CP Project Report

Lujine Elfeky 40-14328, Mostafa ElHayani 40-10547

4^{th} of January 2020

1 Syntax and Semantics

1.1 Problem Model

We defined our model as such follows.
It takes as input:

1. The floor width `FloorWidth`,
2. The floor height `FloorHeight`,
3. A list `Landscapes` of length 4, representing which sides look over a landscape view. The list is ordered as follows: top, bottom, left, then right. If the value in the position of a side is 1 then it overlooks a landscape, and it is 0 otherwise.
4. A list `Open` of length 4, representing which sides are open sides, which is ordered and represented in the same way as the landscape sides.
5. A list `AptTypes` of apartment types, a struct that will be discussed in section 1.2,
6. A list `NumApts` of the same length as `AptTypes`, it contains the number of apartments of each type.

It returns the output:

1. A list `Apartments` of apartments. An apartment is a list of rooms and hallways. A room or a hallway is represented with a rectangle struct,
2. A list `Types` with the same length as `Apartments`. It contains a list of types for each apartment. The list of types indicates the type of each room/hallway. These types are indicated in section 1.3,
3. A list `OuterHallways`, which consists of outer hallways represented as rectangles.
4. A rectangle `Elevator`, representing the coordinates of the elevator,
5. A list `Ducts`, which is the list of ducts on the floor
6. A list of length 4 `[GlobalLandscapeViewConstraint, GlobalElevatorDistanceConstraint, GlobalGoldenRatio ]`, containing variables representing whether each of the optional global constraints was satisfied.

1.2 Structs

1. Rectangle `r(X, W, Y, H)`

   The rectangle struct is used to represent any element on the floor, i.e. room, hallway or elevator. X and Y are the coordinates of the upper left corner of the element, and W and H are its width and height respectively.

2. Apartment Type `apt_type(Num, RoomsTypes, MinRoomSize, MinWidths, MinHeights)`

   The apartment type contains a few parameters,

   - `Num` the number of rooms in the apartment
   - `RoomTypes` A list of the room types, this list is ordered in a specific way to satisfy the adjacency constraints. Dressing room (6) had to be followed by the bedroom (0) it's attached to. Dinning room (2) has to be followed by the Kitchen (1). And a minor bathroom (4) is followed by the room it's attached to, if it's not attached to any room, it comes at the end of the list,
   - `MinRoomSize` The minimum area of the room,
   - `MinWidths` The minimum width of the room,
   - `MinHeights` The minimum length of the room,

1.3 Room Types

The room types are given numerical codes as follows:

- Bedroom = 0
- Kitchen = 1
- Dining Room = 2
- Master bathroom = 3
- Minor bathroom = 4
- Living Room = 5
- Dressing room = 6
- Sun room = 7
- Hallway = 8
- Duct = 9

2 Algorithm Implementation

The main predicate in the code is the function input present in the "main.pl" file. It takes in the inputs and produces the outputs mentioned in section 1.

```
input(FloorWidth, FloorHeight, Landscapes, Open, AptTypes, NumApts, Apartments,
      Types, OuterHallways, Elevator,Ducts, [GlobalLandscapeViewConstraint,
     GlobalElevatorDistanceConstraint, GlobalGoldenRatio])
```

The method is divided into 3 parts, each dealing with a set of constraint, namely hard constraints, soft constraints and optional global constraints. Each of these parts will be discussed in the following subsections.

2.1 Hard Constraints

2.1.1 Room creation Constraints

The first step is defining the domains of the variables.

```
FloorWidth in 10..500,
FloorHeight in 10..500,
NumApts ins 1..10,
```

Then the predicate `createApts` creates the apartments, as well as the inner and outer hallways.
`checkAdjacency(OuterHallways)` checks that the outer hallways are connected.

```
/* Apartment and External Hallways Layout Constraints */
  maplist(createApts(FloorWidth, FloorHeight),
    AptTypes, NumApts,
    ApartmentsList, TypesList,
    AptCoordList,
    InnerHallwaysList, NumInnerHallways, OuterHallwaysList
    ),
```

`ApartmentsList` is a list of the apartments, each apartment is a list of rooms and hallways, `TypesList` is a list of lists containing the type of each room/hallway in the apartments. `InnerHallwaysList` is a list of the inner hallway rectangles and `OuterHallwaysList` is a list of the outer hallway rectangles. `AptCoordList` contains the coordinates of each rectangle that make up all the apartments and hallways, this is created just for the ease of use in labeling.

The predicate `createApts` creates all the apartments of a certain type including their inner hallways and the outer hallway connected to this apartment. The inner hallway makes sure every room in the apartment is accessible from any other room. And the outer hallways make sure that every apartment is accessible.

`createApts` calls the helper method `createAptRooms` which creates the rooms of 1 apartment of a given type. This method calls a helper method `createRoom` which creates a room.

`createRoom` creates a rectangle using the helper method `create_rect_min_area` present in `rectangle_helpers.pl` which creates a rectangle with at least the minimum area specified. It also constraints the width and the height to the minimum values. The last thing it does it making sure a sun room is exposed to one of the open sides by using the helper method `sun_room` present in `apartments.pl`.

```
createRoom(FloorWidth, FloorHeight, Type, MinRoomSize,MinWidthH,MinHeightH,
    Room, Coord):-
  create_rect_min_area(FloorWidth, FloorHeight, MinRoomSize, Room, Coord),
  Room = r(_,W,_,H),
  W #>= MinWidthH, H#>=MinHeightH,
  /* sun room exposed to day light */
  (Type #= 7) #<==> ShouldBeSunRoom,
  sun_room(FloorWidth, FloorHeight, Coord, IsSunRoom),
  ShouldBeSunRoom #==> IsSunRoom.
```

Then the ducts and the elevator are created. Ducts are created such that they're adjacent to kitchens and bathrooms.

```
/*elevator*/
create_elevator(FloorWidth, FloorHeight, OuterHallways, Elevator, ElevatorCoord
    ),
/*ducts*/
create_ducts(FloorWidth, FloorHeight, Apartments, Types, Ducts, DuctsCoord),
```

Next, it's insured that all the created rectangles created of all types do not overlap. And that the entire area of the floor is utilized.

```
/* no overlap constraint */
append(Apartments, Rooms),
append(OuterHallways, [Elevator], Outside),
append(Rooms, Outside, Floor),
disjoint2(Floor),

/* Utility of Apartments and Hallways */
apts_util(Apartments, ApartmentsArea),
calc_util(OuterHallways, HallsArea),
calc_util([Elevator], ElevatorArea),
calc_util(Ducts,DuctsArea),
TotalArea #= ApartmentsArea + HallsArea + ElevatorArea + DuctsArea,
```

2.1.2 Adjacency Constraints

These constraints ensured that dressing rooms are adjacent to the bedroom assigned to them, minor bathrooms may be assigned a room and if that's the case the bathroom is adjacent it, and the dining room is adjacent to the kitchen. To achieve this we assume the input is ordered in a certain way, where the types of adjacent rooms are stated in succession, which was mentioned in details in the description of the `apt_type` struct in section 1.2.

The implementation of this constraint is done using the predicate `adjacentRooms(Rooms, Types)` in the file `apartments.pl`. It looks for a dressing room, a dining room, or a minor bathroom. It then checks that the next element is the room they need to be adjacent to (except for minor bathroom, there is no constraint on the type of room it's adjacent to) and then makes sure these two rooms are adjacent. The following code snippet shows the constraints for the adjacency of dressings and bedrooms.

```
/* RoomH1 and has type TypeH is the head of the list */
/* RoomH2 is the follwing element in the list and has type TypeH2 */
adjacent(RoomH1, RoomH2, Adj),

TypeH #= 6 #<==> IsDressing,
IsDressing #==> Adj,
IsDressing #==> TypeH2 #= 0,
```

2.1.3 Important Helpers

- There is a helper to check the adjacency of any 2 rectangles in `rectangle_helpers.pl`.
- There are various helpers in `apartment.pl` that make sure the hallway accessibility constraints are satisfied.

2.2 Soft Constraints

These predicates are all in the file `soft.pl`. These predicates calculates a cost associated with each of the constraints. The costs of each constraint are added together to form the cost function. We then try to reduce this cost function.

2.2.1 Exposure to sunlight

This predicates tries to maximize the number of rooms exposed to sunlight. It does so by counting the number of rooms that are not exposed to sunlight, and these ones contribute to the cost of the constraint. It uses the predicate `not_exposed_to_light` to check that a room is not at any of the edges of the floor.

```
not_exposed_to_light(FloorWidth, FloorHeight, Room, NotSun):-
  Room = r(X, W, Y, H),
  X #= 0 #<==> IsAtLeft,
  Y #= 0 #<==> IsAtTop,
  X + W #= FloorWidth #<==> IsAtRight,
  Y + H #= FloorHeight #<==> IsAtBottom,
    (IsAtLeft #\/ IsAtTop #\/ IsAtRight #\/ IsAtBottom) #<==> IsSunRoom,
    NotSun #= 1 - IsSunRoom.
```

2.2.2 Bedrooms distance

Bedrooms in an apartment shall be as close to each other. The floor is given a cost based on the distances of each of the apartment's bedrooms to each other. For every apartment every two rooms are checked if both of them are bedrooms, the cost of the apartment is increased by their distance, until all couple of rooms are checked. The total cost of the floor is the sum of the costs of all apartments. The goal is to minimize this cost at labeling. The cost for each apartment is calculated as follows:

```
calculateApartmentBedroomCost([RoomH1,RoomH2|RoomT],[TypeH1,TypeH2|TypeT],Cost)
```

`[RoomH1,RoomH2|RoomT]` is the list of rooms in the apartment, `[TypeH1,TypeH2|TypeT]` is the list of types per room, and the cost is the overall apartmen cost. The distance of the two rooms is calculated with the Manhattan's distance as discussed then the following snippet shows how the cost is calculated per rooms.

```
    TypeH1#=0 #/\ TypeH2#=0 #<==> CostH#=Distance,
    TypeH1#\=0 #\/ TypeH2#\=0 #<==> CostH#=0,
```

The first line makes sure to add the distance to the cost if both rooms are bedrooms, while the other makes sure not to add a cost to the two rooms if either of them is not a bedroom. The next two lines makes sure each two rooms are checked for the overall cost.

```
    calculateApartmentBedroomCost([RoomH1|RoomT],[TypeH1|TypeT],RestCostH1),
    calculateApartmentBedroomCost([RoomH2|RoomT],[TypeH2|TypeT],RestCostH2),
```

and finally the global cost of the apartment is calculated as follows:

```
    Cost #= CostH + RestCostH1 + RestCostH2.
```

2.2.3 Bathroom distance

Main bathroom shall be as easy to get from every other room. As the Bedroom distance constraint, the floor is given a cost based on the distances of all of the apartment's rooms to the main bathrooms. For every apartment every two rooms are checked if one of them is a main bathroom and the other is not, the cost of the apartment is increased by their distance, until all couple of rooms are checked. The total cost of the floor is the sum of the costs of all apartments. The goal is to minimize this cost at labeling. The cost for each apartment is calculated as follows:

```
calculateApartmentBathroomCost([RoomH1,RoomH2|RoomT],[TypeH1,TypeH2|TypeT],Cost
    )
```

`[RoomH1,RoomH2|RoomT]` is the list of rooms in the apartment,`[TypeH1,TypeH2|TypeT]` is the list of types per room, and the cost is the overall apartment cost. The distance of the two rooms is calculated with the Manhattan's distance as discussed then the following snippet shows how the cost is calculated per rooms.

```
(TypeH1#=3 #/\ TypeH2#\=3) #\/ (TypeH1#\=3 #/\ TypeH2#=3) #<==> CostH#=Dist,
(TypeH1#=3 #/\ TypeH2#=3) #\/ (TypeH1#\=3 #/\ TypeH2#\=3) #<==> CostH#=0,
```

The first line makes sure to add the distance to the cost if either of the rooms is a bathroom while the other is not, the other makes sure cost is zero otherwise. The next two lines makes sure each two rooms are checked for the overall cost.

```
    calculateApartmentBathroomCost([RoomH1|RoomT],[TypeH1|TypeT],RestCostH1),
    calculateApartmentBathroomCost([RoomH2|RoomT],[TypeH2|TypeT],RestCostH2),
```

and finally the global cost of the apartment is calculated as follows:

```
    Cost #= CostH+RestCostH1+RestCostH2.
```

2.3 Global Constraints

These predicates are all in the file `global.pl`. These predicates may or may not be satisfied as they are optional. When the corresponding output of the constraint is 1, it is satisfied.

2.3.1 Landscape

All apartments should have look on a landscape view. The assumption here is that for all apartments: an apartment has a landscape view if at least one of its rooms is adjacent to a landscape side. Hence, using the landscape list, we can count the number of rooms, which are adjacent to a landscape side, so the four landscape positions are:

1. (0,Y): if the left side of the apartment is a landscape side,
2. (FloorWidth,Y): if the right side of the apartment is a landscape side,
3. (X,0): if the upper side of the apartment is a landscape side,
4. (X,FloorHeight): if the bottom side of the apartment is a landscape side.

if the room has any of the four values for the top left vertex of the rectangle the room is considered adjacent to the landscape. an apartment with 1 or more rooms adjacent to the landscape side is counted, and if all apartments are adjacent to a landscape side, the global landscape view constraint is satisfied. The predicate is defined as follows:

```
apartmentLandscapeView(Rooms,Landscapes,FloorWidth,FloorHeight,HasLandscape).
```

Where Rooms is the list of rooms per apartment, landscape is the list of landscape sides, FloorWidth and floorHeight are the floor's width and height, and HasLandscape is the output that counts how many rooms have landscape views. The next line of code shows how the vertex constraint is checked.

```
    ((Up#=1 #/\ Y#=0) #\/ (Down#=1 #/\ Y#=FloorHeight) #\/ (Left#=1 #/\ X#=0)
    #\/(Right#=1 #/\ X#=FloorWidth) )#<==> Landscape,
```

Where Up, Down, Left, and Right are the four Boolean landscape sides respectively. The rest of the rooms is checked by the recusrive call:

```
apartmentLandscapeView(RT,Landscapes,FloorWidth,FloorHeight,RestLandscape),
```

And the global landscape number is calculated with the next line of code

```
HasLandscape#=Landscape+RestLandscape.
```

And finally if an apartment HasLandscape value is greater than or equal to 1 that apartment satisfies the constraint.

2.3.2 Elevator distance

All apartments should be of an equal distance to the elevators unit. The assumption here is that getting an average value for the coordinates of all the rectangles' top left vertices is a midpoint of a single apartment. Then, the distances from that midpoint and the elevators midpoint is also calculated using the Manhattan distance: $|X_1 - X_2| + |Y_1 - Y_2|$ The constraint is satisfied if all distances are equal, however due to precision that might be lost in averaging the points, an error of 5 points is allowed for the constraint. So for a list $D = D_1, D_2, D_3, ..., D_n$ where D_i is the distance of apartment i from the elevator, the constraint is satisfied if $max(D) - min(D) \leq 5$.

The predicate used is defined as follows:

```
globalElevatorDistance(AptList,Elevator,DistanceList)
```

Where AptList is the list of apartments, Elevator is the elevator unit, and DistanceList is the list of distances per apartment to the elevator. The midpoint of the elevator is calculated first.

```
Elevator = r(X,W,Y,H),
MidPointElevatorX #= (2*X+W) div 2,
MidPointElevatorY #= (2*Y+H) div 2,
```

Hence, a midpoint for the apartment is calculated,

```
sumAptPoints(Ah,(SumX,SumY)),
length(Ah,N),
AvgX #= SumX div N,
AvgY #= SumY div N,
```

Where SumAptPoints sums all the X and Y coordinates values. The distance is then calculated as

```
DistanceH #= (abs(AvgX-MidPointElevatorX)) + (abs(AvgY-MidPointElevatorY))
```

The rest of the list is calculated recursively. To make sure the constraint is satisfied the difference between the distance needs to be less than a certain value. It is calculated as follows:

```
allDistancesEqual([Dh1,Dh2|Dt],ElevatorDistanceConstraint):-
    abs(Dh1-Dh2) #=<6 #<==> HeadFlag,
    allDistancesEqual([Dh2|Dt],RestFlag),
    ElevatorDistanceConstraint#= HeadFlag+RestFlag.
```

ElevatorDistanceConstraint carries the sum of distances that satisfy the constraint and if the number is equal to the number of apartments the global constraint is satisfied.

2.3.3 Golden Ratio

Aim to allocate spaces with ratios following the divine proportion. A rectangle has the golden ratio if the ratio of its sides is the same as the ratio of their sum to the larger of the two. Like above, due to the loss of precision in the calculation of the ratio and that this is solved in finite domains rather than rational domain the difference is allowed an error value rather than exact equality. So for every rectangle on the floor. A rectangle has the golden ratio if: $\frac{Bigger+Smaller}{Bigger} - \frac{Bigger}{Smaller} <= 5$.

And the global golden ratio constraint is satisfied if all the rectangles on the floor satisfy the golden ratio constraint. The golden ratio for a list of rects is calculated by the predicate:

```
goldenRatio(RectList,GlobalGolden).
```

Where RectList is a list of rectangles and GlobalGolden is the number of rects satisfying the golden ratio constraint. To get the bigger and smaller values of the rect width and height, the following snippet is used

```
W#>=H #<==> Bigger #=W,
W#>=H #<==> Smaller #=H,

W#<H #<==> Bigger #=H,
W#<H #<==> Smaller #=W,
```

And then the constraint is satisfied by the following snippet

```
Ratio1 #= (Bigger+Smaller) div Bigger,
Ratio2 #= Bigger div Smaller,
abs( Ratio1 - Ratio2 )#=<5 #<==> GoldenRect,
```

Then the rest is calculated recursively,

```
goldenRatio(RectT,GoldenRest),
GlobalGolden #= GoldenRect + GoldenRest.
```

2.4 Miscellaneous functionality

- There is a test file `test.pl` which contains test cases.
- There is a plotting file `plot.pl` which creates a very basic plot of the layout of the floor.
- There is a python plotting file `plotter.py` which creates a colour coded plot of the layout of the floor

3 Results and Examples

The outputs of our implemented predicate `input/12` was mentioned in section 1. The following are some output examples of the predicate. Most examples take too long to terminate, but the following ones are done straight away.

the color code of the output is:

Bedroom (255,0,0), Kitchen (0,255,0), Dining Room (255,0,255), Master bathroom (0,0,255), Minor bathroom (255,255,0), Living Room (0,255,255), Dressing room (255,0,255), Sunroom (100,0,100), Hallway (0,0,0), Duct (40,40,40),

3.1 Example 1

The inputs of the predicate are:

1. Width: 38,
2. Height: 38,
3. Landscapes: [0,1,0,1],
4. ApartmentTypes: `[apt_type(8, [0, 4, 0, 2, 1, 3, 5, 7], [3, 3, 3, 3, 3, 3, 3, 3], [3, 3, 3, 3, 3, 3, 3, 3], [1, 1, 1, 1, 1, 1, 1, 1])]`,
5. Number of apartments: [2]

The sample output is:

- Apartments list:`[[r(0, 3, 0, 1), r(0, 3, 1, 1), r(0, 3, 2, 1), r(0, 3, 3, 1), r(0, 3, 4, 1), r(0, 3, 5, 1), r(0, 3, 6, 1), r(0, 3, 7, 1), r(3, 1, 0, 8)], [r(0, 3, 9, 1), r(0, 3, 10, 1), r(0, 3, 11, 1), r(0, 3, 12, 1), r(0, 3, 13, 1), r(0, 3, 14, 1), r(0, 3, 15, 1), r(0, 3, 16, 1), r(3, 1, 9, 8)]]` ,
- Types = `[[0, 4, 0, 2, 1, 3, 5, 7, 8], [0, 4, 0, 2, 1, 3, 5, 7, 8]]`,
- Hallways list: `[r(0, 4, 8, 1), r(4, 34, 0, 18)]`,
- Elevator unit: `r(0, 38, 18, 20)`,
- Duct: `[r(0, 1, 0, 2), r(0, 1, 0, 2)]`,
- Global constraints: `[0,0,0]`.

3.2 Example 2

The inputs of the predicate are:

1. Width: `76`,
2. Height: `76`,
3. Landscapes: `[1,0,0,1]`,
4. ApartmentTypes: `[apt_type(8, [0, 4, 0, 2, 1, 3, 5, 7], [9, 9, 9, 9, 9, 9, 9, 9], [3, 3, 3, 3, 3, 3, 3, 3], [3, 3, 3, 3, 3, 3, 3, 3])]`,
5. Number of apartments: `[2]`.

The sample output is:

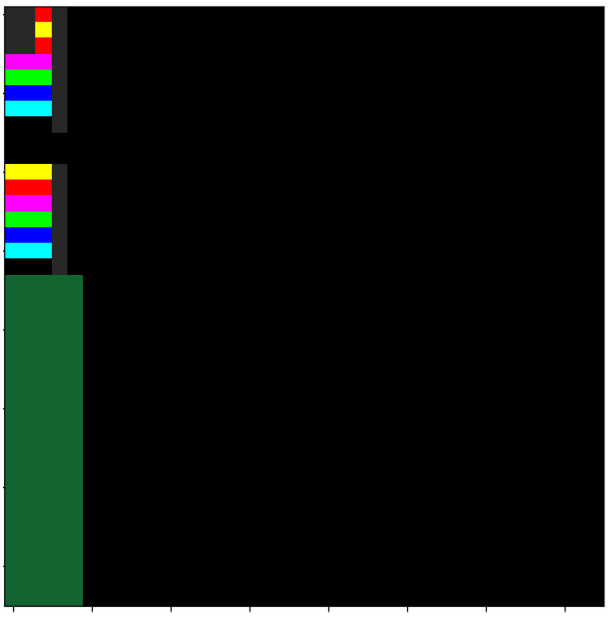

Figure 1: output of example one

- Apartments list: `[[r(0, 3, 0, 4), r(0, 3, 4, 4), r(0, 3, 8, 4), r(0, 3, 12, 4), r(0, 3, 16, 4), r(0, 3, 20, 4), r(0, 3, 24, 4), r(0, 3, 28, 4), r(3, 1, 0, 29)], [r(0, 3, 32, 4), r(0, 3, 36, 4), r(0, 3, 40, 4), r(0, 3, 44, 4), r(0, 3, 48, 4), r(0, 3, 52, 4), r(0, 3, 56, 4), r(0, 3, 60, 4), r(3, 1, 31, 33)]]`,
- Types = `[[0, 4, 0, 2, 1, 3, 5, 7, 8], [0, 4, 0, 2, 1, 3, 5, 7, 8]]`,
- Hallways list: `[r(3, 1, 29, 2), r(4, 72, 0, 65)]`,
- Elevator unit: `r(0, 76, 65, 11)`,
- Duct: `[r(0, 1, 0, 2), r(0, 1, 0, 2)]`,
- Global constraints: `[0,0,0]`.

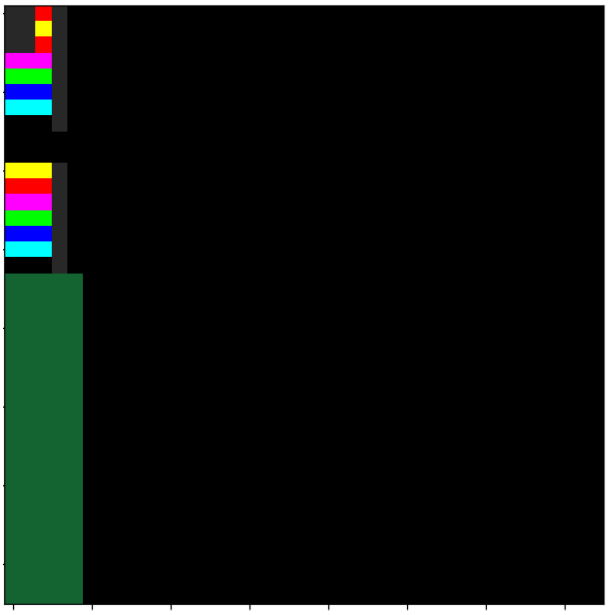

Figure 2: output of example two