbor u, the grid vertices in AB_u can be re-used for this connection. Relatively, merging the edges incident to one vertex will decrease the collective size of all vertex models more than it could increase it by avoiding other vertices' perimeters.
It is noteworthy that according to [14], in a straight line drawing, adjacent vertices cannot be embedded on the same y coordinates in the grid. This condition implies that the i^{th} vertex in the canonical ordering v_i will connect to some of its neighbors $v_{j<i}$ in the i^{th} iteration, and the rest of them $v_{k>i}$ will connect to v_i in later iterations.
Algorithm 1 provides a summary of the SLM algorithm.

Now lets discuss the heuristics used by the A* search algorithm. To improve the search process of A*, we studied two heuristics: the *Euclidean* and the Manhattan distance between two points in the Cartesian plane.
For each two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ in the plane, the Euclidean distance between P_1 and P_2 is defined as follows:

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

This value measures the length of the straight line connecting the two points together. In an arbitrary grid, in which all edges have the same weight, the number of distinct shortest paths between vertices with coordinates $(0, 0)$ and (p, q) is [46]:

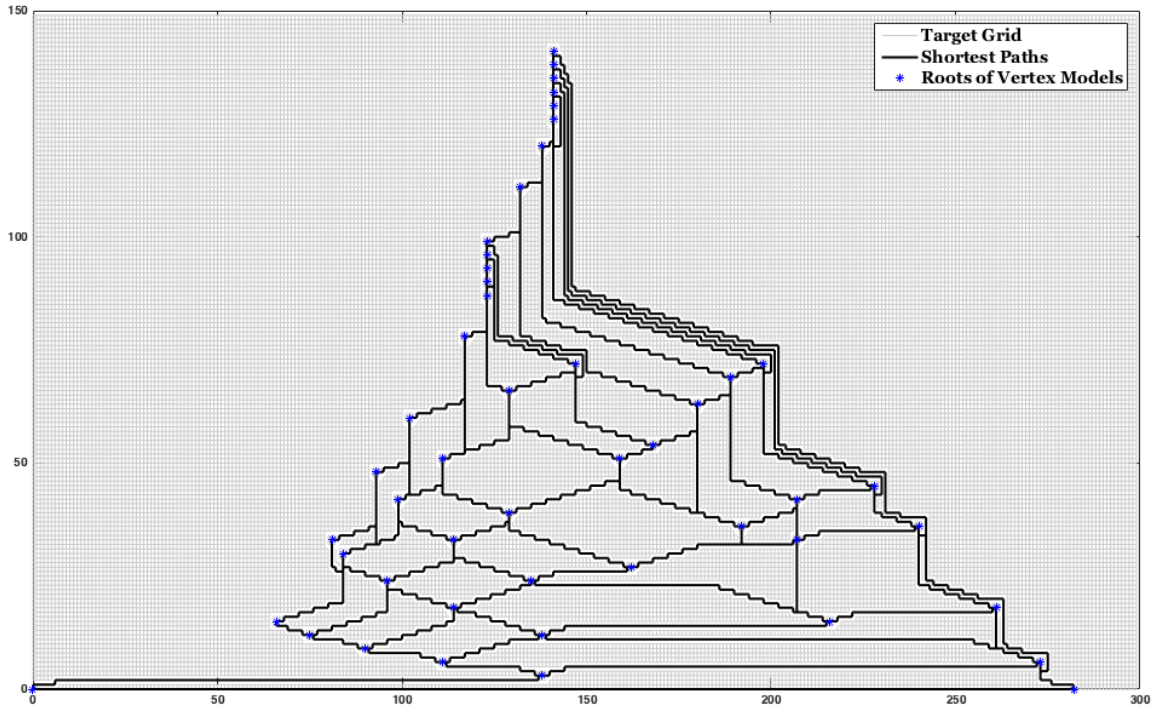
$$\binom{p+q}{p} \quad \text{or} \quad \binom{p+q}{q}$$

The reason is that the cost of the shortest path is exactly $p+q$, and the shortest path itself is a sequence of p edges parallel to the horizontal axis (moves to the right) and q edges parallel to the vertical axis (moves towards the top). The number of shortest paths is determined by the number of distinct selections of edges constituting the paths which define a move to the right (or top).

Algorithm 1 The SLM Algorithm: Minor embedding algorithm based on straight line drawing

Input: A planar graph $H = (V_H, E_H)$ and a grid graph $G = (V_G, E_G)$
Output: Failure or a minor embedding of H in G
Compute a canonical ordering $O = v_1, v_2, \ldots, v_n$ of V_H.
Let (X, Y) be vectors of the x and y coordinates of a straight line drawing of H.
for all $v \in V_H$ **do**
  $r_v = g \in V_G$ such that the position of g in the grid is $P_g = (3X_v, 3Y_v)$.
  Make the perimeter of r_v unavailable.
end for
for all $v \in O$ in order **do**
  Make the perimeter of r_v available.
  for all $u \in N_v$ **do**
    Find vertex $u* \in AB_u$ with minimum Manhattan distance to r_v and let S_{ind} denote the slot that includes $u*$
    if $u*$ is closer than r_u to r_v **then**
      Set $goal = u*$ as the destination of the A* search.
    else
      Set $goal = r_u$ as the destination of the A* search.
    end if
    Make $goal$ available, and make the enclosing rectangle C of r_v and $goal$ inaccessible to the vertices outside of it.
    Attempt to find a shortest path p between r_v and $goal$.
    if There is no path between r_v and $goal$ **then**
      return Failure
    else
      Add the vertices included in p to a pool Π, except the vertices in the perimeters of u and v.
    end if
    if $goal = u*$ **then**
      Delete the vertices in $AB_u(S_{ind})$ that come after $u*$.
    else
      Add the vertices in p to a free slot of AB_u .
    end if
    Make the isolated enclosing rectangle C available again.
    Make the perimeter of r_u and AB_u unavailable.
  end for
  Make sure all the grid vertices in AB_u, $u \in N_v$, have only been used once, if not, remove them.
  Make all $g \in \Pi$, as well as the perimeter of v unavailable.
end for

Figure 3.5: Minor embedding of 7×7 grid graph in a grid of size 141×282 using the straight line drawing algorithm and Euclidean distance as the heuristic function for A*.



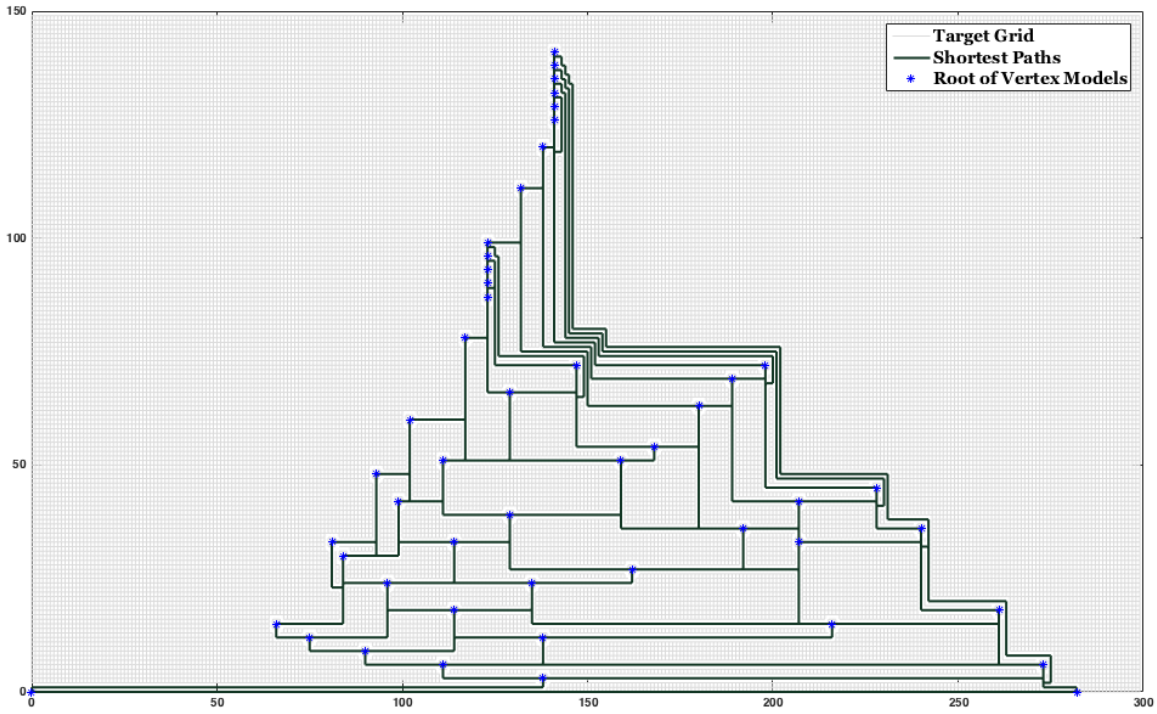
In our application, depending on the results of earlier iterations, a number of the possible shortest paths might be feasible, and A* will return the first-found shortest path. Therefore, if the Euclidean distance is chosen as the heuristic for A*, the selected shortest path will be the closest feasible path to the straight line connecting the two vertices. Figure 3.5 depicts the minor embedding of 7×7 grid graph in the grid, using the Euclidean distance as the heuristic.

In a sequence of edges constituting a path between two points of a grid, a *bend* refers to a subsequence of only horizontal (vertical) edges, being followed by a vertical (horizontal) edge. As illustrated in Fig. 3.5, Euclidean distance roughly maximizes the number of bends in every shortest path. The definition of the Manhattan distance of P_1 and P_2 follows:

$$d_M(P_1, P_2) = |x_2 - x_1| + |y_2 - y_1|$$

This value is exactly equal to the cost of the shortest path between P_1 and P_2, if all edges in E_G have a unit weight. Therefore, the selected shortest path will only have necessary bends unless A* is manipulated to randomly examine the possible options at each iteration, or the order of appearance for edges in the adjacency list is not the same for all the vertices in the grid. For instance, if all edges in the grid are available, using Manhattan distance as the A* heuristic to find the shortest path from the point $(0, 0)$ to (p, q) will lead to a sequence of

Figure 3.6: Minor embedding of 7×7 grid graph in a grid of size 141×282 using the straight line drawing algorithm and Manhattan distance as the heuristic function for A*.



horizontal(vertical) edges of size p (q), followed by q (p) vertical (horizontal) edges. Figure 3.6 illustrates the result of replacing Euclidean distance with Manhattan distance for the example depicted earlier in Figure 3.5.

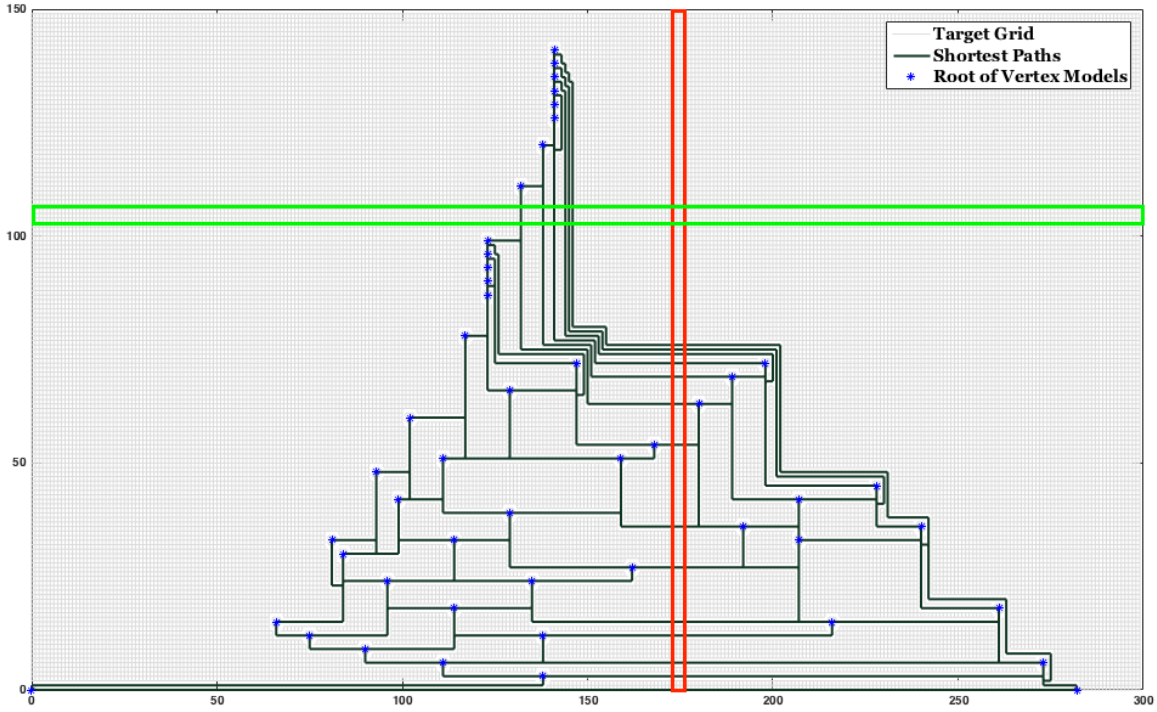
Our objective is to find a minor embedding with vertex models as small as possible. However, the results obtained by the two different heuristics do not demonstrate any meaningful differences in the sizes of vertex models. On the other hand, an easy post-embedding optimization is desirable, and hence, deploying Manhattan distance is preferable. As for the embedding with Euclidean heuristic, we have no reason to prefer that. Next, we will present a very simple technique to reduce the sizes of vertex models. This technique is much more effective when deployed with Manhattan distance as the A* heuristic.

3.2.2 The optimization technique

The post-embedding phase targets the long paths which we are prone to, mainly due to multiplying the coordinates of all vertices by three in the first phase of the SLM algorithm. This optimization technique will be referred to as *Algorithm O* in the rest of this text, and its details are as follows.

Each $m \times n$ grid has m columns of $n-1$ vertical edges, and n rows of $m-1$ horizontal edges. Consider one column in the grid after the embedding is finished. If none of the

Figure 3.7: Minor embedding of 7×7 grid graph by the SLM algorithm: the columns in the red rectangle and the rows in the green rectangle are a sample of unused columns and rows.



vertical edges of this column is a part of at least one of the shortest paths, then we can eliminate the column from the grid without affecting the feasibility of the final solution. Same status holds for a row which none of its horizontal edges have been used in the shortest paths. Figure 3.7 indicates a few columns in a red rectangle that can be removed from the embedding of the 7×7 grid graph. Removing these columns, or the rows in the green rectangle, shortens the paths, and hence, decreases the size of the vertex models without losing the feasibility of the solution.

Consider a feasible minor embedding which is a collection of vertex models. When removing a column, all the parts on the right side of that column should slide left. If that particular column crosses the point $(i, 0)$ of the grid, all vertices v of the minor embedding with $(x_v > i, y_v)$ are removed and replaced by vertices with coordinates $(x_v - 1, y_v)$. A similar statement holds for removing a row of the grid. Each eliminated row or column reduces the area consumed by the embedding, and also reduces the total size of vertex models by the number of shortest paths orthogonal and crossing to that row or column.

Algorithm O can be implemented with the following approach. Rows and columns of the grid essential to the embedding are marked as the shortest paths are being computed. When the SLM algorithm has successfully minor embed the graph, an offset is assigned to each

row r which indicates the number of inessential rows closer than r to the origin of the grid at point $(0, 0)$. To determine the offset for each row, a counter of removed rows c is set to zero, then the rows of the grid are iterated over, from the closest row to the farthest one from the origin. For each row r

- if r is to be removed, increment c,
- else, set a *shift offset* $S_r := c$ that determines the offset to be subtracted from y_v for each vertex v on that row.

A similar procedure is conducted for the columns. Each procedure runs in $O(m + n) < O(|V_G|)$ time. Next, the algorithm iterates over all vertices in all vertex models. For each vertex v, $\forall g \in \phi(v)$

- if g is on a row or a column, which is to be deleted, exclude g from the corresponding vertex model,
- else, find the vertex g' on the obtained coordinates by subtracting the corresponding offsets from the coordinates of g. Replace g with g' in the vertex model.

The complete process runs in $O(|V_G|)$ time.
For instance, in case of the 7×7 grid with $|V| = 49$, the minor embedding based on the straight line drawing coordinates requires a $3(2(49) - 4) \times 3(49 - 2) = 282 \times 141$ rectangular area in G. Figures 3.6 and 3.5 depict the minor embedding using Manhattan and Euclidean distance, respectively. Applying Algorithm O to the former embedding reduces its size down to 64 rows and 55 columns, while for the later, the optimization only compresses the embedding to 114 rows and 179 columns. This huge difference is in favor of Manhattan distance and fewer bends in shortest paths which is expected based on the definition of this optimization method.
The experimental results, provided in Table 3.1, demonstrate that using Manhattan distance will yield better or equally good optimization results comparing to the Euclidean heuristic, particularly for larger graphs. In addition, the comparison between pre- and post-optimization area shows that a great number of columns can be eliminated due to the triangular structure of minor embedding, an unnecessary attribute for the solution.

3.2.3 Time complexity

As stated before, obtaining a canonical ordering and a straight line drawing of H are linear time operations [14]. The time complexity of the algorithm is mostly determined by running A* per each edge $e \in E_H$. We can employ priority queues to handle the vertices and the estimation of their distances. For a priority queue of size $|V_G|$ storing the distance estimates, we can remove the minimum value and update any in $O(\log(|V_G|))$. Each A* search has

Table 3.1: The effectiveness of the post-embedding optimization on outputs of two different versions of the algorithm, one deploying Euclidean, and the other Manhattan distance as A* heuristic.

Graph Name	$\lvert V_H\rvert$	$\lvert E_H\rvert$	Pre-optimization Area	Post-optimization Area (Manhattan distance)	Post-optimization Area (Euclidean distance)	Area Optimization Percentage (Manhattan distance)	Area Optimization Percentage (Euclidean distance)
7×7 Grid	49	84	141×282	64×55	114×179	$55\% \times 81\%$	$20\% \times 37\%$
Ladder 20	40	58	114×228	106×101	111×110	$7\% \times 56\%$	$2.5\% \times 52\%$
Markström	24	36	66×132	56×66	63×97	$15\% \times 50\%$	$4.5\% \times 26.5\%$
Frucht	12	18	30×60	26×25	28×34	$13\% \times 58\%$	$6.5\% \times 43\%$
Dürer	12	18	30×60	27×28	29×47	$10\% \times 53\%$	$3\% \times 21.5\%$
Bidiakis Cube	12	18	30×60	27×31	29×43	$10\% \times 48\%$	$3\% \times 28\%$
Herschel	11	18	27×54	22×20	22×20	$18.5\% \times 63\%$	$18.5\% \times 63\%$

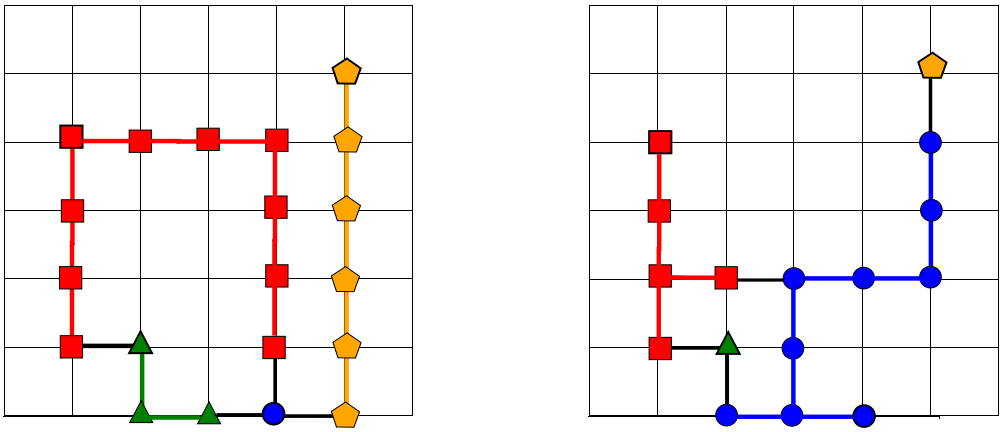
$O(|V_G|)$ steps in which the vertex of minimum distance estimate is chosen and eliminated from the priority queue and its (at most 3) neighbors are updated. Therefore, the running time of a single A* search is $O\big(|V_G| \log(|V_G|)\big)$. Hence, the time complexity of the minor embedding algorithm will be $O\big(|E_H| \times |V_G| \log(|V_G|)\big)$. This statement can be simplified as follows based on the upper bound of the straight line drawing output [14] and planarity of H (using Theorem 2.1.1): $|V_G| = 9(2|V_H| - 4)(|V_H| - 2)$, and $|E_H| \leq 3|V_H| - 6$, then the running time can be restated as $O\big(|V_H|^3 \log(|V_H|)\big)$.

In practice, the SLM algorithm's performance is much better than the mentioned upper bound. As discussed in Section 2.4, the worst case time complexity of A* is $O(b^l)$, where b is the branching factor, and l the length of the shortest path. It is evident that for finding the shortest path between two vertices of a sparse graph, such as a grid with maximum degree of 4, the branching factor will be relatively small. In fact, if the heuristic is optimal, meaning it demonstrates the exact distance between start and destination, then the effective branching factor would be 1 [45]. Here, both the Manhattan and Euclidean distances are optimal heuristics. Even if there are unavailable edges in the target grid to increase the length of the shortest path, these heuristics are still acceptable as part of the A* path finding search in the SLM algorithm. Therefore, the embedding phase of the SLM algorithm runs in almost linear time in $|E_H|$.

3.3 Analysis

In this section, we analyze the performance of the SLM algorithm. Even though the experiments are limited and were not continued further, due to reasons we will discuss shortly, they set the path for the next step of our research. The SLM algorithm had a high rate of success with graphs of maximum degree smaller than 10, and its rate of success dropped as the graphs became more dense.

Figure 3.8: Blue(circle) vertex is adjacent to other vertices, and the minor embedding algorithm embeds the edges incident to the green (triangle), red (square), and orange (pentagon) vertices, respectively. The root vertices are identified by thick black edges. Figures depict the results of two possible strategies for adding vertices to the vertex models.



Unnecessary Long Paths: According to [8], long paths are not desirable in a minor embedding. But in a straight line drawing, vertices are embedded in a triangular shape, to make embedding of edges as straight line segments possible. To create these triangles, the vertices, particularly the ones closer to the bottom of the triangle, are pushed to the right side of the y axis. On the other hand, the coordinates of the straight line drawing provide beneficial information for placing the vertices such that the possibility of crossing is low. Considering the challenging NP-hard problem we are dealing with, graph drawing algorithms can guide us by providing some information regarding the input graph and its structure. In the next chapter, we will explore another technique which is not limited by conditions such as straight edges, or triangulated graphs.

Assignment of Vertices Constituting the Paths to Vertex Models: The current approach is very simple and guarantees the feasibility of the solution. However, the decision regarding the choice of vertex model to include a vertex influences the quality of minor embedding and the size of vertex models. Consider the example illustrated in Figure 3.8, in which the difference between the various decisions and their impact can be observed. Both Algorithm D [8] and our method make a choice and assign the vertices right after finding the shortest paths. In the next chapter, we will explain that in many cases, this decision can be delayed, and deferring the definite assignment of a vertex $v \in V_G$ to a vertex model can contribute to better solutions.

Hard to Optimize: The output minor embedding is not optimal in terms of the size of vertex models and the area of minor embedding (the rectangle enclosing the embedding, see Section 1.3). Algorithm O, despite making progress, is still limited. Its performance is better for paths with fewer bends. However, since the triangular structure of the drawing is maintained throughout the minor embedding, all neighboring vertices have different distances from the x and y axes. Unless the graph is fairly sparse, this structure will abstain many rows and columns of the grid from being excluded.
On the other hand, the area of minor embedding depends on the embedded paths and the areas surrounded by them. The large areas (faces) in the embedding contain a certain number of grid vertices which are unused, and therefore, wasted. Addressing this issue requires a more sophisticated approach toward optimization. Moving the vertices to reduce the area of a specific face, without either increasing the area of other faces or entrapping some vertices, is not a trivial task. Though the mentioned issue is not impossible to resolve, given a few solutions are introduced in the next chapter, the fundamental fact remains that the problem can be avoided if a straight line drawing is not deployed as a base for minor embedding.

Not Always Successful at Finding a Minor Embedding: The limitations of minor embedding in a grid were discussed earlier. The number of available paths in any area of the grid is limited, and as the input graph becomes denser, the probability of failure in finding a path between two vertices during an iteration of the minor embedding increases.

Strong Dependency on G Being a Grid Graph: As mentioned before, we prefer to reduce the dependency of the algorithm on the structure of the target graph. Aside from embedding the roots of vertex models, other aspects of the algorithm, such as perimeters of vertices and the optimization technique, are based on the assumption that G is a grid graph. These dependencies are not essentially weak points of an algorithm designed to minor embed planar graphs in a grid. However, the greater objective would be to introduce an algorithm that with a slight modification can be deployed as an independent or auxiliary procedure to solve the minor embedding problem when G is an arbitrary graph.

Chapter 4

Minor Embedding Algorithm Based on Visibility Representation

In this chapter, we present a minor embedding algorithm that addresses the problems with the algorithm discussed in Section 3.2.1. This algorithm, called *VRM*, is able to embed all planar graphs into grid, with no limitation in regards to their degree or density, by using a different approach to selecting the coordinates of vertices and forming the vertex models. The VRM algorithm has two phases: minor embedding and optimization. In the first phase, the algorithm takes advantage of visibility representations, which are a kind of orthogonal drawing on the plane, to obtain the coordinates of root vertices. It uses a technique we call *dynamic arm assignment* to manage the vertex models efficiently. This technique provides an opportunity to modify vertex models while the embedding is being constructed and this improves the minor embedding quality. Later, in the second phase, this technique enables us to optimize the minor embedding in the grid by reducing the area used for embedding and the total size of vertex models. The optimization phase has two stages. The first stage can be described as a method for shrinking the faces of the minor embedding. The second stage can be pictured as vertical and horizontal sweeps that eliminates unused rows and columns. The first optimization method is referred to as the *primary* algorithm. The second is Algorithm O which was introduced in Section 3.2.2.

In Section 4.1 we will define visibility representations and review an existing algorithm that computes a visibility representation of any planar graph. Then, we will introduce the dynamic arm assignment and explain its benefits relative to our algorithm discussed in Chapter 3 and Algorithm D from [8]. With these two concepts, we can move on to explaining the minor embedding phase in the third section. Section 4.4 provides a detailed description of the optimization phase and its two optimization methods. Finally, Section 4.5 evaluates the correctness and running time of VRM. The overall algorithm is as follows. The first phase employs two embedding methods. One, which we call the *simple method*, always produces an embedding. The other method, called the *advanced method*, attempts

to produce an efficient embedding. The embedding phase executes the advanced method, and uses the simple method only if the advanced method fails. The second phase runs both the primary optimization method and Algorithm O.

4.1 Visibility Representation

The concept of visibility representation was first introduced by Otten and van Wijk [38], and later on, various researchers modified it and proved its generality for all planar graphs [19, 40, 50, 47]. Being used in various planar embedding algorithms [47, 36, 58], visibility representations are a kind of orthogonal drawing with no restriction on degree.
Let Γ be a set containing horizontal line segments in a plane. Any pair of segments α and β are called *visible* if they can be connected by a vertical line segment that does not intersect any other line segments [50].

Definition 4.1.1. Let $G = (V, E)$ be a planar graph. A *visibility representation* for G is a function $\Theta(G) : V \cup E \rightarrow 2^{\mathbb{R}^2}$ that

1. maps each $v \in V$ to a non-intersecting horizontal line segment $\Theta(v)$, called the *vertex segment* for v that does not intersect with the vertex segment $\Theta(u)$ for any $u \neq v$

2. maps each $e = (u, v) \in E$ to a vertical line segment $\Theta(e)$, called the *edge segment* for e, such that $\Theta(e)$ intersects with the vertex segments $\Theta(u)$ and $\Theta(v)$, representing its endpoints, and does not intersect any other vertex segment [50].

Different planar graphs can admit distinct types of visibility representations according to [47]. These types of visibility representations are presented here:

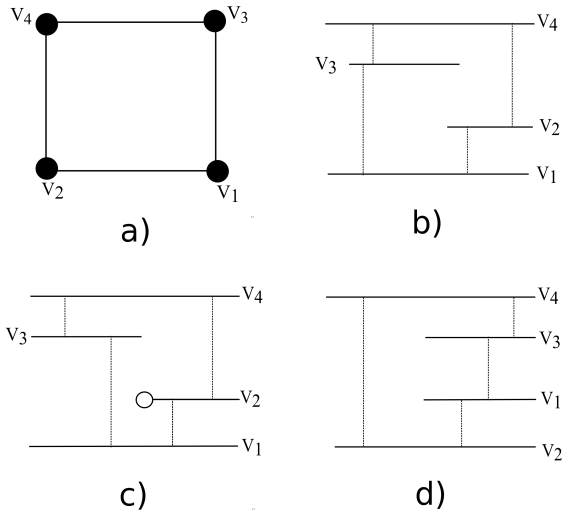
- *Weak visibility representation (w-visibility)*: This is the default type of visibility representation, and corresponds to Def. 4.1.1.

- *Interval visibility representation (ε-visibility)*: Similar to w-visibility except that the vertices are mapped to intervals instead of segments, and those intervals can be connected to each other using vertical line segments if and only if the corresponding vertices are adjacent.

- *Strong visibility representation (s-visibility)*: Similar to w-visibility except any pair of adjacent vertices can be connected to each other using a vertical line segment that do not intersect any horizontal line segment other than these two. In other words, any vertical line between any pair of non-adjacent vertices intersects with at least one other horizontal segment.

Figure 4.1 illustrates the three types of visibility representations just described for a sample graph. Several characteristics can be deduced about a graph based on the type of its

Figure 4.1: Types of visibility representations a) A sample planar graph. b) w-visibility representation. c) ε-visibility representation. d) s-visibility representation.

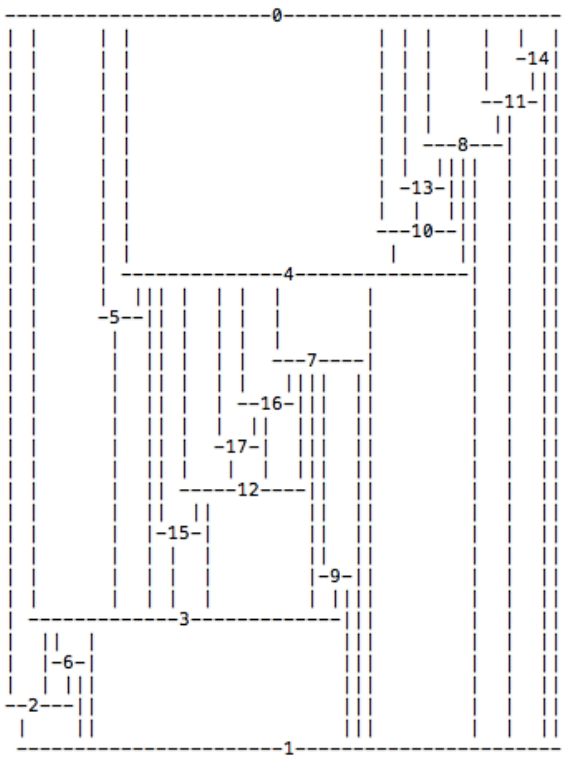


visibility representation. For instance, a graph admits a ε-visibility representation if it has a planar embedding in which all the cut vertices are on the external face boundary [47].
To compute visibility representations, we employed Boyer's implementation [1] of the plane embedding algorithm from [7] which also computes visibility representations in linear time [6]. Figure 4.2 illustrates a sample visibility representation produced by Boyer's algorithm. The output is in the ASCII format with the vertex segments all having unique horizontal coordinates and specific vertical positions.

4.2 Dynamic Arm Assignment

In this section, we explore the different situations that might occur when connecting two vertices on the grid, and introduce an efficient technique to manage the vertex models, along with an explanation regarding how it gives us more control toward optimizing the minor embedding. To make a distinction between a vertex in the grid or in the input graph, the word *node* has been used here to refer to a vertex in the grid.
In the SLM algorithm in the previous chapter, similar to Algorithm D in [8], when processing a vertex v, the nodes forming the shortest path are assigned to a vertex model as soon as the path is chosen. But here, we start with partial assignments of nodes and delay the final decisions to gather more information regarding the structure of the input graph.
Assume v is the current vertex which needs to be connected to its neighbors $u_1, ..., u_k$. Suppose that when connecting v to u_i $(1 \leq i \leq k)$, A* discovers the shortest path $(g_1 = v, g_2, ..., g_m = u_i)$ between them. Then, the VRM algorithm should decide how to assign the nodes g_t, $2 \leq t \leq m-1$, to vertex models. The algorithm uses two data structures. The first one stores the grid vertices assigned to each vertex model, and hence is called the

Figure 4.2: ASCII art of a visibility representation of a random Apollonian network of 18 vertices and 48 edges



available points or *arm*, denoted A_v for vertex v. The term "available point" of v refers to a node that will be available later as a potential destination for A* when embedding an edge incident to v in the input graph, instead of the vertex v. Similar to the SLM algorithm, we avoid crossings by changing the weights of the grid edges, making nodes available or unavailable at certain steps of the VRM algorithm. When connecting a vertex v to another vertex model, the nodes forming that vertex model will be available, and the closest of them to v will be the destination of A* for that connection. The sets of available points is initialized to the roots of vertex models. In this method, we do not require the perimeters we presented for the SLM algorithm, but if we did, we would have also included the vertices on the perimeters. The sets of available points are modified through out consecutive iterations. They shrink and expand until the input graph is successfully minor embedded in the grid. We also employ another data structure, called the *point permit* and denoted P, to record all vertices ever using any node of the target grid G. Given the input graph H, P is a Boolean matrix of size $|V_G| \times |V_H|$, initialized to be all zeros. For each $g \in V_G$, each non-zero element $P(g, y)$ of this matrix identifies to a vertex $y \in V_H$ that uses g and includes it in its vertex model $\phi(y)$. The union of vertices using a node g is denoted $P_g = \cup(\{y \in V_H | g \in \phi(y) \text{or} P(g, y) = 1\})$. We employ this information to check whether a vertex y is permitted to include a node in its vertex model $\phi(y)$. In other words, this

information guides us to make consistent decisions when modifying the available points.
After placing the roots of vertex models in the grid, those nodes are assigned to the vertices they represent, as a part of initialization. Then, considering the mentioned shortest path $(g_1 = v, g_2, ..., g_m = u_i)$, the first time a node g_t, $2 \leq t \leq m-1$, is employed to connect a pair of vertices v and u_i, the entity $P(g_t, v)$ and $P(g_t, u_i)$ will be set to 1. g_t is also added to both available points of v and u_i. The second time g_t is included in a shortest path, it is either connecting v to another neighbor, or another vertex w is connecting to u_i. g_t is unavailable to any vertex other than the two vertices listed as its point-permits. Whichever vertex, v or u_i, that needs this node for the second time, is the one that will permanently include it in its vertex model. Lets assume here that u_i is using g_t again (the case for v will be similar). Then, the algorithm proceeds with the following operations:

- set $P(g_t, v)$ back to zero,
- eliminate g_t from A_v

In general, P_{g_t} can have one of the following three states. It can be empty, meaning g_t does not belong to any vertex model yet. It can be a matrix of size one, meaning g_t has been permanently assigned to a particular vertex model; and it can be a matrix of size two, meaning g_t is assigned to two vertex models, which are adjacent, and the assignment is not yet final. In this case, the node can be eliminated from either vertex model, and hence, become a permanent part of the other. Based on the mentioned states, since each root is assigned to one vertex in the initialization, its status never changes.
In minor embedding we can use a node on multiple paths from a vertex v. A node g in the set of available points A_v can be reused for connecting v to some other vertex u, unless it has already been included in the vertex model of some other vertex y. Nevertheless, after processing the vertex v, all the nodes in its available points are made unavailable before processing the next vertex in the order to prevent A* from using them and mixing the vertex models mistakenly.
After iterating over all vertices, there might be still nodes assigned to two vertex models. One way to make sure the vertex models are exclusive would be to assign the nodes based on the chosen shortest path. We can simply assign the nodes constituting half the path to one vertex, and the others to the other vertex. Another approach would be coming up with an agenda for optimizing the minor embedding based on our application. For instance, in order to minimize the maximum size of vertex models, we can assign the nodes to the two vertex models, starting with the smaller one, to avoid increasing the maximum size of vertex models, if possible.

Figure 4.3 illustrates an example of the dynamic arm assignment. Assume the objective is to embed the following edges: (v_1, v_2), (v_1, v_3), and (v_4, v_2). The VRM algorithm connects each vertex to its neighbors below. There is no vertex below v_1, hence, we start with v_2.

We connect v_2 to v_1 as in Fig.4.3.b and the nodes in the shortest path are added to both available points because they have not been employed before. The point-permits of these nodes is now a set of two $\{v_1, v_2\}$. Then, we move on to the next vertex v_3. There are two shortest paths between v_3 and v_1, but only one of them uses nodes already added to either $\phi(v_1)$ or $\phi(v_3)$. That path is our preference to prevent the algorithm from increasing the size of vertex models. Reusing a node to connect v_3 and v_1 will lead to the following actions: the node is permanently assigned to the vertex which needs it for the second time (here v_1) meaning its point-permit will be $\{v_1\}$, and it is eliminated from the available points A_{v_2}. The results is shown in Fig. 4.3.c. The last vertex v_4 is adjacent to v_2. Similar to the case of v_3, v_4 is connected to its neighbor by reusing some nodes and hence, those nodes are permanently assigned to v_2, as in Fig. 4.3.d. The node in the middle of v_2 and v_1 can be assigned to either one of these two vertices based on our desired criteria and application.

4.3 Minor Embedding Algorithm - First Phase

In this section, we introduce two alternative methods of minor embedding a planar graph using a given visibility representation.

The visibility representation can be considered a minor embedding in the grid. Consider laying the visibility representation on a grid such that all the endpoints of line segments (horizontal and vertical) lie on grid nodes, and the segments themselves lie on some sequence of grid edges. This way, any vertex model $\phi(v)$ of a vertex v includes the horizontal vertex segment representing v. The vertical segments, now comprised of one or more grid edges, are divided and assigned to the vertex models of the vertices they connect. Figure 4.4 represents a visualization of this abstract idea (it is not generated by the VRM algorithm). This way, we can always obtain a minor embedding in the grid using a visibility representation. This approach is called the *simple method*, due to its simple process of minor embedding a planar graph.

We will now discuss some details. Given the visibility representation, we compute the locations of the roots for vertex models. Consider the visibility representation in a Cartesian space. Then, all the segments can be described by Cartesian coordinates. Starting with vertex segments, their unique vertical locations are interpreted as unique coordinates on the y-axis. The distance between the two closest consecutive vertex segments can be considered as the unit of measurement for the Cartesian coordinates. As for the vertical edge segments, their unique locations on the x-axis can be measured based on the chosen unit. We have implemented a parser tailored for this application. We feed the visibility representation as a text file, and parse each line from the top. If the line contains a horizontal vertex segment, its index will be considered as the y axis coordinate for the vertex it is representing. In addition, the horizontal distance from the beginning of a line to the middle point of a vertex segment in that line amounts to the x coordinate of that vertex. In doing so, the vertex on

Figure 4.3: Example of Dynamic Arm Assignment technique: minor embedding the edges (v_1, v_2), (v_1, v_3), and (v_4, v_2). a) Initial state. b) After connecting v_2 and v_1. c) After connecting v_3 and v_1 and assigning all "reused" nodes to v_1. d) Connecting v_4 and v_2 and allocating the reused nodes to v_2.

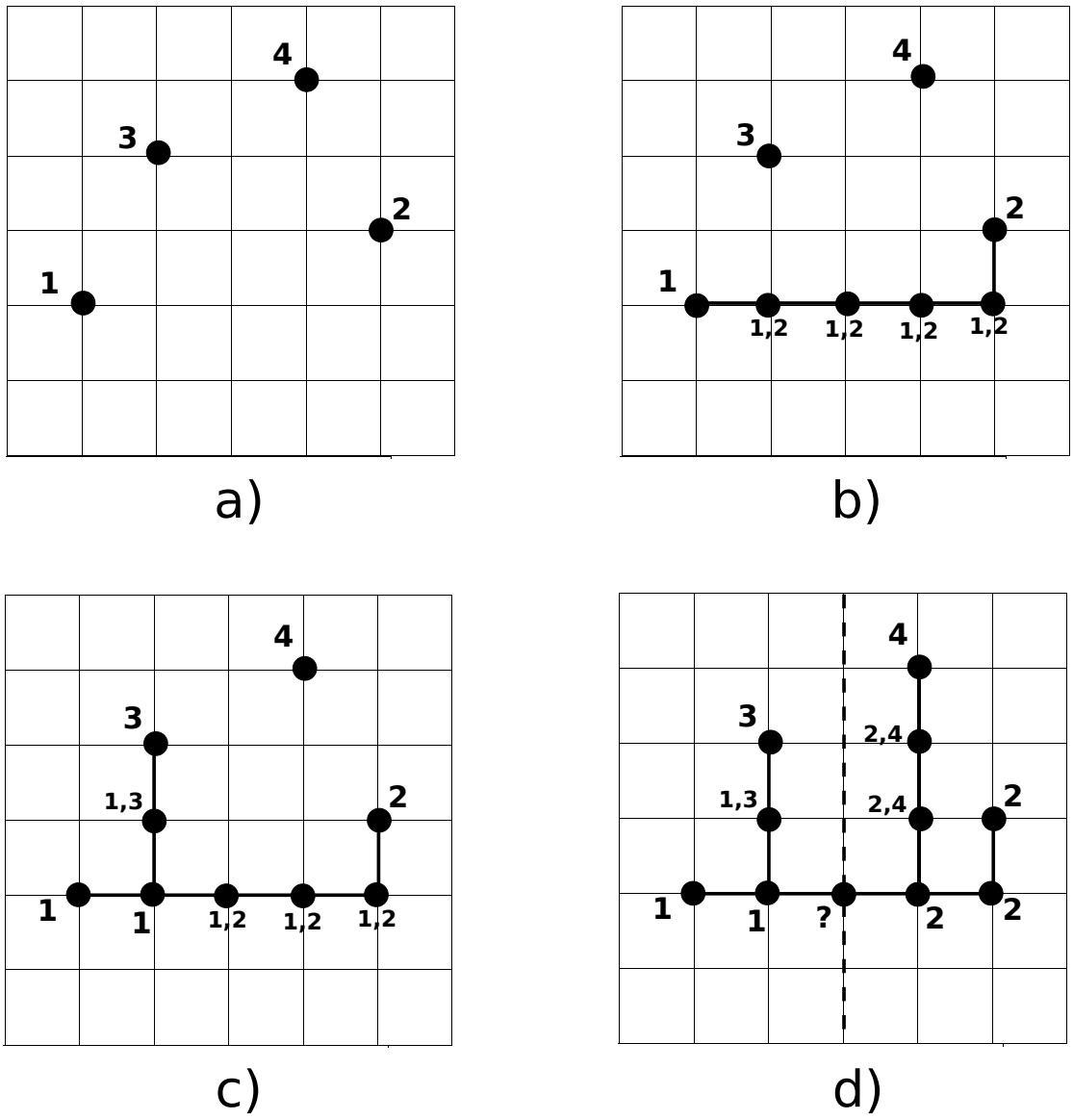
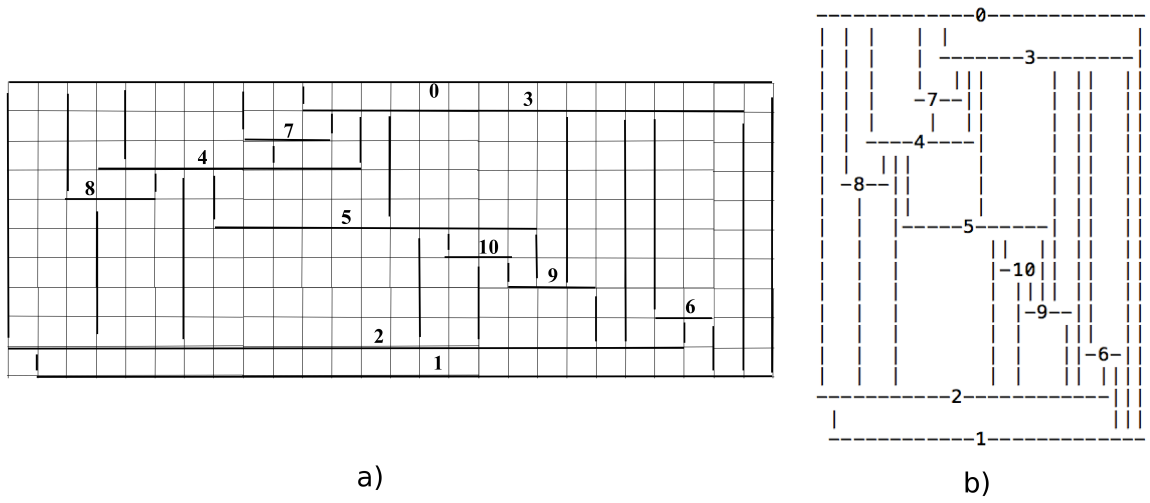


Figure 4.4: A sample of simple visualization of embedding a visibility representation in a grid - this figure is not generated by the VRM algorithm.



top of the visibility representation will be the first vertex to be embedded in grid.
Using the mentioned process, a visibility representation of graph $H = (V, E)$ can be embedded in a grid of size $|V_H| \times |E_H|$. Therefore, the simple method is complete based on these two facts: i) a visibility representation of any planar graph can be computed by [1], and then, ii) it can be easily transformed into a minor embedding in a grid of the mentioned size.
The simple method, though always successful, is not space efficient. The visibility representation embeds the edges incident to a vertex as disjoint lines, where they can partially merge and form a subtree connected to all the endpoints, see Figure 4.2. With this type of modifications, the overall area of embedding can be reduced without losing the feasibility. Therefore, we present the *advanced method* which minor embeds the input graphs using the dynamic arm assignment technique (in the previous section) to better handle the vertex models.

The outline of the advanced method is explained in Algorithm 2. We start with the visibility representation, computed by any existing implementation. Our approach toward processing the vertices is inspired by the bottom-up approach of the canonical ordering in the SLM algorithm. It means that after computing the coordinates of vertices, we sort the vertices in an increasing order based on their positions on the vertical axis. Then, according to the computed order, all vertices are processed and connected to their neighbors that precede them in the order. For instance, when processing a vertex v of degree more than one, its neighbors below are sorted and stored in NB_v in an increasing order of the distances between each root and r_v. We start by embedding the edge $e = (u_i, v)$, incident to the closest neighbor $u_i \in NB_v$. If there are any nodes in the available points of either v or u_i, their weights are set to 1, and hence made available before running the path finding

Algorithm 2 Pseudocode of the advanced method in the embedding phase of the VRM algorithm

Input: Visibility representation θ_H of the input graph $H = (V_H, E_H)$ and a target grid graph G with dimensions $|V_H| \times |E_H|$
Output: The set of vertex models in a feasible minor embedding of graph H in a grid graph G, along with the roots, available points, and point-permits of nodes in G
Initialization:
Compute the coordinates for V_H and the vertical order of vertices O based on θ_H
for all $v \in V_H$ **do**
  Place r_v in the grid based on $\theta(v)$
  Set $A_v = \{r_v\}$, $P_{r_v} = \{v\}$, and make r_v unavailable
  Store the neighbors placed below v in NB_v and sort them in a descending order based on their positions on the vertical axis
end for
for all $i = 1$ to $|V_H|$ **do**
  $v = O(i)$
  for all $w \in NB_v$ **do**
    if A_v is not empty **then**
      make those grid nodes in A_v available to the algorithm to be used in the embedding
    end if
    if A_w is not empty **then**
      make the grid nodes in A_w available to the algorithm to be used in the embedding
    end if
    Run the path finding algorithm (A*) with Manhattan heuristic from r_v to r_w
    Apply the dynamic arm assignment technique on the grid nodes forming the shortest path
    Make the grid nodes in A_v and A_w unavailable
  end for
end for

algorithm (A*). After A* discovers the shortest path based on Manhattan distance heuristic, the dynamic arm assignment technique manages the assignment of the nodes forming that path to the vertex models. At the end, the updated available points decide which grid nodes and edges should be inaccessible in subsequent iterations, and their weights are set to infinity.

The key components here are the vertex models. By manipulating the vertex models, we can achieve all the mentioned objectives. Therefore, in order to improve the minor embedding, we introduce the following optimization techniques to modify the vertex models. Essentially, the optimization phase takes its chance by *re-embedding* the vertex models. The details of this process are explained in the upcoming sections.

4.4 Optimization Method - Second Phase

4.4.1 First stage

The objective of the minor embedding algorithm is to generate a valid embedding of the input graph, and preferably, optimize it. Dependent on application, the goal of optimization can vary. For instance, the goal could be generating an embedding in as small a grid as possible, or with minimum grid nodes possible. Looking at Figure 4.5.a, one visible barrier in embedding this certain graph in a smaller grid would be the areas confined by the embedded edges. Here, we present a post-processing algorithm, called the *primary optimization method*, to minimize these areas, and therefore the total area required for the embedding, while reducing the size of vertex models to address both mentioned goals.
The primary optimization method attempts to improve the minor embedding in a certain number of *rounds*, each of which consists of processing and re-embedding all vertices of V_H. Every iteration in one round processes one vertex model and consists of three major steps: i) finding a new root, ii) safe elimination and preparation, and iii) re-embedding a certain set of edges. On the other hand, this optimization method can be described with its two key techniques. The first technique chooses a new root in the first step, and the second technique offers an alternate procedure for re-embedding the edges in the third step.
Relocating the roots of vertex models, followed by new paths representing the edges, alters the relative positions of vertices (vertically and horizontally). With this alteration, the conditions of visibility representation no longer holds, and therefore, there is no guarantee of a successful minor embedding. The vertex models and the paths connecting them might trap each other. To preserve the feasibility of the solution, when processing a vertex, if at any stage we face a problem, we change everything back to the last valid state. This way, the successful modifications of vertex models will lead to a better minor embedding, and the solution stays feasible.
Consider processing the vertex $v_i \in V_H$ in the i^{th} iteration with r as the root of its vertex model $\phi(v_i)$. In the first step, the goal is to find a node r' in A_{v_i}, permanently assigned to v_i, as the new root which should be closer to the neighbors than the current root. Given a fixed vertex model, the process can be described as a simple re-assignment of the label 'root' to a new node closer to the neighbors. With this condition, we are trying to preserve the faces of the minor embedding and take a chance to improve it, rather than making drastic changes which with a high probability may result in crossings. In order to compare the nodes in A_{v_i}, we introduce a parameter, called the *proximity value*, to calculate the distance between a node to the arms of neighbors:

$$\psi(u) = \max_{w \in N_{v_i}} \big(\min_{x \in A_w} d_E(u, x) \big) \tag{4.1}$$

where $u \in A_{v_i}$ permanently, N_{v_i} represents the set of vertices in V_H adjacent to v_i, and d_E the Euclidean distance between two points in the grid G. The vertex minimizing Equation 4.1 will be chosen as the new root $r'_{v_i} = \text{argmin}_{u \in A_{v_i}, |P_u|=1} (\psi(u))$. Since the nodes in the finalized subset of A_{v_i} are connected to all the neighbors, the new root will likely find new shortest paths to embed these connections. On the other hand, there is a chance that a neighbor might get trapped which will impede the algorithm from embedding all the connections successfully. Then, the change is discarded.
After the choice of the new location for the root r_{v_i}, the algorithm proceeds to the second step which is the safe elimination of the currently employed nodes, and releasing them. The main objective is to clear A_{v_i} (and $\phi(v_i)$) which includes two categories of nodes: permanently assigned to v_i, and shared between v_i and some other vertices adjacent to it. The required operations vary for these two types:

- for a node t permanently assigned to v_i: we just need to simply delete t from A_{v_i}, alter the point permit from $P_t = \{v_i\}$ to $\emptyset$,
- and for a node s shared between A_{v_i} and A_w where $w \in N_{v_i}$: we need to remove s from both available points A_{v_i} and A_w, change the point permit $P_s = \{v_i, w\}$ to $\emptyset$.

Further, any edge (h, g) incident to a node $g \in A_{v_i}$ is also made available and its weight set to 1. Finally, $A_{v_i} = r'_{v_i}$, and $P_{r'} = \{v_i\}$ for a permanent allocation of r' to v_i.
In the third step, we want to re-embed the edges incident to v_i. The process is similar to the embedding phase of the algorithm, but with the difference that we prioritize some edges over other available ones to further encourage the algorithm to reuse the grid nodes. We mentioned this concept as a distinguishing principle between minor embedding and orthogonal drawing, and is implemented by altering the weights of grid edges. As stated, the weight of the grid edges accessible to the path finding algorithm (A*) are set to 1. Here, we set the weights to $\epsilon > 0$ for those whose corresponding end points are already in the vertex models. Therefore, A* is further persuaded to choose the already discovered paths versus choosing new grid nodes. In the experiments, we discuss the great success of this process in reducing the size of vertex models.
Algorithm 3 illustrates a pseudo code for the primary optimization method.

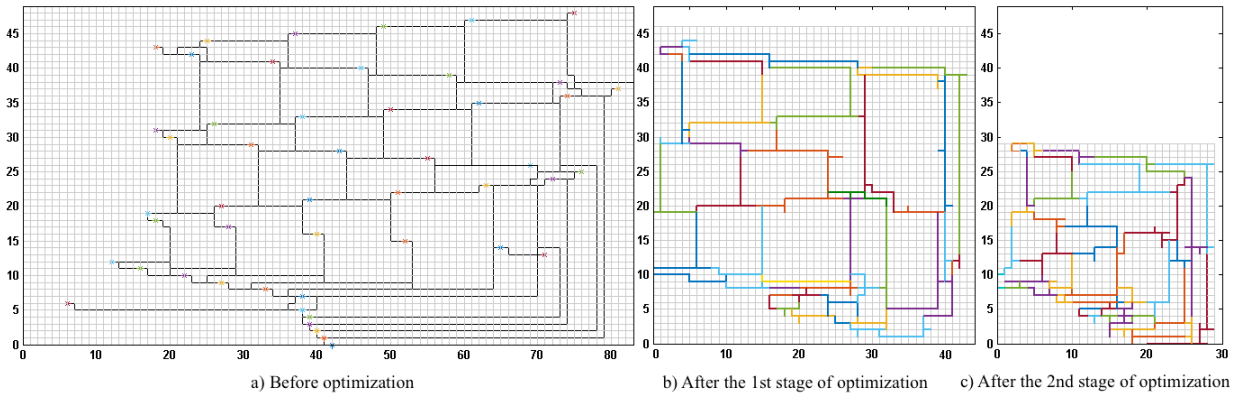
4.4.2 Second stage

There are cases for which the embedding might need some obvious alterations. For instance, Figure 4.5 illustrates a sample minor embedding of a grid graph based on visibility representation. In Fig. 4.5. b, a naive approach would be to eliminate all the columns and rows whose removal will not cause any loss of information. Therefore, we employ Algorithm O, introduced in Section 3.2.2, to further optimize the embedding. The algorithm can be summarized as follows: We iterate over all rows and columns of the grid, and while finding

Algorithm 3 Pseudo code of the primary optimization method

Input: Feasible minor embedding $J$of graph H
Output: A feasible optimized minor embedding K of graph H
for A certain number of rounds **do**
  Generate a random order for vertices O
  for all $v \in O$ **do**
    for all $u \in A_v$ such that $P_u = \{v\}$ **do**
      Compute $\psi(u)$ based on Equation 4.1
    end for
    $r_v = \text{argmin}_{u \in A_{v_i}, |P_u|=1} \left(\psi(u)\right)$
    for all $u \in A_v$ **do**
      for all $w \in P_u$ **do**
        Eliminate u from A_w
      end for
      Set $P_u = 0$
      for all $(x, u) \in E_G$ such that $P_x = 0$ **do**
        Set the weight to 1
      end for
    end for
    Set $A_v = \{r_v\}$ and $P_{r_v} = \{v\}$, then make r_v unavailable
    for all $t \in N_v$ **do**
      Compute $d_E(r_v, r_t)$
    end for
    Initialize a priority queue q to prioritize the neighbors with minimum Euclidean distance to r_v
    while q not empty **do**
      $t =$ Find and eliminate the minimum vertex in q
      Make A_v and A_t available by changing the weights of an appropriate set of edges to ϵ
      Run A* with Manhattan heuristic to discover the shortest path between r_v and r_t.
      Manage the vertex models and the nodes in the chosen path by the dynamic arm assignment technique
      Set the weights back to inf to make the edges inaccessible
    end while
    if the process is not successful **then**
      Discard the change of the root vertex
    end if
  end for
end for

Figure 4.5: The output of the VRM algorithm for a 7×7 grid graph. a) Before optimization b) After the first stage of optimization c) After the second stage of optimization



the ones that should be eliminated, we assign a *shift offset* to those essential to the embedding. In other words, the shift offset of a certain row r is equal to the number of rows marked to be eliminated preceding r while we iterate over all rows, from the closest row to the farthest one from the origin $(0, 0)$. The same process happens to compute the shift offsets for columns. As mentioned, the explained operation runs in $O(|V_G|)$ time.

4.5 Correctness and Time Complexity

The VRM algorithm is always successful in minor embedding a planar graph $H = (V, E)$ in a grid graph of size $|V_H| \times |E_H|$. Its completeness can be proved based on the first phase of the algorithm. As mentioned before, in the first phase, the VRM algorithm attempt to minor embed the input graph using the advanced method. If it fails, the simple method will minor embed the graph. The simple method is complete, as explained in Section 4.3, based on the existence of a visibility representation for any planar graph.

In the second phase, the VRM algorithm iterates over vertices V_H and attempts to improve the solution. In each iteration, a change is acceptable if it improves the solution without making it infeasible. Therefore, a feasible solution will be obtained at the end of the second phase.

As for the time complexity, we can analyze the two phases separately. The visibility representation can be computed in linear time [6], if it is not given to us. After that, the running time of the embedding phase is mainly dominated by the A* search. Similar to our argument for the SLM algorithm in Section 3.2.3, there are $|E_H|$ runs of A* on a grid with $|V_G|$ vertices. Each stage of A* can take time up to $O(log(|V_G|))$, and there are $O(|V_G|)$ stages. So, the running time of the first phase is $O(|E_H| \times |V_G| \log(|V_G|))$. Since $|V_G| = |V_H| \times |E_H|$, the running time can be presented as $O(|E_H|^2 \times |V_H| \log(|V_H| \times |E_H|))$. If H is a planar

graph with $|V_H| \geq 3$, based on Theorem 2.1.1, $|E_H| \leq 3|V_H| - 6$. This way, we can rewrite the upper bound to be $O(|V_H|^3 \log(|V_H|))$. Further, similar to the SLM algorithm, since G is a grid graph, Manhattan distance is an optimal heuristic for A*. Then, the branching factor is close to 1, and practically, we can expect a running time of close to linear in $|E_H|$. The second phase consists of the primary optimization method and Algorithm O. The running time of Algorithm O was shown in Section 3.2.2 to be $O(|V_G|)$ which in this case it can be rewritten as $O(|V_H| \times |E_H|) < O(|V_H|^2)$. As for the primary method, we have a certain number of rounds each of which is $|V_H|$ iterations. Consider one iteration in one of the rounds. The algorithm takes three major steps: i) finding a new root, ii) safe elimination and preparation, and iii) re-embedding a certain set of edges. Here, we analyze each of these steps individually.

Consider processing the vertex $v \in V_H$. The first step attempts to find a new location for the root vertex r_v needs $|A_v| \sum_{u \in N_v} |A_u| O(1)$ computations of Euclidean distance between vertices. Here, $|A_v|$ denotes the size of the set of available points of v, and N_v is the set of vertices adjacent to v. $\sum_{u \in N_v} |A_u|$ can be upper bounded by $C_1|V_G|$, where C_1 is a constant so the first step of each iteration runs in time $|A_v| \times C_1|V_G|$. The second step removes the current vertex model to make room for the new one, hence, it runs in time $O(|A_v|)$. Finally, the third step runs A* to embed each edge incident to v. This step requires $O(|V_H|)$ to initialize a priority queue for neighbors which can be updated in $O(log(|V_H|))$. Other than that, it is similar to the embedding phase which makes its running time $O(|V_H|^3 \log(|V_H|))$. Now if we consider one round, we have $\sum_{v \in V_H} [(|A_v| \times C_1|V_G|) + |A_v| + |V_H|^3]$. The term $\sum_{v \in V_H} |A_v| \leq C_2|V_G|$, where C_2 is a constant so the whole statement can be rewritten as $O(|V_G|^2 + |V_H|^2)$. Since $|V_G| > |V_H|$, the running time is $O(|V_G|^2) = O(|V_H|^4)$.

Chapter 5

Experiments

5.1 Outline of the Experiments

In this chapter, we will discuss two sets of experiments. The first set evaluates the performance of VRM and the impact of its two optimization techniques on the quality of solutions. The second set experiments on Algorithm D, described in Section 2.5, to compare its performance with that of VRM for a better perspective. Both algorithms have been assessed on 145 instances including a set of individually known graphs and instances from 6 different families. The comparison is mainly in terms of success rate and the average size of vertex models.
The data presented as the output of the primary optimization method, as well as the output of Algorithm D, is the same as the contents of Tavakoli's thesis [52].
The upcoming plots and tables are presenting the instances by their number of vertices. The reason is that for most instances, the complexity and density (as in the number of edges) increase with the number of vertices.
Our measuring criteria are defined below, followed by brief descriptions of the different families of graphs we have tested on the algorithms, and how they were generated.

Measuring Criteria

The performance of the algorithms is assessed based on the following criteria:

- **Total size of vertex models (solution size)**: This criterion is the primary factor in our assessment, assuming the algorithm is successful in minor embedding the input graph. It was also included in [8], and its minimization is the objective because it represents the efficient use of resources.

- **Area of embedding (embedding area)**: This refers to the area of the smallest rectangle enclosing the embedding. As the optimization algorithm succeeds in generating a denser and smaller embedding, a greater decrease in area of grid embedding is

expected which is a huge step toward space efficiency. This measurement was inspired by the grid drawing concept in [20].

- **Probability of success**: It is defined by the ratio of the number of successful runs to the total number of attempts for a single instance. This factor helps us to evaluate the performance of Algorithm D which often fails in minor embedding different graphs.

- **Ratio of area used to upper bound**: This criterion indicates how much area is used by the minor embedding at the end with respect to what is allocated to it in the beginning. Recall that for an input planar graph $H = (V, E)$, we can compute its visibility representation that can be embedded in a $|V_H| \times |E_H|$ grid, and hence, $|V_H| \times |E_H|$ is the upper bound on the required area for embedding. That is, the measure is $\frac{\text{area of embedding}}{|V_H| \times |E_H| = |V_G|}$.

Instances

We studied several instances of the following families of graphs.

- Random Apollonian graphs: If we divide a triangle into 3 smaller triangles recursively, an Apollonian graph (or Apollonian network) will be generated. Apollonian graphs are known for their particular properties and applications such as being chordal, following a power law distribution of vertex degrees, and so forth. To generate random Apollonian graphs, we start with one triangle, and for a certain number of iterations, we place new vertices in random faces of the graph, and then connect each vertex to the three vertices bounding the specific face it is in.

- (Square) Grid graphs: Grid graphs are interesting to study since they can be embedded optimally in a grid, given the right choices have been taken.

- Random Series Parallel (random sp) graphs: Consider a graph with two vertices, s and t, distinguished as *source* and *sink*. Any pair of such graphs, called st-graphs, can be converted into a new st-graph by the following operations: i) the sources of the two graphs are merged, as well as their sinks; ii) the source and sink of one graph are merged with the sink and source of the other graph. These operations are called *parallel* and *series compositions*. The st-graph generated by these operations is a *series-parallel* graph (sp-graph). A random sp-graph is generated by a random number of operations on st-graphs of size 2 (path graph P_2) where the types of operations are also chosen randomly.

- Maximal Series Parallel (maximal sp) graphs: If adding an extra edge to a sp-graph results in the loss of its status as a sp-graph, then that graph is a maximal sp-graph.

- Wheel graphs: A wheel graph W_n of n vertices consists of $n-1$ vertices of degree 3, forming a cycle, all connected to a vertex of degree $n-1$ in the center .

- Random planar graphs: For generating a random planar graph $G = (V, E)$, given a pair of fixed values for the size of $|V|$ and $|E|$, we start with an empty graph such that $E = \emptyset$. Then, we randomly add more edges, while preserving planarity, until we reach the limit.

- The following collection of graphs from the literature: Durer graph, Errera graph, Frucht graph, and Bidiakis cube.

In order to obtain a better analysis of random planar graphs and random sp graphs, for each fixed pair of values for the size of vertices and edges, ten graphs were generated, and the outputs of all of them were taken into account to compute an average.

5.2 Evaluation of the VRM algorithm

This section discusses the success rate and efficiency of the VRM algorithm and its components. The VRM algorithm is implemented in MATLAB2014a. It also employs existing implementations from the MATLAB BGL library [24]. Since we need to feed the visibility representation of the input graph to the VRM algorithm, we have deployed software written by Boyer [1] which is available to public and coded in C++. Given the outline and detailed steps of the algorithm in Chapter 4, we only need to set the number of rounds for the primary optimization method, and it is set to 25. Further, for a better perspective of the random behavior of the algorithm, we run it more than once for each instance, exactly 20 times. Therefore, throughout these experiments, the reported data for any instance is an average of 20 runs.

Considering the input graph $H = (V, E)$, a target grid of minimum size $(|V_H|+2)\times(|E_H|+2)$ is required by the implementation to ensure a successful minor embedding. The extra 2 rows and columns were merely added for a simpler experimental implementation by allowing us to reuse our previous implementations of other algorithms we studied. We will demonstrate shortly that our optimization techniques not only omit these two rows and columns, but also can shrink the embedding (in most cases) to use about or less than half of the $|V_H| \times |E_H|$ grid that is initially assigned to it based on the theoretical upper bound. Therefore, we made the choice in favor of a simpler implementation.

Another issue regarding the implementation concerns the running time of the algorithm. We mentioned the use of priority queues in the primary optimization technique (Algorithm 3), but in the implementation, we have chosen a slightly less efficient approach, in favor of simplicity, but a safe one, because this alteration does not impact the output of the algorithm.

Further, we have run Algorithm O to optimize a couple of instances separately. The results will provide an insight toward the role of each optimization technique.

5.2.1 Results and analysis

Among the measuring criteria, the size of vertex models and embedding area are the primary concerns, and minimizing them is the objective. Therefore, the following plots present the performance of the VRM algorithm in terms of these two factors. As mentioned in the previous chapter, the VRM algorithm is complete, so the probability of its success is not an issue here. We stated in Chapter 4 that the algorithm first attempts to minor embed the input graph with the advanced method, and if it fails, the simple method will surely minor embed the graph. It is noteworthy to mention, in our experiments, the advanced method was always successful in minor embedding the instances.

Figure 5.1 illustrates the results of the VRM algorithm for random Apollonian graphs of 11, 14, 18, and 25 vertices. The optimization techniques reduced the average size of vertex models for all instances which indicates their effectiveness in improving the quality of embedding. Together, they have reduced the sizes up to 54%, and evidently, Algorithm O has been more effective for larger graphs. The gap between the results of the primary optimization method and Algorithm O becomes clearer as the graphs grow larger. As for the embedding area, we have an average 60% decrease by the optimization methods.

Next, the results for grid graphs of 9, 16, 25, 36, and 49 vertices are presented in Figure 5.2. The average 78% reduction in the embedding area is impressive for this family of graphs, especially for larger ones. Here again, the gap between the results of the primary optimization method and Algorithm O grew larger with the size of instances. Figure 4.5 depicts the minor embedding of a grid graph with 49 vertices. The optimization phase was able to reduce its embedding area by 79%.

We generated random SP graphs with 13, 16, 19, 21, and 27 vertices. The algorithm's performance in terms of average size of vertex models and embedding area versus the number of vertices is depicted in Figure 5.3. An instance of this family of graphs with n vertices is a graph with two vertices of degree $n-1$, and $n-2$ vertices of degree 2. This attribute results in a sort of "stairway" structure in their visibility representation, and hence, in the minor embedding. Due to this structure, the optimization algorithm is not able to reduce the area as much as it can for other families. Figure 5.4 illustrates embedding of a random series parallel graph of size 27 before and after optimization with only 30 percent decrease in its embedding area.

The following table, Table 5.1, illustrates two observations regarding the embedding area based on the output of the algorithm. The first row states the average decrease of the embedding area, caused by the primary optimization method alone, for different families of graphs. It is evident that we can achieve at least 50% decrease of embedding area except for the case of the maximal series-parallel graphs with their particular "stairway" structure. The second row in Table 5.1 presents the ratio of the final size of the minor embedding to the initial upper bound based on the input visibility representation. As mentioned before,

Figure 5.1: Apollonian graphs: Average size of vertex models before optimization and after each optimization method, in addition to the area of embedding before and after optimization as a function of the number of vertices

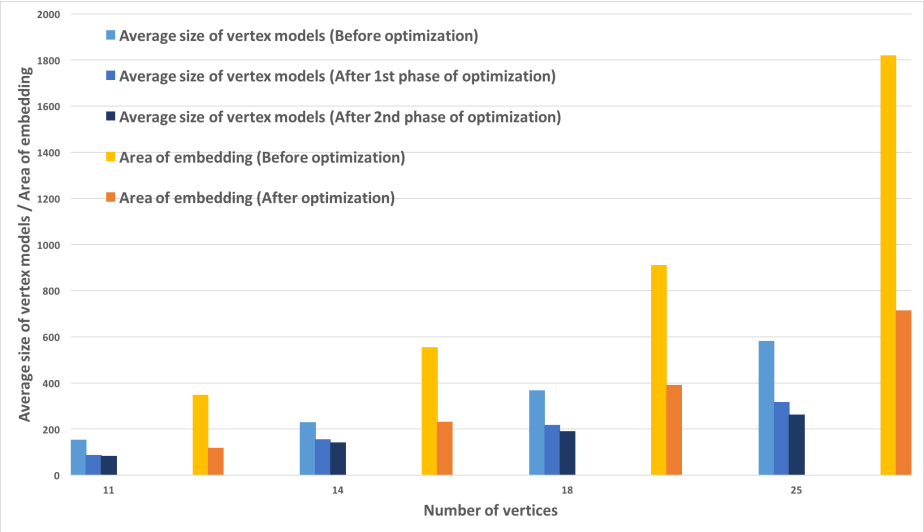
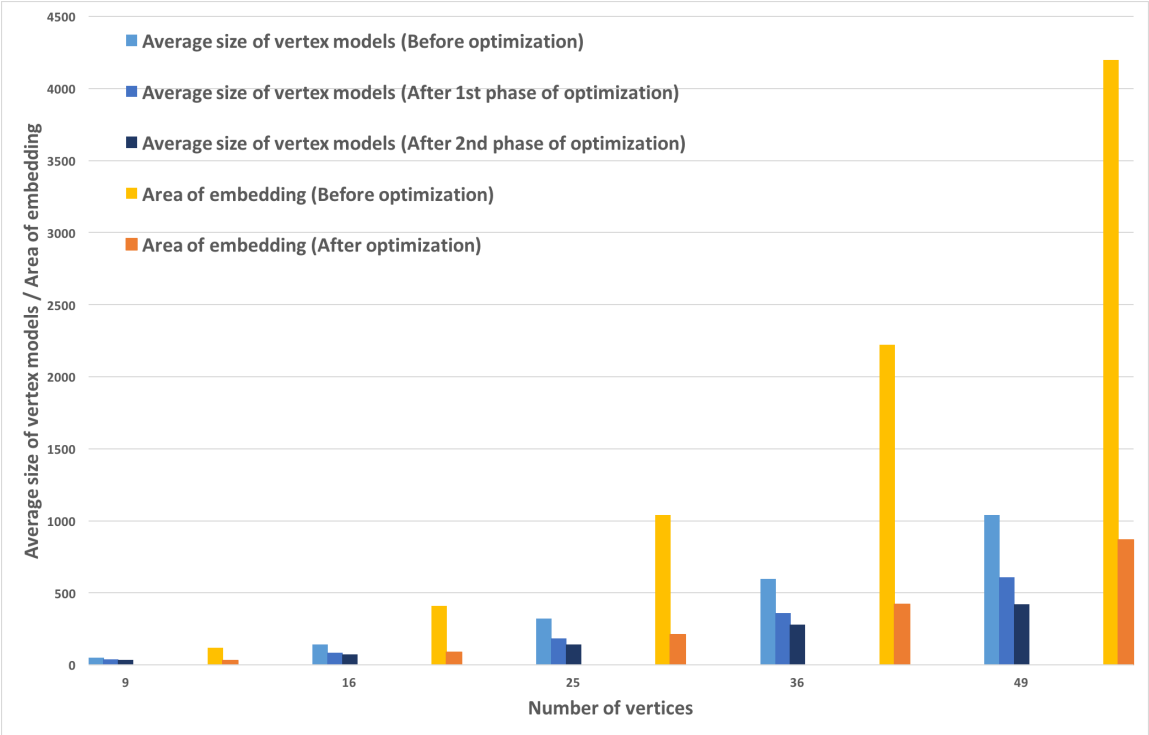


Figure 5.2: Grid graphs: Average size of vertex models before optimization and after each optimization method, in addition to the area of embedding before and after optimization as a function of the number of vertices



for an input planar graph $H = (V, E)$, we can compute its visibility representation which can be embedded in a $|V_H| \times |E_H|$ grid, and hence, $|V_H| \times |E_H|$ is the upper bound on the

Figure 5.3: Random Series Parallel graphs: Average size of vertex models before optimization and after each optimization method, in addition to the area of embedding before and after optimization as a function of the number of vertices

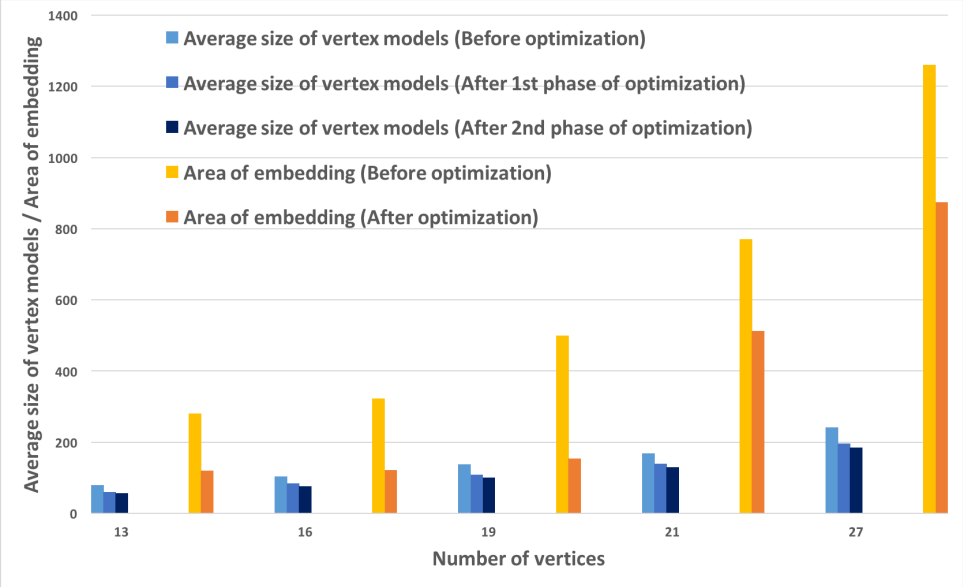
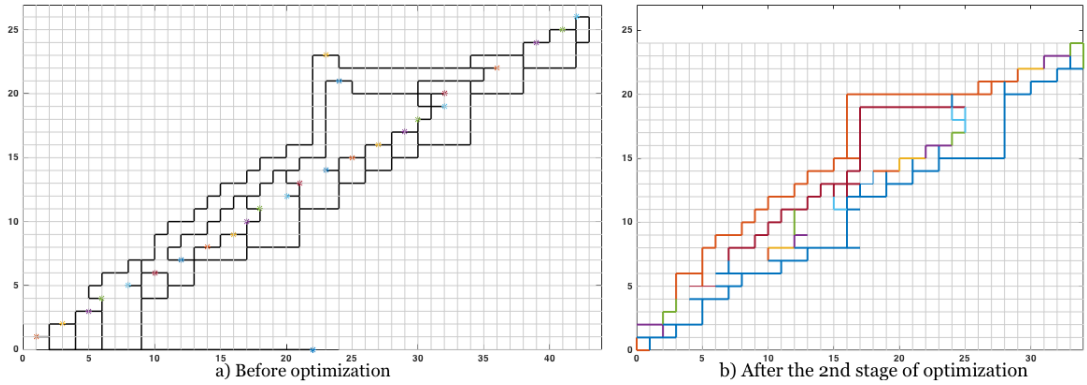


Figure 5.4: The output of the VRM algorithm for a random series parallel graph with 27 vertices. a) Before optimization b) After the second stage of optimization

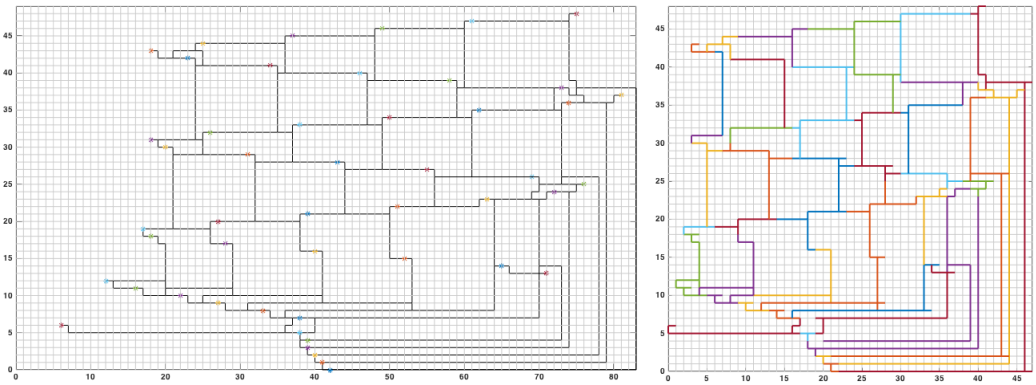


required area for embedding. In other words, the described ratio refers to the ratio of the area of the rectangle enclosing the final embedding (eg. the embedding in Fig. 4.5.c) to the area of the rectangle enclosing the initial embedding (eg. the embedding in Fig. 4.5.a).
To investigate the sole impact of Algorithm O, we have to isolate it from the primary optimization method. Therefore, for a couple of instances, after obtaining a feasible minor embedding, we directly run Algorithm O. For instance, for the 7×7 grid, only a 42% decrease in the embedding area resulted, as depicted in Figure 5.5, versus the 79% achieved by both optimization techniques, depicted in Figure 4.5. Further, Algorithm O was not able to improve the instances of series-parallel graphs due to their specific structure. The primary optimization method is able to improve this type of embedding, despite their struc-

Table 5.1: Average impact of optimization techniques of VRM algorithm on embedding area for instances of each graph family. The numbers in the first row demonstrate the effect of the primary optimization technique in reducing the embedding area in terms of the ratio of decrease after optimization to initial state before optimization. As for the second row, the numbers state the overall impact of both optimization methods to shrink the output embedding for each instance in terms of the ratio of final embedding area to initial state before optimization (which is the upper bound).

Graph Family	Random Apollonian	Grid	Random Planar	Random SP	Maximal SP	Wheel	Named Graphs
Average Area Decrease After Optimization (Percentage)	**60.42**	**78.68**	**70.92**	**50.56**	**32.55**	**52.56**	**67.49**
Average Ratio of Embedding Area to Upper Bound (Percentage)	**46.42**	**23.82**	**33.65**	**56.23**	**74.76**	**47.81**	**36.30**

Figure 5.5: The sole impact of the second optimization method (Algorithm O) on the minor embedding of a 7×7 grid graph. a) Before optimization b) After applying Algorithm O



ture, because it changes the relative placements of vertices and breaks the structure, while preserving feasibility. On the other hand, Algorithm O just shifts vertices on horizontal or vertical lines. The effect of Algorithm O is essential as it tidies up the minor embedding and compresses it as much as it can, but it cannot replace the sophisticated random behavior of the primary optimization method.

5.3 Comparison between Algorithm D and the VRM algorithm

We also implemented Algorithm D [8] in MATLAB2015b to compare its performance with the VRM algorithm. The existing implementations in the BGL library [24] were employed for some graph algorithms, such as A*. The same planar graphs and their corresponding target graphs were given to Algorithm D as input. The size of the target grid graphs are exactly the same as the ones given to the VRM algorithm, meaning they include the two extra rows and columns.

Recall that Algorithm D has two phases, as described in Section 2.5. In the first phase, the input graph is embedded, and in the second phase, Algorithm D attempts to make the embedding feasible and improve it. The second phase is a loop which stops when there is no sign of improvement for t consecutive iterations. In these experiments, t has been set to 10, as in [8].

Further, the description of Algorithm D in [8] was ambiguous in parts. For instance, the multi-source A* was introduced in [8] to place a root vertex and generate its shortest path tree which connects it to its neighbors, as explained in Section 2.5. In order to make sure the implementation obtains the desired goal, which is minimizing the crossings and distance function, in addition to implementing the well-presented sections of [8], we also implemented the Dijkstra algorithm for finding the shortest paths.

Due to the random heuristics in Algorithm D, as for the VRM algorithm, we ran it 25 times for each instance to provide averaged values for size of vertex models and success rate. The success rate refers to the ratio of successful attempts out of 25 runs. We will observe shortly that the success rate of Algorithm D was generally low, and it was only able to embed the smaller graphs. When it did succeed, its embedding was usually smaller than the embedding produced by the VRM algorithm.

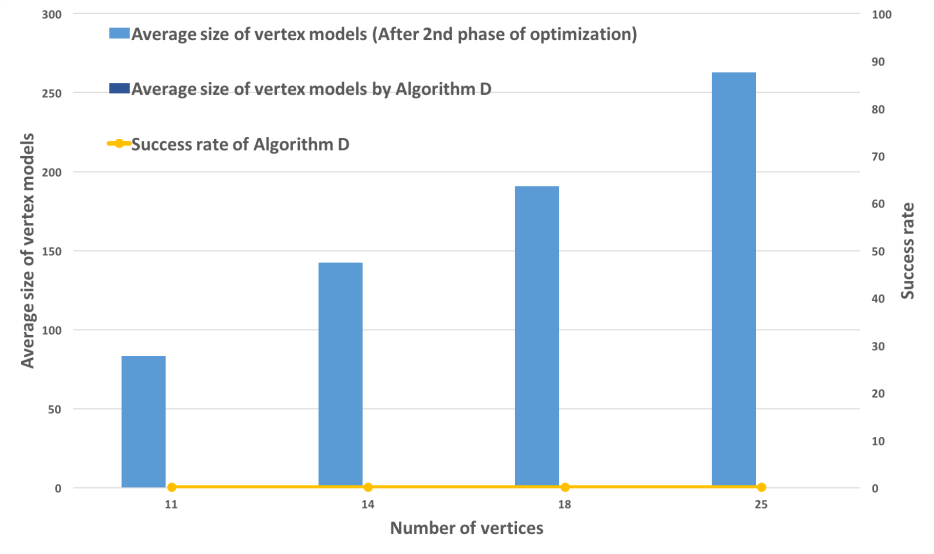
5.3.1 Results and analysis

Algorithm D failed to embed any Apollonian graph, as depicted in Figure 5.6, meaning it was unable to embed this type of graph in a grid of fixed size without crossings.

Figure 5.7 demonstrates the results of Algorithm D for grid graphs. It was able to embed the grids of size 9, 16, and 25, but its success rate dropped drastically from 96% to 8% with the increase in the number of vertices from 9 to 16. Finally, the algorithm was unable to embed grids of 36 or more vertices. On the other hand, it is evident that when Algorithm D succeeds, the size of vertex models are noticeably smaller. In fact, Algorithm D was able to embed the 3×3 grid perfectly (with each vertex mapped to a vertex model of size 1 and each edge to a path of size 1).

Figure 5.8 illustrates the results of Algorithm D for random SP graphs. While being able to

Figure 5.6: Apollonian graphs: Comparison between VRM and Algorithm D on the size of vertex models as a function of the number of vertices



embed the three smaller graphs with 13, 16, and 19 vertices, still the average sizes of vertex models are close to the results obtained by the VRM algorithm. Moreover, the success rate starts from almost 27% and decreases monotonically.

The next table, Table 5.2, gives the average size of vertex models obtained by the VRM algorithm and Algorithm D for instances which Algorithm D was able to embed at least once. The smaller is the graph, the higher is the success rate and the gap between the values obtained by the two algorithms.

As evident in the discussed plots, the greedy nature of Algorithm D, which places each root vertex as close as possible to its already embedded neighbors, makes it more difficult to find feasible embeddings, but when it is successful, the outputs are quite desirable. To analyze the impact of the size of target graphs on the performance of Algorithm D and its probability of success, we repeat the experiments and double the dimensions of the given target graph for each instance, and run the algorithm 25 times. Table 5.3 presents the results. The sets of instances in both tables 5.2 and 5.3 are the same. Based on this fact, we can conclude that providing four times bigger target grids does not lead to non-zero success rates regarding the instances, which Algorithm D failed to embed in the previous conditions (with original target graphs). Therefore, the failure of Algorithm D is not a result of giving it too small a grid.

Table 5.2: Comparison of results obtained by VRM and Algorithm D where Algorithm D was able to minor embed the input graph successfully. The target graph given to Algorithm D is of equal size of the one given to VRM.

Graph Family	No. of Vertices	Avg. Size of Vertex Models Before Optimization (VRM)	Avg. Size of Vertex Models After Optimization (VRM)	Avg. Size of Vertex Models (Algorithm D)	Success Rate (Algorithm D)
Grid	9	49	33.5	19.1	64%
Grid	16	141	73.1	17	8%
Grid	25	322	142.7	30	4%
Wheel	5	24	17	14.5	88%
Wheel	10	70	55.8	24	4%
Maximal Series-Parallel	5	29	14.7	15	68%
Dürer	12	101	66.6	47.5	8%
Frucht	12	83	52.4	56.8	28%
Random Series-Parallel	13	79.90	57.3	50.8	26.8%
Random Series-Parallel	16	104.5	75.6	55.7	19.2%
Random Series-Parallel	19	137.6	101.1	100.5	0.4%
Random Planar	19	144.5	104.6	79	4%

Table 5.3: Comparison of the results obtained by VRM and Algorithm D where Algorithm D was able to minor embed the given input at least once. The target graph given to Algorithm D is four times larger than the one given to VRM.

Graph Family	No. of Vertices	Avg. Size of Vertex Models Before Optimization (VRM)	Avg. Size of Vertex Models After Optimization (VRM)	Avg. Size of Vertex Models (Algorithm D)	Success Rate (Algorithm D)
Grid	9	49	33.5	17.8	68%
Grid	16	141	73.1	74	8%
Grid	25	322	142.7	74.6	12%
Wheel	5	24	17	17.4	88%
Wheel	10	70	55.8	24	4%
Maximal Series-Parallel	5	29	14.7	15.1	72%
Dürer	12	101	68.6	49	48%
Frucht	12	83	52.4	56.8	28%
Random Series-Parallel	13	79.9	57.3	57.7	44.6
Random Series-Parallel	16	104.5	75.6	69.6	38.6
Random Series-Parallel	19	137.6	101.1	133	5.3
Random Planar	19	144.5	104.6	161.9	6.6

Figure 5.7: Grid graphs: Comparison between VRM and Algorithm D on the size of vertex models as a function of the number of vertices

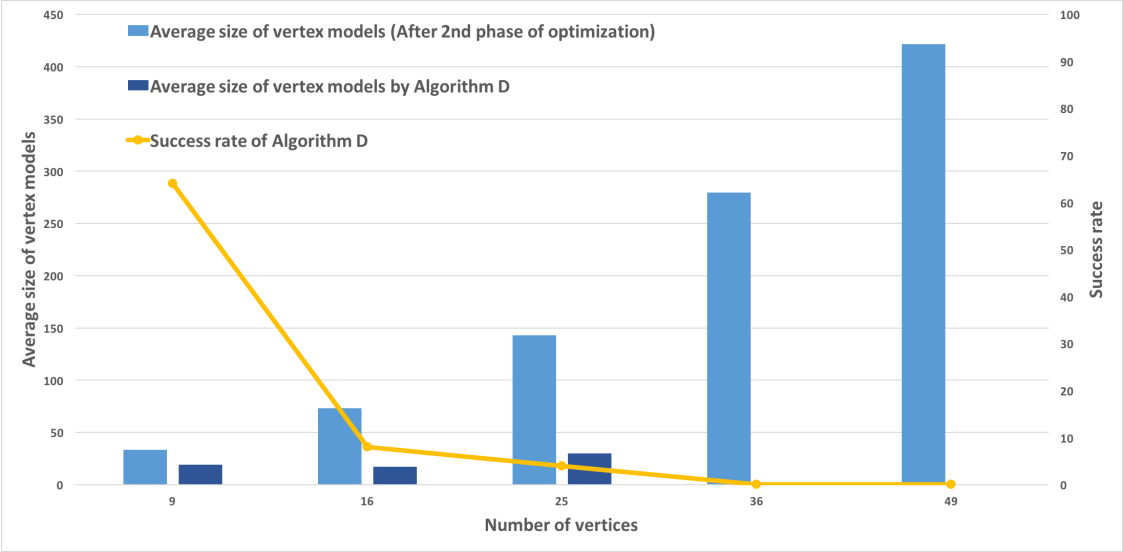
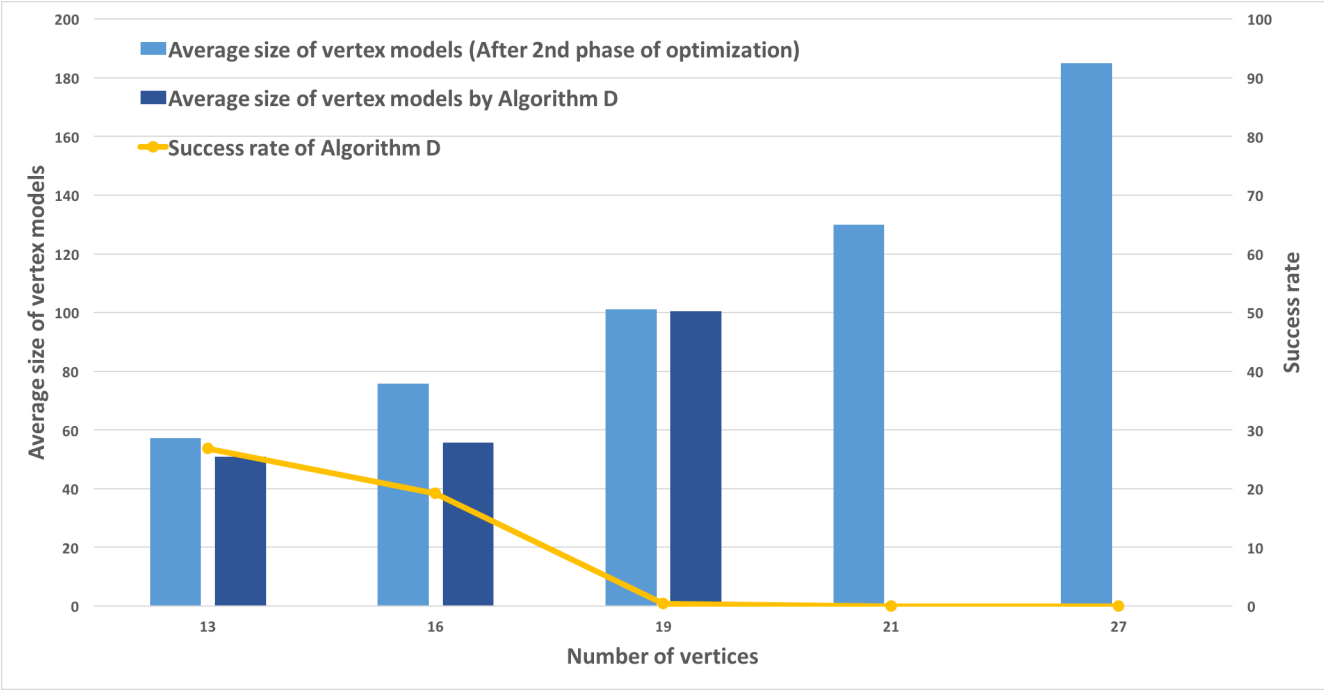


Figure 5.8: Random Series Parallel graphs: Comparison between VRM and Algorithm D on the size of vertex models as a function of the number of vertices



Chapter 6

Conclusion and Future Work

In this thesis, we investigated the problem of minor embedding planar graphs in grid graphs. The introduced algorithms are composed of three sections. The first section of each algorithm pre-processes an input graph using planarity drawing algorithms that provide us with a higher chance to obtain a feasible embedding in a bounded target graph. The second section concerns the minor embedding process where each algorithm uses a unique approach to form the vertex models. The third section of each algorithm is a post-embedding process to improve the quality of the embedding.

We presented two heuristic algorithms. The first algorithm (SLM) minor embeds a planar graph H of n vertices in a $9(2n-4) \times (n-2)$ grid. The SLM algorithm uses a straight line drawing algorithm to place the root vertices on the grid. Then, the path finding algorithm A* connects each vertex to its neighbors below. We used perimeters and gateways to individualize the vertex models. We also explained the impact of the choice of heuristic in the A* algorithm on the output minor embedding and optimization process. The introduced optimization process, Algorithm O, works better with Manhattan heuristic. The output of the SLM algorithm suffers from unnecessary long paths and dependency on grid structure. Based on these conditions and the inconsistent success of SLM in minor embedding planar graphs, we decided to pursue another approach.

The second algorithm (VRM) minor embeds a planar graph H in a $|V_H| \times |E_H|$ grid. We use an already existing implementation to compute the visibility representation of the input graph in linear time. The VRM algorithm processes a visibility representation in two phases: minor embedding and optimization. The embedding phase is carried out efficiently due to the introduced technique, the dynamic arm assignment, for managing the vertex models. Unlike the first algorithm, this technique enables us to defer the final assignment of vertices to vertex models to gather more information regarding the input graph structure. This technique can be used in any algorithm where we have the root vertices and want to connect them using a path finding algorithm. After obtaining a feasible embedding, we apply the two presented optimization methods. One optimization method uses the dynamic arm

assignment technique to re-embed the vertices closer to each other. The other optimization method shrinks the embedding by eliminating the unused rows and columns. These optimization methods are able to improve the embedding by reducing both the size of vertex models and the area of embedding. We compared the VRM algorithm with an existing minor embedding algorithm, Algorithm D. The VRM algorithm outperformed Algorithm D in terms of consistency in obtaining feasible minor embedding for different families of graphs and various sizes. Algorithm D was only able to embed the small instances.

For future extension of the work in this thesis, the following ideas are suggested:

- Recall the two main components of the Chrobak-Payne straight line drawing algorithm: the canonical ordering and the shifting sets. In Chapter 3, we employed the output coordinates of the straight line drawing and the canonical ordering to minor embed planar graphs. The coordinates proved to be undesirable due to the long paths in the minor embedding. Therefore, we suggest using the combination of the shifting sets, as a horizontal order of vertices, in addition to the canonical ordering, as a vertical order. These two can guide the minor embedding process toward obtaining a feasible embedding without keeping the restrictions of the straight line drawing. Further, the dynamic arm assignment is suggested to be employed in any future minor embedding algorithm because of its efficiency in forming the vertex models.

- In implementing the two introduced algorithms, our focus was mainly on correctness and success rather than efficiency of the running time. Use of appropriated data structures and pre-computations are suggested to improve both algorithms, particularly the optimization phase of the VRM algorithm .

- As mentioned in Section 4.4, regarding the primary optimization method, before re-embedding a vertex v, we eliminate both permanent and non-permanent parts of its vertex model. We also eliminate the sections shared between v and its neighbors from the neighbors' vertex models. We did not pursue the impact of this modification on the permanent sections of these vertex models. The nodes are permanently assigned to a vertex model $\phi(v)$ when they have been used to connect the root vertex r_v to more than one neighbor. When the neighbors change their locations, the permanent section of $\phi(v)$ might not be useful anymore for connecting r_v to its neighbors. These *extra* vertices might be included in several vertex models throughout the optimization. We suggest two approaches to address this issue. One option would be to add a post-processing step to identify and remove these extra vertices. Another option would to modify the primary optimization method to update the vertices during the process. These updates are based on the current number of vertices using the permanent section of a vertex model. If that section is used to connect only two vertices, its labels should change to non-permanent.

Bibliography

[1] Edge addition planarity algorithm. `https://code.google.com/archive/p/planarity/`. Accessed: 2015-04-11.

[2] Isolde Adler, Frederic Dorn, Fedor V. Fomin, Ignasi Sau, and Dimitrios M. Thilikos. Faster parameterized algorithms for minor containment. *Theoretical Computer Science*, 412(50):7018 – 7028, 2011.

[3] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335 – 379, 1976.

[4] Tomas Boothby, Andrew D. King, and Aidan Roy. Fast clique minor generation in chimera qubit connectivity graphs. *Quantum Information Processing*, 15(1):495–508, 2015.

[5] John Boyer and Wendy Myrvold. Stop minding your p's and q's: A simplified O(n) planar embedding algorithm. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 140–146, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[6] John M. Boyer. *Graph Drawing: 13th International Symposium, GD 2005, Limerick, Ireland, September 12-14, 2005. Revised Papers*, chapter A New Method for Efficiently Generating Planar Graph Visibility Representations, pages 508–511. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[7] John M. Boyer and Wendy J. Myrvold. On the cutting edge: simplified planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[8] Jun Cai, William G. Macready, and Aidan Roy. A practical heuristic for finding graph minors, 2014.

[9] C. C. Cheng, C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Drawing planar graphs with circular arcs. *Discrete Comput. Geom.*, 25(3):405–418, April 2001.

[10] Norishige Chiba, Kazunori Onoguchi, and Takao Nishizeki. Drawing plane graphs nicely. *Acta Informatica*, 22(2):187–201.

[11] Norishige Chiba, Tadashi Yamanouchi, and Takao Nishizeki. Linear algorithms for convex drawings of planar graphs. *Progress in graph theory*, pages 153–173, 1984.

[12] Vicky Choi. Minor-embedding in adiabatic quantum computation: I. the parameter setting problem. *Quantum Information Processing*, 7(5):193–209, October 2008.

[13] Vicky Choi. Minor-embedding in adiabatic quantum computation: II. minor-universal graph design. *Quantum Information Processing*, 10(3):343–353, June 2011.

[14] M. Chrobak and T. H. Payne. A linear-time algorithm for drawing a planar graph on a grid. *Inf. Process. Lett.*, 54(4):241–246, May 1995.

[15] Sabine Cornelsen and Andreas Karrenbauer. *Graph Drawing: 19th International Symposium, GD 2011, Eindhoven, The Netherlands, September 21-23, 2011, Revised Selected Papers*, chapter Accelerated Bend Minimization, pages 111–122. Springer, Berlin, Heidelberg, 2012.

[16] Sanjeeb Dash. A note on qubo instances defined on chimera graphs. `arXiv:1306.1202`, 2013.

[17] Hubert de Fraysseix, János Pach, and Richard Pollack. Small sets supporting fary embeddings of planar graphs. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 426–433, New York, NY, USA, 1988. ACM.

[18] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 3rd edition, August 2005.

[19] P. Duchet, Y. Hamidoune, M. Las Vergnas, and H. Meyniel. Representing a planar graph by vertical lines joining different levels. *Discrete Mathematics*, 46(3):319 – 321, 1983.

[20] Christian Duncan and Stephen Kobourov. Polar coordinate drawing of planar graphs with good angular resolution. *Journal of Graph Algorithms and Applications*, 7(4):311–333, 2003.

[21] Shimon Even and Robert Endre Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2(3):339 – 344, 1976.

[22] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[23] Ashim Garg and Roberto Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In *Proc. of Graph Drawing 1996, Lect. Notes in Computer Science*, pages 201–216. Springer-Verlag, 1996.

[24] David Gleich. Matlabbgl. `http://www.mathworks.com/matlabcentral/fileexchange/10922-matlabbgl`, 2008.

[25] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications, Second Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2005.

[26] Carsten Gutwenger and Petra Mutzel. *Graph Drawing: 6th International Symposium, GD' 98 Montréal, Canada, August 13–15, 1998 Proceedings*, chapter Planar Polyline Drawings with Good Angular Resolution, pages 167–182. Springer, Berlin, Heidelberg, 1998.

[27] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[28] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[29] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, October 1974.

[30] Wilfried Imrich, Sandi Klavzar, and Douglas F. Rall. *Topics in Graph Theory: Graphs and Their Cartesian Product*. AK Peters Ltd, 2008.

[31] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996.

[32] Kenichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424 – 435, 2012.

[33] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[34] Christine Klymko, Blair D. Sullivan, and Travis S. Humble. Adiabatic quantum programming: Minor embedding with hard faults, 2012.

[35] A. Lempel, S. Even, and I. Cederbaum. *An algorithm for planarity testing of graphs*, pages 215–232. Gordon and Breach, New York, 1967.

[36] Xuemin Lin and Peter Eades. *Computing and Combinatorics: 5th Annual International Conference, COCOON'99 Tokyo, Japan*, chapter Area Minimization for Grid Visibility Representation of Hierarchically Planar Graphs, pages 92–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[37] Catherine C. McGeoch and Cong Wang. Experimental evaluation of an adiabatic quantum system for combinatorial optimization. In *Proceedings of the 2013 ACM Conference on Computing Frontiers*, May 2013.

[38] R. H. J. M. Otten and J. G. Van Wijk. Graph representations in interactive layout design. In *Proceedings of IEEE International Symposium on Circuits and Systems*, pages 914– 918, 1978.

[39] N. Robertson and P.D. Seymour. Graph minors XIII. the disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65 – 110, 1995.

[40] Pierre Rosenstiehl and Robert E. Tarjan. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete & Computational Geometry*, 1(4):343–353, 1986.

[41] Walter Schnyder. Embedding planar graphs on the grid. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 138–148, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[42] P. D. Seymour and R. W. Weaver. A generalization of chordal graphs. *Journal of Graph Theory*, 8(2):241–251, 1984.

[43] Wei-Kuan Shih and Wen-Lian Hsu. A simple test for planar graphs. In *Proceedings of the International Workshop on Discrete Mathematics and Algorithms*, pages 110–122, 1993.

[44] Wei-Kuan Shih and Wen-Lian Hsu. A new planarity test. *Theor. Comput. Sci.*, 223(1-2):179–191, July 1999.

[45] T. Stutzle and H. Hoos. Max-min ant system and local search for the traveling salesman problem. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 309–314, Apr 1997.

[46] L. F. Takacs. Lattice path counting and applications (Sri Gopal Mohanty). *SIAM Review*, 25(4):592–593, 1983.

[47] R. Tamassia and I.G. Tollis. Planar grid embedding in linear time. *Circuits and Systems, IEEE Transactions on*, 36(9):1230–1234, Sep 1989.

[48] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, June 1987.

[49] Roberto Tamassia. *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2007.

[50] Roberto Tamassia and Ioannis G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete & Computational Geometry*, 1(4):321–341, 1986.

[51] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[52] Ehsan Tavakoli. Minor embedding planar graphs in grid graphs. Master's thesis, Simon Fraser University, May 2016.

[53] William Thomas Tutte. How to draw a graph. *Proc. London Math. Soc*, 13(3):743–768, 1963.

[54] K. Wagner. Uber eine eigenschaft der ebenen komplexe. *Mathematische Annalen*, 114(1):570–590, 1937.

[55] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2nd edition, September 2000.

[56] S. G. Williamson. Embedding graphs in the plane - algorithmic aspects. In *Combinatorial Mathematics, Optimal Designs and Their Applications*, volume 6 of *Annals of Discrete Mathematics*, pages 349 – 384. Elsevier, 1980.

[57] Arman Zaribafiyan, Dominic J. Marchand, and Seyed S. Changiz Rezaei. Systematic and deterministic graph-minor embedding for cartesian products of graphs, 2016.

[58] Huaming Zhang and Xin He. Improved visibility representation of plane graphs. *Computational Geometry*, 30(1):29 – 39, 2005.