4th Conference on Production Systems and Logistics

Towards Finding Optimal Solutions For Constrained Warehouse Layouts Using Answer Set Programming

Pascal Kaiser[1], Andre Thevapalan[2], Moritz Roidl[1], Gabriele Kern-Isberner[2], Marco Wilhelm[2]

[1]*Chair of Material Handling and Warehousing, TU Dortmund University, Germany*
[2]*Chair of Logic in Computer Science – Information Engineering, TU Dortmund University, Germany*

Abstract

A minimum requirement of feasible order picking layouts is the accessibility of every storage location. Obeying only this requirement typically leads to a vast amount of different layouts that are theoretically possible. Being able to generate all of these layouts automatically opens the door for new layouts and is valuable training data for reinforcement learning, e.g., for operating strategies of automated guided vehicles. We propose an approach using answer set programming that is able to generate and select optimal order picking layouts with regards to a defined objective function for given warehouse structures in a short amount of time. This constitutes a significant step towards reliable artificial intelligence. In a first step all feasible layout solutions are generated and in a second step an objective function is applied to get an optimal layout with regards to a defined layout problem. In brownfield projects this can lead to non-traditional layouts that are manually hard to find. The implementation can be customized for different use cases in the field of order picking layout generation, while the core logic stays the same.

Keywords

warehousing; layout planning; order picking; optimization; answer set programming

1. Introduction

Warehouse layout planning is a well-studied problem with good results for predefined shapes and designs [1]. However, a completely rule-based generation without predefined shapes and designs is missing. This is considered a poorly structured decision problem [2]. Therefore, planners rely on their personal experience in designing warehouses [3]. Although there are systematic planning approaches to address this problem [2], the automation of the internal layout design problem depends on idealized conditions, e.g., rectangular space, predefined layout (multi-block, fishbone, etc.) or parallel aisles [4–6]. Each of these designs have to follow the same rules to be a feasible solution for an order picking system, e.g., every storage location has to be accessible.

For highly constrained warehouses with spatial or resource-related constraints, it is even more difficult to find viable layout options. This is the case, for example, in brownfield projects with given structural restrictions and other constraints (e.g., existing material flow restrictions). Planning such highly constrained order picking layouts can be challenging even for experts, with no good solutions being found. One reason for this may be the planners bias based on their personal experience, and new layout options may not even be considered. Formalizing this expert knowledge to be machine-readable creates new possibilities for the automated generation of layouts. When all of the rules for the layout generation are formalized, it is no longer necessary to idealize the conditions (e.g., rectangular space, predefined layout). The layouts are generated

DOI: https://doi.org/10.15488/13480
ISSN: 2701-6277

implicitly following the formalized knowledge. We are investigating the use of a declarative rule-based approach for the automated generation of layouts for this problem.

Answer set programming (ASP) is a declarative problem solving approach from the area of logic programming [7] which can solve highly complex combinatorial problems [8]. Such problems are modeled in form of logic programs which consist of possibly defeasible rules that lead to a conclusion when the premise of the rule applies. The defeasibility of a rule is formalized by the use of default negation which causes that the rule is usually applicable unless the default-negated part of the rule is true. Therewith, ASP can resolve conflicts which arise from competing objectives automatically which represents an added value compared to classical constraint programming. Typically, ASP programs have several feasible solutions from which the optimal solutions with regard to a certain objective function can be filtered. In [9,10] the authors showed that ASP is an appropriate method for logistical problems. Here, we formalize the constraints and the basic objectives of the warehouse layout generation task by ASP rules and optimize the area utilization in order to find suitable warehouse layouts.

The goal of this paper is to show that ASP is predestined to generate optimal order picking layouts respecting a defined objective function by reason of its declarative nature, readability, and versatility. After the formalization of the knowledge in a logic program, the implementation is capable of generating every optimal layout that satisfies all building restrictions and constraints, considering the optimization objective. Furthermore, it is possible to add and remove constraints and objectives in form of rules in a straightforward manner. Default negation allows to easily define exceptions to general rules. Together with their declarative nature, which deems the order of rules irrelevant, answer set programs are generally easily extendable. We will show that with all these key advantages, ASP constitutes an effective tool to assist the planning of layouts especially in brownfield projects. To demonstrate the benefits of ASP, we extended the solution presented in [9] in order to showcase an implementation that creates layouts for a manual order picking process based on given building restrictions and constraints abstracted from a real-world example.

The remainder of this paper is structured as follows. After presenting an introduction into answer set programming and warehouse layout generation in Section 2, Section 3 outlines the goals of this paper and provides an overview of the implementation's main functionalities. In Section 4, the developed implementation is illustrated and key components are explained. Then, the program is used to generate a possible layout for a given problem instance (building restrictions, number of racks, base position, etc.). The final section concludes the paper with a short summary and provides an outlook on future work.

2. Preliminaries

This section provides the necessary background on ASP and layout generation. Therefore, we first introduce ASP with the underlying definitions in order to illustrate on which bases the implementation in Section 4 is built. Afterwards we elaborate on the challenges of the layout generation and which indicators and goals can be pursued with different layouts.

2.1 Answer Set Programming

Answer set programming [7] constitutes a declarative programming paradigm based on logic programs which allows for a rule-based description of real-world problems. ASP features the concept of *default negation* with which it is possible to formulate default statements that hold unless an exceptional case arises. Therewith, conflicts in ASP programs can be resolved automatically. For more details, see [8,11]. Here, we consider ASP based on *normal logic programs (NLPs)*, which are sets of *rules r* of the form

$$h :- b_1, \dots, b_m, not\ b_{m+1}, \dots, not\ b_n. \quad (1)$$

where h and each b_i are atoms. An atom a is of the form $p(t_1, \ldots, t_k)$ where p is a predicate of arity k, and each t_i is either a constant or a variable, like in classical first-order logic. Conventionally, variables are written in uppercase letters and constants in lowercase letters. Variable-free programs, rules, respectively atoms are called *ground*. The prefix *not* in *r* indicates a default-negated atom. Intuitively, the rule *r* states that unless any of the default-negated atoms is true, the *rule head h* follows from $b_1, \ldots, b_m$. Otherwise, the rule is blocked. Rules with $n = 0$ are *facts*, in this case we omit the symbol ": -", and rules with an empty head are *constraints*. Constraints serve to filter out unwanted solutions.

The semantics of an NLP $\mathcal{P}$ is defined over *reducts* of $\mathcal{P}$ which are built w.r.t. possible solutions of $\mathcal{P}$, more formally w.r.t. *interpretations* which are sets of ground atoms. The *reduct* $\mathcal{P}^{\mathcal{I}}$ of a ground NLP $\mathcal{P}$ w.r.t. an interpretation $\mathcal{I}$ is basically $\mathcal{P}$ but from which all rules *r* are removed that contain a default-negated atom not b_i with $b_i \in \mathcal{I}$. Further, from all remaining rules the default-negated atoms are removed as well as they are satisfied in $\mathcal{I}$ and, hence, do not block *r*. An interpretation $\mathcal{I}$ now is a solution of $\mathcal{P}$, formally called answer set, if $\mathcal{I}$ satisfies every rule in $\mathcal{P}^{\mathcal{I}}$, i.e., if $\{b_1, \ldots, b_m\} \subseteq \mathcal{I}$ implies $h \in \mathcal{I}$ for every rule $h :- b_1, \ldots, b_m \in \mathcal{P}^{\mathcal{I}}$, and $\mathcal{I}$ is set-minimal with this property. An interpretation $\mathcal{I}$ is an answer set of a non-ground NLP $\mathcal{P}$ if $\mathcal{I}$ is an answer set of the grounding of $\mathcal{P}$ which means the ground NLP $\mathcal{P}'$ that is obtained by replacing every variable in $\mathcal{P}$ by each available constant once. While logic programs have a unique solution, NLPs with default negation may have several answer sets.

The implementation described in this paper is implemented with the ASP system *clingo*[1]. Technically, the grounding of NLPs is executed by *ASP grounders* (e.g., *gringo*) and the answer sets are computed by *ASP solvers* (e.g., *clasp*). Clingo offers several language extensions and the integration of external methods (either in the programming language *Python*[2] or *Lua*[3]) into the solving process which will be shown later.

2.2 Warehouse Layout Generation

A large extent of the warehousing costs is already determined during the design phase. This design phase is characterized by many trade-offs between conflicting objectives which means it is a highly complex task, resulting in a large number of feasible designs. The authors in [12] define the warehouse design as a structured approach to decision making at a strategic, tactical and operational level.

An integral part of the warehouse design process are decisions on the warehouse layouts. It can be distinguished between two types of layout decision problems. The first type of decision problems is the placement of various departments such as receiving, picking, storage, etc. [13]. This is usually called the facility layout problem and it results in a warehouse block layout that is often based on minimizing the handling costs. The second type of problem is called internal layout design or aisle configuration problem. It consists of placing the equipment, storage space, paths, etc. within the departments [4,14].

The warehouse layout problem is usually defined as finding an optimal or at least good layout of the storage or order picking area. In contrast to the other areas, the order picking area layout design is not represented sufficiently in the literature [1,4]. It is mainly focused on conventional layouts or adaptations of it (fishbone or flying-V layouts) which is sufficient for greenfield projects [5,15]. Therefore, a limited amount of different layouts can be found in the majority of warehouses today. Usually these warehouses use a conventional warehouse layout for the storage area and the order picking area. These traditional warehouse layouts have a rectangular shape with parallel straight aisles [16]. There are two possibilities to change the aisles, at the front and rear of the warehouse. These aisles meet the main aisles at a right angle (as shown on

[1] https://potassco.org/clingo/
[2] https://python.org/
[3] https://lua.org/

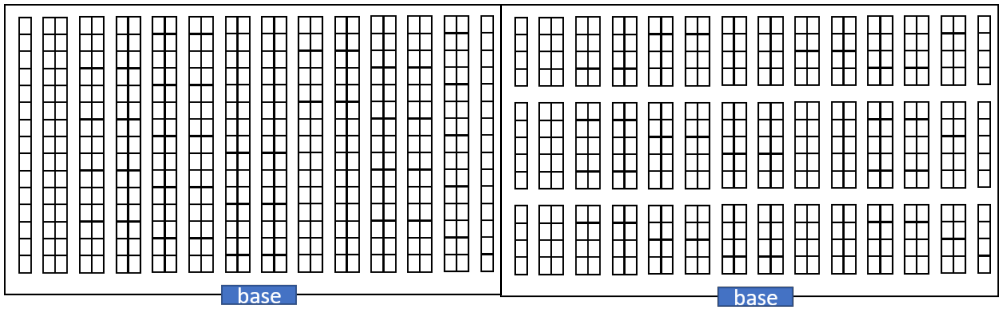

Figure 1: Order picking layouts

the left in Figure 1: Order picking layoutsFigure 1). There are some modifications of this general form that are usually created by adding one or more additional cross aisles [5]. Such layouts are then called multi-block layouts (as shown on the right in Figure 1) [17].

The optimization of layouts usually takes place regarding two objectives, capacity and performance. These two aspects are conflicting with each other, optimizing the capacity usually has negative influence on the performance and vice versa. Several objectives can be pursued within these categories, e.g., minimizing inventory, maximizing storage capacity, maximizing area or space utilization, minimizing travel distance or travel time. Ultimately all these objectives aim at reducing costs [1,18–20]. Within a greenfield project, the capacity is usually set for the warehouse and thus, the optimization aspect is performance. In contrast, in a brownfield project the total available space is set [21]. Maximizing the capacity can be an important objective. Nevertheless, the performance still has to be considered in order to reduce the operational costs. This leads to the problem of warehouse layout planners on how to define the objective function and generate warehouse layouts with given restrictions.

3. Problem Formulation

In order to assist the layout planning process, we propose a modular framework that allows planners to input project-specific key values like the size of the available warehouse, the available types of racks and their respective sizes, and the amount of racks that have to be positioned. They are also able to define more detailed, project-specific properties and constraints regarding the project, e.g., inaccessible parts of the warehouse, constraints regarding the placement of racks of specific types, or the minimum/maximum distances between elements. The implementation presented in this paper constitutes a key part of such a framework with the goal to integrate ASP into the layout planning process. An overview of the framework's workflow is depicted in **Error! Reference source not found.**. It starts with the user who can enter the aforementioned key instance values. This instance data is then added to the problem encoding which contains the general facts and conditions that hold for the typical warehouse layouts like ensuring that each rack must be accessible. The problem encoding can be adapted and extended by further project-specific rules and constraints. The ASP solver computes the answer sets of the logic program where each answer set represents a layout based on the knowledge base modeled in the logic program. Subsequently, the set of layouts can be further reduced using additional algorithms via external functions. The filtered layouts can then be rendered as 2D graphics and displayed to the user.

4. Implementation

In this section, we outline the main aspects of the implementation.

Problem Instance: In ASP, logic programs are often divided into the *problem encoding* and the *problem instance* [8]. The problem instance can be viewed as the input data for the logic program that varies from

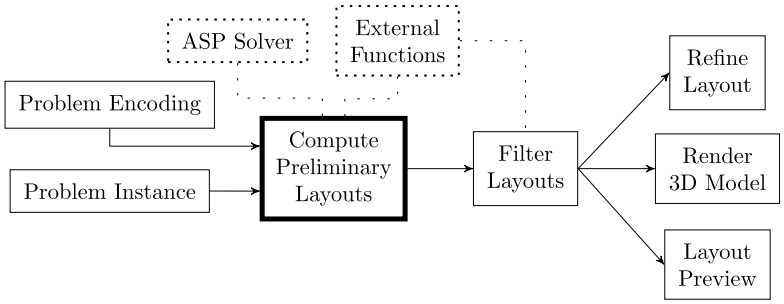

Figure 2: Framework overview

case to case. The base of our implementation is a logic program where the problem encoding contains the encoding of structural warehouse elements, the definition of general dependencies, and constraints regarding their positioning. The problem instance comprises case-based, individual specifications, e.g., details regarding the structure of the order picking area. Thus, the same problem encoding can be used for different use cases.

Let $\mathcal{P}$ be the answer set program of our implementation. The following rules r_1-r_8 constitute a part of the problem instance of $\mathcal{P}$[4].

```
r1: #const size_x = 20.        r5: coords(1,1..size_x/2,1..size_y/2).
r2: #const size_y = 20.        r6: coords(2,size_x/2+1..size_x,1..size_y/2).
r3: elem_quantity(1,47).       r7: blocked(1..2,1..2).
r4: elem_quantity(2,13).       r8: blocked(19..20,1..5).
```

The order picking area is represented by a *grid structure* comprising x columns and y rows where a *cell* stands for a specific position in the area and is denoted by its coordinates. This kind of representation allows a coarse-grained but flexible modeling of the available area. A cell is either free, blocked from the start (which is illustrated by rules r_7 and r_8), or occupied by another element, e.g., a rack. Using the representation of an order picking area as a grid offers high flexibility with respect to scalability. In rules r_1 and r_2, a language extension called *constants* is used that works analogously to constants in common programming languages, i.e., constants are globally accessible but fixed variables. With the constants `size_x` and `size_y`, the size of the area is defined. Other mandatory instance values include the amount of racks for each rack type that each layout has to contain and the area restriction regarding each type of racks, i.e., the planner is able to define that certain rack types must be positioned in specific areas of the layout. Here, r_3 and r_4 define that 47 racks of type 1 and 13 of type 2 must appear in a layout. Rule r_5 ensures that racks of type 1 are only allowed to be positioned inside the lower left quadrant of the warehouse whereas rule r_6 states that racks of type 2 must be positioned in the lower right quadrant.

In the following, we present the key parts of the problem encoding.

Defining Part: In the defining part, we provide auxiliary predicates and specify further general details regarding the order picking area and the structural elements. The following rules lay down the dimensions regarding two of the four rack types and state the possible orientations a rack can have.

```
r9: elem_type(1,"BlockStorage",1,1).    r10: elem_type(2,"ShelvedRacking",3,1).
r11: types_rack(1;2;3;4).               r12: orient(h;v).
```

Rule r_9 represents an element type named “BlockStorage” with id 1 that occupies exactly one cell. With rule r_{10} another rack type named “ShelvedRacking” is defined with id 2 and the width of three cells and depth of one cell. In rule r_{11}, the distinction between racks and other structural elements like conveyor belts or bases is stipulated by explicitly stating that element types with ids 1,2,3 and 4 are rack types. The concept of

[4] The code snippets partially contain language extensions that go beyond the definitions stated in Section 2. For a formal explanation of such extensions, we refer the reader to [22].

rotating an element is represented by *orientation*-literals in rule r_{12}. The available orientations are *horizontal* where an element either faces the north or south side of the layout and *vertical*, where the element faces the east or west side.

Another important functionality in this encoding is the assurance that in every layout, each rack is accessible from the base, that is, there must be a path consisting of free cells from the base to each rack.

```
r13: reach(10,5,1).      r14: reach(11,5,2).
r15: reach(X,Y,TID) :- reach(X+1,Y,TID), not blocked(X,Y), coords(X,Y).
r16: elem_reach(ID,TID,h,X,Y) :- elem(ID,TID,_,h,X,Y), reach(X,Y+1,TID).
```

For this, we define the cells directly above the base as "reachable" (e.g., rules r_{13} and r_{14} for rack types 1 and 2, respectively). Using (recursive) rules like rule r_{15}, every free cell that is adjacent to a reachable cell is also flagged as reachable. In case of r_{15}, we mark all cells right of a reachable cell as reachable if they are not blocked. This rule shows how default negation can be used to define exceptions to a general rule, that is, a cell in the layout is reachable unless it is marked as blocked. By rule r_{16}, the program collects all horizontally positioned elements that have a reachable cell above. Doing this for all possible combinations of racks and neighboring cells, we can later check if all racks are indeed accessible, that is, if all racks have a reachable cell in front of them.

Generating Part: In the generating part of this program, the goal is to calculate the position of all racks that are requested via the instance data. A rack that is positioned in a layout is represented by the cells it occupies and a cell that is occupied by a rack is encoded by an `elem`-literal. The following rules r_{17}-r_{19} constitute a segment of the program's *generating part*.

```
r17: elem_core_coords(T,X,Y) : coords(T,X,Y) :- elem_quant(T,Q).
r18: 1 {elem_core(@get_rack_id(X,Y),T,O,X,Y) : orient(O)} 1
           :- elem_core_coords(T,X,Y), elem_orient(T).
r19: 1 {elem(ID,T,N,O,X,Y) : coords(T,X,Y) , not blocked(X,Y)} 1
           :- X=STX..ENX,Y=STY..ENY, elem_core(ID,T,h,STX,STY),
              elem_type(T,N,SX,SY), ENX=STX+SX-1, ENY=STY+SY-1.
```

The exact positions of a rack are gradually computed. Keep in mind that the declarative nature of ASP does not involve a specific order or preference over rules. Instead, the importance of rules can be indirectly given using certain dependencies between rules. In this case, we compute more general `elem_core_coords`-literals that encode the starting point of each rack, i.e., racks larger and taller than one cell is first represented by a single cell and expanded according to their actual size using another rule. Rule r_{17} states that for every rack type T there have to be Q `elem_core_coords`-literals in the answer set. The coordinate restrictions defined in r_5 and r_6 are used to let the racks appear in specific areas of the layout according to their type. Before the racks can be expanded, their respective orientation have to be established. In r_{18}, to each `elem_core_coords`-literal, an orientation (see r_{12}) is added, yielding fitting `elem_core`-literals. In the head of r18, the external atom `@get_rack_id(X,Y)` represents the return value of a function that is located in a connected Python script. This function uses the coordinate values X and Y to compute and return a unique id for the respective rack based on its coordinates. This does not only create explicit references to each rack, it also induces a total ordering over all racks.

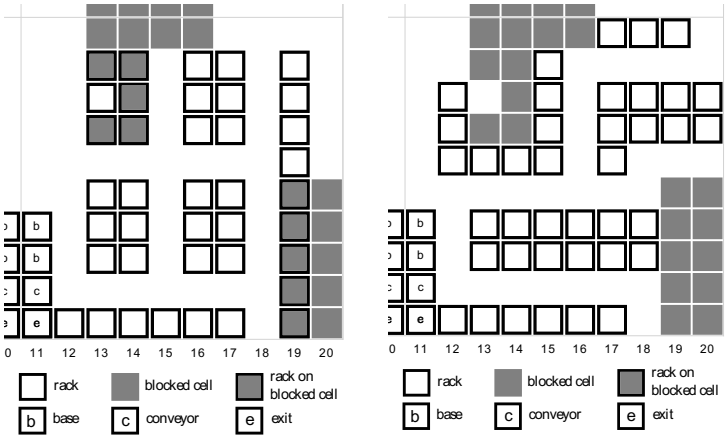

Figure 3: Effect of the default-negated literal not blocked(X,Y) in rule r_{19}: Consideration of blocked areas

For racks that are derived to be horizontal, rule r_{19} states the expansion of each `elem_core`-literal by deriving the aforementioned `elem`-literals. A literal `elem(ID,T,N,O,X,Y)` encodes a (part of a) rack named *N* of type *T* with an id ID that is occupying cell (X, Y). Since during the expansion, the value in ID is adopted from the respective `elem_core_coords`-literal, every such unique identifier in `elem(ID,T,N,O,X,Y)` refers to each part of the expanded rack in the layout. Note that *N* identifies each of these parts as belonging to rack *N*.

The default-negated body literal `not blocked(X,Y)` in rule r_{19} prevents the ASP solver from placing racks on blocked cells. The effect of this literal is that, by default, a rack can be expanded to any cell of the grid unless this cell is blocked (see Figure 3). For vertical racks, a rule analogously to r_{19} is considered in the implementation.

Testing Part: According to the generating part, the layouts contain the correct number of racks and are positioned in the respective area, but there are no restrictions defined yet to ensure that the positioning of the racks is sound (e.g., racks do not overlap with other racks and every rack is accessible). For that reason, in the *testing part* of the program, constraints are added that prevent undesirable layouts.

```
r20: :- elem(ID1,_,_,_,X,Y),elem(ID2,_,_,_,X,Y),ID1!=ID2.
r21: :- elem(_,300,_,_,X,Y),elem(_,T,_,_,X-1..X+1,Y-1..Y+1), types_rack(T).
r22: :- not elem_reach(ID,TID,O,X,Y), elem(ID,TID,_,O,X,Y).
```

As only the starting points of racks are positioned in the layout and expanded separately according to the actual respective size, the expansion can lead to the overlapping of racks. Rule r_{20} prevents such errors by stating that rack parts that share the same coordinates must not belong to different racks. Another aspect to consider are minimum distances, e.g., between racks and the base. With rule r_{21}, any layout that is solved contains a clearance of at least one cell between the base and racks of any type, where `element(_,300,_,_,X,Y)` stands for any element that belongs to the base. Rule r_{22} makes sure that every layout where a positioned rack is not accessible from the base is omitted. In this manner, further constraints can be added to satisfy further restrictions and requirements.

Optimization Criteria: As logic programs output all possible solutions that satisfy all rules, a vast amount of layouts can arise. Therefore, for all computed layouts further properties are calculated externally in order to obtain optimal layouts. From the logistical viewpoint, there are several key values that can be used to define optimal layouts. In the current implementation, we exemplify the optimization of such values by minimizing the path lengths between each pair of racks and additionally minimizing the path from each rack to the base. In both cases, we use an external Python script that computes the different minimal path lengths for each layout using existing algorithms for breadth-first search and outputs those layouts that have minimal values.

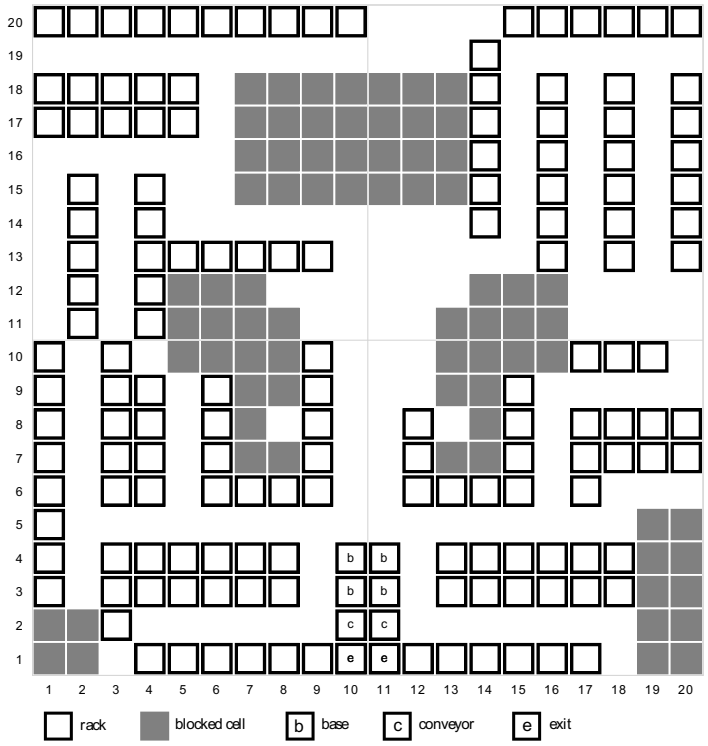

Figure 4: Example of a generated layout

5. Case Study

In order to show the capabilities of the implementation, we conduct a small case study based on a real-world example. The aim of this case study is to illustrate that it is possible to generate order picking layouts based on defined rules and show its different possibilities, e.g., positioning of different systems, constructional constraints and implementation of objective function. The goal of the use case is the positioning of different types of racks inside a predefined layout for a manual order picking system. Such different types of racks can be e.g., shelf storage or pallet racks. The basic layout is a 20 meters by 20 meters square with restricted cells. Within restricted cells, racks cannot be positioned. These cells can be e.g., walls, pillars or rooms and are colored grey in Figure 4. Further restricted cells are the basis, where every picking task starts and ends, and a corresponding conveyor belt. This information is encoded in the problem instance (Section 4).

To show that the implementation is able to handle different types of racks, the task is to position four types of racks. Since from a logistical point of view, it is not advantageous to mix different kinds of racks, for each group of racks an area where to put them is predefined. The racks to be positioned are 47 1x1-racks of size 1 meter by 1 meter, 13 1x3-racks of size 1 meter by 3 meters, seven 1x5-racks of size 1 meter by 5 meters and five 1x6-racks of size 1 meter by 6 meters. Racks are marked as black outlined rectangles. The smallest racks should be positioned in the bottom left quadrant of the layout, the next bigger kind of racks in the bottom right quadrant of the layout. The second largest racks in the top left quadrant and the largest racks in the top right quadrant. The defining part (Section 4) of the encoding specifies these different types of racks. Besides defining the different types of racks, further layout specific restrictions can be defined.

In order to generate valid layouts, basic rules of layout planning have to be obeyed. These are general rules that apply for most of the warehouses, such as two racks cannot be positioned in the same cell or that every rack needs to be accessible from one side. This is encoded in the testing part (Section 4).

With these defined input parameters, we are generating unique layout models for the racks. The layout generation results in 474 different models for the 1x1 racks, 163 different models for the 1x3-racks, 22 models for the 1x5-racks and 826 different models for the 1x6-racks. The generation of all layouts took

approximately 51 seconds. In order to get an optimal solution, we are calculating the travel distance between two racks and between the racks and the base for each model. The objective is to minimize the total travel distance in the layout. For this calculation, the orientation of a rack has to be considered, i.e., a rack must face a reachable cell. In some cases, a rack can have multiple possible orientations. As in each layout a unique orientation must be specified for each rack, additional models can arise. As a consequence, regarding the 1x1-racks, 1x3-racks, 1x5-racks, and 1x6 racks, we obtain 653440, 169, 32, and 1340 different models, respectively. The current implementation allows the calculation of distances in 50-200 milliseconds per model. Here, we reach total execution times of 107263.05 seconds (1x1-racks), 36.5 seconds (1x3-racks), 8.99 seconds (1x5-racks), and 169.47 seconds (1x6-racks). Considering all requirements and the optimization directives, we get ultimately a single layout that is optimal w.r.t the total travel distance (see Figure 4). If desired, a user can always inspect the next best options to obtain a better overview of the results.

To make further improvements to the layout, a manual refinement process can follow. In this case for example, the rack in cell (3,2) creates a dead space to guarantee minimum travel distances. The dead space can be avoided by moving this rack into cell (3,1). Additionally, the rack in cell (2,11)-(2,15) could be moved to cell (1,11)-(1,15) to get a passage between the different areas. To improve the visualization of the layout for a warehouse planner, the output data of the program can be used to generate a three-dimensional layout. This layout can then be used to validate the generated layouts as showcased in [9]. After this refinement process, the new rack positions can then be entered into the program instance in order to check if all defined constraints are still obeyed. This ensures that the warehouse planner does not generate an invalid layout.

6. Conclusion and Future Work

We have shown that ASP poses a powerful tool to enhance the layout planning workflow by presenting a logic program that generates feasible layouts where structural elements are positioned in a highly-constrained warehouse meeting predefined optimization directives. As a solver for ASP computes all possible answer sets that satisfy the program, all computed layouts meet the requirements and conditions defined in the program's rules. To yield the most preferred layouts, specifications can be refined further, e.g., by adapting and adding suitable constraints and optimizations statements. The declarative nature of ASP innately implies a high degree of modularity (e.g., the generated solutions are independent of the actual order of the rules in a program) which allows for a continuous improvement of the program and therefore the layout generation process. This simplifies the process of adding or changing constraints in a logic program. Thus, using ASP allows a step by step approach of implementing the constraints and focusing on rather difficult problems at the beginning.

The presented solution is able to generate restricted warehouse layouts within a short amount of time. In future work, we want to optimize the calculation of the travel distances to reduce the total computation time even further. For this, we will explore ways to decrease the number of models without sacrificing the expressiveness of the encoding and further algorithms to measure the distances.

The generated layouts (including those computed without any optimization directives) can be utilized to create additional input data to train operating strategies of *automated guided vehicles* (*AGVs*). As the solver computes every possible layout w.r.t. the program's rules, the AGVs are able to train on potential edge cases, i.e., layouts that in manual layout planning perhaps would not have been taken into consideration. This kind of reinforcement learning constitutes a significant step towards reliable artificial intelligence.

The proposed framework aims at supporting a layout planner in the decision-making process rather than replacing them. One aspect that reflects the need for the planner is highlighted in the case study where even in technically optimal layouts further refinements might be necessary. The possibility to manually refine generated layouts is also crucial and the implementation of a refinement step is part of our future work.

References

[1] Dukic, G., and Opetuk, T., 2012, “Warehouse Layouts,” Warehousing in the Global Supply Chain.

[2] Schmidt, M., 2018, Distribution Center Design Process: Ein Systemtechnikorientiertes Vorgehensmodell Zur Konzeptplanung von Logistikzentren.

[3] Wunderle, A., and Sommer, T., 2014, “Erfahrung Und Augenmaß Zählen,” Hebezeuge Fördermittel.

[4] de Koster, R., Le-Duc, T., and Roodbergen, K. J., 2007, “Design and Control of Warehouse Order Picking: A Literature Review,” European Journal of Operational Research.

[5] Roodbergen, K. J., and Vis, I. F. A., 2006, “A Model for Warehouse Layout,” IIE Transactions.

[6] Cardona, L. F., Soto, D. F., Rivera, L., and Martinez, H. J., 2015, “Detailed Design of Fishbone Warehouse Layouts with Vertical Travel,” International Journal of Production Economics.

[7] Gelfond, M., and Lifschitz, V., 1991, “Classical Negation in Logic Programs and Disjunctive Databases,” New Gener. Comput.

[8] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T., 2012, Answer Set Solving in Practice.

[9] Kaiser, P., Thevapalan, A., Reining, C., Kern-Isberner, G., and ten Hompel, M., 2021, “Generating Highly Constrained Warehouse Layouts Using Answer Set Programming,” Logistics Journal : Proceedings.

[10] Kaiser, P., Thevapalan, A., Reining, C., Roidl, M., Kern-Isberner, G., and ten Hompel, M., 2022, “Hybrid Production: Enabled By Controlling The Output Sequence Of A Matrix Production Using Answer Set Programming,” Procedia CIRP.

[11] Brewka, G., Eiter, T., and Truszczynski, M., 2011, “Answer Set Programming at a Glance,” Commun. ACM, 54(12), pp. 92–103.

[12] Rouwenhorst, B., Reuter, B., Stockrahm, V., van Houtum, G. J., Mantel, R. J., and Zijm, W. H. M., 2000, “Warehouse Design and Control: Framework and Literature Review,” European Journal of Operational Research.

[13] Mohsen, 2002, “A Framework for the Design of Warehouse Layout,” Facilities.

[14] Grosse, E. H., Glock, C. H., and Neumann, W. P., 2015, “Human Factors in Order Picking System Design: A Content Analysis.”

[15] Öztürkoğlu, Ö., Gue, K. R., and Meller, R. D., 2012, “Optimal Unit-Load Warehouse Designs for Single-Command Operations,” IIE Transactions.

[16] Zhang, Z. Y., Liang, Y., Hou, Y. P., and Wang, Q., 2021, “Designing a Warehouse Internal Layout Using a Parabolic Aisles Based Method,” Advances in Production Engineering & Management.

[17] Sancakli, E., Dumlupinar, I., Akcin, A. O., Cinar, E., Geylani, I., and Düzgit, Z., 2022, “Design of a Routing Algorithm for Efficient Order Picking in a Non-Traditional Rectangular Warehouse Layout,” Digitizing Production Systems.

[18] Arnold, D., and Furmans, K., 2007, Materialfluss in Logistiksystemen: Mit 19 Tabellen, Springer.

[19] Köbler, J., Schöning, U., and Torán, J., 1993, “Decision Problems, Search Problems, and Counting Problems,” The Graph Isomorphism Problem: Its Structural Complexity.

[20] Noortwyck, R., Müller, T., Wehking, K.-H., and Weyrich, M., 2018, “Dezentrale Assistierte Planung: Integrierte Layout- Und Systemplanung von Intralogistiksystemen Auf Grundlage Einer Agentenbasierten Software,” Logistics Journal : Proceedings.

[21] Accorsi, R., Baruffaldi, G., and Manzini, R., 2017, “Design and Manage Deep Lane Storage System Layout. An Iterative Decision-Support Model,” The International Journal of Advanced Manufacturing Technology.

[22] Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S., and Wanko, P., “Potassco Guide Version 2.2.0 (2019).”

Biography

Pascal Kaiser is a research assistant at the Chair of Material Handling and Warehousing, TU Dortmund University.

Andre Thevapalan is a research assistant in the working group Information Engineering at the Chair of Logic in Computer Science at the Department of Computer Science, TU Dortmund University.

Moritz Roidl is the chief engineer at the Chair of Material Handling and Warehousing, TU Dortmund University.

Prof. Dr. Gabriele Kern-Isberner is the head of the working group Information Engineering at the Chair of Logic in Computer Science at the Department of Computer Science, TU Dortmund University

Marco Wilhelm is a research assistant in the working group Information Engineering at the Chair of Logic in Computer Science at the Department of Computer Science, TU Dortmund University.