DOI: 10.2195/lj_Proc_kaiser_en_202112_01
URN: urn:nbn:de:0009-14-54219

Generating Highly Constrained Warehouse Layouts Using Answer Set Programming

Generierung stark eingeschränkter Lagerlayouts mittels Antwortmengenprogrammierung

Pascal Kaiser[1]
Andre Thevapalan[2]
Christopher Reining[1]
Gabriele Kern-Isberner[2]
Michael ten Hompel[1]

[1]*Chair of Materials Handling and Warehousing*
Faculty of Mechanical Engineering
TU Dortmund University

[2]*Chair of Logic in Computer Science - Information Engineering*
Department of Computer Science
TU Dortmund University

Generating highly constrained warehouse layouts is a challenging task for layout planners. Those experts create feasible layouts analytically based on their knowledge and experience. The presented paper proposes an AI-based approach to generate feasible warehouse layouts automatically by using answer set programming. The developed implementation is able to generate all feasible combinations for positioning racks inside a given layout while satisfying all the applicable constraints. This supports layout planners by generating solutions for highly constrained problems.

[Keywords: warehousing, layout planning, picking, answer set programming, logic programming]

Die Erzeugung gültiger Lösungen für stark eingeschränkte Lagerlayouts ist eine Herausforderung für Layoutplaner. Diese Experten generieren zulässige Layouts bisher analytisch basierend auf ihrer Erfahrung und ihrem Wissen. Der vorliegende Artikel stellt einen KI-basierten Ansatz vor, um zulässige Layouts automatisch mithilfe der Antwortmengenprogrammierung zu erzeugen. Das vorgestellte Framework ist in der Lage, alle zulässigen Lösungen zu generieren, um eine gewisse Anzahl an Regalen innerhalb des Layouts zu positionieren. Dadurch werden die Layoutplaner bei der Lösung stark eingeschränkter Probleme unterstützt.

[Schlüsselwörter: Layoutplanung, Kommissionierlager, Antwortmengenprogrammierung, Logische Programmierung]

1 INTRODUCTION

Warehouse layout planning is considered a poorly structured decision problem [Sch18]. There are systematic planning approaches to handle these problems, e.g., like [Sch18], but so far, the automation of holistic warehouse layout planning is considered hardly possible [Gud10]. Usually it is executed by interdisciplinary teams whose efficiency suffers from unclear and conflicting goals. Thus, planners heavily rely on their experience to design warehouses [WS14]. Drafts of feasible layouts are created analytically by experts.

For highly constrained warehouses, it is even more difficult to find feasible layout options. This occurs for example in brownfield projects with given building restrictions and other constraints (e.g., existing material flow constraints). Planning such highly constrained warehouse layouts can challenge even the experts to an extent where no good solutions are found. A reason for that can be the subjective influence of the expert that is based on their experience and new layout options might not even be considered.

Those by experts generated layouts can be assessed and enhanced by IT-supported methods [Gud10]. Unfortunately, they only consider subproblems of the overarching goal and can hardly be deployed without excessive domain knowledge [SMM17]. Also, they are often based on idealized assumptions, such as aisles of the same length, rectangular layouts, the absence of traffic conflicts, and so forth. As a result, warehouse planning is a tedious task that combines domain knowledge by experts and restricted applicable methods.

DOI: 10.2195/lj_Proc_kaiser_en_202112_01
URN: urn:nbn:de:0009-14-54219

Answer set programming (ASP) is an approach for declarative problem solving, suited for solving highly complex problems that are closely connected to domain knowledge [GKKS12]. Logic programs enable the modeling of those problems. A core advantage of ASP is that a user does not have to design the solving process of a problem, it is simply enough to describe the problem. Based on the problem description, the possible solutions for the problems are generated by a solver, a form of artificial intelligence (AI).

The goal of this paper is to show that possible warehouse layouts can be generated with an AI-based approach using ASP. There, the expert knowledge has to be formalized in a logic program once. After the formalization of the knowledge, the implementation is capable of generating every feasible layout for all valid building restrictions and constraints. To demonstrate the developed implementation, we will create a layout for a manual order picking process based on given building restrictions and constraints.

The remainder of this paper is structured as follows. Section 2 gives an introduction into extended logic programs and warehouse planning. In Section 3, 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.). Based on the implementation and the application a discussion about the results takes place in Section 5. The final section of this paper will give an outlook on future work.

2 Preliminaries

This section gives a short introduction into extended logic programs and layout planning.

2.1 Extended Logic Programs

In ASP, a problem is modelled in the syntax of logic programs [GKKS12]. In the following, we look at *nondisjunctive extended logic programs* (ELPs) [GL91]. An ELP is a finite set of rules over a set **C** of constants, a set **P** of predicates and a set **V** of variables. Elements of **C**∪**V** are called *terms*[1]. An atom has the form

$$p(t_1, \ldots, t_n)$$

with a predicate p of arity n and terms $t_1, \ldots, t_n$. An atom will be called *grounded* if all terms are constants. A literal L is either an atom A (*positive literal*) or a negated atom $\neg A$ (*negative literal*). For a literal L, the *complementary* literal $\bar{L}$ is $\neg A$ if $L = A$ and A otherwise. For a set X of literals, $\bar{X} = \{\bar{L} \mid L \in X\}$ is the set of corresponding complementary literals. A set of literals is *inconsistent* if it contains complementary literals. A default-negated literal L is called a *default literal*, and is written as *not L*. A *rule r* is of the form

$$L_0 \text{ :- } L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n.$$

with literals $L_0, \ldots, L_n$ and $0 \leq m \leq n$. The literal L_0 is the *head* of *r*, denoted by $H(r)$ and $\{L_1, \ldots L_m, not\ L_{m+1}, \ldots, not\ L_n\}$ is the *body* of *r*, denoted by $B(r)$. Furthermore, we will call $B^+(r) = \{L_1, \ldots, L_m\}$ the set of *positive body literals* and $B^-(r) = \{L_{m+1}, \ldots, L_n\}$ the set of *negative body literals* in *r*. A rule *r* with $B(r) = \emptyset$ is called a *fact*, and if $H(r) = \emptyset$, rule *r* is called a *constraint.* A rule *r* is *positive* if it does not contain any default literals, i.e., $B^-(r) = \emptyset$. An extended logic program is a *positive logic program* if it only comprises positive rules.

Given an ELP $\mathcal{P}$, the *herbrand universe* $\mathcal{U}_\mathcal{P}$ is the set of all constants $c \in \mathbf{C}$ occurring in $\mathcal{P}$. A *grounded* rule $grnd(r)$ of a rule $r \in \mathcal{P}$ is obtained by replacing every variable of r by a constant $c \in \mathcal{U}_\mathcal{P}$. The *grounded program* $grnd(\mathcal{P})$ of $\mathcal{P}$ is then defined as

$$grnd(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} grnd(r).$$

Naturally, for a variable-free program $\mathcal{P}$, it holds that $\mathcal{P} = grnd(\mathcal{P})$. The *herbrand literal base* $\mathcal{HB}_\mathcal{P}$ of $\mathcal{P}$ is the set of all grounded literals with predicate symbols $p \in \mathbf{P}$ occurring in $\mathcal{P}$ and constants $c \in \mathcal{U}_\mathcal{P}$. An *interpretation* $\mathcal{I} \subseteq \mathcal{HB}_\mathcal{P}$ is a consistent set of literals. Given a positive program $\mathcal{P}$, an interpretation $\mathcal{I} \subseteq \mathcal{HB}_\mathcal{P}$ is a *model of* $\mathcal{P}$ if for every rule $r \in \mathcal{P}$ the following holds: $H(r) \in \mathcal{I}$ whenever $B^+(r) \subseteq \mathcal{I}$ and $B^-(r) \cap \mathcal{I} = \emptyset$. An interpretation $\mathcal{I} \subseteq \mathcal{HB}_\mathcal{P}$ is an *answer set of a positive program* $\mathcal{P}$ if $\mathcal{I}$ is a subset-minimal model of $\mathcal{P}$. Answer sets of a program with default negation are determined by its reduct. Given an extended logic program $\mathcal{P}$ and an interpretation $\mathcal{I} \subseteq \mathcal{HB}_\mathcal{P}$, the *reduct* $\mathcal{P}^\mathcal{I}$ of $\mathcal{P}$ *relative to* $\mathcal{I}$ is defined by

$$\mathcal{P}^\mathcal{I} = \{H(r) \text{ :- } B^+(r). \mid r \in \mathcal{P}, B^-(r) \cap \mathcal{I} = \emptyset\}.$$

An interpretation $\mathcal{I} \subseteq \mathcal{HB}_\mathcal{P}$ is an *answer set* of *an extended logic program* $\mathcal{P}$ if and only if $\mathcal{I}$ is an *answer set* of its reduct $\mathcal{P}^\mathcal{I}$. The set of all answer sets of a program $\mathcal{P}$ will be denoted by $AS(\mathcal{P})$, and $\mathcal{P}$ is called *consistent* if and only if $AS(\mathcal{P}) \neq \emptyset$. We say a literal L is *derivable* in an ELP $\mathcal{P}$ if and only if $L \in \bigcup AS(\mathcal{P})$.

The grounding of ELPs is executed by *ASP grounder* (e.g., *gringo*) and the answer sets are then computed by *ASP solvers* (e.g., *clasp*). The implementation described in this paper is implemented with the ASP system *clingo*[2] which uses both clasp and gringo. One major advantage in

[1] Modern ASP systems also allow function symbols but we will omit their definition due to lack of space.

[2] https://potassco.org/clingo/

using clingo is the availability of several language extensions. Furthermore, clingo offers the integration of external methods (either in the programming language *Python*[3] or *Lua*[4]) into the solving process. Examples of ASP rules for the logistics domain and using the language extensions in clingo are given in Section 3.

2.2 Layout Planning

Designing warehouses is the strategical task of determining their size, processes, technologies, composition and organization in accordance to given requirements. Unfortunately, the task is a poorly structured decision problem that suffers from a wide variety of interdependencies. Thus, warehouse designers are highly valuable experts. Formalizing their knowledge and experience for creating computer-aided tools is expected to facilitate warehouse planning.

Several goals can be pursued through warehouse planning, e.g., minimizing transport intensity, minimizing inventory, maximizing storage capacity or maximizing area or space utilization. Ultimately all these goals aim at reducing costs [AF07, Kov17, NMWW18].

The layout planning is one part of the warehouse design. It focuses on the arrangement of resources and functional areas [AF07]. The layout generation is a critical challenge within this task. Layouts are dependent on several goals and constraints. Thus, layout planning is often characterized as a special optimization problem, because the number of alternatives is often infinite. Therefore generating all layout alternatives is impossible [Kov21]. Currently experts rely on their experience when creating possible layouts for further consideration during the planning process [WS14]. Kovács concludes in [Kov21]: “a uniform and standard procedure for the warehouse layout design is not available either in practice or in literature.”

In order to receive feasible layout alternatives, constraints have to be obeyed. The following list contains some of the more common constraints in the layout design [Kov21, NMWW18]:

- Architectural constraints (e.g., position of walls, gates, supporting pillars)
- Legal requirements (e.g., emergency exits and fire protection)
- Technical requirements
- Material flow requirements
- Financial requirements

A warehouse consists of different processes. The most important processes are receiving, put-away, storing, picking, sorting and packing. In a warehouse layout, functional areas are loosely connected to the aforementioned processes. Some processes imply their own functional area such as receiving, sorting and shipping. Others share a common functional area such as put-away, storing and picking.

Order picking is the process of retrieving items from a warehouse to satisfy customer orders. It is usually followed by consolidation and packaging. In most cases, these labor-intensive processes make up more than half of the total operating expenses of a warehouse [dKLDR07, GGN15]. In contrast to other warehouse functions, their level of automation is still relatively low. This is because it remains challenging to imitate the cognitive and motor skills of humans by machines in an economic manner [GGN15]. While automated solutions imply standardized layouts, this is not the case for manual order picking, making its layout planning a challenging task.

An order picking system’s layout consists of at least three basic elements. At the base, an empty collecting unit is handed to the picker. This can be a small load carrier, a pallet, a cart, etc. At the retrieval locations, the articles are stored in a manner that allows for efficient picking. Articles are usually stored in racks or on pallets, etc. [Man12]. When the picker completes an order line, the collecting unit is handed over to the next process step at a drop-off point. This may be the same location as the base, or a conveyor belt, etc.

3 Implementation

The implementation described in this paper serves as a general proof of concept. In the following, we will outline some key aspects of the implementation w.r.t. layout planning. The base of the implementation is a logic program $\mathcal{P}$ that contains the encoding of structural warehouse elements and the definition of dependencies and constraints regarding their positioning.

3.1 Overview

The presented implementation constitutes a proof of concept for a general framework to integrate answer set programming into the workflow of warehouse planning. An overview of such a framework is depicted in Figure 1. The process workflow starts with the user who can enter key instance values, e.g., the size of the warehouse, the number of racks inside the warehouse, and their size. This instance data is then added to the problem encoding which contains the general facts and conditions that hold for the

[3] https://www.python.org/

[4] https://www.lua.org/

intended warehouse layouts. The ASP solver computes the answer sets of the logic program where each answer set represents a preliminary layout based on the knowledge base modelled in the logic program. Subsequently, a Python application filters out those layouts where certain additional constraints are not met. The filtered layouts can then be rendered as 2D graphics and displayed to the user.

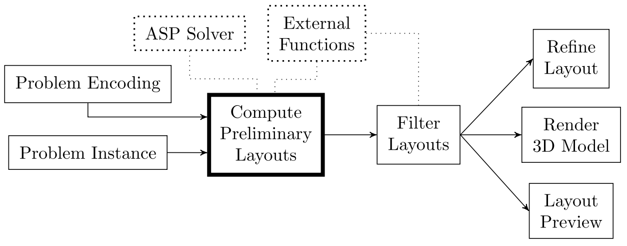


Figure 1. Framework Overview

3.2 Problem Instance and General Encoding

In ASP, logic programs are often divided into the *problem encoding* and the *problem instance* [GKKS12]. The problem encoding comprises those rules that describe the general problem that has to be solved whereas the problem instance contains those rules that establish specific parameters. The problem instance can, thereby, be viewed as the input data for the logic program that varies from case to case.

```
r1:  size_x(30). r2: size_y(20).
r3:  coords_blocked(1,1). r4: coords_blocked(1, 2).
r5:  racks_size(5). r6: racks_quantity(55).
r7:  handover(cell_line(coords(30, 10), coords(30, 13))).
```

The storage 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 accessible by its coordinates. This kind of representation allows a coarse-grained but flexible modeling of the available area. The size of the storage area can be set by literals *size_x* and *size_y*. This kind of representation offers high flexibility with respect to scalability. Each cell has one of three states:

blocked A *blocked* cell is not accessible by definition, e.g., due to an additional wall, a pillar or other building elements.

occupied A cell is *occupied* if a structural element is placed at this position, e.g., (part of) a rack.

idle A cell is *idle* if it is accessible but not occupied by a structural element.

Blocked cells can be determined by *coords_blocked*-literals as exemplified with rules r_3 and r_4. Other mandatory instance values are the rack size (r_5), the amount of racks that each layout has to contain (r_6) and the position of the handover (r_7).

In the following, with a *path* $C = \{c_1, \dots, c_n\}$, we mean a finite sequence of cells such that cells c_i , c_{i+1} ($1 \le i < n$) are adjacent. A path C is a *cell line* if all cells in C are located in either one row or one column of the grid. Two cell lines C_1, C_2 are *adjacent* if for every cell $c \in C_1$ there exists a cell $c' \in C_2$ such that c, c' are adjacent. The handover point inside the storage is also represented by a cell line as shown in r_7. The *cell_line*-literal has two arguments which represent both ends of the cell line.

```
r8:   distance_relevant(D) :- rack_size(D + 1).
r9:   cell_line_fixed_length(C1, C2, D, h)
      :- C1 = coords(C1X, C1Y), C2 = coords(C2X, C2Y),
      cell(C1), cell(C2), C1X == C2X, |C1Y − C2Y| == D,
      distance_relevant(D), C1 <= C2.
r10:  cell_line(C1, C2) :-
      cell_line_fixed_length(C1, C2,_,_).
r11:  cell_in_cell_line(@getCellOfCellLine(C1, C2),
      cell_line(C1, C2)) :- cell_line(C1, C2).
r12:  cell_line_blocked(C1, C2) :-
      cell_in_cell_line(C, cell_line(C1, C2)),
      cell_blocked(C).
```

Rules r_8-r_{10} define the cell line literals. Rules like r_8 serve to gather the sizes the cell lines are supposed to have, in this case, we only want to compute cell lines that have the same length as the racks. If we do not restrict the size of the considered cell lines, the program would compute all possible cell lines of all possible lengths (1 to *x* or *y* resp.) which in our case is not necessary. Therefore, in this example we restrict the relevant cell lines to those that have the length of a rack and the base. With rule r_9, all horizontal cell lines in the grid can be derived that have the relevant sizes[5] . With r_{10}, we then get a *cell_line*-literal for every possible cell line in the grid that have specified length(s).

Until now, it can only be derived whether single cells are blocked due to the initial instance data. Rule r_{12} shows how we can determine that a cell line is blocked. Intuitively, this rule determines that a cell line *CL* is blocked whenever a blocked cell *C* is part of *CL*. Here, we use the literal *cell_in_cell_line* to state which cells are contained in a given cell line. The *cell_in_cell_line*-literals are obtained by rule r_{11}, using an external Python function *@getCellInCellLine* that returns for a given cell line all contained cells by processing the coordinates of the cell line.

[5] We omitted the corresponding rule for vertical cell lines due to space restrictions.

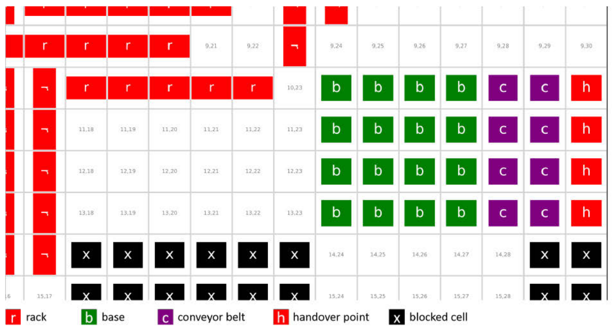


Figure 2. Base Location

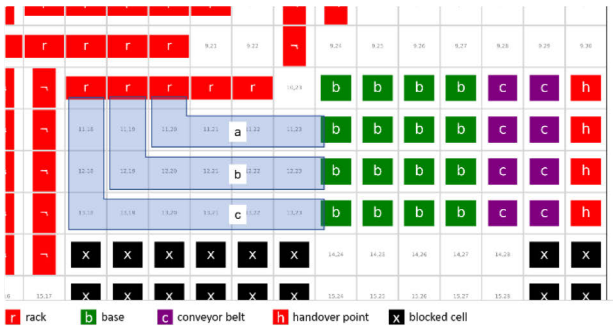


Figure 3. Paths of Width 1, 2 and 3 Cells

3.3 STRUCTURAL ELEMENTS

The implementation generates layouts where racks, a base and a conveyor belt are placed in the storage area. We focus on a limited number of conditions and constraints that have to be considered when positioning each structural element.

r13: R{rack_position(cell_line(C1, C2))
: cell_line_fixed_length(C1, C2, L, _),
rack_size(L + 1), cell(C1), cell(C2), C1 < C2}R
:- racks_quantity(R).
r14: :- rack_position(cell line(C1, C2)),
cell_line_blocked(C1, C2).
r15: :- rack_position(CL), rack_position(CL2),
cell_in_cell_line(C, CL), cell_in_cell_line(C, CL2),
CL! = CL2.

Racks The positioning of the racks are represented by *rack_position*-literals which have an argument *cell_line*. The *rack_position*-literals in an answer set, therefore, tell us where the racks are located in the corresponding layout. Rule r_{13} defines how the *rack_position*-literals are derived. Intuitively, with r_{13}, every answer set of $\mathcal{P}$ contains exactly R different *rack_position*-literals. With *cell_line_fixed_length* in $B(r_{13})$, we furthermore specify the positions a rack can have as we are only interested in cell lines that have the size of the racks. Rules r_{14} and r_{15} illustrate how the positioning of the racks can be refined by additional constraints. Rule r_{14} prevents that racks are positioned on top of blocked cells. By rules r_{15}, answer sets with overlapping racks are discarded. Here, the overlapping is defined as two racks that share a common cell. Note, that the order of the rules does not affect the answer sets and, therefore, especially constraints can be added as needed.

Base and Conveyor Belt The base in our implementation is a squared area of cells of size n (which can be predefined), i.e., a sequence **C** of adjacent cell lines C where each C and **C** itself have size n. The position of the base (see Figure 2) depends on the position of the handover (r_7), among other things. The position of the handover point is represented by a cell line located at a border of the warehouse. Every layout will contain a conveyor belt. To keep the implementation simple, we determine that the conveyor belt is also a sequence of adjacent cell lines and that the conveyor belt does not contain any curves.

3.4 REACHABILITY

In order to store and retrieve materials in a warehouse, each storage compartment needs to be accessible. A rack can be accessible from one side or from two sides, this depends on the type and position of the rack. This means, to get a valid layout from a logistics perspective, each rack has to be accessible from at least one side. In the following, with *reachability*, we will denote the property of a warehouse layout that there is a path from the base to each rack.

Finding paths in graph-like structures is itself a complex task [Dij59]. In layout generation based on a grid, the problem of reachability is even more complex than standard pathfinding as we also have to ensure that based on the scale of the grid, a path can have an individual *path width*.

Based on who (e.g., only persons, forklifts) is using the path, a different path width is necessary. The concept of path widths is illustrated by Figure 3 where paths with widths ranging from one to three cells can be constructed. Paths *a*, *b* and *c* constitute standard paths with the width of one cell. Paths with a width of two cells can then be formed by merging two “adjacent” paths. In this case the combination of either *a* and *b* or *b* and *c* are valid paths with a path width of two cells. Analogously, a path width of three cells can be achieved by combining all standard paths *a*, *b* and *c* since each path is adjacent to at least one other path.

It is easy to see that by taking different possible path widths into account, the complexity of computing paths in logistics settings increases. Furthermore, logic programs themselves do not directly support the utilization of heuristics or brute-force methods as they only allow for a declarative specification of constraints. However, since the answer sets of a logic program represent all possible solutions (those where reachability is satisfied and those where it is not), testing for reachability can be extracted from the actual layout generation process. We therefore propose that the logic program generates layouts without considering reachability and that the resulting layouts are then filtered by an additional external application which checks for reachability in each layout. The subsequent filtering of impractical layouts w.r.t. reachability by an external application has two advantages: extracting the reachability problem allows the usage of existing pathfinding algorithms

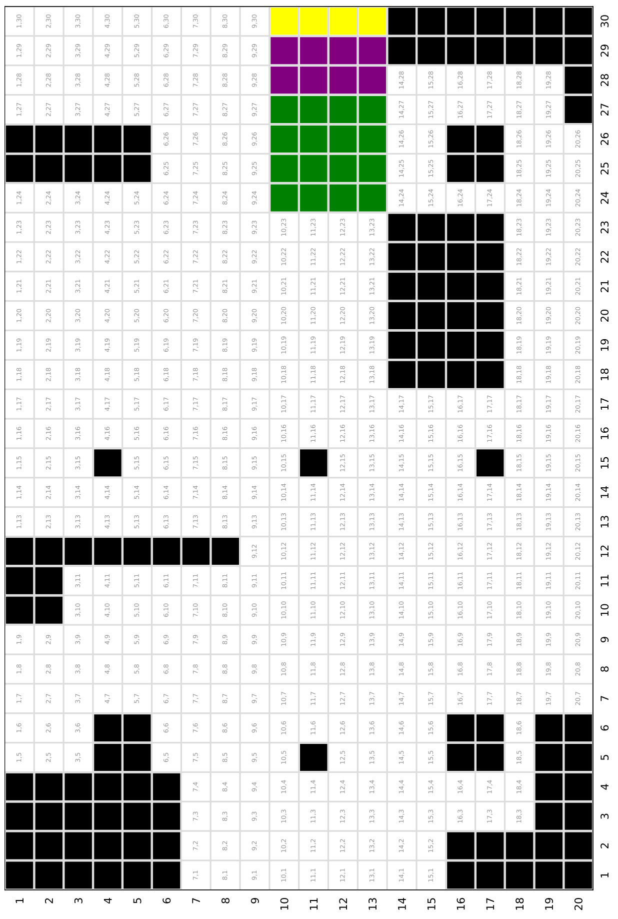
DOI: 10.2195/lj_Proc_kaiser_en_202112_01
URN: urn:nbn:de:0009-14-54219

(e.g., $A^\star$ search algorithm [HNR68]), and by precomputing all possible preliminary layouts by $\mathcal{P}$, we now have a decision problem instead of search problem, meaning, instead of computing all paths in a layout (as it would have been the case if reachability was considered in the logic program), we now only have to decide, whether a layout does or does not satisfy reachability. Thus, the delegation of the reachability computation to an external script is not more complex than computing layouts that consider such constraints [KST93]. Rather, by certain improvements to the used pathfinding algorithm, we claim that "normally" the external filtering is way more efficient.

4 Case Study

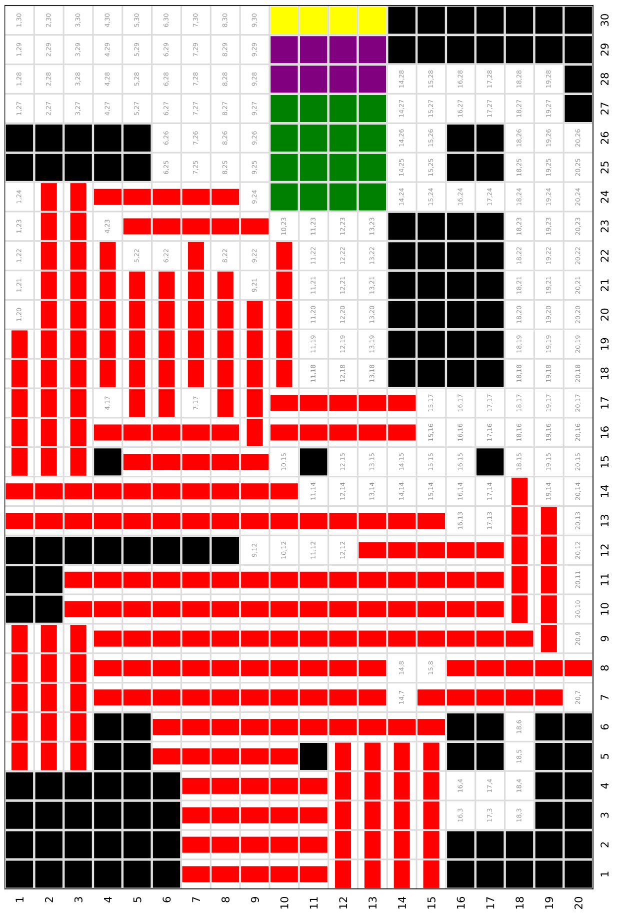
To illustrate the previous described implementation, a small case study was conducted. Therefore, the building restrictions of a warehouse are given and the task is to position the basic elements of a manual order picking system inside. The warehouse has a length of 30m and a width of 20m with several restricted areas (see Figure 4). These areas can be occupied by walls or pillars for example. As explained in Section 3.2, the logic program $\mathcal{P}$ consists of the problem instance and the problem encoding. The problem encoding consists of the basic knowledge of general planning rules, such as the storage area is a grid. In the problem instance, the use case specific information is encoded. In this case, the grid has 600 cells with 1x1m, where 114 cells are occupied. Additionally, the handover point to the next functional area is set (yellow squares). The base has to be connected to this handover point through the conveyor belt. The blocked cells are marked with a black square in Figure 4.

The goal is to position a base (4x4m), if necessary the conveyor belts and 55 racks (1x5m) inside the basic layout. In this case study, cells containing a rack consist of the rack itself and the path in front of the rack.



⬛ blocked cell 🟨 handover point 🟪 conveyor belt 🟩 base

Figure 4. Basic Layout Without Racks



⬛ blocked cell 🟨 handover point 🟪 conveyor belt 🟩 base 🟥 rack

Figure 5. 2D Layout With Racks

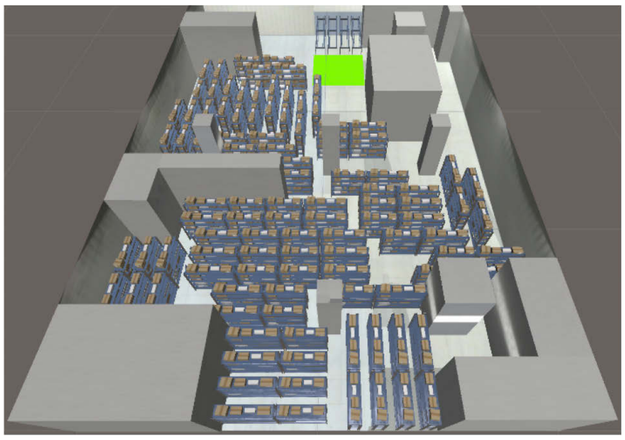


Figure 6. 3D Layout

The instance data has to be added as the problem instance to the general problem encoding in order to calculate all possible answer sets. In this case, the solver generated all 12096 different answer sets in under 35 seconds. This implies that in this use case, there are over 12000 possibilities to position the racks inside the layout. Note that since this paper aims at a proof of concept, the generated layouts comprise all feasible solutions and not exclusively optimal solutions.

After the solver calculated all answer sets, the implemented framework (see Figure 1) can be used to refine a layout, give a 2D layout preview and render a 3D model. For this case, one feasible layout preview is shown in Figure 5.

For a better visualization of the result, the 2D layout generated by the program can then be rendered as a 3D model (see Figure 6). With this 3D model the warehouse planner then can validate the generated layout by their expert knowledge.

5 Conclusion

This paper proposed an ASP-based approach to generate highly constrained warehouse layouts. As shown in the case study, an implementation with ASP is able to generate all layouts satisfying the constraints in an automated way. Using ASP systems like clingo also allow the additional incorporation of modern programming languages like python (e.g., see function *@getCellInCellLine* in Section 3.2).

Besides using the directly generated layouts, the answer sets can also be used to validate manually generated layouts by a layout planner or to add missing layouts. It should also help to expand the scope of a layout planner to innovative or unexpected layouts and not only create standard solutions.

The reason for these "new" solutions stems from the declarative paradigm of ASP. Hence, it suffices that the knowledge expert describes the desired solutions on an abstract level such as "the grid has to contain x racks of size y and must not overlap with other structures" by using rules and constraints. The solver will then calculate all answer sets satisfying the conditions given by the logic program. Conversely, in modern programming languages, the problem solving itself has to be defined.

Another consequence of using a declarative approach is that the generated solutions are independent of the actual order of the rules in a program. This simplifies the process of adding or changing constraints in a logic program. This also allows a step by step approach of implementing the constraints and focusing on rather difficult problems at the beginning.

In addition, the resulting answer sets of a logic program are in a format that allows further processing. This means, those answer sets can be, e.g., visualized or filtered. In the presented case study in Section 4, the answer sets were visualized in a first step as a 2D layout for a better comprehensibility and in a second step 3D layouts were generated from the answer sets.

We illustrated that the presented approach proposes important groundwork to generate highly constrained warehouse layouts, and moreover, AI-based approaches using ASP, therefore, present a promising and interesting foundation towards solving general highly constrained problems in practice.

6 Future Work

The implemented solution generates a large number of different layouts. As a next step, the evaluation of the generated layouts has to be implemented. This will be based on logistics indicators (e.g., picking performance), which are calculated for each answer set. The user then can decide based on which indicators the layouts should be optimized and determine how many layouts are necessary.

Furthermore, aggregating the generated layouts based on similarities into groups should help navigating through the answer sets. With these aggregated groups, representatives of each group can be compared at first before doing a deep dive into a single group. Therefore, such groups and their criteria have to be developed.

In addition, the generated solutions should be made explainable using AI-formalisms like *justifications* [ST16, PSEK09, CF16]. Such explanations can be visualized by comprehensible graphs that could help the layout planner to understand the positions of the different elements inside a layout and how they were derived from the rules in the logic program. Such explanations can then be used improve the refinement of programs to gradually obtain the intended layouts.

DOI: 10.2195/lj_Proc_kaiser_en_202112_01
URN: urn:nbn:de:0009-14-54219

LITERATURE

[AF07] Dieter Arnold and Kai Furmans. *Materialfluss in Logistiksystemen: mit 19 Tabellen*. Springer, Berlin, 5., erw. aufl edition, 2007.

[CF16] Pedro Cabalar and Jorge Fandinno. *Justifications for programs with disjunctive and causal-choice rules*. Theory Pract. Log. Program., 16(5-6):587–603, 2016.

[Dij59] Edsger W. Dijkstra. *A note on two problems in connexion with graphs*. Numerische Mathematik, 1:269–271, 1959.

[dKLDR07] René de Koster, Tho Le-Duc, and Kees Jan Roodbergen. *Design and control of warehouse order picking: A literature review*. European Journal of Operational Research, 182(2):481–501, October 2007.

[GGN15] Eric H. Grosse, Christoph H. Glock, and W. Patrick Neumann. *Human Factors in Order Picking System Design: A Content Analysis*. IFAC-PapersOnLine, 48(3):320–325, January 2015. Number: 3.

[GKKS12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning.* Morgan & Claypool Publishers, 2012.

[GL91] Michael Gelfond and Vladimir Lifschitz. *Classical negation in logic programs and disjunctive databases*. New Gener. Comput., 9(3/4):365–386, 1991.

[Gud10] Timm Gudehus. *Logistik: Grundlagen - Strategien - Anwendungen*. Springer, Berlin, 4., aktualisierte aufl edition, 2010.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. *A formal basis for the heuristic determination of minimum cost paths.* IEEE Trans. Syst. Sci. Cybern., *4(2):100–107, 1968.*

[Kov17] György Kovács. *Warehouse Design – Determination of the optimal storage structure.* page 5, 2017.

[Kov21] György, Kovács. *Special Optimization Process for Warehouse Layout Design.* In Károly Jármai and Katalin Voith, editors, Vehicle and Automotive Engineering 3, pages 194–205, Singapore, 2021. Springer Singapore. Series Title: Lecture Notes in Mechanical Engineering.

[KST93] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *Decision Problems, Search Problems, and Counting Problems*, pages 11–50. Birkhäuser Boston, Boston, MA, 1993.

[Man12] Riccardo Manzini, editor. *Warehousing in global supply chain: advanced models, tools and applications for storage systems*. Springer, London, 2012.

[NMWW18] Ruben Noortwyck, Timo Müller, Karl-Heinz Wehking, and Michael Weyrich. *Dezentrale assistierte Planung: Integrierte Layout- und Systemplanung von Intralogistiksystemen auf Grundlage einer agentenbasierten Software*. Logistics Journal: Proceedings, Vol. 2018.

[PSEK09] Enrico Pontelli, Tran Cao Son, and Omar El-Khatib. *Justifications for logic programs under answer set semantics.* Theory and Practice of Logic Programming, 9(1):1–56, 2009.

[Sch18] Michael Schmidt. *Distribution Center Design Process: ein systemtechnikorientiertes Vorgehensmodell zur Konzeptplanung von Logistikzentren.* Logistik für die Praxis. Verlag Praxiswissen, Dortmund, 2018. OCLC: 1031719835.

[SMM17] Timothy Sprock, Anike Murrenhoff, and Leon F. McGinnis. *A hierarchical approach to warehouse design.* International Journal of Production Research, 55(21):6331–6343, November 2017. Publisher: Taylor & Francis eprint: https://doi.org/10.1080/00207543.2016.1241447.

[ST16] Claudia Schulz and Francesca Toni. *Justifying answer sets using argumentation.* Theory and Practice of Logic Programming, 16(01):59–110, 2016.

[WS14] Alexandra Wunderle and Tobias Sommer. *Erfahrung und Augenmaß zählen.* Hebezeuge Fördermittel, (08), 2014.

DOI: 10.2195/lj_Proc_kaiser_en_202112_01
URN: urn:nbn:de:0009-14-54219

Pascal Kaiser, M.Sc., Research Assistant at the Chair of Materials Handling and Warehousing, TU Dortmund University.

Andre Thevapalan, M.Sc., 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.

Christopher Reining, M.Sc., Chief Scientist at the Chair of Materials Handling and Warehousing, TU Dortmund University.

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

Prof. Dr. Dr. h. c. Michael ten Hompel, Head of the Chair of Materials Handling and Warehousing, TU Dortmund University and Managing director of the Fraunhofer Institute for Material Flow and Logistics.

Address: Lehrstuhl für Förder- und Lagerwesen, TU Dortmund, Joseph-von-Fraunhofer-Str. 2-4, 44227 Dortmund, Germany
Phone: +49 231 755-2794, Fax: +49 231 755-4768,
E-Mail: pascal3.kaiser@tu-dortmund.de