Visualization of Graph Products

Stefan Jänicke, Christian Heine, Marc Hellmuth, Peter F. Stadler, and Gerik Scheuermann

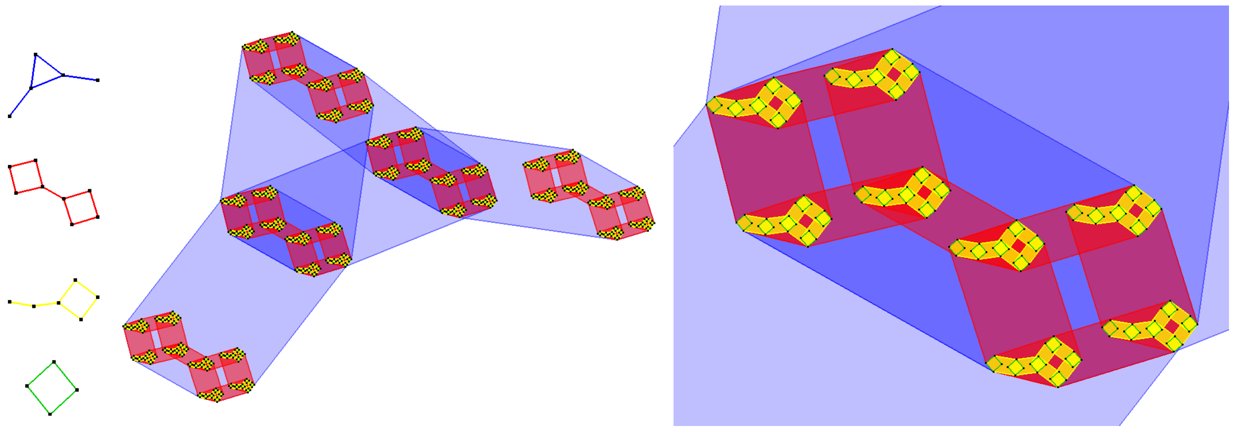


Fig. 1. Hierarchical graph product layout: the four factors (left), top-layer view (middle) and a zoom to the second layer (right).

Abstract—Graphs are a versatile structure and abstraction for binary relationships between objects. To gain insight into such relationships, their corresponding graph can be visualized. In the past, many classes of graphs have been defined, e.g. trees, planar graphs, directed acyclic graphs, and visualization algorithms were proposed for these classes. Although many graphs may only be classified as "general" graphs, they can contain substructures that belong to a certain class. Archambault proposed the TopoLayout framework: rather than draw any arbitrary graph using one method, split the graph into components that are homogeneous with respect to one graph class and then draw each component with an algorithm best suited for this class. Graph products constitute a class that arises frequently in graph theory, but for which no visualization algorithm has been proposed until now. In this paper, we present an algorithm for drawing graph products and the aesthetic criterion graph product's drawings are subject to. We show that the popular High-Dimensional Embedder approach applied to cartesian products already respects this aestetic criterion, but has disadvantages. We also present how our method is integrated as a new component into the TopoLayout framework. Our implementation is used for further research of graph products in a biological context.

Index Terms—Graph drawing, graph products, TopoLayout.

1 Introduction

To gain insight into binary relationships between objects, the relations are often coded into a graph, which is then visualized. The visualization is usually split in the layout and the drawing phase. The layout is a mapping of graph elements to points and curves in $\mathbb{R}^d$. The drawing assigns graphical shapes to the graph elements and draws them using the positions computed in the layout.

Many classes of graphs have been defined, e.g. trees, planar graphs, acyclic directed graphs, etc. However, most binary relationships that arise in nature do not fall in either of these classes and may only be represented as *general graphs*. While many layout algorithms exist that are tuned for special graph classes, there is no single sophisticated approach for general graphs, in the sense that if such a general method is applied to instances of a special graph class, the results are often inferior than using a specialized layout algorithm.

The structure of general graphs is often inhomogeneous. For many graphs it was found that there are parts that connect more with each other than with the rest of the graph. Detecting these clusters and successively collapsing them to single vertices results in a graph's hierarchical decomposition that can be visualized and explored at different levels of detail. A related idea is present in the TopoLayout [2] approach: Even if a graph does not belong to a special graph class, substructures may. Therefore, their detection and collapsing also leads to a hierarchy, usually with a different graph class at each node of the hierarchy. TopoLayout can be seen as a generalization of graph clustering, as it treats clusters as a regular graph class in this framework.

A graph class that has not yet been discussed for automatic visualization is the class of *graph products*. A graph product is the result of a multiplication operation defined for graphs. For an general overview we refer the interested reader to [23]. According to [23] there are six options for defining a product for graphs, where the multiplication is associative, commutative, and has a unit. These products all have a vertex set that is the Cartesian product of the factors' vertex sets and differ only in their edge sets. Fig. 2 shows an example of the two graph products we are mainly concerned with in this paper: the Cartesian and the strong product.

Many graphs have a product structure. Rectangular meshes are Cartesian products of paths. The Cartesian product of two cycles is a surface mesh of a torus. Each complete graph K_n is a strong prod-

- *Stefan Jänicke, Christian Heine, Gerik Scheuermann are with Image and Signal Processing Group, Institute for Computer Science, Universität Leipzig, Germany,*
E-mail: {stjaenicke,heine,scheuermann}@informatik.uni-leipzig.de.
- *Marc Hellmuth and Peter F. Stadler are with Max Planck Institute for Mathematics in the Sciences, Germany, Bioinformatics Group, Department of Computer Science, and Interdisciplinary Center for Bioinformatics, Universität of Leipzig, Germany,*
Peter F. Stadler is also with Fraunhofer Institut für Zelltherapie und Immunologie, Germany, Department of Theoretical Chemistry University of Vienna, Austria, Santa Fe Institute, 1399 Hyde Park Rd., Santa Fe, NM 87501, USA,
E-mail: {marc,studla}@bioinf.uni-leipzig.de.

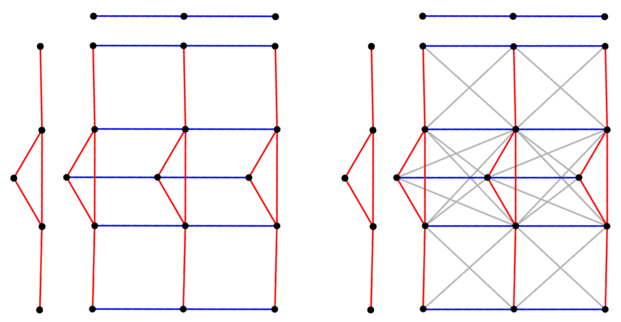


Fig. 2. Two factors and their Cartesian and strong product.
In the image each factor is associated with a color. The colored edges of the products are called Cartesian edges and the grey edges of the strong product are the cross edges. If only the colors of one factor are considered and all other edges deleted, the remaining graph consists of multiple connected components: its fibers.

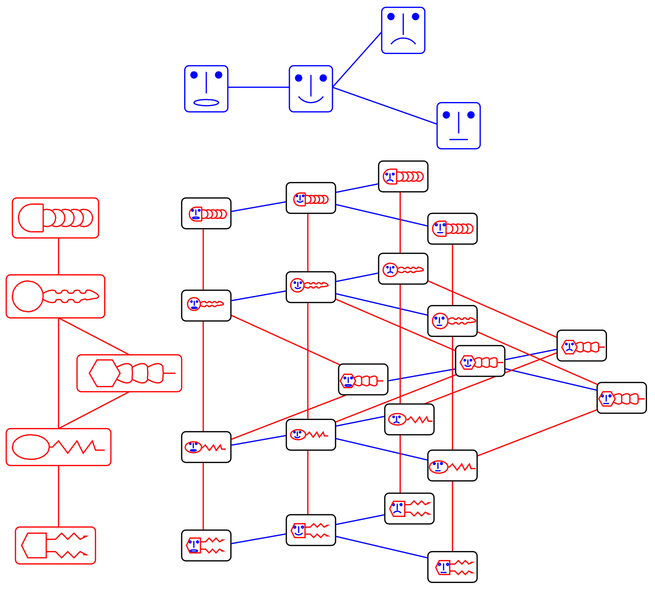


Fig. 3. Phenotype space graph of two characters.
The factors representing variational properties of phenotypes. The red factor indicates the "bodies" and the blue factor the "faces" of animals. Their multiplication forms a graph product: the phenotype space graph. It describes the possible evolution of phenotypes over short evolutionary time-scales. adapted from: [33].

uct of $K_{p_1},\dots,K_{p_k}$ where $p_1,\dots,p_k$ are n's prime factors. Hamming graphs are the Cartesian product of complete graphs. A well-known Hamming graph is the d-dimensional hypercube, that is the Cartesian product of d edges. Therefore, graph products can be seen as a generalization of many graphs with regular structure.

The visualization of graph products was motivated from a biological model proposed by Wagner and Stadler [34] that provided a concept concerning the topological theory of the relationships between genotypes and phenotypes. In this framework, a so-called "character" (trait) is identified with a factor of a generalized topological space that describes the variational properties of a phenotype. While these characters are usually not directly visible as an attribute of an organism, e.g. length of fingers, number of limbs, etc., the attributes are a combination of these traits. A graph can be constructed from the set of phenotypes and an "accessibility relation", that describes which phenotypes are interconvertible over short evolutionary time-scales. This evolution of phenotypes is reflected in the corresponding phenotype graph, that is itself (at least on a local level) a product graph (Fig. 3). The problem is thus to find the factors that represent the character's evolution from the phenotype graph, i.e. their product. Therefore, a visualization for product graphs is needed that can effectively communicate the quality of results by emphasizing the regularity of graph structure through regularity of layout.

Other areas where graph products play an important role can be found in computational engineering, e.g., for the formation of finite element models or construction of localized self-equilibrating systems in computational engineering, see [29, 27, 28]. Typical tasks in scientific computing, like solving discretized partial differential equations, need computational meshes. Hamming graphs can be used to organize peers in a P2P network [1, 32].

2 Related Work

For a general overview of graph layout for general graphs and special graph classes we refer the reader to [5, 3, 26, 22]. However, there does not exist an automatic layout algorithm tuned for graph products.

As one of our methods is a modification of the Fruchterman-Reingold algorithm [12], which is an algorithm for drawing general undirected graphs, we will give a brief account of it. Fruchterman and Reingold based their work on Eades [8], who viewed a graph's vertices as electrically charged particles that repel each other and edges as springs that attract particles. The layout of the graph is then found as an equilibrium state of that particle system by letting each particle move according to the forces acting on it and slowly cooling the system by restricting the maximum movement per iteration. Fruchterman and Reingold extended this work by using a different force formula for springs, removing the repelling forces between distant particles, and constraining the layout to an area, so that the area could be subdivided into cells and repelling forces are only needed to be computed inside the cells and their neighbors. This general approach has been modified often, mostly to improve convergence and run-time. Frick *et al.* [9] defined a local temperature for each vertex and gave heuristics to detect oscillations of subgraphs and cooling them faster. Hachul and Jünger [15] presented a multi-pole approach called FM^3 that approximates repelling forces to distant vertices rather than computing them exactly and were thereby able to speed up the layout generation significantly. Different multi-level approaches exist [13, 35, 18] that first build ever coarser versions of the input graph and then compute layout for the graphs from coarsest to finest in each step using the layout of the coarser graph as a template. They improve the convergence and run-time of the layout algorithm. Hachul and Jünger [16] gave an experimental evaluation for the different methods identifying FM^3 as the most versatile layout algorithm. More recent work focused on using the GPU to compute the layouts with different acceleration structures [10, 11].

A different notion of force-directed methods is the spring embedder of Kamada and Kawai [25] which uses springs of a certain stiffness and ideal length. The springs can act both attracting and repelling dependent on their current length. Rather than using repelling forces for vertices, each pair of vertices is connected by one spring with ideal length being the graph-theoretic distance between the vertex pair. The systems energy is coded in the *stress* which is then minimized. Gansner *et al.* [14] identified the similarity of the problem with classical multi-dimensional scaling and proposed to use stress majoriziation to minimize the stress.

Although the preservation of symmetry is generally attributed to force- and spring-based algorithms, this seems to be only true for local substructures like clusters. Substructures that span the whole graph, like fibers of graph products, are usually not represented similarly; they often are subject to continuous deformation from one border of the layout to the other. For this reason, neither the Fruchterman-Reingold method, Kamada-Kawai, nor their many variations can be applied directly to draw graph products.

Harel and Koren [19] presented a very fast and robust method called High-Dimensional Embedding (HDE) for drawing large graphs by carefully selecting landmark vertices of the graph that "span" the graph and then using a maximum variance projection of the graph distances to these landmarks.

A recent direction is drawing graphs with constraints [7]. However, constraints between groups of vertices only ensure that these groups do not overlap. Constraining groups of vertices to have the same layout is yet not possible in this framework.

The TopoLayout [2] algorithm builds a hierarchy of a graph by successive finding and splitting of substructures. The first phase splits the graph in its connected components. The second phase splits trees from each connected component. The third phase splits the graph into biconnected components and recurses in them. Each biconnected component is then checked whether it is suited to be drawn as an HDE component, whether is is a complete graph, or whether it consists of clusters (in that order). In the last case, the graph is split in its clusters and a graph for the inter-cluster connections. To these graphs the TopoLayout algorithm is applied recursively.

3 Graph Products

In this paper, we use definitions similar to [6] and [23]. A *graph* $G = (V,E)$ is an ordered pair of a *vertex set* V and an *edge set* E that consists of 2-element subsets of V. We assume that V is finite. For each edge $\{u,v\} \in E$, u and v are called *adjacent* to each other. A graph $G' = (V',E')$ is a *subgraph* of $G = (V,E)$, written as $G' \subseteq G$, if $V' \subseteq V$ and $E' \subseteq E$. A subgraph $G' = (V',E')$ is a *spanning subgraph* of $G = (V,E)$, if $V' = V$.

Given two *factors* $G_1 = (V_1,E_1)$ and $G_2 = (V_2,E_2)$, the *Cartesian product* $G_\square = (V,E_\square) = G_1 \square G_2$ and the *strong product* $G_\boxtimes = (V,E_\boxtimes) = G_1 \boxtimes G_2$ are defined as follows:

$$\begin{aligned} V &= V_1 \times V_2 = \{(v_1,v_2) | v_1 \in V_1, v_2 \in V_2\} \\ E_\square &= \{\{(v_1,u_2),(v_1,v_2)\} | v_1 \in V_1, \{u_2,v_2\} \in E_2\} \\ &\cup \{\{(u_1,v_2),(v_1,v_2)\} | \{u_1,v_1\} \in E_1, v_2 \in V_2\}. \\ E_\boxtimes &= E_\square \cup \{\{(u_1,u_2),(v_1,v_2)\} | \{u_1,v_1\} \in E_1, \{u_2,v_2\} \in E_2\}. \end{aligned}$$

It is imminent from the definition that the Cartesian product is a spanning subgraph of the strong product. The edges the strong product shares with the Cartesian product are called the *Cartesian edges*, and the other edges are called *cross edges*.

A graph G is *prime* with respect to one of these products if the only product that is isomorphic to it is the product of G with the graph $(\{v\},\emptyset)$ (the unit).

The definitions extend naturally to multiple factors. For instance, for the Cartesian product $G_\square = G_1 \square \ldots \square G_k$ we get:

$$V = \{(v_1,\ldots,v_k) | v_i \in V_i, 1 \le i \le k\}$$

$$E_\square = \{\{(v_1,\ldots,u_l,\ldots,v_k),(v_1,\ldots,v_l,\ldots,v_k)\} | v_i \in V_i, \{u_l,v_l\} \in E_l\}.$$

From this we can observe two things: each vertex of V is uniquely defined by a list of vertices from each factor, its *coordinates*, and each edge of $E_\square$ has exactly one *origin edge*, $\{u_l,v_l\}$, from one factor G_l. If we create a spanning subgraph of $G_\square$ that contains only the edges that originate from G_l for one particular l, the connected components of that subgraph are called the G_l-fibers; each one being isomorphic to G_l. For each vertex v_l of each factor G_l, we can define the set of *instances* of that vertex in the different fibers as follows:

$$I_{v_l} = \{(x_1,\ldots,v_l,\ldots,x_k) | x_i \in V_i, 1 \le i \le k, i \ne l\}.$$

4 Visualization of Graph Products

In previous work that treated graph products from a theoretical point, e.g. [23], illustrations were created manually, but one property was visible in almost all of them: all fibers of a factor G_l were drawn congruently. More precisely, if the layout of one fiber was fixed, the layouts of the others were simple translations of the first. This is necessary so that all fibers of all factors can be drawn congruently. This style is illustrated in Fig. 2. In order to stay consistent with established practice, we adopted this as an aesthetic criterion for drawing graph products.

Generally, computation of a graph layout is subject to aesthetic criteria, which describe the relationship between properties of the layout and its impact on human understanding. See [3] for an overview. Among these aesthetic criteria are, e.g. the minimization of edge crossings, which has been shown to have the greatest impact on human understanding [31], minimization of total edge length, equal distribution of vertices, and preservation symmetry. The last criterion is fulfilled, if isomorphic substructures of a graph have a similar layout. The aesthetic criterion for drawing graph products can be seen as a specialization of the symmetry criterion.

In the remainder of this section, we present algorithms for automated creation of a graph product's layout. They assume that the factors are given and construct the product from them. The methods are straight-line layouts and reduce the layout problem to a positioning of vertices. The method presented in Section 4.3 uses a modification of the Fruchterman-Reingold layout [12] to the layout of each factor so that the layout of the graph product looks nice. The method presented in Section 4.4 requires a user-given order of factors to construct a layered view which can be used for exploration of large graph products.

4.1 Congruent Layout

Before we present the layout algorithms, we show how, by defining each factor's straight-line layout, a congruent layout can be obtained by adding the positions of each product graph vertex's coordinates.

Given k factors, $G_1,\ldots,G_k$ and their resulting graph product with vertex set V. Let $p_i : V_i \to \mathbb{R}^d$ denote the position of each vertex of the factor $1 \le i \le k$ in an Euclidean space $\mathbb{R}^d$, the position of each vertex of V is then given by:

$$\begin{aligned} p : V &\to \mathbb{R}^d \\ (v_1,\ldots,v_k) &\mapsto \sum_{1 \le i \le k} p_i(v_i). \end{aligned}$$

Now we show that two fibers of the same factor are equivalent up to translation. Let u_l be a vertex of a factor G_l, and A and B some fibers of G_l. The two instances of u_l on these fibers are: $a_i \in V_i, 1 \le i \le k, i \ne l$ and $b_i \in V_i, 1 \le i \le k, i \ne l$. u_l's position with respect to A and B is given by:

$$p(a_1,\ldots,u_l,\ldots,a_k) = p_l(u_l) + \sum_{1 \le i \le k, i \ne l} p_i(a_i)$$

$$p(b_1,\ldots,u_l,\ldots,b_k) = p_l(u_l) + \sum_{1 \le i \le k, i \ne l} p_i(b_i).$$

The difference vector of u_l from the fiber A to B is therefore:

$$p(b_1,\ldots,u_l,\ldots,b_k) - p(a_1,\ldots,u_l,\ldots,a_k) = \sum_{1 \le i \le k, i \ne l} p_i(b_i) - p_i(a_i).$$

Let v_l be another vertex of factor G_l. If the difference vector that was used to move u_l from fiber A to fiber B is added to v_l in fiber A, it becomes imminent that this moves v_l to the same point as using p directly.

$$\begin{aligned} & p(a_1,\ldots,v_l,\ldots,a_k) + p(b_1,\ldots,u_l,\ldots,b_k) - p(a_1,\ldots,u_l,\ldots,a_k) \\ &= p_l(v_l) + \sum_{1 \le i \le k, i \ne l} p_i(a_i) + \sum_{1 \le i \le k, i \ne l} p_i(b_i) - p_i(a_i) \\ &= p_l(v_l) + \sum_{1 \le i \le k, i \ne l} p_i(b_i) = p(b_1,\ldots,v_l,\ldots,b_k) \end{aligned}$$

As all edges are drawn as straight lines, i.e. their points are defined by linear interpolation between their incident vertices positions, they are equivalent up to translation, too.

4.2 High-Dimensional Embedder (HDE)

The High-Dimensional Embedder of Harel and Koren [19] was already briefly mentioned in the related work, but in this section we show that it gives a congruent layout for Cartesian graph products. HDE selects m landmark vertices and then computes the graph-theoretic distances of all input graph vertices to these m landmarks. Conceptually, the resulting m-dimensional distance vectors of each vertex are treated as positions in an m-dimensional space. A covariance matrix is then computed from these positions and its two biggest eigenvalue/eigenvector pairs are determined. These eigenvectors describe the directions of maximum variance and the final layout is obtained by projecting the distance vectors into the plane spanned by these vectors.

For Cartesian graph products, the graph-theoretic distances in the product are strongly related to the distances in the factors. For any two vertices of a Cartesian graph product, the distance between them is simply the sum of distances inside each factor. For example, for any two factors G_1, G_2 and any two vertices (u_1, u_2) and (v_1, v_2) of the product $G_1 \Box G_2$, the distance between them is the same as the distance between u_1 and v_1 in G_1 plus the distance between u_2 and v_2 in G_2. Because of this relation, application of HDE to a Cartesian graph product will create positions in the m-dimensional space that are already a congruent layout. Linear operations on a congruent layout result in a congruent layout again. Therefore, after the projection phase of HDE the result is a congruent layout, independently of the landmarks and eigenvectors used. However, we have observed that HDE often gives unsatisfactory results, due to a tendency to collapse the layout of fibers to lines even if they are tree-like or contain cycles. We also observed that factors with a big diameter (maximum shortest distance between any two vertices in a graph) dominate the drawing and factors with low diameter are then only visible at great magnification. Therefore we turned our attention to an algorithm which does not favor certain factors.

4.3 Force-Directed Layout

Although it is trivial to construct the layout of a graph product from the layout of its factors, not all factor layouts give the same qualitative result for the product. The aim is therefore to find factor layouts that maximize the quality of the product. The layout is nice if the vertices are spread out uniformly across the drawing area. To avoid edge crossing, the edges should be kept short. As force-directed approaches achieve these goals for general graphs, we based our algorithm on one of them: the Fruchterman-Reingold algorithm [12]. Although algorithms exist, that perform better in terms of quality and run-time, they are not that easily extended for our constraint to keep fibers congruent.

Like in the Fruchterman-Reingold algorithm we proceed iteratively, moving vertices according to acting forces in each iteration. However, to ensure that the layout of fibers stays congruent in each iteration, vertices cannot be moved independently of each other. We found that the product's layout being a sum of the factors' layouts is not only a sufficient but also a necessary condition. Therefore, we do not move vertices of the product, but vertices of the factors. However, forces are still computed for the products vertices to ensure that it looks nice.

At each iteration we execute the following steps: first the current product layout is computed from the layouts of its factors. Then, forces $f(v_1, \ldots, v_k)$ on the product's vertices are calculated like in the original Fruchterman-Reingold algorithm. The algorithm then computes forces for all factor's vertices by averaging the forces of their instances:

$$F(v_l) = |I_{v_l}|^{-1} \sum_{v \in I_{v_l}} f(v).$$

Then the vertices of each factor are moved just like in the Fruchterman-Reingold algorithm, i.e, every vertex v_l of each factor G_l is shifted a bit in the direction of its corresponding force. We constrain each factor's layout to a certain area, which in turn constrains the layout of the product to an area, and therefore we can use the optimization of Fruchterman and Reingold to reduce the number of vertex pairs for which repelling forces have to be computed.

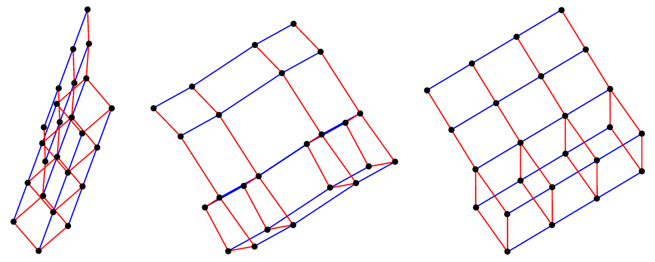

(a) no iterations (b) 15 iterations (c) 100 iterations

Fig. 4. Force-directed graph product layout example.
(a) after the initialization for the layout of $G_\Box$: Many vertices are drawn close together and some edges overlap. (b) intermediate result: the graph unfolds. (c) final layout showing good vertex distribution and no overlapping edges.

We perform a traditional Fruchterman-Reingold on each factor to obtain a starting configuration and finish the algorithm after a user-provided fixed number of iterations. In contrast to a full Fruchterman-Reingold layout on the product, where the number of iterations scale roughly linear with the total number of vertices, the number of iterations needed for our method scales linearly with the number of vertices in the factors. As a rule-of-thumb, we use 10 times this total as the number of iterations. Fig. 4 illustrates the algorithm for an example of a two factor Cartesian product.

The complexity of this algorithm is defined by the complexity of the underlying force directed method of Fruchterman and Reingold. For the graph product with n vertices and m edges we have to calculate $O(m)$ attracting and an average of $O(n)$ repelling forces in each iteration. The calculation of the product layout from the factor layout and the calculation of the factors' forces from the product's forces each take $O(k \cdot n)$ time, where k denotes the number of factors.

4.4 Hierarchical Layout

Applying the force-directed algorithm of the former section for graph products that are composed of thousands of vertices, we suspected that the readability of the final layout was lost. In this case, the large number of edges produce many edge crossings because of the uniform vertex distribution in the product's layout. Now we present a hierarchical algorithm that, for a given sequence of factors, presents the graph in a hierarchical style, fibers being meta-nodes and edges between the fibers being bundled. As this bundeling, that uses convex hulls, is only efficient in $\mathbb{R}^2$, we only apply it in this case. The algorithm conceptually composes the layout step by step starting with all vertices positioned at origin and then shifting the vertices with each new factor that is considered. We assume that each factor has been laid out independently with traditional Fruchterman-Reingold algorithm.

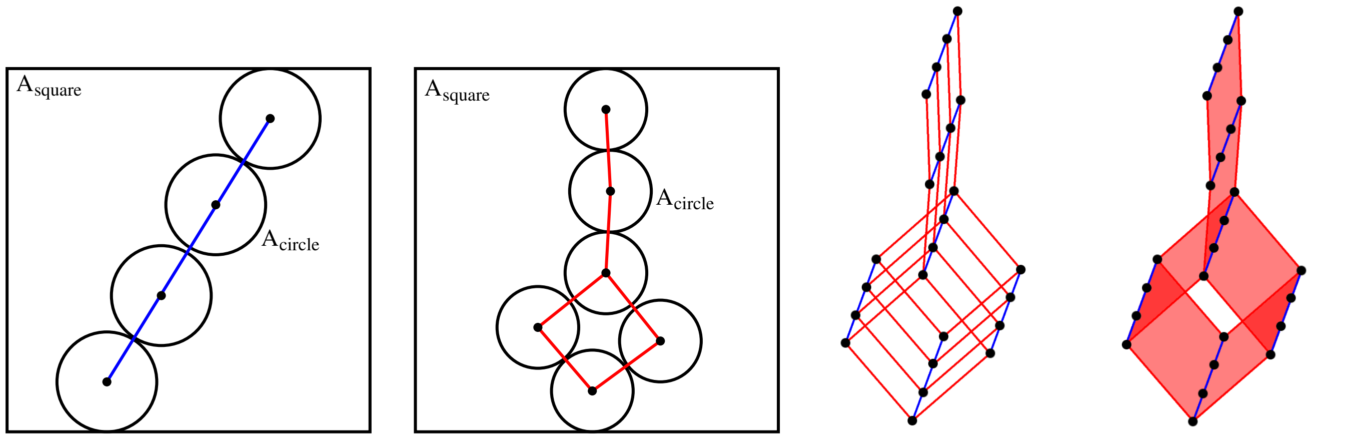
Initially, each vertex $v \in G_\Box$ is located at the origin of $\mathbb{R}^d$. The first layout of $G_\Box$ is defined by the layout of the first factor G_1 in the following way: every vertex with the same coordinate v_1 according to G_1 will be shifted to the same position. For all other factors G_i, each vertex $v \in G_\Box$ is moved subject to its i-th coordinate by:

$$p(v) \leftarrow p(v) + \alpha \cdot p(v_i)$$

We choose α in a way that it removes overlaps of meta-nodes:

$$\alpha = \frac{d_{max}(G_1 \Box \ldots \Box G_{i-1})}{d_{min}(G_i)}$$

d_{min} is the minimum Euclidean distance between two vertices in the layout of the factor graph G_i and d_{max} defines the maximum Euclidean distance between two vertices in the previously calculated layout L_{i-1} for $G_1 \Box \ldots \Box G_{i-1}$. The multiplication with α ensures, that the edges of the factor graph G_i will be long enough, that the components of L_{i-1}, that will be placed on every meta node of G_i, do not overlap



(a) coverage ratios of the factors of $G_\square$

(b) hierarchical layout of $G_\square$

(c) hierarchical layout of $G_\square$ with convex hulls

Fig. 5. Example hierarchical graph product layout.
(a) The red factor produces a higher coverage ratio than the blue factor and thus precedes it in the initial order of factors for the hierarchical layout (b). Using convex hulls (c) the dominance of the red factor is emphasized.

each other. Only after processing the last factor, all vertices of $G_\square$ have distinct positions. At this point, we get a hierarchical drawing for $G_\square$, where the highest hierarchical level corresponds to the last factor G_k and its meta nodes are representations for the layout of $G_1\square\ldots\square G_{k-1}$.

The order of factors have a huge impact on the final layout of the hierarchical graph product. Although this order can be specified by the user, we also determine a "best" order of factors by finding a configuration that uses a maximum of screen space and thus optimizes the overall resolution. To create the ordering, we compute a coverage ratio r_i for each factor G_i. Let d_{min} denote the minimum distance between two vertices in the layout of factor G_i, we replace each vertex of G_i with a hyper-sphere, that has a diameter of d_{min}. We contrast the sum of the hyper-spheres' volumes with an axis-parallel hypercube whose side length equals the maximum vertex distance in any of the d dimensions plus d_{min}. The coverage ratio r_i is determined as the ratio of the hyper-sphere volumes to hypercube volume. Finally, the factors $G_1,\ldots,G_k$ are sorted from smaller to larger coverage ratios. The coverage ratio for two factors is illustrated in Fig. 5, which also shows the hierarchical layout.

Because the hierarchical method produces long edges, this usually leads to more edge crossings. Fortunately, the edges have very regular layout. As all edges that originate from the same factor edge are equivalent up to translation, their instances can be bundled and represented as one graphical shape. This shape is the convex hull of all instances of the vertices the origin edge connects. As many of these convex hulls repeat, especially in lower levels, rather than computing them each time anew, we only need to compute one for each layer of the hierarchy, and translate it for each node of that hierarchy level.

The complexity is one of the great advantages of this algorithm. Let n be the number of vertices in $G_\square$. At first, the coverage ratios must be calculated for each factor. Therefore, getting the exact minimum distance between two vertices and the maximum distance between two hyper-sphere borders will both take $O(n_i^2)$, whereas n_i is the amount of vertices in G_i. Since $n_i \ll n$, the complexity of this phase is bounded by $O(k \cdot n)$.

For the second part, we have to update the graph product layout in every iteration. This takes $O(n)$ time. Additionally, we must calculate the maximum Euclidean distance d_{max} between two vertices in the temporary layout of $G_\square$. The exact distance can be computed in $O(n^2)$ by comparing each vertex pair. Since this scales badly for large values of n, we use the following upper bound instead: the maximum distance of all vertices to their barycenter. It produces useful spaces between the components of a layer, too, and de-stresses the final layout. The complexity for this step is linear and for the whole phase we get $O(k \cdot n)$, since we have to iterate over k factors.

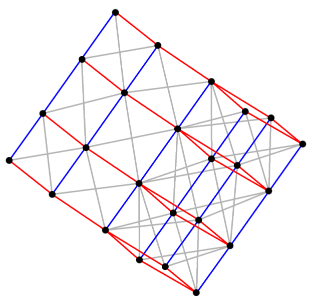
Calculating the convex hulls for $k-1$ hierarchy layers needs additional time. Except the highest layer, every convex hull occurs several times on each of the other $k-2$ layers. We only have to compute one of these duplicate convex hulls, which is a representation for an edge of the factor, a layer corresponds to. For other fibers we shift the convex hull by the difference of the barycenters of the involved vertices. So, we need to find m_c convex hulls, where m_c is the total number of edges of the factors $G_2,\ldots,G_k$. On the highest layer, we find the largest complexity, since the point-set that is used for convex hull calculation has its maximum there. Let n_k be the number of vertices in G_k. Then, the point-set has a size of $n_c = 2\frac{n}{n_k}$. With this, we can conclude, that the computation of all convex hulls is bounded by $O(m_c \cdot n_c \log n_c)$.

5 Visualization of Strong and Approximate Graph Products

Our force-directed algorithm (Section 4.3) extends naturally to strong and so-called approximate graph products, as long as the factors and the vertices' coordinates are known.

It is quite simple to find the drawing for strong graph products. Fortunately, there is no need to change any part of the layout algorithms. In the algorithmic description we avoided any assumptions on the particular edge set, so the algorithm is directly applicable to any type of graph product. The forces alter slightly because of the additional cross edges of the strong graph product. The force-directed layout for the strong product of two factors can be seen in Fig. 6(a). In comparison to the Cartesian product, the red factor's structure is narrower here and the cross edges, which are colored grey, cause additional edge crossings. We still obtain a similar layout for the strong product due to symmetry.

An approximate product is a "perturbed" product graph, where some edges or vertices are added or removed, see [20, 21]. Visualizing an approximate graph product needs additional work. On the one hand, there are vertices with their incident edges which are missing from the product, and on the other hand there are additional vertices which are connected to the graph product, but do not have any coordinates. To solve these problems, we calculate the layout for the corresponding graph product G which can be constructed from the given factors. Then, we map the resulting positions for the vertices of G to the existing vertices in the approximate product $\tilde{G}$ using vertices' coordinates. If a vertex has no coordinates it is ignored for now. When the product part has been laid out, the vertices' positions are fixed and a regular Fruchterman-Reingold is run on the additional vertices. Finally, we receive a complete layout for $\tilde{G}$. Fig. 6(b) shows an approximate Cartesian graph product of two factor graphs. In comparison to the unperturbed version (4(c)), three vertices are missing and four vertices without coordinates were inserted. These vertices are easy to figure out, since they only have grey colored outgoing edges.


(a) Strong Graph Product

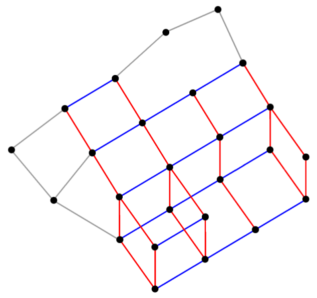

(b) Approximate Cartesian Graph Product

Fig. 6. Layouts for a strong and an approximate graph product with 2 factors.

6 TopoLayout Integration

The class of graph products can easily be integrated into the TopoLayout framework. In contrast to clusters, which are local features, factors span the graph or subgraph like a mesh, thereby pose an interesting complement. However, as the algorithm for drawing products requires each vertex's coordinates, the analysis phase of TopoLayout must be extended by an algorithm that recognizes and splits graph products into their factors. We use the algorithm by Imrich and Peterin [24] to detect Cartesian products. It runs in linear time and requires linear space. For determining the strong product we use the algorithm provided by Hammack and Imrich [17] that runs in linear time for graphs that have bounded degree.

We now discuss where in the TopoLayout pipeline the detection of graph products is best situated. As every tree is prime, there is no need to check for products before trees. Graph products are biconnected, therefore it makes sense to check biconnected components. If a graph product is encountered by the original TopoLayout, it is usually detected as a HDE component, so checking after HDE would be too late. We therefore put it after biconnected component and before HDE. For each product that was detected, the factors are fed back to TopoLayout, so that they can be further split into trees and biconnected components.

7 Results

To illustrate the benefits of using one of our presented algorithms to draw graph products, we compared the results to two widely used algorithms, namely HDE [19], which also generates congruent layouts, and Fast Multipole Multilevel Method (FM^3) [15]. We compared the quality of the generated layouts and the running time of the considered algorithms. The system used in all experiments was a 3.07 GHz Intel Core i7-950 CPU with 12 GB RAM running Linux.

We have tested the algorithm on a multitude of self-generated graph products and graph products that we found in the University of Florida Sparse Matrix Collection [4]. From these, we have selected three graph products to show the differences between the algorithms mentioned. The properties of these graphs are listed in Table 1. G_1 is the graph *G12* of the matrix group *Gset* of the University of Florida Collection. It is only one representative of numerous mesh-like graph products we found in this graph collection. Some other examples are *qc324* and *cdde1-cdde6* of the *Bai* matrix group, *G11 and G13* of the *GSet* group and *nos7* of the *HB* group.

Table 1. Characteristics of the graph products from Fig. 7

graph	G_1	G_2	G_3
product size vertices/edges	800/1600	180/510	1440/7680
factors	2	3	5
factor sizes vertices/edges	16/16 50/50	3/3, 6/5, 10/10	3/3, 4/4, 5/5, 4/6, 6/5
HDE	$0.24s$	$0.06s$	$0.72s$
FM^3	$0.49s$	$0.1s$	$1.12s$
force-directed	$3.17s$	$0.37s$	$14.07s$
hierarchical	$0.06s$	$0.01s$	$0.14s$

Fig. 7 shows different 2-dimensional layouts for the three graph products. Note that neither HDE nor FM^3 recognize graph products and are therefore unable to color edges based on the origin factor. Despite this, we choose to use these colors for their layouts and thereby improve their readability, because we think that our algorithms still perfom better with respect to quality. The obtained running times for calculating the different drawings are provided in Table 1. First of all, we recognize that the hierarchical algorithm is the fastest method, which is due to the near-linear running time. HDE and FM^3 also generate their layouts in a relatively short time. Although we use the grid variant of the Fruchterman-Reingold algorithm, the force directed approach has a very high computation time compared to the other algorithms. This is due to the numerous iterations and a high computational effort in calculating repelling forces.

However, the graphical output points out the advantages of using the force directed approach to visualize graph products. Particulary for smaller graphs we get highly symmetrical layouts with very clearly displayed adjacencies of vertices. The reason is that different fibers of each factor stay congruent to each other in every iteration. Furthermore, the underlying force model leads to a good distribution of the vertices. So every fiber of each factor is pointed out clearly. Thus, it is easy to locate vertices with a specific coordinate vector. In terms of graphical output, HDE and FM^3 unveil their weaknesses in visualizing graph products. The distance-based calculation of HDE leads to good representations of fibers of factors with large diameters at the expense of factors with small diameters. The resulting piles of vertices can be seen in most HDE-layouts. Another problem is the selection of landmarks in HDE. We often observed collapse of factors' fibers to a line, depending on the (random) selection of the first landmark. Finally, we conclude that HDE may provide the important congruent layouts fast, but its graphical quality is worse than in the force directed approach. FM^3 shows better vertex distributions as HDE, which is due to its force directed model. Nevertheless, the multi-level strategy leads to a globally stable graph structure, but locally we receive deformations of fibers in many parts of the graph product layout. This creates many additional edge crossings, and, thus, deteriorates the readability of the layouts significantly. However, the readability of the layouts is lost in products with more than a thousand vertices and a high amount of edge crossings and overlaps, regardless of using HDE, FM^3, and our own force directed algorithm. Then, the hierarchical approach is particularly well suited for visualizing. It also guarantees the congruency of the different fibers and it always creates aesthetic layouts because of the hierarchical structure. Even for products of several factors with more than 100,000 vertices we obtain attractive layouts in a few seconds.

The main advantage of the hierarchical layout for drawing product graphs is the ability to see and to understand how the factors look like. As one can see in Fig. 7(h), resp. in Fig. 7(l), it is much easier to understand what the factors are, than in Fig. 7(g), resp. in Fig. 7(k), where the same graphs are represented. Moreover, as shown in Fig. 1 and 8 it is possible to zoom into the product graph. In this way we can visualize the local structure, i.e., the contained subproducts of the given product graph. In particular, this method is an advantage if

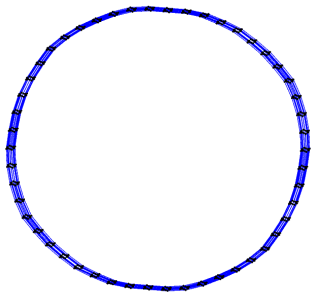
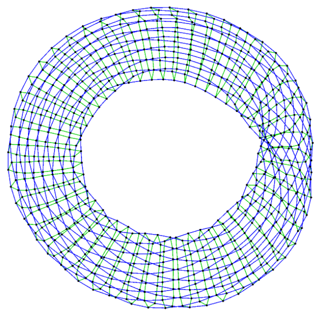
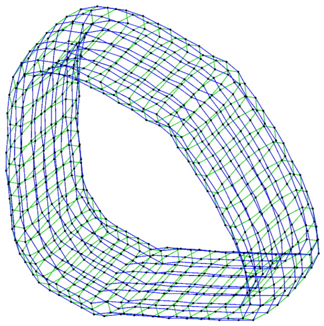
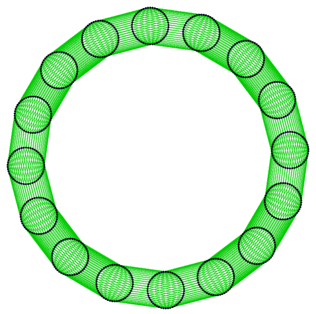
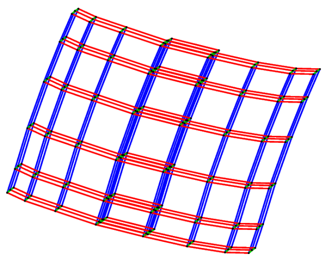
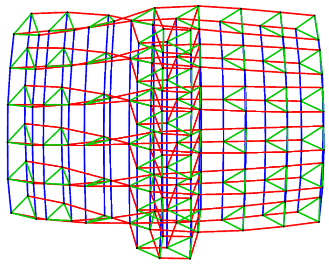
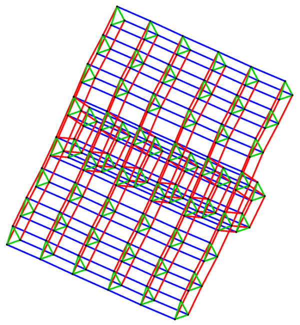
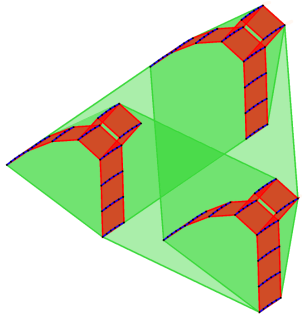
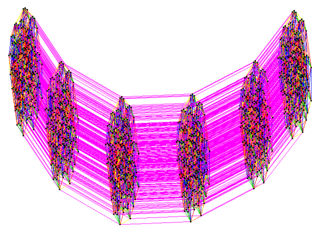
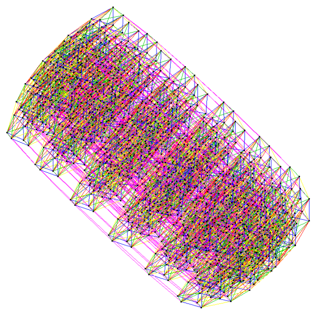
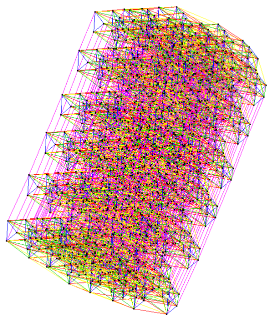
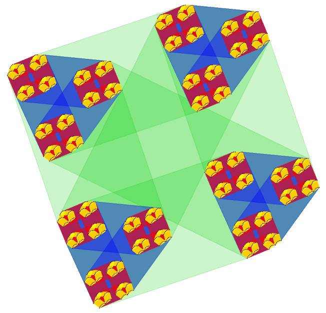

(a)

(b)

(c)

(d)

(e)

(f)

(g)

(h)

(i)

(j)

(k)

(l)

Fig. 7. Layouts for Cartesian graph products G_1 with 2 factors and $|V| = 800$, G_2 with 3 factors and $|V| = 180$, and G_3 with 5 factors and $|V| = 1440$
(a),(e),(i) HDE; (b),(f),(j) FM^3; (c),(g),(k) Force Directed; (d) Hierarchical,(h),(l) Hierarchical with convex hulls

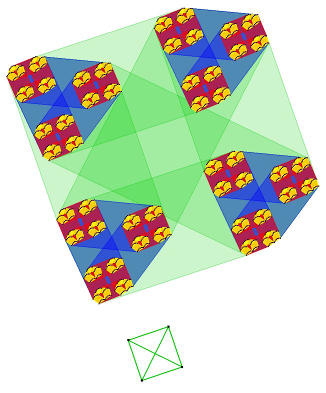
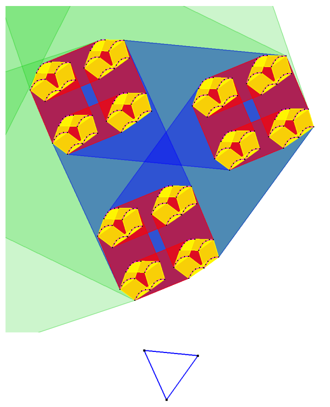
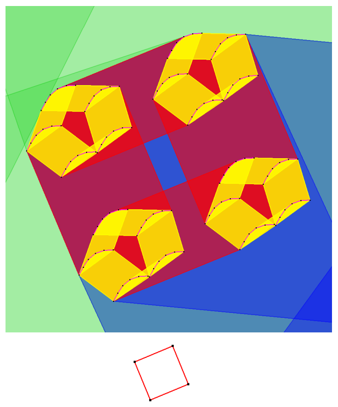
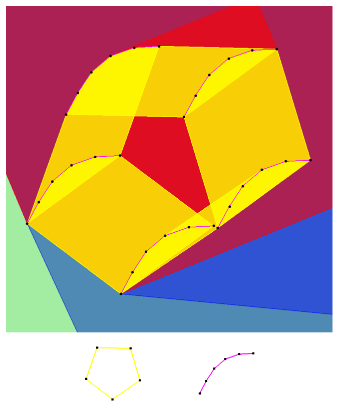

(a)

(b)

(c)

(d)

Fig. 8. Zoom into different layers for the hierarchical layout of 7(l)
(a) top-layer 1, (b) layer 2 of the bottom-right component of layer 1, (c) layer 3 of the top-left component of layer 2, (d) layer 4 of the bottom-left of layer 3

one wants to visualize product graphs with many factors. Zooming into those graphs makes clear how the different factors look like and helps to understand the product structure on the several local levels that exist.

We furthermore wanted to evaluate, whether the class of graph products is occuring often in real-world data. In the University of Florida Sparse Matrix Collection [4] which comprises 2101 graphs, we classified 55 graphs as a graph product or to contain a graph product according to TopoLayout. We classified graphs or graph components as graph products only if their size exceeded 20 vertices, in order to avoid counting the numerous $K_2\Box K_2$ and K_4 that were found. The products we found consisted of two or three factors and factors were usually simple paths. The sizes ranged from 900 to 3200 vertices and 1250 to 7840 edges. For TopoLayout, the *HB/watt1* was particularly interesting as it detected a 27x8x8 lattice augmented by many small trees. These numbers do not indicate that graph products occur frequently in nature, however, they do indicate that graph products are of interest in other research contexts.

8 Conclusion and Future Work

We have presented two algorithms for drawing product graphs. Both are fast and produce nice looking pictures. In the force-directed method each factor has the same influence on the layout, but it is only useble if the number of factors is small. By keeping edges short, it avoids edge crossings. For many factors, the edge crossings become unavoidable and it is beneficial to make some edges long and bundle them. This approach is present in the hierarchical layout of product graphs which also captures on the notion of the factors defining a hierarchy inside the product and allows an explorative analysis of large graph products. The methods however are still limited to rather few factors (usually less then 10) because the product grows enormously in size with each additional factor.

Both methods were integrated into the TopoLayout framework and can be applied to the six known graph products as well as perturbations of them, as long as all factors and the coordinates of each vertex are known.

In future work, we want to extend the recognition of graph products inside TopoLayout to more types of products and at one time also to perturbed graph products. Existing approaches only detect factors, but not vertices' coordinates.

Graph products have a general disadvantage: the product operation lets the number of edges grow faster than the number of vertices. Considering that this has a very bad influence on the number of crossings, it seems more natural to show the factors instead of the product. In our current visualization tool we always show both. Once products have been established in the visualization community, it may be possible to never show the product but only the factors and symbol indicating the used product operation.

Acknowledgments

The authors would like to thank Waldemar Beser for extending their TopoLayout implementation for the recognition of strong products.

References

[1] S. B. Akers and B. Krishnamurthy. A group-theoretic model for symmetric interconnection networks. *IEEE Trans. Comput.*, 38(4):555–566, 1989.

[2] D. Archambault, T. Munzner, and D. Auber. Topolayout: Multilevel graph layout by topological features. *IEEE Trans. Vis. Comput. Graph.*, 13(2):305–317, 2007.

[3] G. D. Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing*. Prentice Hall, 1998.

[4] T. A. Davis. The University of Florida sparse matrix collection. Submitted to ACM Transactions on Mathematical Software.

[5] G. di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994.

[6] R. Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2005.

[7] T. Dwyer. Scalable, versatile and simple constrained graph layout. *Comput. Graph. Forum*, 28(3):991–998, 2009.

[8] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[9] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing*, volume 894 of *LNCS*, pages 388–403. Springer, 1994.

[10] Y. Frishman and A. Tal. Multi-level graph layout on the GPU. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1310–1319, 2007.

[11] Y. Frishman and A. Tal. Online dynamic graph drawing. *IEEE Trans. Vis. Comput. Graph.*, 14(4):727–740, 2008.

[12] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[13] P. Gajer and S. G. Kobourov. Grip: Graph drawing with intelligent placement. *J. Graph Algorithms Appl.*, 6(3):203–224, 2002.

[14] E. R. Gansner, Y. Koren, and S. C. North. Graph drawing by stress majorization. In Pach [30], pages 239–250.

[15] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In Pach [30], pages 285–295.

[16] S. Hachul and M. Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In P. Healy and N. S. Nikolov, editors, *Graph Drawing*, volume 3843 of *LNCS*, pages 235–250. Springer, 2005.

[17] R. Hammack and W. Imrich. On Cartesian skeletons of graphs. *Ars Mathematica Contemporanea*, 2(2):191–205, 2009.

[18] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. *J. Graph Algorithms Appl.*, 6(3):179–202, 2002.

[19] D. Harel and Y. Koren. Graph drawing by high-dimensional embedding. *J. Graph Algorithms Appl.*, 8(2):195–214, 2004.

[20] M. Hellmuth, W. Imrich, W. Klöckl, and P. F. Stadler. Approximate graph products. *European Journal of Combinatorics*, 30:1119 – 1133, 2009.

[21] M. Hellmuth, W. Imrich, W. Klöckl, and P. F. Stadler. Local algorithms for the prime factorization of strong product graphs. *Mathematics in Computer Science*, 2(4):653 – 682, 2009.

[22] I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 06(1):24–43, 2000.

[23] W. Imrich and S. Klavzar. *Product Graphs*. Wiley-Interscience, New-York, 2000.

[24] W. Imrich and I. Peterin. Recognizing Cartesian products in linear time. *Discrete Math.*, 307:472–482, 2007.

[25] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, 1989.

[26] M. Kaufmann and D. Wagner, editors. *Drawing graphs: methods and models*. Springer-Verlag, London, UK, 2001.

[27] A. Kaveh and K. Koohestani. Graph products for configuration processing of space structures. *Comput. Struct.*, 86(11-12):1219–1231, 2008.

[28] A. Kaveh and R. Mirzaie. Minimal cycle basis of graph products for the force method of frame analysis. *Communications in Numerical Methods in Engineering*, 24(8):653–669, 2008.

[29] A. Kaveh and H. Rahami. An efficient method for decomposition of regular structures using graph products. *Intern. J. for Numer. Methods in Engineering*, 61(11):1797–1808, 2004.

[30] J. Pach, editor. *Graph Drawing, 12th International Symposium, GD 2004, New York, NY, USA, September 29 - October 2, 2004, Revised Selected Papers*, volume 3383 of *LNCS*. Springer, 2004.

[31] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In G. D. Battista, editor, *Graph Drawing*, volume 1353 of *LNCS*, pages 248–261. Springer, 1997.

[32] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. Hypercup - hypercubes, ontologies and efficient search on p2p networks. In *LNCS*, pages 112–124. Springer, 2002.

[33] B. M. R. Stadler and P. F. Stadler. The topology of evolutionary biology. In *In Ciobanu, editor, Modeling in Molecular Biology, Natural Computing Series*, pages 267–286. Springer Verlag, 2004.

[34] G. P. Wagner and P. F. Stadler. Quasi-independence, homology and the unity of type: A topological theory of characters. *J. Theor. Biol*, 220:505–527, 2003.

[35] C. Walshaw. A multilevel algorithm for force-directed graph-drawing. *J. Graph Algorithms Appl.*, 7(3):253–285, 2003.