FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Vít Kološ

Kitchen Layout Generation as an Optimization Problem

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: Mgr. Martin Koutecký, Ph.D.
Study programme: Computer Science

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I am deeply grateful to my supervisor Mgr. Martin Koutecký, Ph.D., whose invaluable insights greatly enriched this work. I would like to express my genuine gratitude to Ing. Jiří Hon, Ph.D. for offering me the opportunity to work on such a practical issue and facilitating the discussion with experts.

My sincere thanks goes to Bc. Kateřina Kratochvílová and Ing. Adam Kořený, Ph.D. for providing me with kitchen design expertise and testing the created web application. I gratefully acknowledge Salusoft89 s.r.o. for supporting my research.

To conclude, I would like to thank my family and friends for their unwavering support and encouragement throughout my academic journey.

Title: Kitchen Layout Generation as an Optimization Problem

Author: Vít Kološ

Institute: Computer Science Institute of Charles University

Supervisor: Mgr. Martin Koutecký, Ph.D., Computer Science Institute of Charles University

Abstract: When designing a kitchen for a client in a kitchen design studio, the designer proposes a layout of the cabinets, fixtures, and appliances that should be in the kitchen. To automate this step, we propose a new formal model of a kitchen layout and define the corresponding Kitchen Layout Problem. To solve this problem efficiently in practice, we create two implementations, one using mixed integer linear programming, and another using the constraint modeling language MiniZinc. The generated layouts are realistic and capture well the intent of our model. We compare the performance of several available solvers and modeling approaches. While some solvers take unrealistically long to finish, the commercial solver Gurobi and the open-source solver CP-SAT both finish reliably within tens of seconds, with CP-SAT usually being faster.

Keywords: kitchen, layout, design, optimization, integer programming, constraint programming

Název práce: Generování rozvržení kuchyně jako optimalizační problém

Autor: Vít Kološ

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: Mgr. Martin Koutecký, Ph.D., Informatický ústav Univerzity Karlovy

Abstrakt: Při plánování kuchyně v kuchyňském studiu designér navrhuje rozmístění skříněk a spotřebičů v místnosti. Naším cílem je tento krok automatizovat. Nejprve jsme zavedli nový formální model rozvržení kuchyně a na jeho základě definovali výpočetní problém (Kitchen Layout Problem). Pro efektivní řešení tohoto problému v praxi jsme vytvořili dvě implementace: první pomocí smíšeného celočíselného lineárního programování, druhou za použití modelovacího jazyka MiniZinc. Generovaná rozvržení jsou realistická a dobře zachycují záměr našeho modelu. Dále jsme srovnali výkon několika dostupných řešičů a přístupů k modelování. Zatímco některým řešičům trvá nalezení řešení nepřiměřeně dlouho, komerční řešič Gurobi a svobodný řešič CP-SAT problém spolehlivě vyřeší během desítek sekund, CP-SAT obvykle rychleji než Gurobi.

Klíčová slova: kuchyň, rozvržení, návrh, optimalizace, celočíselné programování, omezující podmínky

Contents

Introduction **8**
Our Contribution 8
Related Work 9
Organization 10

1 Preliminaries **11**
1.1 Kitchen Layout 11
1.1.1 Types of Fixtures 11
1.1.2 Kitchen Shapes 11
1.1.3 Kitchen Zones 11
1.1.4 Dimensions of Fixtures 12
1.1.5 Fixture Preferences 12
1.2 The Manual Approach 13
1.3 Mixed Integer Linear Programming 13
1.3.1 Modeling Techniques 13
1.4 Constraint Programming 15
1.4.1 MiniZinc 15

2 Analysis **17**
2.1 Program Requirements 17
2.2 Information Used by the Designer 17
2.2.1 Example of Client's Requirements 17
2.3 Loading the Information into the Program 18
2.4 Layout Rules and Requirements 19
2.5 Suitable Model 20

3 Model Creation **21**
3.1 Model of the Kitchen Layout 21
3.1.1 Sets and Mappings 21
3.1.2 Constraints 23
3.1.3 Objective Function 24
3.1.4 Kitchen Layout Problem 24
3.2 Model Implementations 25
3.2.1 Segments 25
3.2.2 Decision Variables 26
3.2.3 MiniZinc Model 27
3.2.4 Kitchen Layout Reconstruction 27
3.3 Implemented Constraints 27
3.3.1 Contiguous Fitting 28
3.3.2 Present Fixture 29
3.3.3 Widths, Vertical Placement 29
3.3.4 Tall Fixtures 29
3.3.5 Side Panels 31
3.3.6 Placement Rules 32
3.3.7 Sorting the Duplicates 33

3.4 Objective Function in MILP and MiniZinc 34

4 Design Rules **36**
4.1 Implementation 36
4.1.1 Grouping 36
4.1.2 Alignment / Vertical Continuity 37
4.1.3 Fixtures with the Same Widths 41
4.2 Remaining Rules 42
4.2.1 Relation Rules 42
4.2.2 Standard Cabinet Widths 42
4.2.3 Alternating Widths 43
4.2.4 Group by Zone 43
4.2.5 Worktop Size, Storage Space 43
4.2.6 Corners 44

5 Deployment **45**
5.1 Coding the Models 45
5.1.1 MILP 45
5.1.2 MiniZinc 45
5.2 Command-Line Tool 46
5.2.1 Running an Experiment 46
5.2.2 Evaluating a Solution 47
5.2.3 Debugging an MILP Model 48
5.2.4 File Formats 48
5.3 Web Tool 48
5.3.1 Architecture 49

6 Evaluation **50**
6.1 Quality of the Layouts 50
6.1.1 Reading Layout Diagrams 50
6.1.2 First Layout 50
6.1.3 Second Layout 51
6.1.4 Third Layout 51
6.1.5 Discussion of Layout Quality 53
6.1.6 Effect of Individual Design Rules 53
6.1.7 Multiple Optimal Solutions 53
6.2 Reduction of the Solving Time 54
6.2.1 Comparing Models and Solvers 54
6.2.2 Deactivating Rules 57
6.2.3 Alignment: Binary vs. Piecewise Approach 57
6.2.4 Discussion of the Performance 59
6.3 Review of the Web Application 60

Conclusion **61**
Future Work 61

Bibliography **63**

List of Figures **66**

A Attachments **67**
A.1 Electronic Attachment 67
A.2 User Documentation 67
A.2.1 Web Application 67
A.2.2 Command-Line Interface 70
A.2.3 File Formats 71

Introduction

A client wanting a fitted kitchen often visits a kitchen design studio. The process in the studio usually consists of several meetings. During the first, the designer tries to elicit as much of the preferences of the client as possible. Then, the designer manually sketches the placement of individual *fixtures* (cabinets, appliances, etc.) in the kitchen – creates a *layout sketch* (see Figure 1). However, as it can take more than an hour to prepare the first sketch, the client usually does not wait in the studio, but comes back another day to review the layout and discuss possible improvements.

This process may be repeated multiple times until the customer's requirements are satisfied. Eliciting precise client requirements can be challenging; it is the job of the designer to find out. It might be necessary to prepare several different layouts to see what the client prefers. If we automate the creation of the layout sketch, it will speed up the process and benefit both the studio and the client: The studio can serve more customers and reduce the cost of the layout creation. The customer spends less time in the studio or traveling to the studio and may be more satisfied with the process as a whole – therefore less likely to change their mind and go to the competition.

Our Contribution

We have automated layout sketch generation using two different approaches – mixed integer linear programming (MILP) and constraint programming (CP). We have formulated a mathematical model and described the main design principles as an optimization problem, i.e., a set of constraints and an objective function. We have compared two modeling approaches, an MILP model and a MiniZinc [1] model, and assessed their suitability for the task.

Our main takeaways are as follows:

- It is possible to express the main design rules and guidelines using the two modeling approaches. The generated layouts follow the rules and are satisfactory from the design perspective.

- We need to carefully decide which rules to include in the model and how to express them mathematically. Otherwise, the solving time may exceed practical time limits.

- The two formulations (MILP, MiniZinc) achieve very similar results regarding the solutions and solving times when solved by MILP solvers.

- Surprisingly, the open-source solver CP-SAT [2] outperforms the commercial solver Gurobi [3] when compared on our test data.

The topic of this thesis was proposed by Salusoft89, a company supplying software for furniture manufacturing. Its experts provide most of the domain knowledge as for the kitchen design and layout creation process and make sure that the constructed models respect the requirements of designers and clients. Equally important are the contributions of Kateřina Kratochvílová, who has

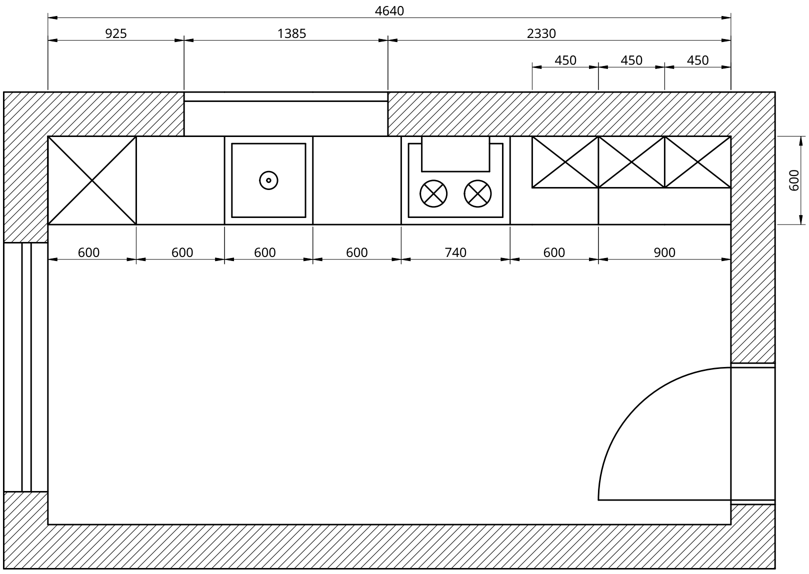


Figure 1 Kitchen layout sketch. Such a sketch should contain all the main kitchen fixtures (a sink, a stove, a fridge, some cabinets, etc.), their positions and dimensions. Usually, it does not specify the insides of the cabinets – this is decided later in the design process.

practical experience with manual kitchen design, provides feedback on the user interface of the automation program and evaluates the quality of the solutions. This assessment guides our mathematical formulation of the objective function and related constraints.

Related Work

Prior research in the area of kitchen design has experimented with various approaches. They are often based on a database of kitchen layouts.

Ervum [4] uses case-based reasoning; the program holds a database of layouts designed in the past and returns its subset based on the input data. It selects the proposed layouts according to their applicability to the current situation. Unlike Ervum, we generate the layout from scratch, so we are not limited to existing layouts in a database.

Pejic and Pejic [5] apply machine learning methods (random forests) to generate in-line kitchen layouts. They treat the kitchen layout as a single row of fixtures; every upper fixture is linked with a lower fixture. In contrast, our approach can generate various shapes of layouts (L-shaped, galley, etc.), and the placement of upper fixtures is not that rigidly dependent on the lower fixtures. As illustrated in Figure 1, it may be useful to structure upper fixtures differently than the lower ones. Moreover, Pejic and Pejic need a set of training data to train the machine learning model, while we do not rely on a comprehensive dataset.

Sun and Ji [6] optimize kitchen layouts using an improved particle swarm optimization algorithm. Their approach does not generate new layouts but chooses the best among the provided designs (or “preliminary design schemes”). However, we solve the problem as a whole: We generate the possible layouts and choose the optimal one. Again, their model of kitchen layout is simpler than ours as they do not treat lower and upper fixtures separately.

Hsieh et al. [7] approach the automation of kitchen layout design as a game of two players: a computer and a designer. They use deep reinforcement learning to ensure that the program can cooperate with the designer on creating the layout. Their model of kitchen is very simple as their kitchen plan is a 15×15 matrix and the upper fixtures are inseparable from the lower fixtures. To approximate the layout in Figure 1, the area would be probably divided into squares 15 cm wide to achieve a similar size of the matrix. We present a more general approach: the MILP model treats variables as continuous, and the MiniZinc integer model uses centimeters in the default configuration.

Previously, Hsieh [8] also automated layout creation using an evolutionary algorithm: there was no matrix, but all fixtures had fixed widths. Again, the upper fixtures could not be separated from the lower fixtures. We allow custom-made fixtures with variable widths. To a certain extent, we position the lower, upper, and tall fixtures independently.

Lee et al. [9] propose a 0–1 integer programming model to optimize a kitchen layout. This framework is related to the MILP approach we use, but their representation of fixtures is very abstract and simplified. Similarly to Hsieh et al. [7], they position the fixtures on a two-dimensional grid of squares 10×10 cm. They optimize the ergonomics of the layout by minimizing the movement distance required to perform certain tasks. This may, however, lead to layouts where some fixtures block access to other fixtures. We focus on generating layouts that can be implemented in practice. While we try to create ergonomic layouts, we also consider other criteria like visual attractiveness or production cost.

There exist online applications – often provided by kitchen design studios and furniture retailers [10] to support their sales – that allow a user to generate a kitchen layout. Experts on kitchen design argue that the solutions proposed by such tools seem to be generated using templates, so there is probably no optimization performed, and the layouts are lacking variety. The user cannot influence the generation process to express their preferences. Our goal is to allow the designer to have a direct influence on the generated layouts by assigning different priorities to various aspects of the design.

Organization

In Chapter 1, we cover the necessary notions from the area of kitchen design, mixed integer linear programming, and constraint programming. In Chapter 2, we give an overview and analysis of relevant information obtained from domain experts. Based on this information, we then, in Chapter 3, propose a formal model, dubbed the Kitchen Layout Problem, and describe how it can be modeled as an MILP and in MiniZinc, respectively. In Chapter 4, we describe and discuss the implementation of more advanced layout constraints. Chapter 5 describes the details of the deployment. Finally, in Chapter 6, we evaluate the results, both from the perspective of quality and performance. We close with a list of possible directions for Future Work.

1 Preliminaries

To be able to create a suitable model of the problem, we need to first analyze what a kitchen usually looks like and how a designer works when sketching the layout manually. Also, it is necessary to understand the theory related to each of the modeling approaches.

1.1 Kitchen Layout

In the initial phase of kitchen design, the designer prepares a kitchen layout: Their main goal is to position cabinets, fixtures, and appliances in the kitchen. For simplicity, we use the word *fixtures* as an umbrella term.

1.1.1 Types of Fixtures

Generally, there are three types of fixtures: *lower*, *upper*, and *tall*. Both lower and tall fixtures stand on the floor. On top of the lower fixtures, there is usually a worktop. The upper fixtures hang on the wall above the lower fixtures. The depth of the tall and lower fixtures is usually the same; upper fixtures have a smaller depth to allow better access to the worktop. All three types are illustrated in Figure 1.1.

1.1.2 Kitchen Shapes

The fixtures are usually placed along the walls without any spaces between them to maximize the size of the continuous worktop. There are several typical shapes that the plan view of the kitchen layout can have according to the walls used. In the Architect's Pocket Book of Kitchen Design [12], Baden-Powell lists the following four shapes:

- In-line: The fixtures are situated along a single wall.
- Two-sided or galley: There are two opposite walls – two opposite rows of fixtures.
- L-shaped: The fixtures are placed along two perpendicular walls sharing a corner.
- U-shaped: There are three walls with fixtures, two corners.

Sometimes, a group of lower fixtures can even be positioned freely in the form of a kitchen island, or it can be extending perpendicularly from a block of fixtures along a wall.

1.1.3 Kitchen Zones

Kitchen fixtures can be divided by their purpose into several *kitchen zones*. The zones correspond to the activities that the kitchen is used for: cooking, washing, storage, preparation. Some of these zones may contain only one or two

Figure 1.1 Types of fixtures. The leftmost and rightmost fixtures are tall. In the middle, we can see three upper fixtures. There are also five lower fixtures – one of them contains a sink. Image by Vecislavas Popa [11].

fixtures, so it is not unusual to use the names of the fixtures directly instead (stove, sink, refrigerator).

The three main kitchen zones – cooking, washing, and storage – form *the work triangle*. According to Baden-Powell [12], the total length of the three sides of this triangle should not be less than 3.5 m or more than 6.5 m long.

1.1.4 Dimensions of Fixtures

While positioning fixtures in the kitchen, the designer must consider their dimensions. Both heights and depths are usually fixed for each of the three main types of fixtures, though the designer sets the heights to accommodate the client's body height. However, different fixtures often have different widths. Although most appliances have a specific width, it is possible to include some custom-made cabinets in the layout.

1.1.5 Fixture Preferences

The designer commonly asks the client about their preferences regarding the fixtures and appliances. For example, some clients prefer an oven that is directly below the stove top, while others want their oven to be beneath the microwave as a part of a tall cabinet. Other clients ask for an American-style fridge.

However, there are some characteristics of the fixtures that we do not need to know in order to propose a kitchen layout. For instance, it does not matter whether a cabinet contains drawers or shelves. We can put a general cabinet in the layout – the decision on its contents can be made in the next phase of the kitchen design.

1.2 The Manual Approach

When a designer prepares the initial draft of the layout, they start by drawing the dimensions of the room and the shape of the kitchen. Then, they proceed from the most constrained fixtures, the ones whose position or width is determined by the circumstances. That is usually the case of corner cabinets (the corner does not move), the sink and the dishwasher (the plumbing is often fixed – even though the designer may decide to move it), the stove (due to the ventilation), and other appliances with a fixed width.

Afterwards, the remaining space is to be filled with cupboards and cabinets. There are several rules that the designer follows in order to create a kitchen that is usable and safe, e.g.: The worktop should not be too small. There needs to be enough space around the sink and stove to access them and to lay things aside while using them.

There are also some rules revolving around the visual attractiveness of the layout and the production costs: The sides of the upper and lower fixtures should be aligned. The cabinets next to each other should have the same widths. There should be as few cabinets with atypical widths as possible.

The designer does not have much space for creativity in this phase of the whole design process, since it is often quite strictly determined by the requirements of the client, the shape of the room, and the design rules.

1.3 Mixed Integer Linear Programming

We will be using the optimization framework of linear programming and its variants. We follow the definitions and notation of Matoušek and Gärtner [13] and invite the reader to see this textbook for more background if desired. A **linear programming** (LP) problem consists of a set of linear constraints and a linear objective function. The set of constraints can be expressed as $A\boldsymbol{x} \leq \boldsymbol{b}$, the objective function is $\boldsymbol{c}^T\boldsymbol{x}$, where $A \in \mathbb{R}^{m \times n}$ is a given matrix and $\boldsymbol{b} \in \mathbb{R}^m$, $\boldsymbol{c} \in \mathbb{R}^n$ are given vectors. The aim is to maximize the value of the objective among all vectors $\boldsymbol{x} \in \mathbb{R}^n$ satisfying the constraints. The relation $\leq$ needs to hold componentwise.

Any vector $\boldsymbol{x} \in \mathbb{R}^n$ satisfying the constraints is a *feasible solution.* If such vector $\boldsymbol{x}^*$ gives the maximum possible value of $\boldsymbol{c}^T\boldsymbol{x}$ among all feasible $\boldsymbol{x}$, it is an *optimal solution.*

In an **integer linear programming** (ILP) problem, the components of $\boldsymbol{x}$ are restricted to be integers. If there are both real and integer components, it is an instance of **mixed integer linear programming** (MILP) problem.

Even though there exists a polynomial algorithm [14] for linear programming, integer linear programming is NP-complete [15]. However, there exist both open-source [16] and commercial [3] solvers that can often solve ILP and MILP problems in reasonable time.

1.3.1 Modeling Techniques

While the expression abilities of the MILP paradigm seem quite limited, there are some widely used techniques that allow us to formulate unhoped-for concepts.

We describe several of them, see the AIMMS Optimization Modeling [17] book for more.

Indicator Constraints

It may be useful to be able to disable a given linear constraint based on the value of a binary variable, i.e., a variable than can be either equal to zero or one. Even though mixed integer programs do not support if-then-else expressions, we can use *indicator* or *big-M* constraints to achieve that.

For example, say we have two decision variables: a binary variable $b \in \{0, 1\}$ and a real variable $x \in [0, U]$, where U is a constant. We would like to restrict the variable x by the inequality $x \leq c$, where c is a non-negative constant or another non-negative variable. Importantly, we want the inequality to be enforced if and only if b equals zero.

This can be achieved using the constraint

$$x \leq c + M \cdot b \tag{1.1}$$

where M is a large constant. Here, we can use $M = U$. Note that we must use such a constant M that $x - c \leq M$ always holds.

Also, if we want to ensure that exactly one of two given constraints holds, we can add $M \cdot b$ to the first constraint and $M \cdot (1 - b)$ to the second one.

Logical Statements

When using binary variables, it is often necessary to express logical connectives. To give an example, we consider binary variables a, b, c, d, all with values from the set $\{0, 1\}$, where 1 corresponds to *true* and 0 to *false*.

An implication $a \to b$ is modeled as $a \leq b$. A negation $\neg a$ can be expressed as $1 - a$.

There are multiple ways to model a conjunction $c = a \wedge b$. We show the one stated by Amer [18], which uses the following two inequalities:

$$0 \leq a + b - 2c \leq 1. \tag{1.2}$$

Note that we can express even conjunctions containing more variables such as $d = a \wedge b \wedge c$ in a similar fashion:

$$0 \leq a + b + c - 3d \leq 2. \tag{1.3}$$

“Less Than” Condition

Let x, y be real variables such that $x \in [0, U],\ y \in [0, U]$ where U is a constant. Suppose we want a binary variable $b \in \{0, 1\}$ to be equal to one if $x \leq y$, otherwise it should equal zero. This can be achieved using the constraint

$$0 \leq x - y + M \cdot b \leq M. \tag{1.4}$$

We use $M = U$. Note that if x equals y, the variable b can be equal to either one or zero as the inequalities $0 \leq 0 + M \cdot b \leq M$ hold in both cases.

Auxiliary Variables

In mixed integer programs, some non-linear relationships can be expressed by adding new auxiliary variables.

To illustrate, let us consider two decision variables: a binary variable $b \in \{0, 1\}$ and a real variable $x \in [0, U]$, where U is a constant. We want to include the value of x in the objective function if and only if the value of b is equal to one. However, we cannot put $b \cdot x$ in the function as that would be a non-linear component.

To solve that, we introduce an auxiliary variable $a \in [0, U]$ and three constraints

$$a \leq M \cdot b, \tag{1.5}$$

$$a \leq x, \tag{1.6}$$

$$x \leq a + M \cdot (1 - b), \tag{1.7}$$

where $M = U$.

1.4 Constraint Programming

Another useful optimization paradigm is constraint programming (CP). We proceed from the definitions presented by Apt [19]. An instance of the **constraint satisfaction problem** (CSP) consists of a finite set of variables $X = \{x_1, \ldots, x_n\}$ with respective domains $D_1, \ldots, D_n$ and a finite set $\mathcal{C}$ of constraints, each on a subset of X. By a constraint C on a subset $\{x_{i_1}, \ldots, x_{i_k}\}$ we mean a subset of $D_{i_1} \times \cdots \times D_{i_k}$.

We say that an n-tuple $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ satisfies a constraint $C \in \mathcal{C}$ on the variables $x_{i_1}, \ldots, x_{i_m}$ if $(d_{i_1}, \ldots, d_{i_m}) \in C$. If an n-tuple $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ satisfies every constraint $C \in \mathcal{C}$, we say that it is a *solution* to the instance of CSP.

Often, it is useful to express that some solutions are more convenient than others. An instance of the **constrained optimization problem** (COP) is composed of an instance of CSP together with an objective function $\mathrm{obj} : \mathrm{Sol} \to \mathbb{R}$, where Sol is the set of all solutions to the instance of CSP – the *solution space*. Note that the objective function is usually defined for all n-tuples $d \in D_1 \times \cdots \times D_n$. An *optimal solution* is a solution $X^* \in \mathrm{Sol}$ such that for every solution $X \in \mathrm{Sol}$, it holds $\mathrm{obj}(X) \leq \mathrm{obj}(X^*)$.

Clearly, LP, ILP and MILP problems are special cases of constrained optimization problems: If we consider the problem mentioned in Section 1.3, there is a set of variables $\boldsymbol{x} = \{x_1, \ldots, x_n\}$, a set of constraints $\mathcal{C} = \{\sum_{j=1}^{n} a_{ij} x_j \leq b_i \mid i \in \{1, \ldots, m\}\}$ and an objective function $\mathrm{obj}(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x}$.

1.4.1 MiniZinc

MiniZinc [1] is a high-level constraint modeling language. It can be used to create a CSP model and find its solution using one of the supported solvers. Before solving, the MiniZinc model needs to be translated to a format that can be processed by the specific solver. This is done automatically by the MiniZinc compiler.

There are multiple approaches to solving CSPs, e.g., constraint propagation [20], lazy clause generation [21], or mixed integer linear programming (MILP). The compiler ensures that the compiled model is optimized for the specific approach the solver uses. This may lead to a different definition of variables or to using built-in global constraints that the solver supports (e.g., `alldifferent`). For instance, a variable that can contain one of k possible values (such variables are often called *categorical*) can be compiled as k binary variables and a constraint ensuring that the sum of the variables is equal to one. Refer to the paper by Belov et al. [22] for more examples.

2 Analysis

Our main goal is to automate the manual process described in Section 1.2. First, we state the requirements for the automation program. Second, we discuss the kind of information used by the designer during this process. Then we list the rules that the designer usually follows so that the model could accommodate them. Finally, we propose a model that fulfills the requirements.

This chapter is based on discussions with experts on kitchen design and presents their demands and opinions. We state an exact formulation of the resulting Kitchen Layout Problem in the following chapter.

2.1 Program Requirements

A program that would automate the creation of kitchen layout should have the following properties:

1. It should allow the designer to quickly enter all the important data; it should be easy to use.
2. It should return solutions that follow the design rules and present them in a comprehensible way.
3. The solutions do not have to be perfect, but should be returned reasonably quickly. We estimate that the designer would be willing to wait at most one or two minutes, depending on the size of the kitchen.

2.2 Information Used by the Designer

The designer usually bases the manually created kitchen layout on the following information:

- dimensions of the room, windows, walking space,
- current position of the plumbing and ventilation,
- desired placement of tall cabinets and other special requirements,
- preferred shape of the kitchen (in-line, two-sided, L-shape, U-shape, etc.),
- a set of fixtures and appliances that the client wants to fit in the kitchen.

2.2.1 Example of Client's Requirements

To illustrate, we present some of the requirements of a real client that Kratochvílová [23] lists in her case study. This client has an old kitchen that needs to be renovated. Along with the floor plan indicating the exact dimensions of the room and the plumbing position, the client provides the following demands regarding the space planning and fixture placement:

- There should be more (natural) light in the kitchen.

- The kitchen table should accommodate at least 4 people, 6 if possible.
- There must be enough storage space for appliances.
- The worktop should be larger than before.
- Two people use the kitchen regularly: an older woman and an older man. The woman uses the kitchen more frequently, she is 160 cm tall.
- These are the appliances that the kitchen layout should ideally accommodate: gas stove top, cooker extractor hood, hot air oven, microwave oven, combined refrigerator, dishwasher, multifunction pot, hand mixer, toaster, juicer, coffee machine, kettle, rice cooker.
- The worktop should be 900 mm high.

The designer then drafted two layout sketches. Both proposals are L-shaped, both require the plumbing to be relocated.

2.3 Loading the Information into the Program

When we automate the process of designing the kitchen layout, we need to take most of the mentioned information into account. We leave the communication with the customer up to the designer and let the designer then pass the necessary information to our program.

Not only would it be complicated for the designer to capture all the particularities of the room in a single input file, but it would be even harder for our program to process it and propose an appropriate kitchen layout. Therefore, we let the designer determine the general areas where the fixtures should be placed.

As we mentioned in Subsection 1.1.2, we can assume that the fixtures will not be standing freely in the space and that each fixture belongs to a larger unit, *"kitchen part"*, usually situated along a wall. A kitchen part is simply a row of fixtures, so if there is a row of upper fixtures and a row of lower fixtures both along the same wall, these are two different kitchen parts.

The designer defines the location and size of the kitchen parts – the areas to place the fixtures. This allows them to describe the room quite precisely and to express their or the client's expectations about the shape of the kitchen. Although it limits the set of possible layouts that the program can produce for given input data, the designer is able to try various settings of the kitchen parts if the solutions can be generated fast enough.

Along with descriptions of the kitchen parts, the designer needs to pass some other information to the program. We can use a term *kitchen scheme* to refer to the set of information related to the kitchen layout – including the kitchen parts.

The kitchen scheme has to contain a list of fixtures that can be placed in the layout. This list may be based on a standard set of fixtures that every kitchen contains and then adjusted by the designer to follow the client's requirements. All the fixtures are optional by default, but the designer can mark fixtures that must be included in the generated layout.

2.4 Layout Rules and Requirements

Our model needs to support the rules followed by the designer (see Section 1.2); the program has to evaluate the results based on the extent to which they comply with the rules. Because of that, we compile a list of possible rules and sort them by their importance in descending order:

- We define the kitchen parts; they should be filled with fixtures as much as possible. There must not be any gaps between fixtures.
- It is cheaper to fill the kitchen with a few wide fixtures than with many narrow fixtures.
- We can specify a place where a fixture must or must not be.
- Fixtures with visible sides have neat side panels.
- Tall fixtures are grouped.
- Fixtures that can support a worktop are grouped.
- The ends of the lower and upper fixtures are aligned if possible.
- Adjacent fixtures of the same widths look good.
- We can demand various relations between the fixtures and their surroundings.
- Cabinets of standard widths are preferred over custom-made cabinets because custom-made cabinets are more expensive.
- Alternating fixture widths are desirable (if the same widths cannot be achieved).
- Fixtures of the same kitchen zone are close together; the perimeter of the work triangle is between 3.5 m and 6.5 m.
- The larger the worktop area, the better.
- The more the storage space, the better.
- Corner fixtures need special treatment.

We can see that some of the rules are closely linked. It is possible that if we implement constraints related to one rule, the solutions will also fulfill some other rules. Our goal is to construct a model that is simple and transparent enough but also produces solutions that meet most of the requirements.

2.5 Suitable Model

We have described in Section 2.3 that the designer defines several kitchen parts where the fixtures should be placed. Neither the depth nor the exact height of the fixtures is subject to calculation. Then the model of a kitchen with fitted fixtures can be expressed as assignments of individual fixtures to their kitchen parts in a specific order. The widths of some fixtures may be variable, so we need to include that in the model, too.

As was mentioned in Section 2.4, there are both strict rules to be followed and vague criteria (visual attractiveness, production cost, usability) which we can use to differentiate between a better layout and a worse one. If we perceive the rules and criteria as a basis for constraints and an objective function, it becomes clear that the kitchen layout is a *constrained optimization problem*.

Even for such a simplified model, there are too many candidate solutions for brute-force enumeration. If we consider a kitchen where the input set of fixtures contains 3 tall, 8 lower, and 6 upper fixtures, there are $3! \cdot 8! \cdot 6! = 174{,}182{,}400$ options to arrange them in their kitchen parts if each type has its own.

This does not take into account the variable widths of custom-made fixtures and the fact that the input set should contain more fixtures than we expect in the result so that the solution space is not too limited and there is some room for optimization. Also, it does not suffice to just generate the arrangements; we need to test if they satisfy the constraints and find the best solution according to the objective function.

Clearly, we need a smart approach to search in the solution space. We will use the methods of mixed integer linear programming (Section 1.3) and constraint programming (Section 1.4).

3 Model Creation

We propose a formal mathematical model – the Kitchen Layout Problem. Then, we create two independent implementations, a pure MILP model and a MiniZinc model, so that we can find out which approach is better. For each model, we specify the input data, decision variables, constraints, and the objective function.

When proposing the model, we ensure that it may support most of the rules presented in Section 2.4. However, in this chapter, we focus on the rules that can be defined exactly and leave most of the looser "guidelines" for the next one.

3.1 Model of the Kitchen Layout

3.1.1 Sets and Mappings

Fixtures

To begin, we define a set of fixtures $F = \{f_1, \ldots, f_n\}$. There is a partition $F = L_{\text{fix}} \mathbin{\dot\cup} U_{\text{fix}}$ into lower and upper fixtures. A set $T \subseteq F$ contains tall fixtures, each has a lower and upper half encoded as $\mathcal{T} \subseteq \binom{T}{2}$ such that $\forall t, t' \in \mathcal{T} : t \cap t' = \emptyset$.

We introduce mappings $\text{wmin}, \text{wmax} : F \to (0, \infty)$ encoding the minimum and maximum widths, respectively, satisfying $\text{wmin}(f) \leq \text{wmax}(f)$; we define $\text{wmin}(\emptyset) = 0$, $\text{wmax}(\emptyset) = \infty$. Let there be mappings $\text{pleft}, \text{pright}, \text{worktop} : F \to \{0, 1\}$ indicating if the fixtures have a left panel, right panel, or worktop, respectively.

For the two types of fixtures, we will present two types of kitchen parts (lower and upper). There exist alternative ways to model tall fixtures: One of them is to designate tall kitchen parts as a third type, but it requires to fix the boundary between tall and lower/upper fixtures. Another way is to divide every kitchen part into three subparts – the two marginal subparts would be designated for tall fixtures.

Kitchen Parts and Groups

Let $P = \{p_1, \ldots, p_k\}$ denote the set of kitchen parts. There is a partition $P = L_{\text{part}} \mathbin{\dot\cup} U_{\text{part}}$ corresponding to lower and upper kitchen parts. We define mappings $\text{width} : P \to (0, \infty)$, stating the width of the kitchen part, and $\text{vleft}, \text{vright} : P \to \{0, 1\}$, indicating the visibility of its sides.

To be able to describe vertical relations between kitchen parts located along the same wall, we group them into *kitchen groups* denoted by $G = \{g_1, \ldots, g_m\}$. Let a mapping $\text{group} : P \to G$ define a partition of parts into groups. The mapping $\text{offset} : P \to (0, \infty)$ corresponds to the number of units by which the kitchen part is shifted from the beginning (left side) of the group.

Note that a group usually contains one upper part and one lower part, but this may not always be the case. For example, if there is a window in the given wall, the area for upper fixtures is separated into two kitchen parts, and such a group contains a total of three kitchen parts. See Figure 3.1 for illustration.

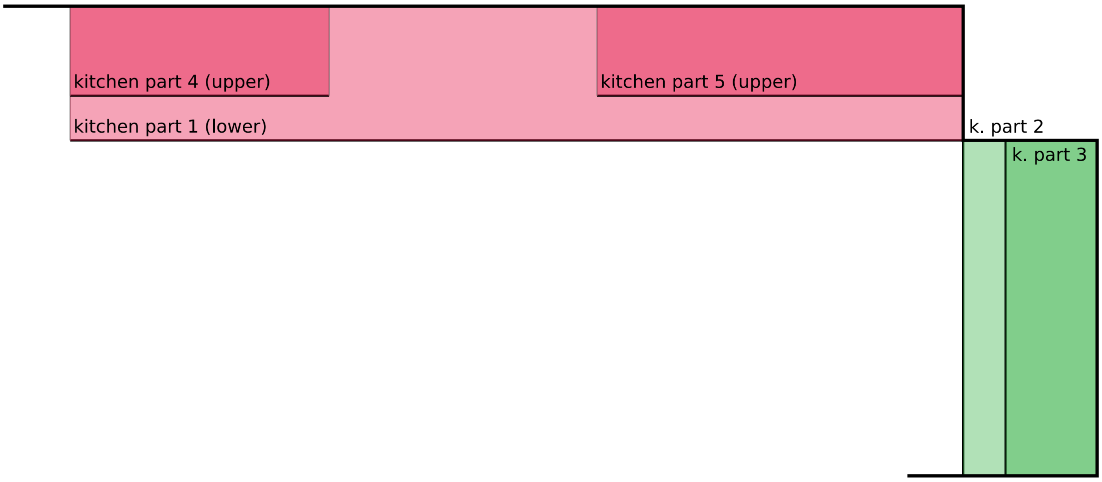


Figure 3.1 Kitchen parts and kitchen groups – a plan view of a kitchen scheme. There are two kitchen groups: a red group and a green group. The red group contains two upper kitchen parts and one lower part; the green group contains a lower kitchen part and an upper kitchen part. The only kitchen part with a non-zero offset is kitchen part 5.

Our model does not store any information about the relative positions of kitchen parts in different groups.

Placement Rules

One of the most important things the designer needs to specify is the exact placement of some fixtures. In other words, they already have a basic idea about the positions of some fixtures and want the solutions to reflect that. This may happen due to the plumbing or ventilation, client's preferences, etc.

To model these placement requirements, we use two types of placement rules: *exclude rules* and *include rules*. An exclude rule enforces that the given area does not contain any of the given fixtures – not even partially. An include rule states that the area has to contain at least one of the given fixtures entirely.

We introduce a set of rules $R = \{r_1, \ldots, r_q\}$. There is a partition $R = R_{\mathrm{ex}} \mathbin{\dot\cup} R_{\mathrm{in}}$ dividing them into exclude and include rules. Rules are partitioned into those applying to the entire kitchen (*global* rules), a kitchen group, and a section of a group described by an interval, denoted R_k, R_g, R_s, respectively, with $R = R_k \mathbin{\dot\cup} R_g \mathbin{\dot\cup} R_s$.

We define the following mappings:

- $\mathrm{rfix} : R \to 2^F$ describing the fixtures that are affected by the rule,
- $\mathrm{rgroup} : R_g \cup R_s \to G$ specifying the selected group,
- $\mathrm{rint} : R_s \to \{[a, b] \mid a, b \in [0, \infty)\}$ selecting the interval for section rules.

The designer may sometimes decide not to respect the current placement of the plumbing or ventilation if they believe that it allows them to achieve a better layout. Our model leaves this decision up to the designer – it follows the defined placement rules and does not try to ignore any of them.

Kitchen Layout

In Section 2.5, we mentioned that the model of a kitchen with fitted fixtures can be expressed as

1. assignments of individual fixtures to their kitchen parts in a specific order,
2. widths of the fixtures.

In our mathematical formulation, each kitchen part can be subdivided into multiple ordered segments of specific widths. We let the fixtures be assigned to the segments.

Let $\mathcal{W}$ be a collection of partitions of kitchen parts into line segments:

$$\mathcal{W} = \{W(p) \mid p \in P\}. \tag{3.1}$$

More precisely, for each part $p \in P$, $W(p)$ is a finite partition of the line segment $[0, \mathrm{width}(p)]$.

Now, we define an assignment of fixtures to segments:

$$\varphi : \bigcup_{p \in P} W(p) \to F \cup \{\emptyset\}. \tag{3.2}$$

Note that $\bigcup_{p \in P} W(p)$ might be a multiset and that we treat each segment as unique.

A *kitchen layout* is a pair $\mathcal{L} = (\mathcal{W}, \varphi)$.

To enforce the desired meaning of the presented sets and mappings, we introduce several constraints.

3.1.2 Constraints

Each fixture can be used at most once:

$$\forall f \in F \text{ there is at most one } w \text{ such that } \varphi(w) = f. \tag{3.3}$$

We do not allow empty segments between fixtures. There is at most one empty segment at each end of the kitchen part:

$$(\forall p \in P)(\forall w \in W(p)) : \varphi(w) = \emptyset \to l(w) = 0 \vee r(w) = \mathrm{width}(p), \tag{3.4}$$

we use the notation $w = [l(w), r(w)]$.

Each assigned fixture has a correct width with respect to segment size:

$$(\forall p \in P)(\forall w \in W(p)) : |w| \in [\mathrm{wmin}(\varphi(w)), \mathrm{wmax}(\varphi(w))]. \tag{3.5}$$

Fixtures are in correct kitchen parts:

$$(\forall p \in P)(\forall w \in W(p)) : \varphi(w) \in U_{\mathrm{fix}} \to p \in U_{\mathrm{part}}, \tag{3.6}$$

$$\varphi(w) \in L_{\mathrm{fix}} \to p \in L_{\mathrm{part}}. \tag{3.7}$$

The lower and upper halves of tall fixtures are matching:

$$\begin{aligned}
&(\forall p \in P)(\forall w \in W(p)) : \\
&\qquad \varphi(w) \in T \cap L_{\mathrm{fix}} \to (\exists p' \in P)(\mathrm{group}(p) = \mathrm{group}(p')) \\
&\quad (\exists w' \in W(p'))(\{\varphi(w), \varphi(w')\} \in \mathcal{T} \wedge w + \mathrm{offset}(p) = w' + \mathrm{offset(p')}).
\end{aligned} \tag{3.8}$$

The visible side fixtures of kitchen parts have neat panels:

$$
\begin{aligned}
&(\forall p \in P)(\forall w \in W(p)) : \\
&\quad \text{if } (\nexists w' \in W(p))(w' < w \wedge \varphi(w') \neq \emptyset) \text{ then } \mathrm{vleft}(p) \rightarrow \mathrm{pleft}(\varphi(w)), && (3.9)\\
&\quad \text{if } (\nexists w' \in W(p))(w' > w \wedge \varphi(w') \neq \emptyset) \text{ then } \mathrm{vright}(p) \rightarrow \mathrm{pright}(\varphi(w)), && (3.10)
\end{aligned}
$$

where we use an ordering such that $w < w'$ if $r(w) \leq l(w')$.

The placement rules are enforced:

$$
\begin{aligned}
&(\forall r \in R_{\mathrm{ex}} \cap R_k)(\forall f \in \mathrm{rfix}(r))(\forall p \in P)(\forall w \in W(p)) : \varphi(w) \neq f, && (3.11)\\
&(\forall r \in R_{\mathrm{ex}} \cap R_g)(\forall f \in \mathrm{rfix}(r))(\forall p \in P)(\forall w \in W(p)) : && (3.12)\\
&\quad \mathrm{group}(p) = \mathrm{rgroup}(r) \rightarrow \varphi(w) \neq f,\\
&(\forall r \in R_{\mathrm{ex}} \cap R_s)(\forall f \in \mathrm{rfix}(r))(\forall p \in P)(\forall w \in W(p)) : && (3.13)\\
&\quad \mathrm{group}(p) = \mathrm{rgroup}(r) \wedge \mathrm{offset}(p) + w \cap \mathrm{rint}(r) \neq \emptyset \rightarrow \varphi(w) \neq f,\\
&(\forall r \in R_{\mathrm{in}} \cap R_k)(\exists f \in \mathrm{rfix}(r))(\exists p \in P)(\exists w \in W(p)) : \varphi(w) = f, && (3.14)\\
&(\forall r \in R_{\mathrm{in}} \cap R_g)(\exists f \in \mathrm{rfix}(r))(\exists p \in P)(\exists w \in W(p)) : && (3.15)\\
&\quad \mathrm{group}(p) = \mathrm{rgroup}(r) \wedge \varphi(w) = f,\\
&(\forall r \in R_{\mathrm{in}} \cap R_s)(\exists f \in \mathrm{rfix}(r))(\exists p \in P)(\exists w \in W(p)) : && (3.16)\\
&\quad \mathrm{group}(p) = \mathrm{rgroup}(r) \wedge \mathrm{offset}(p) + w \subseteq \mathrm{rint}(r) \wedge \varphi(w) = f.
\end{aligned}
$$

3.1.3 Objective Function

For a kitchen layout $\mathcal{L}$, the objective function $\mathrm{obj}(\mathcal{L})$ consists of multiple components. It is defined and modified iteratively based on the client's requirements. There are therefore many variants of the function, and we will not describe them all formally.

With that being said, we introduce a simplified version of the function that expresses our main objective – to fill kitchen parts with fixtures, i.e., to *maximize* the sum of widths of non-empty segments:

$$
\mathrm{obj}(\mathcal{L}) = \sum_{p \in P} \sum_{\substack{w \in W(p) \\ \varphi(w) \neq \emptyset}} |w|. \tag{3.17}
$$

3.1.4 Kitchen Layout Problem

The Kitchen Layout Problem (KLP) is to find such a kitchen layout $\mathcal{L} = (\mathcal{W}, \varphi)$ for given sets F, P, G, R that satisfies the constraints and the value of the objective function $\mathrm{obj}(\mathcal{L})$ is maximized.

Lemma 1. *KLP is NP-complete.*

Proof. KLP belongs to the NP complexity class because, for any given k, if a layout $\mathcal{L}$ with at least k units of used space exists, then the layout itself is a polynomial certificate of this fact.

To show that KLP is NP-complete, we reduce from the Subset Sum Problem (SSP) as formulated by Kleinberg and Tardos [24, p. 491]:

Given natural numbers $w_1, \ldots, w_n$, and a target number W, is there a subset of $\{w_1, \ldots, w_n\}$ that adds up to precisely W?

To reduce SSP to KLP, we define a following instance of KLP:

- $F = \{f_1, \ldots, f_n\} = L_{\text{fix}}$,
- $\forall i \in \{1, \ldots, n\} : \text{wmin}(f_i) = \text{wmax}(f_i) = w_i$,
- $P = \{p_1\} = L_{\text{part}}$,
- $\text{width}(p_1) = W$.

It is clear that a kitchen layout with no unused space, that is, with objective value W exists if and only if the SSP instance is a YES instance.

We have shown that the KLP belongs to NP and that there exists a reduction from an NP-complete problem – the SUBSET SUM PROBLEM. Therefore, KITCHEN LAYOUT PROBLEM is NP-complete. □

This justifies our choice of tools that were made to deal with NP problems: mixed integer linear programming and constraint programming in general. We can now present the particular formulation of both the MILP and MiniZinc models.

3.2 Model Implementations

As for the set of fixtures F, set of kitchen parts P, set of kitchen groups G, and set of placement rules R, our MILP and MiniZinc formulations closely follow the notation and meaning of KLP. To keep things simple, we introduce some additional mappings:

- $\text{topp} : P \to \{0, 1\}$ equal to one for upper kitchen parts, zero for lower kitchen parts,
- $\text{topf} : F \to \{0, 1\}$ equal to one for upper fixtures (or halves), zero for lower fixtures,
- $\text{tall} : F \to \{0, 1\}$ indicating if the fixture f is one of the halves of a tall fixture,
- $\text{compl} : F \to F \cup \{\emptyset\}$ defining the complementary half of a tall fixture (applicable only for tall fixtures).

3.2.1 Segments

The main difference between KLP and the MILP and MiniZinc implementations is in the definition of segments and the assignment function. In our implementations, we divide the process of splitting the kitchen part into segments and assigning the fixtures to segments into two parts:

1. choosing a specific number of segments for each part,
2. setting the widths of the segments and assigning the fixtures to them.

The number of segments in a kitchen part p is determined during the preprocessing of the input data by the formula $\left\lceil \frac{\mathrm{width}(p)}{\mathrm{avgw}} \right\rceil$, where avgw is a constant provided by the designer – it should be close to the expected average width of the fixtures in the kitchen. The solver then only decides the widths of the segments and the assignments.

Note that the constant avgw impacts the solution space and the solving time significantly; choosing avgw poorly could prevent finding truly optimal or even feasible solutions if it leads to too few or too many segments.

In the mathematical model, every segment w is an interval with a defined width. In our implementations, we use a different kind of segments that do not hold any information about their widths implicitly. We denote the set of these segments by $S = \{s_1, \ldots, s_\ell\}$. There is a partition $S = S_{p_1} \dot\cup \cdots \dot\cup S_{p_k}$ into kitchen parts.

There is a left-to-right ordering of the segments in a kitchen part. Also, we need to be able to sort the segments globally in order to break symmetries in solutions. A simple way to ensure this is to use $S = \{1, 2, \ldots, M_n\}$, where M_n is the total number of segments (or a maximum segment number), and set $S_{p_1} = \{1, 2, \ldots, s_i\}$, $S_{p_2} = \{s_i+1, s_i+2, \ldots, s_j\}, \ldots, S_{p_k} = \{s_h, \ldots, M_n\}$. We will assume this onwards. While $\emptyset$ stands for "no fixture", we use 0 for "no segment".

We add the following mappings:

- part : $S \to P$, mapping a segment s to the kitchen part it belongs to,
- first : $S \to \{0, 1\}$, and last : $S \to \{0, 1\}$, stating if the segment is first/last in the kitchen part part(s),
- pre : $S \to S \cup \{0\}$, equal to the segment that precedes segment s in the kitchen part part(s) (not applicable for the *first* segments).

To simplify the models, we will require that a segment $s \in S$ has a zero width if and only if it does not contain any fixture. Because of that, we need to represent empty spaces at the ends of the kitchen parts in a different way: For the left padding, we add decision variables; the right padding can be deduced from the other decision variables and constants.

3.2.2 Decision Variables

For the MILP model, we define the decision variables

$$(\forall p \in P) : \mathrm{padding}_p \in [0, M_c], \tag{3.18}$$

$$(\forall s \in S) : \mathrm{width}_s \in [0, M_w], \tag{3.19}$$

$$(\forall s \in S)(\forall f \in F) : \mathrm{pair}_{sf} \in \{0, 1\}. \tag{3.20}$$

The variable $\mathrm{padding}_p$ equals to the left padding of the kitchen part p. The variable width_s corresponds to the width of the segment s. The variable pair_{sf} equals one if and only if the segment s contains the fixture f.

The variables are bounded by constants deduced from the input or explicitly defined by the designer: M_w corresponds to the maximum permitted width of any fixture; M_c is the maximum canvas size, i.e., the width or height of the kitchen floor plan, whichever is greater.

3.2.3 MiniZinc Model

In the MiniZinc model, the variables have the same ranges, but we restrict the values to integers as using continuous variables would discriminate against some of the available solvers.

Alternatively, we could assign the fixtures to segments directly using a set of $|F|$ categorical variables with $|S \cup \{0\}|$ values instead of the set of $|S| \times |F|$ binary variables. However, when an MILP solver is used to solve the MiniZinc model, a categorical variable is automatically converted to a set of binary ones (and a sum constraint, see Subsection 1.4.1), so there might be no additional benefit. It could improve the performance of some solvers (mainly those using other methods than MILP), but for simplicity, we stick to the binary formulation of pair_{sf}.

Here, we describe the MiniZinc model using standard mathematical notation and predicate logic. To simplify the expressions and make them more similar to the formulations in MiniZinc language, we may treat any binary variable $b \in \{0, 1\}$ as a predicate $b = 1$. Then $\neg b$ means $b \neq 1$ or $b = 0$.

3.2.4 Kitchen Layout Reconstruction

From the values of the decision variables in a given solution, we can construct a kitchen layout $\mathcal{L} = (\mathcal{W}, \varphi)$ as follows. For each $p \in P$ and its segments $S_p = \{s_1, s_2, \ldots, s_{\ell_p}\}$, we define a partition

$$\begin{aligned} W(p) = \{&[0, \mathrm{padding}_p], \\ &[0, \mathrm{width}_{s_1}] + \mathrm{padding}_p, \\ &[0, \mathrm{width}_{s_2}] + \mathrm{padding}_p + \mathrm{width}_{s_1}, \\ &\ldots, \\ &[0, \mathrm{width}_{s_{\ell_p}}] + \mathrm{padding}_p + \mathrm{width}_{s_1} + \cdots + \mathrm{width}_{s_{\ell_p - 1}}, \\ &[\mathrm{padding}_p + \mathrm{width}_{s_1} + \cdots + \mathrm{width}_{s_{\ell_p}}, \mathrm{width}(p)]\}. \end{aligned} \tag{3.21}$$

For the first and last items of the set $W(p)$, the mapping $\varphi(w)$ is equal to $\emptyset$ as they correspond to the left or right padding. For any other interval $w \in W(p)$ such that $w = [0, \mathrm{width}_{s_i}] + \mathrm{padding}_p + \mathrm{width}_{s_1} + \cdots + \mathrm{width}_{s_{i-1}}$, it holds that $\varphi(w) = f$ if there is such an $f \in F$ that $\mathrm{pair}_{sf} = 1$. Otherwise, φ maps to $\emptyset$.

To make the models return valid layouts and possibly reduce the time needed to find the best solutions, we now need to implement some constraints.

3.3 Implemented Constraints

Each segment contains at most one fixture:

$$(\forall s \in S) : \sum_{f \in F} \mathrm{pair}_{sf} \leq 1. \tag{3.22}$$

Every fixture can be assigned to at most one segment (see Constraint 3.3):

$$(\forall f \in F) : \sum_{s \in S} \mathrm{pair}_{sf} \leq 1. \tag{3.23}$$

3.3.1 Contiguous Fitting

Similarly to Constraint 3.4, we require that there are no empty segments between the fixtures. As for the ends of the kitchen parts, we allow that the empty segments are present only at the right end – there can be multiple of them.

In the MILP model, we introduce an auxiliary variable

$$(\forall s \in S) : \mathrm{used}_s \in \{0, 1\}, \tag{3.24}$$

where used_s equals one if and only if there is a fixture assigned to that segment. This is ensured using the set of constraints

$$(\forall s \in S) : \mathrm{used}_s = \sum_{f \in F} \mathrm{pair}_{sf}. \tag{3.25}$$

Note that we can use two inequalities to express an equality constraint.

Then, for all segments that are not first in their kitchen part, we add constraints requiring that every used segment s is preceded by another used segment s':

$$\mathrm{used}_s \leq \mathrm{used}_{s'}. \tag{3.26}$$

If we treat the variables as logical, this expresses the implication "if the segment s is used, then the previous segment s' is used too". It means that if there are any unused segments, they need to be at the very end of the kitchen part.

With the new auxiliary variables, we can even remove the constraint 3.22 as the sum is constrained by the domain of the variable.

Width of Empty Segments

Let us have a closer look at the unused segments. It seems natural that such segments should always have zero width as there is no benefit in dividing empty space into segments. A set of constraints

$$(\forall s \in S) : \mathrm{width}_s \leq M_w \cdot \mathrm{used}_s \tag{3.27}$$

will ensure that.

Also, it does not make sense for any fixture to have zero width. To avoid such cases, we need a constant m_w corresponding to a minimum width of a fixture. Then we can use the constraint

$$(\forall s \in S) : \mathrm{width}_s \geq m_w \cdot \mathrm{used}_s. \tag{3.28}$$

MiniZinc Implementation

Instead of the auxiliary variable used_s we define a predicate $\mathrm{used}(s)$ such that

$$\mathrm{used} = \{s \in S \mid (\exists f \in F) : \mathrm{pair}_{sf} = 1\}. \tag{3.29}$$

Note that instead of "$(\exists f \in F) : \mathrm{pair}_{sf} = 1$", we could write "$(\exists f \in F) : \mathrm{pair}_{sf}$".

We then rewrite the constraints 3.26, 3.27 and 3.28 as

$$(\forall s \in S) : \mathrm{first}(s) \lor (\mathrm{used}(s) \rightarrow \mathrm{used}(\mathrm{pre}(s))), \tag{3.30}$$

$$(\forall s \in S) : (\mathrm{width}_s > 0 \rightarrow \mathrm{used}(s)) \land (\mathrm{used}(s) \rightarrow \mathrm{width}_s \geq m_w). \tag{3.31}$$

3.3.2 Present Fixture

The variable used_s or the predicate $\text{used}(s)$ tells us whether any fixture was assigned to the segment s. Likewise, we use a variable present_f in the MILP model to see if the fixture is present in the layout, i.e., whether it was assigned to any segment:

$$(\forall f \in F) : \text{present}_f \in \{0, 1\}, \tag{3.32}$$

$$(\forall f \in F) : \text{present}_f = \sum_{s \in S} \text{pair}_{sf}. \tag{3.33}$$

With this constraint in place, we can remove Constraint 3.23.

In the MiniZinc model, we add a predicate instead:

$$\text{present} = \{f \in F \mid (\exists s \in S) : \text{pair}_{sf} = 1\}. \tag{3.34}$$

Note that the pairing constraints 3.22 and 3.23 cannot be removed from the MiniZinc model.

3.3.3 Widths, Vertical Placement

It is necessary to check that the lower fixtures are not placed in the upper kitchen parts and vice versa (see constraints 3.6 and 3.7), that the individual width of each fixture fits in its allowed range (as in Constraint 3.5), and that the real width of the kitchen part is not exceeded by the total width of its contents. In the MILP model, we need four sets of constraints: For each pair of segment s and fixture f that are incompatible, we add the constraint

$$\text{pair}_{sf} \leq 0 \tag{3.35}$$

or directly remove such variables. The constraints related to the fixture width look like this:

$$(\forall s \in S)(\forall f \in F) : \text{width}_s \geq \text{wmin}(f) \cdot \text{pair}_{sf}, \tag{3.36}$$

$$(\forall s \in S)(\forall f \in F) : \text{width}_s \leq \text{wmax}(f) + M_w \cdot (1 - \text{pair}_{sf}). \tag{3.37}$$

We can express it all using one constraint in the MiniZinc model:

$$\begin{aligned}(\forall s \in S)(\forall f \in F) : \text{pair}_{sf} \rightarrow (&\text{topf}(f) = \text{topp}(\text{part}(s))\\ &\wedge \text{width}_s \geq \text{wmin}(f) \wedge \text{width}_s \leq \text{wmax}(f)).\end{aligned} \tag{3.38}$$

The kitchen part width constraint is the same for both models:

$$(\forall p \in P) : \text{padding}_p + \sum_{s \in S_p} \text{width}_s \leq \text{width}(p). \tag{3.39}$$

3.3.4 Tall Fixtures

We need to make sure that both halves of a tall fixture are connected, i.e., that the widths and offsets of both segments match and that they belong to the same kitchen group. This corresponds to Constraint 3.8 in the mathematical model.

In the MILP model, we need to introduce several auxiliary variables and the corresponding constraints. We start with the widths of the fixtures. We introduce an auxiliary variable width_f which is 0 if the fixture is not present in the layout, and equals the width of the segment containing the fixture f otherwise:

$$(\forall f \in F) : \text{width}_f \in [0, M_w], \tag{3.40}$$
$$(\forall s \in S)(\forall f \in F) : \text{width}_s - M_w \cdot (1 - \text{pair}_{sf}) \leq \text{width}_f, \tag{3.41}$$
$$(\forall s \in S)(\forall f \in F) : \text{width}_f \leq \text{width}_s + M_w \cdot (1 - \text{pair}_{sf}), \tag{3.42}$$
$$(\forall s \in S)(\forall f \in F) : \text{width}_f \leq M_w \cdot \text{present}_f. \tag{3.43}$$

In order to use the offsets of the fixtures, we first need to express the offsets of the segments – their distances from the beginning of their kitchen group:

$$(\forall s \in S) : \text{offset}_s \in [0, M_c], \tag{3.44}$$
$$(\forall s \in S) : \text{offset}_s = \text{offset}(p) + \text{padding}_p + \sum_{\substack{s' \in S_p \\ s' < s}} \text{width}_{s'}, \tag{3.45}$$

where p corresponds to $\text{part}(s)$.

Then, we get the offsets of fixtures as

$$(\forall f \in F) : \text{offset}_f \in [0, M_c], \tag{3.46}$$
$$(\forall s \in S)(\forall f \in F) : \text{offset}_s - M_c \cdot (1 - \text{pair}_{sf}) \leq \text{offset}_f, \tag{3.47}$$
$$(\forall s \in S)(\forall f \in F) : \text{offset}_f \leq \text{offset}_s + M_c \cdot (1 - \text{pair}_{sf}), \tag{3.48}$$
$$(\forall f \in F) : \text{offset}_f \leq M_c \cdot \text{present}_f. \tag{3.49}$$

Again, the offset equals 0, if the fixture is not present in the layout.

Last but not least, we use an auxiliary variable presgroup_{gf} telling us if the fixture f is present in the group g:

$$(\forall g \in G)(\forall f \in F) : \text{presgroup}_{gf} \in \{0, 1\}, \tag{3.50}$$
$$(\forall g \in G)(\forall f \in F) : \text{presgroup}_{gf} = \sum_{\substack{s \in S \\ \text{group}(p) = g}} \text{pair}_{sf}, \tag{3.51}$$

where p is $\text{part}(s)$.

Then, for every half f of a tall fixture and its complementary half f', we add these constraints:

$$\text{offset}_f \leq \text{offset}_{f'}, \tag{3.52}$$
$$\text{width}_f \leq \text{width}_{f'}, \tag{3.53}$$
$$(\forall g \in G) : \text{presgroup}_{gf} \leq \text{presgroup}_{gf'}. \tag{3.54}$$

In the MiniZinc model, we need to first define four functions: $\text{offsetSegment} : S \to \mathbb{N}$, $\text{offsetFixture} : F \to \mathbb{N}$, $\text{groupFixture} : F \to G$, and $\text{width} : F \to \mathbb{N}$, where

$$\text{offsetSegment}(s) = \text{offset}(p) + \text{padding}_p + \sum_{\substack{s' \in S_p \\ s' < s}} \text{width}_{s'}, \tag{3.55}$$

$$\text{offsetFixture}(f) = \sum_{s \in S} \text{pair}_{sf} \cdot \text{offsetSegment}(s), \tag{3.56}$$

$$\text{groupFixture}(f) = \sum_{s \in S} \text{pair}_{sf} \cdot \text{group}(p), \tag{3.57}$$

$$\text{widthFixture}(f) = \sum_{s \in S} \text{pair}_{sf} \cdot \text{width}_s \tag{3.58}$$

while p corresponds to part(s).

After that, we just define one set of constraints:

$$\begin{aligned}(\forall f \in F) : \text{compl}(f) = 0 \vee (&\text{groupFixture}(f) \leq \text{groupFixture}(\text{compl}(f)) \\ &\wedge \text{offsetFixture}(f) \leq \text{offsetFixture}(\text{compl}(f)) \\ &\wedge \text{widthFixture}(f) \leq \text{widthFixture}(\text{compl}(f))).\end{aligned} \tag{3.59}$$

3.3.5 Side Panels

A kitchen part may adjoin a wall on both sides. If it does not, it means that its one or both sides are visible. Not all fixtures are meant to be placed at the visible edges of the kitchen parts, so only the fixtures with neat side panels can be put at the visible edge.

Even though it is possible to add such side panel to any fixture, we introduce constraints that enforce this rule if needed. See constraints 3.9 and 3.10 for a mathematical formulation.

We start with the MILP model. For a pair of a segment $s \in S$ and a fixture $f \in F$ such that s is the first segment of its kitchen part p, vleft(p) equals one, and pleft(f) equals zero, we add the constraint $\text{pair}_{sf} = 0$. Similarly, if s is the last, vright(p) is one, and pright(f) is zero, we use the same constraint.

However, if the last segment is empty, we need to find the last non-empty segment and check its fixture. We do this by saying: "If a segment contains a fixture without a right side panel, there must be another used segment directly after it." For each segment s that is not first in its kitchen part p, for each $f \in F$, if vright$(p) = 1$ and pright$(f) = 0$, we define the constraint

$$\text{pair}_{\text{pre}(s),f} \leq \text{used}_s. \tag{3.60}$$

We do not have any mapping to the segment after a given segment, so we use pre(s) as the current segment – then s is the next one.

In the MiniZinc model, it is nearly the same:

$$\begin{aligned}(\forall s \in S)(\forall f \in F) : &(\text{first}(s) \wedge \text{vleft}(\text{part}(s)) \wedge \neg\text{pleft}(f) \rightarrow \neg\text{pair}_{sf}) \\ &\wedge (\text{last}(s) \wedge \text{vright}(\text{part}(s)) \wedge \neg\text{pright}(f) \rightarrow \neg\text{pair}_{sf}),\end{aligned} \tag{3.61}$$

$$\begin{aligned}(\forall s \in S)(\forall f \in F) : \text{if } \ &\neg\text{first}(s) \wedge \text{vright}(\text{part}(s)) \wedge \neg\text{pright}(f) \ \text{ then} \\ &\text{pair}_{\text{pre}(s),f} \rightarrow \text{used}_s \ \text{ endif.}\end{aligned} \tag{3.62}$$

3.3.6 Placement Rules

In the practical implementation of exclude and include rules, each rule contains a definition of the area and a selector determining the set of affected fixtures. We can apply the global exclude rules before passing the set of the fixtures F to our model, so we ignore these rules now.

Note that the combination of include and exclude rules is quite powerful. For example, if we want to make sure that a given fixture is present only in a selected kitchen part and nowhere else, we can use an exclude rule to exclude the fixture from the other kitchen parts. To require that it really is present in the kitchen part, we may use an include rule. However, for $n > 1$, it is not possible to ensure that there are at least n elements of a set $F' \subseteq F$ present in a given kitchen part. But it would not be complicated to add this functionality to the formulations stated below.

Regarding the group section rules, it is important to notice that an include rule treats the “range” (a combination of an offset and a width) differently than an exclude rule. The exclude rule forbids the fixtures to overlap the range. The include rule requires one of the fixtures to be fully contained in the range.

We add the following mappings:

- $\mathrm{roffset} : R_s \to [0, \infty)$ such that $x \mapsto l(\mathrm{rint}(x))$,
- $\mathrm{rwidth} : R_s \to [0, \infty)$ such that $x \mapsto r(\mathrm{rint}(x)) - l(\mathrm{rint}(x))$.

MILP Formulation

For global and group rules, we introduce the constraints

$$(\forall r \in R_{\mathrm{in}} \cap R_k) : \sum_{f \in \mathrm{rfix}(r)} \mathrm{present}_f \geq 1, \tag{3.63}$$

$$(\forall r \in R_{\mathrm{in}} \cap R_g) : \sum_{f \in \mathrm{rfix}(r)} \mathrm{presgroup}_{\mathrm{rgroup}(r),f} \geq 1, \tag{3.64}$$

$$(\forall r \in R_{\mathrm{ex}} \cap R_g) : \sum_{f \in \mathrm{rfix}(r)} \mathrm{presgroup}_{\mathrm{rgroup}(r),f} \leq 0. \tag{3.65}$$

To model the include section rules, we introduce an auxiliary variable $\mathrm{included}_{rf}$ which equals one if the fixture f satisfies the include rule r; we also add several constraints:

$$(\forall r \in R_{\mathrm{in}} \cap R_s)(\forall f \in \mathrm{rfix}(r)) : \mathrm{included}_{rf} \in \{0, 1\}, \tag{3.66}$$

$$\mathrm{presgroup}_{gf} \geq \mathrm{included}_{rf}, \tag{3.67}$$

$$\mathrm{offset}_f \geq \mathrm{roffset}(r) \cdot \mathrm{included}_{rf}, \tag{3.68}$$

$$\mathrm{offset}_f + \mathrm{width}_f \leq \mathrm{roffset}(r) + \mathrm{rwidth}(r) + M_c \cdot (1 - \mathrm{included}_{rf}), \tag{3.69}$$

$$(\forall r \in R_{\mathrm{in}} \cap R_s) : \sum_{f \in \mathrm{rfix}(r)} \mathrm{included}_{rf} \geq 1. \tag{3.70}$$

The exclude section rules are modeled using another auxiliary variable and the following two constraints, which guarantee that if the fixture is present in the concerned kitchen group, it has to be either before or after the given section. The variable after_{rf} equals one if the fixture is after the section:

$$(\forall r \in R_{\text{ex}} \cap R_s)(\forall f \in \text{rfix}(r)) : \text{after}_{rf} \in \{0, 1\}, \tag{3.71}$$

$$\text{offset}_f + \text{width}_f \leq \text{roffset}(r) + M_c \cdot \text{after}_{rf} + M_c \cdot (1 - \text{presgroup}_{gf}), \tag{3.72}$$

$$\text{roffset}(r) + \text{rwidth}(r) \leq \text{offset}_f + M_c \cdot (1 - \text{after}_{rf}) + M_c \cdot (1 - \text{presgroup}_{gf}). \tag{3.73}$$

Note that the two introduced auxiliary variables can be declared as a single variable defined for every $r \in R_s$.

MiniZinc Formulation

In the MiniZinc model, we describe the rules as a single constraint:

$$\begin{aligned}
&(\forall r \in R) : \\
&\quad \text{if} \ \ r \in R_{\text{in}} \cap R_k \ \ \text{then} \quad (\exists f \in \text{rfix}(r))(\exists s \in S) : \text{pair}_{sf} \\
&\quad \text{elseif} \ \ r \in R_{\text{in}} \cap R_g \ \ \text{then} \quad (\exists f \in \text{rfix}(r))(\exists s \in S_g) : \text{pair}_{sf} \\
&\quad \text{elseif} \ \ r \in R_{\text{in}} \ \ \text{then} \quad (\exists f \in \text{rfix}(r))(\exists s \in S_g) : \\
&\qquad\quad \text{pair}_{sf} \wedge \text{offsetSegment}(s) \geq \text{roffset}(r) \\
&\qquad\quad \wedge \text{offsetSegment}(s) + \text{width}_s \leq \text{roffset}(r) + \text{rwidth}(r) \\
&\quad \text{elseif} \ \ r \in R_g \ \ \text{then} \quad (\forall f \in \text{rfix}(r))(\forall s \in S_g) : \neg\text{pair}_{sf} \\
&\quad \text{else} \quad (\forall f \in \text{rfix}(r))(\forall s \in S_g) : \\
&\qquad\quad \neg\text{pair}_{sf} \vee \text{offsetSegment}(s) + \text{width}_s \leq \text{roffset}(r) \\
&\qquad\quad \vee \text{roffset}(r) + \text{rwidth}(r) \leq \text{offsetSegment}(s) \quad \text{endif},
\end{aligned} \tag{3.74}$$

where S_g denotes a set of all the segments that belong to the kitchen parts in a kitchen group g. We benefit from the fact that MiniZinc supports if-then-else expressions. The conditional branches are stated in this order: global include rules, group include rules, group section include rules, group exclude rules, and group section exclude rules.

3.3.7 Sorting the Duplicates

Even though there is generally no need to put multiple stoves or fridges in the kitchen, it is common to have multiple similar cabinets there. We stated that every fixture $f \in F$ can be assigned to at most one segment. However, if we introduce two (distinct) fixtures f_1, f_2 with the same properties, it may cause symmetries in the model that are known to increase the solving time – symmetry handling [25] is a common challenge in MILP.

To resolve that, we break the symmetries by marking the fixtures as duplicates and introducing constraints that enforce a certain order of fitting the duplicates in the kitchen.

First, we link the number of the segment with the fixture through a new auxiliary variable segnum_f:

$$(\forall f \in F) : \text{segnum}_f \in S \cup \{0\}, \tag{3.75}$$
$$(\forall s \in S)(\forall f \in F) : s - M_n \cdot (1 - \text{pair}_{sf}) \leq \text{segnum}_f, \tag{3.76}$$
$$(\forall s \in S)(\forall f \in F) : \text{segnum}_f \leq s - M_n \cdot (1 - \text{pair}_{sf}), \tag{3.77}$$
$$(\forall f \in F) : \text{segnum}_f \leq M_n \cdot \text{present}_f. \tag{3.78}$$

Note that the variable equals zero if and only if the fixture is not present. Otherwise, it is equal to the number of the segment that contains the fixture f.

We introduce a mapping $\text{older} : F \to F \cup \{\emptyset\}$ pointing to the "older sibling" – a duplicate of the fixture f that was created before f. This ensures that the set of duplicates is ordered. It is only applicable for the duplicated fixtures, the oldest points to $\emptyset$.

For each fixture with an older sibling $\text{older}(f)$, we then add the constraint

$$\text{segnum}_{\text{older}(f)} \leq \text{segnum}_f. \tag{3.79}$$

In the MiniZinc model, it is similar, we need a function $\text{segmentNumber} : F \to S \cup \{0\}$ and the constraint as follows:

$$\text{segmentNumber}(f) = \sum_{s \in S} s \cdot \text{pair}_{sf}, \tag{3.80}$$
$$(\forall f \in F) : \text{older}(f) = \emptyset \vee \text{segnum}_{\text{older}(f)} \leq \text{segnum}_f. \tag{3.81}$$

3.4 Objective Function in MILP and MiniZinc

So far, we have not defined any objective function for the two implementations. Let us consider an objective function $\text{obj} : D_1 \times \cdots \times D_n \to \mathbb{R}$, where $D_1, \ldots, D_n$ are the domains of the decision and auxiliary variables. As we mentioned in Subsection 3.1.3, the function consists of multiple components – the function corresponds to their sum.

Our main goal is to fit some fixtures in the kitchen. However, it is generally cheaper to fill the kitchen with a few wide fixtures than with many narrow fixtures, because every cabinet needs its own metal fitting, door, etc.

To reflect that, we reward the total width of the fixtures that are put in the kitchen. Furthermore, we add a penalty for every fitted fixture. This results in the two following components:

$$\text{width}^+ \cdot \sum_{s \in S} \text{width}_s, \tag{3.82}$$
$$(-5) \cdot \text{fixture}^- \cdot \sum_{s \in S} \sum_{f \in F} \text{pair}_{sf}, \tag{3.83}$$

where fixture^- and width^+ are coefficients (or *preferences*) that influence the "weight" of the components in the objective function. The default value of each of the constants is 1. The superscript "+" can be interpreted as a *bonus* while "−" denotes a *penalty*.

The ability to set the *preferences* is an important feature of the model. It allows to specify what the client and the designer consider important. The coefficient

(-5) helps to set a useful baseline so that we can leave the preferences set to one in most cases (when using centimeters as units of length).

This objective function can be used for both the MILP and MiniZinc models. The component 3.82 corresponds to the objective function 3.17.

4 Design Rules

After creating the basic MILP and MiniZinc models, it is now time to extend them by more items from the list of layout rules and requirements (see Section 2.4). We cannot use all of them because such a model may be too complicated. Therefore, we proceed in order of priority.

4.1 Implementation

For each rule, we present the necessary context and introduce new constraints and components of the objective function where needed.

It is hard to express our true intentions using simple constraints. Because of that, we assign some penalties even to perfectly fine and suitable kitchen designs. We try our best to balance the bonuses and penalties in such a way that the returned solutions follow the design rules reasonably well.

4.1.1 Grouping

In most kitchens, the tall fixtures are grouped together. It is very unusual to see a kitchen part where tall fixtures and lower/upper fixtures alternate. The reason is obvious: the space between the lower and upper fixtures is used for food preparation, washing dishes, etc. As illustrated in Figure 4.1, it is inconvenient to divide the space by a tall fixture.

For very similar reasons, we want to group the lower fixtures that can carry a worktop: it holds to a certain extent that the more spacious a worktop is, the better.

To ensure that the tall fixtures are grouped together as well as the "worktop fixtures", we add two sets of constraints. The two rules are surely not independent, as a tall fixture may never carry a worktop. We may even find out that one of the rules does not need to be enforced using its own constraints in order to be applied.

We start by grouping the tall fixtures in the MILP model. We introduce an auxiliary variable $\mathrm{talldisc}_s \in \{0, 1\}$ for each segment $s \in S$; *disc* stands for

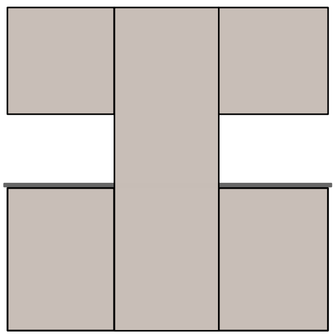


Figure 4.1 Tall fixture in the middle of a kitchen part. This is usually undesirable as the tall fixture divides the space between lower and upper fixtures and makes it harder to use the space for food preparation, washing the dishes, etc.

discontinuity. The variable is equal to one if the segment s is used and has a different value of $\text{tall}(f)$ (where f is the fixture placed in s) than the previous segment s'. Otherwise, it can be equal to either zero or one, so if we maximize $-\sum_s \text{talldisc}_s$, it will be equal to zero for optimal solutions in such cases. See the constraints:

$$\sum_{f\in F} \text{tall}(f) \cdot \text{pair}_{s'f} - \sum_{f\in F} \text{tall}(f) \cdot \text{pair}_{sf} \leq \text{talldisc}_s + (1 - \text{used}_s), \tag{4.1}$$

$$\sum_{f\in F} \text{tall}(f) \cdot \text{pair}_{sf} - \sum_{f\in F} \text{tall}(f) \cdot \text{pair}_{s'f} \leq \text{talldisc}_s + (1 - \text{used}_s). \tag{4.2}$$

To group the worktop fixtures, we introduce an auxiliary variable worktopdisc_s for each segment $s \in S$. We omit the constraints as they are nearly the same, instead of $\text{tall}(f)$, we use $\text{worktop}(f)$.

Now, we add two components to the objective function to account for this grouping:

$$(-5) \cdot \text{talldisc}^- \cdot \sum_{s\in S} \text{talldisc}_s, \tag{4.3}$$

$$(-5) \cdot \text{worktopdisc}^- \cdot \sum_{s\in S} \text{worktopdisc}_s. \tag{4.4}$$

In the MiniZinc model, we group the tall fixtures using two new predicates:

$$\text{isTall} = \{s \in S \mid (\exists f \in F) : \text{pair}_{sf} \wedge \text{tall}(f)\}, \tag{4.5}$$

$$\begin{aligned}\text{tallDisc} = \{s \in S \mid \neg\text{first}(s) \wedge \neg\text{topp}(\text{part}(s)) \wedge \text{used}(s) \\ \wedge\, \text{isTall}(s) \neq \text{isTall}(\text{pre}(s))\},\end{aligned} \tag{4.6}$$

where the inequality of two predicates corresponds to the operation "exclusive or".

The predicates for worktop fixtures are very similar. The MiniZinc objective function components then closely resemble the MILP components:

$$(-5) \cdot \text{talldisc}^- \cdot \sum_{s\in S} \text{tallDisc}(s), \tag{4.7}$$

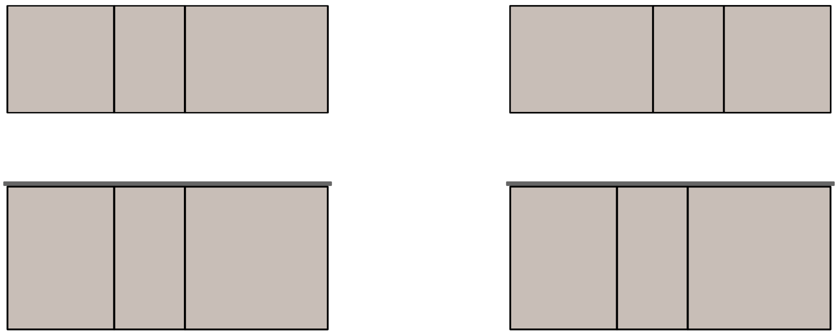
$$(-5) \cdot \text{worktopdisc}^- \cdot \sum_{s\in S} \text{worktopDisc}(s). \tag{4.8}$$

Here, we consider the numeric value of a predicate P to be equal to one if and only if P holds, otherwise the value equals zero.

4.1.2 Alignment / Vertical Continuity

Probably the most interesting requirement designers have when preparing the layout draft is to align the ends of the lower and upper fixtures that are directly above each other. For illustration, we show two simple layouts containing the same sets of fixtures. In Figure 4.2a, the fixtures are aligned; fixtures in Figure 4.2b are not. The former is generally preferred.

Another way to see this is that the designer is trying to make the *vertical boundaries* between the lower fixtures directly *continue* in some upper boundaries. There are multiple ways to define a *vertical discontinuity.* For the MILP model, we show two different definition and implementation approaches.



(a) aligned fixtures **(b)** misaligned fixtures

Figure 4.2 Illustration of alignment

MILP: Binary Approach

We look at the beginning of each segment $s \in S$, i.e., its first (usually left) vertical boundary, and say that s creates a discontinuity whenever the boundary intersects another segment $t \in S$. The segment s intersects t if and only if both are in the same kitchen group, one belongs to an upper kitchen part and the other to a lower kitchen part, and the first boundary of s is between the beginning and the end of t. The beginning and end of a segment generally correspond to its left and right boundaries, respectively.

Let S^{vc} denote the set of pairs $(s,t) \in S^2$ such that both segments are in the same kitchen group, one belongs to an upper kitchen part and the other one to a lower kitchen part.

For every segment $s \in S$, we define

$$\text{offset}'_s := \text{offset}(p) + \text{padding}_p + \sum_{\substack{s' \in S_p \\ s' < s}} \text{width}_{s'} \tag{4.9}$$

and use this expression in the following constraints, because it turned out during the deployment of the model that this approach is faster than using the auxiliary variable offset_s.

Now, for every $(s,t) \in S^{vc}$, we introduce the variables and constraints to capture the discontinuities as described:

$$\text{begin}_{st} \in \{0,1\}, \tag{4.10}$$
$$\text{end}_{st} \in \{0,1\}, \tag{4.11}$$
$$\text{intersect}_{st} \in \{0,1\}, \tag{4.12}$$
$$0 \leq -\text{offset}'_s + \text{offset}'_t + M_c \cdot \text{begin}_{st} \leq M_c, \tag{4.13}$$
$$0 \leq \text{offset}'_s - (\text{offset}'_t + \text{width}_t) + M_c \cdot \text{end}_{st} \leq M_c, \tag{4.14}$$
$$0 \leq \text{begin}_{st} + \text{end}_{st} + \text{used}_s - 3 \cdot \text{intersect}_{st} \leq 2. \tag{4.15}$$

In detail, begin_{st} equals one if t begins before the left boundary of s; end_{st} is equal to one if t ends after the left boundary of s. The variable intersect_{st} then indicates that the left boundary of s intersects the segment t.

To speed up the solution process, we add several constraints that describe the implied relations between the auxiliary variables; e.g., if a segment t begins before a segment s, then the segment $\text{pre}(t)$ also has to begin before s. These constraints are usually called *valid inequalities* and are commonly used in MILP to help solvers find the optimal solutions faster [26].

Let s' denote $\mathrm{pre}(s)$ for every $s \in S$ that is not the first in its kitchen part. Similarly, we define t'. Then we can add the constraints for the pairs $(s,t) \in S^{vc}$ where the expressions are applicable:

$$\mathrm{begin}_{s't} \leq \mathrm{begin}_{st}, \tag{4.16}$$

$$\mathrm{end}_{st'} \leq \mathrm{end}_{st}, \tag{4.17}$$

$$\mathrm{end}_{s't} \geq \mathrm{end}_{st}. \tag{4.18}$$

If we added the fourth missing constraint $\mathrm{begin}_{st'} \geq \mathrm{begin}_{st}$, it would slow down the solver. We have not found out the exact reason why this happens.

So far we have observed only the first (left) boundaries of the used segments. For most of the used segments, their second boundary corresponds to the first boundary of the following segment – except for the last ones. To take these boundaries into account, we add very similar variables and constraints for the pairs of a kitchen part $p \in P$ and a segment $t \in S$ that are in the same kitchen group but the kitchen part is upper and the segment is lower or the other way around (we denote this set by P^{vc}). The only difference is that we look at the ending (right) boundary of the kitchen part and check if it intersects any segment.

Then we can extend the objective function by this component:

$$(-4) \cdot \mathrm{vertdisc}^{-} \cdot \left(\sum_{(s,t) \in S^{vc}} \mathrm{intersect}_{st} + \sum_{(p,t) \in P^{vc}} \mathrm{intersect}_{pt} \right). \tag{4.19}$$

MILP: Piecewise Approach

An alternative way to penalize the discontinuities is by using a piecewise constraint. Piecewise linear formulation is a standard modeling technique [17] widely supported by modeling tools – we describe a function using a set of line segments (or by its breakpoints).

The perspective is different from the binary approach: We look at each pair of segments that could be aligned (same group, different elevations – one lower, the other upper) and check their offsets. If they are the same or differ by a lot, the penalty is zero as the segments are aligned or in completely different places, respectively. However, we penalize the situations when the offsets differ by a small amount – this means that the segments are related; they could be aligned but are not.

Figure 4.3 illustrates the difference between the binary approach and this piecewise approach. We consider the fixtures to be aligned. The piecewise approach does not assign any penalty to this layout, but the binary approach penalizes the right upper segment for intersecting the lower segment.

First, we define an auxiliary variable to get the offset difference:

$$\forall (s,t) \in S^2 : \mathrm{offsetdif}_{st} \in [-M_c, M_c], \tag{4.20}$$

$$\forall (s,t) \in S^{vc} : \mathrm{offsetdif}_{st} = \mathrm{offset}_s - \mathrm{offset}_t, \tag{4.21}$$

$$\forall (s,t) \in S^2 \setminus S^{vc} : \mathrm{offsetdif}_{st} = 0. \tag{4.22}$$

where $\mathrm{offset}_s, \mathrm{offset}_t$ are the auxiliary variables 3.44.

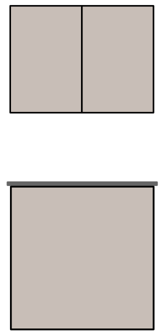


Figure 4.3 Two narrow upper fixtures above a wide lower fixture. The binary approach to alignment assigns a non-zero penalty to this layout; the piecewise approach considers the fixtures to be perfectly aligned.

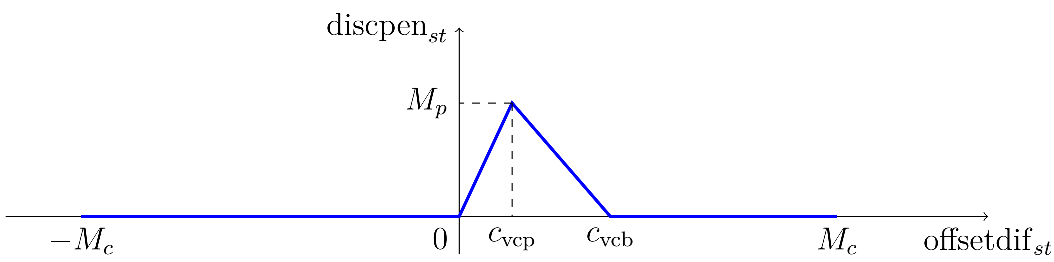


Figure 4.4 Discontinuity penalty as a function of offset difference

We introduce a constant M_p denoting maximum penalty and add an auxiliary variable:

$$\forall (s,t) \in S^2 : \mathrm{discpen}_{st} \in [0, M_p]. \tag{4.23}$$

The variable $\mathrm{discpen}_{st}$ (discontinuity penalty) is constrained by a piecewise constraint (see Figure 4.4) with the following breakpoints:

- $\mathrm{offsetdif}_{st} = -M_c,\ \mathrm{discpen}_{st} = 0;$
- $\mathrm{offsetdif}_{st} = 0,\ \mathrm{discpen}_{st} = 0;$
- $\mathrm{offsetdif}_{st} = c_{\mathrm{vcp}},\ \mathrm{discpen}_{st} = M_p;$
- $\mathrm{offsetdif}_{st} = c_{\mathrm{vcb}},\ \mathrm{discpen}_{st} = 0;$
- $\mathrm{offsetdif}_{st} = M_c,\ \mathrm{discpen}_{st} = 0,$

where $c_{\mathrm{vcp}}, c_{\mathrm{vcb}}$ are constants corresponding to a peak and a baseline value, respectively. In our implementation, we use $c_{\mathrm{vcp}} = 5$, $c_{\mathrm{vcb}} = 30$ as we expect the centimeters to be used by default (this could be generalized to support other units). The shape of the piecewise penalty function might need further tuning in order to improve the quality of generated layouts.

This would be the objective function component:

$$(-1) \cdot \mathrm{vertdisc}^{-} \cdot \sum_{(s,t) \in S^2} \mathrm{discpen}_{st}. \tag{4.24}$$

MiniZinc Formulation of the Binary Approach

In the MiniZinc model, we formulate only the binary approach. We define the following predicate:

$$\begin{aligned}\text{vertDiscSegment} = \{(s,t) \in S^2 \mid\ & \text{topp}(\text{part}(s)) \neq \text{topp}(\text{part}(t)) \\ & \wedge \text{group}(\text{part}(s)) = \text{group}(\text{part}(t)) \\ & \wedge \text{used}(s) \wedge \text{offsetSegment}(s) > \text{offsetSegment}(t) \\ & \wedge \text{offsetSegment}(s) < \text{offsetSegment}(t) + \text{width}_t\}.\end{aligned} \tag{4.25}$$

To include the ending boundaries of the kitchen parts, we define a function $\text{partEndOffset} : P \to \mathbb{N}$,

$$\text{partEndOffset}(p) = \text{offset}(p) + \text{padding}_p + \sum_{s' \in S_p} \text{width}_{s'}, \tag{4.26}$$

and a predicate

$$\begin{aligned}\text{vertDiscPart} = \{(p,t) \in P \times S \mid\ & \text{topp}(p) \neq \text{topp}(\text{part}(t)) \\ & \wedge \text{group}(p) = \text{group}(\text{part}(t)) \\ & \wedge \text{partEndOffset}(p) > \text{offsetSegment}(t) \\ & \wedge \text{partEndOffset}(p) < \text{offsetSegment}(t) + \text{width}_t\}.\end{aligned} \tag{4.27}$$

These are the new objective function components:

$$(-4) \cdot \text{vertdisc}^{-} \cdot \sum_{(s,t) \in S^2} \text{vertDiscSegment}(s,t), \tag{4.28}$$

$$(-4) \cdot \text{vertdisc}^{-} \cdot \sum_{(p,t) \in P \times S} \text{vertDiscPart}(p,t). \tag{4.29}$$

4.1.3 Fixtures with the Same Widths

There is another visually attractive trait that the kitchen can have: neighboring fixtures with the same widths. To accentuate it, we penalize the pairs of neighboring fixtures with different widths.

In the MILP model, we need to add an auxiliary variable $\text{wdiff}_s \in [0, M_w]$ that will be equal to the width difference between the segment s and the previous one. For segments that are first in their kitchen parts, we set it to zero. For every segment s of the other segments, we define an auxiliary variable $\text{prelarg}_s \in \{0,1\}$ (indicating that the previous segment s' is larger than s) and a set of constraints

$$\text{wdiff}_s \geq \text{width}_{s'} - \text{width}_s - M \cdot (1 - \text{used}_s), \tag{4.30}$$

$$\text{wdiff}_s \geq \text{width}_s - \text{width}_{s'} - M \cdot (1 - \text{used}_s), \tag{4.31}$$

$$\text{wdiff}_s \leq \text{width}_{s'} - \text{width}_s + M \cdot (1 - \text{prelarg}_s), \tag{4.32}$$

$$\text{wdiff}_s \leq \text{width}_s - \text{width}_{s'} + M \cdot \text{prelarg}_s, \tag{4.33}$$

$$\text{wdiff}_s \leq M \cdot \text{used}_s, \tag{4.34}$$

where s' equals $\text{pre}(s)$. The constant M equals $2M_w$ as we are dealing with two fixtures at once – if there is a narrow fixture after a wide fixture, the constant M_w might not be big enough.

We check the difference using another auxiliary variable and the constraint:

$$(\forall s \in S) : \text{wnotsame}_s \in \{0, 1\}, \tag{4.35}$$
$$(\forall s \in S) : 0 \leq -\text{wdiff}_s + M_w \cdot \text{wnotsame}_s \leq M. \tag{4.36}$$

The variable wnotsame_s equals one if the widths of two neighboring segments are not the same.

We include the penalty in the objective function:

$$(-2) \cdot \text{wnotsame}^- \cdot \sum_{s \in S} \text{wnotsame}_s. \tag{4.37}$$

The MiniZinc formulation is a bit simpler; instead of the variable wnotsame_s, we use this predicate:

$$\text{widthNotSame} = \{s \in S \mid \neg\text{first}(s) \wedge \text{used}(s) \wedge \text{width}_s \neq \text{width}_{\text{pre}(s)}\}. \tag{4.38}$$

The objective function then uses $\text{widthNotSame}(s)$ in place of wnotsame_s.

It is also possible to only look at the upper fixtures when checking the width differences as in the real kitchens, there are other more important influences that affect the positioning of the lower fixtures. This leads only to a slight modification of the formulations.

4.2 Remaining Rules

Up to now, we have implemented the most important rules. If we kept adding more without proper analysis and benchmark, we would risk creating a heavy-footed model that is hard to use and hard to debug. In this section, we discuss the remaining rules, their necessity, and possible ways of expression.

4.2.1 Relation Rules

Designers often need to express that a fixture should have various relations with other fixtures or its surroundings: That it should minimize its distance from a given point in the plan or a given fixture, e.g., to minimize the distance of a sink from a water supply or to reduce the distance between a dishwasher and a sink. We may also require that a certain fixture has at least a given distance from a wall or a specified fixture, or that there is a worktop adjacent to the fixture.

It is possible to enforce a certain distance from a wall using an exclude placement rule. Other *relation rules* could be added as specialized constraints or objective function terms. Alternatively, the rules could be implemented in the automation program to notify the designer if the proposed layout does not follow them.

4.2.2 Standard Cabinet Widths

Since custom-made cabinets are more expensive than cabinets of standard widths, we may want to penalize cabinets with non-standard widths. To enforce it, we could apply a penalty for every fixture with a variable width that is present in the layout. Precisely speaking, a fixture with a variable width is a fixture whose minimum width is different from its maximum width.

Clearly, this measure does not have any influence on the width of such fixtures, it only minimizes their use. It is therefore necessary to include a large enough set of standard cabinets.

4.2.3 Alternating Widths

If neighboring fixtures cannot have the same width, it is possible to achieve another "pattern": There can be a narrow fixture between two wide fixtures or vice versa so that the widths alternate. However, such large patterns are hard to describe in the model. As they occur rarely, it is advisable to leave them unimplemented.

4.2.4 Group by Zone

As described in Subsection 1.1.3, the fixtures are divided into several groups based on the activities they are used for: cooking, washing, storage, preparation. For some of the zones, we expect that their fixtures are close together in the layout.

For such a zone, we could define a point in the plan and require that the fixtures are placed as close to the point as possible. Without specifying any specific point, it is also possible to determine the current centroid of the zone and minimize the distance of the fixtures from the centroid. However, our current mathematical model would not suffice. Likely, we would need to use global coordinates in order to group the fixtures across the kitchen parts.

The grouping of fixtures by zone can be partially enforced by placement rules and by grouping fixtures with a worktop (see Subsection 4.1.1). As for the perimeter of the work triangle, we decided that there is no simple way to incorporate it into our model and that it should be enforced by the designer.

4.2.5 Worktop Size, Storage Space

To facilitate food preparation, it is desirable to have a worktop as large as reasonably possible. We could maximize the largest uninterrupted worktop, but we decided to select a simpler approach – to group the fixtures carrying a worktop (see Subsection 4.1.1).

A viable way of maximizing the storage space is to assign a *storage coefficient* to each fixture and then maximize the total sum of the coefficients of the present fixtures. Naturally, the storage coefficient of a fixture should be proportional to the storage space available in the fixture.

If we expressed the worktop size in a similar way, the user could tell the designer whether they prefer a large worktop or a lot of storage space. The designer would then set the preferences (penalties and bonuses) so that the layout reflects the client's priorities.

4.2.6 Corners

As discussed in Subsection 1.1.2, L-shaped and U-shaped kitchens are common. There are various ways to deal with the corners in such layouts. Nevertheless, there is usually nothing to optimize; the designer can directly ask the client how they prefer to fill the corner and then base the layout on this decision. The corners can then be excluded from the kitchen scheme, which leads to a simplification of the model.

5 Deployment

We have proposed two possible models – their variables, constraints, and objective functions. Now, it is time to create a computer program that uses the models to design kitchen layouts so that we can test them in practice.

We have listed several program requirements, see Section 2.1. However, our program is not going to fulfill them all. One of the goals of this thesis is to examine various modeling approaches and compare them. We focus on the ability to quickly obtain valid solutions. We keep the user interface general enough so that it is easier to modify it, add more input fields, etc., even if this slightly compromises user-friendliness.

We create a Python application that provides two user interfaces. We use a command-line interface to test the models during their development, evaluate their performance, run benchmarks, and compare the results for different inputs. We also provide a web interface, which is interactive and makes it easier for designers to use our program. The main features of the interfaces are described in the following sections; we also provide user documentation in Section A.2.

5.1 Coding the Models

5.1.1 MILP

To describe and solve the MILP model in Python, we use Pyomo [27, 28] – an open-source optimization modeling package.

The model can be described using a Python code: Sets can be initialized by lists of Python objects. Constraints and objective functions are described using Python expressions. The decision and auxiliary variables can be indexed by the sets and used in the expressions. It is even possible to define constraints only for some elements of the sets using conditional execution.

Pyomo allows us to use several different MILP solvers through a unified interface. We can therefore compare the performance of different solvers on the same input data.

There are solvers that support some of the common modeling techniques (see Subsection 1.3.1) directly – it is not necessary to write them literally as expressions. However, most of them are not available through the unified interface and need an individual approach. As we do not have the capacity to fine-tune every solver, we use the generic approach and write the constraints literally.

5.1.2 MiniZinc

A MiniZinc model is naturally described in the MiniZinc modeling language. We use the official Python package to populate the model with input data, command it to be solved, and collect a solution. Like in Pyomo, we can use various solvers – we have already discussed that in Subsection 1.4.1.

5.2 Command-Line Tool

The command-line tool has three main features: it can run *experiments*, evaluate solutions, and debug MILP models.

5.2.1 Running an Experiment

To run an experiment, we need to provide input data and optionally also some other information such as an output directory or a number of repetitions. An experiment is usually described in a single JSON file or as a series of command-line arguments. The input data are then contained in other JSON files.

One kitchen is described using a single input file. The JSON data are loaded and transformed into Python objects describing the kitchen. They are then used to fill the MILP or MiniZinc model. It is solved, and the solution is loaded into the Python representation of the kitchen.

An experiment can be run on multiple kitchens at once. We can solve one kitchen using multiple solvers. It is even possible to configure a solver to return several solutions. In any case, the solutions are evaluated (see the next subsection Evaluating a Solution) and the results can be presented to the user – various forms of the output are supported.

To be able to easily compare two slightly different inputs (e.g., with different values of global constants), it is possible to run an A/B test: We can automatically modify the input data and compare the returned results for both versions.

As the time needed to find the optimal solution is an important metric for us, we return it as a part of the output. To make the measurement more robust, we provide the option to set the number of repetitions. The solve time is then averaged over these runs, but only the last generated layout is returned at the output as it should not differ from the previous results.

In Section 3.4, we introduced the possibility of expressing preferences in the form of multiplication constants (contained in the input file). When they are set to zero, the model ignores the criteria, and it may speed up the solving process. We will use this technique to check the effects of the rules and to compare solvers under various conditions.

Output

When running an experiment, it is possible to define which kinds of output are desired. The basic form suitable for further automated processing is a JSON file containing a single result for a single kitchen – the layout and some additional information. If there are multiple kitchens or solutions, there will be multiple output files.

The program supports displaying a plan view of the kitchen layout using the Matplotlib library. Each solution can be saved as an image file. The fixtures in the plan are colored according to their kitchen zones. It is also possible to save the solved Pyomo model into a file to further examine the constructed MILP model.

5.2.2 Evaluating a Solution

Since we need to compare the results of two different models, we define an additional way to evaluate the solution. It is all carried out by a Python function:

- It calculates the values of components of the objective function for the given objects describing the kitchen.
- It loops over its internal definitions of the majority of the constraints we have previously defined and checks if they hold; we assume that some elementary constraints hold in every kitchen and thus do not need to be checked.
- Finally, it collects other important data about the kitchen: a number of fitted fixtures, a number of distinct worktops, a number of vertical discontinuities, etc.

The evaluation results are then reported in a structured form. The result object consists of a list of objective function component values, *errors* indicating violated constraints, and *labels* describing the properties of the kitchen. There is also a single *score* calculated as the sum of the objective function components. For every error, a large constants gets subtracted from the score.

We can “manually” evaluate a single solution by providing the program with an input file and an output file. The evaluation procedure is also usually carried out automatically on every obtained solution; it is a part of the pipeline.

However, its main purpose is to evaluate solutions produced en masse when running an experiment. Then we do not need to check every single solution visually; we can rely on the metrics returned by the evaluation function.

It is even possible to test if the components of the objective function have a sufficient effect on the solutions: In each input file, there can be a list of requirements regarding the labels. If there is an unfulfilled one, it is reported as an error. For example, we can state that we expect the kitchen to have at least one pair of fixtures with the same width. If there is none, an item is added to the list of errors.

Test Set

To see if the implemented rules and objective function components have the desired effects, we have compiled a set of test kitchens (input files).

For instance, there are several input files testing the rule that the ends of the fixtures should be aligned. Such input data were created for the sole purpose of assessing if the solutions with aligned fixtures are really preferred. Some of the files also contain the requirement about the maximum number of misaligned fixtures (or discontinuities) in the solution.

For almost every rule, there are such “kitchens” that test it.

Benchmark Set

While it is important to verify that the objective function and all the constraints are working as expected, one of our priorities is to get the solutions quickly enough. To compare the performance of different formulations and modeling approaches, we need to test these approaches on a large number of various inputs.

The test files are usually very simple and focus on a single rule or only a few rules. We need a set of data that is closer to reality. Together with Salusoft89 and Kateřina Kratochvílová, we have compiled a set of input data based on kitchens that were manually designed in kitchen design studios. In the next chapter, we use the data as our benchmark set enabling us to compare the solvers and approaches.

Note that both the benchmark and test sets are quite small; they contain tens of layouts, which is minuscule compared to usual machine learning datasets.

5.2.3 Debugging an MILP Model

It may happen that for the same input data, MiniZinc returns a better solution than Pyomo. Or it may occur that we see that the solution proposed by our MILP model is not optimal. Both situations indicate that we made an error while defining the model.

To find the bug, we can construct an output file containing the solution we consider optimal. The output file can then be loaded into the model by fixing the values of the decision variables. Presumably, the model becomes infeasible – there is no solution that satisfies all the constraints. The program then prints an irreducible infeasible set (constraints and variable bounds making the model infeasible), which allows us to find the cause of the problem.

5.2.4 File Formats

The program loads or produces three types of machine-readable files: input files, output files, and experiment files. All of them use the JSON format, each type has its own JSON Schema.

An input file contains a list of kitchen parts, a set of fixtures, descriptions of kitchen zones, kitchen rules, definitions of constants and preferences. We can also define the tests (required labels) and instructions stating which platform and solver should be used, how many solutions should be returned, etc.

In an output file, there are lists of fixtures along with their widths that were fitted in each kitchen part and also the left padding sizes of the kitchen parts. Then there are some data describing the solution, the solver settings, and the evaluation results.

An experiment file may consist of a list of input files or directories, an output directory, output settings, and a number of repetitions to get reliable benchmarks. We can also use it to define pieces (objects or lists) that should be replaced in the input files – this can be used for the A/B testing.

Refer to Subsection A.2.3 for a detailed description of the file formats.

5.3 Web Tool

As designers usually do not have the technical skills to download and manage a Python command-line tool, we introduce a web application. It simplifies entering the input data and provides an interactive interface, which improves the overall user experience.

The website can be used to

- draw the plan of the kitchen – the positions and the dimensions of the kitchen parts,
- use a form to enter and edit all the input data,
- submit the input to be solved,
- load the solutions and view them,
- modify a solution and evaluate it,
- load input data that were submitted in the past along with their solutions.

Currently, the application provides access to Gurobi [3], which is a commercial solver. We are using an academic license, which means that the website can be used solely for research and educational purposes.

5.3.1 Architecture

The frontend application providing the user interface is created in Svelte[1] – a JavaScript framework – and uses SvelteKit[2] to be built into a bundle of static HTML, CSS, and JavaScript files. It sends its requests through an API to a Python backend which runs the solvers using Pyomo and MiniZinc interfaces.

As it may take some time to find the solutions and we might even want to run multiple solvers in parallel, we need to use the solvers independently of the request processing (or web server) logic. The API requests are processed using a Flask[3] framework running on a Gunicorn[4] server, the tasks are then passed to Celery[5] workers which can run the solvers. Celery uses a RabbitMQ[6] broker to pass messages and a MySQL[7] database to store results. We use the same database to store the submitted input data and the solutions.

The Gunicorn server is deployed behind an Nginx[8] reverse proxy. The Nginx server is also used to serve the static files of the frontend application.

The web application is meant to be hosted on a server that can run Docker[9] containers, because it consists of the aforementioned services that are all described in a Docker Compose file.

[1] `https://svelte.dev`
[2] `https://svelte.dev/docs/kit`
[3] `https://flask.palletsprojects.com`
[4] `https://gunicorn.org`
[5] `https://docs.celeryq.dev`
[6] `https://www.rabbitmq.com`
[7] `https://www.mysql.com`
[8] `https://nginx.org`
[9] `https://www.docker.com`

6 Evaluation

To evaluate the quality of our models, we carry out a series of experiments.

6.1 Quality of the Layouts

We first assess the solutions that the models return using the design rules and criteria. In this way, we can review whether the general formulation makes sense and does not have any fundamental issues. We ignore the solving times for now. To find the optimal solutions, we solve our MILP model using Gurobi [3], a commercial MILP solver.

6.1.1 Reading Layout Diagrams

As was mentioned in Section 5.2.1, our program can return the designed layouts in the form of diagrams – Figure 6.1 shows one such diagram.

It is a kind of a plan view: Every kitchen part has a red border; the upper kitchen parts are slightly offset so that all the fixtures are visible clearly – the original placing of an upper kitchen part is hatched. Each segment has three labels:

1. number of the segment,
2. name of the fixture if there is any, `None` otherwise,
3. width of the segment.

The coloring of the segments corresponds to the colors of the kitchen zones their fixtures belong to. Note that the empty segments are all at the right ends of the kitchen parts as we require in our model.

Every tall fixture is represented by a pair of an upper and a lower fixture, the names contain T for *top* or B for *bottom*; there is no other link in the diagram. The names of duplicates of a fixture contain an underscore and the number of duplicates.

The 2D coordinates are included in the input data in order to make the visual output realistic; they are not used in the model.

6.1.2 First Layout

We start by designing a layout of an in-line kitchen with an island. The result is shown in Figure 6.1. There were four placement rules in the *kitchen scheme* (input data) – they set the position of the fridge, cupboard3, sink, and stove. There are four kitchen zones: cooking (red), washing (blue), preparation (green), and storage (yellow).

The layout seems quite usable and nice: The tall fixtures are all on one side. There are several cases of pairs of fixtures with the same width. As for alignment, there is only one case of misaligned lower and upper fixtures which was unavoidable due to the limitations of the provided set of fixtures.

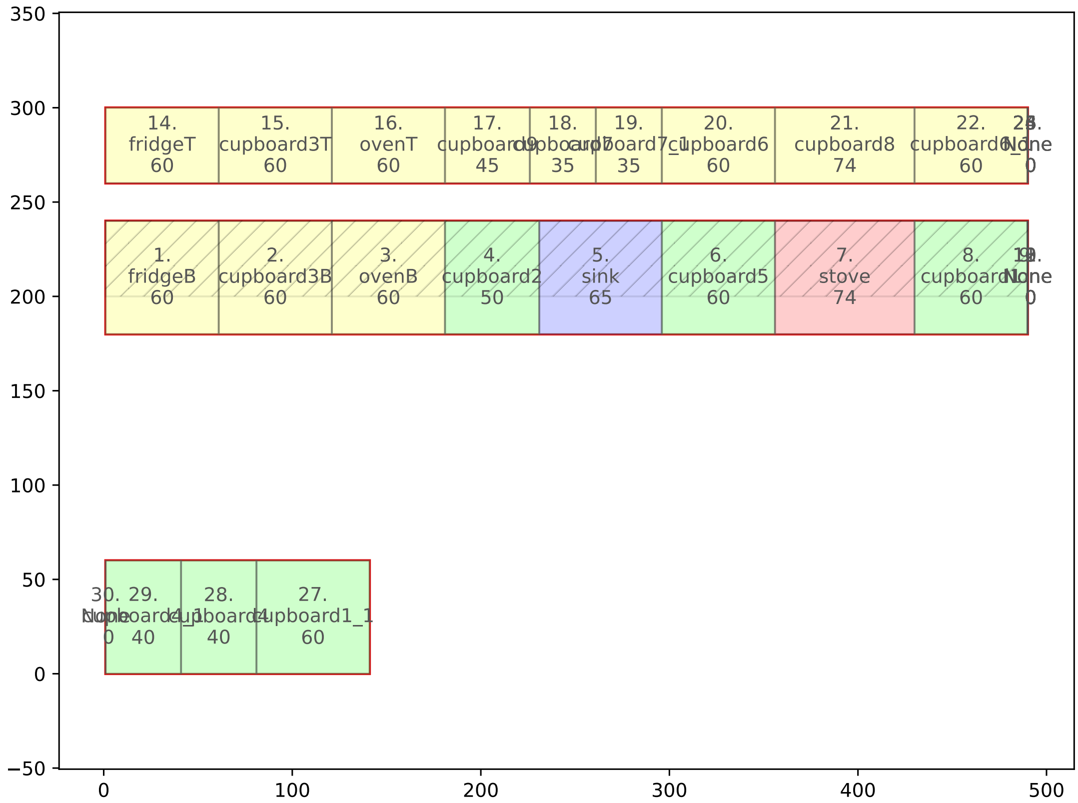


Figure 6.1 Layout of an in-line kitchen with an island

It is worth noting that the worktop is split into three pieces in the main lower kitchen part, but it leads to the sink and stove being surrounded by a worktop, which is practical.

6.1.3 Second Layout

Our next kitchen is in-line and contains two upper kitchen parts – probably due to a window. The kitchen zones are the same as in the previous example. There are no placement rules.

Figure 6.2 shows the resulting layout: The worktop is separated from the stove and the sink by the tall cabinet with an oven. It is usually necessary that both a stove and a sink have an adjacent worktop. However, our model does not explicitly enforce this rule.

Also, there is a component in the objective function which ensures that the fixtures with a worktop are grouped together. Without any placement rules, it leads to this layout being evaluated as optimal.

6.1.4 Third Layout

Third, we design a layout of an L-shaped kitchen. One of the upper kitchen parts does not span the whole width of the lower kitchen part, so it seems there is a window. Note that the corner is not a part of the kitchen scheme – there is a wall.

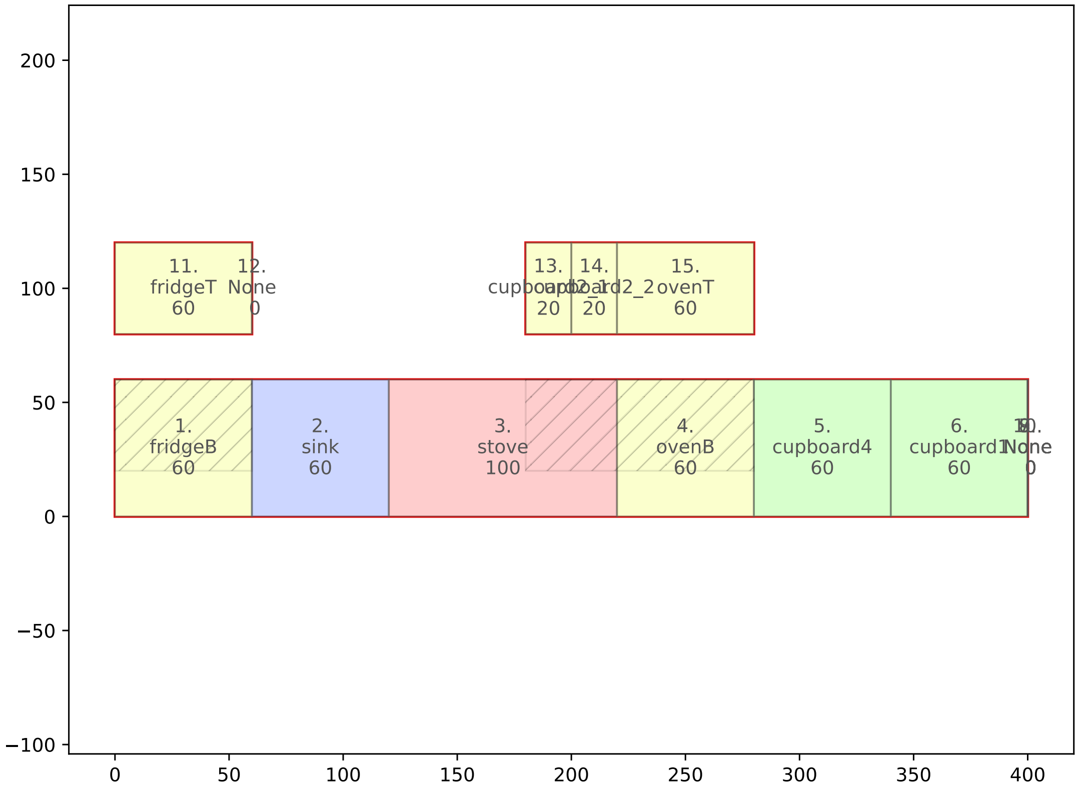


Figure 6.2 Layout of an in-line kitchen with a window

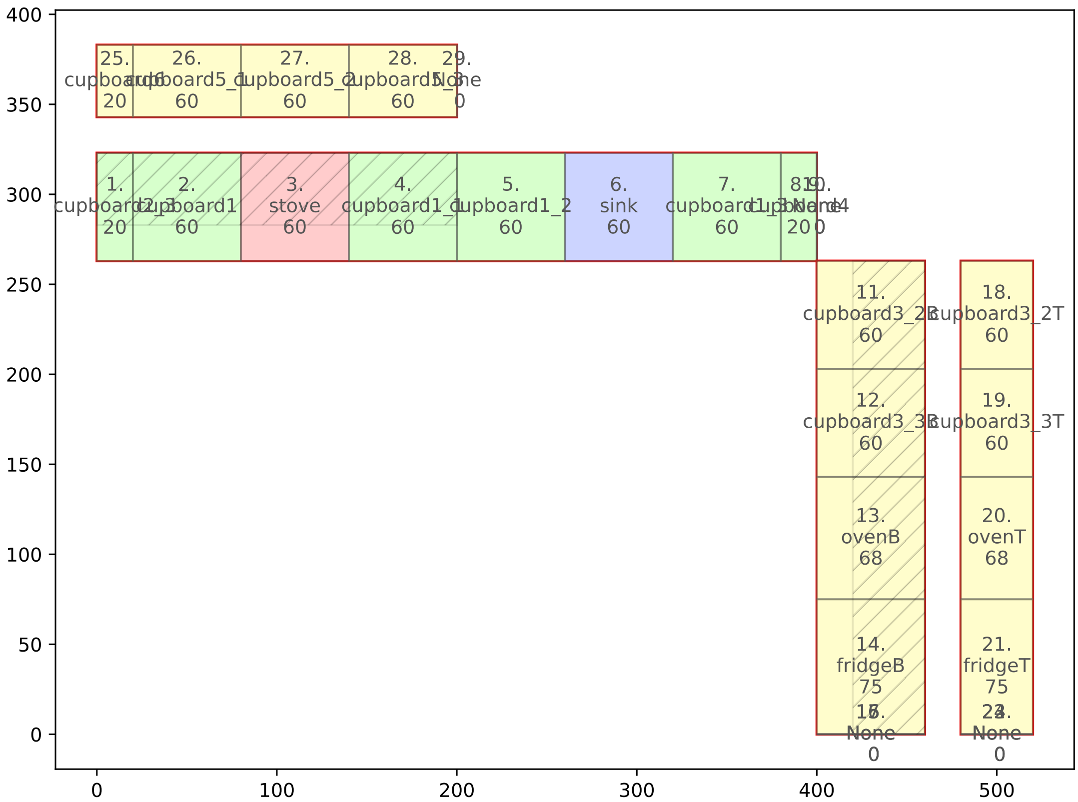
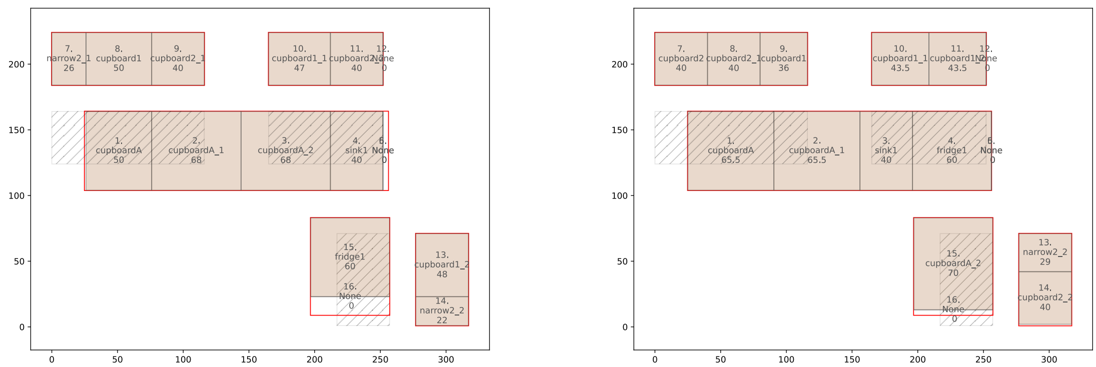


Figure 6.3 Layout of an L-shaped kitchen with a window



(a) with alignment **(b)** without alignment

Figure 6.4 Layouts illustrating the alignment rule

In Figure 6.3, we see the result. There were three placement rules defining the positions of a stove, a sink, and a cabinet belonging to the storage zone – `cupboard3_3` in this layout. The result looks promising: The tall fixtures are all on one side. There are many pairs of adjacent fixtures with the same widths. Also, all the fixtures are aligned.

6.1.5 Discussion of Layout Quality

All the produced layouts quite match our expectations. We can see that the rules we defined and the ways we implemented them directly affect the resulting layouts.

To further improve the results, we could modify the current implementation of the rules or add new constraints and objective function components.

6.1.6 Effect of Individual Design Rules

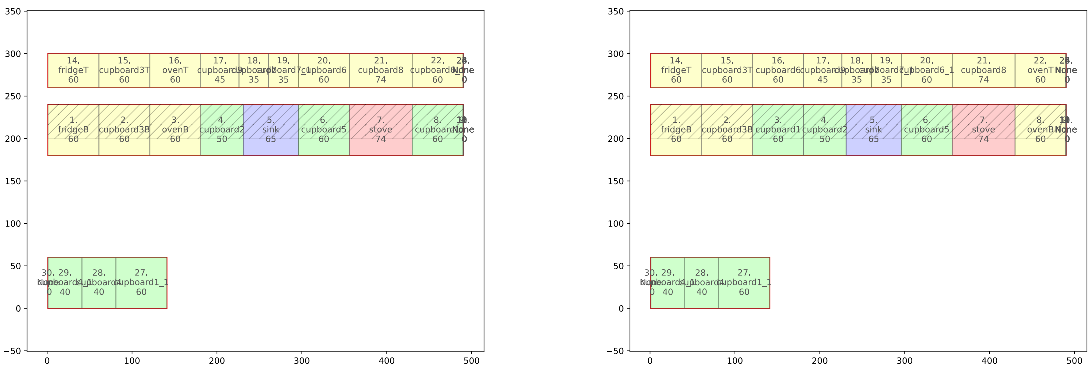
To see if the introduced constraints and components of the objective function have the intended influence on the solutions, we can verify what happens if we disable them. This is very useful when adding new constraints – we can see if they work as expected.

We illustrate that on the alignment of fixtures (vertical continuity); we use an example from the test set (see Section 5.2.2). In Figure 6.4a, we see a layout generated for an input with the vertical discontinuity penalty $\mathrm{vertdisc}^{-}$ is set to one, so the optimal solution has to contain as few discontinuities (misaligned fixtures) as possible. If we set the penalty to zero, the corresponding constraints are disabled and the rule is no longer enforced (see Subsection 5.2.1). The result is shown in Figure 6.4b.

6.1.7 Multiple Optimal Solutions

It may occur that there are several different layouts with the same value of the objective function and that the layouts are all optimal (the value is maximum).

For example, the layouts may differ in the order of fixtures in the kitchen parts as the objective function does not favor any specific ordering (even though duplicated fixtures are sorted using constraints), see Figure 6.5.



(a) MILP model solved by Gurobi   **(b)** MiniZinc model solved by Chuffed

Figure 6.5 Multiple optimal layouts based on a single input file. The value of the objective function equals 2077 in both cases.

If we considered some of the optimal layouts to be worse than the other ones, we could introduce more constraints or extend the objective function.

6.2 Reduction of the Solving Time

We have previously declared that we aim for the shortest possible solving time while preserving the quality of the solutions. In this section, we compare various approaches and measure the time that the solvers actually need to find the optimal solutions.

6.2.1 Comparing Models and Solvers

We compare the performance of several solvers when solving the MILP and MiniZinc models. To solve our MILP model, we use the commercial solver Gurobi [3] and two open-source solvers: SCIP [16] and HiGHS [29].

As for the MiniZinc solvers, we compare two MILP solvers – Gurobi and HiGHS – and two open-source solvers that are based on different principles: the lazy clause generation solver Chuffed [30] and the CP-SAT [2] solver. CP-SAT uses satisfiability (SAT) methods in its core; it also draws its inspiration from the Chuffed solver and MILP solvers [31]. Both Chuffed and CP-SAT support only integer variables.

We also tried to solve the models using a Gecode [32] constraint solver, but it did not perform well, so we excluded it from the comparison.

Gurobi and CP-SAT can use multiple threads in parallel, so apart from the parallel execution (using up to 12 threads), we test the solvers in a single-threaded mode.

Note that there is an important difference between the two models we solve: The MiniZinc model works with integer values only. We compare the performance of the solvers on kitchens from our benchmark set (see Section 5.2.2).

In Figure 6.6, we can see the comparison of the solving times. Gurobi performs really well for both models. Interestingly, Chuffed is a worthy competitor. However, the most surprising result is that CP-SAT outperforms all the solvers –

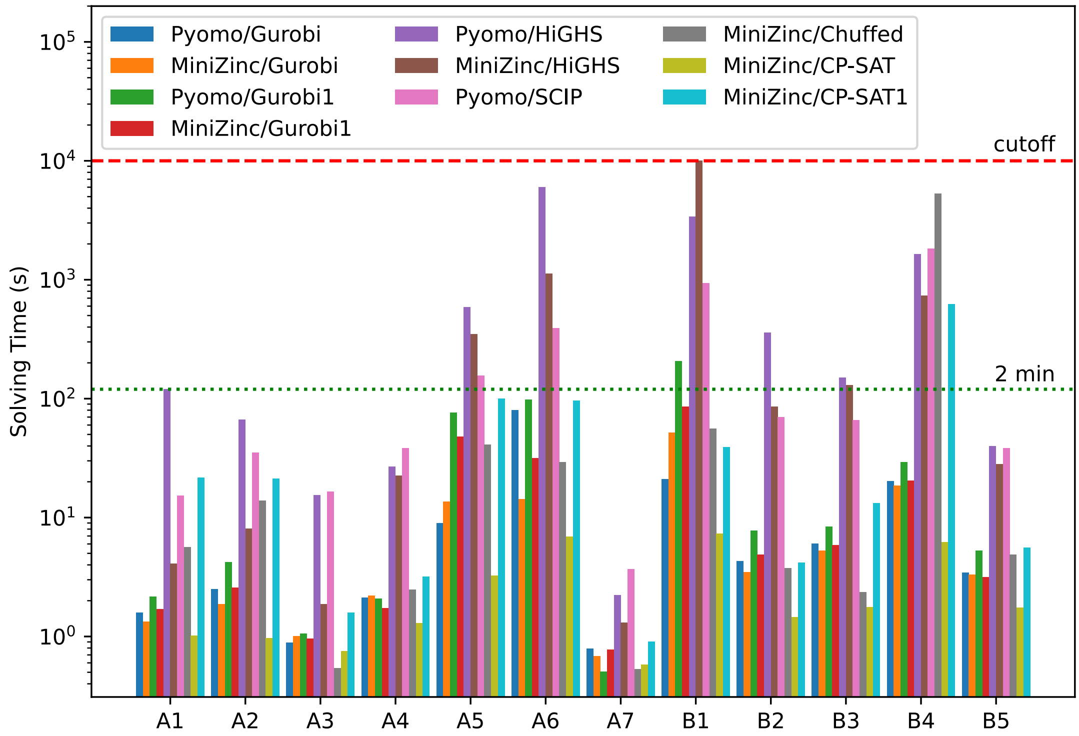


Figure 6.6 Solving times of the two models solved by different solvers. We set the maximum allowed solving time to 10,000 seconds; the maximum time a designer is willing to wait (2 minutes) is also highlighted. Mind the logarithmic scale of the vertical axis. For Gurobi and CP-SAT, single thread runs are marked by 1.

	objective		solving time (s)									
model	P	M	P	M	P	M	P	M	P	M	M	M
solver			G	G	G1	G1	H	H	S	Ch	CP	CP1
A1	658	658	1.6	1.3	2.2	1.7	120.0	4.1	15.3	5.6	1.0	21.6
A2	564	564	2.5	1.9	4.2	2.6	66.9	8.1	35.2	13.9	1.0	21.3
A3	476	476	0.9	1.0	1.1	1.0	15.5	1.9	16.5	0.5	0.8	1.6
A4	621	621	2.1	2.2	2.1	1.7	27.0	22.6	38.3	2.5	1.3	3.2
A5	676	674	9.0	13.6	76.4	48.2	588.7	350.4	156.2	41.3	3.3	100.1
A6	728	728	79.9	14.3	98.1	31.6	6000.5	1128.9	392.8	29.3	6.9	96.6
A7	501	501	0.8	0.7	0.5	0.8	2.2	1.3	3.7	0.5	0.6	0.9
B1	852	852	21.1	51.9	207.6	86.0	3382.0	$> 10^4$	938.1	56.1	7.3	39.3
B2	803	803	4.3	3.5	7.8	4.9	359.1	85.6	70.2	3.8	1.4	4.2
B3	2142	2142	6.0	5.3	8.4	5.9	150.4	130.0	66.2	2.4	1.8	13.2
B4	992	992	20.3	18.5	29.4	20.5	1650.6	735.5	1822.3	5276.5	6.2	622.4
B5	2077	2077	3.4	3.3	5.3	3.2	40.1	28.2	38.6	4.9	1.7	5.6

Table 6.1 Objective values and solving times of MILP (P) and MiniZinc (M) models. We used five solvers to solve the models: Gurobi (G), HiGHS (H), SCIP (S), Chuffed (Ch), and CP-SAT (CP). For Gurobi and CP-SAT, single thread runs are marked by 1.

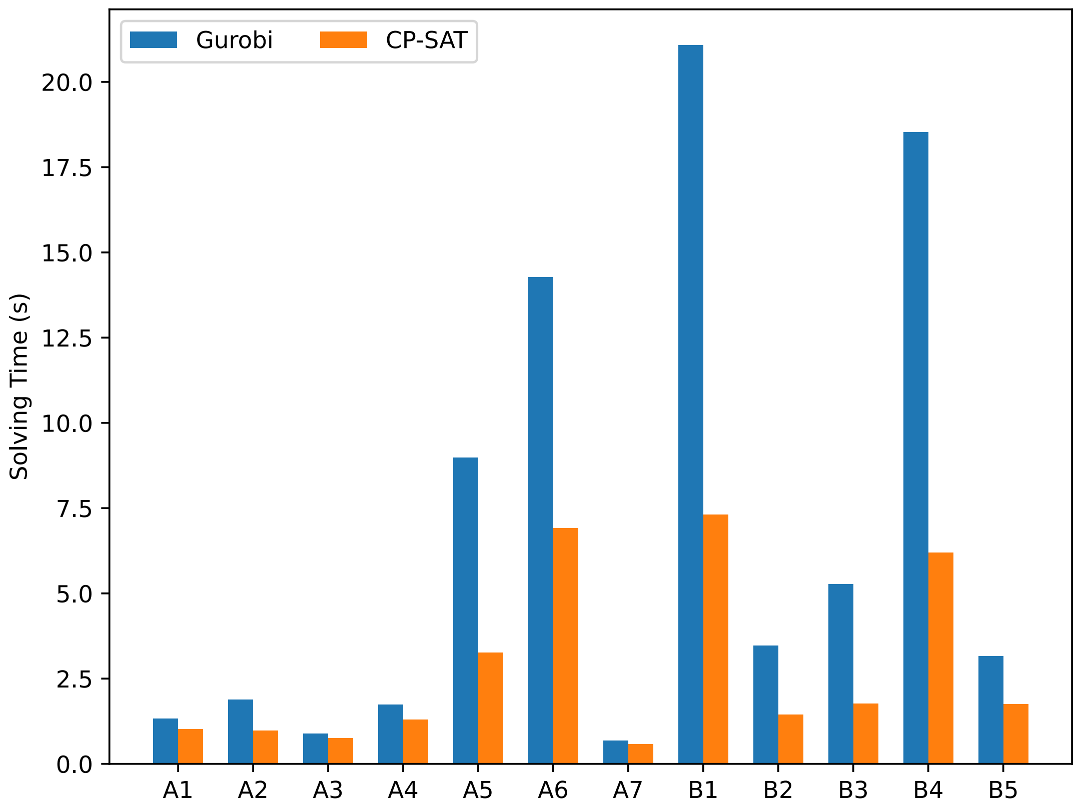


Figure 6.7 Comparison of Gurobi and CP-SAT. Both solvers use multithreading. For Gurobi, we show the minimum solving time achieved on any of the two models.

including Gurobi. For a direct comparison of Gurobi and CP-SAT (both using multithreading), see Figure 6.7.

Note that CP-SAT rarely performs better than Gurobi if the solvers are run on a single thread. This suggests that CP-SAT can make better use of parallelism.

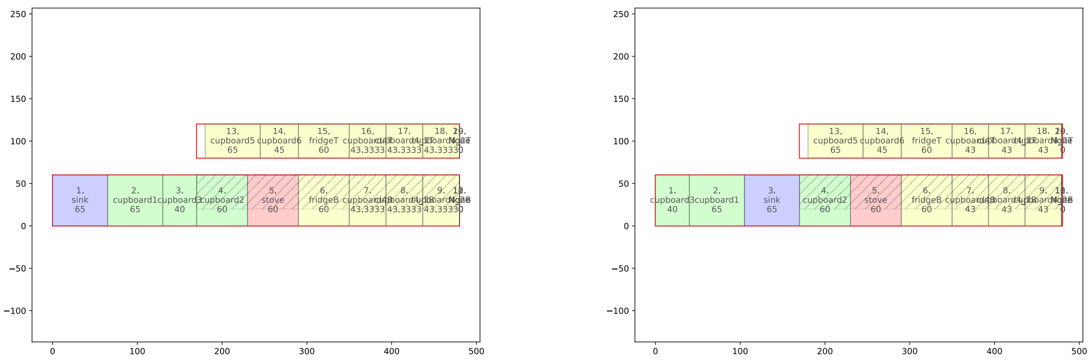
Overall, the MILP and MiniZinc formulations achieve similar solving times when solved by MILP solvers. HiGHS and single-threaded Gurobi mostly perform better when solving the MiniZinc model compared to the MILP model. The difference might be caused by the integer restriction of the MiniZinc model or by the MiniZinc model being compiled to an MILP formulation that is more efficient than ours.

Integer Variables

In Table 6.1, we can see the values of the objective function and the exact solving times. Note that the values of the objective function differ for Kitchen A5 because the MiniZinc model only supports integer widths. See Figure 6.8 to compare the generated layouts.

This limitation of the MiniZinc model can affect the quality of the solutions: For example, if there is a lower fixture 65 cm wide, it might be reasonable to fit two upper fixtures directly above it, each of 32.5 cm. The MiniZinc implementation supports only integer widths, so the fixtures cannot have the same widths and thus will be penalized for the difference.

Although in some cases this may cause the MiniZinc model to generate worse



(a) MILP

(b) MiniZinc

Figure 6.8 Optimal layouts with different objective values due to the integer restriction of the MiniZinc model

layouts than the MILP formulation, we assume that the problem occurs rarely. Also, using millimeter widths or adding a tolerance in the widthNotSame predicate should help.

6.2.2 Deactivating Rules

When discussing the reduction of solving times, it seems natural to consider disabling some rules in order to simplify the model, possibly leading to a decrease in time needed to find an optimal solution.

We try deactivating two rules: a rule demanding that the fixtures are aligned whenever possible and a rule penalizing neighboring fixtures with different widths, and we examine the effect on the solution time.

As shown in Figure 6.9, it really takes longer to find an optimal solution when the mentioned rules (or guidelines) regarding the alignment and widths are active.

Even though we usually do not want to deactivate these rules when designing the layouts, there may be a use case: We could use this “fast mode” to provide the designer with feedback on the input they sent to the program. If the designer makes a mistake when entering the input, they can find out before submitting the final (more complex) model for solution.

6.2.3 Alignment: Binary vs. Piecewise Approach

In Subsection 4.1.2 we proposed two different MILP approaches to achieve that the lower and upper fixtures are aligned. Their results are not identical: Each assigns a penalty based on different criteria; the resulting value of the objective function differs for the same layout. There can even be different optimal layouts for the two formulations.

However, we can still compare the times needed to find the optimal solution using each of the two approaches. In this way, we can decide which approach to focus on and possibly further refine. In all the other comparisons, we use the binary approach; it is now time to see whether it was a wise choice.

In Figure 6.10, we see that even though neither approach is always better than the other, the piecewise formulation seems to be more stable: Whenever one

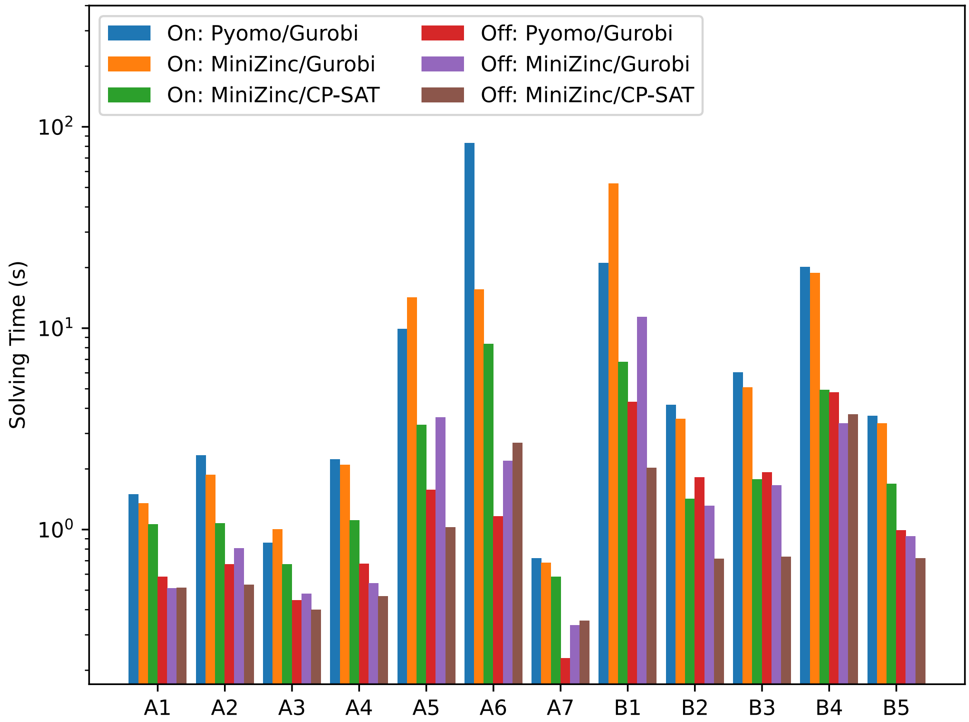


Figure 6.9 MILP and MiniZinc models solved using Gurobi and CP-SAT. There are two variants of the models compared: "On" with the *alignment* and *same width* rules active and "Off" with these rules deactivated. Mind the logarithmic scale of the vertical axis.

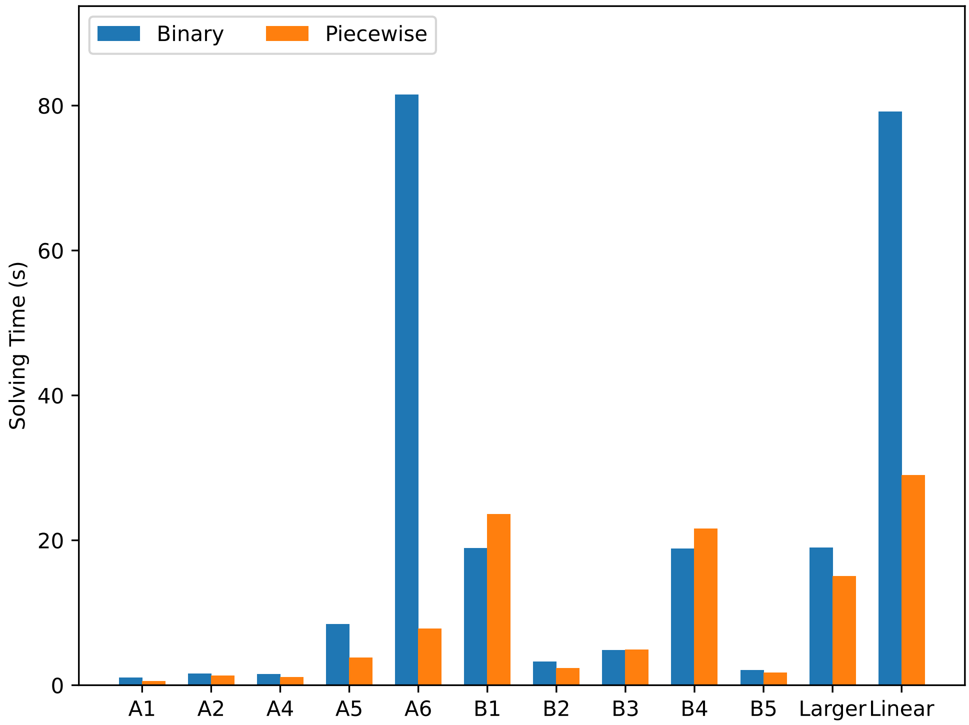
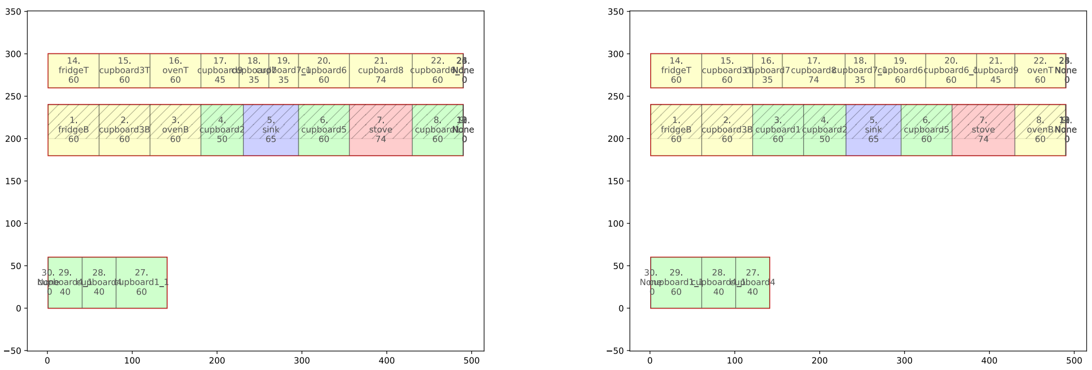


Figure 6.10 Different approaches to the alignment rule. We compared the binary approach and the piecewise approach, both in the MILP model solved by Gurobi. The last two input files come from our test set and were created for testing the alignment rule.



(a) binary **(b)** piecewise

Figure 6.11 The alignment rule formulations – optimal layouts

solving time vastly exceeds the other, it is always the binary approach that is slower.

To illustrate the difference between the two formulations, we show the optimal layouts produced for the input file B5. In Figure 6.11a, the layout is the same as in Subsection 6.1.2 – the input was identical. As shown in Figure 6.11b, the number of vertical discontinuities increases when using the piecewise formulation, because this formulation penalizes only some of the discontinuities: A non-zero penalty is assigned when the offsets of the fixtures differ by a small amount.

Due to this difference between the formulations, some optimal layouts for the piecewise version may be considered worse than the layouts returned when using the binary formulation. To achieve better results, we would need to adjust the constants that the constraint uses.

6.2.4 Discussion of the Performance

We have shown that both our models perform quite well. To achieve the fastest solving times, we should solve the MiniZinc model using the CP-SAT solver. We have also found that the piecewise formulation of the alignment rule can often be solved faster than its binary formulation.

When further reducing the solving times, these are some of the pathways we could take:

- Use the piecewise formulation of the alignment rule in the MILP model, add it to the MiniZinc model.
- Modify the MiniZinc formulation to better support non-MILP solvers; use categorical variables instead of binary pairings.
- Reformulate some of the constraints and rules to make them faster in general – based on our benchmark set.
- Use solver-specific methods to model the constraints.

6.3 Review of the Web Application

As described in Chapter 5, we have created a web application allowing the designer to quickly draft the kitchen schemes, submit them, and receive the solutions – kitchen layouts. It soon became clear that the generality of the user interface may cause trouble to designers: To describe a single placement rule, they have to fill several input fields!

Needless to say that the current web interface is not much usable in practice – designers would argue it takes less time to design the layout manually than to enter the input in the application.

However, our program suits a different purpose: It is meant to assist the developers and designers with exploring the possibilities – what such program could and could not do, how the designer would use it.

Based on this experience, it is then possible to create a new program with a better user interface which is easier to use and tailored to the designers' needs. That program could be connected to other design tools so that the proposed layout can be modified and used during the next phases of the kitchen design process.

Conclusion

To automate the process of kitchen layout design, we have compiled a set of requirements including design rules and guidelines. Then we created a formal model of a kitchen layout and defined the Kitchen Layout Problem. We have shown that the problem is NP-complete and proposed two approaches used to dealing with such problems: mixed integer linear programming and constraint programming.

For each approach, we have described a possible formulation – an implementation of the formal model. We tuned the implementations to generate viable layouts. We also discussed rules that could be added to the model to make it more precise.

We have illustrated how to deploy these MILP and MiniZinc models to an application that can be used via web or command-line interface. Finally, we used the program to generate several layouts; we assessed their quality and the time that the generation took.

Future Work

We have presented formulations generating layouts which follow the most significant design rules, but we left some of the less important guidelines unimplemented, see Section 4.2. Adding them might improve the quality of the proposed layouts. However, it is important to carefully monitor the solving time when doing so.

The time required to generate the layouts is one of our main concerns. We now list several ways of reducing it:

- Some of the constraints could be reformulated and optimized. The alignment rule (Subsection 4.1.2) is one of the candidates. To decide between analogous formulations, the benchmark set (Section 5.2.2) should be used. It is advisable to further extend the benchmark set so that it reliably contains all typical kitchen schemes.

- Our MiniZinc model is based on the MILP formulation. To achieve better results with non-MILP solvers, we can consider making the model more general. We would then benefit from MiniZinc compiling the model into a solver-specific formulation (refer to Subsection 1.4.1).

- If we chose a specific solver we prefer, we could adapt the model for the given solver: use solver-specific methods and settings.

- We could investigate other ways of breaking symmetries. We discussed one type of symmetry in Subsection 3.3.7.

- We could make use of the fact that the layouts do not need to be perfect (see Section 2.1) and progressively return intermediate solutions that the solver finds before identifying the optimal solution.

- For larger kitchens, the number of possible segment and fixture combinations might be unbearable. To solve it, we might modify the model to deal with groups of fixtures instead of individual fixtures, not to divide tall fixtures into two fixtures, or to somehow reuse the fixtures in different places.

- We could mimic the standard approach that designers use: fit the lower and tall fixtures first and then add the upper fixtures.

- We expect that the designer may use our tool iteratively, adjust the input to get the best possible layout. Some solvers allow to reoptimize the modified model based on the previous solution. We could take advantage of this feature.

It is even possible to completely change the paradigm: Our program creates the entire layout at once. There could be a tool that assists the designer by fitting fixtures only in a selected part of the layout. Such a tool might be easier for designers to use as it would not be a groundbreaking change to their routine.

As there are some design rules and criteria that are hard to describe formally and thus almost impossible to include in the model, it might be useful to give the designer and the client multiple layouts to choose from – they could then select the best layout based on the unexpressed criteria. MILP solvers often support finding multiple optimal or near-optimal solutions, MiniZinc also provides a way to get multiple solutions. However, there may be a lot of similarities among these solutions. It would be necessary to ensure that the layouts presented to the designer and the client are diverse, because providing them with many nearly identical layouts would not help. This could be a rather challenging task, as there is no exact way to decide if two layouts are different enough.

Bibliography

1. Nethercote, Nicholas; Stuckey, Peter J.; Becket, Ralph; Brand, Sebastian; Duck, Gregory J.; Tack, Guido. MiniZinc: Towards a Standard CP Modelling Language. In: Bessiere, Christian (ed.). *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*. Springer, 2007, vol. 4741, pp. 529–543. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-540-74970-7_38`.
2. Perron, Laurent; Didier, Frédéric. *CP-SAT*. Google, 2024. Version v9.12. Available also from: `https://developers.google.com/optimization/cp/cp_solver/`.
3. Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2024. Available also from: `https://www.gurobi.com`.
4. Ervum, Johannes Grindal. *Proposing Kitchen Designs: A Case-Based Reasoning Approach*. 2019. Available also from: `https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2628785`. MA thesis. NTNU.
5. Pejic, Jelena; Pejic, Petar. Linear kitchen layout design via machine learning. *Artif. Intell. Eng. Des. Anal. Manuf.* 2022, vol. 36, e9. Available from DOI: `10.1017/S089006042100038X`.
6. Sun, Xin; Ji, Xiaomin. Integrated kitchen design and optimization based on the improved particle swarm intelligent algorithm. *Comput. Intell.* 2020, vol. 36, no. 4, pp. 1638–1649. Available from DOI: `10.1111/COIN.12301`.
7. Hsieh, Poyen; Benros, Deborah; Dogan, Timur. Conversational co-creativity with deep reinforcement learning agent in kitchen layout. In: *Design computing and cognition'20*. Springer, 2022, pp. 399–409.
8. Hsieh, Poyen. *Generative Design: Kitchen Layout Automation*. 2019. Available also from: `https://www.youtube.com/watch?v=P8A6ovlZP7U`.
9. Lee, Jane; Ryu, Eunseo; Kim, Jun Young; Lee, Joong Hee; Yun, Myung Hwan. A simulation approach to validating optimized kitchen layouts: Reassessing the work triangle through real apartment floor plan analysis. *WORK: A Journal of Prevention, Assessment & Rehabilitation*. 2025, p. 10519815241300413. ISSN 1051-9815, ISSN 1875-9270. Available from DOI: `10.1177/10519815241300413`.
10. IKEA. *IKEA Kitchen Planner*. [N.d.]. Available also from: `https://kitchen.planner.ikea.com/cz/en/`.
11. Popa, Vecislavas. 2020. Available also from: `https://www.pexels.com/photo/glass-dining-table-against-hi-tech-kitchen-3741316/`.
12. Baden-Powell, Charlotte. *Architect's Pocket Book of Kitchen Design*. Oxford: Routledge, 2005. ISBN 978-0-7506-6132-4.
13. Gärtner, Bernd; Matoušek, Jiří. *Understanding and using linear programming*. Springer, 2007. Universitext. ISBN 978-3-540-30697-9.

14. Karmarkar, Narendra. A new polynomial-time algorithm for linear programming. *Comb.* 1984, vol. 4, no. 4, pp. 373–396. Available from doi: 10.1007/BF02579150.

15. Karp, Richard M. Reducibility Among Combinatorial Problems. In: Miller, Raymond E.; Thatcher, James W. (eds.). *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Plenum Press, New York, 1972, pp. 85–103. The IBM Research Symposia Series. Available from doi: 10.1007/978-1-4684-2001-2_9.

16. Bolusani, Suresh; Besançon, Mathieu; Bestuzheva, Ksenia; Chmiela, Antonia; Dionísio, João; Donkiewicz, Tim; Doornmalen, Jasper van; Eifler, Leon; Ghannam, Mohammed; Gleixner, Ambros; Graczyk, Christoph; Halbig, Katrin; Hedtke, Ivo; Hoen, Alexander; Hojny, Christopher; Hulst, Rolf van der; Kamp, Dominik; Koch, Thorsten; Kofler, Kevin; Lentz, Jurgen; Manns, Julian; Mexi, Gioni; Erik Mühmer; Pfetsch, Marc E.; Schlösser, Franziska; Serrano, Felipe; Shinano, Yuji; Turner, Mark; Vigerske, Stefan; Weninger, Dieter; Xu, Lixing. *The SCIP Optimization Suite 9.0*. 2024. Technical Report. Optimization Online. Available also from: https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/.

17. Bisschop, J. *Aimms Optimization Modeling*. Paragon Decision Technology, 2006. isbn 9781847539120.

18. Amer, Abdelmonem Mahmoud. *Express boolean logic operations in zero-one integer linear programming (ILP)* [Computer Science Stack Exchange]. 2015. Available from eprint: https://cs.stackexchange.com/q/43884.

19. Apt, Krzysztof R. *Principles of constraint programming*. Cambridge University Press, 2003. isbn 978-0-521-82583-2.

20. Apt, Krzysztof R. The Essence of Constraint Propagation. *Theor. Comput. Sci.* 1999, vol. 221, no. 1-2, pp. 179–210. Available from doi: 10.1016/S0304-3975(99)00032-8.

21. Stuckey, Peter J. Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving. In: Lodi, Andrea; Milano, Michela; Toth, Paolo (eds.). *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*. Springer, 2010, vol. 6140, pp. 5–9. Lecture Notes in Computer Science. Available from doi: 10.1007/978-3-642-13520-0_3.

22. Belov, Gleb; Stuckey, Peter J.; Tack, Guido; Wallace, Mark. Improved Linearization of Constraint Programming Models. In: Rueher, Michel (ed.). *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*. Springer, 2016, vol. 9892, pp. 49–65. Lecture Notes in Computer Science. Available from doi: 10.1007/978-3-319-44953-1_4.

23. Kratochvílová, Kateřina. *Studie postupů návrhu interiéru v kuchyňských studiích [online]*. 2024. Available also from: https://theses.cz/id/kte4x1/. SUPERVISOR: Ing. Adam Kořený, Ph.D.

24. Kleinberg, Jon M.; Tardos, Éva. *Algorithm design.* Addison-Wesley, 2006. isbn 978-0-321-37291-8.

25. Pfetsch, Marc E.; Rehn, Thomas. A computational comparison of symmetry handling methods for mixed integer programs. *Math. Program. Comput.* 2019, vol. 11, no. 1, pp. 37–93. Available from doi: `10.1007/S12532-018-0140-Y`.

26. Wolsey, L.A. *Integer Programming.* Wiley, 1998. Wiley Series in Discrete Mathematics and Optimization. isbn 9780471283669.

27. Bynum, Michael L.; Hackebeil, Gabriel A.; Hart, William E.; Laird, Carl D.; Nicholson, Bethany L.; Siirola, John D.; Watson, Jean-Paul; Woodruff, David L. *Pyomo–optimization modeling in python.* Vol. 67. Third. Springer Science & Business Media, 2021.

28. Hart, William E; Watson, Jean-Paul; Woodruff, David L. Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation.* 2011, vol. 3, no. 3, pp. 219–260.

29. Huangfu, Qi; Hall, J. A. J. Parallelizing the dual revised simplex method. *Math. Program. Comput.* 2018, vol. 10, no. 1, pp. 119–142. Available from doi: `10.1007/S12532-017-0130-5`.

30. Chu, Geoffrey; Stuckey, Peter J.; Schutt, Andreas; Ehlers, Thorsten; Gange, Graeme; Francis, Kathryn. *Chuffed, a lazy clause generation solver.* 2018. Available also from: `https://github.com/chuffed/chuffed`.

31. Perron, Laurent; Didier, Frédéric; Gay, Steven. The CP-SAT-LP Solver. In: Yap, Roland H. C. (ed.). *29th International Conference on Principles and Practice of Constraint Programming (CP 2023).* Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, vol. 280, 3:1–3:2. Leibniz International Proceedings in Informatics (LIPIcs). isbn 978-3-95977-300-3. issn 1868-8969. Available from doi: `10.4230/LIPIcs.CP.2023.3`.

32. Team, Gecode. *Gecode: Generic Constraint Development Environment.* 2006. Available also from: `http://www.gecode.org`.

List of Figures

1 Kitchen layout sketch 9

1.1 Types of fixtures 12

3.1 Kitchen parts and kitchen groups 22

4.1 Tall fixture in the middle of a kitchen part 36
4.2 Illustration of alignment 38
4.3 Two narrow upper fixtures above a wide lower fixture 40
4.4 Discontinuity penalty as a function of offset difference 40

6.1 Layout of an in-line kitchen with an island 51
6.2 Layout of an in-line kitchen with a window 52
6.3 Layout of an L-shaped kitchen with a window 52
6.4 Layouts illustrating the alignment rule 53
6.5 Multiple optimal layouts based on a single input file 54
6.6 Solving times of the two models solved by different solvers 55
6.7 Comparison of Gurobi and CP-SAT 56
6.8 Optimal layouts with different objective values due to the integer restriction of the MiniZinc model 57
6.9 MILP and MiniZinc models with specific rules inactive or active 58
6.10 Different approaches to the alignment rule 58
6.11 The alignment rule formulations – optimal layouts 59

A.1 Entering the kitchen scheme in the web application 68
A.2 Viewing the generated kitchen layout in the web application 69

A Attachments

A.1 Electronic Attachment

The electronic attachment contains the Python application described in Chapter 5 and related files:

- `kitchendesigner/` – Python application code,
- `sketcher/` – Svelte-based web interface,
- `docker/` – configuration of Docker containers,
- `migrations/` – database migrations,
- `experiments/` – experiments performed in Chapter 6,
- `tests/` – benchmark set and test set,
- `docs/` – user documentation,
- `requirements.txt` – Python requirements file,
- `.env.example` – template for a `.env` file,
- `compose.yaml` – Docker Compose file,
- `LICENSE` – software license.

A.2 User Documentation

As we have described in Chapter 5, there are two interfaces that can be used to generate layouts: a web application and a command-line interface.

A.2.1 Web Application

Forms

In its core, the web tool only acts as a way to load input files into the program and retrieve the output files. All the files use the JSON format. On the right-hand side of the webpage (see Figure A.1), there are four tabs:

1. input form,
2. input JSON file,
3. output form,
4. output JSON file.

The forms display the JSON files in a user-friendly way.

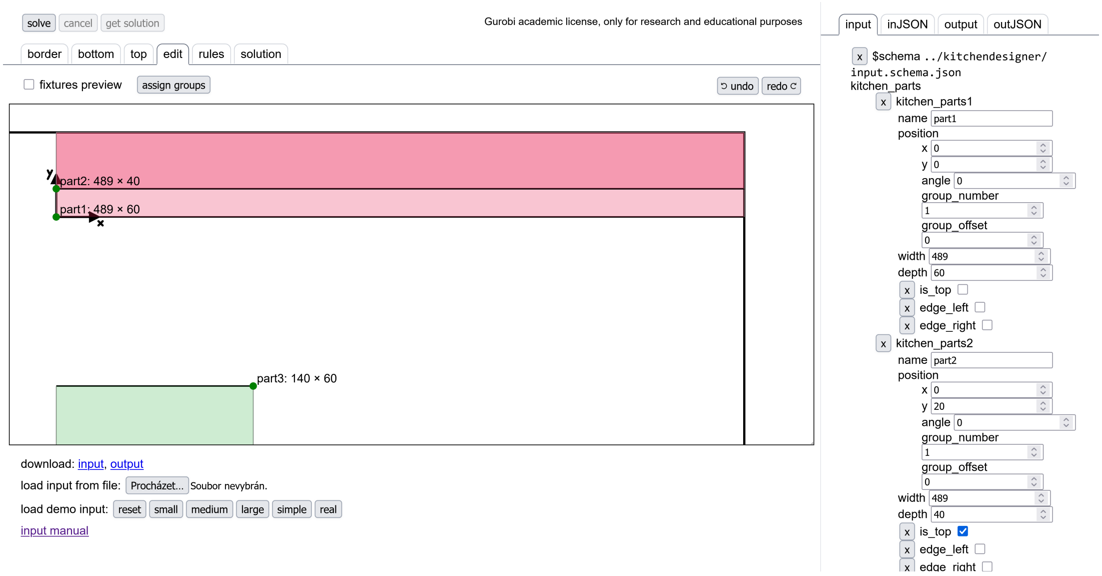


Figure A.1 Entering the kitchen scheme in the web application

Canvas

In the middle of the page, there is a canvas that suits for viewing both the input kitchen scheme and the output layout of the fixtures. It is also possible to modify the position and orientation of the kitchen parts there.

To simplify the manipulation with the objects on the canvas, there are tabs dividing the whole process of working with the web application into six stages:

1. drawing the border (walls) of the kitchen,
2. adding the lower kitchen parts,
3. adding the upper kitchen parts,
4. checking that the kitchen groups were assigned correctly,
5. viewing the placement rules,
6. viewing the final layout.

The border is a polygonal chain (several connected line segments, polyline). Its vertices can be moved with a mouse. When the *add new* checkbox is checked, a new vertex can be added by clicking the canvas. The border is not used by the solver; it is only a piece of visual information displayed to the designer.

To add a lower or upper kitchen part, it is necessary to activate the *add new* checkbox. Otherwise, they can be moved and resized by selecting them and dragging their vertices. When a kitchen part is selected, its data are highlighted in the input form.

In the *edit* stage, the kitchen parts in the same kitchen group have the same colors. If the groups were assigned incorrectly, it is possible to correct them in the input form by changing the group numbers. The green circles mark the beginning of the kitchen group as they were calculated from offsets of the kitchen parts and should be all at one side of the group. If the kitchen parts are modified through the canvas, the kitchen groups and offsets are recalculated.

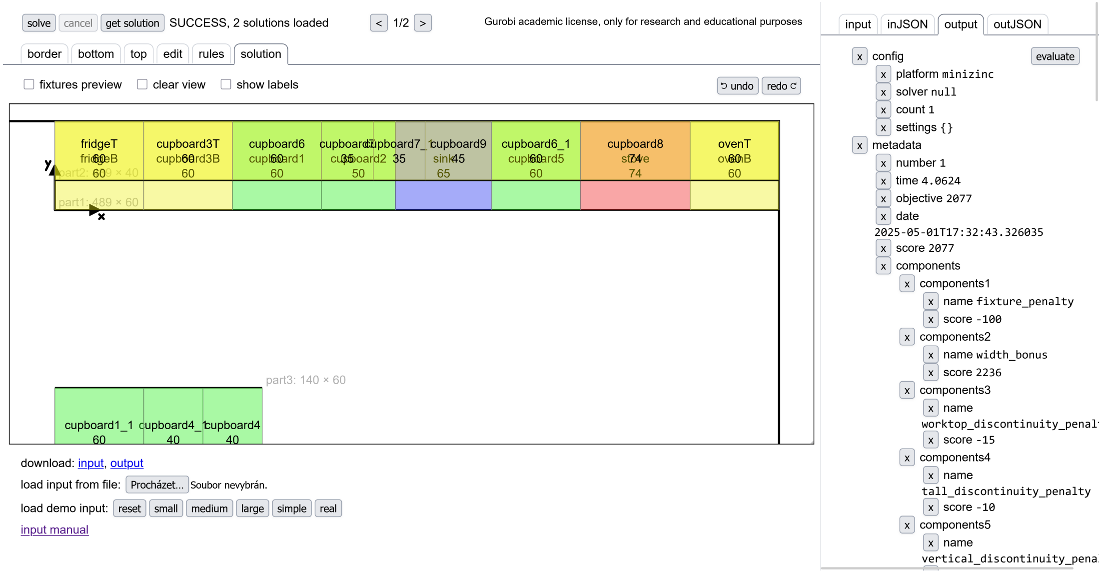


Figure A.2 Viewing the generated kitchen layout in the web application

The fixtures can be entered solely through the input form on the right. However, it is possible to view them on the canvas by activating the *fixture preview* checkbox. Each fixture has to belong to a kitchen zone; these can be defined in the input form as well.

Similarly, the constants and preferences can be set through the form. The most important constant is `avg_fixture_width` as it notably reduces the solving time when set correctly. It directly determines the maximum number of fixtures that can be fit in each kitchen part, see Subsection 3.2.1 and the constant avgw.

After typing the placement rules in the input form, you can view them in the *rules* mode. However, they are not editable on the canvas.

It is possible to generate the layout using a platform and solver that are different than the default ones (Pyomo, Gurobi) or to use multiple solvers at once. This can also be configured in the input form.

Controls

You can load the input file from a local JSON file or load the demo input using the buttons at the bottom of the page. The current input and output files can be downloaded. If you make a mistake when editing the input, you can use the buttons above the upper right corner of the canvas to navigate the history of changes.

To send the input to be solved, you can click the *solve* button in the upper left corner. Immediately, your kitchen scheme is sent to the server to be solved. After some time, the solution is automatically fetched from the server and displayed on the canvas (see Figure A.2).

After submitting the input file, you are assigned an URL address pointing to the kitchen scheme. It is even possible to save the address and open it later if the solving takes too long. You can also *cancel* the solving if you change your mind. Even though the solutions are fetched automatically, you can press the button *get solution* to check for the solutions manually.

Solution

When viewing the generated layout, you can select the *clear view* checkbox to shift the upper kitchen parts so that the view of the lower kitchen parts is clear.

There is also an option to show labels that were assigned to different kitchen parts or segments by the evaluation function. To read the contained information, click on the labels. If the evaluation function detects any errors, they are shown in the form of red labels that are always visible.

The evaluation function can be run manually on the input and output data at any given time. You can change the output file in the web interface and let it be evaluated using the *evaluate* button.

If multiple solutions were returned by the server, you can switch between them using the arrows on top of the page.

A.2.2 Command-Line Interface

Setup

To run the command-line tool, you first need to install the required Python packages using the following command:

```
pip install -r requirements.txt
```

To use MILP solvers, you have to download them and make sure that Pyomo can access them. The Gurobi solver is contained in its Python package `gurobipy`, you only need to provide the license file.

To use MiniZinc, you need to install it from the official website[1]. Most of the MiniZinc solvers are included in the package.

Running an Experiment

The main way to use the program is by running experiments. An experiment is usually described by a single JSON file, but the file can be overridden by command-line flags. You can even run an experiment described only by the flags without any experiment file.

These are the supported flags:

- `-x`, `--experiment` followed by a path of an experiment file,
- `-i`, `--input` with one or multiple paths to input files or directories containing input files,
- `-o`, `--output` defining the output directory,
- `-s`, `--solver` setting the platform and solver as `platform.solver`,
- `-m`, `--model`, if used, the Pyomo model is written into a file,
- `-nw`, if used, the layout is not displayed in a window.

The program may then be used as follows:

```
python kitchendesigner -i input.json -o solutions -nw
```

[1] `https://www.minizinc.org`

Other Features

There are also two other flags that can only be used together with a single input file. To assess a given solution using the evaluation function, run this command:

```
python kitchendesigner -i <input_file> -e <output_file>
```

To debug the MILP model by fixing the values of variables (see Subsection 5.2.3), you can use the following command:

```
python kitchendesigner -i <input_file> -f <output_file>
```

A.2.3 File Formats

For all the three types of files we provide JSON Schema[2] descriptions. Input files are validated against the schema.

Input File

An input file is a JSON object that may contain the following fields:

- `constants` – an object containing constants (required),
- `preferences` – an object with preferences (optional),
- `kitchen_parts` – a list of kitchen parts (required),
- `zones` – a list of kitchen zones (required),
- `available_fixtures` – a list of fixtures (required),
- `placement_rules` – a list of placement rules (optional),
- `solve` – a list of solving instructions (optional),
- `tests` – a list of tests used to evaluate the solutions in the command-line tool (optional),
- `visual` – an object used exclusively by the web application to display additional visual information (optional).

As the MILP model was originally larger, there are some (optional) fields that are supported but not currently in use. We ignore these fields in this documentation.

These are the **constants** that can be specified:

- `min_fixture_width` – a minimum allowed width of a fixture (optional, inferred from minimum widths of individual fixtures),
- `max_fixture_width` – a maximum allowed width of a fixture (optional, inferred from maximum widths of individual fixtures),

[2] `https://json-schema.org`

- `avg_fixture_width` – an average width of a fixture, affects the number of segments in a kitchen part (optional but recommended, by default equal to the value of `min_fixture_width`),
- `multiple_fixture_copy_count` – a number of duplicates that is created for fixtures selected to be duplicated (3 by default).

The following **preferences** with numeric values (modifying the objective function) can be stated:

- `width_bonus`,
- `fixture_penalty`,
- `vertical_discontinuity_penalty`,
- `worktop_discontinuity_penalty`,
- `width_not_same_penalty`.

When using the Chuffed solver, the values of the preferences have to be integers.

Also, to activate the piecewise formulation of vertical continuity, we can set the preference `vertical_discontinuity_alternative` to `true`. To only look at the upper fixtures when penalizing different widths of neighboring fixtures, we can set the preference `width_not_same_only_top` to `true`.

A **kitchen part** can have the following fields:

- `name` of the kitchen part (required, should be unique),
- `width` (required),
- `depth` (required),
- `position` – an object containing its `x` and `y` coordinates, `angle` of rotation (in degrees), `group_number` specifying the number of its kitchen group (can be any positive integer) and `group_offset` (all is required),
- `is_top` – Boolean value indicating if it is an upper kitchen part (optional),
- `edge_left` and `edge_right` (both optional).

Every **zone** should have a unique `name` and a `color` (HEX color code).

Each **fixture** can have the following fields:

- `name` of the fixture (required, should be unique),
- `type` – a more general descriptor of the fixture, can be used to select multiple similar fixtures in position rules (required),
- `zone` – name of the kitchen zone the fixture belongs to,
- `width_min` (required),
- `width_max` (required),

- position_top and position_bottom Boolean fields – both of them are true for tall fixtures, otherwise only one of them is true (for upper fixtures, only position_top should be true),
- has_worktop – a Boolean field (optional),
- allow_edge_left, allow_edge_right (both optional),
- allow_multiple – a Boolean field indicating that the fixture should be duplicated (optional).

Every placement rule can have the listed fields:

- rule_type – exclude or include (required),
- area – kitchen, group, or group_section (required),
- attribute_name – indicates the attribute that will be used to select the affected fixtures, possible values: name, type, zone, height (required),
- attribute_value – states the value of the attribute used to select the affected fixtures (required); if the attribute height is used, we can select either tall or other;
- for group and section rules, we need to specify group,
- for section rules, we also specify section_width and section_offset.

An object of **solving instructions** can contain information about

- platform – pyomo or minizinc (required),
- solver – gurobi, highs, scip, chuffed, or cp-sat (optional),
- count – number of requested solutions (optional, 1 by default),
- settings – other solver-specific settings (optional); some solvers support setting a number of threads that can be used for the computation (it has to be provided to achieve multithreading in MiniZinc), it can be defined using a threads field.

Output File

An output file consists of three objects: config, metadata, and solution.

The config object contains the solving instructions that were used to obtain the solution. In the metadata object, there is a number of the solution (if the solver returned multiple solutions, it can be greater than 1), the time needed to find the solution, the value of the objective function, the time and date of generating the solution, the score returned by the evaluation function, and components of the evaluation function.

The solution object describes the fitting of the fixtures in the kitchen parts. Each field in the object corresponds to a single kitchen part identified by its name. There is a padding indicating the left padding of the kitchen part. Then there is a list of fixtures. Each has a name, an original_name, and a width. The original_name corresponds to the name stated in the input file. It is possible that multiple output fixtures map to a single input fixture (due to the duplication and the division of tall fixtures).

Experiment File

The experiment file may contain the following fields (all optional):

- input – a list of input files or directories of input files (a leading underscore causes the file or directory to be ignored),
- output – a path to an output directory,
- benchmark_repetitions – how many times each model should be solved; used to achieve better quality of the benchmarks (the times are averaged over all attempts);
- report – an object specifying all the information that should be reported or saved,
- common – an object used to replace some portions of the input files; any field we specify here will be replaced in the input files;
- variant_a and variant_b – can be used for A/B testing, function similarly to common,
- variants – a list of named variants (keys are the names, values are the replacement objects).

The **report** object may contain the following optional Boolean fields (the first two are true by default):

- json – the JSON output should be saved,
- window – the layout should be displayed in a window,
- image – the layout should be saved as an image,
- model – the Pyomo model should be saved in a file.