German University in Cairo

CSEN 906 Constraint Programming

Residential Buildings Layouts Generator

Mohamed Adel 40-2175

Zeyad Khattab 40-13106

January 4, 2021

Contents

1 Introduction

1.1 Motivation

Spatial Layout is a problem encountered in many real world domains. Such domain is building floor plans in Architecture, which is usually done manually, where an architect aims typically to divide a specific area into different high level components, and try to find suitable placements for some sort of low level modules inside these high level components.

In the frame of a residential building, these high level components could correspond to apartments, corridor, stairs and elevator rooms, and may also include units like ducts (next to modules like kitchens or bathrooms), while the low level modules include typically different room types such as bedrooms, kitchens, bathrooms, living room, sun-room, dressing room, hallways or corridors inside an apartment, etc.

1.2 Use Case

In the modern day world everything is done through automated processes to facilitate the work of individuals. In particular, architects are responsible for the task for Residential Buildings Layouts. In this report, a constraint programming approach is provided that automates the process for the architects producing possible layout results given a set of constraints.

1.3 Problem Definition

Given a set of constraints on the different apartments in a residential floor the constraint program is expected to output a possible layout that respect these constraints. For example, a straightforward constraint would be that no two rooms should overlap.

2 Implementation

The implementation used in this report uses Google's OR-tools library in python. In particular, OR-tools CP-SAT solver[1] is used. The way integer and boolean variables are initialized is as follows:

OR-tools demo

```
from ortools.sat.python import cp_model

example_integer = model.NewIntVar(LOWER_BOUND, UPPER_BOUND, NAME)
example_boolean = model.NewBoolVar(NAME)
```

Where `LOWER_BOUND` and `UPPER_BOUND` define the domain of the integer. And the variable `NAME` reflects the name of the variable that is to be used in the model.

2.1 Modelling

The implementation provides a way to plan the layout of a single floor in a building. The floor consists of a group of apartments and some other high level modules. Each apartment is a set of rooms, where each room is one of the types listed in section 2.3 as well as a number of corridors. The high level modules includes elevator, stair, ducts, and floor corridors. In the report, we use the term "rooms" to refer to both actual rooms that exist in an apartment as well as the high level modules mentioned.

2.2 Classes

2.2.1 Rectangle

All rooms are modelled as rectangles, thus a `Rectangle` class is implemented to abstract all the details of a room. The main instance variables of a rectangle are the `(x,y)` coordinates of the top left corner and the bottom right corner. In addition to that, multiple instance variables are added that are particular to a room in our design. These instance variables include the room_type, the room that is adjacent to a room, min area, length and width. Variables which are not known beforehand, and thus depend on the model, are of type $IntVar$ and $BooleanVar$. All the rooms exist in the continuous (x, y) plane where x is the horizontal axis and y is the vertical axis. The x coordinate is referred to as column (col) while the y coordinate is referred to as row.

Rectangle Constructor Definition

```
Rectangle(self, room_type, min_area=1, width=0, height=0, adjacent_to=-1, apartment=-1)
```

2.2.2 Solution Printer

Solution Printer is a class that is used to print the solutions found using the function
$SearchForAllSolutions(model, SolutionPrinter)$

2.3 Enums

Enums Used

```
class Equality(Enum):
    GREATER_THAN = 0
    LESS_THAN = 1
    NONE = 2

class Room(Enum):
```

[1] *CP-SAT Solver*. URL: `https://developers.google.com/optimization/cp/cp_solver`.

```
    DININGROOM = 1
    KITCHEN = 2
    MAIN_BATHROOM = 3
    MINOR_BATHROOM = 4
    DUCT = 5
    DRESSING_ROOM = 6
    BEDROOM = 7
    SUNROOM = 8
    CORRIDOR = 9
    STAIR = 10
    ELEVATOR = 11
    LIVING_ROOM = 12
    OTHER = 13

class FloorSide(Enum):
    LANDSCAPE = 1
    OPEN = 2
    NONE = 3
```

The `Equality` Enum is used in operations to signify if the constraint should be greater/less than some variable. The `Room` Enum is used to represent the different types of room available in our design. The `FloorSide` Enum is used to represent the different possibilities of a floor side.

2.4 Constants

```
    IGNORE_REACHABILITY = [Room.DUCT]
    ROOM_TYPE_MAP = {'Room.DININGROOM': 'DR', 'Room.KITCHEN': 'KT', 'Room.MAIN_BATHROOM'
        : 'MB', 'Room.MINOR_BATHROOM': 'mb', 'Room.DRESSING_ROOM': 'DRS', 'Room.BEDROOM'
        : 'BD', 'Room.SUNROOM': 'SR', 'Room.CORRIDOR': 'C', 'Room.DUCT': 'D', 'Room.
        STAIR': 'S', 'Room.ELEVATOR': 'E', 'Room.LIVING_ROOM': 'LR', 'Room.OTHER': 'X'}

    UNIMPORTANT_ROOMS = [Room.CORRIDOR, Room.DUCT, Room.ELEVATOR,
            Room.STAIR, Room.MAIN_BATHROOM, Room.MINOR_BATHROOM]
```

`IGNORE_REACHABILITY` is used to signify that this room should not be considered while getting the reachability of rooms. `ROOM_TYPE_MAP` is used in the visualization method to map each room type to a shorter acronym to be displayed on screen. `UNIMPORTANT_ROOMS` signifies the rooms that should not be considered in some of the constraints such as the sunlight and landscape constraints for example.

3 Constraints

Constraints is the heart of our implementation. Every criteria that the user specifies in the floor design is mapped to some constraint that the program must satisfy. The constraints in our implementation are classified into three types: hard constraints, soft constraints, and optional hard global view constraints. Hard constraints must be met by definition, but soft constraints are constraints which are preferable to be met, but not necessary. And optional hard global view constraints are to be met just as the hard constraints, however they are not always enabled unless the user requests them.

3.1 Helper Constraints

Helper Constraints are constraints that are not requested by the user in particular, but they occur as a sub-problem in the required constraints

3.1.1 2D Grid

To facilitate the implementation of some of the constraints, a 2D grid of variables was created that indicates what room occupies this cell. In particular, given an array that contains all rooms in the floor, every cell in the array contains the index of the room that occupies this cell or -1 if it is empty. This constraint is implemented using reification, where for every cell (x, y), we say that it lies in room R if $start\_col \leq x < end\_col$ and $start\_row \leq y < end\_row$. If the cell (x, y) is contained in room R then $grid[x][y] =$ index of Room R

3.1.2 Reachability Constraints

One of the constraints the user is interested in is whether a certain room is exposed to the sunlight or landscape views. These constraints are modelled as reachability constraints. The output of this constraint is a 2D array of size $FLOOR\_WIDTH$ x $FLOOR\_LENGTH$ where each cell is boolean variable indicating whether a certain cell is reachable to sun/landscape view. To implement these constraints, we start by saying that the cells that lie on the border have direct access to sun/landscape view if this side is open/has landscape view respectively. A cell is accessible from the sun either if it is directly facing the sun or is adjacent to a cell that is empty or a cell of the same room, thus not blocking the sun. A room is accessible to the sun if one its cells is accessible to the sun. Similarly, the reachability of a landscape view is modelled in the same exact way.

3.1.3 Adjacency

One of the constraints that is used in different constraints throughout the implementation is the adjacency constraint. So to ensure that `ROOM_A` and `ROOM_B` are adjacent, we check if the both rooms share some rows and columns. This constraint is satisfied by ensuring that the intersection of the pair of intervals `([ROOM_A.START_ROW, ROOM_A.END_ROW]` and `[ROOM_B.START_ROW, ROOM_B.END_ROW])` and `([ROOM_A.START_COLUMN, ROOM_A.END_COLUMN]` and `[ROOM_B.START_COLUMN, ROOM_B.END_COLUMN])` are non-empty.

3.2 Hard Constraints

3.2.1 Rooms and Apartments Should Not Overlap

This constraint is implemented using the CP Sat function *AddNoOverlap2D* which accepts a pair of intervals on two dimensions (x and y). The intervals are provided using the set of `START_ROW`, `END_ROW`, `START_COL`, `END_COL` for all the rooms that are specified in the input.

3.2.2 Corridors That Make All Rooms Accessible

A room is said to be accessible if it can be reached directly from a room or through a shared corridor. If this constraint is achieved as well as ensuring that corridors have a connected path, then an apartment is said to be a connected component. Meaning, it does not consist of disjoint sets of scattered rooms. To ensure that this constraint is met, we first assume that there is a main corridor that has sub-corridors connected to it. So, we apply the adjacency constraint on each sub-corridor as such `add_adjacency_constraint(MAIN_CORRIDOR, SUB_CORRIDOR_I)`. Once that connection is added, we enforce that each room in the apartment is connected to at least one of the corridors. These constraints added together form the connected component that we call an apartment.

3.2.3 Corridors That Make All Floor Units Connected

Similar to the apartment corridors, floor corridors are used to form a connected component that consists of the elevators, stairs, and the sub-connected components formed by the apartments. We also assume that there is a main corridor that is connected to all the sub-corridors. Then we make sure that the main corridor of each apartment is connected to at least one of the floor corridors `add_adjacency_constraint(MAIN_CORRIDOR_APARTMENT_I, FLOOR_CORRIDOR)` where `FLOOR_CORRIDOR` is one of the floor corridors. Finally, we apply the same constraint on the stairs and elevator of the floor.

3.2.4 Floor Utilization

Using the grid result from 3.1.1 a constraint is added that `grid[i][j] != -1` for all $0 \leq i < FlOOR\_LENGTH$ and $0 \leq j < FlOOR\_WIDTH$. This enforces that all unit cells contribute to the floor.

3.2.5 Kitchens and Bathrooms Should Be Adjacent To Ducts

The apartment kitchens and bathrooms needs to be adjacent to floor ducts, so to enforce that we add that each kitchen or bathroom needs to be adjacent to at least one duct. But that brings us to another question, wouldn't ducts then lack the property of being connected components? To enforce that we make sure that each duct is also connected to at least one kitchen or bathroom. By doing so ducts are always adjacent to bathrooms and kitchens, and kitchens and bathrooms are always adjacent to ducts. It is assumed that the number of ducts equals $max(n\_apartments - 1, 1)$ where $n\_apartments$ is the number of apartments in the specified input.

3.2.6 Sun Rooms Need To Be Exposed To Sunlight

Once the 2D reachability grid *reachability* described in section 3.1.2 as well as the *grid* describing which rooms occupies a certain cell described in section 3.1.1 are calculated, all sunrooms must have at least one cell that is reachable to the sun, *i.e* there must be at least one cell (row, co) where $grid[row][col] =$ index of sunroom and $reachable[row][col] = True$.

3.2.7 Adjacency Constraints

The adjacency constraint is defined for pairs of rooms, these pairs are specified in the input as described in Section 4. It is assumed that there is exactly on kitchen in an apartment, thus dining rooms will always be adjacent to that kitchen.

3.3 Soft Constraints

In contrast to hard constraints, soft constraints do not necessarily have to be met. Thus they are implemented as an objective function in our code, which is either maximized or minimized. Since OR-tools only supports one Maximize/Minimize statement, all soft constraints are merged into one Maximize statement of the from $\sum c_i$ x f_i where c_i is the coefficient associated with function f_i and i ranges from 1 to the number of soft constraints specified. Thus, if an objective function is meant to be minimized, its coefficient will be negative.

3.3.1 All rooms shall be exposed to some form of day-lighting

This is an application of the reachability constraint described in 3.1.2. To implement it, the number of rooms that are accessible to the sun is maximized.

3.3.2 Distance Between Two Rooms

Distance, not necessarily euclidean, between two specific rooms shall be greater/less than some value. The distance between two rooms is constrained to be equal to some variable `dist`, then `dist` is ensured to be greater or less than the specified value. This constraint is handled by the function `enforce_distance_constraint(first_room, second_room, distance, equality)` where `equality` is either `Equality.GREATER_THAN` or `Equality.LESS_THAN`.

3.3.3 Bedrooms in an apartment shall be as close to each other.

The maximum distance between all pairs of bedrooms was used to measure how close bedrooms are in general. This is implemented by adding $\frac{n*(n-1)}{2}$ variables where n is the number of bedrooms in the specified apartment indicating all the pairwise distances between a pair of bedrooms. Then the OR-tools function `AddMaxEquality` is used to find the maximum of these distances, this maximum is then minimized on. Thus, our implementation aims to minimize the maximum distance between a pair of bedrooms in an apartment.

3.3.4 Main Bathrooms Proximity

Main bathroom, regular ones, shall be as easy to get from every other room, specially the living room. The maximum distance to the main bathroom from all other rooms is used as a metric for this constraint. To emphasize the preference of keeping the living room, a larger coefficient for the living room is given compared to all other rooms. In particular the objective function for this constraint is *max* (2 * *living room distance*, *max distance from other rooms*)

3.4 Optional Hard Global View Constraints

3.4.1 All apartments should have a look on landscape view.

The goal of this constraint is to ensure that each apartment contains at least one room that is accessible to the landscape view. A 2D reachability array is calculated as described in Section 3.1.2. Using this 2D array, a 1D array, `reachable_apartment`, for each apartment is calculated representing whether each room in the apartment has a landscape view or not. To ensure that at least one room is accessible to the landscape view, the constraint `sum(reachable_apartment) > 0` is added to the model.

3.4.2 All Apartments Should Be Of An Equal Distance To the Elevator

Similar to the assumption made in the floor corridors constraint 3.2.3 where we connect a floor corridor to the main apartment corridor, here we enforce that the distance from the main corridor of each apartment to the elevator is the same.

3.4.3 Symmetry Constraints over floor or over same type apartments.

Symmetry is implemented over two apartments of the same type. Either both apartments appear to be reflected over some horizontal line or some vertical line. This is handled by the model. To ensure that that a pair of rooms is symmetric over a horizontal line, it must be that the the top-left and the bottom right corners are reflected over this horizontal line. More formally, given two points $(x1, y1), (x2, y2)$ and some line $y = C$, to enforce that the two points are symmetrical over the line $y = C$, it must be that $x1 = x2$ and $|y1 - C| = |y1 - C|$. To ensure that two rooms are symmetrical over a horizontal line, it must be that one room is above the line $y = C$ and the other room is below it. Without loss of generality, it is assumed that the first room is above the line $y = C$. The top left corner of the first room is compared to the bottom left corner of the second room while the bottom right corner of the first room is compared to the top right corner of the second room.

3.4.4 Allocation of Space Is To Be Done Using The Divine Ratio

For every unit in the floor we enforce that it can have a `WIDTH, HEIGHT` assignment of either one of the following assignments `[3, 5], [5, 3], [5, 8], [8, 5]`. This is based on the assumption that we will not support divine ratios greater than that as they take too much space. An important note would be that this constraint needs to be in sync with the floor utilization constraint 3.2.4, meaning the floor dimensions also need to respect the divine ratio so the user needs to be careful when using this constraint and to make sure that they are using proper floor dimensions. The constraint of assigning these variables is done as follows:

```
model.AddAllowedAssignments([WIDTH, HEIGHT], [[3, 5], [5, 3], [5, 8], [8, 5]])
```

4 I/O Demo

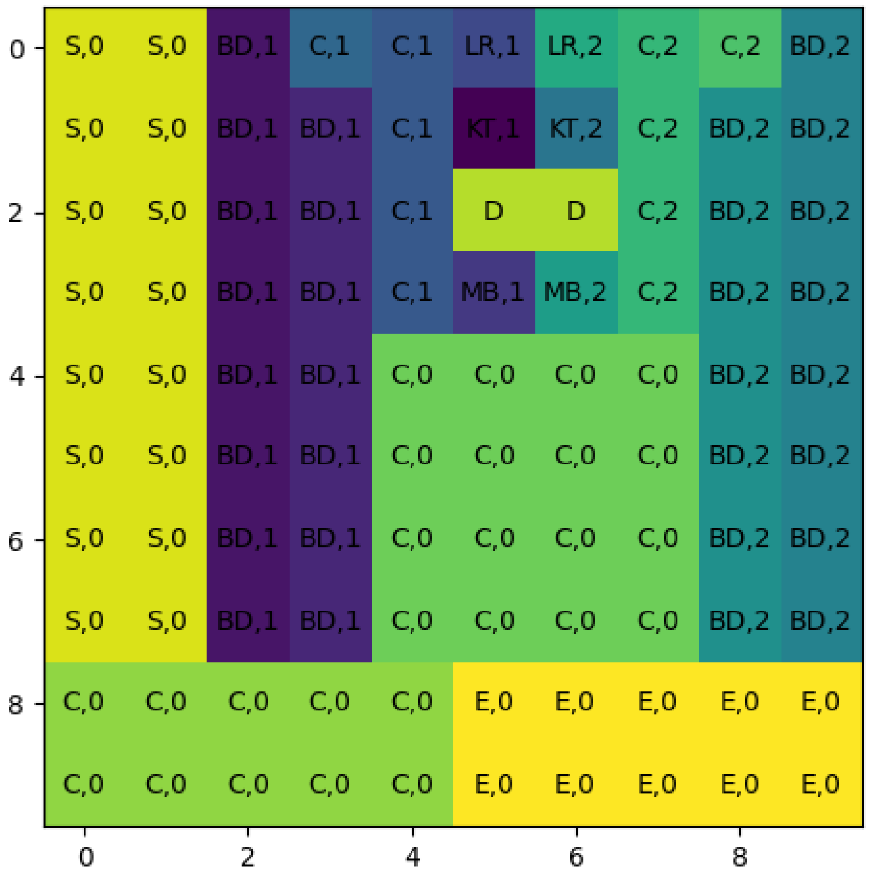
4.1 Input Format

The input is expected to be read from an input file named **input.txt**. The file must have a certain format that the code expects, the format is as follows.

1. `FLOOR_WIDTH FLOOR_LENGTH`
2. A stream of 4 **characters** in a line, where each character is reflected in the `FloorSide` Enum in 2.3 is expected in the following format:
   (a) L $\rightarrow$ LANDSCAPE
   (b) O $\rightarrow$ OPEN
   (c) N $\rightarrow$ NONE
3. The number of apartment types, where for each apartment type the following input is expected:
   (a) number of apartments with this type
   (b) number of rooms in this apartment n_rooms
   (c) number of corridors in this apartment
   (d) for each $room$ in n_rooms a stream of the following **characters** in a line is expected:
      i. room type
      ii. min area
      iii. optional width (0 if no width needs to be specified)
      iv. optional height (0 if no height needs to be specified)
      v. the index of the room that it is adjacent to, given that the room is of type dressing room or minor room as reflected in the `Room` Enum in 2.3.
   (e) For the soft constraints, a stream of 4 **lines** is expected in the format below:
      i. 0 or 1 indicating whether the sun reachability constraint 3.3.1 is enforced or not
      ii. a set of (4-tuples) to be used in 3.3.2, where each tuple consists of the following:
         A. $room\_idx1$
         B. $room\_idx2$
         C. the type of equality ($<$ or $>$) reflected in the `Equality` Enum seen in 2.3.
         D. the value used in that equality.
      iii. 0 or 1 to indicate whether the bedroom proximity constraint 3.3.3 is enforced or not
      iv. 0 or 1 to indicate whether the bathroom proximity constraint 3.3.4 is enforced or not
4. Optional Hard Global View Constraints
   (a) 0 or 1 to indicate whether the landscape constraint 3.4.1 is enforced or not.
   (b) 0 or 1 to indicate whether the elevator constraint 3.4.2 is enforced or not.
   (c) 0 or 1 to indicate whether the symmetry constraint 3.4.3 is enforced or not.
   (d) 0 or 1 to indicate whether the divine ratio constraint 3.4.4 is enforced or not.

4.2 Output Format

The visualizer method basically takes in the grid from 3.1.1 and iterates through the different apartments to plot them using matplotlib using the apartment number, and the room type which is mapped using the Enum discussed in 2.3. The final result is a 2D grid visualized using matplotlib where each cell carries he room type and the apartment number. For units such as the floor corridor, elevator, stairs, and ducts the apartment number is set to **0**. An example of the visualized output can be seen in 1.



0	S,0	S,0	BD,1	C,1	C,1	LR,1	LR,2	C,2	C,2	BD,2
	S,0	S,0	BD,1	BD,1	C,1	KT,1	KT,2	C,2	BD,2	BD,2
2	S,0	S,0	BD,1	BD,1	C,1	D	D	C,2	BD,2	BD,2
	S,0	S,0	BD,1	BD,1	C,1	MB,1	MB,2	C,2	BD,2	BD,2
4	S,0	S,0	BD,1	BD,1	C,0	C,0	C,0	C,0	BD,2	BD,2
	S,0	S,0	BD,1	BD,1	C,0	C,0	C,0	C,0	BD,2	BD,2
6	S,0	S,0	BD,1	BD,1	C,0	C,0	C,0	C,0	BD,2	BD,2
	S,0	S,0	BD,1	BD,1	C,0	C,0	C,0	C,0	BD,2	BD,2
8	C,0	C,0	C,0	C,0	C,0	E,0	E,0	E,0	E,0	E,0
	C,0	C,0	C,0	C,0	C,0	E,0	E,0	E,0	E,0	E,0
	0		2		4		6		8	

Figure 1: A possible layout for a 10x10 floor.