A Modeling and Encoding Method for Relative Layout Generation and Optimization in Manufacturing

Marc Pouly
Jana Koehler
Lucerne University of Applied Sciences and Arts
Department of Engineering and Architecture, 6048 Horw, Switzerland

June 1, 2016

Abstract

Generating optimal layouts or placements of technical components is a computationally intensive task. The induced optimization problems encounter very large search spaces, which are to be explored in a systematic or randomized fashion. Whereas many publications discuss modeling aspects and solution algorithms, little gets published on the actual encoding scheme used to represent different layouts, despite of the influence of efficient encoding schemes on the overall project success.

In this article, we present a very compact encoding scheme and efficient encoding algorithms that describe layouts and placements by the relative positioning of components to each other. Horizontal and vertical arrangements as well as rotations of components and sub-components can be modeled and encoded as compact integer vectors. The manipulation of these vectors reduces to counting with integers, which provides a very efficient foundation for any state-based search algorithm and which can be easily tailored and configured to the needs of a specific application.

Our encoding fits layout requirements where the relative arrangement of components takes center stage and specific physical layouts can be computed in a post-processing step. In particular, our techniques are applicable to manufacturing problems where processing orders of the components are predefined by the manufacturing processes.

1 Introduction

Many industrial applications require to generate layouts of technical components in order to place them in a specific arrangement for further processing. To solve such a placement or layout problem, one usually proceeds in four phases:

1. Elicitation of problem requirements

2. Modeling of the problem
3. Encoding of the model
4. Application of a solution algorithm to solve the encoded model and generate one (or several) solution(s)

The research literature mainly deals with the first and last phase in the solution process, often omitting details of the model and almost never discussing the specific encoding that is fed into the solution algorithm. In particular, layout and placement problems are often described with the focus on the solution algorithm. However, in practice, compact modeling and encoding techniques significantly contribute to the solvability of a given problem class and therefore to the success of the overall project. Furthermore, generating layout or placement solutions that meet all requirements is not enough, but optimal solutions that minimize or maximize a given objective function are usually required. Such constrained optimization problems in general encounter very large search spaces, which are to be explored in a systematic or random fashion. The size of the encoding of a single state, the ease by which the state transition function can be described, and the means to control the search all influence the scalability and applicability of solution algorithms.

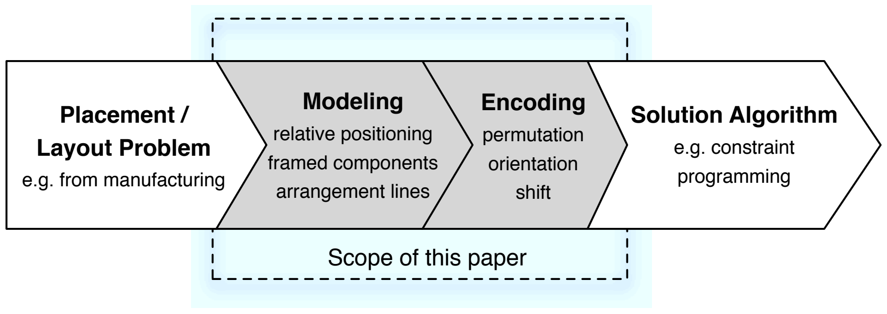


Figure 1: Contribution of the article within the generic process of solving a placement or layout problem.

In this article, we focus on Steps 2 and 3 of the solution process as summarized in Figure 1. We discuss a model and encoding that we believe is of interest to many layout and placement problems. Our work was originally motivated by a problem of placing technical components in a manufacturing environment with a predefined processing order caused by the design of the manufacturing tools. We will discuss this class of problems in more detail in Section 3. However, our method can also be beneficial to other types of layout problems where the *relative* positioning of layout components can be investigated independently, and their concrete physical positioning can be derived from the relative positioning in a post-processing step.[1] In order to make the class

[1]Our method was originally developed for 2-dimensional spaces and vertical layouts only. However, mixed vertical and horizontal layouts can also be generated if the layout problem is separable as we discuss in Section 8.

of problems more clear and to introduce the key ideas of our modeling and encoding methods, let us look at the example shown in Figure 2.

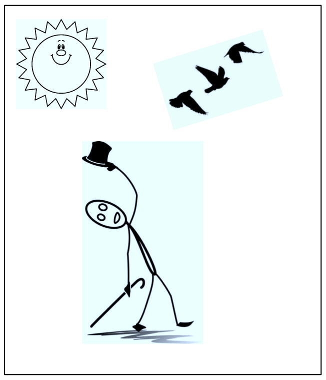


Figure 2: A layout scene with three components.

Figure 2 shows a scene of 3 components, a man with a hat, a group of birds, and a smiling sun, for which we want to compute different possible layouts. Our method is applicable if the components can be characterized by rectangular frames as shown in Figure 3. Furthermore, each component is assigned a vertical reference line. This reference line can for example be placed on the center point of the frame or any other point within the framed area, i.e., its exact positioning can be arbitrary. Each component has a unique identifier. In our method, we simply enumerate the components with integers $0,1,2$. The relative horizontal placement of the components with respect to each other is described by a *permutation* vector of these three integers. The permutation vector describes the order in which we encounter the reference lines when performing a left-to-write sweep over the 2-dimensional plane.

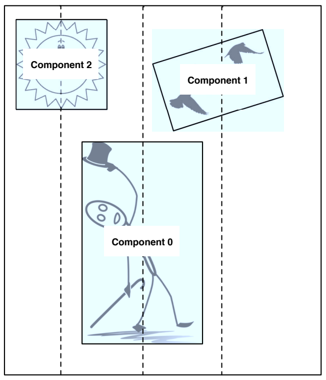


Figure 3: Object frames and reference lines encoded by the permutation $(2, 0, 1)$.

Components also possess an *orientation* specified in degrees and can rotate around their reference points. In Figure 4, the components 2 and 0 (the man and the sun) are rotated by 0 degrees, whereas component 1 (the group of birds) is rotated by 25 degrees. We will show later in this article how component orientations can be very compactly encoded by integers, which allows us to efficiently enumerate possible rotations.

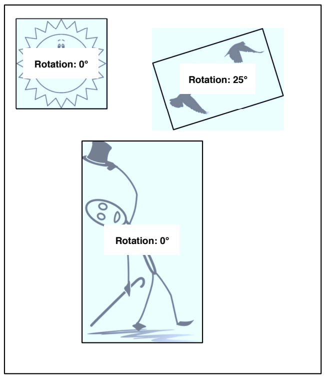


Figure 4: Orientation of layout components.

Of course, a simple vertical placement and orientation of components is not sufficient for many applications, which usually require that components are also horizontally aligned with each other. Such a horizontal alignment should not be limited to the boundaries of the component frames, but should allow designers to specify arbitrary alignment points. In Figure 5 we can see that one alignment point has been added to the mouth of the sun and another one has been added to the upper wing of the middle bird. Horizontal *arrangement lines* are drawn through these points. Moreover, the designer imposed a layout constraint on the two components by mutually aligning the two arrangement lines. We speak of a relative alignment between the two components because their exact (absolute) positioning on the plane does not matter, i.e., if one component is shifted vertically, the position of the other component must be adapted.

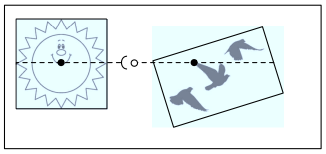
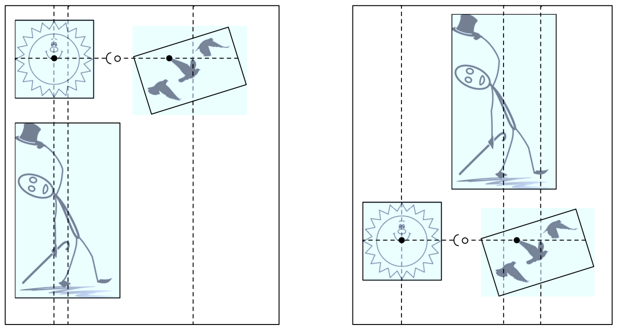


Figure 5: Relative horizontal alignment of components through arrangement lines.

The alignment pair $(1, 2)$ specifies that the two components 1 and 2 are to be aligned with respect to each other and ensures that if one component is shifted by say 10 units, the very same shift must be applied to the other component as well. An arbitrary number of such arrangement lines can be added to components, thus allowing for an arbitrary number of horizontal layout constraints represented by alignment pairs. For this purpose, our method introduces the modeling concept of an alignment graph between components and provides an efficient encoding by Prüfer codes.

With the permutation $(2,0,1)$, the orientation $(0,25,0)$ and the alignment pair $(1,2)$ our relative layout in this introductory example is fully specified. It is clear that many physical layouts can be generated from this relative layout or, in other words, many different physical layouts realize this relative layout. Any physical layout that preserves permutation, orientation and alignment is considered a valid instantiation. Two such possibilities are shown in Figure 6.



(a) Moving the man to the left

(b) Moving the man to the right and to the top

Figure 6: Layouts can be freely generated as long as permutation, orientation, and alignment constraints are respected.

In the left sub-figure of Figure 6, the man is moved to the left, but its reference line must remain right of the reference line of the sun to preserve the specified permutation of components. The sun and group of birds are correctly aligned. Their alignment is also preserved in the right subfigure, where the man is moved to the top and to the right, but its reference line must remain left of the bird group's line. Hence, both physical layouts are valid instantiations of our relativ layout. In summary, our modeling approach relies on the following concepts:

- Objects are represented by rectangular frames.
- Their relative positioning is characterized by the concepts of *permutation*, *orientation*, and *alignment*.

It may seem counter-intuitive to the reader that the two physical realizations in Figure 6 are considered equivalent, given the rather different positioning of the man. In fact, we will later introduce the additional concept of a processing order that, when present, will separate these two realizations in different equivalent classes of physical layouts.

For each modeling concept (including the here missing processing order), we develop very compact encodings based on integers. This means, when enumerating different layouts, we can simply count with integers, which provides a very efficient foundation for any solution algorithm and makes it very easy to enumerate the search space of all placements. On the one hand side, this can be used to generate and test placements in combination with standard search procedures and various optimization criteria. On the other hand, more sophisticated optimization methods can be obtained by combining our modeling method and encoding scheme with e.g., constraint programming over finite domains or other well-established optimization techniques.

The class of placement and layout problems to which our method is applicable, satisfies the following assumptions:

- The problem consists in placing components on a plane having a global coordinate system with a reference point.
- Components (and distinguished sub-components inside components) can be framed and abstracted by rectangles with a reference point and given width and height.
- Arrangement lines that are added to components are sufficient to express constraints of horizontal alignment for components.
- Components must not satisfy specific requirements of overlap with or distance from each other. We usually do assume that components must not overlap in manufacturing, but when moving (rotated) components within the specification of a relative layout, it is possible that some physical layout instances contain overlapping components. It is up to the application to control how specific physical layout instances are obtained from our generated relative layout during a post-processing phase.
- Relative layouts represent equivalence classes of physical realizations that are defined by the horizontal positioning, orientation, and mutual alignment of components only. We will later bring in the concept of processing sequences as additional element to distinguish the relative vertical positioning of components.

We would like to emphasize once more that only the relative placement of components with respect to each other matters in our method and that two physical layouts are considered different only if their relative layout (permutation, orientation, and alignment) differ (under the pre-defined processing order). Our method thus abstracts from the absolute or geometric placement of components on the plane. In fact, one relative placement enumerated by our method represents many specific geometric layouts. We do not consider the generation of such specific geometric layouts of a given placement in this article. It can be added as a post-processing step and is generally highly application-specific.

The article is organized as follows: We proceed with a review of related work in Section 2 to further detail out the contribution and positioning of our method in the light of other approaches. Then we consider in more detail placement problems under processing sequences and discuss examples from industrial manufacturing in Section 3. This section formally characterizes the class of placement problems that we consider and discusses the notion of equivalence of placements induced by the predefined processing order. Readers interested in finding out if our method works for their layout problems, can skip this section and directly proceed to Section 4, which introduces the key elements of an integer-based encoding for (equivalence classes of) component placements and gives an overview on the algorithms used to generate placements. Section 5 presents the details of the encoding algorithms and illustrates them with examples. In Section 6 we explain how physical layouts of components can be obtained from the encoded placement representation. In Section 7, we discuss how the proposed encoding scheme can be embedded into an exhaustive or local search approach to directly generate component placements that satisfy instance-specific constraints and that are optimal with respect to a given objective function. We also briefly sketch how to use our encoding in combination with other optimization techniques such as constraint programming. In Section 8, we discuss how our method can be applied to generate mixed horizontal and vertical placements. We conclude with an outlook on current work in Section 9.

2 Related Work

We consider a placement problem in the context of industrial manufacturing where the intended processing order of the technical components influences the placements that we need to generate. Under a predefined processing order, certain placements fall into the same equivalence class and therefore, only one representative of each class needs to be investigated, for which it is sufficient to only consider the relative placement of components to each other. We have not been able to identify related work in the literature, which would combine layout requirements with technical processing orders.

Layout problems consider specific geometric arrangements of objects in a two- or three-dimensional space. Our model and encoding is applicable to layout problems if the relative placement of objects that we enumerate is sufficient to characterize a layout and if the specific geometric arrangement of objects can be computed in a separate post-processing step. Usually, our relative placement represents an infinite set of specific geometric layouts satisfying the generated permutation, orientation, and alignment. It is up to the application and details of the post-processing which layout from this infinite set will be instantiated.

Layout problems are studied in several fields of computer science and consequently, the literature on this problem is vast and spread across numerous conferences and journals. The two most important fields are graph layout and VLSI design, see [25, 9] for selected overviews on these two fields.

Our approach does not focus on solving the compaction problem, i.e., it does not directly generate placements of minimal size as they are for example required in VLSI design, but instead solely generates relative component placements that satisfy specified object alignment and orientation relations. Therefore, our method can still generate valid layout candidates in such application contexts, but the compaction of a placement must be added as a post-processing step removing white space or gaps, while preserving the relative placement. This may not be very effective as the separation of layout candidate generation and posterior compaction may hinder an application to directly steer a solution algorithm towards layouts of minimal size.

Layout problems have also been studied in relation with graphical user interfaces, for example when placing windows on a computer screen [16]. With the emergence of the world wide web, layout problems also occur when designing or generating web pages [4]. The ACM conference series on Advanced Visual Interfaces [1] gives a good overview on different approaches. In the context of computer graphics, layout problems have also been studied when printing labels [2]. Document layout has been in the focus of researchers for quite some time and is often using a “grid and boxes”-based approach [15]. Our placement method shares a similar “grid and boxes” approach as we also abstract our technical components by rectangular frames and use arrangement lines to specify additional reference alignment points. Newer variants of layout problems occur in the context of design mock-ups, see for example [7]. More generally, information presentation often requires to address placement or more general layout problems, see [19] for an overview of activities until the year 2000.

Constraint-based layout has been of particular interest to intelligent graphical editors that preserve spatial relationships between graphical elements, i.e., that move a set of related elements when one element is moved. Constraint-based layout, however, studies a very different problem where a user works with an editor moving around objects, and constraints are used to move connected objects along or to detect that certain move operations are not valid, see for example [3]. We share with these approaches the representation of objects using rectangular frames and arrangement lines, but differ significantly in the class of layouts that we enumerate and which are restricted to the equivalence classes under our predefined processing order. Furthermore, general layouts would also allow or require that objects overlap in specific ways, which we cannot express in our method as this does not occur in the class of manufacturing problems that we consider. Our modeling concepts place object relative to each other, but do not allow to express specific geometric values or distances.

In the context of industrial manufacturing, facility layout is a widely studied problem, see for example [18, 14, 13] for recent pointers to the literature and a discussion of the problem and potential solution approaches. Facility layout is becoming a more dynamic problem with the increasing flexibility of modern manufacturing approaches and trends such as mass customization, rapid product changes and adaptable processes. Production facilities need to be rearranged and relocated to minimize production times and material-flow costs. At the same time, the cost for dynamically changing the facility layout should be minimized. The dynamic facility layout problem is related to our placement problem under a predefined processing order. In our manufacturing application, we are seeking a processing order of minimal production time over a large set of possible layouts, which is in fact similar to the facility layout problem. In other words, we search to generate a layout that leads to a minimal-time processing order. Requirements for the processing order are specified by numerous constraints describing properties of machines and manufactured products. In our application, we are able to abstract the layout problem to a model of relative positioning of objects and we can use very compact integer-based encoding techniques. Some of our techniques could be helpful to encode some aspects of facility layout problems and to effectively control the heuristic search approaches that are discussed in [14]. For example, describing seeds for local search methods or directing a search algorithm into a specific direction becomes easy when applying our integer-based encoding. However, our emphasis is on the relative placement of objects (machines, products) to each other, but less on assigning facilities to locations. This assignment problem, which constitutes the core of dynamic facility layout, completely abstracts from geometric details and focuses on material flow costs between locations.

Cable network layout problems constitute another class of layout problems of technical components, see for example [20]. A recent application of spanning trees to optimize the layout of sewer systems is described in [5]. In this approach, cyclic graphs are cut into trees and the search space of possible spanning trees of low maximum path length is explored to find high-quality placement solutions. Spanning trees are commonly used to solve general graph layout problems and are more specifically used in VLSI design to solve routing problems [24]. We share with these approaches the idea to use spanning trees to encode layout relationships between objects, which in our case represent the alignment of components.

A manufacturing problem, that is related to the class of problems we consider, is the problem of punching metallic sheets where punches are applied in specific orders to a sheet in order to cut out holes [12]. The arrangement of these holes has a significant influence on the sequence of the punching operations and, when using progressive dies, the order in which dies can be applied is constrained. However, as we are not experts in this field of application, we were not able to determine precisely to which extent our method could be beneficial for this class of problems. If there is some freedom in the arrangement of the wholes during the design of metallic sheets, our method could be applicable to explore options in the sheet design.

Decomposing a larger layout problem into sub-problems and then overlaying the individual solutions to obtain an overall solution is a common technique and instantiates the divide-and-conquer principle in the domain of layout problems. VLSI lithography approaches among many others heavily rely on this technique. Similarly, we present an approach where we develop three efficient computational sub-procedures to generate solutions for different aspects of our placement, which we model by permutation, orientation, and shift. We are able to obtain the complete specification of the placement by a simple combination of the solutions to each of the aspects.

3 The Class of Placement Problems under Predefined Processing Orders

Typical manufacturing problems consist of a sequence of processing operations that have to be applied to components in a specific order. The components to be processed are usually arranged with the help of fixtures such that machines (or humans) can apply the required processing operations. Depending on the manufacturing problem, the placement of the components can be straightforward with no or only very few placement options. However, quite often many degrees of freedom for placing components exist, making the placement problem a challenge in itself as its solution influences the available options to achieve an optimal sequence of process operations. The sequence of processing operations usually must obey a predefined set of hard constraints that enforce a partial processing order. These hard constraints origin from two sources. First, the type of processing operation that must be applied, e.g., drill before paint. Second, the construction of machines that can restrict how certain of its parts can move, e.g., grippers, jet nozzles or other tools. Typically, some of these hard constraints are specific to the problem or application area, but independent of a concrete problem instance, while others only apply to particular problem instances. Once a partial processing sequence has been found that satisfies these hard constraints, the sequence can be completed by taking additional optimization criteria into consideration. Finding such a sequence that optimizes some multi-criteria objective function is a key requirement when automating applications in industrial manufacturing. Finding a good placement of components that enables optimal, e.g., low-cost, processing sequences is a problem in itself and can be quite challenging when many degrees of freedom exist. Often, the available placement options lead to a combinatorial explosion in the number of possible placements and for each possible placement, an optimal processing sequence must be searched. It is thus of interest to have algorithms at hand that can generate and evaluate placement options in an efficient and compact manner.

In this article, we investigate a class of manufacturing problems that is characterized by the following properties:

- One or several operating tools move in a predefined order as a consequence of design and construction.
- The tools perform processing tasks on technical components that are to be arranged in a specific placement. Each placement potentially induces a different processing sequence under the predefined order.
- Each processing sequence can in turn be evaluated with respect to different optimization criteria.

3.1 Typical Examples of Placement Problems

Figure 7 shows the bird's eye view of a conveyer belt on which components are placed for further processing by tools, e.g., a soldering rod that is installed above the belt. The belt moves from left to right, whereas the tools move from one side of the belt to the other performing their processing or assembly tasks, i.e., top to bottom in this view. The movement of the tools and the belt is given by construction, inducing a predefined top-down, right-to-left processing order that is specific to the problem, i.e., the assembly line work, but independent of the actual problem instance, e.g., the specific processing operations on the components placed on the belt. The placement of components on the belt can be freely chosen such that different processing sequences can be realized under the predefined processing order.

Figure 8 illustrates another typical application scenario. A robot spraying components of a technical device, e.g., a car body, will likely follow a top-down processing order because of the flow direction of paint or other coating materials. If the processing involves different colors or coating materials, the robot will likely paint components in one color first, then change the paint and color the remaining surfaces. Both constraints define a partial processing order again. They are specific to the problem, i.e., painting parts on a vertical surface, but independent of the actual problem instance, i.e., the specific components to be painted.

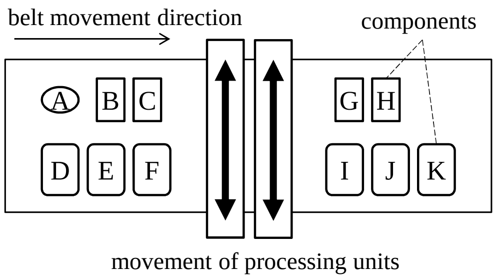


Figure 7: A conveyor belt with processing tools dictating a top-down, right-to-left processing order.

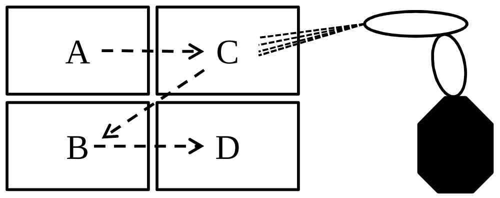


Figure 8: A spray robot following a left-to-right, top-down processing order.

Another important family of applications is illustrated in Figure 9. It originates from the automated manufacturing of switchboards or electrical cabinets, where processing consists in wiring a possibly large number of electrical components or devices. Usually, the components to be wired are arranged on an upright positioned board. The placement of the components has to satisfy certain constraints. For example, some components are to be placed at the bottom of the cabinet to be close to certain connectors, or there may be constraints on wire length such that the corresponding components must be close to each other. These are constraints with respect to a specific problem instance, namely the specific switchboard or electrical cabinet that is manufactured. In addition, wires may occlude components once they are added to the cabinet. This is a natural consequence of gravity when connecting vertically placed components with wires, which is independent of the specific product to be assembled. Hence, a human or robot plugging wires will obey a bottom-up processing order to generally avoid the occlusion of components that need to be accessed later in the processing sequence. Moreover, a left-to-right processing order can result from requirements of production safety. For example, a robot arm should avoid moving over wired connections as it risks to get caught. Therefore, when assuming that the robot is fed with wires from the right-hand side, it will follow a left-to-right processing order. The constraints for improving production safety are independent of the actual problem instance, i.e., the exact wiring of a specific electrical switchboard. The placement of the electrical components on the board can be changed within the placement-specific constraints to evaluate different processing sequences with respect to the given processing order and additional wiring-specific constraints and optimization criteria.

Finally, let us briefly mention an example not from manufacturing, but agriculture.

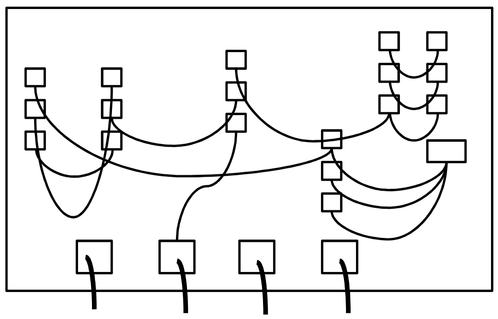


Figure 9: A schematic wiring of an electrical cabinet. Hanging wires occlude components such that a bottom-up processing order must be followed. A left-to-right or right-to-left order is often desirable for production safety reasons.

A farmer who is cultivating different patches of land with a sowing machine will avoid crossing already processed patches. Likewise when applying herbicide, the farmer is recommended to process patches following an upwind order for health reasons. Again, such constraints are specific to the problem, i.e., application of herbicide, but independent of the actual problem instance, i.e., the farmer's plot of land with specific patch locations.

The distinction between constraints specific to the problem and those specific to a concrete problem instance is crucial. The first class applies to all problem instances, such that every processing sequence that is considered as a solution to any problem instance must satisfy these constraints. We assume such a predefined order arising from a set of problem constraints and, as suggested above, derive different processing sequences by changing the placement of components for a specific problem instance. This yields a simple equivalence relation of component placements: Two placements are equivalent if, and only if, they lead to the same processing sequence of components under the given processing order.

3.2 Equivalence of Placements

Let us capture this particular class of processing problems more precisely:

- A tool processes components placed on a two-dimensional plane or grid. The exact dimensions of this plane do not matter. We only assume a global coordinate system with a reference point. Components can be framed and thus abstracted by rectangles with a reference point and given width and height.
- Components to be processed must be accessible for the tool and therefore must not overlap each other.
- There is a universal set of constraints defining an order that every valid processing sequence must obey. Without loss of generality, we assume a left-to-right, bottom-up processing order of the tool motivated by our introductory examples.

- Given the universal left-to-right, bottom-up processing order, every placement of components induces a potentially different processing sequence, see Figure 10.

Figure 10 illustrates how two possible component placements generate different processing sequences. In the situation displayed in this figure, we have 6 components and therefore $6! = 720$ possible processing sequences. For each sequence, we can easily find a corresponding placement of components that induces a specific sequence under the universal left-to-right, bottom-up order.

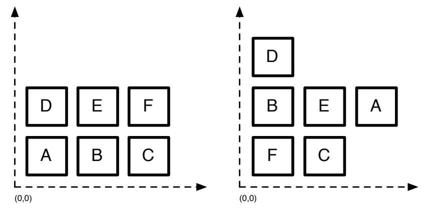


Figure 10: Two possible placements of components that lead to different processing sequences under a left-to-right, bottom-up processing order: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$ for the left-hand placement and $F \rightarrow C \rightarrow B \rightarrow E \rightarrow A \rightarrow D$ for the right-hand placement.

Assuming a left-to-right, bottom-up universal processing order for the tool comes without loss of generality since other processing orders can be taken into account by reflection and rotation of the plane. For example, a top-down, right-to-left processing order can be transformed into a left-to-right, bottom-up processing order by reflecting the plane horizontally followed by a 90° rotation, cf. Figure 11 for illustration.

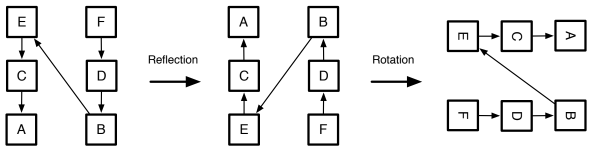


Figure 11: A top-down, right-to-left movement of the tool turned into a left-to-right, bottom-up processing order by reflecting the plane horizontally followed by a 90° rotation.

Conversely, there are many placements inducing the same processing sequence as illustrated in Figure 12. These placements are considered equivalent with respect to the predefined processing order.

For each placement of the kind in Figure 12 we can easily find a horizontal alignment of components that generates the same processing sequence. This means that in principle, it is sufficient to place all components in one horizontal line, in the example A, B, C, D, E, F, to obtain the same processing sequence. Consequently, using the vertical dimension for placing components does not generate new, non-equivalent pro-

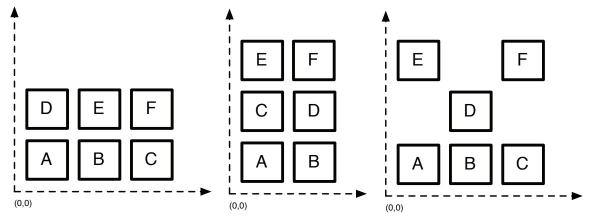


Figure 12: Three placements that induce the same processing sequence under a left-to-right, bottom-up processing order, i.e., $A \to B \to C \to D \to E \to F$. These placements are therefore considered equivalent.

cessing sequences. We will exploit this property later in our algorithms to arrive at the desired compact encoding, see Section 5.

3.3 Dealing with Multiple Processing Areas Within Components

So far, we only considered components without any inner structure. In many technical applications, however, a component contains multiple *processing areas* and the manufacturing steps operate on these areas. For example, when modeling problems from soldering or from the wiring of electrical components, we usually have several soldering points or cavities per component, whose geometric positions inside the component are static. In such a situation, we can obtain new processing sequences by rotating components as illustrated in Figure 13. It might of course happen that some components are too wide, e.g., to be turned by 90 degrees and would then either overlap with other components or range beyond the space on which they can be placed. Such constraints are specific to the problem instance and can be thrown in as additional hard constraints at a later moment.

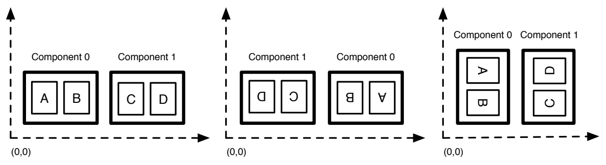


Figure 13: In case of multiple processing areas per component new processing sequences are obtained from rotating individual components. For the left-most example the induced processing sequence is $A \to B \to C \to D$, for the middle example it is $D \to C \to B \to A$ and for the right-most example the processing sequence is $B \to C \to A \to D$.

We argued above that only the relative geometric placement of components with respect to each other matters, and that it is sufficient to consider only horizontal alignments of components when counting equivalence classes or searching for non-equivalent placements. The situation becomes more sophisticated when components with multiple processing areas are involved. Consider the example in Figure 14 with two components having 4 processing areas each. Assuming that each processing area can also be framed and abstracted by a rectangle, we refer to the lower horizontal edge of each processing area with the help of an *arrangement line*. So-called *arrangement lines* or *phantom lines* have been commonly used in the description of layout problems to describe additional constraints of arrangement for parts of graphical objects, see for example [8].

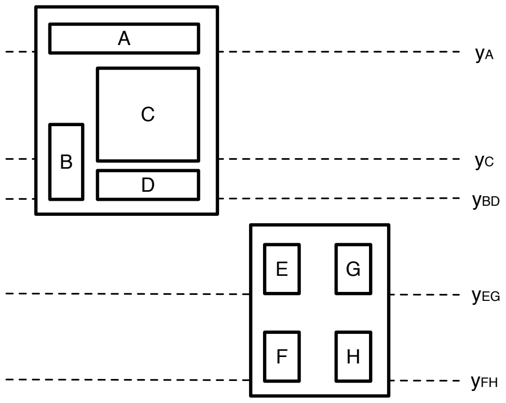


Figure 14: Complex component placements with processing areas are modeled with the help of arrangement lines. Reading arrangement lines along the predefined order gives the processing sequence. In this example: $F \to H \to E \to G \to B \to D \to C \to A$.

In the example in Figure 14, the line y_{BD} denotes the common arrangement line of areas B and D, whereas y_C refers to the arrangement line of processing area C, which are all part of the same upper-left component. Similarly, lines y_{EG} and y_{FH} specify arrangement lines of the areas E, G and F, H within the lower-right component. The processing order for Figure 14 is $F \to H \to E \to G \to B \to D \to C \to A$. Note that we only need to enumerate processing areas along the arrangement lines in the predefined left-to-right, bottum-up order.

Next, imagine that the right-hand component in Figure 14 is moving upwards. As long as we move just a little bit, the processing sequence does not change. In other words, we still have an equivalent placement of components under the universal order. Sooner or later, however, the arrangement line y_{EG} of the right-hand component will match the arrangement line y_{BD} of the left-hand component leading to the situation displayed in Figure 15. This placement is not equivalent anymore to the one in Figure 14 as the predefined order induces now a different processing sequence $F \to H \to B \to D \to E \to G \to C \to A$.

When we keep moving the right-hand component upwards, the processing sequence again does not change until two other arrangement lines match, producing the situation in Figure 16 with processing sequence $F \to H \to B \to D \to C \to E \to G \to A$. We conclude from this observation that the relative vertical alignment between (the processing areas of) components can be represented by arrangement lines. The num-

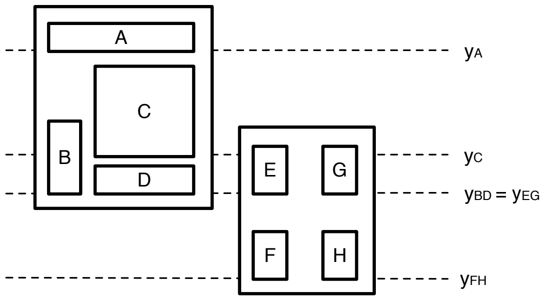


Figure 15: Alignment of arrangement lines y_{BD} and y_{EG} induces processing sequence $F \to H \to B \to D \to E \to G \to C \to A$.

ber of different processing sequences corresponds to the possible number of mutual alignments of arrangement lines between the processing areas.

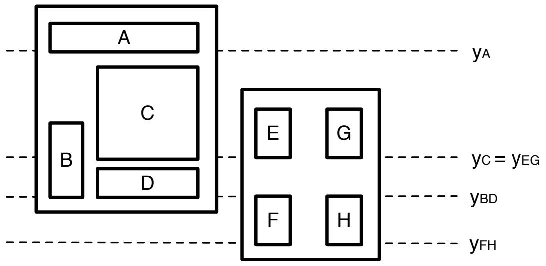


Figure 16: Alignment of arrangement lines y_C and y_{EG} gives processing sequence $F \to H \to B \to D \to C \to E \to G \to A$.

4 Essential Elements of the Placement Encoding and Algorithm

Our discussion has shown that non-equivalent placements of components under the predefined processing order of a tool can be obtained by solely changing the orientation and the horizontal positioning of the components combined with the mutual alignment of the processing areas, which are abstracted by their arrangement lines. This property allows us to describe a placement equivalence class by three different vectors and to devise independent algorithms for each computational problem represented by each vector.

Let us consider the example in Figure 17. Our encoding of this example begins with the vector $C = (c_0, c_2, c_1)$ where each component is identified by a name c_0, c_1, c_2 and a relative horizontal position. The vector states that component c_0 is at the horizontal position 0, component c_2 is at position 1 and component c_1 is at position 2. In the following, when we speak of a component k, we mean the component at position k in this vector. Every permutation of components in this vector leads to a new processing sequence with respect to the predefined order and therefore to a non-equivalent placement.

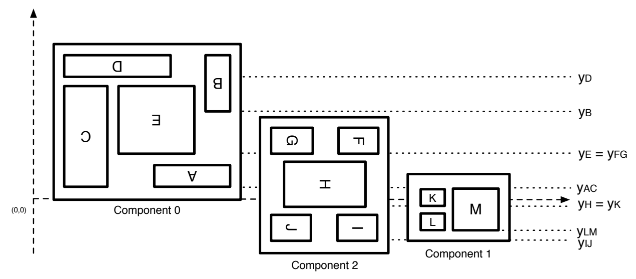


Figure 17: Placement encoded as permutation vector $(0,2,1)$, orientation vector $(2,0,1)$ and shift vector $(0,-180,-200)$. The processing sequence under the universal order is $J \rightarrow I \rightarrow L \rightarrow M \rightarrow H \rightarrow K \rightarrow C \rightarrow A \rightarrow E \rightarrow G \rightarrow F \rightarrow B \rightarrow D$.

Furthermore, new placements can also be created by rotating components with more than one processing area and express these rotations by a second vector. We assume that components can commonly be turned by 0, 90, 180, or 270 degrees without loss of generality. Other angles are possible and can be expressed as multiples of a common basis. In the present example, the basis is 90 degrees. For each component we only need to express the number of turns in the corresponding basis, e.g., 0 standing for 0 degrees, 1 standing for 90 degrees, 2 standing for 180 degrees and so on. In the example, component c_0 has been turned two times, component c_1 has not been turned, whereas component c_2 has been turned once.

Finally, components are moved up or down by a vertical shift of the component arrangement lines to create new, non-equivalent placements. The shift is described by some distance measured in the underlying coordinate system. In the example, component c_0 is aligned along the X-axis of the plane, component c_1 is shifted by -180 units, and component c_2 is shifted by -200 units. Remember that only the mutual alignment of arrangement lines can reveal non-equivalent placements.

Hence, placements are encoded as three independent vectors for *permutation*, *orientation*, and arrangement line *shift*, respectively. For the example in Figure 17, we obtain the following vectors to encode a complex placement with three components and 13 processing areas A to M with arrangement lines depicted as dotted lines.

- *permutation* $(0,2,1)$ means that component c_0 is at position 0, component c_1 is at position 2, and component c_2 is at position 1.
- *orientation* $(2,0,1)$ means that component c_0 is turned by 180 degrees, component c_1 is turned by 0 degrees, and component c_2 is turned by 90 degrees (assuming a basis of 90 degrees).

- *shift* $(0, -180, -200)$ means that component c_0 is aligned along the X-axis of the plane, component c_1 is shifted by -180 units, and component c_2 is shifted by -200 units. Here, the shifts were chosen such that arrangement line y_E of component c_0 is aligned with arrangement line y_{FG} of component c_2, and arrangement line y_H of component c_2 is aligned with arrangement line y_K of component c_1.

This vector representation not only devises a very compact encoding the complex placing problem. For each vector, an independent algorithmic procedure can be devised that enumerates the desired (or alternatively all) permutations, orientations, and shifts. In addition, a placement can be easily controlled through the vector representation. For example, if a specific component k must always occupy the third position in a setup, the order vector is constrained to contain the number 3 at its k-th position. Likewise, if a component only contains a single processing area, such that we cannot obtain new equivalence classes by rotation, the corresponding position in the orientation vector always contains the number 0.

Algorithm 1 gives an overview on how these three procedures play together to generate all possible placements under a predefined processing order.

```
Input: set of $m \in \mathbb{N}$ components
Output: optimal placement of the components

best ← ∞;
candidate ← null;
foreach p : permutation of components do
    foreach o : orientation of components do
        foreach s : shift for the alignment of arrangement lines do
            placement ← generate(p, o, s);
        end
    end
end
return candidate;
```

Algorithm 1: Exhaustive search for optimal placements.

The algorithm uses three sub-procedures to compute the order, orientation, and shift of each component. When combined in an exhaustive search skeleton as shown above, this algorithm enumerates all possible placements. Of course, such an exhaustive generate-and-test approach is prohibitive in many applications due to a large number of potential placements and a complex evaluation procedure. However, the presented compact encoding and the ability to easily control which of the placements are generated, facilitates the development of application-specific algorithms, which can easily trade completeness for performance and only generate promising placements controlled by heuristics that govern certain vector values. In the following, we look at each of the algorithms for the sub-procedures in more detail.

5 Details of the Placement Generation Sub-Procedures

In the previous section, we introduced three vectors to encode the relative placement of a set of components with processing areas in a very compact format. We also sketched an exhaustive search algorithm to compute placement candidates, which uses three sub-procedures to compute the order, orientation, and shift of each component. These sub-procedures rely on the following algorithms:

- permutation: an iterative algorithm well-known from the literature,
- orientation: a simple enumeration of possible orientations encoded as integers,
- shift: a computation of possible arrangement line matchings based on spanning trees and Prüfer codes.

Whereas the first two algorithms are straightforward and can be directly taken from the literature, the computation of the shifts is more elaborate, but can also be effectively achieved by a very powerful and original algorithm that we present further below.

5.1 Generating the Permutation

Each component is encoded by its position in the horizontal component vector. Hence, to generate all possible horizontal alignments of components, we need to generate all possible permutations of a set $\{1,\ldots,n\}$ of numbers. Existing algorithms exploit the fact that permutations can be ordered lexicographically. For example, for the numbers $\{1,2,3\}$ this order is: $(1,2,3) \rightarrow (1,3,2) \rightarrow (2,1,3) \rightarrow (2,3,1) \rightarrow (3,1,2) \rightarrow (3,2,1)$.

Given the i-th permutation in the lexicographic order, an iterative algorithm can easily produce the next permutation $i+1$ for $1 \leq i \leq n!$. This makes it possible to enumerate all permutations in factorial time $O(n!)$, but constant space $O(1)$. The following pseudo-code can be found in most textbooks on algorithms [17]. It takes a permutation vector $a[]$ as input and generates the next permutation in the lexicographic order.

1. Find the largest index k such that $a[k] < a[k+1]$. If no such index exists, the permutation is the last permutation.
2. Find the largest index l such that $a[k] < a[l]$. Since $k+1$ is such an index, l is well-defined and satisfies $k < l$.
3. Swap $a[k]$ with $a[l]$.
4. Reverse the sequence from $a[k+1]$ up to and including the final element $a[n]$.

Step 1 identifies an index k such that swapping only elements with index strictly higher than k cannot give a new permutation that is lexicographically larger. To advance to the next permutation, one must increase $a[k]$. Step 2 finds the smallest value $a[l]$ to swap with $a[k]$. The smallest value is necessary to find the next permutation in the lexicographic order (and not any other larger permutation). Reversing the subsequence in Step 4 then produces the lexicographically minimal permutation larger than the current.

5.2 Generating the Orientation

To efficiently generate all orientations of components in a setup, we adopt the encoding known from binary numbers. Let us assume that m rotations are possible for n different components. This requires to generate m^n orientation variants. For exhaustive search, we use an algorithm that counts from 0 to $m^n - 1$. Each generated number is interpreted as the encoding of a vector of base m, which directly gives the number of turns for all components. For example, let us again assume the possible orientations 0, 90, 180, and 270 degrees. For a component vector of size 3 we obtain $4^3 = 64$ different orientation variants. By counting from 0 to 63 we can generate the vectors $(0,0,0)_4, \ldots, (3,3,3)_4$ where 0 means that all 3 components are 0° rotated, whereas $(3,3,3)_4$ means that all components are 270° rotated. The subscript $m = 4$ refers to the number of possible rotations, such that each vector can be interpreted as a base-m encoded integer. For example, if variant number 57 is requested, we simply encode 57 in base-4 and obtain $(3,2,1)_4$. This corresponds to the setup where the first components is 270° rotated, the second is 180° rotated, and the third is 90° rotated. For local search, we randomly draw a number from $\{0, \ldots, 63\}$ and proceed identically.

5.3 Generating the Shift

We compute the shift for the vertical alignment of components and their selected arrangement lines in two steps.

First, we introduce the notion of an *alignment graph* to represent the desired vertical alignments of the components only. Each component corresponds to a node in this graph labeled with the index of the component. Two nodes are connected if the corresponding components are aligned with respect to some, not yet further specified pair of arrangement lines.

Second, we enrich the alignment graphs with an array specification that tells us exactly which of the arrangement lines of the components are aligned with each other. We discuss the computation of these arrays in Subsection 5.3.2 further below, but let us first focus on the alignment graphs to facilitate the understanding of our approach.

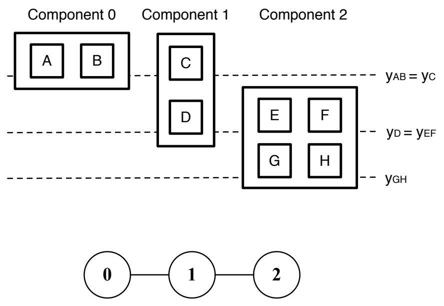


Figure 18: A placement of three components and its corresponding alignment graph.

Alignment graphs must not contain cycles. This can best be seen by an example. Figure 18 shows three components with processing areas A to H, arrangement lines depicted as dotted lines, and the corresponding alignment graph. We observe that arrangement line y_{AB} of component 0 is being aligned with arrangement line y_C of component 1, and arrangement line y_D of component 1 is being aligned with arrangement line y_{EF} of component 2. This induces the alignment graph in the same figure. Imagine now that we additionally wanted to align component 0 with component 2 in order to create a cycle in the alignment graph. This, however, would break the alignment of component 1 with either 0 or 2. Alignment graphs are therefore cycle-free and they are always connected since only mutual alignments of arrangement lines can reveal non-equivalent placements. Consequently, alignment graphs can be represented as trees. Note further that alignment graphs are undirected because the connectivity property is symmetric. Furthermore, our notation of alignment graphs does not refer to a particular alignment of selected arrangement lines. A connection between node i and j precisely means that *some* arrangement line of component i is being aligned with *some* arrangement line of component j. We subsequently write (i, j) for a mutual alignment of components i and j and assume without loss of generality that $i < j$.

In the following, we explain in detail how alignment graphs can be represented in a very compact manner that also facilitates to systematically enumerate all possible alignment graphs for a given set of components.

5.3.1 Generating Alignment Graphs with Prüfer Codes

Since alignment graphs are undirected trees, Cayley's theorem [6] states that there exist n^{n-2} different alignment graphs with $n \in \mathbb{N}$ nodes. This can be proven by establishing a bijection between trees with n nodes and so-called Prüfer codes [21], i.e., vectors of length $n-2$ containing integers $\{0, \ldots, n-1\}$. Similar to the encoding of orientations, such a vector can be interpreted as a base-n encoding of a number between 0 and $n^{n-2}-1$. Hence, we obtain a one-to-one mapping between alignment graphs and the numbers in $\{0, \ldots, n^{n-2}-1\}$. This again allows us to iteratively enumerate alignment graphs by counting numbers. Given a tree with n nodes, Algorithm 2 outputs its corresponding Prüfer code.

```
Input: tree $\mathscr{T}$ with nodes $\{0, 1, \ldots, n-1\}$
Output: Prüfer code for $\mathscr{T}$

result ← $\langle\rangle$;
for $1 \ldots n-2$ do
    $v$ ← leaf with smallest label;
    $k$ ← neighbor($v$);
    result.add($k$);
    remove $v$ from tree;
end
return result;
```

Algorithm 2: Prüfer Encoding

The setting in Figure 18 with only $n = 3$ components is too simple to illustrate Algorithm 2. Let us therefore consider a more complex problem with $n = 5$ components, which is shown in Figure 19. As one can see, the mutual alignments of components are $(0,1),(2,4),(1,3)$ and $(1,4)$.

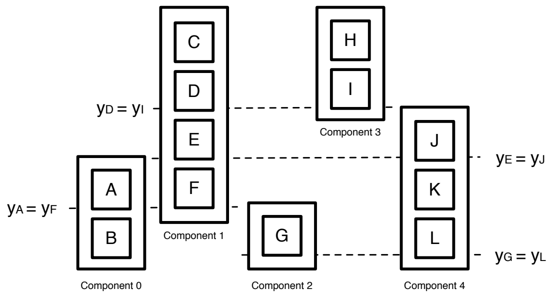


Figure 19: A placement with 5 components mutually aligned according to the following pattern: $(0,1),(2,4),(1,3),(1,4)$.

Figure 20 shows on the left-hand side the alignment graph that corresponds to the placement of components in Figure 19. On the right hand side, the figure shows the alignment graph for a different alignment of the same components $\mathscr{T}_2 = \{(1,2),(2,4),(3,4),(0,3)\}$ used further below.

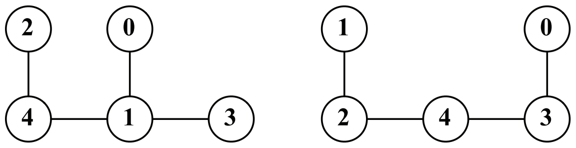


Figure 20: Two alignment graphs for the 5 components from Figure 19: $\mathscr{T}_1 = \{(0,1),(2,4),(1,3),(1,4)\}$ (left) and $\mathscr{T}_2 = \{(0,3),(1,2),(2,4),(3,4)\}$ (right).

Next we illustrate how the Prüfer encoding algorithm computes the Prüfer code for the left-hand tree in Figure 20. To do so, it performs the following three steps as it iterates from 1 to $n-2$ with $n = 5$ in our example:

1. leaf with smallest label: $v = 0 \rightarrow k = 1 \rightarrow \langle 1 \rangle$. Now we remove the node 0 from the graph and obtain node 2 as the leaf with the smallest label.
2. leaf with smallest label: $v = 2 \rightarrow k = 4 \rightarrow \langle 1,4 \rangle$. Now we remove the node 2 from the graph and obtain node 3 as the leaf with the smallest label.
3. leaf with smallest label: $v = 3 \rightarrow k = 1 \rightarrow \langle 1,4,1 \rangle$

The Prüfer code for this example is therefore $\langle 1,4,1 \rangle$. More generally, the set of all possible Prüfer codes for $m = 5$ nodes has $5^3 = 125$ elements ranging from $0 = \langle 0,0,0 \rangle_5$ to $124 = \langle 4,4,4 \rangle_5$. We can again interpret these elements as base-5 encoded integers.

In order to transform any Prüfer code back into its corresponding alignment graph, we apply Algorithm 3 to a Prüfer code.

```
Input: Prüfer code $P = \langle p_1, \dots, p_{n-2} \rangle$
Output: tree $\mathscr{T}$
start with $n$ isolated nodes $V = \{0, 1, \dots, n-1\}$;
for $i = 1 \dots n-2$ do
    $v \leftarrow$ smallest node in $V$ not contained in $P$;
    connect $v$ to node with label $p_i$;
    remove $v$ from the set $V$;
    remove $p_i$ from the sequence $P$;
end
connect the two remaining elements in $V$;
return $\mathscr{T} \leftarrow$ set of connected pairs;
```

Algorithm 3: Prüfer Decoding

To illustrate this algorithm, let us consider the example Prüfer code $P = 89 = \langle 3, 2, 4 \rangle_5$, which represents the alignment graph as shown on the right-hand side of Figure 20. Since $|P| = n-2$, we know that $n = 5$ and therefore initialize $V = \{0, 1, 2, 3, 4\}$. The algorithm performs the following steps:

1. smallest label $\notin P$: $v = 0 \rightarrow p_1 = 3 \rightarrow$ new edge $(0,3) \rightarrow$ update $V = \{1,2,3,4\}, P = \langle 2,4 \rangle$
2. smallest label $\notin P$: $v = 1 \rightarrow p_2 = 2 \rightarrow$ new edge $(1,2) \rightarrow$ update $V = \{2,3,4\}, P = \langle 4 \rangle$
3. smallest label $\notin P$: $v = 2 \rightarrow p_3 = 4 \rightarrow$ new edge $(2,4) \rightarrow$ update $V = \{3,4\}, P = \langle \rangle$
4. new edge $(3,4)$

As expected, the output is $\mathscr{T} = \{(0,3), (1,2), (2,4), (3,4)\}$, i.e., the right-hand tree of Figure 20.

To sum up, iterating over the n^{n-2} different alignment graphs can be implemented by counting from 0 to $n^{n-2} - 1$, producing a Prüfer code by encoding the number in base n and creating the corresponding tree using Algorithm 3. The output of this algorithm is a tree with n nodes represented by a set of edges $\mathscr{T} \subseteq \{0, \dots, n-1\}^2$. For the left-hand tree in Figure 20, we have for example $\mathscr{T} = \{(2,4), (1,4), (0,1), (1,3)\}$. This tree contains the information about pairs of components that are mutually aligned. We are now prepared to enter the second step of the shift computation where we determine the specific pairs of arrangement lines within the components that we want to align.

5.3.2 Generating Arrangement Line Pairs

So far, our placement algorithm has chosen a permutation of the components, followed by an orientation for each component, and finally an alignment graph using Algorithm 3. However, the specific arrangement lines for the alignment of the components have not yet been chosen. We explain this second step of the shift computation next.

If we write $n_i \in \mathbb{N}$ for the number of arrangement lines of component $i \in \{0, \ldots, n-1\}$, the number of possible different alignments of pairs of arrangement lines induced by an alignment graph is

$$seq(\mathscr{T}) = \prod_{(i,j)\in\mathscr{T}} (n_i \cdot n_j). \tag{1}$$

Note that the number of arrangement lines of a specific component depends on its orientation. Figure 21 illustrates this situation. If the component is not turned (left-hand side), we identify two arrangement lines for the two rows of processing areas, which are of interest for a potential alignment. If the component is turned by 270°, we can identify three arrangement lines. It is thus important that the components' orientation is known when the alignment of arrangement lines is computed in Algorithm 1. In general, the specification of arrangement lines for a specific component is application-dependent and may depend on other reference points than processing areas which can be of interest for a potential alignment.

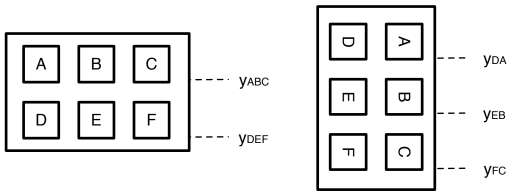


Figure 21: The number of arrangement lines may depend on component orientation.

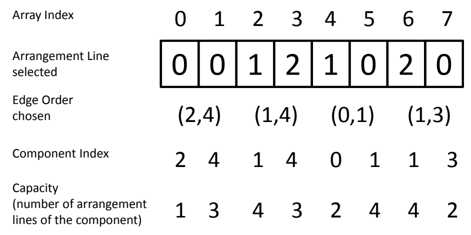
Let us return to the example in Figure 19 and its component alignment, which is depicted by the alignment graph on the left-hand side in Figure 20. We determine the arrangement line counts for the components n_0 to n_4 and obtain the following values: $n_0 = 2$, $n_1 = 4$, $n_2 = 1$, $n_3 = 2$ and $n_4 = 3$.[2] The number of possible arrangement line combinations therefore is $seq(\mathscr{T}) = (1\cdot 3)\cdot(3\cdot 4)\cdot(2\cdot 4)\cdot(4\cdot 2) = 2304$.

We use arrays of length $2(n-1)$ to enumerate all possible alignments of arrangement lines for a given alignment graph $\mathscr{T}$ with $n-1$ tree edges and each edge consisting of 2 nodes. We introduce a particular order to represent the edges in the tree and can thereby identify each pair of neighboring array cells with a tree edge or, equivalently, each array cell with a node (component) in the edge sequence. It does not matter how exactly this order of edges is chosen. Each array cell must hold integers between 0 and

[2]Note that not all arrangement lines are shown in Figure 19 for reasons of clarity. The figure only depicts those lines which are aligned with each other.

the arrangement line count of the corresponding component (minus 1). We refer to this number as the *array cell capacity*.

Figure 22 illustrates the array representation for the example in Figure 19. We create an array of length 8 as displayed, which contains 8 cells to represent the potential alignments of arrangement lines for the 5 components because of $2(n-1)=8$. We have selected the edge order $(2,4),(1,4),(0,1),(1,3)$. Cell 0 thus refers to an arrangement line of component 2, whereas cells 5 and 6 refer to (possibly different) arrangement lines of component 1. Cell 0 has capacity (arrangement line count) $n_2 = 1$ and can therefore contain the integers $\{0,1\}$, cells 5 and 6 have capacity 4 and can therefore contain the integers



Array Index	0	1	2	3	4	5	6	7
Arrangement Line selected	0	0	1	2	1	0	2	0
Edge Order chosen	(2,4)		(1,4)		(0,1)		(1,3)	
Component Index	2	4	1	4	0	1	1	3
Capacity (number of arrangement lines of the component)	1	3	4	3	2	4	4	2

Figure 22: Data structure for enumerating all possible, mutual alignments of arrangement lines.

The array representation allows us to systematically enumerate all alignments of arrangement lines for a particular alignment graph. Again we count with integers. We initialize an integer array of length $2(n-1)$ with zeros $\langle 0,0,0,0,0,0,0,0\rangle$ and iteratively enumerate the arrays as follows:

1. If the right-most cell is smaller than its capacity - 1, increment its value.
2. If the right-most cell has reached its capacity - 1, set it back to 0 and carry to the next cell.
3. If all array entries become zero again, the complete set has been enumerated.

Each array corresponds to a particular alignment of arrangement lines with respect to the previously determined permutation, orientation, and alignment graph. For the example from Figure 19 and the selected order of tree edges, we obtain the vector $\langle 0,0,1,2,1,0,2,0\rangle$ as shown in Figure 22. It represents an alignment where arrangement line 0 of component 2 is aligned with arrangement line 0 of component 4 ($y_G = y_L$), arrangement line 2 of component 4 is aligned with arrangement line 1 of component 1 ($y_E = y_J$), arrangement line 1 of component 0 is aligned with arrangement line 0 of component 1 ($y_A = y_F$), and arrangement line 2 of component 1 is aligned with arrangement line 0 of component 3 ($y_D = y_I$).

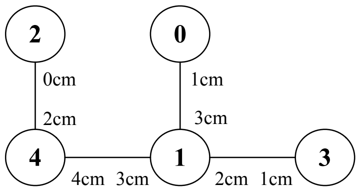
With this final computation step, the characteristics of a placement are completely determined. In the next section, we discuss how specific physical layouts can be constructed from the relative placements generated by our method.

6 From Relative Placements to Physical Layout

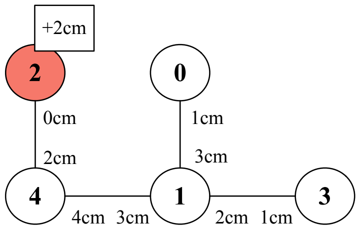
Once the desired placement of components has been generated, we need to translate the relative positioning of the components into the corresponding physical layout. Placing the components in the correct order and with the right orientation for each component from the permutation and rotation information is straightforward. How to remove white space or to generally compact a layout has been studied by many authors and therefore, we do not want to discuss this problem here. We refer the interested reader to [11] for an overview on classical VLSI design approaches and [26] for a more recent discussion of the problem in the context of two-dimensional cell layout. It is of more interest how the generated alignments of arrangement lines can be further processed to find the exact geometrical shift distances for each component such that all arrangement lines are correctly aligned. For this purpose, we developed the following recursive coloring algorithm. The algorithm step-wise marks the nodes of the alignment graph, initially assuming that all nodes are unmarked.

1. Choose a component $i \in V$ with exactly one unmarked neighbor $j \in V$ such that $(i, j) \in \mathscr{T}$. Initially, all leaves satisfy this condition. If multiple candidates exist, choose randomly.
2. Extract the arrangement lines b_i and b_j to be aligned from the alignment vector.
3. Determine the shifting distance for component i by calculating $d = b_j - b_i$.
4. Shift component i by distance d and recursively shift all components in the subtree of i by the same distance d.
5. Mark component i and repeat until only one unmarked component is left.

Reconsider the alignment graph $\mathscr{T} = \{(2,4),(1,4),\ (0,1),(1,3)\}$ and the alignment vector $\langle 0,0,1,2,1,0,2,0\rangle$. We need the y-coordinate of each arrangement line in order to calculate the exact vertical shift for each component. These coordinates are displayed next to each edge in the figures below.



Start



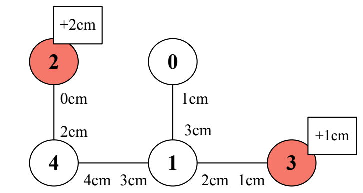
Step 1

We display the unit *cm* next to the coordinates to distinguish between an arrangement line number in the arrangement line vector and its corresponding y-coordinate. According to the alignment vector, we must for example align arrangement line 0 for component 2 and arrangement line 0 for component 4. We find the coordinates $0cm$ and $2cm$ along the edge between components 2 and 4 in the alignment graph, which means that arrangement line 0 of component 2 has y-coordinate $2cm$ and arrangement line 0 of component 4 has y-coordinate $1cm$ with respect to the global coordinate system. Note that these vertical displacements were not shown in Figure 19 when this example was introduced. Our algorithm now proceeds as follows, arbitrarily selecting component 2 as the first component to begin with:

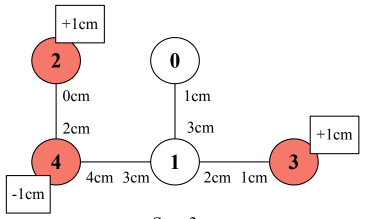
1. We choose component 2 that has only one unmarked neighbor, which must be aligned with component 4. The arrangement line height of 2 is $0cm$ and the arrangement line height of 4 is $2cm$. Hence, components 2 and 4 can be aligned by shifting 2 by a distance of $2cm - 0cm = 2cm$. This is shown in the box added to component 2. We mark component 2 and continue.

2. We choose component 3 that has only one unmarked neighbor. It can be aligned with component 1 by shifting 3 by $2cm - 1cm = 1cm$. We mark 3 and continue.

3. We choose component 4 with only one unmarked neighbor. It is aligned with component 1 by shifting component 4 $3cm - 4cm = -1cm$. As component 2 has already been aligned with component 4 the new shifting distance $-1cm$ also applies to component 2. We mark 4 and continue.

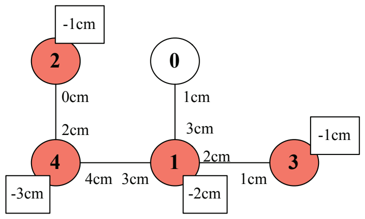


Step 2



Step 3

4. We choose component 1 with only one unmarked neighbor. It can be aligned with component 0 by shifting component 1 by $1cm - 3cm = -2cm$. As component 1 has already been aligned with components 3 and 4 (and 4 itself with 2), the new shifting distance $-2cm$ recursively applies to all these components. We mark 1 and continue.



Step 4

5. Only one unmarked component is left, so the algorithm stops.

These steps geometrically create the selected placement on the plane. Remember that only the relative positioning of components with respect to each other matters. When placing the components in the selected order, the horizontal distances can be chosen arbitrarily (or compacted following domain-specific design rules) as long as the components do not overlap.

7 Exhaustive and Local Placement Search

Our modeling and encoding method and the three computational sub-procedures constitute a powerful algorithm to generate all possible placements of technical components with distinguished processing areas that are subject to a predefined processing order as it can often be found in manufacturing problems. The predefined processing order naturally induces an equivalence relation between different component placements, where only the rotation of individual components, their relative horizontal positioning as well as mutual vertical alignments with respect to arrangement lines can produce new non-equivalent placements.

At the heart of our algorithm lies a very compact and efficient encoding scheme that allows us to represent placements by the three aspects of permutation, orientation, and shift and devise efficient sub-procedures to compute each aspect individually. Our sub-procedures can be easily embedded into local, systematic, and evolutionary search approaches and can be combined with other well-established finite domain optimization techniques such as constraint programming. In the following, we briefly discuss how an embedding into exhaustive (systematically enumerating all placements) or local (only generating selected placements) search algorithms can be achieved.

We showed that all relevant placement information can be represented by integers. The relative horizontal position is encoded by a permutation index, the orientation of each component is a base-*m* encoded number with *m* being the number of possible turns, mutual alignments of components are represented by Prüfer codes, and the choice of specific arrangement lines for the alignment is also a counting number. As all algorithms are iterative and use only constant or linear memory, they can be used in standard search procedures to optimize some given placement evaluation function.

In Algorithm 4 we complete the skeleton shown in Algorithm 1 for exhaustive search with a generic penalty function that represents the evaluation of a concrete placement with respect to one or several optimization criteria. Each placement is then evaluated by some evaluation procedure that assigns a penalty to a placement. The placement with the smallest penalty is returned as the output of the algorithm. The evaluation procedure can also consist of another search algorithm, which determines the optimal processing sequence that is enabled by a specific placement. It would assign the cost of this processing sequence as penalty to the placement.

We are fully aware that the high exponential time complexity generally does not allow complete search for more than small placement problems. However, our efficient encoding scheme in combination with the iterative nature of each sub-procedure facilitates the implementation of arbitrary search algorithms. It is therefore not difficult to transform this algorithm into a randomized or evolutionary search procedure or, with a suitable application-specific heuristic, into a heuristic or local search procedure. Consult [23] for an introduction to generic search methods.

Our encoding scheme further shows that all relevant information about equivalence classes of component placements can be represented as counting numbers with precisely determined lower and upper bound. This makes our approach particularly well-suited to be combined with other well-established optimization techniques such as finite domain constraint programming [22] for example. In this case, we would model the permutation index with the *all-different* constraint and, due to the one-to-one mapping between Prüfer codes and spanning trees, also benefit from specialized global constraints for spanning-trees to capture component alignment [10]. The alignment of arrangement lines then corresponds to edge weights of the spanning tree. This means that our operational specification of the algorithm can also be translated into a declarative constraint-based representation, which can then be fed into arbitrary constraint

```
Input: vector of $n \in \mathbb{N}$ components
Output: best evaluated placement

best ← ∞;
candidate ← null;
p ← firstPermutation();
repeat
    for $i = 0, \ldots, m^n - 1$ do
        o = toBase(i,m) (rotation vector)
        for $j = 0, \ldots, m^{m-2} - 1$ do
            $P$ ← toBase(j,n)
            (index to Prüfer code)
            $\mathscr{T}$ ← decode($P$)
            (Prüfer code to tree)
            $align \leftarrow [0, \ldots, 0]$;
            repeat
                placement ← generate(p, o, s);
                value ← penalty(placement);
                if $value < best$ then
                    best ← value;
                    candidate ← placement;
                end
            until (align ← next(align,$\mathscr{T}$))==null;
        end
    end
until (p ← nextPermutation(p)) == null;
return candidate;
```

Algorithm 4: Exhaustive search algorithm enumerating the complete space of non-equivalent placements under the predefined processing order.

solvers. In addition, such a declarative representation makes it easy to add additional domain- or instance-specific constraints to the model, which can then be considered by a solver when searching for a solution to a placement problem.

8 Mixed horizontal and Vertical Placements

So far, we considered the special case of placements with only horizontal or only vertical arrangement lines. Imagine next the following setup with three objects A, B, C. A has three arrangement lines (left, upper and lower edge), B has one arrangement line (left edge) and C has one arrangement line (upper edge). For simplicity, no rotations are possible in this example. We first look at the objects having at least one horizontal arrangement line (A and C in this example) and apply the above algorithm to find all possible placements for this subset A and C. Then, we do the same for all objects having at least one vertical arrangement line (A and B in this example). The possible placements for the horizontal and vertical alignment sub-problems are shown in Figure 23.

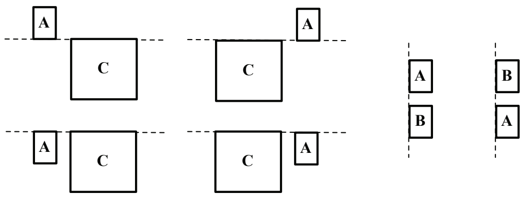


Figure 23: Horizontal and vertical alignments of subsets A and C (left) and A and B (right).

Now we create all consistent overlay placements from the horizontal and vertical sub-problem placements. For example, the second placement in the upper row in Figure 24 is obtained by combining the first horizontal placement with the first vertical placement. For this simple example, all 8 combinations of horizontal and vertical placements are possible (although only 4 are displayed here). This is, however, not always the case. When the overlay placements are created, a consistency check must ensure that all arrangement line constraints are satisfied. If this is not the case, the overlay is not a valid solution to the placement problem for the entire object set.

9 Conclusion

In this article, we studied layout and placement problems of technical components that can be abstracted to relative arrangements of components. Horizontal and vertical arrangements between components and sub-components can be captured as well as requirements of rotation. We describe these modeling elements by the concepts of permutation, orientation and shift and present a compact encoding based on integer vectors together with algorithms that compute and manipulate these integer vectors by

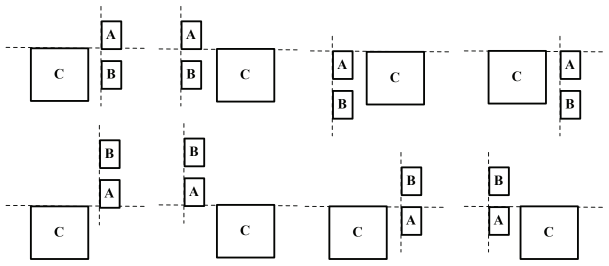


Figure 24: Four consistent overlay placements.

counting algorithms. The compact encoding can provide a very efficient foundation for any state-based search algorithm, and it is easy to configure to the needs of a specific application.

We show that for placement problems under predefined processing orders, where the processing order of the components is given by a manufacturing process, only different relative placements can induce different processing orders.

Our current work focuses on the combination of the placement generation with an evaluation function that determines optimal placements. In our case, the evaluation function involves a highly complex optimization problem with a large search space. The optimization problem results from the fact that any placement will enable many different possible processing sequences under the predefined order and that we need to find the optimal processing sequence for a given placement using the evaluation function. We therefore investigate techniques to heuristically generate only "promising" placements and then apply local search to find good, but not necessarily optimal processing sequences. Our compact encoding as integer vectors allows us to effectively implement the heuristic control by constraining specific integer values.

Acknowledgement

This work was partially funded by the Swiss Commission for Technology and Innovation CTI. We would like to thank Res Gilgen for an implementation of the presented algorithms, which allowed us to experiment with placement problems in a domain of manufacturing electrical appliances.

References

[1] ACM conferences on advanced visual systems (2012). Http://dblp.uni-trier.de/db/conf/avi/index.html

[2] Awofala, A.O., Singh, N.: Constraint network approach to the design and manufacture of labels in a high-variety label-printing environment. Journal of Intelligent Manufacturing **7**, 499–514 (1996)

[3] Badros, G.J.: Constraints in interactive graphical applications (1998)

[4] Badros, G.J., Borning, A., Marriott, K., Stuckey, P.: Constraint cascading style sheets for the web. Technical Report UW CSE 99-05-01, University of Washington (1999)

[5] Burch, N., Holte, R.C., Müller, M., O'Connell, D., Schaeffer, J.: Automating layouts of sewers in subdivisions. In: Proc. European Conference on Artificial Intelligence (ECAI), *Frontiers in Artificial Intelligence and Applications*, vol. 215, pp. 655–660. IOS Press (2010)

[6] Cayley, A.: A theorem on trees. Quart. J. Math **23**, 376–378 (1889)

[7] Ceylan, D., Li, W., Mitra, N.J., Agrawala, M., Pauly, M.: Designing and fabricating mechanical automata from mocap sequences. ACM Trans. Graph. **32**(6), 186 (2013)

[8] Cruz, I.F.: Expressing constraints for data display specification: A visual approach. In: V.A. Saraswat, P.V. Hentenryck (eds.) Principles and Practice of Constraint Programming, pp. 445–470. MIT Press (1995)

[9] Das, D.: VLSI Design. Oxford Univ Press (2011)

[10] Dooms, G., Katriel, I.: The minimum spanning tree constraint. In: F. Benhamou (ed.) Principles and Practice of Constraint Programming - CP 2006, *Lecture Notes in Computer Science*, vol. 4204, pp. 152–166. Springer Berlin Heidelberg (2006)

[11] Gerez, S.: Algorithms for VLSI Design Automation. John Wiley & Sons (1999)

[12] Ghatrehnaby, M., Arezoo, B.: Automatic strip layout design in progressive dies. Journal of Intelligent Manufacturing **23**, 661—677 (2012)

[13] Hosseini, S., Khaled, A.A.: A survey on the imperialist competitive algorithm metaheuristic: Implementation in engineering domain and directions for future research. Applied Soft Computing **24**, 1078—1094 (2014)

[14] Hosseini, S., Khaled, A.A., Vadlamani, S.: Hybrid imperialist competitive algorithm, variable neighborhood search, and simulated annealing for dynamic facility layout problem. Neural Computing and Applications **25**, 1871—1885 (2014)

[15] Jacobs, C.E., Li, W., Schrier, E., Bargeron, D., Salesin, D.: Adaptive grid-based document layout. ACM Trans. Graph. **22**(3), 838–847 (2003)

[16] Kandogan, E., Shneiderman, B.: Elastic windows: Improved spatial layout and rapid multiple window operations. In: Proc. Workshop on Advanced Visual Interfaces, pp. 29–38. ACM Press (1996)

[17] Knuth, D.E.: Generating all tuples and permutations. The art of computer programming Vol. 4 Fasc.2. Addison-Wesley (2005)

[18] Lin, Q.L., et al.: Integrating systematic layout planning with fuzzy constraint theory to design and optimize the facility layout for operating theatre in hospitals. Journal of Intelligent Manufacturing **26**, 87—95 (2015)

[19] Lok, S., Feiner, S.: A survey of automated layout techniques for information presentations (2001). From the webpage http://graphics.cs.columbia.edu/publications/

[20] Malakooti, B.: Unidirectional loop network layout by a LP heuristic and design of telecommunications networks. Journal of Intelligent Manufacturing **15**, 117–125 (2004)

[21] Prüfer, H.: Neuer Beweis eines Satzes über Permutationen. Arch. Math. Phys. **27**, 742–744 (1918)

[22] Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)

[23] Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach (3. internat. ed.). Pearson Education (2010)

[24] Sait, S.M.: VLSI Physical Design Automation: Theory and Practice. World Scientific Publishing (2004)

[25] Tamassia, R. (ed.): Handbook of Graph Drawing and Visualization. Chapman and Hall/CRC (2013)

[26] Ziesemer, A., et al.: Automatic layout synthesis with ASTRAN applied to asynchronous cells. In: 5th IEEE Latin American Symposium on Circuits and Systems (LASCAS), pp. 1–4. IEEE (2014)