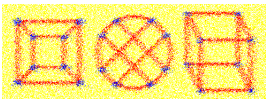

CHALMERS
UNIVERSITY OF TECHNOLOGY

# Enumerating grid layouts of graphs


Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 24, no. 3, pp. 433–460 (2020)
DOI: 10.7155/jgaa.00541

# Enumerating Grid Layouts of Graphs

*Peter Damaschke*

Department of Computer Science and Engineering, Chalmers University, 41296 Göteborg, and Fraunhofer-Chalmers Research Centre for Industrial Mathematics, 41288 Göteborg, Sweden

**Abstract**

We study algorithms that generate layouts of graphs with $n$ vertices in a square grid with $\nu$ points, where adjacent vertices in the graph are also close in the grid. The problem is motivated by graph drawing and factory layout planning. In the latter application, vertices represent machines, and edges join machines that should be placed next to each other. Graphs admitting a grid layout where all edges have unit length are known as partial grid graphs. Their recognition is NP-hard already in very restricted cases. However, the moderate number of machines in practical instances suggests the use of exact algorithms that may even enumerate the possible layouts to choose from. We start with an elementary $n^{O(\sqrt{n})}$ time algorithm, but then we argue that even simpler exponential branching algorithms are more usable for practical sizes $n$, although being asymptotically worse. One algorithm interpolates between obvious $O^*(3^n)$ time and $O^*(4^\nu)$ time for graphs with many small connected components. It can be modified in order to accommodate also a limited number of edges that can exceed unit length. Next we show that connected graphs have at most $2.9241^n$ grid layouts that can also be efficiently enumerated. An $O^*(2.6458^n)$ time branching algorithm solves the recognition problem, or yields a succinct enumeration of layouts with some surcharge on the time bound. In terms of the grid size we get a slightly better $O^*(2.6208^\nu)$ time bound. Moreover, if we can identify a subgraph that is rigid, i.e., admits only one layout up to congruence, then all possible layouts of the entire graph are extensions of this unique layout, such that the combinatorial explosion is then confined to the rest of the graph. Therefore we also propose heuristic methods for finding certain types of large rigid subgraphs. The formulations of these results is more technical, however, the proposed method iteratively generates certain rigid subgraphs from smaller ones.

| Submitted: | Accepted: | Final: | Published: |
|---|---|---|---|
| August 2019 | July 2020 | August 2020 | August 2020 |
| Article type: Regular Paper | | Communicated by: Antonis Symvonis | |

*E-mail address:* ptr@chalmers.se (Peter Damaschke)

# 1 Introduction

Throughout this paper, a *grid* means any set of *grid points*, that is, points with integer Cartesian coordinates in the 2-dimensional plane. Every grid defines a *grid graph* where any two points with Euclidean distance 1 are joined by an arc. We also call such points *neighbors*, whereas *diagonal neighbors* have Euclidean distance $\sqrt{2}$. The *discrete plane* is the infinite grid consisting of all points with integer coordinates.

We also deal with general graphs $G = (V, E)$ which are undirected and have no loops or parallel edges. To avoid confusion, we speak of *vertices* and *edges* of $G$, but of *points* and *arcs* of a grid graph. We assume that the vertices are distinguishable, informally, they have "names".

An *embedding* of a graph into a grid is any injective mapping of its vertex set into the set of grid points,. That is, distinct vertices must be mapped to distinct points. A *layout* is an equivalence class of congruent embeddings, i.e., embeddings that can be transformed into each other by translations, rotations through multiples of 90 degrees, and reflections. Equivalently, a layout is uniquely determined by the Euclidean distances between all pairs of vertices. Note carefully that two layouts are considered different if they give some pair of vertices (specified by their names) different Euclidean distances.

In an embedding or layout, an edge is called *short* if it has been mapped to an arc, and *long* otherwise. In an *ideal layout* of a graph, all edges are short. A *partial grid graph* is a graph that admits an ideal layout in the discrete plane.

**Our problem is:** Decide whether a given graph $G$ is a partial grid graph, and if so, construct an ideal layout. We may even want to enumerate all ideal layouts. Similarly, a graph $G$ and a finite grid may be given, and the problem is to construct an ideal layout of $G$ that fits in this grid, if possible. Unless said otherwise, we always denote by $n = |V|$ the number of vertices of the input graph $G = (V, E)$, whereas $\nu$ denotes the number of points in the grid, in cases when the grid is finite. We can, trivially, assume $n \leq \nu$.

The decision problem is NP-complete even for trees with certain degree restrictions, see [7] and earlier work cited there. Actually, in [7] the complexity is classified with respect to the vertex degrees that occur in the graph. A polynomial but sophisticated algorithm for a very special case can be found in [20].

The problem is of interest in VLSI layout and graph drawing, however, we were led to it by facility layout planning [17], which is the task of placing several "resources" (machines, workplaces, etc.) in a factory hall so as to optimize workflow and ergonomy. For recent surveys and extensive bibliographies we refer to [2, 3, 8]. Research on layout planning has hitherto focused on various optimization methods and heuristics. In the present paper we take a fresh view and consider graph layouts in the grid as a graph-theoretic abstraction, and we attack the problem by exact algorithms.

We may represent machines as vertices of a graph and the floor plan of the factory hall as a grid. Two vertices are joined by an edge if the machines should

be placed close to each other, e.g., because they carry out consecutive steps in some production process, or the same worker must operate both machines.

It is natural to discretize the floor, with an appropriately chosen unit length, and place machines only on grid points. In reality their positions may still be adjusted, starting from such a discrete solution. The two directions parallel to the walls are also the preferred moving directions for workers and materials, whereas layouts with sinuous ways between machines may be perceived as confusing. (One might also consider hexagonal grids, but the algorithmic problems and ideas would be similar, just with different details.) An issue is that resources, i.e., machines with their surrounding work spaces, can have different sizes and shapes. However, shapes are often limited to simple polygons with axis-parallel sides, and we may allow rotations only through multiples of 90 degrees. Then, large resources are represented as subgraphs with prescribed pairwise distances of their vertices. Thus we are still in the realm of graph layouts in grids.

A related problem which is not addressed here is to place departments (rather than machines) that have prescribed areas but flexible shapes, where certain departments must be neighbored. In our problem, resources have fixed shapes, and the the worst case for the number of layouts appears when every resource occupies just one grid point.

In the envisioned application. the edge set of the graph is user-defined, in that an expert decides which pairs of machines are most important to be neighbors on the floor. Then, an algorithm presents some (or all) possible layouts. They are further examined by ergonomic or even esthetic criteria. The user may also interactively change the edge set, in particular, remove some edges if the instance was over-constrained, and run the algorithm repeatedly.

We do not explicitly consider "negative" edges joining machines that should be far apart (e.g., due to emissions). However, if an algorithm enumerates all suitable layouts, one can afterwards choose one that also maximizes the lengths of such edges.

## 2 Related Work and Overview of Contributions

Since grid graphs are planar graphs, the problem of finding an ideal layout in a grid is a special case of the Planar Subgraph Isomorphism problem. The result from [16, Theorem 5.14] implies an algorithm for finding an ideal layout of a connected graph, that runs in $O(n^{t+1} \cdot \nu)$ time, where $t$ is the treewidth of the grid. (We do not explain the notion of treewidth here, because we will not use it, and this information is only some context to put our own results into.) The mentioned result assumes a fixed maximum degree of the graph, which is satisfied here, since graphs of degree larger than 4 cannot have an ideal layout. The treewidth of planar graphs of $\nu$ vertices is bounded by $O(\sqrt{\nu})$. (For instance, $3.182\sqrt{\nu}$ is shown in [10], and $\sqrt{\nu} \times \sqrt{\nu}$ grids have treewidth $\sqrt{\nu}$.) This yields a subexponential time bound of $n^{O(\sqrt{\nu})} \cdot O(\nu)$.

The situation is different if the grid is not given. Of course, since the graph

is finite, we may first restrict the discrete plane to some finite grid. But the catch is that we may need $\nu = \Theta(n^2)$ points, since the "shape" of the layout is not known in advance. (Finding a layout is the very problem to be solved.) At least, we do not see an easy way to overcome this issue, except in special cases like graphs with small diameter. Hence the above results do not seem to imply a subexponential algorithm. Still we can design one from scratch, using dynamic programming on a tree of recursive partitionings of the graph along small, balanced, and geometrically simple separators, guessing where in the grid these separators land. While the overall method is pretty much standard, a difficulty is to devise such separators without knowing the layout. In Section 3 we resolve this problem by using a more complicated H-shaped separator and achieve a time bound of $O^*(2^{10.25\sqrt{n}} \cdot n^{17.1\sqrt{n}+3\log_2 n+2})$.

Recent work [18] provides a subexponential algorithm for the much more general problem of finding and counting patterns in arbitrary planar graphs, however, the algorithm is far more complicated, and the time bound has additional logarithmic factors in the exponent. The result from [5, Theorem 7] yields an algorithm for finding an ideal layout of a graph that is not necessarily connected, in a finite grid. It runs, essentially, in $2^{O(\sqrt{\nu}+n/\log n)} \cdot n^{O(1)}$ time. The result holds for the Subgraph Isomorphism problem in general, just assuming that the graphs to be embedded have some excluded minor, which is satisfied for planar graphs. The $\sqrt{\nu}$ term comes again from the treewidth of the host graph (here: a grid). Moreover, the time cannot be improved under the Exponential Time Hypothesis [5, Theorem 15].

However, for our purposes, these subexponential algorithms (including our own) do not appear to be practical, for two main reasons: They are not well suited for the enumeration of alternative layouts, and the big machinery of recursive decomposition apparently makes them slower than even simple branching algorithms on relevant instance sizes, due to large constants in the exponents of their time bounds. Therefore we complement them with exponential branching algorithms that are asymptotically slower but conceptually simpler, easier to implement, and should run faster for realistic input sizes. They might even be valuable for other applications with large graphs, as a hybrid approach may do only a few steps of recursive partitioning and then switch to branching algorithms to deal with the small subgraphs in the partitionings.

Note that the sheer number of layouts of connected graphs can be exponential in the worst case, and even higher for disconnected graphs. Despite this fact, in Section 4 we give an $O^*(\nu^c \cdot 3^g \cdot 4^h)$ time algorithm to enumerate all ideal layouts of a graph in a finite grid, where $c$ is a user-defined integer parameter, $g$ is the total number of vertices in the $c$ largest connected components, and $g + h = \nu$. We can also efficiently cope with graphs whose layouts require a limited number $\ell$ of long edges, as shown in Section 5. We incur an extra time factor of $n^{O(\ell)}$.

While Sections 4 and 5 care about disconnected graphs, the following two sections focus on improved branching for the enumeration of layouts of the connected components, or of connected graphs. In Section 6 we improve upon the obvious branching number 3: We show that connected graphs have at most $2.9241^n$ grid layouts, and we enumerate them within the corresponding time

bound. An $O^*(2.6458^n)$ time branching algorithm solves the recognition problem, or alternatively, it can be used to obtain a succinct enumeration of layouts in $O^*(2.7822^n)$ time. Note that these time bounds are expressed in terms of the graph size, taking advantage of connectivity of the graph. In the short Section 7, however, we go back to finite grids again, to show that also limited space can support branching. We get a slightly better $O^*(2.6208^\nu)$ time bound, which might be interesting when a connected graph must be embedded in a barely larger grid. Practically more relevant than the marginally smaller branching number is the fact that the used branching rules are much simpler.

But perhaps the biggest practical advantage of the simple approach to successively add vertices and branch on their grid positions is that it facilitates the early recognition of subgraphs that admit only a few ideal layouts, far below the exponential worst-case number. In order to exploit such opportunities, it is sensible to try and efficiently find increasing subgraphs of $G$ with only few ideal layouts. In the best case, these are nested sequences of rigid subgraphs, i.e., such with only one ideal layout. Once we have built up some large rigid subgraphs of $G$ and computed their unique layouts, it only remains to extend them by the remaining vertices of $G$. That is, the combinatorial explosion in the NP-hard layout enumeration problem is then confined to the rest of $G$.

This leads to the second part of the paper. From Section 8 on we present an efficient algorithm for detecting a rather general type of rigid subgraphs. (In the following we spare the exact technical formulations of most results, in order to avoid an overly long introduction.) A structural result in Section 8 might be of independent interest: Trees, except the trivial one-edge graph, are never rigid. This speeds up the search for rigid graphs. We list all rigid graphs up to eight vertices and observe that they can be constructed in a specific way. This suggests one main idea, followed in Section 9. There we show that a rigid graph $H$ extended by a path of new vertices that connects two vertices of $H$ remains rigid if this new path is the unique shortest path between its endpoints in the grid without the embedded graph $H$. Since unique shortest paths can be found in polynomial time, this extension procedure for rigid graphs is efficient. Next, this gives rise to a more general framework presented in Section 10, where we suppose that some fast procedure is available, that extends a rigid subgraph of $G$ to a larger rigid subgraph or reports that it cannot find a larger one. We derive a general time bound for generating all rigid subgraphs of $G$ that are reachable by such extensions. In Section 11 we make this mechanism more powerful, in that we capture a larger class of rigid graphs without increasing the overall time bound. Specifically, besides extensions we also use pairwise unions of rigid graphs. The time analysis uses a potential function argument.

Rigidity of graphs with given edge lengths, with respect to embeddings in the plane or higher-dimensional Euclidean space has been studied intensively (see, e.g., [1, 4, 9, 14, 15, 19]), but analogous concepts for embeddings into grids are novel, to our best knowledge. We remark that also [9] deals with a different type of problem, despite the title.

Section 12 concludes the paper with some discussion of a few aspects and possible directions of further research.

**Terminology remarks.** We use the $O^*$ notation that suppresses polynomial factors, which allows us to focus on the superpolynomial parts and to skip some less interesting data structure details. A problem *instance* is a pair of a graph and a grid (in which we want to embed the graph). The word *component* refers to a connected component of a graph, that is, we omit the word "connected" for brevity. We say that a vertex is *decided* if we have already mapped it irrevocably to some grid point, during the construction of a layout.

# 3 A Subexponential Algorithm

**Theorem 1** *We can recognize in* $O^*(2^{10.25\sqrt{n}} \cdot n^{17.1\sqrt{n}+3\log_2 n+2})$ *time whether a given graph* $G$ *has an ideal layout in the discrete plane.*

**Proof:** We can assume $G$ to be connected, and otherwise solve the problem on each connected component. For a connected graph $G$, the problem is equivalent to finding an embedding into the $n \times n$ grid.

In the following, a row (column) means a horizontal (vertical) line of grid points. Suppose that some embedding of $G$ is already given. We introduce the following concepts for analysis purposes. We consider all columns from the leftmost to the rightmost column that contain any vertices, and we add one more column (not containing vertices) to the left and to the right. A column with at most $\sqrt{n}$ vertices is called sparse. (In particular, the two extra columns are sparse.) A column with more than $\sqrt{n}$ vertices is called dense. Two sparse columns are called consecutive if only dense columns are between them. Now let $(L, R)$ be the leftmost pair of consecutive sparse columns. If more than $n/2$ vertices are to the right of $R$, then let $L := R$, and let $R$ be the next sparse column. This new pair $(L, R)$ is still a consecutive pair, with at most $n/2$ vertices to the left of $L$. By induction we conclude that two consecutive sparse columns $L$ and $R$ exist, with at most $n/2$ vertices to the left of $L$ and to the right of $R$, respectively.

Consider the stripe between $L$ and $R$. By s similar argument as before, there exists a row $M$ within this stripe, such that at most $n/2$ vertices are above and below $M$, respectively. Moreover, $M$ has a length at most $\sqrt{n}$, since otherwise there would be more than $n$ vertices in the stripe, as it consists of dense columns only.

Altogether, there exist three lines forming an "H", each containing at most $\sqrt{n}$ vertices, that together partition the grid into four subgrids none of which hosts more than $n/2$ vertices. We call it a H-shaped separator. This gives rise to an algorithm as described below.

We generate all possible H-shaped separators and place a set $S$ of vertices there. We decide on the position of the separator in the $n \times n$ grid (including the length of $M$) and on the vertices placed on the separator. There are at most $n^3$ ways to place the separator in the grid. On each of $L$ and $R$ we can place vertices in at most $n^{2\sqrt{n}}$ ways, since we can choose both a vertex and a grid point, at most $\sqrt{n}$ times. Row $M$ can be populated with vertices in at

most $n^{\sqrt{n}}$ ways. Together we have at most $n^{5\sqrt{n}+3}$ ways to place a vertex set $S$ on a separator. For each of them we also decide in which of the four subgrids we place every component of $G - S$. This step will be detailed in the next paragraph. Then, we further divide the subgrids and the allocated subgraphs recursively and independently.

Let $G'$ denote the subgraph allocated to any subgrid, prior to the recursive partitioning step, with a separator $S$. Every component of $G' - S$ must be placed entirely in one of the four resulting smaller subgrids. We distinguish two types of components of $G'$.

(i) Every component $C$ of $G'$ not intersected by $S$ is still a component of $G' - S$, and since the entire graph $G$ was connected, $C$ contains at least one decided vertex, placed at the border of the current subgrid. In particular, it is clear to which smaller subgrid $C$ belongs.

(ii) Let $U$ be the union of all components of $G'$ that intersect $S$. Consider any component $C$ of $U - S$. Observe that some vertex $c \in C$ is adjacent to some vertex $s \in S$. Hence there exist at most $6\sqrt{n}$ such components $C$, namely at most two for every vertex in $S$. But whenever two components are adjacent to the same $s \in S$, they must end up on opposite sides of the line of the H-shaped separator that contains $s$. It follows that the subgrids for all components $C$ in $U$ can be chosen in at most $2^{3\sqrt{n}}$ ways. Once we have chosen the subgrid $C$ belongs to, $c$ becomes a decided vertex.

For every recursion step we conclude as above that we can divide any subgrid in at most $2^{3\sqrt{k}} \cdot n^{5\sqrt{k}+3}$ ways, where $k$ denotes the number of vertices in the considered subgrid. (The base $n$ can be improved, but this would terribly complicate the calculations and improve the final bound only marginally.)

We obtain a partitioning tree, with layers of tree nodes that have, alternatingly, "many" children (the partitionings by separators) and up to four children (the resulting smaller instances). Finally we do dynamic programming bottom-up in this tree, where the tree nodes alternatingly serve as AND and OR gates: When *all* sub-instances, separated by *some* set $S$, have a layout, then we can combine them to some layout of the instance.

It remains to bound the number of leaves of the partitioning. Since the maximum number $k$ of vertices of the subgraphs decreases at least by a factor 2 upon every splitting, we have at most $\log_2 n$ recursion levels, and the exponents of both 2 and $n$ (containing $\sqrt{k}$) form a geometric series with quotient $\sqrt{1/2} = 1/\sqrt{2}$, which has the sum $2 + \sqrt{2}$. Together this yields a factor $2^{3(2+\sqrt{2})\sqrt{n}} \cdot n^{5(2+\sqrt{2})\sqrt{n}+3\log_2 n}$. Since each subgrid is split into 4 smaller ones, the corresponding tree nodes contribute another factor no larger than $4^{\log_2 n} = n^2$. Numerical calculation finally yields the bound. □

# 4 Enumerating Layouts of Disconnected Graphs

We begin with a trivial branching algorithm that enumerates all layouts of a connected graph in $O^*(3^n)$ time: Initially, place some vertex on some grid point. Then, successively add some vertex that is adjacent to some decided vertex, and

place it on a grid point, in all (at most 3) possible ways. It is also easy to see that the number of different layouts of a connected graph can, in fact, be exponential. (As an example, consider paths of $n$ vertices.) We stress that the bounds depend on the vertex number $n$, but not on the grid size $\nu$, and this basic algorithm can be run in the discrete plane.

For disconnected graphs, the number of layouts can be superexponential, even in a finite grid: In the extreme case, if the graph consists of $n$ isolated vertices and $\nu = n$, there are $n!$ different layouts. However, it would be silly to explicitly list them, as it is clear that isomorphic components can be permuted arbitrarily. A more interesting question is whether we can enumerate, in single exponential time in $\nu$, all "essentially different" layouts in a finite grid, i.e., up to automorphisms (which can be permutations of isomorphic components and automorphisms within the components). We give an affirmative answer, with a worst-case base as small as 4. In order to take advantage of the even smaller base 3 for connected graphs, we give a refined result where large and small components are treated differently. In the following theorem, the number $c$ is a free parameter, and the best choice of $c$ depends on the sizes of the components.

**Theorem 2** *Given a grid, a graph $G$, and an integer $c$, let $g$ denote the total number of vertices in the $c$ largest components of $G$ (where ties are broken arbitrarily), and $h := \nu - g$. We can, in $O^*(\nu^c \cdot 3^g \cdot 4^h)$ time, enumerate all ideal layouts of $G$ that fit in the grid, up to automorphisms of $G$.*

**Proof:** We first generate all ideal layouts of the $c$ largest components of $G$ (in the discrete plane, not yet caring about their placements in the given grid). This costs $O^*(3^g)$ time altogether.

For each of the $c$ largest components we select an ideal layout and place it on the grid. This results in at most $\nu^c \cdot 3^g$ valid partial solutions. Each of them leaves a grid of $h$ yet unused points, that must host the small components (i.e., all components except the $c$ largest).

Since a grid has maximum degree 4, a grid graph with $h$ points has at most $2h$ arcs. For every arc we decide whether it shall be used to map some edge of $G$ on it, or not. We have at most $2^{2h} = 4^h$ choices that we call *prepared grids*.

For each of the, at most $\nu^c \cdot 3^g \cdot 4^h$, prepared grids we finish up as follows. We temporarily delete the unused arcs and determine the components of the remaining partial grid graph $P$ with $h$ points. For every pair of a small component of $G$ and a component of $P$ we test for graph isomorphism (and in the positive case, establish a corresponding bijective mapping of points and vertices). Since partial grid graphs are planar, this can be done in polynomial time altogether [13]. This also partitions the set of components, both in $G$ and in $P$, into isomorphism classes.

Finally, a solution exists if and only if, for every class of isomorphic small components of $G$, there exist at least as many isomorphic components of $P$. In this case, every injective mapping of the former into the latter isomorphism classes yields a solution, and the isomorphisms also yield, for every small component of $G$, a layout which is congruent to the component of $P$ it is mapped to. □

We mention a variant of the algorithm from Theorem 2 that does not need a planar isomorphism test as an external routine. In the beginning we may generate all ideal layouts of all components of $G$ in $O^*(3^n)$ time. Furthermore, we observe that every small component has at most $n/c$ vertices. Thus, the list of ideal layouts of all small components has a total length $O^*(3^{n/c})$. For every component of any prepared grid we may therefore find all congruent layouts of small components of $G$ directly in $O^*(3^{n/c})$ time, by traversing this list. This is no longer polynomial but, for instance, for $c := \sqrt{n}$, the factors $n^c = 2^{\sqrt{n}\log_2 n}$ and $3^{n/c} = 3^{\sqrt{n}}$ are subexponential, such that the exponential part of the time bound is still $3^g \cdot 4^h$. Another difference to the original algorithm is that the automorphisms of every small component are now explicitly listed (via the congruent layouts of that component), however, their number is subexponentially bounded, as opposed to the permutations of isomorphic components.

# 5 Inserting Long Edges

Assume that the input graph $G$ is not exactly a partial grid graph but admits a layout with at most $\ell$ long edges. In this supplementary section we show that the enumeration result from Section 4 can be extended to this case, at cost of an additional factor $n^{O(\ell)}$ in the time bound. Hence, as long as $\ell = O(n/\log n)$, the time bound from Section 4 even remains single exponential. A possible way is the following.

A partial grid graph with $n$ vertices has at most $2n$ edges, due to the maximum degree 4. A graph capable of a layout with at most $\ell$ long edges has therefore at most $2n + \ell$ edges. In $G$ we may select $\ell$ edges that we allow to become long, in at most $(2n + \ell)^\ell/\ell! = n^{O(\ell)}$ ways. For each of these choices, let $G'$ be the graph after deletion of these selected edges.

We proceed with $G'$ as in Theorem 2, with a minor modification: We must separately treat the components of $G'$ that are incident to long edges, as their positions determine the placements of the long edges. But there exist at most $2\ell$ such components, and we may first decide on their positions in the grid (similarly as we did for the $c$ largest components in Theorem 2). This only incurs another $n^{O(\ell)}$ factor.

Some further remarks are in order:

After enumerating the layouts we may want to pick some that minimize some given monotone function of the edge lengths. Since the vertices incident to long edges become decided vertices, this also works with our succinct enumerations up to automorphisms.

If a bound $\ell$ is not given in advance, and we wish to minimize $\ell$, we may run the algorithm for $\ell = 0, 1, 2, \ldots$ until success. The last round dominates the time complexity.

The $n^{O(\ell)}$ factor is very generous, as the worst case of getting $2\ell$ components for every choice of long edges is unlikely. Moreover, further heuristics can easily restrict the family of subsets of the $\ell$ candidate edges: Forbidden subgraphs of partial grid graphs include odd cycles (as they are not bipartite) and graphs with

too many vertices within some radius (as they cannot fit in a grid). Breadth-first search around every vertex can quickly identify small forbidden subgraphs, and clearly, at least one edge of every forbidden subgraph must be long.

# 6 Improved Branching for Connected Graphs

As we argued earlier, a connected graph has, up to translations and rotations, at most $3^n$ ideal layouts that can be enumerated in $O^*(3^n)$ time, but it is worthwhile to reduce the base 3, and hence the $3^g$ factor in Theorem 2.

**Theorem 3** *A connected partial grid graph $G$ with $n \geq 6$ vertices has at most $\sqrt[3]{25}^n < 2.9241^n$ ideal layouts, up to translations and rotations.*

**Proof:** We say that a path $P = (v_p, \ldots, v_1, v_0)$ of length $p \geq 1$ is *pendant* if $v_0$ has degree 1, each vertex $v_i$, $0 < i < p$, has degree 2, and $v_p$ has a degree either equal to 1 or larger than 2. Let $P' = (v_{p-1}, \ldots, v_1, v_0)$. Note that $G - P'$ remains connected. Independently of that, $G$ has always some vertex $v$ such that $G - v$ remains connected.

If $G$ has minimum degree 2, then let $v$ be such a vertex. Since $v$ has at least 2 neighbors, to every ideal layout of $G - v$ we can add $v$ in at most 2 ways.

If $G$ has minimum degree 1, then let $P$ be some pendant path. If $p \leq 3$, then $v_p$ has at least 2 neighbors in $G - P'$ (since $n \geq 6$). Thus, given any ideal layout of $G - P'$, we can add $v_{p-1}$ in at most 2 ways. Consequently, $P'$ can be appended to $G - P'$ in at most $2 \cdot 3^{p-1}$ ways, for $p = 1, 2, 3$. Note that 2, $\sqrt{6}$, $\sqrt[3]{18}$ are all smaller than $\sqrt[3]{25}$.

If $p \geq 4$, then, given any ideal layout of $G - \{v_2, v_1, v_0\}$, we can append $(v_2, v_1, v_0)$ in at most 25 ways: $v_2$ is a neighbor of $v_3$ which has already another neighbor $v_4$, thus we can append the mentioned 3 vertices in $3^3$ ways, but in 2 of them, $v_0$ would collide with $v_4$.

From these cases, induction on $n$ yields the assertion. □

Theorem 3 can be turned into an $O^*(2.9241^n)$-time algorithm, which can also replace the basic $O^*(3^n)$-time algorithm in Theorem 2. However, the base can be further improved significantly by doing branching steps only until polynomial-time solvable residual problem instances remain. This requires some refined, yet simple and local branching rules. To avoid a lengthy and artificial theorem statement, the following result is formulated for the recognition problem only, but the proof provides more, and we will comment on the implications afterwards.

**Theorem 4** *We can recognize in $O^*(\sqrt{7}^n) = O^*(2.6458^n)$ time whether a given connected graph $G$ has an ideal layout in the discrete plane.*

In the remainder of this section we prove Theorem 4. In the algorithm description, the phrase "branch on" means to do the indicated step in all possible ways and to generate the resulting residual problem instances.

**Branching algorithm.**

First we map one edge to an arc. Its two vertices are marked as *placed*. Vertices are marked as *active* if they are placed on the grid and may be adjacent to further, not yet placed vertices. We iterate the following steps until no active vertices exist any more. Since $G$ is connected, all vertices of $G$ are then placed.

We arbitrarily pick some active vertex $u$ and some vertex $v$ being adjacent to $u$ but not yet placed. The following cases can appear.

(1) If $v$ (as specified above) does not exist, we mark $u$ as *passive*. Henceforth we assume that some $v$ exists.

(2) If $v$ is adjacent to yet another placed vertex besides $u$, we branch on the grid point of $v$, and we mark $v$ as placed and active. Henceforth we assume that $u$ is the only placed vertex adjacent to $v$.

(3) If $v$ is adjacent to at least 2 not yet placed vertices $w$ and $w'$, we branch on the grid points of $v, w, w'$, and we mark all these vertices as placed. We mark $v$ as passive and the other newly placed vertices as active.

(4) If $v$ is adjacent to exactly one not yet placed vertex $w$, we branch on the grid point of $w$ (rather than $v$). If $w$ gets Euclidean distance 2 from $u$, then obviously $v$ must be placed between $u$ and $w$. If $w$ becomes a diagonal neighbor of $u$, we mark $v$ as *undecided* and leave it open on which of the 2 possible points we will place $v$. In either case we mark $v$ as passive and $w$ as active, and we mark them both as placed (even if $v$ is undecided).

(5) If $v$ is adjacent to no further vertex ($v$ has degree 1), we mark $v$ as placed, undecided, and passive.

**Analysis of the branching number.**

We examine all possible cases regarding the active vertex $u$ and its selected neighbor $v$ processed in a branching step. Sometimes we work with coordinates, where $(0, 0)$ denotes the point of $u$. A placed vertex is decided if it is not labeled undecided. In particular, $u$ is always decided (since undecided vertices are always passive), Note that at least one neighbor or diagonal neighbor of the point of $u$ is already occupied by another decided vertex. In cases (3) and (4) we distinguish between these two subcases. No branching happens in cases (1) and (5).

(2) In this case $v$ can be placed on at most 2 possible points, which yields a branching number at most 2.

(3.1) Some neighbor of $(0, 0)$ is occupied, say $(-1, 0)$. Then we can place $v$ at one of $(1, 0)$, $(0, -1)$, $(0, 1)$, and place $w$ and $w'$ in $3 \cdot 2 = 6$ ways. These are together 18 ways. Hence the branching number is at most $\sqrt[3]{18} < \sqrt{7}$ (since $18^2 = 324 < 343 = 7^3$).

(3.2) Some diagonal neighbor of $(0,0)$ is occupied, say $(-1,-1)$. If we place $v$ at $(-1,0)$ or $(0,-1)$, then we can place $w$ and $w'$ in only 2 ways. If we place $v$ at $(1,0)$ or $(0,1)$, then we can place $w$ and $w'$ in 6 ways. These are together only 16 ways.

(4.1) Some neighbor of $(0,0)$ is occupied, say $(-1,0)$. Then $w$ cannot be placed on $(-2,0)$.

(4.2) Some diagonal neighbor of $(0,0)$ is occupied, say $(-1,-1)$. Then, trivially, we cannot occupy it by $w$, too.

Thus, in case (4) we can place $w$ on at most 7 points. Hence 2 new vertices, $v$ and $w$. are placed in at most 7 ways. This yields a branching number at most $\sqrt{7}$ here, too. It is correct to include $v$ already in the calculation of the branching number, even if $v$ is undecided, provided that we can decide the points of all undecided vertices afterwards in polynomial time. We study this matter below.

From now on we consider any output generated by the branching algorithm. That is, decided vertices are already mapped to specific grid points, whereas undecided vertices are not.

**Undecided vertices of degree 2.**

Consider any undecided vertex $v$ of degree 2, and let $u$ and $w$ be its adjacent vertices, according to the notation used above. Then $u$ and $w$ are decided vertices whose grid points are diagonal neighbors. Let $d_v$ denote the other diagonal of the square (4-cycle) of the grid that comprises $u$ and $w$. The candidate points for $v$ are the two end points of $d_v$. Whenever one end point of $d_v$ is already occupied by some vertex, we place $v$ at the other end point and mark $v$ as decided. This step is iterated exhaustively. For placing the remaining undecided vertices of degree 2, we use an auxiliary graph:

**Definition 5** *For the given graph $G$ and for the considered layout of the decided vertices, we define a graph $D$ as follows. Its edges are the diagonals $d_v$, for all undecided vertex $v$ of degree 2, and its vertices are the end points of all the diagonals $d_v$.*

*Let $p$ and $q$ denote the end points of any diagonal $d_v$. When we decide to place $v$ at $p$ ($q$), we turn the undirected edge $pq$ of $D$ into a directed edge $\overrightarrow{pq}$ ($\overrightarrow{qp}$), in other words, we orient the edge $d_v$ of $D$ away from the point of $v$.*

Thus, the possible placements of the undecided vertices of degree 2 correspond exactly to the orientations of the edges of $D$ where every vertex has at most one outgoing directed edge. We call them *unique-exit* orientations.

A connected graph is *unicylic* if it has exactly one cycle, or equivalently, it is a cycle, possibly with trees attached. For brevity, a *tree component* or *unicyclic component* of a graph is a component being a tree or unicyclic, respectively.

**Lemma 1** *A graph $D$ has a unique-exit orientation if and only if every component of $D$ is a tree or unicyclic. Furthermore, all unique-exit orientations are*

*obtained as follows. In every tree component, pick an arbitrary root and direct all edges towards the root. In every unicyclic component, choose any of the two orientations of the cycle and direct its edges accordingly, and direct all edges of the attached trees towards the cycle.*

**Proof:** The described orientations are obviously unique-exit. Conversely, consider an arbitrary unique-exit orientation.

Let $r$ be any vertex without outgoing edge, and let $R$ be the subgraph containing $r$ and all adjacent vertices. All edges incident to $r$ are directed towards $r$, hence no further edges exist between the vertices of $R$. In particular, $R$ is a tree where all edges are directed towards $r$, and $r$ is the only vertex without an outgoing edge. Next, consider any such tree $R$ and add another edge $\overrightarrow{st}$ from $D$, with $s \in R$ or $t \in R$. The case $s \in R$ is impossible, since $s \neq r$, and $s$ has already an outgoing edge. Hence only $t \in R$, and no other edge $\overrightarrow{su}$ or $\overrightarrow{us}$ with $u \in R$ can exist. By an inductive argument, the component of $D$ containing $r$ must be such a tree.

Finally consider any component where every vertex has exactly one outgoing edge. Starting in any vertex and following a directed path we find a directed cycle $C$. Let $r$ be any vertex on $C$. Since $r$ has an outgoing edge in $C$, all other incident edges are directed towards $r$. In particular, $C$ has no chords (i.e., additional edges joining non-consecutive vertices). By the same argument as above, all further edges must form trees attached to $C$. □

**Undecided vertices of degree 1.**

Let $p$ be any grid point that is neither occupied by a decided vertex nor contained in $D$. We consider any such $p$ as a tree component consisting of just $p$ and without edges. This allows a convenient formulation of the remaining problem of placing the undecided vertices $v$ of degree 1. Recall that every such $v$ is adjacent to exactly one vertex $u$ of $G$, $u$ is decided, and the candidate points for $v$ are the free neighbors of the point of $u$.

We define another auxiliary graph $B$ which is bipartite: The vertices on its two sides represent the undecided vertices $v$ of degree 1, and the tree components $T$, respectively. Any $v$ and $T$ are adjacent in $B$ if and only if $T$ contains some candidate point where $v$ may be placed.

Every tree component $T$ can be used by at most one vertex $v$. This is trivial if $T$ is a single point, and for other $T$ this follows from Lemma 1 as said above. That is, the possible placements of the undecided vertices of degree 1 correspond exactly to the matchings in $B$ that cover all vertices $v$. As is well known, the bipartite maximum matching problem can be solved in polynomial time [12]. (We can also use the plain Ford-Fulkerson algorithm which runs here in $O(n^2)$ time, since the degrees of vertices on one side of $B$ are bounded by 3, such that $B$ has only $O(n)$ edges.)

**Overall algorithm.**

To be explicit, we summarize the resulting overall algorithm. First run the branching algorithm, leaving some vertices undecided in every branch. For every

placement of the decided vertices continue as follows. Construct the graph $D$. In every unicyclic component $C$, choose (arbitrarily) one unique-exit orientation and place the undecided vertices of $G$ therein according to Definition 5. Next, find some maximum matching in the graph $B$. Place every undecided vertex $v$ in the tree component $T$ assigned by that matching, orient $T$ towards the point of $v$, and place the remaining undecided vertices according to Definition 5.

We see that also this algorithm is enumeration-based: It enumerates in $O^*(2.6458^n)$ time certain partial layouts, and for placing the remaining undecided vertices, the components of the auxiliary graph $D$ are treated independently. If the input graph $G$ happens to have minimum degree 2, this is even a succinct enumeration of all ideal layouts of $G$. A detail is that the set of grid points occupied by every unicyclic component is uniquely determined.

To get such a succinct enumeration in the general case, we must also branch on the yet undecided vertices $v$ of degree 1. For every $v$ with at most 2 possible positions, this can be trivially done with branching number 2. Every $v$ with 3 possible positions is adjacent to only one vertex which has degree 2. By simple combinatorics, a connected graph of maximum degree 4 has at most $0.4n$ vertices $v$ of this kind. (It may be possible to prove a sharper bound in partial grid graphs.) In the worst case, all vertices of degree 1 are appended with branching number 3. This yields a time bound $O^*(\sqrt{7}^{0.6n} \cdot 3^{0.4n}) = O^*(2.7822^n)$.

# 7 Connected Graphs Filling a Grid

**Theorem 6** *Given a grid and a connected graph $G$, we can enumerate, in $O^*(2.6208^\nu)$ time, all ideal layouts of $G$ that fit in the grid.*

**Proof:** We first place one edge, in $O(\nu)$ ways, and mark its two vertices as active. Then we pick any active vertex $u$, branch on the positions of *all* its remaining neighbors, mark them as active, and mark $u$ as passive. This step is iterated until all vertices are passive. Since $G$ is connected, $G$ is then placed entirely. This is the whole algorithm.

For the analysis, let $w$ be some positive constant to be fixed later. In the grid graph, we give every point the weight $w$ and every arc the weight 1. Since the grid has $\nu$ points and at most $2\nu$ arcs, its total weight is at most $(w+2)\nu$. We say that a *point is settled* if some vertex is placed on it. We say that an *arc is settled* if either some edge of $G$ is placed on it, or we have decided not to use this arc for hosting any edge.

Whenever the algorithm processes an active vertex $u$, we decide in every branch on all neighbors of $u$, hence an incident arc not used now will never be used later on. That is, while we settle only the points where we place the remaining neighbors of $u$, we settle all arcs incident to the point of $u$.

We use the following notations: $u$ is the active vertex processed in a step, $k$ is the number of neighbors of $u$ yet to place, $A$ is the number of ways to place them. and $m$ is the number of yet unsettled arcs incident to $u$ (which is also the number of free neighbors of the point of $u$ in the grid). The branching number

of a step is then $A^{1/(kw+m)}$. This leads to the following cases and branching numbers:

$k=3$, $m=3 \Longrightarrow 6^{1/(3w+3)}$; $k=2$, $m=3 \Longrightarrow 6^{1/(2w+3)}$;
$k=1$, $m=3 \Longrightarrow 3^{1/(w+3)}$; $k=2$, $m=2 \Longrightarrow 2^{1/(2w+2)}$;
$k=1$, $m=2 \Longrightarrow 2^{1/(w+2)}$; $k=1$, $m=1 \Longrightarrow 1$.

Clearly, for any fixed $w$, the maximum can only be some of $6^{1/(2w+3)}$, $3^{1/(w+3)}$, and $2^{1/(w+2)}$. Since points and arcs of a total weight no larger than $(w+2)\nu$ must be settled, the base in the time bound is at most the maximum of $6^{(w+2)/(2w+3)}$, $3^{(w+2)/(w+3)}$, and $2^{(w+2)/(w+2)}$. The latter number is simply 2, and by choosing $w := 5.127$, numerical calculation yields the claimed branching number. □

# 8 Rigid Partial Grid Graphs

We call a partial grid graph $G$ *rigid* if $G$ admits exactly one ideal layout, up to translations, reflections, and rotations through multiples of 90 degrees. In this case we call it the *unique layout* of $G$. Note that "rigid" implies "partial grid graph" in our terminology. Remember that the vertices have individual names, that is, we do distinguish between layouts that are congruent but place some vertices at different distances, due to automorphisms of the graph. For instance, a 4-cycle is rigid, but a 4-cycle with a 5th vertex of degree 1 attached has two different ideal layouts and is therefore not rigid.

Given the motivation from Section 2, we aim in the following at efficient algorithms for finding large rigid subgraphs of a given graph $G$, or at least certain types of rigid subgraphs. Due to the known NP-hardness of partial subgraph recognition, it is important that such algorithms do not rely on the assumption that $G$ is a partial grid graph. Rather, they have to identify rigid (partial grid) subgraphs of any input graph $G$.

To get a first idea how rigid graphs look, let us enumerate the smallest ones. For this purpose it is helpful to exclude some classes of graphs that cannot be rigid. The following fact might also be interesting in its own right.

**Theorem 7** *No tree with more than two vertices is rigid.*

**Proof:** Assume that $G$ is a tree with more than two vertices, and $G$ is rigid. Consider its unique layout. We work with an $x-y$ coordinate system. Without loss of generality, the diagonal $y=x$ contains at least one vertex $u$ of $G$, and no vertex is in the halfplane above the diagonal. (Otherwise we can move the embedding of $G$ accordingly.) Since all neighbors of $u$ are below the diagonal, $u$ has degree 1 or 2.

Assume that $u$ has degree 2, and let $v$ and $w$ denote the two neighbors of $u$. The tree $G$ without $u$ consists of two subtrees $G_v$ and $G_w$, containing $v$ and $w$, respectively. We reflect one of them, say $G_w$, through the diagonal $y=x$. Reflection through a diagonal maps grid points to grid points. The distance of $u$ and $w$ remains 1. Furthermore, vertices of $G_w$ on the diagonal

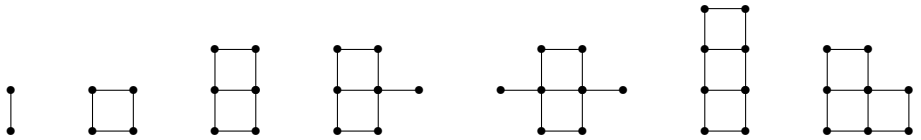


Figure 1: All rigid graphs with up to 8 vertices.

do not change their positions, and all vertices of $G_w$ below the diagonal end up above the diagonal, where they cannot collide with vertices of $G_v$. Finally, the distance of $v$ and $w$ changes from $\sqrt{2}$ to 2. Hence $G$ is another valid layout. This contradicts our assumption that $G$ is rigid.

It follows that $u$ has degree 1. Without loss of generality, $u$ is on the point $(0, 0)$, and its unique neighbor $v$ is on $(1, 0)$. Since $G$ is a connected partial grid graph with more than two vertices, not all vertices other than $u$ can be on the straight line $y = -x + 1$ going through $v$. In other words, there exists a vertex on some grid point $(x, y) \neq (0, 0)$ with $y \neq -x + 1$. Note that the Euclidean distances from $(x, y)$ to $(0, 0)$ and $(1, 1)$ are different.

Assume that no vertex is on the point $(1, 1)$. Then we can place $u$ alternatively on $(1, 1)$. Since this changes at least one Euclidean distance, we have got another layout, contradicting again the rigidity of $G$. Hence there is some vertex $u'$ on $(1, 1)$. We have already seen that all vertices on the diagonal $y = x$ have degree 1. Assume that the unique neighbor of $u'$ is $v$ on $(1, 0)$. Then we can swap the positions of $u$ and $u'$, and exactly as above this yields another layout and a contradiction. It follows that the neighbor $v'$ of $u'$ is on $(2, 1)$.

To summarize, we have shown that, for any vertex of degree 1 on the diagonal $y = x$ that has its unique neighbor to the right, there exists another such pair of vertices one row higher. But this yields an infinite sequence of vertices, and this final contradiction concludes the proof. □

Theorem 7 says that every rigid graph must contain a cycle. Using this fact and simple case inspections we see that the list of rigid graphs with up to 8 vertices in Figure 1 is complete.

We notice in Figure 1 that some rigid graphs are obtained from smaller ones by adding new vertices of degree 1. More precisely, we call an edge $uv$ a *hair* if $u$ has degree 3 and $v$ has degree 1. Then we have: If $H$ is rigid, $u$ has degree 3 in $H$, and in the unique layout of $H$, one of the four neighbors of the grid point of $u$ is still free, then we can add a hair $uv$ (with a fresh vertex $v$ that was not in $H$) and still obtain a rigid graph. This is obviously true, since $v$ can be added to the layout of $H$ in only one way. We call this operation a *hair extension* of the rigid graph $H$.

We also notice a second, much more powerful extension operation: Consider again a rigid graph $H$ and its unique layout. Informally, let $u$ and $v$ be two vertices of $H$, with the property that there exists exactly one shortest $u - v$ path $P$ in the grid without the points occupied by $H$. Then we can attach $P$

to $H$ and obtain a graph which is still rigid, since $P$ can be added to the layout of $H$ in only one way. We call this operation a *unique shortest path (USP) extension* of the rigid graph $H$. Interestingly, all graphs in Figure 1 (apart from the trivial graph with 2 vertices) can be obtained from a 4-cycle by a series of hair and USP extensions. More generally, one can give an intuitive explanation why probably "most" rigid graphs are built up in this way. In any case, the above operations yield a rich class of rigid graphs.

This suggests the following method for finding certain rigid subgraphs of $G$, of arbitrary sizes: For any rigid subgraph of $G$ that is already detected, we compute all possible hair and USP extensions, then we check whether the corresponding hairs and paths really exist in $G$, and if so, we obtain larger rigid subgraphs. The crucial point is that unique shortest paths can be found efficiently. In the following sections we develop this idea in detail.

Some more graph-theoretic terminology will be needed. Symbol $G[U]$ denotes the subgraph of $G = (V, E)$ induced by a subset $U \subseteq V$ of vertices. The *union* of two induced subgraphs $G[X]$ and $G[Y]$ is defined as $G[X \cup Y]$.

An $u - v$ *path* is a path whose ends are the vertices $u$ and $v$. The *length* of a path is the sum of its edge lengths, or the number of edges, if the edges have unit length. The *distance* of vertices $u$ and $v$ in a graph is the length of a shortest $u - v$ path. If we mean instead the Euclidean distance of two vertices placed in the grid, we say this explicitly.

The single-source shortest path problem in graphs with $n$ vertices and $m$ edges with positive lengths can be solved by a variant of Dijkstra's algorithm in $O(m + n \log n)$ time [11]. If all edges have unit length, Dijkstra's algorithm becomes *breadth-first search (BFS)* and needs only $O(m)$ time.

Recall that grid points at Euclidean distance 1 are *neighbors* in the grid. We call a set of points in the plane *collinear* if all its points are on one straight line. Without risk of confusion we may identify vertices of an embedded graph with the grid points they are mapped to.

## 9 Unique Shortest Path Extensions

Let $H$ be some rigid graph with $h$ vertices. We fix one embedding of $H$ in the grid. Let $\overline{H}$ be the graph defined as follows: Its vertices are all grid points, and any two points with Euclidean distance 1 are joined by an edge, unless both points are occupied by vertices of $H$. Informally, $\overline{H}$ is the entire grid but without any edges between the vertices of $H$. This is an infinite graph, but only some finite subgraph around $H$ will be relevant later on. We also remark that $\overline{H}$ is uniquely determined up to isomorphism.

Given any graph $G = (V, E)$ and a set $T \subset V$ of *terminal* vertices, we say that a pair $\{u, v\}$ of distinct terminal vertices $u, v \in T$ is *tight* if, among all $u - v$ paths whose internal vertices are not in $T$, exactly one path has minimum length. For brevity we call it the *unique shortest $u - v$ path*, when $T$ is clear from context.

**Lemma 2** *Let $H$ be a rigid graph, and let $u$ and $v$ be two vertices of $H$. Suppose that $\{u, v\}$ is a tight pair in $\overline{H}$, with the vertex set of $H$ as the set of terminal vertices. Let $p$ denote the length of the unique shortest $u - v$ path in $\overline{H}$. Then, the graph $H$ extended by some $u - v$ path $P$ of length $p$, whose internal vertices are not in $H$, is still a rigid graph.*

**Proof:** Let $Q$ denote the unique shortest $u - v$ path in $\overline{H}$. The only possible way to add $P$ to the unique layout of $H$ is to follow the path $Q$, since all other $u - v$ paths in $\overline{H}$ whose interval vertices are not in $H$ are longer. This implies rigidity. □

As already mentioned, we call the extension of a rigid graph by a path, as described in Lemma 2, an *USP extension*. Suppose that we have already computed the tight pairs $\{u, v\}$ of vertices of $H$ in $\overline{H}$, and the lengths of their unique shortest $u - v$ paths. Then we also know all possible USP extensions of $H$, that is, we know the pairs $\{u, v\}$ and the lengths of the $u - v$ paths to attach. In the following we show how to compute this information. A variant of Dijkstra's shortest path algorithm yields:

**Theorem 8** *Suppose that we are given a graph with $n$ vertices and $m$ edges, positive edge lengths, and a set $T$ of terminal vertices. For any fixed vertex $u \in T$, we can compute all tight pairs $\{u, v\}$ and the lengths of their unique shortest $u - v$ paths, in $O(m + n \log n)$ time. In graphs with unit edge lengths, this takes only $O(m)$ time. Moreover, the unique shortest $u - v$ paths form a tree rooted at $u$.*

**Proof:** First we run Dijkstra's algorithm with source $u$, in the graph where all other terminal vertices and their incident edges are removed. We say that a pair $\{u, v\}$ (where $v \notin T$) is *firm* if there exists exactly one shortest $u - v$ path in this graph. Let $\ell(u, v)$ denote the given length of the edge $uv$ (if it exists), and let $d(u, v)$ denote the distance of $u$ and $v$ in the mentioned graph. Observe that $\{u, v\}$ is firm if and only if there exists a unique vertex $w$ with $d(u, v) = d(u, w) + \ell(w, v)$, and moreover, either $u = w$ or $\{u, w\}$ is firm. Based on this equivalence we can mark the firm pairs during execution of Dijskstra's algorithm. Also remember that Dijskstra's algorithm is just BFS in the case of unit edge lengths.

Finally we do the following separately for any single terminal vertex $v \in T$: We re-insert $v$ and its incident edges. Similarly as above, $\{u, v\}$ is tight if and only if there exists a unique vertex $w$ with minimum sum $d(u, w) + \ell(w, v)$, and moreover, either $u = w$ or $\{u, w\}$ is firm. Furthermore, this minimum sum is then the length of the unique shortest $u - v$ path without internal vertices in $T$. After checking whether $\{u, v\}$ is tight, we remove $v$ again and proceed to the next vertex of $T$. This costs $O(m)$ additional time in total.

The unique shortest paths form a tree, since every vertex $v$ such that $\{u, v\}$ is firm has a unique predecessor $w$. □

Let $\overline{H}^*$ be the finite subgraph of $\overline{H}$, consisting of the smallest axis-parallel rectangle that encloses the embedded graph $H$, plus a margin of one line of grid

points at all four sides. Remember that $\overline{H}$ (and thus $\overline{H}^*$) does not contain any edges between the vertices of $H$, which are our terminal vertices.

**Lemma 3** *The same pairs $\{u, v\}$ of vertices of $H$ are tight in $\overline{H}^*$ and in $\overline{H}$, and for each of them, the length of the unique shortest $u - v$ path is the same in both graphs.*

**Proof:** Let $u$ and $v$ be two vertices of $H$, and let $Q$ be any path from $u$ to $v$ in $\overline{H}$, whose inner vertices are not from $H$. If $Q$ leaves and re-enters $\overline{H}^*$, then we can obviously replace the sub-path of $Q$ outside $\overline{H}^*$ with a shorter path along the border of $\overline{H}^*$. This contradiction shows that any shortest $u - v$ path must be entirely in $\overline{H}^*$. Both assertions follow. □

Since $\overline{H}^*$ has $O(h^2)$ vertices and edges, we can now compute all tight pairs and their path lengths in $O(h \cdot h^2) = O(h^3)$ time, using Theorem 8 and Lemma 3.

Next, let $G[U]$, $U \subset V$, be some induced subgraph of our input graph $G = (V, E)$, and suppose that $G[U]$ is rigid, and its unique layout is already computed. Let $G_U$ be the graph obtained from $G$ by deleting all edges (but not the vertices) of $G[U]$. Let $H := G[U]$ and $h := |U|$. We compute all possible USP extensions of $H$ in $O(h^3)$ time, in the way shown above. For every tight pair $\{u, v\}$ of vertices of $H$ we check whether $\{u, v\}$ is tight also in $G_U$, with $U$ as the set of terminal vertices. If this is the case, and if the unique shortest $u-v$ path in $G_U$ has the same length as in $\overline{H}^*$, we can attach this path to $G[U]$ and obtain a larger subgraph of $G$ that is still rigid, due to Lemma 2.

For the sake of completeness we remark that the possible "negative" cases are: $\{u, v\}$ is not tight in $G_U$, or it is tight, but the unique shortest $u - v$ path is longer than in $\overline{H}^*$. In these cases we cannot apply an USP extension to $G[U]$ at the pair $\{u, v\}$. If the $u - v$ path in $G_U$ is shorter than in $\overline{H}^*$, or if several shortest $u-v$ paths exist in $G_U$, then we can conclude that $G$ was not a partial grid graph.

All tight pairs of vertices of $U$ and the lengths of their unique shortest paths in $G_U$ can be computed in $O(hn)$ time using Theorem 8 again, since $G_U$ has $O(n)$ vertices and edges, or $G$ is not a partial grid graph. With $h \leq n$, this shows altogether:

**Theorem 9** *Given a graph $G$, a rigid induced subgraph $G[U]$, and its unique layout, we can compute all rigid subgraphs of $G$ that are USP extensions of $G[U]$ (or recognize that $G$ is not a partial grid graph) in $O(n^3)$ time.*

We conjecture that the time can be improved to $O(n^2)$, with a slightly modified goal. These are the issues: There can exist $O(n^2)$ tight pairs, and every unique shortest path may have length $O(n)$. However, $G$ has only $n$ vertices in total. Instead of returning all these paths separately, we could just return the union of their vertex sets, which equals the union of the trees from Theorem 8 for all roots $u \in U$. Then we can take the subgraph induced by $U$ and this union. This graph is also rigid. The caveat is that this still costs $O(n \cdot n^2)$ time if we run BFS in the graph $\overline{G[U]}^*$ that can have size $O(n^2)$. We

conjecture that each tree can be computed in $O(n)$ time, by using some more geometry of the grid. This would result in $O(n^2)$ overall time.

Actually, paths in grids with some forbidden vertices (those in $H$) are a special case of rectilinear shortest paths in the plane with rectilinear obstacles, where both the line segments of the paths and the sides of the obstacles must be axis-parallel. The latter problem is well studied and very efficiently solvable, as shown in [6] and subsequent work. However, the algorithms are complex, and it is not clear whether they can be adapted such that also the (non-)uniqueness of shortest paths can be recognized.

# 10 Iterated Extensions of Rigid Graphs

Our goal was to find large rigid subgraphs of a graph $G$ efficiently. To this end we may iterate, as long as possible, an extension procedure as the one from Theorem 9. The question is how much time this costs in total. Since the following considerations do not depend on the details of the chosen extension procedure, it is simpler and cleaner to consider a generic extension operation from now on.

**Definition 10** *An* extension operation $Ext$ *takes as input a graph* $G = (V, E)$ *and a rigid induced subgraph* $G[U]$, $U \subset V$, *along with its unique layout.* $Ext$ *finds some vertex set denoted* $\epsilon(U)$, *such that* $U \subseteq \epsilon(U) \subseteq V$ *and the graph* $G[\epsilon(U)]$ *is rigid, and it computes the unique layout of* $G[\epsilon(U)]$. *Let* $t$ *denote a time bound for* $Ext$. *Furthermore, extensions have to be monotone, that is, if* $U \subseteq U'$ *then* $\epsilon(U) \subseteq \epsilon(U')$.

Note that $\epsilon(U) = U$ is possible, in which case $Ext$ does not find a proper extension (for instance, because $G[U]$ is already a maximal rigid subgraph). For notational convenience we write the time bound $t$ without arguments.

In addition we specify a set $S$ of rigid graphs called *start graphs.*

**Definition 11** *Given an extension operation* $Ext$, *a set* $S$ *of rigid start graphs, and a graph* $G$, *we define the family of* $(G, Ext, S)$-rigid *induced subgraphs of* $G$ *inductively as follows. All induced subgraphs of* $G$ *isomorphic to some start graph in* $S$ *are* $(G, Ext, S)$*-rigid. If* $G[U]$ *is* $(G, Ext, S)$*-rigid, then the result* $G[\epsilon(U)]$ *of applying* $Ext$ *to* $G[U]$ *is* $(G, Ext, S)$*-rigid.*

*Let* $s$ *denote a time bound for finding all induced subgraphs of* $G$ *being isomorphic to some start graph, and computing their unique layouts.*

**Example.** Let $ExtHUSP$ be the extension operation that applies all possible hair extensions and all possible USP extensions to $G[U]$ and takes the union of the resulting induced subgraphs. Then, e.g., all graphs in Figure 1 (except the single edge) are $(G, ExtHUSP, \{C_4\})$-rigid if they occur in $G$, where $C_4$ denotes the 4-cycle.

A simple observation is that all $(G, Ext, S)$-rigid subgraphs of $G$ can be enumerated in $O(stn)$ time: There are $O(s)$ start graphs (as they can be produced

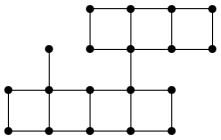


Figure 2: Assume that the displayed graph is an induced subgraph of $G$. This graph is rigid but not $(G, ExtHUSP, \{C_4\})$-rigid. However, it is the union of two $(G, ExtHUSP, \{C_4\})$-rigid graphs.

in time $s$ by definition), each of them can be extended at most $n$ times, and an extension takes time $t$. Another simple fact is:

**Lemma 4** *For any fixed finite set $S$ of start graphs we have $s = O(n)$.*

**Proof:** Since start graphs are rigid by definition, they are connected. In order to find all occurrences of start graphs in $G$, we do $n$ times BFS, starting from each of the vertices of $G$ as the root. Since the sizes of the start graphs are bounded by some constant, only vertices within some constant distance from the root must be reached. Since partial grid graphs have vertex degrees bounded by another constant (namely 4), the number of these vertices is also bounded by some constant, otherwise we recognize that $G$ is not a partial grid graph. Even when we extract the start graphs from these neighborhoods of constant size by naive exhaustive search, the time is constant for every root. □

Thus we have $O(stn) = O(tn^2)$ in this case. In the following section we will apply, besides extensions, also pairwise unions of rigid subgraphs. We will see that this enables us to find more rigid induced subgraphs than by just extensions, without increasing the overall time. Figure 2 shows a motivating example from which the reader may recognize the general reason why unions enhance the possibilities to detect rigid graphs.

# 11 Combining Extensions and Unions

We need some small preparatory lemmas about unions of rigid graphs.

**Lemma 5** *For $X, Y \subset V$, suppose that $G[X]$ and $G[Y]$ are rigid, and their unique layouts are given. Then we can check whether $G[X \cup Y]$ is rigid, and if so, compute its unique layout, in $O(n)$ time.*

**Proof:** We check in $O(n)$ time that $G[X \cup Y]$ is connected (otherwise it cannot be rigid). Fix an embedding of $G[X]$ in the grid. Due to connectivity, $G[Y]$ can be added to this embedding in only $O(1)$ ways. Some of these embeddings of $G[Y]$ may collide with that of $G[X]$, but it is straightforward to check in $O(n)$ time whether any vertices from $X$ and $Y$ occupy the same point. Finally, $G[X \cup Y]$ is rigid if and only if exactly one of the mentioned $O(1)$ candidate embeddings of $G[Y]$ yields a layout of $G[X \cup Y]$. □

**Lemma 6** *For $X, Y \subset V$, suppose that $G[X]$ and $G[Y]$ are rigid, and that $X \cap Y$ is not collinear (in the unique layout of $G[X]$ or $G[Y]$). Then $G[X \cup Y]$ is rigid, or it is not a partial grid graph.*

**Proof:** Fix an embedding of $G[X]$ in the grid. Since $X \cap Y$ is not collinear, $G[Y]$ can be added to this embedding in at most one way. □

**Lemma 7** *If $X \subseteq Y$ and $G[X]$ is rigid, then every ideal layout of $G[Y]$ contains the unique layout of $G[X]$. Consequently, if both $G[X]$ and $G[Y]$ are rigid, then the unique layout of $G[Y]$ contains the unique layout of $G[X]$.*

**Proof:** The assertion is trivial. Since $G[X]$ has only one ideal layout, this layout cannot change by adding the vertices of $Y \setminus X$. □

The next lemma is slightly more complex.

**Lemma 8** *Suppose that $G$ is a partial grid graph. Let $R$ be a family of induced subgraphs of $G$ such that every graph in $R$ is rigid but the union of any two of them is not rigid. Then the number of graphs in $R$ is $O(n)$.*

**Proof:** We make three observations:

(a) Let a *corner* be any set of three grid points within one square of the grid, in other words, three grid points with Euclidean distances 1, 1, and $\sqrt{2}$. Every rigid graph $H$ is connected, and an embedding of $H$ cannot be merely a path on a horizontal or vertical line (since a path is not a rigid graph). Thus, the unique layout of $H$ must contain some corner.

(b) Let $T$ be any triple of vertices in $G$, let $H, H' \in R$ be any two graphs that contain $T$, and assume that $T$ is a corner in the unique layout of $H$ and $H'$, respectively. By Lemma 6, their union is rigid, which contradicts our assumption on $R$. Thus, at most one graph in $R$ has the following property: $H$ contains $T$, and $T$ is a corner in the unique layout of $H$.

(c) Let us fix one layout of $G$. Every triple $T$ of vertices that is a corner in the unique layout of some rigid subgraph of $G$ is also a corner in our layout of $G$, due to Lemma 7. Trivially, a layout of $G$ contains only $O(n)$ corners. Thus only $O(n)$ triples $T$ can be corners in rigid subgraphs.

The conclusions of (a),(b),(c) together imply the assertion. □

We enhance the concept from Definition 11 as follows. The number 2 indicates unions of two rigid graphs.

**Definition 12** *Given an extension operation $Ext$, a set $S$ of start graphs, and a graph $G$, we define the family of $(G, Ext, 2, S)$-*rigid *induced subgraphs of $G$ inductively as follows. All induced subgraphs of $G$ isomorphic to some start graph in $S$ are $(G, Ext, 2, S)$-rigid. If $G[U]$ is $(G, Ext, 2, S)$-rigid, then the result $G[\epsilon(U)]$ of applying $Ext$ to $G[U]$ is $(G, Ext, 2, S)$-rigid. If $G[X]$ and $G[Y]$ are $(G, Ext, 2, S)$-rigid and $G[X \cup Y]$ is rigid, then $G[X \cup Y]$ is $(G, Ext, 2, S)$-rigid.*

Now we define and analyze the following procedure. For convenience, by a graph "larger than" $H$ we mean a graph containing $H$ as an induced subgraph.

**Rigid(G,Ext,2,S):** List all induced subgraphs of $G$ being isomorphic to start graphs in $S$. Then apply the following steps in any order: Either take any graph $G[U]$ from the list and replace it with the result $G[\epsilon(U)]$ of $Ext$, or take any two graphs from the list and test whether their union is rigid, and if so, replace them with their union. Stop when none of the extensions and pairwise unions generates a new rigid subgraph.

**Lemma 9** *Suppose that $G[U]$, $U \subseteq V$, is a $(G, Ext, 2, S)$-rigid graph. Then the procedure Rigid$(G, Ext, 2, S)$ generates (among other graphs) at least one $(G, Ext, 2, S)$-rigid graph $G[U']$, $U' \supseteq U$. In other words, $G[U]$ or some larger $(G, Ext, 2, S)$-rigid graph appears in the list after termination.*

**Proof:** Every step of Rigid$(G, Ext, 2, S)$ replaces graphs with larger graphs, unless an extension step produces the same graph; this monotonicity property is tacitly used in the following.

Let $h$ be the number of operations (i.e., extensions and unions) needed to build $G[U]$ from start graphs. We prove the assertion by induction on $h$. The induction base $h = 0$ is trivial, since Rigid$(G, Ext, 2, S)$ generates all start graphs appearing as induced subgraphs of $G$.

For the induction step, fix some $h > 0$ and suppose that the assertion is true for all numbers smaller than $h$. Our graph $G[U]$ is an extension of some $(G, Ext, 2, S)$-rigid graph $G[X]$ (case 1) or the union of some $(G, Ext, 2, S)$-rigid graphs $G[X']$ and $G[X'']$ (case 2), where the mentioned graphs are built with fewer than $h$ operations. By the induction hypothesis, Rigid$(G, Ext, 2, S)$ generates, in both cases, the mentioned graphs or larger graphs. We denote them $G[Y]$, $G[Y']$, $G[Y'']$, respectively, where $Y \supseteq X$, $Y' \supseteq X'$, $Y'' \supseteq X''$.

*Case 1:* If $Y \supseteq U$, we are done. Otherwise, the extension of $G[Y]$ contains the extension of $G[X]$ which is $G[U]$. If Rigid$(G, Ext, 2, S)$ generates the extension of $G[Y]$, we are done. If it does not, then it takes the union of $G[Y]$ with some other graph. We rename the union and denote this strictly larger graph $G[Y]$ again. Since this step can be iterated only finitely often, we eventually arrive at some $Y \supseteq U$.

*Case 2:* Since $Y' \cup Y'' \supseteq X' \cup X'' = U$, the graph $G[Y' \cup Y'']$ includes $G[U]$. Since $G[X']$, $G[X'']$, $G[X' \cup X'']$, $G[Y']$, $G[Y'']$ are all rigid, we also get that $G[Y' \cup Y'']$ is rigid: We apply Lemma 7 to $X' \subseteq Y'$ and to $X'' \subseteq Y''$, and we use that $G[X' \cup X'']$ is rigid. It follows that the unique layout of $G[X' \cup X'']$ can be extended to a layout of $G[Y' \cup Y'']$ in only one way. Remember that Rigid$(G, Ext, 2, S)$ generates $G[Y']$ and $G[Y'']$, and as we have seen, $G[Y' \cup Y'']$ includes $G[U]$ and is rigid. If Rigid$(G, Ext, 2, S)$ generates the union of $G[Y']$ and $G[Y'']$, we are done. If it does not, it generates the extension of some of them, say of $G[Y']$. Then we rename the extension and denote this strictly larger graph $G[Y']$ again. Since this step can be iterated only finitely often, we eventually arrive at $Y' \supseteq U$ or $Y'' \supseteq U$, or Rigid$(G, Ext, 2, S)$ generates their union. □

**Lemma 10** *After termination of Rigid$(G, Ext, 2, S)$, the list of graphs contains exactly the maximal $(G, Ext, 2, S)$-rigid subgraphs of $G$ (i.e., maximal with respect to inclusion), regardless of the order of operations.*

**Proof:** Let $G[U]$ be any maximal $(G, Ext, 2, S)$-rigid subgraph of $G$. Due to Lemma 9, $G[U]$ or some larger $(G, Ext, 2, S)$-rigid graph is in the final list, but the latter case is not possible, as $G[U]$ is already maximal.

Let $G[U']$ be a non-maximal $(G, Ext, 2, S)$-rigid subgraph of $G$. Then $G[U']$ is contained is some maximal $(G, Ext, 2, S)$-rigid subgraph $G[U]$, and as just said, $G[U]$ is in the final list. Since Rigid$(G, Ext, 2, S)$ performs all possible union operations that have rigid results, $G[U']$ and $G[U]$ would be replaced with another copy of $G[U]$, or in simpler words, $G[U']$ would be removed from the list, by this or another operation. □

**Theorem 13** *We can generate all maximal $(G, Ext, 2, S)$-rigid subgraphs of a given partial grid graph $G$ in $O(s^2 + tn^2 + n^4)$ time.*

**Proof:** We run Rigid$(G, Ext, 2, S)$, however, we first perform the operations in a special order. (Due to Lemma 10, the final list does not depend on the order of operations.) We split the list in two parts denoted $L_1$ and $L_2$, which are initially empty. We define a potential, which is twice the number of graphs in $L_1$ plus the number of graphs in $L_2$.

First we generate all subgraphs of $G$ being isomorphic to start graphs, and we put them in $L_1$. Since this takes time at most $s$ by definition, the sum of the vertex numbers of these graphs is $O(s)$, and so is the potential at this moment.

We take some graph $H$ from $L_1$ and test for all subgraphs $H'$ in both $L_1$ and $L_2$ whether the union of $H$ and $H'$ is rigid. Lemma 5 implies that this costs $O(s)$ time in total, for the chosen graph $H$. As soon as we find such $H'$, we stop and replace $H$ and $H'$ with their union and put it in $L_1$. If no such $H'$ exists, we just move $H$ to $L_2$. We call this procedure with $H$ a *union test*.

Union tests are iterated until $L_1$ is empty. Note that every union test strictly decreases the potential. Thus we can do only $O(s)$ union tests, and the entire phase needs $O(s^2)$ time. Moreover, it is an invariant that the graphs in $L_2$ have pairwise non-rigid unions.

Due to Lemma 8, only $O(n)$ graphs remain in the list, and trivially, this remains true also after further operations. From now on, the order of operations, i.e., extensions and union tests, is arbitrary. Each of the $O(n)$ graphs in the list is involved in $O(n)$ further extension operations that produce larger graphs, and every such operation costs time $t$. Multiplication yields $O(tn^2)$ total time for all extensions. Whenever we extend a graph in $L_2$, we move it to $L_1$, (This is necessary in order to maintain the above invariant, because a larger graph may be capable of new unions that are rigid.) Since every extension increases the potential by at most 1, the total increase of the potential is $O(n^2)$, which also bounds the number of further union tests by $O(n^2)$. Now every union test costs only $O(n^2)$ time, due to the number of remaining graphs (which may overlap) and their sizes. This yields the $O(n^4)$ term. □

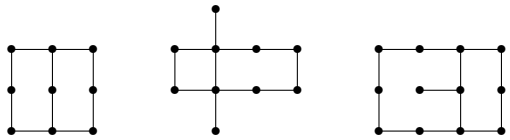


Figure 3: The smallest rigid graphs that cannot be obtained by USP and hair extensions of rigid subgraphs. In fact, they do not contain any proper rigid subgraphs.

Since recognizing partial grid graphs is NP-hard, we have to discuss how to apply Theorem 13 to a general input graph $G$. If the union of two subgraphs turns out not to be a partial grid graph, we know that $G$ was not a partial grid graph, and we may abort the procedure, as we are only interested in the positive case (and we may leave the result undetermined in the negative case).

We conjecture that the time bound in Theorem 13 can be improved by a more sophisticated analysis. Anyway, for $ExtUSPH$ with any fixed set of start graphs, the current bound becomes $O(tn^2 + n^4) = O(tn^2)$, which is no worse than the time we needed for generating the $(G, ExtHUSP, S)$-rigid subgraphs only. Also remember that we know $t = O(n^3)$ for $ExtHUSP$, but $t = O(n^2)$ might be possible.

# 12 Conclusions

Due to our application that deals with graphs of rather limited size, we are interested in capturing most rigid graphs up to some size $n$ in an efficient way.

By case inspection we find that for $n \leq 9$, *all* rigid graphs with $n$ vertices are also $(G, ExtHUSP, 2, \{C_4\})$-rigid, except the first graph in Figure 3. The picture also shows the next largest rigid graphs that cannot be obtained by USP and hair extensions from smaller rigid graphs. Their rigidity is seen by ad-hoc arguments.

We may successively add some of them to the set of start graphs. However, Figure 3 suggests a more fruitful idea: Note that these graphs contain 6-cycles. The 6-cycle as such is not rigid, but it has only $r := 3$ different layouts. Once the "correct" layout of a 6-cycle is fixed, each of the graphs in Figure 3 can again be obtained by USP and hair extensions; the other two layouts yield collisions and are discarded.

Inspired by this observation, we may extend our approach and the algorithm from Theorem 13 as follows. For some small fixed integer $r$, we work with subgraphs that have at most $r$ different layouts (rather than rigid graphs where $r = 1$). Since $r$ layouts must be processed simultaneously, the time bound increases by a factor at least $r$ (also, the list of graphs in Theorem 13 may become longer), but we capture considerably more rigid subgraphs. The choice of parameter $r$ should depend on the input size. One could also try and invent

more (yet fast) extension operations to produce further rigid graphs missed out by $ExtHUSP$.

A full characterization of rigid partial grid graphs, and the complexity of recognizing them, or more generally, of finding a largest rigid subgraph in a given graph, is open. This is not too important for our application, but it remains a nice theoretical problem. For layout planning, the rigid subgraphs are not an end in itself, but only a heuristic means for accelerating the branching. Nevertheless it would be interesting to learn how the proposed methods perform in general, how many long edges must typically be allowed, etc. But this would require testing on considerable sets of real instances.

Another question of more theoretical interest is how far our exponential bounds on the number of ideal layouts, with or without undecided vertices, are away from lower bounds, that is, to construct "bad" graphs with as many ideal layouts as possible.

## Acknowledgments

This work has been done by the author in the role of a scientific advisor at the Fraunhofer-Chalmers Research Centre for Industrial Mathematics (FCC) in Göteborg. Support from FCC is greatly acknowledged. The author would also like to thank a number of people: Fredrik Ekstedt, Raad Salman, and Peter Mårdberg (FCC) for encouraging discussions of the practical aspects, anonymous reviewers of an earlier manuscript for the approach to subexponential time and for further literature hints, the anonymous reviewers of JGAA for other valuable suggestions and a correction, and the students Maxim Goretskyy and Jesper Jaxing for bringing up the initial idea of path extensions of rigid graphs in their master's thesis supervised by the author.

## References


[1] Z. Abel, E. D. Demaine, M. L. Demaine, S. Eisenstat, J. Lynch, and T. B. Schardl. Who needs crossings? hardness of plane graph rigidity. In S. P. Fekete and A. Lubiw, editors, *32nd International Symposium on Computational Geometry, SoCG 2016, June 14-18, 2016, Boston, MA, USA*, volume 51 of *LIPIcs*, pages 3:1–3:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.SoCG.2016.3.

[2] A. Ahmadi, M. S. Pishvaee, and M. R. A. Jokar. A survey on multi-floor facility layout problems. *Comput. Ind. Eng.*, 107:158–170, 2017. doi:10.1016/j.cie.2017.03.015.

[3] M. F. Anjos and M. V. C. Vieira. Mathematical optimization approaches for facility layout problems: The state-of-the-art and future research directions. *Eur. J. Oper. Res.*, 261(1):1–16, 2017. doi:10.1016/j.ejor.2017.01.049.

[4] S. Bereg. Certifying and constructing minimally rigid graphs in the plane. In J. S. B. Mitchell and G. Rote, editors, *21st ACM Symposium on Computational Geometry, SoCG 2005, Pisa, Italy, June 6-8, 2005*, pages 73–80. ACM, 2005. doi:10.1145/1064092.1064106.

[5] H. L. Bodlaender, J. Nederlof, and T. C. van der Zanden. Subexponential time algorithms for embedding h-minor free graphs. In I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.9.

[6] P. J. de Rezende, D. T. Lee, and Y. Wu. Rectilinear shortest paths in the presence of rectangular barriers. *Discrete Comput. Geom.*, 4:41–53, 1989. doi:10.1007/BF02187714.

[7] V. G. P. de Sá, G. D. da Fonseca, R. C. S. Machado, and C. M. H. de Figueiredo. Complexity dichotomy on partial grid recognition. *Theor. Comput. Sci.*, 412(22):2370–2379, 2011. doi:10.1016/j.tcs.2011.01.018.

[8] A. Drira, H. Pierreval, and S. Hajri-Gabouj. Facility layout problems: A survey. *Ann. Rev. Control*, 31(2):255–267, 2007. doi:10.1016/j.arcontrol.2007.04.001.

[9] Z. Fekete and T. Jordán. Rigid realizations of graphs on small grids. *Comput. Geom.*, 32(3):216–222, 2005. doi:10.1016/j.comgeo.2005.04.001.

[10] F. V. Fomin and D. M. Thilikos. New upper bounds on the decomposability of planar graphs. *J. Graph Theory*, 51(1):53–81, 2006. doi:10.1002/jgt.20121.

[11] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. doi:10.1145/28869.28874.

[12] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. doi:10.1137/0202019.

[13] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In R. L. Constable, R. W. Ritchie, J. W. Carlyle, and M. A. Harrison, editors, *6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 172–184. ACM, 1974. doi:10.1145/800119.803896.

[14] T. Jordán and Z. Szabadka. Operations preserving the global rigidity of graphs and frameworks in the plane. *Comput. Geom.*, 42(6-7):511–521, 2009. doi:10.1016/j.comgeo.2008.09.007.

[15] H. Maehara. Distances in a rigid unit-distance graph in the plane. *Discrete Appl. Math.*, 31(2):193–200, 1991. doi:10.1016/0166-218X(91)90070-D.

[16] J. Matoušek and R. Thomas. On the complexity of finding iso- and other morphisms for partial k-trees. *Discrete Math.*, 108(1-3):343–364, 1992. doi:10.1016/0012-365X(92)90687-B.

[17] R. Muther. *Systematic Layout Planning*. Cahners Books, 1973.

[18] J. Nederlof. Detecting and counting small patterns in planar graphs in subexponential parameterized time. In K. Makarychev, Y. Makarychev, M. Tulsiani, G. Kamath, and J. Chuzhoy, editors, *52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 1293–1306. ACM, 2020. doi:10.1145/3357713.3384261.

[19] S. Tanigawa. Sufficient conditions for the global rigidity of graphs. *J. Comb. Theory, Ser. B*, 113:123–140, 2015. doi:10.1016/j.jctb.2015.01.003.

[20] C. Umans and W. Lenhart. Hamiltonian cycles in solid grid graphs. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 496–505. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646138.