House (Re-)Configuration Problem (HRP)

Original sources:

- Mayer, Wolfgang; Bettex, Marc; Stumptner, Markus; Falkner, Andreas A. (2009): On Solving Complex Rack Configuration Problems using CSP Methods. Proceedings of the IJCAI 2009 Workshop on Configuration (ConfWS 2009). Available from https://kse.cis.unisa.edu.au/wp/wp-content/papercite-data/pdf/confws-mayers09.pdf
- Friedrich, Gerhard; Ryabokon, Anna; Falkner, Andreas A.; Haselböck, Alois; Schenner, Gottfried; Schreiner, Herwig (2011): (Re)configuration using Answer Set Programming. Proceedings of the IJCAI 2011 Workshop on Configuration (ConfWS 2011). Available from http://ceur-ws.org/Vol-755/paper03.pdf
- Ryabokon, Anna (2015): Knowledge-Based (Re)Configuration of Complex Products and Services. Dissertation. Alpen-Adria-Universität, Klagenfurt. Available from https://netlibrary.aau.at/urn:nbn:at:at-ubk:1-26431

Parts of this problem description and all figures have been copied from Anna Ryabokon's dissertation. The problem description has been extended and revised by Richard Taupe and Antonius Weinzierl, and has been submitted with the consent of Anna Ryabokon. The original encoding by Anna Ryabokon and Gerhard Friedrich has been slightly revised and used with their consent.

Problem Description

The House (Re-)Configuration Problem (HRP) is an abstract version of (re)configuration problems occurring in practice. In its general form, the HRP is a reconfiguration problem. The task is, given a legacy configuration, to find an (optimal) reconfiguration satisfying various constraints. If the legacy configuration is empty, a reconfiguration problem in this sense degenerates to a configuration problem. For reasons of presentation, we first introduce this special case of the HRP, the House Configuration Problem, and afterwards the more general House Reconfiguration Problem.

House Configuration Problem (HCP)

Let P be a set of persons, T be a set of things and $\mathit{owner}\colon T \to P$ be the ownership function mapping each thing to the person that owns it. Moreover, let C and R be the sets of available cabinets and available rooms, respectively. The problem is to find:

- a set $C_S \subseteq C$ of cabinets in use,
- a set $R_S \subseteq R$ of rooms in use,
- an assignment function of things to cabinets $\mathit{cabinet}\colon T \to C_S$, and

- an assignment function of cabinets to rooms $room \colon C_S \to R_S$

such that the following requirements are satisfied:

- each thing must be stored in exactly one cabinet;
- a cabinet can contain at most 5 things;
- every cabinet can be placed in exactly one room;
- a room can contain at most 4 cabinets;
- every room can belong only to one person; and
- a room may only contain cabinets storing things of the owner of the room.

The tuple $(P, T, owner, C, R)$ defines an instance of the House Configuration Problem.

An input to the configuration problem, i.e. sets of persons, things and the ownership relation between persons and things, corresponds to customer requirements, while the constraints stated above correspond to so-called configuration requirements. In order to keep the example simple we only consider configurations of one house and represent all individuals using unique integer identifiers.

Let a sample house problem instance include two persons such that the first person owns five things numbered from 3 to 7 and the second person owns one thing 8. A solution to this house configuration problem instance is shown in the following figure, where the house configuration includes two rooms identified by the integers 15 and 16, two cabinets identified by 9 and 10, and six things numbered from 3 to 8.

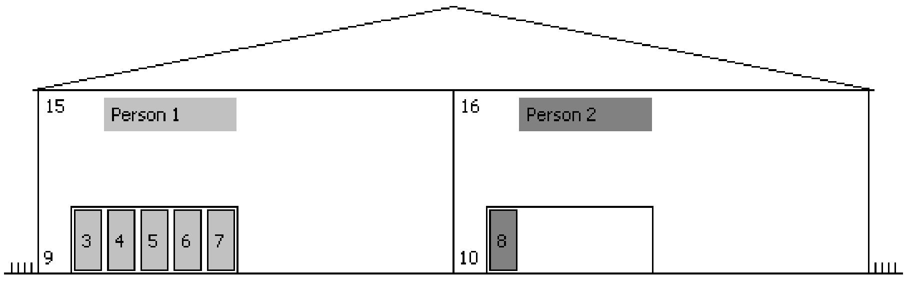

Figure 1: HCP Example

Reconfiguration is necessary whenever the customer requirements or configuration requirements are changed. For instance, if it becomes necessary to differentiate between long and short things.

House Reconfiguration Problem (HRP)

In the House Reconfiguration Problem, the requirements are changed: Now, things can be short or long and cabinets can be small or high.

Given a set of persons P, a set of things T, an ownership function $owner\colon T \to P$, the sets of cabinets C and rooms R, let $(C_L, R_L, cabinet_L, room_L)$ be a legacy solution of an instance of the House Configuration Problem (HCP). In addition, let $T_{long} \subseteq T$ denote a set of things that are considered as long. All other things are considered as short. Moreover, all legacy cabinets $c \in C_L$ are initially considered as small. The problem is to find a set of cabinets $C_S \subseteq C$, a set of high cabinets $C_{high} \subseteq C_S$, the set of rooms $R_S \subseteq R$ and assignment functions *cabinet* and *room* such that both the requirements of HCP and the following new requirements are satisfied:

- a cabinet $c \in C_S$ is either small or high (high cabinets are in $C_{high} \subseteq C_S$);
- a long thing $t \in T_{long}$ can only be put into a high cabinet $c \in C_{high}$;
- a small cabinet occupies 1 and a high cabinet 2 of 4 slots available in a room.

It holds that $C_L \subseteq C$ and $R_L \subseteq R$. It does not hold, however, that $C_L \subseteq C_S$ or that $R_L \subseteq R_S$: In a solution to a reconfiguration problem, legacy objects can be reused or discarded, and new objects can be introduced.

HCP is a special case of HRP in which $C_L = R_L = cabinet_L = room_L = \emptyset$.

In the case of the reconfiguration problem, the customer requirements are extended with a definition for each thing whether it is long or short. For instance, for the previously given house example the customer provides information that the things 3 and 8 are long; all others are short. Moreover, the first person gets an additional long thing 21. The changes to the legacy configuration are summarized in the following figure showing an inconsistent configuration, where thing 21 is not placed in any of the cabinets, and cabinets 9 and 10 are too small for things 3 and 8.

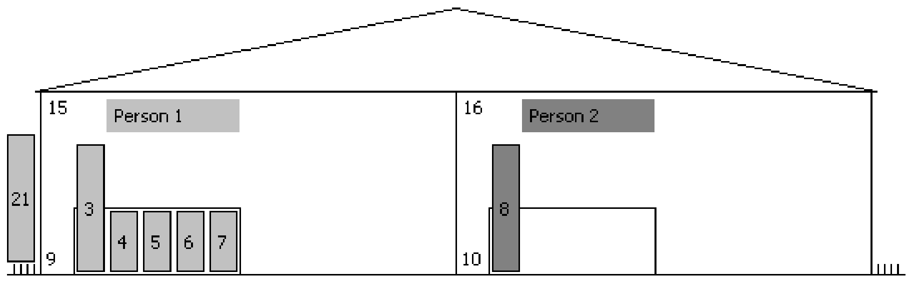

Figure 2: HRP Example

To obtain a solution which is shown in the following figure, the reconfiguration process changes the size of cabinets 9 and 10 to high and puts the new thing 21 into cabinet 9. A new small cabinet 22 is created for thing 7.

In the reconfiguration process every modification to the existing configuration (reusing, deleting and creating individuals and their relations) is associated with

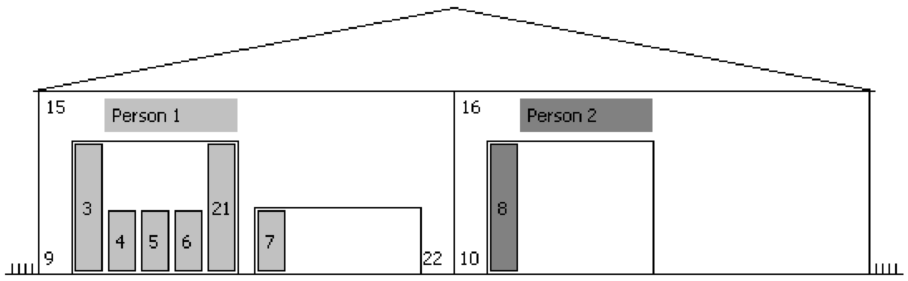

Figure 3: HRP Example Solution 1

some cost. Therefore, the reconfiguration problem is to find a consistent configuration by removing the inconsistencies and minimizing the costs involved. Different solutions will be found depending on the given modification costs specified by a customer. If, for example, the costs for adding a new high cabinet are less than the cost for changing an existing small cabinet into a high cabinet, then the previous solution should be rejected as its costs are too high. One of the solutions with less reconfiguration costs, shown in the following figure, includes two new cabinets 22 and 23, because the creation of new cabinets is cheaper than converting the existing small cabinets into high cabinets. Also it contains the empty cabinet 10, because it's cheaper to keep the cabinet than to delete it. Note, this behavior can be controlled by the domain-specific costs.

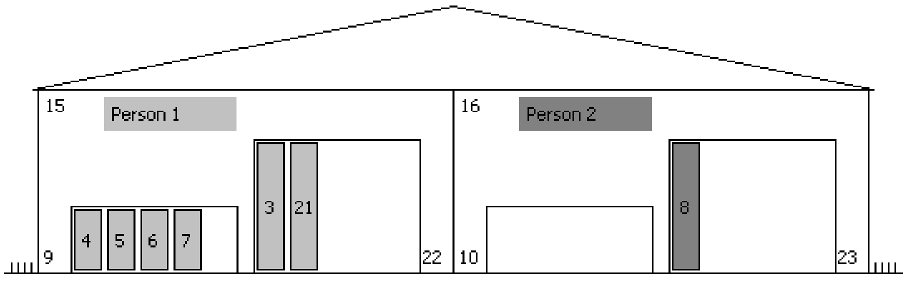

Figure 4: HRP Example Solution 2

Input Description

An instance consists of facts over the predicates that are mapped to concepts introduced above in the following table.[1]

[1]Note that input as well as output predicates often have other names than are used for the formal concepts introduced in this problem description. The reason for this is that the

For convenience of definitions, we define an additional function $\mathit{occupant}\colon R \to 2^P$ with $\mathit{occupant}(r) = \{p \mid \exists t\colon \mathit{room}(\mathit{cabinet}(t)) = r \text{ and } \mathit{owner}(t) = p\}$. Analogously, $\mathit{occupant}_L\colon R_L \to 2^P$ with $\mathit{occupant}_L = \{p \mid \exists t\colon \mathit{room}_L(\mathit{cabinet}_L(t)) = r \text{ and } \mathit{owner}(t) = p\}$.

ASP predicate	Concept	Description
`thingLong/1`	T_{long}	long things
`thingShort/1`	$T \setminus T_{long}$	short things (optional[2])
`cabinetDomainNew/1`	$C \setminus C_L$	IDs that can be used to create new cabinets
`roomDomainNew/1`	$R \setminus R_L$	IDs that can be used to create new rooms

In an HRP instance, the legacy configuration is given by facts over the `legacyConfig/1` predicate. This predicate is also used to define the sets of persons and things and the ownership function between them. Functions *inside* the `legacyConfig/1` are used to define the data structures as defined in the following table. For example, `legacyConfig(person(1))` means that $1 \in P$ and `legacyConfig(cabinet(9))` means that $9 \in C_L$.

ASP function	Concept	Description
`person/1`	P	persons
`thing/1`	T	things
`personTOthing/2`	owner^{-1}	mapping of persons (first argument) to owned things (second argument)
`cabinet/1`	C_L	cabinets in the legacy configuration
`room/1`	R_L	rooms in the legacy configuration
`cabinetTOthing/2`	$\mathit{cabinet}^{-1}$	placement of things (second argument) in cabinets (first argument)
`roomTOcabinet/2`	room^{-1}	placement of cabinets (second argument) in rooms (first argument)

Atoms of the following predicates, all of which are optional, define costs used for optimization as explained below in the section on Scoring Schema:

ASP predicate	defines the cost associated with …
`cabinetTOthingCost/1`	placing a thing in a cabinet
`roomTOcabinetCost/1`	placing a cabinet in a room

problem description has been revised while aiming to stay compatible with existing problem instances.

[2]Since `thingShort/1` is only optionally given in the input, an encoding should include the rule `thingShort(T) :- legacyConfig(thing(T)), not thingLong(T).`

ASP predicate	defines the cost associated with …
`personTOroomCost/1`	assigning a room to a person (in *occupant*)
`reuseCabinetTOthingCost/1`	reusing an existing placement of a thing in a cabinet
`reuseRoomTOcabinetCost/1`	reusing an existing placement of a cabinet in a room
`reusePersonTOroomCost/1`	reusing an existing assignment of a room to a person (in *occupant*)
`reuseCabinetAsSmallCost/1`	reusing an existing cabinet while keeping its size *small*
`reuseCabinetAsHighCost/1`	reusing an existing cabinet while changing its size to *high*
`reuseRoomCost/1`	reusing an existing room
`removeCabinetTOthingCost/1`	removing a thing from a cabinet
`removeRoomTOcabinetCost/1`	removing a cabinet from a room
`removePersonTOroomCost/1`	un-assigning a person from a room (in *occupant*)
`removeCabinetCost/1`	removing an existing cabinet entirely
`removeRoomCost/1`	removing an existing room entirely
`cabinetHighCost/1`	creating a new high cabinet
`cabinetSmallCost/1`	creating a new small cabinet
`roomCost/1`	creating a new room

Output Description

A solution can be read off from atoms of the following predicates:

ASP predicate	Concept	Description
`cabinet/1`	C_S	cabinets used in the solution
`cabinetHigh/1`	C_{high}	high cabinets
`cabinetSmall/1`	$C_S \setminus C_{high}$	small cabinets
`room/1`	R_S	rooms used in the solution
`cabinetTOthing/2`	*cabinet*	placement of things (second argument) in cabinets (first argument)
`roomTOcabinet/2`	*room*	placement of cabinets (second argument) in rooms (first argument)

Examples

Let us review the HRP example already introduced above.

- $P = \{1, 2\}$
- $T = \{3, ..., 8\}$
- $owner = \{3 \mapsto 1, ..., 7 \mapsto 1, 8 \mapsto 2, 21 \mapsto 1\}$
- $C_L = \{9, 10\}$
- $R_L = \{15, 16\}$
- $cabinet_L = \{3 \mapsto 9, \dots, 7 \mapsto 9, 8 \mapsto 10\}$
- $room_L = \{9 \mapsto 15, 10 \mapsto 16\}$
- $C = C_L$
- $R = R_L \cup \{17, 18, 19, 20\}$

This HRP instance can be defined by the following set of facts, which also includes cost factor definitions:

```
legacyConfig(person(1..2)).
legacyConfig(thing(3..8)).
legacyConfig(thing(21)).
legacyConfig(personTOthing(1,3..7)).
legacyConfig(personTOthing(1,21)).
legacyConfig(personTOthing(2,8)).

legacyConfig(room(15)).
legacyConfig(roomTOcabinet(15,9)).
legacyConfig(room(16)).
legacyConfig(roomTOcabinet(16,10)).
legacyConfig(cabinet(9)).
legacyConfig(cabinetTOthing(9,3..7)).
legacyConfig(cabinet(10)).
legacyConfig(cabinetTOthing(10,8)).

thingLong(3).
thingLong(8).
thingLong(21).

cabinetDomainNew(22..23).
roomDomainNew(17..20).

reuseCabinetTOthingCost(0).
reuseRoomTOcabinetCost(0).
reusePersonTOroomCost(0).
reuseCabinetAsHighCost(3).
reuseCabinetAsSmallCost(0).
reuseRoomCost(0).
removeCabinetTOthingCost(2).
removeRoomTOcabinetCost(2).
removePersonTOroomCost(2).
removeCabinetCost(2).
removeRoomCost(2).
```

```
cabinetHighCost(100).
cabinetSmallCost(10).
roomCost(5).
```

This instance has four optimal solutions, one of which is:[3]

```
cabinet(9) cabinet(10) cabinet(22) room(15) room(16)
cabinetTOthing(9,3) cabinetTOthing(9,4) cabinetTOthing(9,5)
cabinetTOthing(9,6) cabinetTOthing(10,8) cabinetTOthing(9,21)
cabinetTOthing(22,7) roomTOcabinet(15,9) roomTOcabinet(16,10)
roomTOcabinet(15,22) cabinetHigh(9) cabinetHigh(10) cabinetSmall(22)
```

Scoring Schema

The following table shows sets whose cardinality is relevant for scoring, predicates defining associated cost factors (weight) and default cost factors.

Relevant set	ASP predicate
$cabinet \backslash cabinet_L$	`cabinetTOthingCost/1`
$room \backslash room_L$	`roomTOcabinetCost/1`
$occupant \backslash occupant_L$	`personTOroomCost/1`
$cabinet \cap cabinet_L$	`reuseCabinetTOthingCost/1`
$room \cap room_L$	`reuseRoomTOcabinetCost/1`
$occupant \cap occupant_L$	`reusePersonTOroomCost/1`
$(cabinet_S \backslash cabinet_{high}) \cap cabinet_L$	`reuseCabinetAsSmallCost/1`
$cabinet_{high} \cap cabinet_L$	`reuseCabinetAsHighCost/1`
$R_S \cap R_L$	`reuseRoomCost/1`
$cabinet_L \backslash cabinet$	`removeCabinetTOthingCost/1`
$room_L \backslash room$	`removeRoomTOcabinetCost/1`
$occupant_L \backslash occupant$	`removePersonTOroomCost/1`
$C_L \backslash C_S$	`removeCabinetCost/1`
$R_L \backslash R_S$	`removeRoomCost/1`
$C_{high} \backslash C_L$	`cabinetHighCost/1`
$C_S \backslash (C_{high} \cup C_L)$	`cabinetSmallCost/1`
$R_S \backslash R_L$	`roomCost/1`

The *score* for a valid solution is given by the cardinalities of the sets given in the table above, weighted by the respective cost factors. The default value for undefined cost factors is 0. This value is used to determine the *cost* of a solution, while its *priority* is always 0.

[3]The answer set has been projected to include only atoms over output predicates.