Report for assignments for Automated Reasoning

Authors:
Dibran Dokter s1047390 dibran.dokter@ru.nl
Sijmen van Bommel s1038820

Problem 1: Pallets

Eight trucks have to deliver pallets of obscure building blocks to a magic factory. Every truck has a capacity of 8000 kg and can carry at most eight pallets. In total, the following has to be delivered:

- Four pallets of nuzzles, each of weight 700 kg.
- A number of pallets of prittles, each of weight 400 kg.
- Eight pallets of skipples, each of weight 1000 kg.
- Ten pallets of crottles, each of weight 2500 kg.
- Twenty pallets of dupples, each of weight 200 kg.

Skipples need to be cooled; only three of the eight trucks have facility for cooling skipples.
Nuzzles are very valuable: to distribute the risk of loss no two pallets of nuzzles may be in the same truck.
1. Investigate what is the maximum number of pallets of prittles that can be delivered, and show how for that number all pallets may be divided over the eight trucks.
2. Do the same, with the extra information that prittles and crottles are an explosive combination: they are not allowed to be put in the same truck.

Solution:

We start by defining a 2 dimensional array of integer values $trucks$ for the number of trucks ($ntrucks$) and the number of different pallets ($npallettypes$).
This leads to a $n \times k$ array of integers where n is the number of trucks and k is the number of pallet types.
Where the integer value $trucks_{i,j}$ is the number of pallets of type j in truck i.

Besides this list we have a list for the weights for the different types of pallets, this list is used as a lookup table.

$$weightlookup = [nuzzlesweight, prittlesweight, skipplesweight, crottlesweight, dupplesweight]$$

which for this specific case is:

$$weightlookup = [700, 400, 1000, 2500, 200]$$

We define another list for the number of pallets that are required for the different types of pallets.

$$itemtypes = [nuzzlesreq, prittlesreq, skipplesreq, crottelsreq, dupplesreq]$$

which for this specific instance is:

$$itemtypes = [4, 0, 8, 10, 20]$$

Where the 0 for the number of prittles means we want to put as many as possible since there is no limit.

After this we can start defining the different constraints on the solution.

All the number of pallets on the trucks must be non-negative:

$$nonnegative = \bigwedge_{i=1}^{ntrucks} \bigwedge_{j=1}^{itemtypes} (trucks_{i,j} \geq 0).$$

Constrain the weight of the pallets per truck. we do this by looking up the weight of the itemtype and multiplying it by the number of pallets of that type for every itemtype. Then we sum these for every truck and check if the total weight us under the maximum weight:

$$weight = \bigwedge_{i=1}^{ntrucks} \left(\sum_{j=1}^{itemtypes} (weighttable_j \cdot trucks_{i,j}) \leq maxweight. \right.$$

Constrain the capacities of the pallets per truck by summing the number of items per truck:

$$capacities = \bigwedge_{i=1}^{ntrucks} \sum_{j=1}^{itemtypes} (trucks_{i,j}) \leq capacity$$

For each pallet type, require at least the given number of pallets, except for the prittles, example for nuzzles:

$$nuzzles = \sum_{i=1}^{ntrucks} (trucks_{i,nuzzles}) \geq nuzzlesrequired$$

This can be generalized as follows:

$$items = \bigwedge_{j=1}^{itemtypes} \sum_{i=1}^{trucks} (trucks_{i,j}) \geq itemsrequired_j$$

After defining these constraints the basic case without the cooling or distribution of nuzzles is satisfied.

Then we define the further constraints as follows.

Skipples need to be cooled

Only three of the eight trucks have facility for cooling skipples

We do this by taking all the non-cooled trucks and making sure there are 0 nuzzles in them:

$$ncooledtrucks = 3 \tag{1}$$

$$cooledskipples = \bigwedge_{i=ncooledtrucks+1}^{notcooledtrucks} (trucks[i][nuzzles] == 0) \tag{2}$$

Nuzzles are very valuable

To distribute the risk of loss no two pallets of nuzzles may be in the same truck:

$$valuablenuzzles = \bigwedge_{i=1}^{ntrucks} (trucks_{i,nuzzles} < 2)$$

Now we have all constraints, the total formula consists of the conjunction of all these constraints:

$$nonnegative \wedge capacities \wedge items \wedge cooledskipples \wedge valuablenuzzles$$

1.

Investigate what is the maximum number of pallets of prittles that can be delivered, and show how for that number all pallets may be divided over the eight trucks.

After defining these constraints we optimise for the number of prittles by using the z3 Optimizer, and we define the following requirement:

$$\bigwedge_{i=1}^{ntrucks} maximize(trucks_{i,prittles})$$

Applying `python trucks.py` yields the following result within half a second:

```
[0, 5, 1, 2, 0]
[0, 4, 3, 1, 0]
[0, 0, 4, 0, 4]
[1, 0, 0, 2, 5]
[1, 0, 0, 2, 5]
[1, 5, 0, 1, 1]
[0, 8, 0, 0, 0]
[1, 0, 0, 2, 5]
```

which contains:
$5 + 4 + 0 + 0 + 0 + 5 + 8 + 0 = 22$ prittles

2.

Do the same, with the extra information that prittles and crottles are an explosive combination: they are not allowed to be put in the same truck. We restrict the model that not both prittles and crottles can have a capacity of more than zero for each truck. :

$$\bigwedge_{i=1}^{ntrucks} \neg(trucks_{i,prittles} > 0 \wedge trucks_{i,crottles} > 0)$$

Again applying `python trucks.py` yields the following result within 4 seconds:

```
[0, 0, 2, 2, 4]
[1, 0, 2, 2, 1]
[0, 4, 4, 0, 0]
[1, 0, 0, 2, 5]
[0, 8, 0, 0, 0]
[0, 8, 0, 0, 0]
[1, 0, 0, 2, 5]
[1, 0, 0, 2, 5]
```

This solution has $4 + 8 + 8 = 20$ prittles

Generalization

Our solution can be generalized by giving a different number for *ntrucks* and *npallettypes* and defining a different lookup list for the weights and required number of pallets per type.

This way the problem can be generalized to fitting any number of items in any number of trucks according to some weight limit and required items.

The problem is scale-able by changing these values but this will negatively impact the performance for a big enough number of trucks and required pallets since it has to check a larger space for a solution.

Problem 2: Chip Design

Problem

Give a chip design containing two power components and ten regular components satisfying the following constraints:

- Both the width and the height of the chip is 30.
- The power components have width 4 and height 3.
- The sizes of the ten regular components are 4×5, 4×6, 5×20, 6×9, 6×10, 6×11, 7×8, 7×12, 10×10, 10×20, respectively.
- All components may be turned 90°, but may not overlap.
- In order to get power, all regular components should directly be connected to a power component, that is, an edge of the component should have a part of length ¿ 0 in common with an edge of the power component.
- Due to limits on heat production the power components should be not too close: their centres should differ at least 16 in either the x direction or the y direction (or both).

What if this last distance requirement of 16 is increased to 17? And what if it is increased to 18?

Solution

First we define a data structure which will hold the different components of the chip. This is a 2 dimensional array with the different *components* and their *attributes*.
A component has the following attributes: X position on the top-left corner of the chip, Y position on the top-left corner of the chip, the width of the component and the height of the component.

Besides this we have a list of components where every component has its corresponding width and height. So $components = [(4,3),(4,3),(4,5),...]$ for the power components and the regular components.
From this list of components we take the first 2 components to be the power components, so $powercomponents = [components_0, components_1]$

After defining this data structure we define the following constraints:

The constraint that the X and Y values for the different components have to be within the 30×30 bounds of the size of the chip. We do this by checking the X and Y values for all the components and ensuring they are within the width and height of the chip:

$$\bigwedge_{c=1}^{components} chip_{c,X} \geq 0 \wedge chip_{c,Y} \geq 0$$
$$\wedge chip_{c,X} + chip_{c,Width} \leq ChipWidth \wedge chip_{c,Y} + chip_{c,Height} \leq ChipHeight$$

The constraint that the components on the chip have to adhere to the sizes of the components from the components list. The components on the chip can be rotated so they either have to match the standard orientation or the rotated orientation. We do this by checking that any component on the chip has the same width and height as either the normal orientation of a component in the components list or a rotated orientation of a component in the components list:

$$\bigwedge_{c=1}^{components} (chip_{c,Width} == components_{c,Width} \wedge chip_{c,Height} == components_{c,Height})$$
$$\vee (chip_{c,Width} == components_{c,Height} \wedge chip_{c,Height} == components_{c,Width})$$

The constraint that chips cannot overlap. We do this by taking 2 different components and ensuring they do not overlap, we do this for all components. We check if component do not overlap by checking if one component is either above, below, left or right from another component.

we define a lambda function:

$$Overlap_{c1,c2} = chip_{c2,X} \geq (chip_{c1,X} + chip_{c1,Width}) \vee chip_{c2,Y} \geq chip_{c1,Y} + chip_{c1,Height}$$
$$\vee (chip_{c2,X} + chip_{c2,Width}) \leq chip_{c1,X} \vee (chip_{c2,Y} + chip_{c2,Height}) \leq chip_{c1,Y}$$

for every two components:

$$\bigwedge_{c1=1}^{components} \bigwedge_{c2=1}^{components} (c1 < c2 \wedge Overlap_{c1,c2})$$

In order to get power, all regular components should directly be connected to a power component, that is, an edge of the component should have a part of length > 0 in common with an edge of the power component:
For this we take the components without the powercomponents and just the powercomponents and we define this constraint in 2 parts, first we check if the chip is connected above or below the power chip:

$$aboveOrBelow =$$
$$\bigwedge_{c=1}^{components} \bigwedge_{pc=1}^{powercomponents} chip_{c,X} < (chip_{pc,X} + chip_{pc,Width}) \wedge (chip_{c,X} + chip_{c,Width}) > chip_{pc,X}$$
$$\wedge (chip_{c,Y} + chip_{c,Height} == chip_{pc,Y} \vee chip_{c,Y} == chip_{pc,Y} + chip_{pc,Height})$$

After this we check if the chip is connected to the left or the right of the power chip:

$$leftOrRight =$$
$$\bigwedge_{c=1}^{components} \bigwedge_{pc=1}^{powercomponents} chip_{c,Y} < (chip_{pc,Y} + chip_{pc,Height}) \wedge (chip_{c,Y} + chip_{c,Height}) > chip_{pc,Y}$$
$$\wedge (chip_{c,X} + chip_{c,Width} == chip_{pc,X} \vee chip_{c,X} == chip_{pc,X} + chip_{pc,Width})$$

After creating these constraints we add their disjunction:

$$aboveOrBelow \vee leftOrRight$$

Due to limits on heat production the power components should be not too close: their centres should differ at least 16 in either the x direction or the y direction (or both). We define this constraint as follows, where *dist* is the distance the power components should differ.

We do this by taking the chip X and Y and multiplying it by 2 to not get any float values, then we add the width or height depending on whether we are checking for X or Y. After this we compare the difference between the components with the distance multiplied by 2.

We define a lambda function:

$$Heat_{pc1,pc2} = ((chip_{pc1,X} * 2 + chip_{pc1,width} - chip_{pc2,X} * 2 + chip_{pc2,width} >= dist * 2)$$
$$\vee (chip_{pc1,Y} * 2 + chip_{pc1,height} - chip_{pc2,Y} * 2 + chip_{pc2,height} >= dist * 2))$$

for every two powercomponents:

$$\bigwedge_{pc1=1}^{powercomponents} \bigwedge_{pc2=1}^{powercomponents} (pc1 < pc2 \wedge Heat_{c1,c2})$$

Heat 16

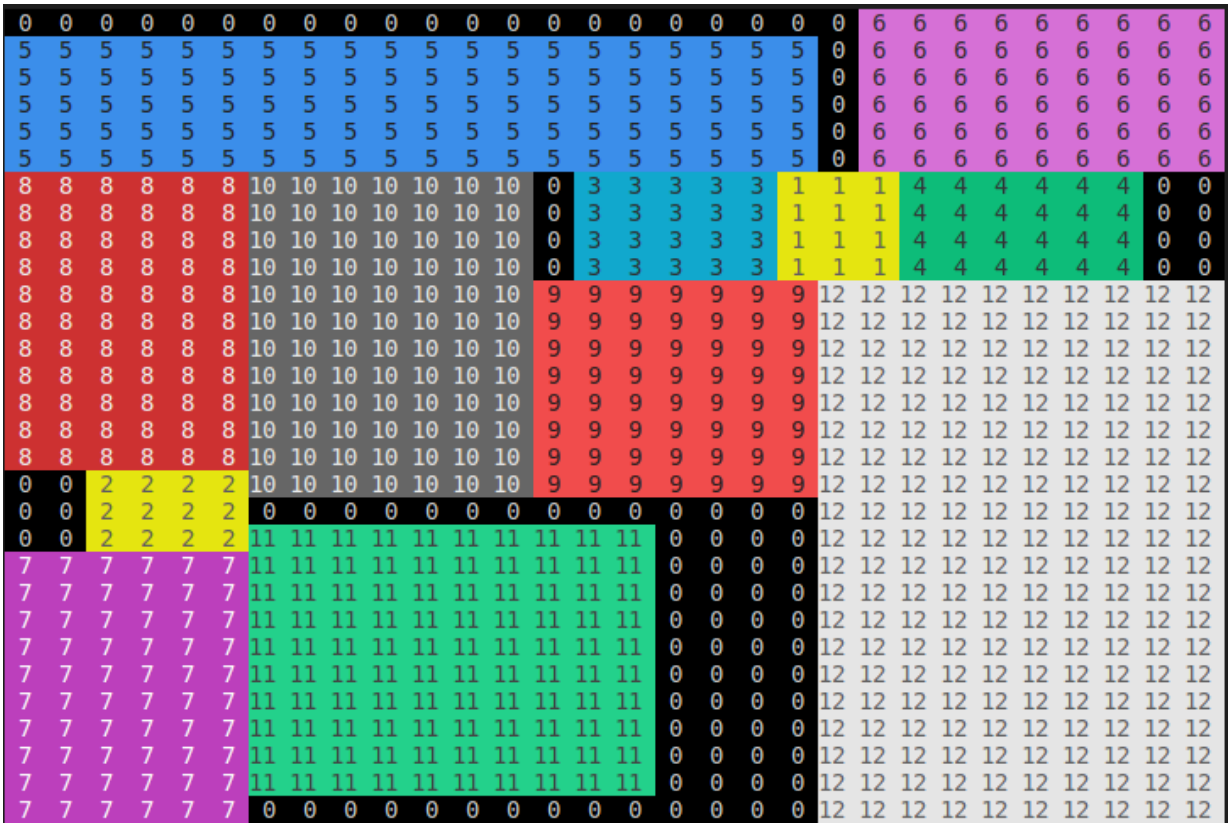
After we have defined these we can find a solution for the distance requirement of 16. We did this by applying `python chipdesign.py` which yields the following result within around 6 seconds:

sat

```
[19, 6, 3, 4]
```

```
[2, 17, 4, 3]
[14, 6, 5, 4]
[22, 6, 6, 4]
[0, 1, 20, 5]
[21, 0, 9, 6]
[0, 20, 6, 10]
[0, 6, 6, 11]
[13, 10, 7, 8]
[6, 6, 7, 12]
[6, 19, 10, 10]
[20, 10, 10, 20]
```

Which can be visualised to show the following where components 1 and 2 are the power components:



When we calculate the distance between the power components, $X = |(19 + 1.5) - (2 + 2)| = 16$ and $Y = |(6 + 4) - (17 + 1.5)| = 8.5$ we see that this is correct.

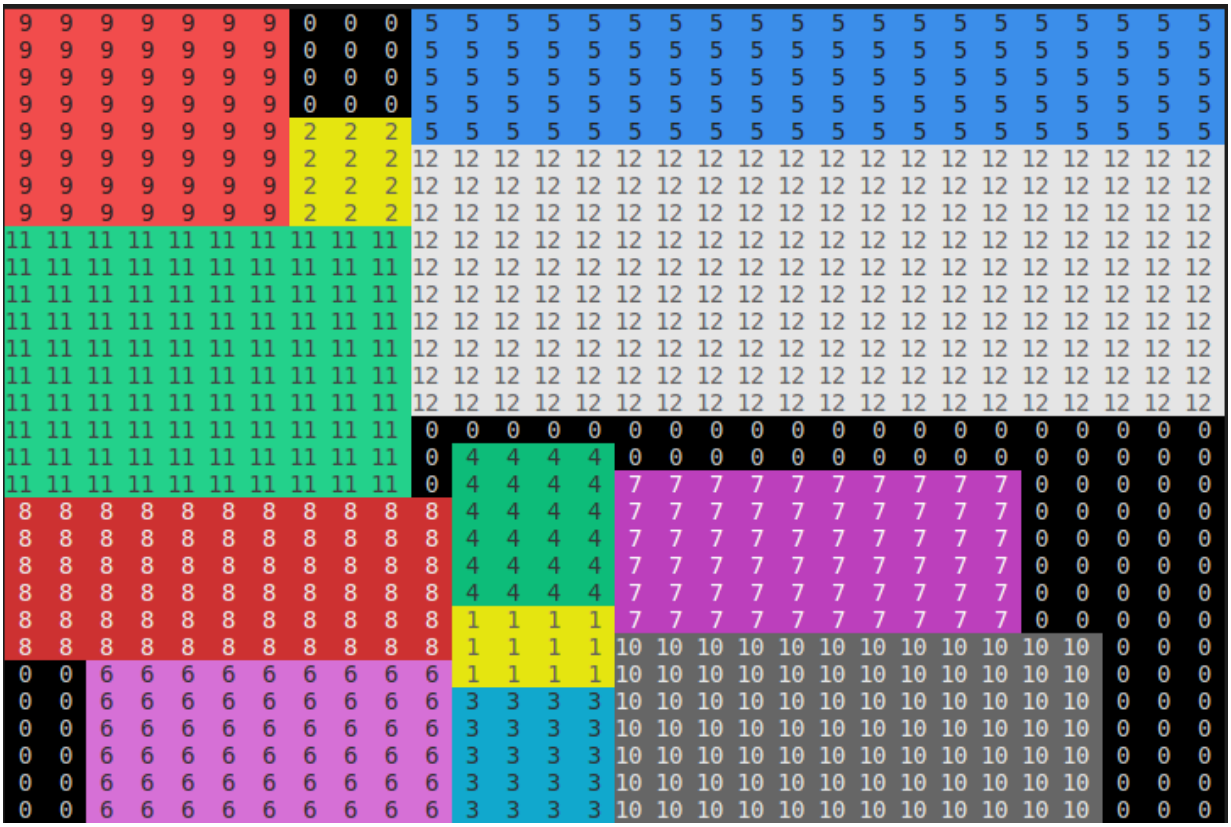
Heat 17

We did this by again applying `python chipdesign.py`, but this time with the heat distance value set to 17 which yields the following result within around 6 seconds.

sat

```
[11, 22, 4, 3]
[7, 4, 3, 4]
[11, 25, 4, 5]
[11, 16, 4, 6]
[10, 0, 20, 5]
[2, 24, 9, 6]
[15, 17, 10, 6]
[0, 18, 11, 6]
[0, 0, 7, 8]
[15, 23, 12, 7]
[0, 8, 10, 10]
[10, 5, 20, 10]
```

Which can be visualised to show the following where components 1 and 2 are the power components:



When we calculate the distance between the power components, $X = |(11 + 2) - (7 + 1.5)| = 4.5$ and $Y = |(22 + 1.5) - (4 + 2)| = 17.5$ we see that this is correct.

Heat 18

We did this by again applying `python chipdesign.py`, but this time with the heat distance value set to 17 which yields the following result within around 1 minute and 44 seconds:

unsat

Generalization:

This problem can be generalized by creating a bigger chip and adding components and changing the number and size of the power components.

When defining a bigger chip with the same number of components the problem becomes easier but if we also require a lot more components to be placed and add no more power components it becomes unsat quickly since only a certain number of components can be connected to power components.

But if we also add some power components it becomes a lot harder to place all the components. Thus our solution is easily scale-able but the efficiency depends on the specific problem.

Our approach can also be used in other problems like fitting parts in 3 or 4 dimensional space by changing the structure to check for a extra dimension, for example Z in the case of 3 dimensions.

Problem 3: Dinner

Problem:

Five couples each living in a separate house want to organize a dinner.
Since all restaurants are closed due to some lock-down, they will do it in their own houses.
The dinner will consist of 5 rounds.
Due to the 1.5 meter restriction in every house presence of at most 5 people is allowed, by which every round has to be prepared and served in two houses simultaneously, each with the corresponding couple and three guests.
Every couple will serve two rounds in their house, and in between the rounds participants may move from one house to another.
Every two people among the 10 participants meet each other at most 4 times during these 5 rounds. Further there are four desired properties:

(A) Every two people among the 10 participants meet each other at least once.

(B) Every two people among the 10 participants meet each other at most 3 times.

(C) Couples never meet outside their own houses.

(D) For every house the six guests (three for each of the two rounds) are distinct.

1. Show that (A) is possible, both in combination with (C) and (D), but not with both (C) and (D).

2. Show that (B) is possible in combination with both (C) and (D).

Solution:

We start by defining a data structure for the dinner schedule, for this we use a number of *rounds*, *houses* and *people*. where $rounds = 5, houses = 5, people = 10$
This leads to a $n \times m \times k$ array of Booleans where n is the number of rounds, m is the number of houses and k is the number of people.

Which leads to the following output, where we get 5 of these arrays, one for every round.
One such round:

```
[1, 1, 1, 0, 0, 1, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 1, 0, 1, 1]
```

Where every row is a house, and every column is a person. a one denotes that a person is in a house in that round.

We need to define which people are couples. we do this by providing a list that maps people to houses. if two people live in the same house they are a couple. We made sure that couples are next to each other in the schedule. (e.g. the first two columns are the first couple the 3rd and 4th column are the second couple etc.) this is defined as follows:

$$personToHouseIndex = \bigwedge_{p=0}^{people} \lfloor p/(\lfloor people/houses \rfloor) \rfloor$$

This list allows us to find the house that corresponds to a given person.

After defining the data structure we start defining the constraints for the problem.
First we define some basic constraints.

A person is exactly in one house in each round. so we sum every column over each round where this sum must be 1. When we sum booleans we treat the booleans as integers

where $false = 0$ and $true = 1$:

$$\bigwedge_{r=1}^{rounds} \bigwedge_{p=1}^{people} \sum_{h=1}^{houses} (dinner\_schedule_{r,h,p} == 1)$$

At most five people per house AND each round dinner is served in two houses. with 10 people this leads to the constraint that there are precisely 5 people per occupied house (since $10/2 = 5$). Since we know that a person cannot be in multiple houses at the same time it is sufficient to say that each house has exactly 5 guests or is unoccupied:

$$\bigwedge_{r=1}^{rounds} \bigwedge_{h=1}^{houses} \sum_{r=1,h=1}^{rounds,houses} \left((dinner\_schedule_{r,h}) == 5 \vee \sum_{p=1}^{people} (dinner\_schedule_{r,h,p}) == 0 \right)$$

Each occupied house needs to have their couple in it. we check this by checking if the house is empty or if the house has it's couple in it. For finding the people that are in the house we could use the personToHouseIndex. but because of how the indexes align it we can use $hi * 2$ and $hi * 2 + 1$ where hi is the house index, instead. We use (h-1)*2+1 and (h-1)*2+2 since we count from 1 and not from zero:

$$\bigwedge_{r=1}^{rounds} \bigwedge_{h=1}^{houses} (\left(dinner\_schedule_{r,h,(h-1)*2+1} \wedge dinner\_schedule_{r,h,(h-1)*2+2} \right) \vee \sum_{p=1}^{people} (dinner\_schedule_{r,h,p}) == 0)$$

Each couple is in their house exactly 2 times. For finding the house which belongs to a person we use the personToHouseIndex, after this we can say that this person is in the corresponding house exactly 2 times. Since we do this for every person we check that every person in the couple is in their corresponding house exactly 2 times:

$$\bigwedge_{p=1}^{people} \sum_{r=1}^{rounds} \left(dinner\_schedule_{r,persontohouseindex_p,p} \right) == 2$$

Every two people among the 10 participants meet each other at most 4 times during these 5 rounds. we do this by looking at every combination of two people, then going over every round and seeing if there is at least one occurrence of both people being in the same house. we sum for each set of people how often this occurs and then just check if it is fewer or equal to 4 times:

we define a lambda function:

$$Lam1_{p,p2} = \sum_{r=1}^{rounds} \left(\bigvee_{h=1}^{houses} (dinner\_schedule_{r,h,p} \wedge dinner\_schedule_{r,h,p2}) \right) <= 4$$

for every two people

$$\bigwedge_{p=1}^{people} \bigwedge_{p2=1}^{people} (p < p2 \wedge Lam1_{p,p2})$$

After defining these basic constraints we define the extra constraints A,B,C and D.

(A) Every two people among the 10 participants meet each other at least once. same as the "meet at most 4 times" constraints but now the number for meetings being at least 1:
we define a lambda function:

$$Lam2_{p,p2} = \sum_{r=1}^{rounds} \left(\bigvee_{h=1}^{houses} (dinner\_schedule_{r,h,p} \wedge dinner\_schedule_{r,h,p2}) \right) >= 1$$

for every two people

$$\bigwedge_{p=1}^{people} \bigwedge_{p2=1}^{people} (p < p2 \wedge Lam2_{p,p2})$$

(B) Every two people among the 10 participants meet each other at most 3 times. same as the "meet at most 4 times" constraints but now the number for meetings being at most 3:

we define a lambda function:

$$Lam3_{p,p2} = \sum_{r=1}^{rounds} \left(\bigvee_{h=1}^{houses} (dinner\_schedule_{r,h,p} \wedge dinner\_schedule_{r,h,p2}) \right) <= 3$$

for every two people

$$\bigwedge_{p=1}^{people} \bigwedge_{p2=1}^{people} (p < p2 \wedge Lam3_{p,p2})$$

(C) Couples never meet outside their own houses:
we define a lambda function:

$$Lam4_{p1,p2,h} = \bigwedge_{r=1}^{rounds} (\neg (dinner\_schedule_{r,h,p1} \wedge dinner\_schedule_{r,h,p2}))$$

for each couple and every house except theirs:

$$\bigwedge_{c=1}^{houses} \bigwedge_{h=1}^{houses} \left(c \neq h \wedge Lam4_{(c-1)*2+1,(c-1)*2+2,h}\right)$$

(D) For every house the six guests (three for each of the two rounds) are distinct. we check for all houses and all people not in living in that house how often they visit said

house across all rounds. this must be smaller than two otherwise the same guest is not distinct:

$$\bigwedge_{p=1}^{houses} \bigwedge_{p=1}^{people} \left(persontohouseindex_p \neq h \wedge \sum_{p=1}^{rounds} (dinner\_schedule_{r,h,p}) < 2 \right)$$

Now that we have defined all the constraints we can show the outputs for the assignments.

1. Show that (A) is possible, both in combination with (C) and (D), but not with both (C) and (D).

When we run the basic constraints using `python dinner.py` with A and C we get the following output after 1 second:

sat

round:1

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 0, 1, 1, 0, 0, 0, 1]
[1, 0, 0, 1, 0, 0, 1, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

round:2

```
[1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 1, 0, 1, 1, 1]
```

round:3

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 1, 1, 1, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 1, 0, 0, 0, 1, 0, 1, 1]
```

round:4

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 1, 1, 1, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 1, 1, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

round:5

```
[1, 1, 0, 0, 0, 1, 1, 0, 0, 1]
[0, 0, 1, 1, 1, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

When we run the basic constraints using `python dinner.py` with A and D we get the following output after around 1 second:

sat

round:1

```
[1, 1, 0, 0, 1, 1, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 1, 0, 0, 1, 0, 1, 1]
```

round:2

```
[1, 1, 1, 0, 0, 0, 0, 0, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 1, 1, 1, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

round:3

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 1, 1, 0, 0, 1, 1]
[0, 1, 1, 1, 0, 0, 1, 1, 0, 0]
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

round:4

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 1, 0, 1, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

round:5

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 1, 1, 1, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 1, 0, 1, 1, 1]
```

When we run the basic constraints using `python dinner.py` with A and both C and D we get the following output after around 42 seconds:

unsat

2. Show that (B) is possible in combination with both (C) and (D).

When we run the basic constraints using `python dinner.py` with C and D we get the following output after around 1.5 seconds :

sat

round:1

```
[1, 1, 0, 1, 1, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 1, 1, 1, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

round:2

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 0, 1, 1, 0, 0, 0, 1]
[1, 0, 0, 1, 0, 0, 1, 1, 1, 0]
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

round:3

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 0, 0, 0, 1, 1, 1, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

round:4

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 1, 1, 1, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 1, 0, 1, 1, 1]
```

round:5

```
[1, 1, 1, 0, 0, 1, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 1, 0, 1, 1]
```

Generalization

The problem can be generalised by changing the number of people,houses and rounds (these are constants in our code). Although the number of people must remain dividable across rooms of 5 people each and the number of people must consist out of couples so NO_PEOPLE must be a multiple of 10, and therefore the number of rooms must be $NO\_PEOPLE/5$. Although the constraint for the guest limit is easily changed. (This change was not needed for this exercise, agile development and all that.)

Problem 4: Program Safety

Problem:

Consider the following program:

```
a := 1; b := 1;
for i := 1 to 10 do
```

```
   if ?  then a := a + 2b; b := b + i else b := a + b; a := a + i;
if b = 700 + n then crash
```

Here ' ?' is an unknown test that may yield false or true in any situation. Note that the test on crash is outside the loop, so is only tested at the end. Establish for which values of n = 1, 2, . . . , 10 it is safe, that is, will not reach 'crash'. Show for one of the non-safe values of n how b = 700 + n can be reached.

Solution:

Since we want to know what the crash values are we will be negating the problem and thus searching for possible values where (b=700+n). If we find such a solution we can provide it as a non-safe value.

We use the following data structure, we have a variable for a ,b, i each round and a 0 or a 1 as a Boolean denoting what the unknown condition was to lead to this state. this Boolean for the initial 0 state is ignored.

We store these values for the different rounds in a list of size 11 (10 for the iterations of the loop and 1 for the initial state). we call this list $iter$ we also store NO_ROUNDS which is the total number of rounds, in this case 11 (10 for the loops and 1 for the in-ital state). we use the following constraints:
Initial state, a b and i are all 1:

$$iter_{0,a} == 1 \wedge iter_{0,b} == 1 \wedge iter_{0,i} == 1$$

Adjacent states have a valid transition, meaning that for each state the next state corresponds either by $a := a + 2b; b := b + i$ or $b := a + b; a := a + i$ here we make sure to also set our Boolean. We define a lambda function:

$$\begin{aligned} true\_case_r = (&iter_{r+1,a} == iter_{r,a} + 2 * iter_{r,b} \wedge iter_{r+1,b} == iter_{r,b} + iter_{r,i} \\ &\wedge iter_{r+1,i} == iter_{r,i} + 1 \wedge iter_{r+1,Boolean} == 1) \end{aligned}$$

We define a lambda function:

$$\begin{aligned} false\_case_r = (&iter_{r+1,a} == iter_{r,a} + iter_{r,i} \wedge iter_{r+1,b} == iter_{r,a} + iter_{r,b} \\ &\wedge iter_{r+1,i} == iter_{r,i} + 1 \wedge iter_{r+1,Boolean} == 0) \end{aligned}$$

For all rounds

$$\sum_{r=0}^{NO\_ROUNDS-1} (true\_case_r \vee false\_case_r)$$

Final state is 700+n

$$iter_{NO\_ROUNDS-1,b} == 700 + n$$

Using these conditions we can let the sat solver check if for a given value of n there is a case where the last b is $700 + n$ if it finds one we know that the program can crash. if it is unsat we know that our program can never crash.

We have one more condition to help speed up the program a tiny bit. We check if all the values of b are smaller or equal to $700 + n$, we can do this because we see that b can never decrease. This allows us to trow out candidates quicker so that speeds it up a bit.

$$\sum_{r=0}^{NO\_ROUNDS} (iter_{r,b} <= 700 + n)$$

To get our final solution we run this solver for all n-s by executing `python pragramsafety.py` and store the unsat n-s in a list and display one route when it's satisfiable. This gives us the following result after around 1.5 seconds:

non crashing n-s = [2, 3, 5, 7, 9]

To show that the program crashes for $n = 1$ we show the following path for the if-statements with the values for $[[a, b, i, ?]]$ where ? is the value for the if-statement condition. If this is 1 we set it to be true, if 0 we set it to be false.

```
round:0 [[1], [1], [1], [0]]
round:1 [[2], [2], [2], [0]]
round:2 [[4], [4], [3], [0]]
round:3 [[12], [7], [4], [1]]
round:4 [[26], [11], [5], [1]]
round:5 [[48], [16], [6], [1]]
round:6 [[80], [22], [7], [1]]
round:7 [[87], [102], [8], [0]]
round:8 [[291], [110], [9], [1]]
round:9 [[300], [401], [10], [0]]
round:10 [[310], [701], [11], [0]]
```

As we can see, after round 10 the value for b is 701 which is equal to $700 + n$ for $n = 1$ so this path would crash.

Generalization

This solution can be generalized for any number of rounds where we will the solver will check for 700+n for the last round. For bigger values of NO_ROUNDS we expect the solver to get slower since it has more values to check. we also expect our optimisation condition to save more time with more rounds since the values can grow quicker. changing the value of n can also easily be done.
The pattern we used can also be used to cover other functions than $a := a + 2b; b := b + i$ or $b := a + b; a := a + i$. Looking at $true\_case$ and $false\_case$ we can see a simple pattern where $a := a + 2b$ becomes $iter_{r+1,a} == iter_{r,a} + 2 * iter_{r,b}$ where we say that a in loop $r + 1$ is defined as a in loop r plus two times b in loop r. This approach also works for similar variable assignments, although more variables might be needed.