CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Embedding of Rigid Graphs in Grids

A Fast Special Case of Grid Embedding

Master's thesis in Computer science and engineering

Jesper Jaxing
Maxim Goretskyy

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Embedding of Rigid Graphs in Grids

A Fast Special Case of Grid Embedding

Jesper Jaxing
Maxim Goretskyy

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Embedding of Rigid Graphs in Grids
A Fast Special Case of Grid Embedding
Jesper Jaxing
Maxim Goretskyy

Supervisor: Peter Damaschke, Department of Computer Science and Engineering
Examiner: Graham Kemp, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Embedding of Rigid Graphs in Grids
A Fast Special Case of Grid Embedding
Jesper Jaxing
Maxim Goretskyy
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis concerns the problem of finding a unit embedding of a graph in a square grid. We introduce the concept of a rigid graph, which is a graph with a unique unit embedding. We present a few polynomial algorithms for embedding large rigid subgraphs of an arbitrary graph.

Keywords: Computer, science, factory layout, grid embedding, unit embedding, graph, theory, rigid, graphs, grid.

Acknowledgements

We want to thank Peter Damaschke for always going out of his way in order to help us throughout this project. Words cannot describe how much he has helped us, but math symbols can: ∞.
We also want to extend our thanks to Peter Lindh from Squeed AB for giving us moral support throughout the project as well as providing us with office space.

Jesper Jaxing, Gothenburg, June 2019
Maxim Goretskyy, Gothenburg, June 2019

Contents

List of Figures

List of Algorithms

1
Introduction

According to the Fraunhofer-Chalmers Research Centre for Industrial Mathematics (FCC), the design of optimal factory layouts, i.e. how to place machines with respect to each other, is still an active area of development with strong needs from industry. An optimal factory layout can have many different requirements, each with its own set of complex problems [1]. One such requirement, according to our supervisor who collaborates with FCC, is to minimise the time a product spends between machines. To minimise the time between machines is a complex requirement in itself. Additionally, there can be many choices for how to connect the machines and a real-world application might require a dynamic layout such that it is possible to replace machines and have machines of different sizes.

This chapter will introduce the problem statement, delimitations of the thesis, related work to put the thesis into a greater context and ethical considerations that come with this thesis.

1.1 Background

A factory layout can be represented as a graph, where the nodes are machines and the edges represent the dependencies between machines. A layout is optimal if it minimises the length of all edges. The theoretical minimum is then any layout that is unit embedded in a 2-D square grid. A unit embedding can be seen as a one to one mapping of edges from a given graph onto a part of the grid, for more details

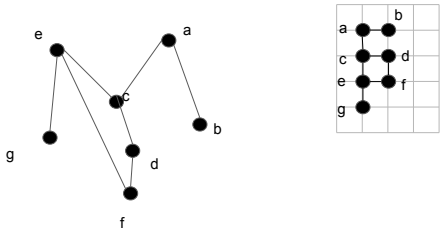


Figure 1.1: An example of mapping a graph into a grid. This embedding is a unit embedding, which makes the layout optimal since the sum of all edge lengths equals the number of edges.

see Section 2.4. Then the total length equals the number of edges as can be seen in Figure 1.1.

Placing machines in a factory and placing electrical components on a circuit can both be described as a dependency graph. From a graph theoretical standpoint, this makes the two problems similar and both are an instance of the layout planning problem [6].

The problem of placing electrical components on a circuit was examined by Bhatt and Cosmadakis [2]. They tried to find a unit embedding of a graph in a grid and showed that this problem is NP-complete. However, a more recent paper by de Sá et al. showed that for specific graphs there exist polynomial algorithms for finding an embedding of a graph in a grid [5]. Unfortunately, those specific graphs are not general enough for practical uses.

1.2 Problem

This thesis presents a class of graphs, rigid graphs, that only have one unit embedding in the grid. The reason to define these graphs is that for rigid graphs we do not need to consider a large number of embeddings as there is only one. If the entire graph is not rigid, a large subgraph that is rigid might still be found and embedded. The idea is that it might be practically feasible to find an embedding of the remaining smaller graph by brute force. The algorithms for embedding rigid graphs and some properties for rigid graphs are presented in Chapter 3.

This thesis aims to answer the following questions:

- Is it possible to completely characterise rigid graphs, in the context of 2D-grid embeddings?
- Is it possible to recognise rigid graphs in polynomial time?
- Is it possible to recognise large rigid subgraphs of arbitrary graphs?
- Is it possible to develop a polynomial or better-than-exponential embedding algorithm for rigid graphs?

1.2.1 Delimitations

We will limit the project to the following scope:

- We will ignore the dynamic constraints of factory layouts and instead assume that we know beforehand all of the machine dependencies and that they do not change over time.
- Machines in the layout will have the same size, meaning each machine is represented by exactly one node.
- We will only consider unit embeddings into square grids, see Section 2.3.
- All graphs will be connected. Otherwise, the components can be solved independently.

1.3 Ethical Considerations

The algorithms and theory that we will develop cannot be unethical, however, ethics are relevant for how the algorithms are used. To develop the algorithms we make many simplifications from reality, and our optimisation goal might not be the only important factor when designing a factory. For instance, the algorithm does not consider ergonomics of human workers, accessibility of machines for maintenance, or that the time between machines may serve as a cooling period making the product more durable. Therefore, the results of optimisation should be used for guidance rather than an absolute truth.

2
Preliminaries

This chapter introduces the necessary theoretical concepts and definitions. We assume that the reader is familiar with general graph theory and time complexity analysis.

2.1 Embedding

An embedding is a mapping of the nodes and edges from a graph onto a surface, such that each node corresponds to exactly one point on the surface and there is a curve between the points if there is an edge between the corresponding nodes in the graph.

2.2 Planar Graphs

A graph is planar if it is possible to embed it in a plane without edges crossing each other. Such an embedding is called a *planar embedding*.

2.3 Grid Graph

A grid graph of size $M \times N$, where $M, N \in \mathbb{Z}^+$, is a graph $G = (V, E)$ such that:

$$V = \{(i, j) : 1 \leq i \leq M, 1 \leq j \leq N\}$$

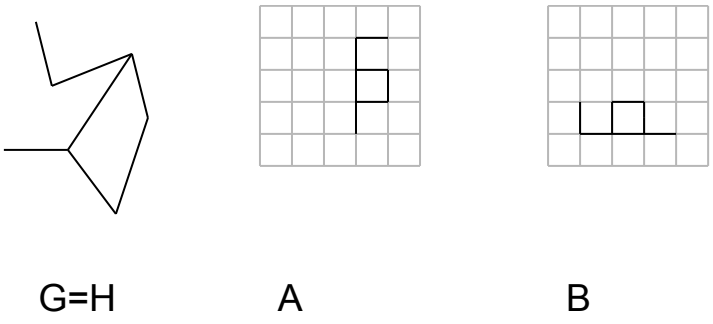
and

$$E = \{(i, j)(k, l) : |i - k| + |j - l| = 1, (i, j), (k, l) \in V\}$$

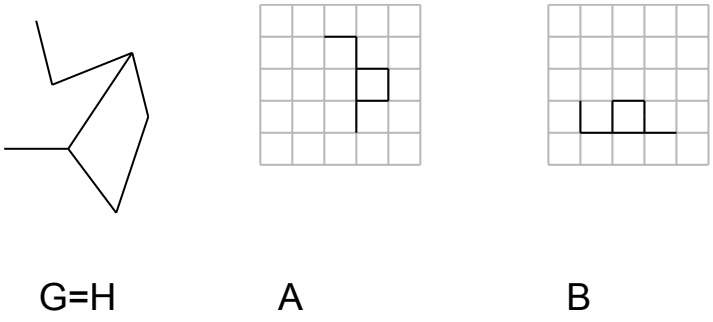
We will use grid and grid graph interchangeably. From this definition it follows that every node in a grid has at most four neighbours and the grid is planar. It is also easy to show that a grid is bipartite. Any subgraph of a grid is a *partial grid graph*.

2.4 Unit Embedding in a Grid

Let G be a graph and Gr a grid graph. A *unit embedding* is a mapping of the nodes in G to nodes in Gr such that if there is an edge between two nodes in G then the corresponding nodes in Gr are also neighbours. This is only possible if G is isomorphic to some partial grid graph.



(a) Two equivalent embeddings of the same graph



(b) Two different embeddings of the same graph

Figure 2.1: Examples of embeddings.

Two embeddings A and B of graphs G and H respectively are equivalent if $G = H$ and B is a rotation, reflection or translation of A. See Figure 2.1 for examples of equivalence of embeddings.

To denote that a node u is embedded at coordinates (x, y) we will write $u = (x, y)$.

2.5 Parameterized Complexity

There are no known fast algorithms for NP-complete or NP-hard problems. It is impossible to find an optimal solution to these problems in polynomial time, unless P=NP. Sometimes, however, it is possible to parameterize the input and move the hard part of the problem to this new parameter. Two complexity classes where this is done are the XP and FPT (fixed-parameter tractable) classes.

For a problem to belong to XP there must be an algorithm with complexity $O(n^{f(k)})$, where n is the input size, f is any function and k is a parameter.

The complexity of a problem in FPT, $O(p(n) \cdot f(k))$, consists of two functions $p(n)$ and $f(k)$, where p is any polynomial function, n the input size, f is any function and k is a parameter.

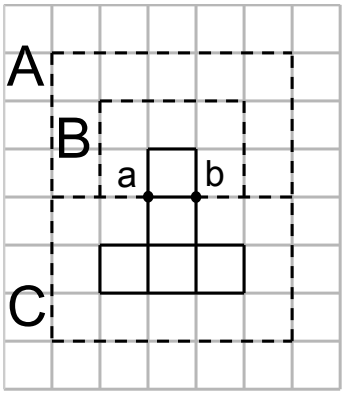


Figure 2.2: An example of some paths in the grid that can extend a graph between a and b. The original graph is drawn with solid lines and the paths, A, B, and C, that can extend it are drawn with dotted lines. A and C are not unique paths but B is.

2.6 Definitions

In this section, we will present definitions that will be used in proofs later on.

2.6.1 Rigid Graph

We define the central concept of a rigid graph as a partial grid graph that has a unique embedding. Some subgraphs of a rigid graph may also be rigid. We say that a graph G is a *minimal rigid graph* if all subgraphs of G that have more than two nodes are non-rigid.

Since a rigid graph is a partial grid graph, this thesis will only consider input graphs that fulfil the necessary conditions for partial grid graphs. These conditions are that the graph is planar, bipartite and has maximum node degree of four. The planarity testing is known to have a $O(n)$ algorithm [3], and bipartiteness can be checked by either BFS (breadth-first search) or DFS (depth-first search). These checks are considered a pre-processing step for the algorithms later in the thesis.

2.6.2 Path Length

We define the path length as the number of edges rather than the number of nodes.

2.6.3 Unique Path

We call a path P of length L between two nodes u and v in G unique if there are no other paths between u and v of length L, see Figure 2.2.

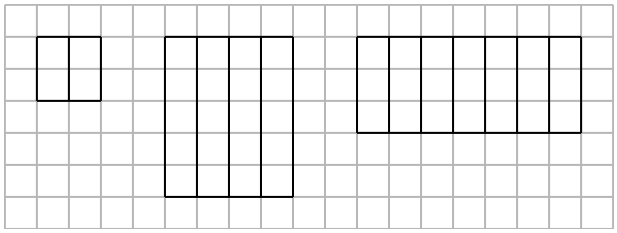


Figure 2.3: Example of a few cycle stacks.

2.6.4 Corner

A corner $u = (x, y)$ is a node with exactly two neighbours $v = (x, y \pm 1)$ and $w = (x \pm 1, y)$.

2.6.5 Border

Let H be a rigid graph and let P be a path in H, such that $u = (x, y)$ and $v = (x+c, y)$ are corners and endpoints of P. P is a border in the embedding of H if and only if the i^{th} node w in P is embedded at $(x+i, y)$ such that there is also a node embedded at $(x+i, y-1)$ for $i = 0...c$. Note that because an embedding does not change when it is rotated, this definition includes both vertical and horizontal borders.

2.6.6 Cycle Stacks

We define an $n \times m$-cycle stack as a series of $m > 1$ cycles $A_1, A_2, ..., A_m$, each consisting of $2n$ nodes. For $1 \leq i \leq m$, every cycle A_i consists of two paths P_i and Q_i that have n nodes each and are disjoint, and $Q_i = P_{i+1}$. See Figure 2.3 for examples.

3
Results

In this chapter, we describe the difficulties of finding general characteristics for rigid graphs and present solutions to the problem of finding large rigid subgraphs.

3.1 Characteristics of a Rigid Graph

To fully characterize rigid graphs is hard, because the rigidity of a graph does not, in general, depend on local properties such as neighbours of a node or surrounding cycles but rather global properties. This can be seen in some minimal rigid graphs, for an example see Figure 3.7. Just looking at some part of the graph is not enough to determine if it is rigid. Details of the rigidity proof of the graph can be seen in Section 3.3.

3.2 A Recursive Approach to Rigid Graphs

Without characteristics of rigid graphs, general algorithms for recognising and embedding any rigid graph will be hard to achieve. Therefore we focus on finding large rigid subgraphs of any given graph G. This section will start by presenting methods for recursively extending a rigid subgraph of the input graph by at most a fixed number of nodes. Later, other methods will be presented that no longer depend on a fixed number of nodes.

3.2.1 Extending by k Nodes

As mentioned above, this section will present the idea of recursively extending H, a rigid subgraph with a known embedding of the input graph G. H is extended by at most k nodes from G such that the new graph is still rigid, where k is a small fixed number.

There is a brute force solution implemented by Algorithm 1 that generates all rigid subgraphs of G by extending H. Algorithm 1 uses Algorithm 2 to create all ways to extend H by at most k nodes and generate all embeddings of these extended graphs. These embeddings will be compared. If there are two or more embeddings of the same graph, clearly that graph is not rigid. The exact implementation of comparisons will be left to the implementer, and we will use the function $\Lambda(m)$, where m is the number of embeddings to compare, to denote the time complexity of the comparison. Algorithm 1 then recursively adds k more nodes to each of the

Algorithm 1: Brute force algorithm: $\alpha(G, H, k)$

Input: G, $H \subseteq G$ where H is rigid and has a known embedding, k
Output: X, a set of embeddings for rigid subgraphs of G

```
Generate all embeddings of supergraphs of H adding at most k nodes, using
  Algorithm 2.
if No supergraphs are generated then
    return {H}
Remove all supergraphs that are not rigid.
X ← {}
foreach supergraphs H' do
    X ← X ∪ α(G, H', k)
return X
```

Algorithm 2: All Supergraphs: $\omega(G, H, k)$

Input: $G = (V, E)$, $H = (V', E') \subseteq G$ where H is rigid and has a known embedding, k
Output: X, set of supergraphs of H

```
foreach node v ∈ V' do
    N ← the set of neighbours of v ⊆ V\V'
    foreach u ∈ N do
        foreach direction d ∈ {LEFT, RIGHT, UP, DOWN} do
            H' ← embed u in H to v in direction d
            X ← X ∪ {H'}
            if k ≥ 1 then
                X ← X ∪ ω(G, H', k − 1)
return X
```

rigid graphs, and terminates when it is no longer possible to create any rigid graph from a given H using at most k nodes from G.

Step 5 in Algorithm 2 says "embed u in H to v in direction d". This attempts to add a node u to an embedding of H at node v, in a direction d (up, down, left, or right), see Figure 3.1. The following algorithms will use the same language when attempting to embed nodes. The algorithms skip an embedding if it is not possible to embed u in direction d. We also add all edges from u to its neighbours in H to the new graph. To make sure that embedding u yields a valid unit embedding we need to check that all neighbours of u in H have a distance one to u.

Lemma 3.2.1. *Algorithm 2 runs in $O(n^k)$ time.*

Proof. For the first call of the function the algorithm will reach Step 8 $O(3 \cdot 3 \cdot n)$ times. The first three comes from Step 3 which denotes the number of neighbours which can be embedded. Although a node can have four, the algorithm will at most consider three neighbours, as the node is always connected to at least one other node. The second three comes from the fact that we have at most three directions to choose from since one direction is already taken by the neighbour the graph was extended from. The variable n is denoting the loop in Step 1. The calls of the function can be visualised as a tree where each node is a call and will have $O(n)$ child nodes. The tree will be of depth k as that parameter is reduced by one each level. The total number of nodes in the tree is then $O(n^k)$. □

The problem with the brute force solution is that it generates too many graphs at each recursive step. Algorithm 2 generates $O(n^k)$ new rigid subgraphs each time it is called as we proved in Lemma 3.2.1. If the calls of Algorithm 1 were visualised as a tree each node would have $O(n^k)$ child nodes. The tree would have a depth of $O(n)$, making the final complexity of Algorithm 1 $O((n^k + \Lambda(n^k))^n)$.

A natural next step to improve the complexity is to reduce the number of supergraphs produced at each level. We propose a few algorithms that implement this idea. They will be introduced in the order of most to least general in terms of which rigid supergraphs they will recognise. The gain from lost generality will be improved time complexity.

We will iterate on Algorithm 3 to improve the time complexity. Firstly, the algorithm tests all nodes from H as start nodes. For each start node it generates some connected subgraph of G that extends H and embeds them. Exactly how these graphs will be generated will be determined by which of the Algorithms 4, 6 and 7 that is used. Like in Algorithm 1, Algorithm 3 terminates when it is not possible to generate more rigid supergraphs.

To prevent the combinatorial explosion from Algorithm 1, this algorithm tries to merge all the generated graphs. This is done at step 6 by taking the union of all the graphs. The correctness of this step comes from Theorem 3.2.2. However, sometimes it is not possible to merge two rigid graphs, for example when the input graph is not a partial grid graph, in which case the algorithm is terminated. By introducing the

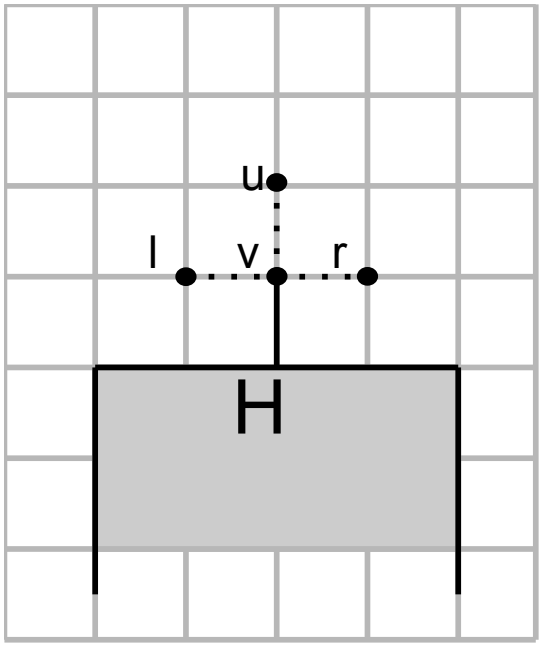


Figure 3.1: H is a rigid graph that will be extended with a node c, from an existing node v in H. c can be placed in three different directions d in H, l (Left), r (Right), u (Up), each of which results in a different embedding.

Algorithm 3: Template algorithm: $\psi(G, H, k)$

Input: G, $H \subseteq G$ where H is rigid and has a known embedding, k
Output: A rigid subgraph of G, with an embedding

```
foreach node v ∈ H do
    Generate and embed supergraphs of H, starting from v.
    Remove all supergraphs that are not rigid.
if No supergraphs are found then
    return H
Merge the generated graphs
if There is more than one rigid graph after merge then
    H' ← largest of the remaining graphs
    return H'
else
    H' ← result of merge
    return ψ(G, H', k)
```

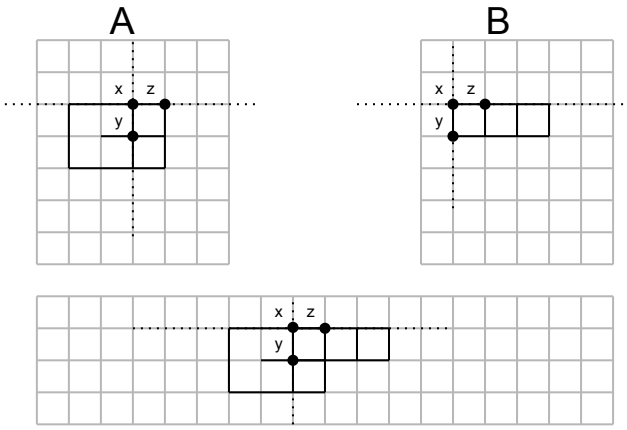


Figure 3.2: Illustrates the merging of two rigid graphs. The dotted lines show that x and y have the same horizontal position and that x and z have the same vertical position. The result of the merge is the embedding at the bottom.

merging step, and substituting Steps 1-2 of Algorithm 3 by a call to Algorithm 2, Algorithm 3 shows a huge improvement in complexity, $O(n(n^k + \Lambda(n^k)))$ compared to $O((n^k + \Lambda(n^k))^n)$ for Algorithm 1.

Theorem 3.2.2. *Let A and B be two rigid subgraphs of a partial grid graph G. Then the union of A and B is also a rigid graph if the intersection has at least three nodes such that these nodes are embedded so no line goes through all three. See Figure 3.2.*

Proof. Let A and B be two rigid graphs sharing at least nodes x, y, and z, such that x, y, and z define two non-parallel lines in the embeddings of both A and B. Then x, y, and z define a plane, containing the embeddings of both A and B. If we transform the embedding A with any of the allowed transformations, see Section 2.4, one or more of x, y, or z will be moved and thereby changing the plane and by extension the embedding of B. □

Algorithm 4 generates all connected supergraphs such that the k nodes form a connected graph on their own. The idea is to go from an XP algorithm, Algorithm 2, to an FPT algorithm by reducing the number of nodes to choose from. Given a starting node from Algorithm 3, Algorithm 4 will try to embed each of the node's neighbours that are not in H, creating one set C for each neighbour. It then recursively increases the size of C by choosing a node from C and tries to embed all of its neighbours that are not in H. It stops when the size of C is k. At each recursive step there will be $O(k)$ number of possible starting nodes to extend H from, as opposed to $O(n)$ for Algorithm 2. By only choosing from the nodes added to H and not the whole subgraph, Algorithm 4 takes $O(k^k)$ time, and the total time of Algorithm 3 is $O(n(n \cdot k^k + \Lambda(n \cdot k^k)))$.

Algorithm 2 and 4 consider a large number of nodes at each step. An alternative solution, though less general, would be to take the next node from the neighbours of the previously chosen node. This idea is implemented in Algorithm 6, which will cre-

Algorithm 4: Wrapper k-Connected

Input: $G = (V, E)$, $H = (V', E') \subseteq G$ where H is rigid and has a known embedding, $v \in V'$, k

Output: X, set of supergraphs of H using k nodes from G

```
1  H' ← H
2  if k is 0 then
3      return {H'}
4  N ← set of neighbours of v ⊆ V\V'
5  foreach u ∈ N do
6      C ← {u}
7      foreach direction d ∈ {UP, LEFT, RIGHT, DOWN} do
8          H' ← embed u in H' to v in direction d
9          X ← X ∪ {H'} ∪ χ(G, H', C, k − 1) using Algorithm 5
10 return X
```

Algorithm 5: k-Connected: $\chi(G, H, C, k)$

Input: $G = (V, E)$, $H = (V', E') \subseteq G$ where H is rigid and has a known embedding, C set of added nodes, k

Output: X, set of supergraphs of H using k nodes from G

```
1  H' ← H
2  if k is 0 then
3      return {H'}
4  foreach node c ∈ C\V' do
5      N ← a set of neighbours of c ⊆ V\V'\C
6      foreach node v ∈ N do
7          C ← C ∪ {v}
8          foreach direction d ∈ {UP, LEFT, RIGHT, DOWN} do
9              H' ← embed v in H' to c in direction d
10             X ← X ∪ {H'} ∪ χ(G, H, C, k − 1)
11 return X
```

Algorithm 6: k-Path: $\phi(G, H, v, k)$

Input: $G = (V, E)$, $H = (V', E') \subseteq G$ where H is rigid and has a known embedding, $v \in V'$, k
Output: X, set of supergraphs of H using k nodes from G

```
1 N ← set of neighbours of v ⊆ V\V'
2 H' ← H
3 if k is 0 then
4     return {H'}
5 foreach u ∈ N do
6     foreach direction d ∈ {UP, LEFT, RIGHT, DOWN} do
7         H' ← embed u in H' to v in direction d
8         X ← X ∪ {H'} ∪ φ(G, H', u, k − 1)
9 return X
```

ate all paths that start in some u of H, of at most k nodes. By reducing the number of choices, Algorithm 6 has a complexity of $O(9^k)$, resulting in $O(n(n9^k + \Lambda(n9^k)))$ time complexity for Algorithm 3.

Theorem 3.2.3. *Let H be a rigid subgraph of G. Then if a path P in G has two endpoints on a single border B of H and each node u that is not one of the endpoints of P, is embedded at distance one to some node v in B then $P \cup H$ is rigid.*

Proof. Can be trivially seen given our definition of a border in Section 2.6.5. It is only possible to place such a path on at most one side of the border. □

Algorithm 7: Wrapper for parallel k-Paths

Input: $G = (V, E)$, $H = (V', E') \subseteq G$ where H is rigid and has a known embedding, $v \in V'$, k
Output: X, set of supergraphs of H using k nodes from G

```
1 N ← set of neighbours of v ∈ V\V'
2 H' ← H
3 if k is 0 then
4     return {H'}
5 foreach u ∈ N do
6     foreach direction d ∈ {UP, LEFT, RIGHT, DOWN} do
7         H' ← embed u in H to v in the direction perpendicular to d
8         X ← X ∪ ρ(G, H', u, d, k − 1), using Algorithm 8
9 return X
```

By choosing direction only once, we can improve the time complexity further. The idea behind Algorithm 7 is to generate a path P that will be embedded parallel to a border B of the subgraph as in Theorem 3.2.3. Initially, the algorithm tests the two

Algorithm 8: Parallel k-Paths: $\rho(G, H, v, k, d)$

Input: $G = (V, E)$, $H = (V', E') \subseteq G$ where H is rigid and has a known embedding, $v \in V'$, $d \in \{LEFT, RIGHT, UP, DOWN\}$, k

Output: X, set of supergraphs of H using k nodes from G

```
1 N ← set of neighbours of v ∈ V\V'
2 H' ← H
3 if k is 0 then
4     return {H'}
5 foreach u ∈ N do
6     H' ← embed u in H to v in direction d
7     X ← X ∪ ρ(G, H', u, d, k − 1)
8 return X
```

possible directions and all ways of adding k nodes in those directions. This allows us to verify the rigidity of the new graph faster because of Theorem 3.2.3.

To verify the rigidity we need to make sure that the endpoints of the path have an edge to the corresponding node on the border. Let P be a path with end-nodes u and v, where u is the neighbour of the starting node, u', given as input the Algorithm 7. Then we know that u has an edge to its corresponding node on the border. If v also has an edge to its corresponding node, i.e. the k^{th} node in the chosen direction from u', then we know by Theorem 3.2.3 that $P \cup H$ is rigid. If v does not have an edge to its corresponding node we check if the node before v has an edge to the $(k-1)^{th}$ node from u'. If that edge exists then we know that $(P \backslash v) \cup H$ is rigid. We keep on doing this until we find a node that has an edge to its corresponding node on the border. In the worst case, no node in P has an edge to its corresponding node and we check k nodes. Since we do not test all ways to embed each path we will only have $O(3^k)$ embeddings of paths to verify. The complexity of Algorithm 7 is thus $O(k3^k)$ and the total time complexity for Algorithm 3 is $O(n^2 k3^k)$.

3.2.2 Shortest Path in Grid Algorithm

We can generalise the idea of adding a path to a border and improve the time complexity. We propose finding a unique shortest path in the grid between any two nodes of the embedded graph, which we show in Theorem 3.2.5 gives a rigid graph. However, that theorem requires that any partial grid graph of n nodes can be embedded in an $n \times n$ grid which we state in Lemma 3.2.4.

Lemma 3.2.4. *A partial grid graph G that has n nodes can be embedded in an $n \times n$ grid.*

Proof. This is trivial since G has n nodes. □

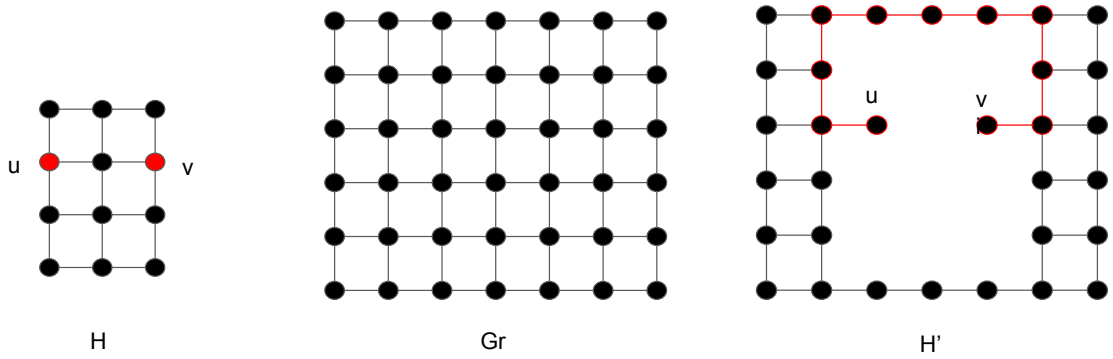


Figure 3.3: An example of how the graph H' from Theorem 3.2.5 is created

Theorem 3.2.5. *Let $G = (V, E)$ be a partial grid graph of n nodes and Gr an $(n+2) \times (n+2)$ grid. Let $H = (V', E')$ be a rigid subgraph of G, and u and v two nodes in H. Embed H in Gr such that all nodes of H have a distance of at least one to the boundary of Gr. Create H' by removing all nodes from Gr belonging to H except for u and v. If there exists a unique shortest path P between u and v in H' and a path of the same length in the subgraph of G containing no nodes from H except u and v, then $P \cup H$ is rigid. For an example see Figure 3.3.*

Proof. In the worst case H can have n nodes and be embedded in an $n \times n$ grid as mentioned in Lemma 3.2.4. Since Gr is $(n + 2) \times (n + 2)$ and we are looking for the shortest path between any pair of nodes, there are two cases: either the path is inside the grid Gr or the path has some nodes outside Gr. If P goes outside of Gr then it will cross the border of Gr twice, once to go out from H and once to come back to H. Therefore, we could create a shorter path by going along the border from where P exits Gr to where it enters again. This contradicts that P is shortest, i.e. the shortest path has to be inside Gr. Since P is unique we know that there is only one way to embed it hence the union is rigid. □

The shortest path can be computed efficiently by BFS and by modifying BFS we can verify uniqueness in constant time. The modified BFS can be seen in Algorithm 9 and works as follows. Let each node x have an internal counter which denotes the number of shortest paths from the root node to x, the root node's counter starts at one. The counter will be used to determine if there is a unique shortest path between the root and another node. Since BFS is implemented in a level-order fashion, we know what level we are currently at. When a node x is reached, the counter is updated as follows. If x has not yet been visited, we copy the counter from the parent in order to propagate the number of shortest paths. If x has been visited by another node in the previous level, we take the sum of the current counter and parent's counter. Since the node is reachable from multiple parents, the number of ways to get to the node x is the number of ways to reach each of its parents.

In order to improve this even further, we can store the results in a matrix, such that each entry of a matrix is a 3-tuple consisting of the parent, length, and the counter. This means that when we check if there exists a unique path between node u and v,

we will only look at the column of u in order to check uniqueness and reconstruct the path if needed. Using the BFS in this way, the complexity of the shortest path algorithm, Algorithm 10, is $O(n^4)$.

Algorithm 10 begins by marking all nodes and edges in the grid belonging to the rigid subgraph H. Then it uses the modified version of BFS which can only go through non-marked edges and is required to end and start in a marked node. Once the algorithm reaches a marked node that branch of the algorithm terminates. Marking nodes and edges in this way ensures that we will use edges not in H in order to go between all pairs of nodes in H. Algorithm 10 will run the BFS for every marked node. This will create a matrix of the shortest paths between two marked nodes. BFS has linear time complexity in the number of nodes of the graph. Since our grid is size $n \times n$ the BFS runs in $O(n^2)$. Doing this for every node of H in the worst case becomes $O(n^3)$, and retrieving the path takes $O(n)$ time as there can be no path longer than that in G. Moreover, in the worst case, we will always add one path of length two. This makes the do-while loop equivalent to another for-loop ranging from 1 to n nodes with $O(n^3)$ being an inner loop. This results in $O(n^4)$ time complexity. We use a matrix with $O(n^2)$ rows, one for each node in the grid, and $O(n)$ columns, one for each node in H, which leads to a $O(n^3)$ space complexity. The algorithm does not depend on k at all and is in fact polynomial, which is a significant improvement to the previously presented algorithms.

Algorithm 9: Modified BFS: $BFS(r, d)$

Input: r, d where r is the starting node, and d is a matrix of 3-tuples (parent, length, count)

Output: None, modifies d

```
1  currentQ ← empty queue
2  level ← 1
3  d[r, r] ← (−, level, 1)
4  flag ← false
5  while currentQ is not empty and !flag do
6      nextQ ← empty queue
7      while currentQ is not empty do
8          parent ← currentQ.pop()
9          foreach neighbour n in parent.neighbours do
10             if n is marked to belong to H then
11                 flag ← true
12             if edge (parent,n) is not marked then
13                 if n is not visited then
14                     d[r, n] ← (parent, level, d[r, parent].count)
15                 if n is visited and d[r,n].level = level then
16                     d[r, n].count = d[r, n].count + d[r, parent].count
17                 nextQ.push(n)
18     currentQ ← nextQ
19     level + +
```

Algorithm 10: Unique shortest-Path: $usp(G, H)$

Input: $G = (V, E)$, $H = (V', E') \subseteq G$ where H is rigid and has a known embedding

Output: H

```
1  Gr ← a grid of size |V| × |V|
2  foreach e = (u, v) ∈ E' do
3      mark nodes u,v in Gr
4      mark edge e in Gr
5  found ← false
6  do
7      distances ← empty matrix of size |V| × |V|
8      foreach u ∈ V' do
9          BFS(u, distances)
10     found ← false
11     foreach u ∈ V' do
12         foreach v ≠ u ∈ V' do
13             check if shortest path in distances is unique for node u, v
14             if unique then
15                 if G has a path of the same length between u and v then
16                     H ← embed the path by retrieving it from d[u,v]
17                     found ← true
18                     break out to do-while loop
19 while found
20
21 return H
```

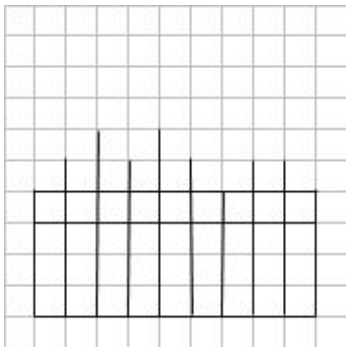


Figure 3.4: Example of a border being extended using spikes.

3.2.3 Spikes

We have shown that we can extend the border with a path in Theorem 3.2.3. However, there are cases when only one of the path's endpoints are part of the border and the graph remains rigid. This leads us to the idea of spikes.

Theorem 3.2.6. *Let H be a rigid graph and B a border of c nodes of H, b_i denote the i^{th} node, $1 < i < c$, on B. Let T be a set of paths. Then we have: $T \cup B$ is rigid if every $S_i \in T$, where one of S_i's end nodes is b_i, satisfies the condition that S_i is at most one longer than both neighbours S_{i-1} and S_{i+1}, with the corners of the border having paths of length zero. See Figure 3.4 for an example.*

Proof. Let a path S_i in T have endpoints u_i and v_i, $u_i \in H$. First, let us prove the base case. Assume all S_i in T are of length one. Any S_i consists of only two nodes u_i and v_i. We know that, for all i, u_i is not a corner and belongs to a border. This means that it has three occupied adjacent grid nodes, i.e. there is only one free grid node adjacent to u_i, so $T \cup H$ is rigid.

Now assume, as induction hypothesis, that the theorem holds when the longest path has length l, where $l \geq 1$ is any fixed number.

Let S_j be a longest path of length $l + 1$ and all paths in T be at most one longer than their neighbours. Then there are two cases:

i. Both of the neighbouring paths of S_j are of length l. Then we need to prove that there is only one way to embed v_j. When we want to embed v_j, three out of four grid nodes are already occupied by v_{j-1}, v_{j+1} and the previous node in S_j, meaning that there is only one free grid node to embed v_j.

ii. One or both of S_j's neighbours are of length $l + 1$. This doesn't change the fact that when we try to embed v_j there is only one free adjacent grid node.

By induction the theorem holds. □

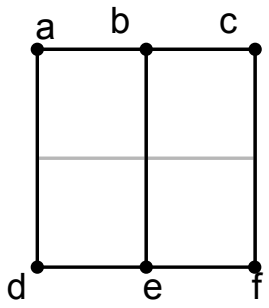


Figure 3.5: 3x2-cycle stack.

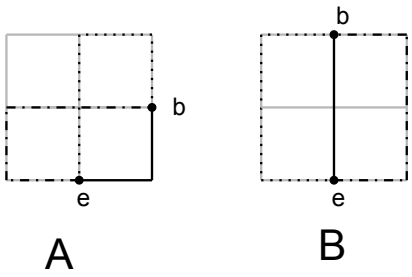


Figure 3.6: Possible embeddings of 3x2-cycle, the dotted lines are possible extentions of paths of length four.

3.3 Minimal Rigid Graphs

All the previous suggested solutions are dependent on that we start with a rigid subgraph of the input graph. Without general characteristics, finding such graphs is hard. Any edge could be used as a starting graph. However, in that case some rigid graphs will not be detected by our algorithms. Among others, *minimal rigid graphs* will not be detected, see Section 2.6.1. Therefore we shall present proofs of some minimal rigid graphs and prove that a special case of cycle stacks are minimal rigid graphs. Minimal rigid graphs will be used when searching for suitable starting graphs.

Lemma 3.3.1. *The graph in Figure 3.5 is rigid.*

Proof. Consider the nodes labelled b and e in Figure 3.5. These two nodes have to be embedded at a Manhattan distance of two from each other. There are two ways they can be embedded. To complete the embedding of the graph there have to be exactly two paths from b and e of length four that do not use the same grid nodes. In embedding A in Figure 3.6 there are two paths of length four shown as dotted lines. However, they share one grid node and therefore this embedding is not valid. Embedding B in Figure 3.6 also has exactly two paths from b to e which both are exactly four long. However, in B, the paths do not share any nodes, hence B is the only valid embedding. □

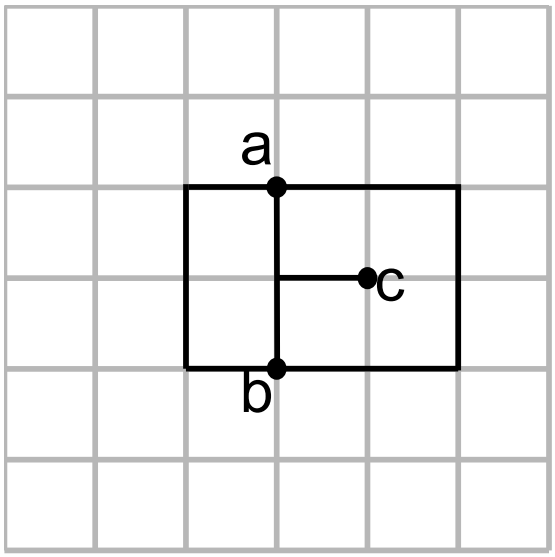


Figure 3.7: A rigid graph consisting of a 6- and 8-cycle with a node c inside the 8-cycle.

Lemma 3.3.2. *The graph in Figure 3.7 is rigid.*

Proof. Consider the node c in Figure 3.7, it has to be embedded in either the 6-cycle or the 8-cycle. Since a 6-cycle cannot be embedded with any grid nodes inside it, c has to be embedded inside the 8-cycle. The only way to embed an 8-cycle with one free grid node inside it is as a square. Hence, the only way to embed the graph in Figure 3.7 is the way shown in that figure. □

Lemma 3.3.3. *The graph in Figure 3.8 is rigid.*

Proof. There are seven ways in which two six-cycles can be embedded by sharing exactly two nodes as seen in Figure 3.9. We can see that in the original graph both a and b has a neighbour, c and d respectively, that is not part of the cycles. In all embeddings but one, all grid nodes adjacent to a and b are occupied. For the one with non-occupied grid nodes there is exactly one adjacent grid node to a and b respectively. This means there is only one way to embed c and d, therefore the graph is rigid. □

Lemma 3.3.4. *Graph in Figure 3.10 is rigid.*

Proof. See Figure 3.10 for possible configurations of the two cycles. In order for node c and d to connect to a and b respectively, a and b need at least one free grid node each. There is only one configuration that has free grid nodes to do that. □

Lemma 3.3.5. *Graph in Figure 3.11 is rigid.*

Proof. The 8-cycle can at most have one node drawn inside it, by being embedded as a square, meaning neither the $a - b$ or the $c - d$ path can be inside the 8-cycle. This means all four nodes a, b, c, d must be inside the 12-cycle. There is only one possible configuration of the 12-cycle that allows four nodes to be inside, that is as

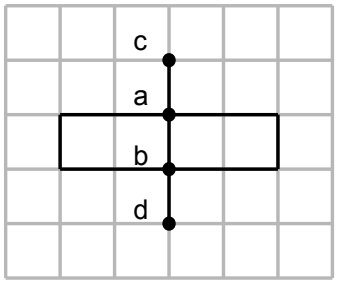


Figure 3.8: A rigid graph consisting of two 6-cycles with two nodes going out from them.

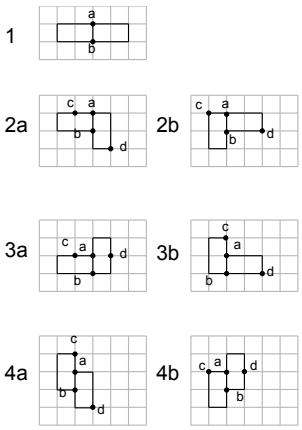


Figure 3.9: Possible configurations when two six-cycles share exactly two nodes.

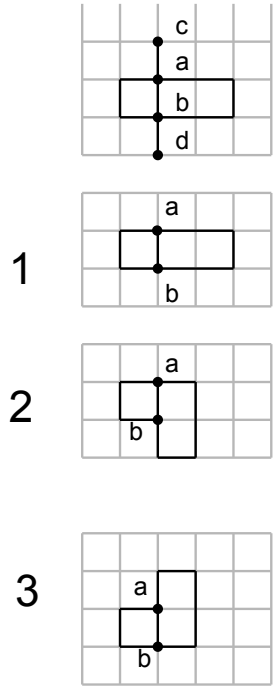


Figure 3.10: A rigid graph of 4- and 6-cycle, configurations below.

of a square. Having fixed the 12-cycle as a square, there is only one way that the 8-cycle can be attached to it. Hence, what is left to prove is showing that there is only one configuration of the two paths within the 12-cycle. Since a and c only have two neighbours each, where one of each is already fixed in the 12-cycle, it means a and c only have one valid place to be embedded. From this follows that b and c also only have one free grid node to be embedded in. □

Lemma 3.3.6. *The graph in Figure 3.12 is rigid.*

Proof. Assume the graph is not rigid, then some of the nodes a, b, c, or d will be embedded outside the cycle of nodes $1-12$, i.e. some of the nodes of the outer cycle

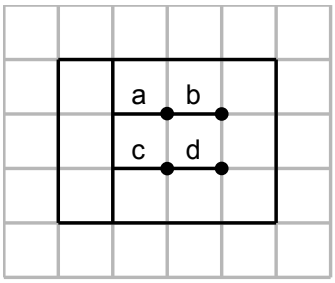


Figure 3.11: A rigid graph of an 8- and 12-cycle with four nodes, a, b, c and d, inside.

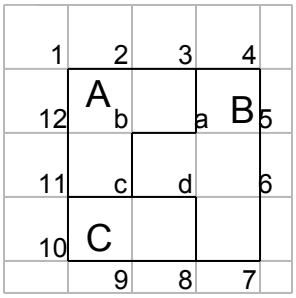


Figure 3.12: A rigid graph where A, B, and C denote cycles.

will be replaced by the nodes denoted by letters. The possible new outer cycles are

$$X : 1 - 2 - 3 - a - b - c - d - 8 - 9 - 10 - 11 - 12$$

$$Y : 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - c - b - a$$

$$Z : 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - d - c - 11 - 12$$

If X is the new outer cycle, the nodes $4, 5, 6$, and 7 will be embedded in either A or C. If they are embedded in A there is a node, 7, inside A with an edge to a node, 8, outside A, which result in an edge crossing. Edge crossings are not allowed by definition of planar embeddings in Section 2.2. Edge crossing also occurs if the nodes are embedded in C, because 4 has an edge to 3.

If Y is the new outer cycle, the nodes $12, 1$, and 2 will be embedded in either B or C. If they are embedded in C there is a node, 2, inside C with an edge to a node, 3, outside C. Same if they are embedded in B, then 12 has an edge to 11.

If Z is the new outer cycle, the nodes 9 and 10 will be embedded in either A or B. If they are embedded in A there is a node, 9, inside A with an edge to a node, 8, outside A. Same if they are embedded in B, then 10 has an edge to 11.

Therefore a, b, c, and d have to be inside the cycle $1 - 12$ and the only way to have four free grid nodes inside a 12-cycle is by making a square. There is also only one way to place these four nodes inside the cycle. That is because $1, 4, 7$ and 10 have to be corner nodes. If they are not then either $2, 5, 8$ and 11 or $3, 6, 9$ and 12 will be, which is not possible since $3, 8$, and 11 have three neighbours and can therefore not be corners. Hence, there is only one position for the nodes $3, 8$, and 11. Since a, c, and d have to be at distance one from $3, 8$ and 11 there is exactly one position for them as well, and then there is also only one possible position for b.

□

3.3.1 Cycle Stacks

Until now we have only proved one graph at the time to be minimal rigid. In particular, we have shown in Lemma 3.3.1 that a 3×2-cycle stack is rigid. Now, we show in Theorem 3.3.8 that similar cycle stacks are rigid. Consider the cycle stack

in Figure 3.13, we will use this cycle stack to name a few parts of the cycle stacks. In the cycle stack there are two horizontal paths that go between two corners, we will call these T, the top path, and B, the bottom path. Every path that contains exactly one node from T and one node from B path will be called a vertical path. Let the nodes in T be denoted by t_i, and the nodes in B be denoted by b_i. We will denote the vertical paths as P_i for $i = 1..m+1$ for an $n \times m$-cycle stack. We consider the cycle stack encircled by the cycle that is the union of P_1, T, P_{m+1}, and B and when we say that we embed a vertical path outside the cycle stack we mean that the nodes of that path are embedded outside that cycle.

Lemma 3.3.7. *An $n \times (n-1)$-cycle stack has to have all paths embedded inside the cycle stack.*

Proof. Let C_i be the cycle that is the union of the paths P_i, P_n and the paths $t_i - t_n$ and $b_i - b_n$, for $1 < i \leq \lceil \frac{n}{2} \rceil$. The proof for $\lceil \frac{n}{2} \rceil < i \leq n$ is symmetric because the cycle stack can be rotated.

Assume that some path P_i can be embedded outside the cycle stack. Trying to embed P_i outside the cycle stack is the same as trying to embed all nodes of $P_1 ... P_{i-1}$ inside C_i.

We claim that if we cannot embed the second vertical path outside then no other path can be embedded outside either. No matter which path P_i we embed outside the total number of nodes to be embedded will be n^2, however as i grows the perimeter of C_i shrinks. So if we cannot embed the nodes in the perimeter given by C_i we will not be able to embed them in the perimeter given by C_{i+1}. Therefore, if we cannot embed the nodes in perimeter given by C_2 it will not be possible to embed any vertical path outside the cycle stack.

The perimeter of the cycle C_2 is $2(n-3)+2n$. One can compute the maximum area of a rectangle with integer sides and perimeter p with the following formula:

$$f(p) = \lfloor \frac{p}{4} \rfloor * \lceil \frac{p}{4} \rceil$$

Since the optimal shape is given by four equally long sides we divided the perimeter by four. However, this might not be integer so we round two sides up and two sides down. Note that this is the side length, but we are interested in the number of nodes therefore we will add one to each of the lengths. We will use the following formula:

$$f(p) = (1 + \lfloor \frac{p}{4} \rfloor) * (1 + \lceil \frac{p}{4} \rceil)$$

The maximum area of C_2 is $(n-1)(n)$. We have n^2 number of nodes in total and C_2 can at most cover $(n-1) * n$ grid nodes, which leads us to a contradiction. □

Theorem 3.3.8. *An $n \times (n-1)$-cycle stack is rigid.*

Proof. Given a perimeter, the maximum rectangular area is achieved by a square. From Lemma 3.3.7 we know that all paths have to be inside the perimeter and by definition of cycle stacks all grid points inside will be occupied. This means that in

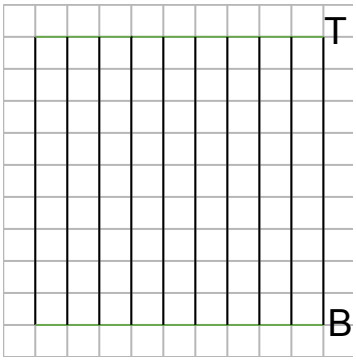


Figure 3.13: A cycle stack embedded as a rectangle. The top and bottom paths are marked by a green line.

order to have space for all the nodes the outer perimeter needs to form a square. Now that we know the shape of the perimeter, what is left to do is to i.) show that there is only one way to place the nodes on this perimeter, and then ii.) show that all lines inside are straight.

i. We claim that t_1, b_1, t_n and b_n are corners. Assume that t_1 is not a corner, that means that one of the nodes of $t_2 ... t_{n-1}$ will be a corner. This can not happen since those nodes have three neighbours which is a contradiction of the definition of a corner. Therefore, t_1 is a corner. Same holds for t_n, b_1 and b_n. At what corners they are placed does not matter, as it would be a reflected or rotated version. Given that the four corners are fixed, we implicitly get the position of all other nodes of the perimeter since we know how the nodes are connected given the definition of cycle stacks.

ii. Since the outer perimeter is fixed, we only need to show that the paths are vertical. This is easy since for each pair of t_i and b_i there is only one path of length $n - 1$ between them. This means that the lines are straight and can only be embedded in one way.

These two subcases mean that there is only one way to embed an $n \times (n-1)$-cycle stack. □

From Theorem 3.3.8 it follows that any cycle stack, where $m + 1 \geq n$, is rigid, we prove this in Theorem 3.3.9.

Theorem 3.3.9. *Any $n \times m$-cycle stack is rigid if $m + 1 \geq n$.*

Proof. By Theorem 3.3.8 we know that a $n \times m$-cycle stack is rigid if $n = m + 1$. We can create any $n \times (m + k)$-cycle stack, where $k \geq 1$ by using the shortest path algorithm to add more cycles. □

4
Discussion

In this chapter, we discuss the results and how current ideas can take advantage of multithreaded architectures. Furthermore, we propose some ideas for continuing the research.

4.1 Minimal Rigid Graphs

We have focused on proving minimal rigid graphs with fewer than 20 nodes. This is because as we see in the proof from Lemma 3.3.3, not all rigid graphs have elegant proofs. Sometimes we have to show all possible embeddings of a subgraph of the minimal rigid graph and then prove that only one of these embeddings is able to be extended to the minimal rigid graph. When the number of nodes grows it is hard for us to even have an intuition about if the graphs are minimal rigid or not.

4.2 Generality Comparison of Algorithms

In Chapter 3 we introduced all the algorithms, and how they were improved from one to the other in terms of complexity. We have mentioned briefly that the time complexity came at the cost of generality. This section discusses the difference in generality between the different algorithms.

The difference between Algorithm 5 and 6 is how we choose what nodes to extend from. It is very hard to compare the generality loss in this case, since so many different graphs can be found. The only thing we can say is that Algorithm 5 will find more graphs. Both algorithms depend on the comparison function $\Lambda(m)$. The complexity of $\Lambda(m)$ is polynomial, as we can just brute force by comparing all pairs of embeddings and check that the same nodes are embedded on same positions, taking into account translation, reflection and rotation. We believe that it is not necessary to find a tighter bound on $\Lambda(m)$, as using shortest-path and spikes algorithms together should find many of the graphs found by the FPT algorithms.

For small k it is likely that the shortest path algorithm recognises more rigid graphs than the Algorithm 6, and as k grows the number of graphs Algorithm 6 will recognise also grows. So there is a trade-off when to use one or the other method. If computing power is not a big concern, Algorithm 6 with a large k will perform better than the shortest path. An example of when Algorithm 6 outperforms the shortest path algorithm, in terms of the number of rigid graphs that it recognises, is when

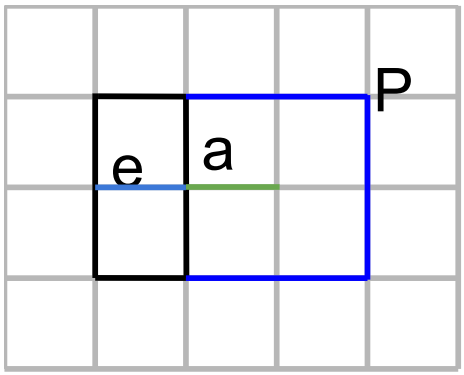


Figure 4.1: A rigid graph with two marked edges e(light blue) and a(green), as well as path P(blue).

the unique shortest path between two nodes does not exist in the input graph but the second shortest path is also rigid and would be found by Algorithm 6 given large enough k.

The one algorithm that stands out is the spikes algorithm. It is very limited in terms of generality, but its purpose is to help the shortest path algorithm to get out of local optima. For example, see the rigid graph in Figure 4.1. We showed that the same graph without edge e is rigid in Lemma 3.3.2. If we start with the two squares sharing the edge e, we have no way of extending it with path P since it will not be rigid. However, an iteration of the spikes algorithm can embed the edge a, and then the shortest path would allow us to embed P.

Even if we are not able to recognise and embed rigid graphs in general, we can find large rigid subgraphs. The hope is that there will not be too many nodes left from the original graph, such that we can bruteforce the rest in order to find a valid unit embedding, not necessarily rigid.

4.3 Parallelization

There are two ways in how the algorithms could take advantage of parallelization. The first idea is a simple fork-join, where we fork on different starting subgraphs. The join routine consists of the merging step of two graphs. Because of the merging step, which allows two rigid graphs to be merged into a rigid one, we are able to control how many forks of different subgraphs we want to do. This would give us a new parameter that can be optimised for different graphs, or that can change during the execution of the algorithms.

Next, most of our proofs for minimal rigid graphs rely on the fact that we are able to find all possible configurations of some smaller subgraphs since they are easier to enumerate. We reason how some configurations will not be possible. This process can be automated. For small graphs, there will not be that many possible configurations. For example in the case of 6-cycle, there are only three. Even though it is not rigid, we can temporarily treat it like that. In this case, we would need three threads to execute in parallel. The advantage is that we do not need to prove or search for graphs manually. A thread is terminated when no more nodes can be added. The embeddings of the same number of nodes are then compared to check uniqueness of the embedding. The unique embeddings are compared by size, then we choose the largest unique one. This would make it easier to find larger subgraphs.

The disadvantage is that the number of threads needed is linear to the number of different configurations of a graph, and the number of different configurations could be exponential in the number of nodes of a given graph. For example, there are three configurations of a 6-cycle, but 14 configurations of an 8-cycle.

4.4 Future Work

As mentioned in Section 3.2.2, the time complexity of using the unique shortest path idea is $O(n^4)$. We claim that the lower bound of the idea is $O(n^3)$ since we check all pairs of nodes which is $O(n^2)$ and we have to do that until we cannot add any more paths. In the worst case, we add a path of length one each iteration, making it $O(n^3)$. This implies that the current bottleneck of the algorithm is the loop in Step 8, which pre-computes all pairs of unique shortest paths. The reason being is that BFS takes $O(n^2)$ time because of the quadratic grid size. As future work, one could investigate how to optimise this further. It might not be necessary to consider all nodes in the grid. de Rezende et al. show an algorithm for finding the shortest grid path with obstacles in the plane [4]. What is left to do is to find a way to check the uniqueness constraint.

Since it is hard to analyse how many graphs can be detected by the shortest path and spikes algorithms, it is necessary to benchmark these algorithms on real data.

In Section 3.3.1 we prove that all $n \times m$-cycle stacks where $n \leq m + 1$ rigid. From this, we conjecture that those are the only cycle stacks that are rigid.

5
Conclusion

We have shown the difficulties of solving the problem for general rigid graphs and presented how to combat that by defining rigid graphs recursively.
We have introduced a novel idea of merging two rigid graphs, which is a huge improvement in terms of complexity even for naive brute force implementations as can be seen in Section 3.2.1. A few different algorithms with different generalities have been presented, and some of them are polynomial. Unfortunately, we cannot make any guarantees about being able to find the largest rigid subgraph, only that it is large. We explained the drawbacks with current solutions in the form of minimal rigid graphs. This is addressed by giving examples of a few possible minimal rigid graphs and their proofs in Section 3.3.

Bibliography

[1] S. Benjaafar, S. S. Heragu, and S. A. Irani. Next generation factory layouts: research challenges and recent progress. *Interfaces*, 32(6):58–76, 2002.

[2] S. N. Bhatt and S. S. Cosmadakis. The complexity of minimizing wire lengths in VLSI layouts. *Information Processing Letters*, 25(4):263 – 267, 1987.

[3] J. M. Boyer and W. J. Myrvold. On the cutting edge: simplified O (n) planarity by edge addition. *J. Graph Algorithms Appl.*, 8(2):241–273, 2004.

[4] P. J. de Rezende, D. Lee, and Y.-F. Wu. Rectilinear shortest paths in the presence of rectangular barriers. *Discrete & Computational Geometry*, 4(1):41–53, 1989.

[5] V. G. de Sá, G. D. da Fonseca, R. C. Machado, and C. M. de Figueiredo. Complexity dichotomy on partial grid recognition. *Theoretical Computer Science*, 412(22):2370–2379, 2011.

[6] M. M. Hassan and G. L. Hogg. A review of graph theory application to the facilities layout problem. *Omega*, 15(4):291–300, 1987.