Knowledge-based (re)configuration: an ASP approach

Kostyantyn Shchekotykhin

Alpen-Adria-Universität Klagenfurt

Outline

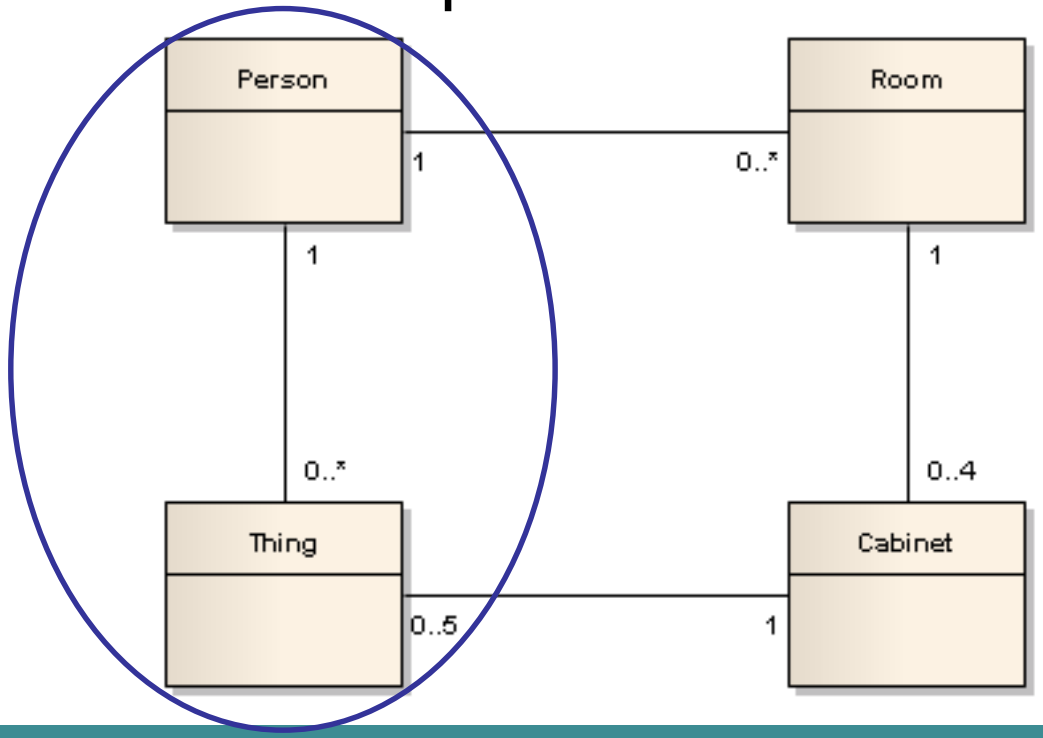
- House (re)configuration problem
  - Configuration problem in ASP
  - Extension for the reconfiguration problem
  - Performance comparison with CSP
- Partner Units Problem
  - Problem description
  - Encoding of the configuration problem
  - Heuristics and performance analysis

House (re)configuration problem

Configuration problem

- Given: a model (classes and their relations), its partial instantiation and requirements
- Find: a complete instantiation of the model that satisfies all requirements

Input



(Siemens©)

House configuration problem

Customer requirements:

Declaration of available persons, things and ownership relations between them

Configuration requirements:

1. each thing must be stored in exactly one cabinet
2. a cabinet can contain at most 5 things
3. every cabinet must be placed in exactly one room
4. a room can contain at most 4 cabinets
5. each room belongs to a person
6. and a room may only contain cabinets storing things of the owner of the room

Goal: store all things in a house such that the set of requirements is fulfilled

Definitions

- Classes – unary predicates: `thing`, `person`, `cabinet`, `room`
- Relations – binary predicates: `p2t, t2c, c2r, p2r`
- Input – facts:
  - Persons, things and relations between them
  - Domains of cabinets and rooms – unary predicates: `cabinetDomain` and `roomDomain`
  - Solution schema predicates: `cabinet, room, t2c, c2r and p2r`

Sample customer requirements

- Customer requirements include:
  - Person 1 owns things 3,4,5,6,7 and
  - Person 2 owns thing 8
  - There are 2 cabinets and 2 rooms in the components catalog
- Encoding

```
person(1). person(2). thing(3..8).

p2t(1,3..7). p2t(2,8).

cabinetDomain(9..10). roomDomain(15..16).
```

ASP Encoding (Gringo 4)

```
% each thing must be stored in exactly one
cabinet and a cabinet can contain at most 5
things

1 {t2c(T,C): cabinetDomain(C)} 1 :- thing(T).
:- cabinet(C), 6 {t2c(T,C) : thing(T)}.
cabinet(C) :- t2c(_,C).
```

ASP Encoding

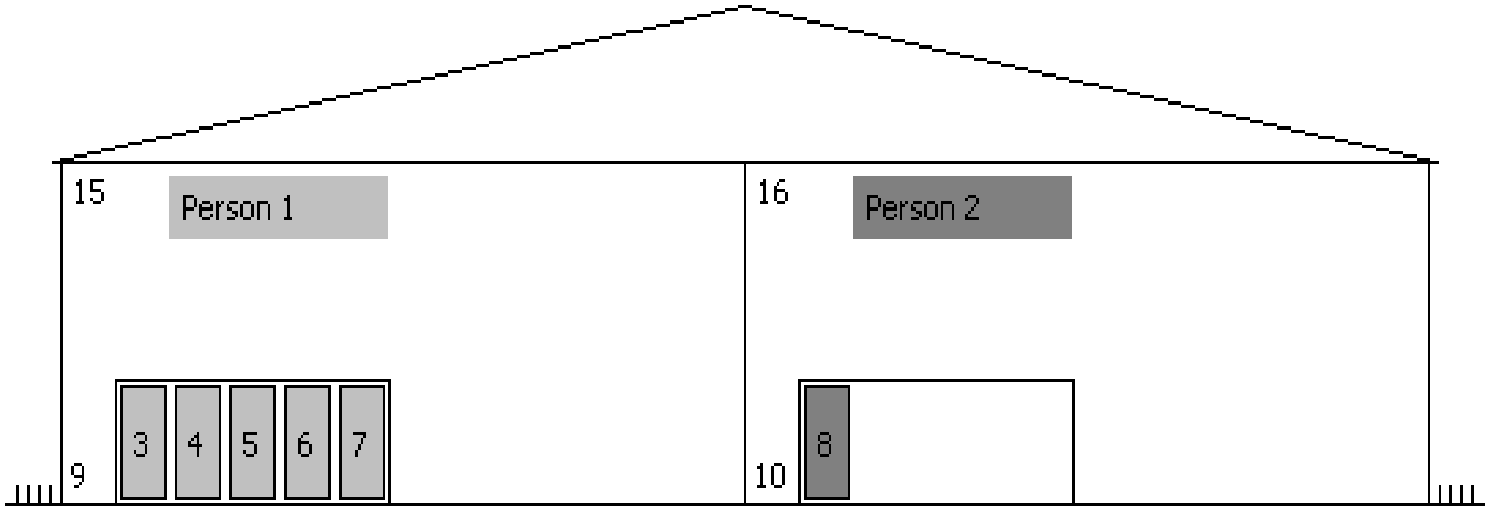
ALPEN-ADRIA UNIVERSITÄT
KLAGENFURT | WIEN GRAZ

```
% each room belongs to a person

p2r(P,R) :- p2t(P,T), t2c(T,C), c2r(C,R).

% a room may only contain cabinets storing
things of one person

:- p2r(P1,R), p2r(P2,R), P1 != P2.
```

Sample solution: ASP encoding

- Solution:

```
cabinet(9). cabinet(10). room(15). room(16).
p2r(1,15). p2r(2,16). c2r(9,15). c2r(10,16).
t2c(3,9).  t2c(8,10). ...
```



House reconfiguration problem I

Additional customer requirements:

- definitions of long and short things – unary predicates `thingLong` and `thingShort`

```
thingLong(3). ...thingShort(7). thingLong(8).
thing(21). thingLong(21).
```

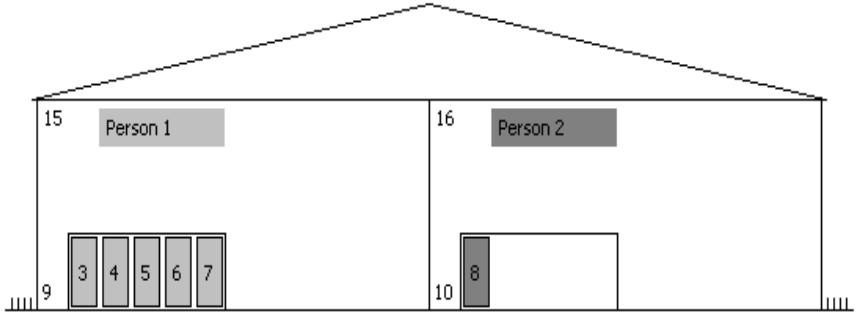
- definitions of new things

```
thing(21). p2t(1,21).
```

Legacy configuration

- Use terms to encode the legacy solution (reification of a solution)
- Unary predicate `legacyConfig`

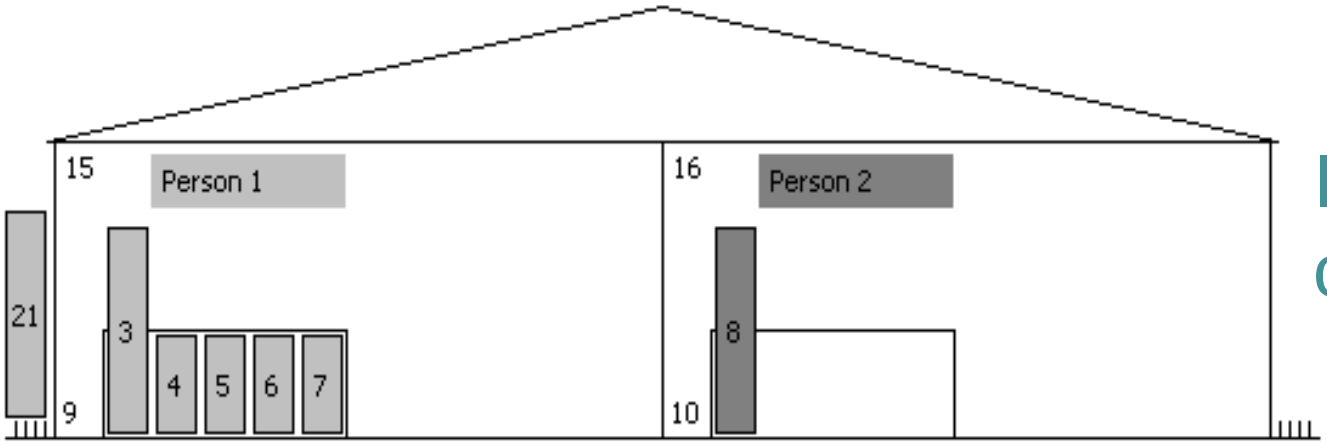
```
legacyConfig(cabinet(9)).
legacyConfig(cabinet(10)). ...
legacyConfig(c2r(9,15)). ...
legacyConfig(t2c(3,9)). ...
```



House reconfiguration problem II

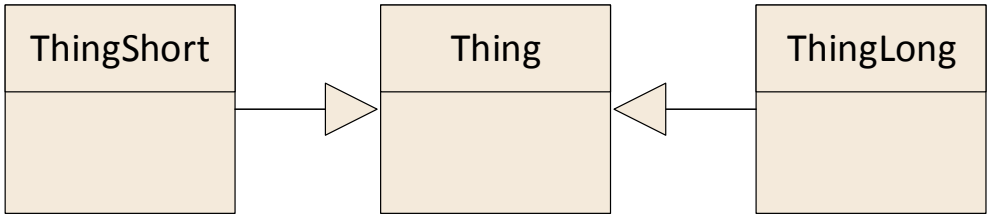
Additional configuration requirements:

- a cabinet is either small or high
- a long thing can only be put into a high cabinet
- a small cabinet occupies 1 and a high cabinet 2 of 4 cabinet places available in a room
- all legacy cabinets are small



Inconsistent configuration

Additional constraints I




```
thing(X) :- thingLong(X).
thing(X) :- thingShort(X).
1 {thingLong(X); thingShort(X)} 1 :- thing(X).
% same for cabinets
1 {cabinetHigh(X); cabinetSmall(X)} 1 :-
                                    cabinet(X).
cabinetHigh(C) :- thingLong(T), t2c(T,C).
```

Additional constraints II

```
% 4 slots in a room

cabinetSize(X,1) :- cabinetSmall(X).
cabinetSize(X,2) :- cabinetHigh(X).
:- 5 <= #sum{S,C : cabinetSize(C,S),
                   c2r(C,R)}, room(R).

% all legacy cabinets are small: holds only
for cabinets reused in a new configuration
```

Transformation rules

- Parts of the legacy configuration can either be reused or deleted

$$\forall \bar{X}\left[legacyConfig\left(p(\bar{X})\right) \rightarrow reuse\left(p(\bar{X})\right) \vee delete\left(p(\bar{X})\right)\right]$$

  where $p(\bar{X})$ is an n-place predicate p used in a configuration with a vector of terms $\bar{X}$

- Reused individuals, relations are asserted

$$\forall \bar{X}[reuse\left(p(\bar{X})\right) \rightarrow p(\bar{X})]$$

- Deleted individuals, relations should be excluded

$$\forall \bar{X}\left[delete\left(p(\bar{X})\right) \wedge p(\bar{X}) \rightarrow \perp\right]$$

House transformation rules

- Some parts of the legacy configuration must be preserved

```
person(P) :- legacyConfig(person(P)).
```

- Other parts can either be reused or deleted

```
1 {reuse(t2c(T,C)); delete(t2c(T,C))} 1 :-
                        legacyConfig(t2c(T,C)).

cabinetDomain(X) :- legacyConfig(cabinet(X)).
1 {reuse(cabinet(X)); delete(cabinet(X))} 1
                    :- legacyConfig(cabinet(X)).
```

Objective function I

- Introduce costs for each action:
  - Creation costs: individuals and relation tuples absent in the legacy configuration
  - Reuse costs: individuals and relation tuples present in both reconfiguration and legacy configuration
  - Deletion costs: individuals and relation tuples of legacy configuration absent in the reconfiguration
- Optimization criterion: minimize sum of all costs

Objective function II

- Creation costs

$$\forall \bar{X}, \bar{Y}, W\left[new(p(\bar{X})) \wedge p(\bar{X}) \wedge \alpha(\bar{X}, \bar{Y}, W) \rightarrow cost\left(create\left(p(\bar{X})\right), W\right)\right]$$

- Reuse costs

$$\forall \bar{X}, \bar{Y}, W\left[reuse(p(\bar{X})) \wedge \beta(\bar{X}, \bar{Y}, W) \rightarrow cost\left(reuse\left(p(\bar{X})\right), W\right)\right]$$

- Deletion costs

$$\forall \bar{X}, \bar{Y}, W\left[delete\left(p(\bar{X})\right) \wedge \gamma(\bar{X}, \bar{Y}, W) \rightarrow cost\left(delete\left(p(\bar{X})\right), W\right)\right]$$

where α, β, γ are conjunctions of atoms defining case specific costs; new is domain predicate which is *true* for every $p(\bar{X})$ which is not in a legacy configuration

- Optimization criterion: $\min \sum_{W \in \bar{W}} W$

  where $\bar{W} = \{W : cost(op(p(\bar{X})), W), p \in SolutionSchema\}$ and $op \in \{create, reuse, delete\}$

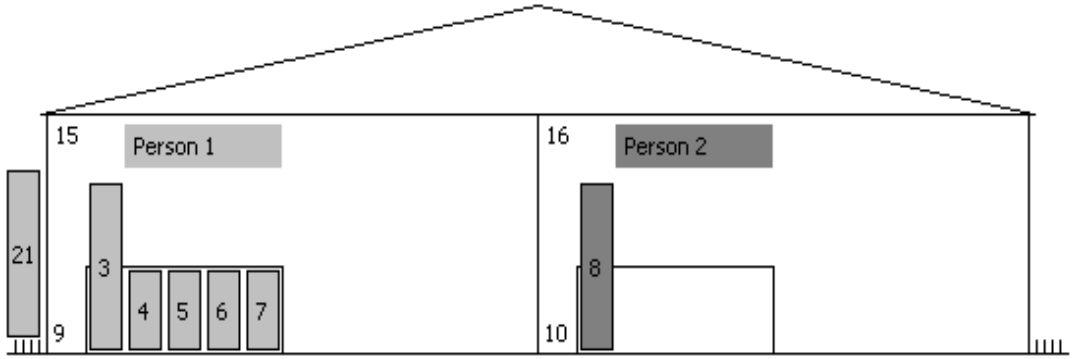
House objective function

- Differentiate between domain elements used in legacy solution and new once
- Encoding using xDomainNew/1 predicate
  - x stands for a type, e.g. cabinet

```
cost(create(cabinetHigh(X)),W) :-
      cabinetHigh(X), cabinetHighCost(W),
                      cabinetDomainNew(X).
```

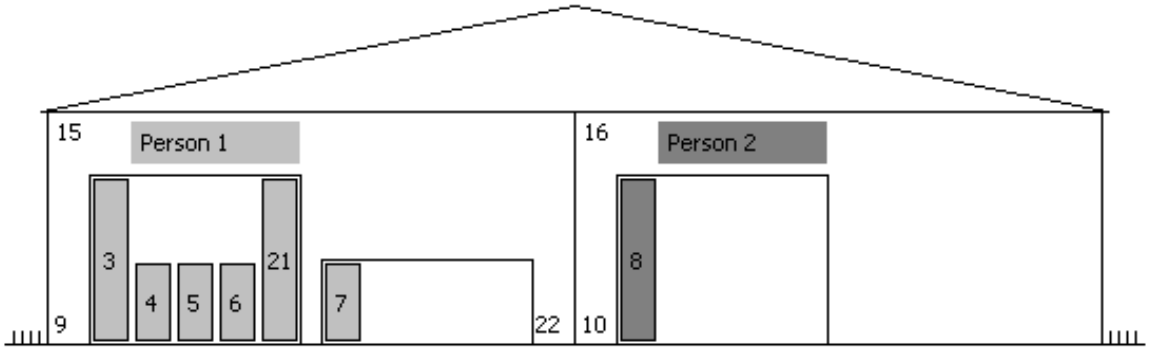
Reconfiguration solution I

Initial state



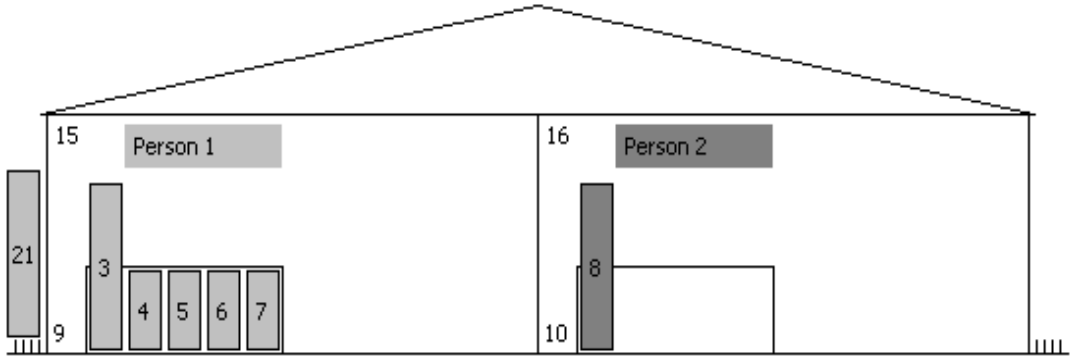
- Reusing and changing a cabinet to high costs less than creating a new one
- Creation of a high cabinet costs more than reuse of an old one modified to high

Reconfiguration



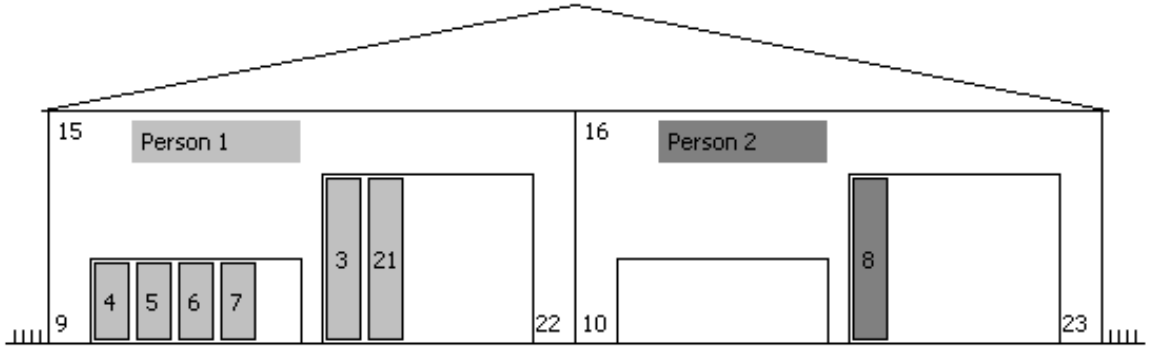
Reconfiguration solution II

Initial state



- Creation of a new cabinet costs less than reuse of an existing cabinet and changing its size
- Deletion of a cabinet costs more than 0

Reconfiguration



Reconfiguration problems

- Empty: the legacy solution is empty
- Long: some of the things are declares as long
- New room: at least one new room must be created to find a solution
- Swap: any solution describes a rearrangement of cabinets in rooms

Modeling approaches

- ASP
  - Problem representation is encoded in *ASP (Gringo)* modeling language
  - ASP program is solved by Gringo 3.0.3 + clasp 2.0.2
- CP
  - CP model is written in *MiniZinc*
  - The instances were solved using Gecode 3.7.1, which is a Finite Domain solver written in C++

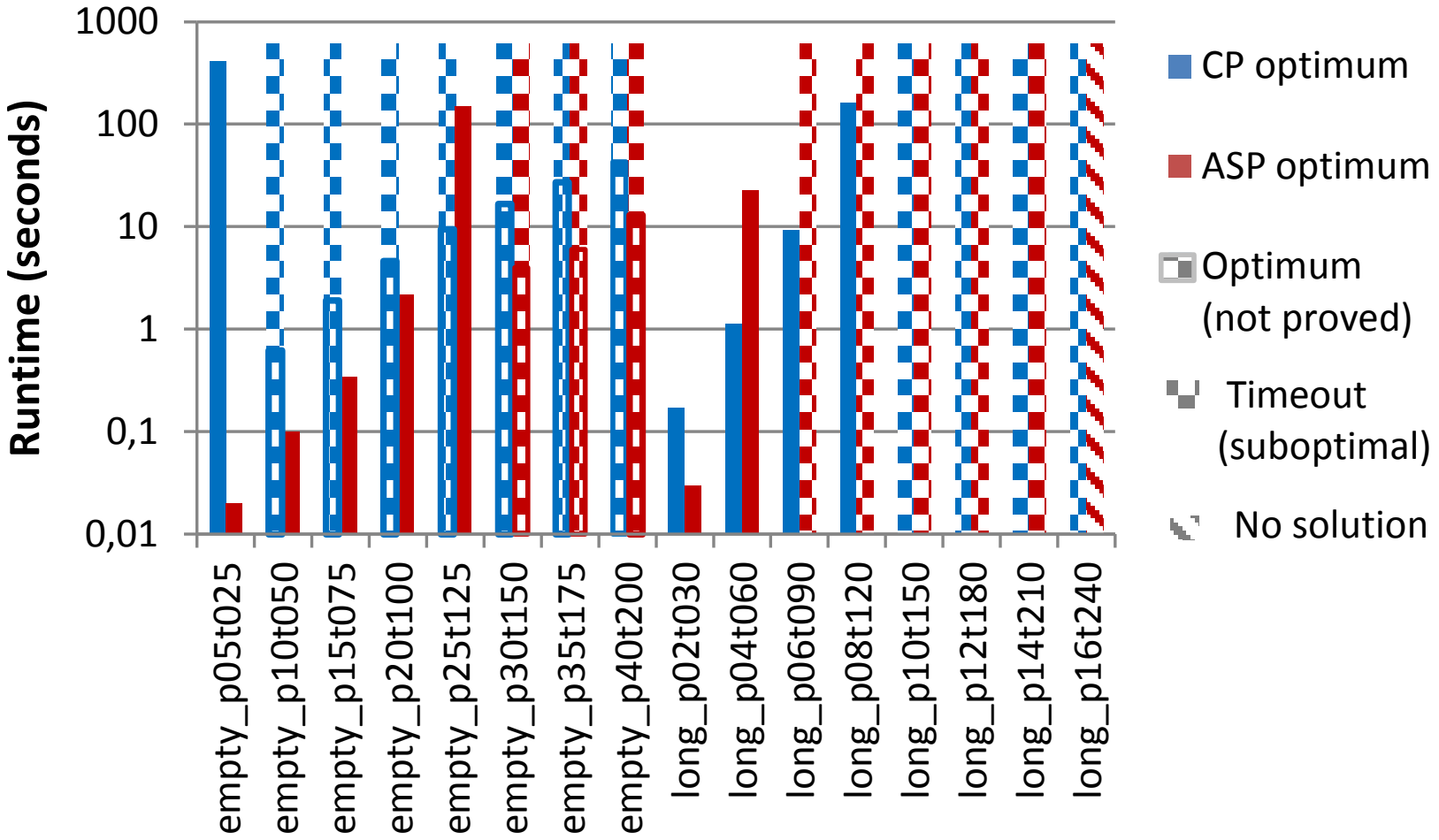
CP variable orderings I

- Fill cabinets first:
  - assign things to cabinets, i.e. fix values of the variables in the array `t2c`
  - variables in the array are selected using `first_fail` heuristic and values `indomain_min` heuristic
  - assign values to variables in `c2r`, i.e. place cabinets in rooms, using the same heuristics as above
  - the order of other variables is determined by the solver (Gecode)

CP variable orderings II

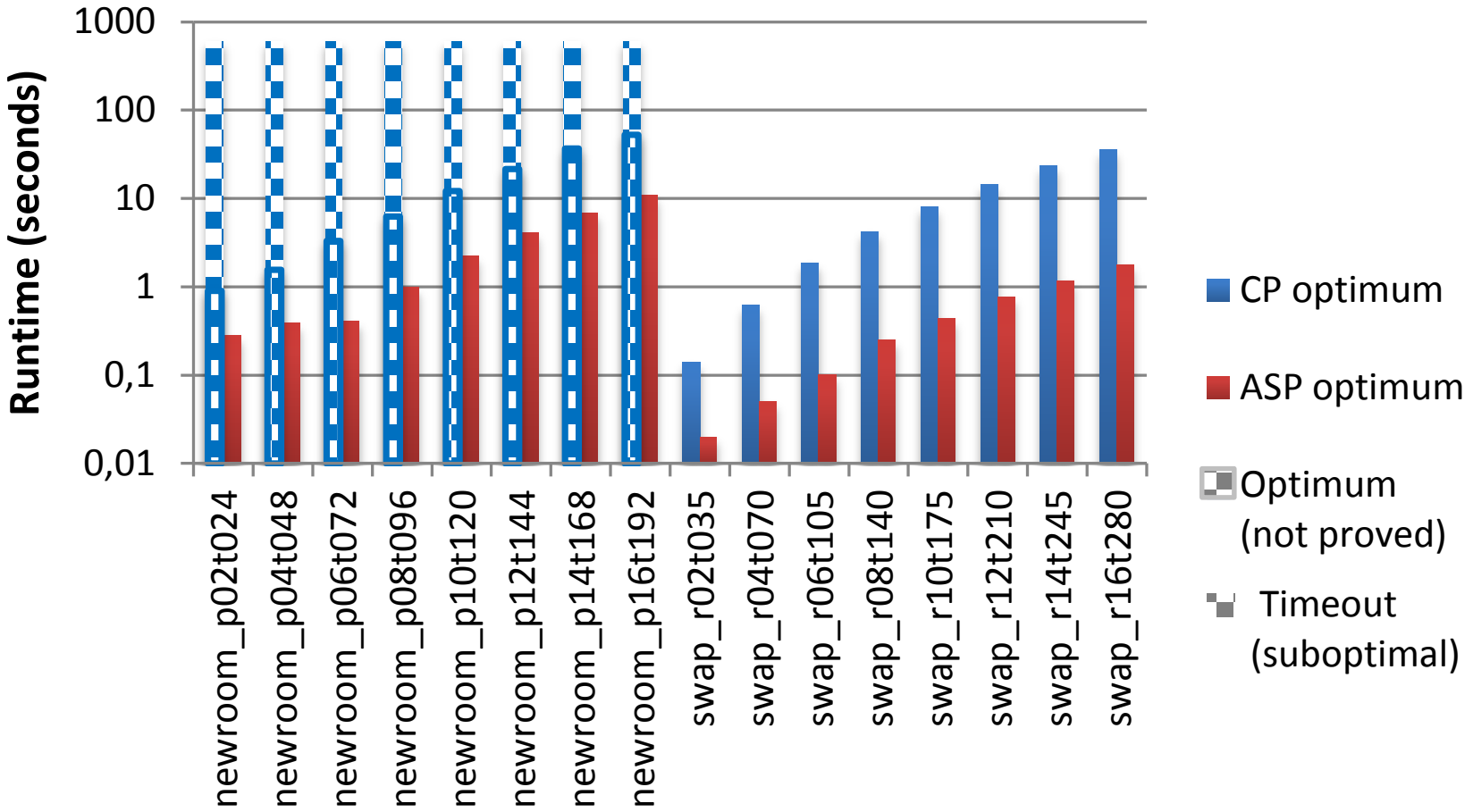
- Long things first:
  - for each person place first longs things into cabinets and then short things
  - after each fifth thing assignment place a cabinet into a room
  - `input_order` variable selection heuristic is used to apply the ordering described above
  - `indomain_min` heuristic is used to select the values
  - the order of other variables is determined by the solver (Gecode)

Evaluation I



*only creation costs for individuals are taken into account, CP – long things first

Evaluation II



*only creation costs for individuals are taken into account, CP – fill cabinets first

Evaluation summary

The overall runtime of Clasp is better than runtime of Gecode. ASP outperformed CP on the Empty, New Room and Swap instances.

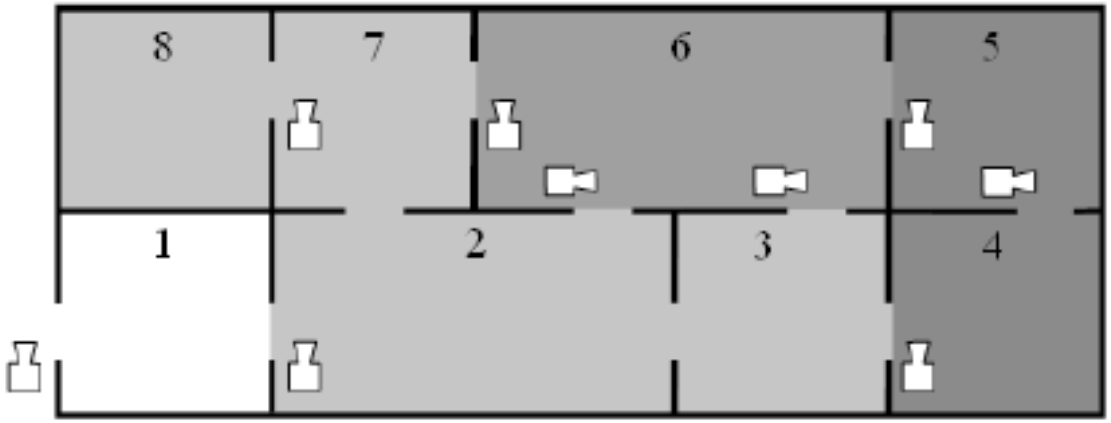
CP allows incorporation of problem-relevant heuristics by means of search annotations which improved the runtime for Long scenario.

In many cases CP was able to identify the optimal model, but failed to prove the optimality.

Partner Units Configuration problem

Partner Units Problem (PUP)

Given a consistent configuration of door sensors and zones, find a valid assignment of units that satisfies all requirements, minimizing the number of units used



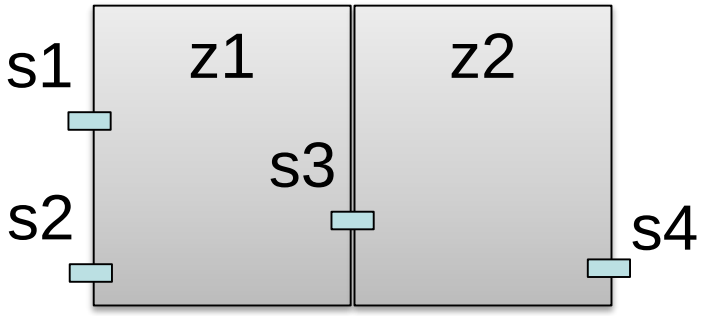
Requirements

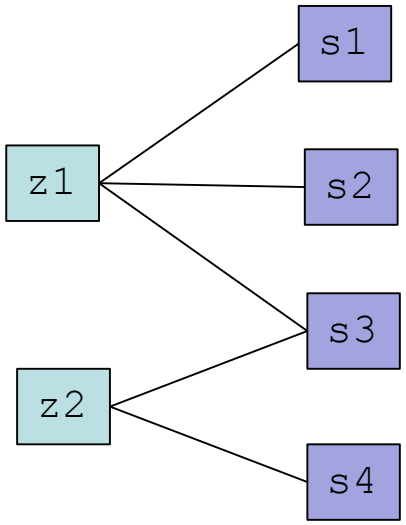
- each zone as well as each door sensor must be connected to exactly one unit;
- each unit can control at most two door sensors and at most two zones;
- if a unit controls a sensor that contributes to a zone controlled by another unit, then the two units must be connected directly, i.e. one unit becomes a partner unit of the other and vice versa;
- each unit can have at most *n* partner units

[ASP Competition, 2011]

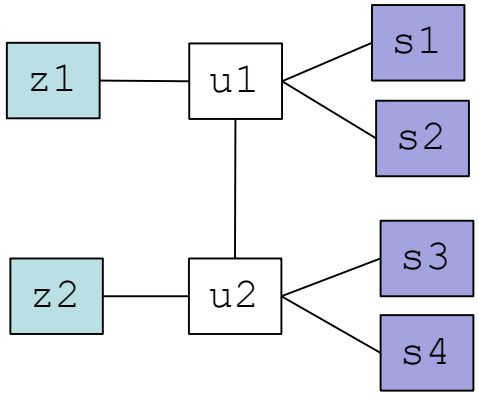
PUP example

Input

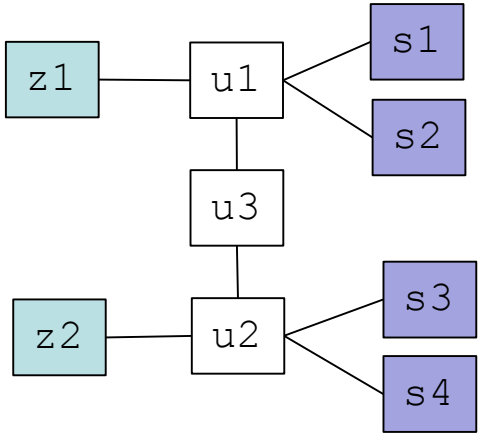




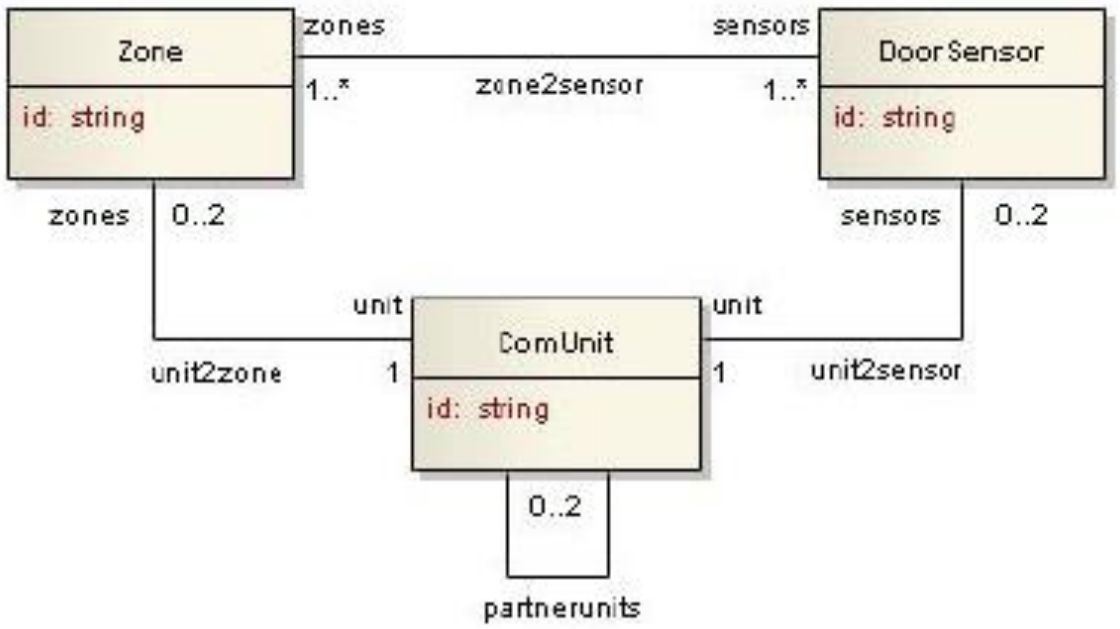
Solution



Not a solution



UML representation



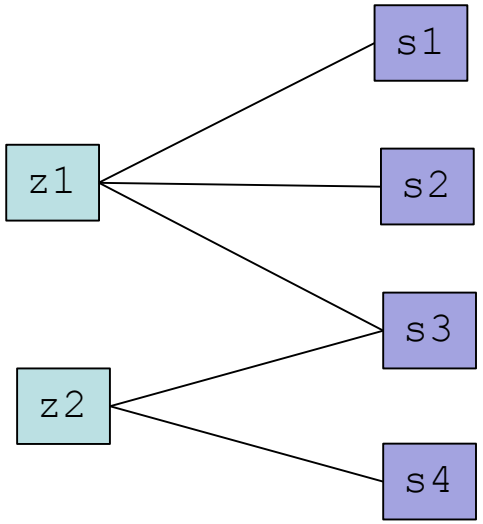
Relations

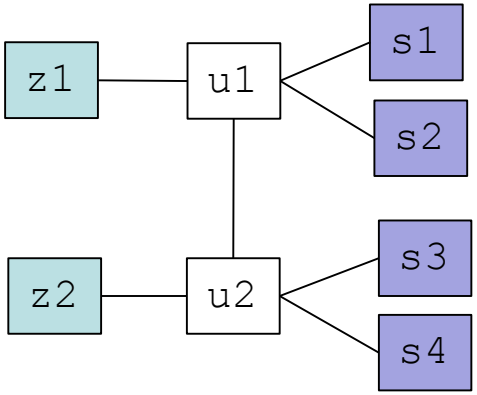
Input: `z2s/2, zone/1, sensor/1`

Output: `unit/1, u2z/2, u2s/2, pu/2`

ASP encoding I

```
z2s(z1,s1; z1,s2; z1,s3).
z2s(z2,s3; z2,s4).

zone(Z)   :- z2s(Z,S).
sensor(S) :- z2s(Z,S).

#const maxPU=2.
#const upper=7. #const lower=2.
```





$$lower = \left\lceil \max(\#zones, \#sensors) / 2 \right\rceil$$

$$upper = \#zones + \#sensors$$

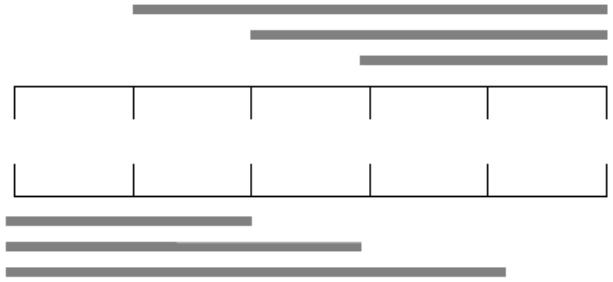
ASP encoding II

```
lower {unit(1..upper)} upper.

1 { u2z(U,Z) : unit(U) } 1 :- zone(Z).
:- unit(U), 3 { u2z(U,Z): zone(Z) }.

1 { u2s(U,D) : unit(U) } 1 :- sensor(D).
:- unit(U), 3 { u2s(U,D): sensor(D)}.

pu(U,P) :- u2z(U,Z), z2s(Z,D), u2s(P,D), U!=P.
pu(U,P) :- pu(P,U), unit(U), unit(P).
:- unit(U), maxPU+1 { pu(U,P): unit(P)}.
```

Test cases

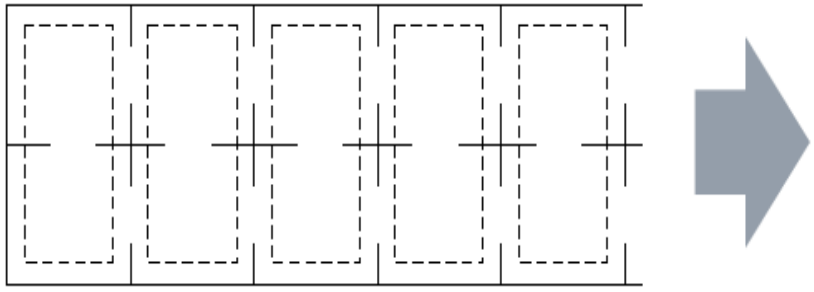
single-11:

11Z, 6S, 6U, 22 connections between zones and sensors, each door is a sensor



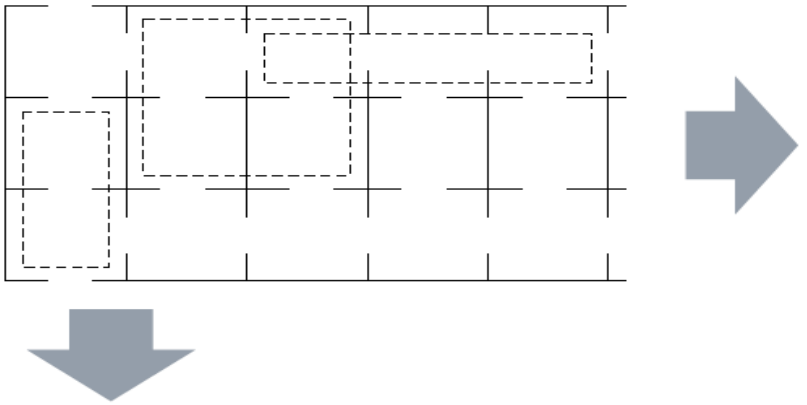
double, double variant:

- a double row of connected rooms, each room being a zone
- a variant has additional zones for each 2 connected rooms vertical to the row



triple:

- each room being a zone
- in some cases with additional 2 or 4 zones consisting of 2-4 rooms

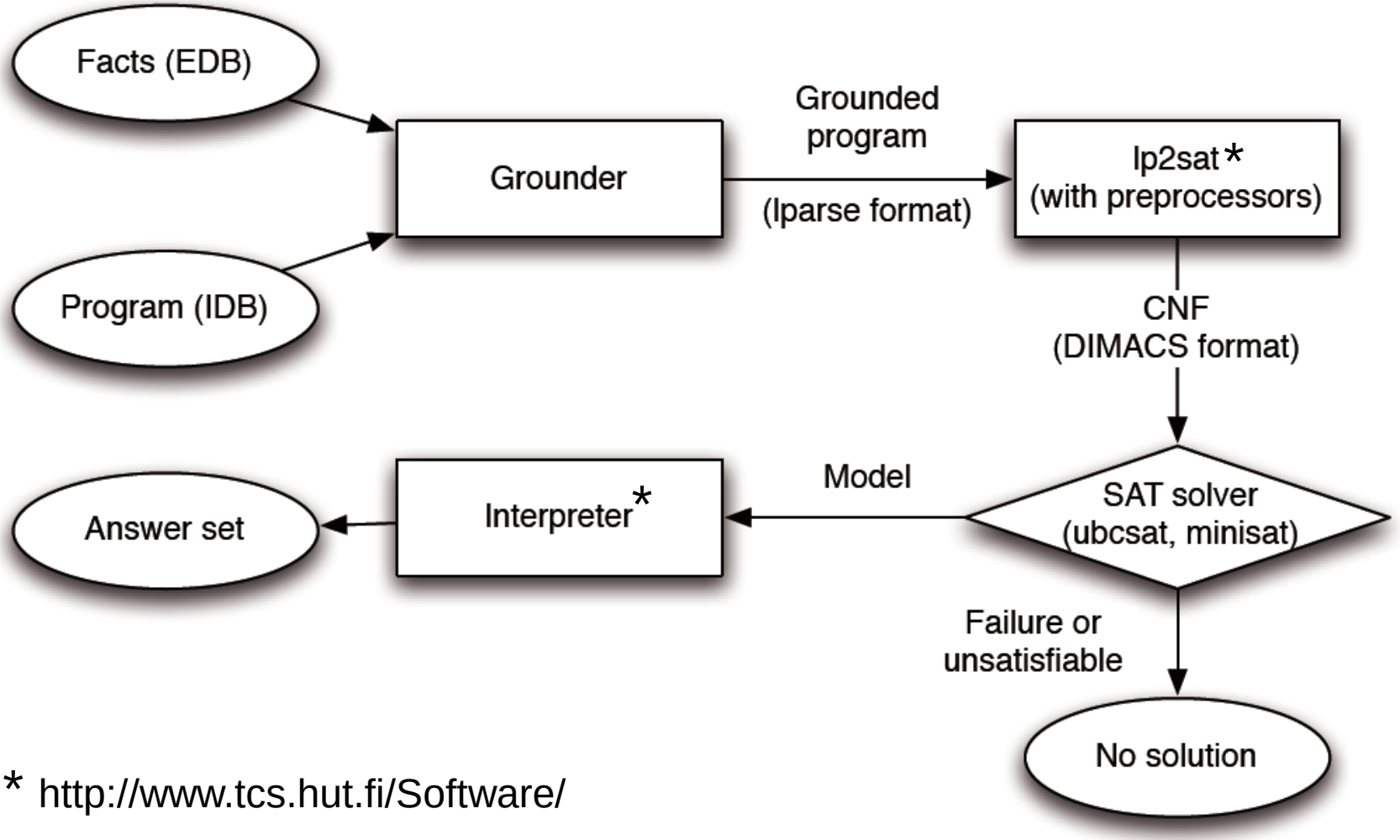


Evaluation results

EDB	Configuration	Cos*	Simple program
p10v3.edb	15U,30Z, 28D,3PU	00:15,0	00:59,65
p10v4.edb	30U,60Z, 58D,4PU	timeout	00:01,56
p20v4.edb	20U,30Z, 40D,4PU	00:16,0	00:00,25
p102.edb	14U,20Z, 28D,2PU	timeout	00:00,10
p203.edb	29U,40Z, 58D,3PU	00:40,0	timeout
p303.edb	44U,60Z, 88D,3PU	02:30,0	timeout, solved in 08:46 min
p403.edb	59U,80Z,117D,3PU	timeout	timeout
r10.edb	20U,32Z, 40D,4PU	00:02,0	00:00,42
r12.edb	20U,34Z, 40D,4PU	00:10,0	00:00,74
r20.edb	40U,64Z, 79D,4PU	<1 sec	00:41,68
r22.edb	40U,60Z, 79D,4PU	<1 sec	00:17,72
r30.edb	59U,90Z,118D,4PU	timeout	timeout, solved in 08:38 min

*Cos – Legacy constraints system, timeout 3 minutes

Stochastic local search



* http://www.tcs.hut.fi/Software/

Evaluation details

Solver: UBCSAT[1,2]

Tested algorithms: **GSAT** (greedy SAT), **GWSAT** modification of GSAT with a simple random walk procedure, **WalkSAT**, **Conflict-Directed Random Walk** also known as Papadimitriou's algorithm, **adaptive Novelty+** one of the most effective SLS algorithms[2], **adaptive G2WSAT** combines adaptive noise and look-ahead in local search

Results: experiments showed that SLS methods are unable to find solutions even of the mid-sized PU problem instances provided by SIE.

1. http://www.satlib.org/ubcsat/algorithms/
2. http://ubcsat.dtompkins.com/home
3. Biere, A., Heule, M., van Maaren, H., and Walsh, T.: Handbook of Satisfiability, IOS Press, 2009

Selected evaluation results

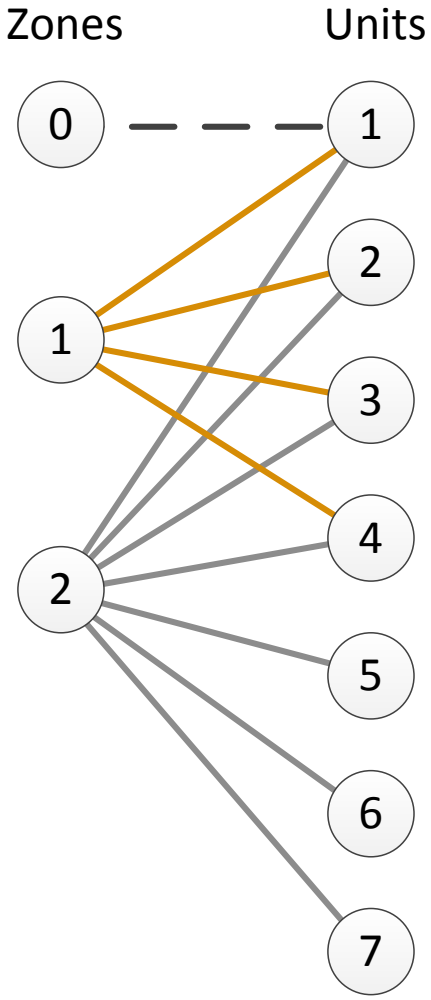
ALPEN-ADRIA UNIVERSITÄT
KLAGENFURT | WIEN GRAZ

Test case	Input	MiniSAT	GSAT	GWSAT
simple-3	3Z, 4S, 2U, 2PU	S	TO	S
doublev-60	30Z, 40S, 20U, 4PU	S	TO	TO
double-20	20Z, 28S, 14U, 3PU	S	TO	TO
double-40	40Z, 58S, 29U, 3PU	TO	TO	TO

Test case	WalkSAT	CRWalk	ANovelty+	AG2WSAT
simple-3	S	S	S	S
doublev-60	TO	TO	TO	TO
double-20	TO	TO	TO	TO
double-40	TO	TO	TO	TO

S – solution found **TO** – time out (3 minutes)

Modification 1



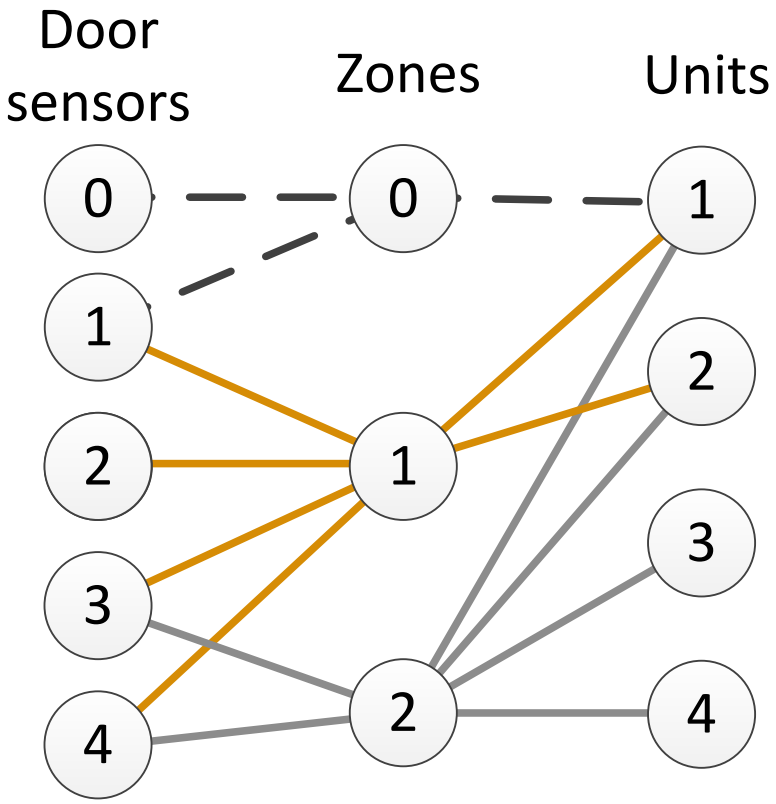
- In the worst case, $maxPU + 1$ units units can be used to connect zones to sensors via units.
- A zone can be connected either to one of the units controlling previously connected zones and their door sensors or to an additional unit.

$$UP(Z) = Z * (maxPU + 1) + 1$$

Program Mod.1

```
1 {u2z(U,Z) : unit(U) } 1 :- zone(Z).

1 {u2z(U,Z) : unit(U), U<=UP} 1 :- zone(Z),
                                   UP=Z*(maxPU+1)+1.
```

Modification 2



- More precise approximation of the number of units required to control all zones with smaller indexes and their door sensors

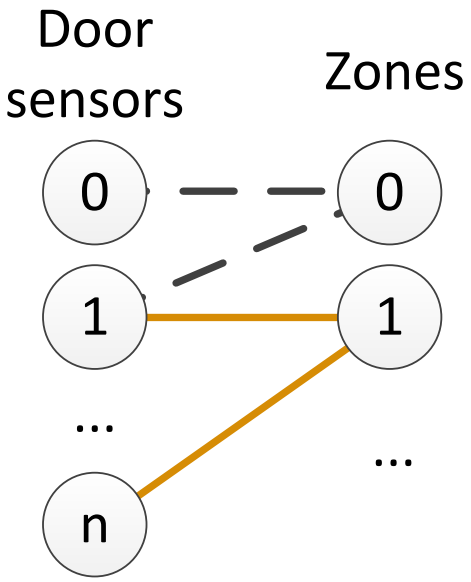
$$ZD_Z = \{z2s(Z_i, D) : Z_i < \mathrm{Z}\}$$

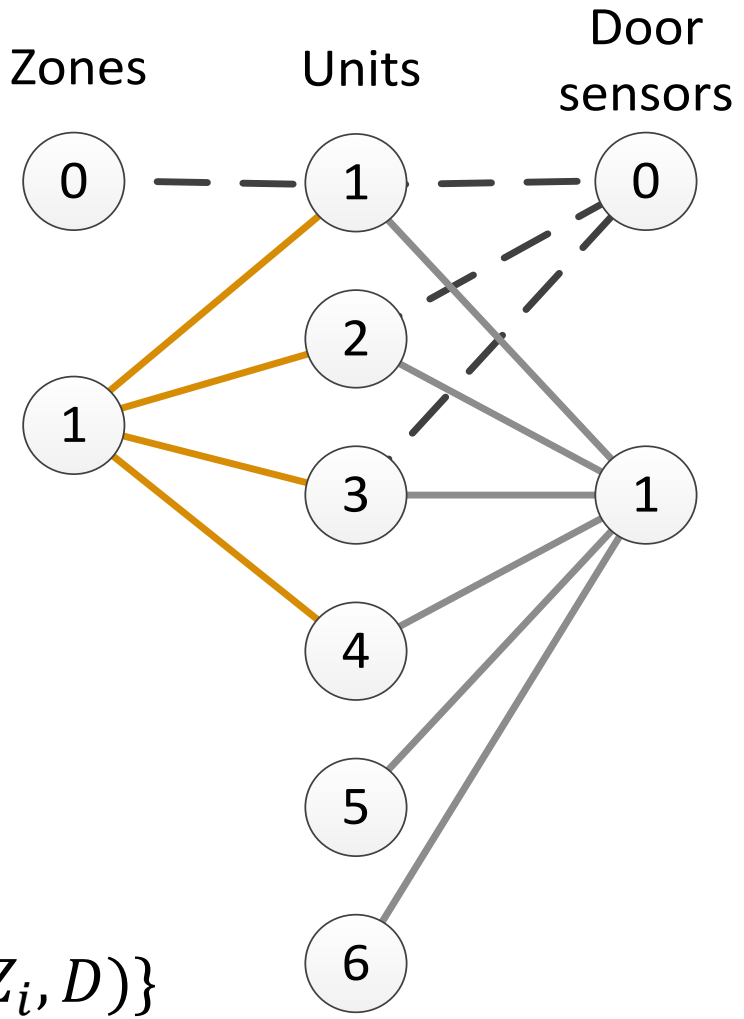
$$UP(Z) = |ZD_Z|/2 + 1$$

Program Mod.2

```
1 {u2z(U,Z) : unit(U) } 1 :- zone(Z).

1 {u2z(U,Z) : unit(U), U<=UP} 1 :- zone(Z),
      UP=Y/2+1, Y={z2s(Z1,D):Z1<Z}.
```

Modification 3





The number of units that can control a door sensor D is determined as:

$$UP(D) = \max(ZD_D)$$

$$ZD_D = \{(Z_i + 1)(maxPU + 1) : z2s(Z_i, D)\}$$

Program Mod.3

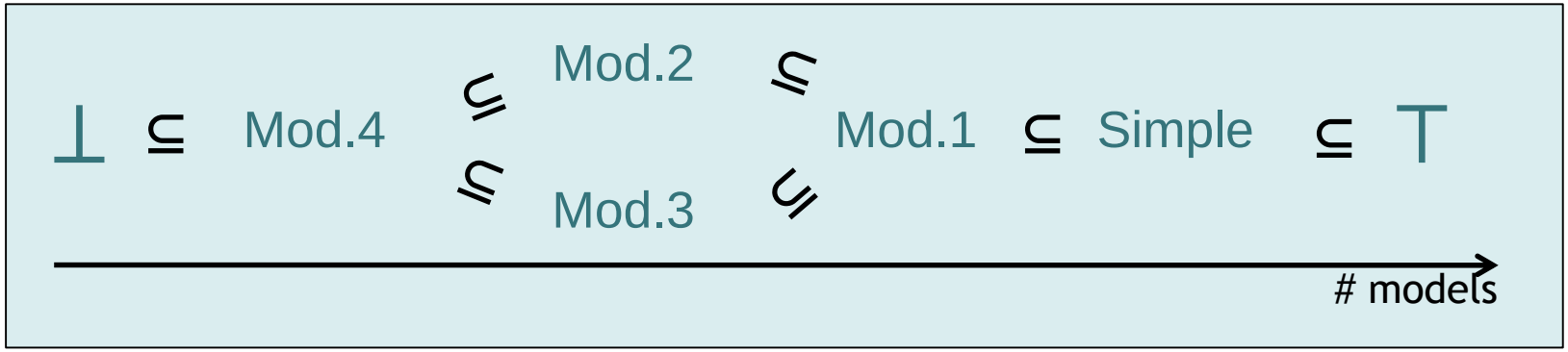
```
1 { u2z(U,Z) : unit(U) } 1 :- zone(Z).
1 { u2s(U,D) : unit(U) } 1 :- sensor(D).
```

```
1 {u2z(U,Z) : unit(U), U<=UP} 1 :- zone(Z),
                                   UP=Z*(maxPU+1)+1.
1 {u2s(U,D) : unit(U), U<=UP} 1 :- z2s(Z,D),
                                   UP=(Z+1)*(maxPU+1).
```

Modification 4

- Combines Modification 2 with limits on possible unit-sensor relations as in Modification 3

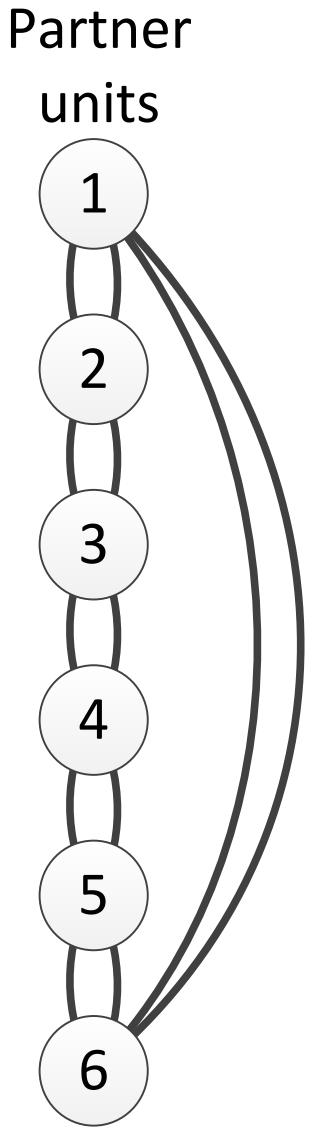
- Overview of modifications:



Modification 5

If a maximum number of partner unit equals 2 and a configuration exists, then there will be a configuration with the following partner units connection.

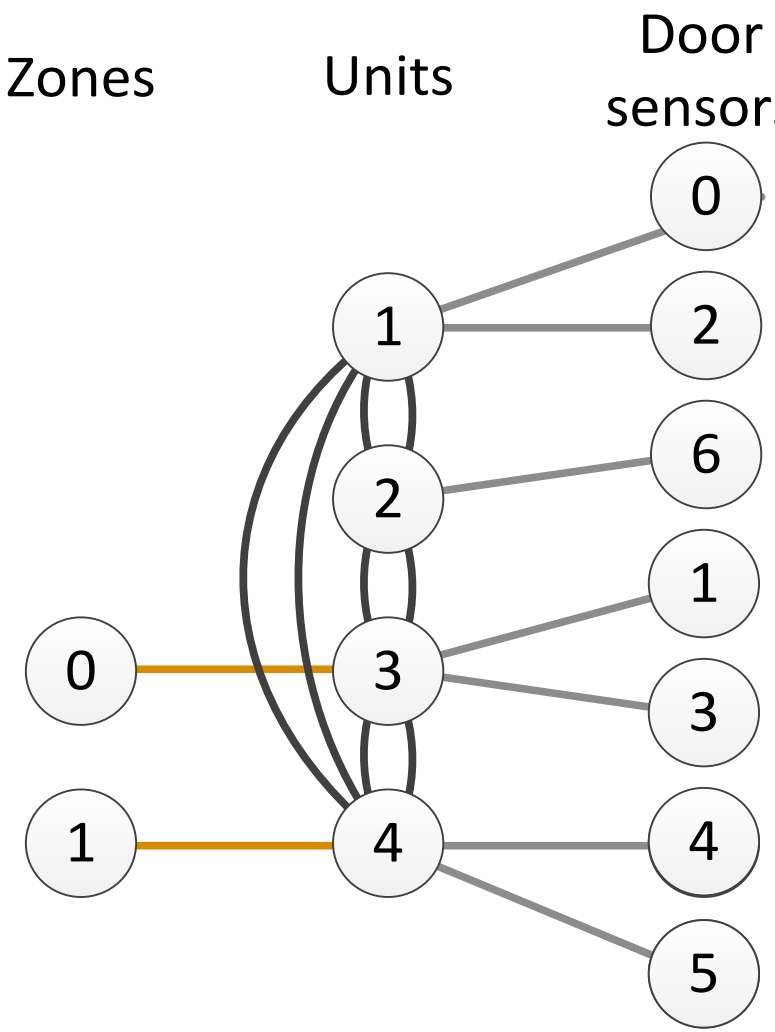
[Aschinger at al., 2011]



Program Mod.5

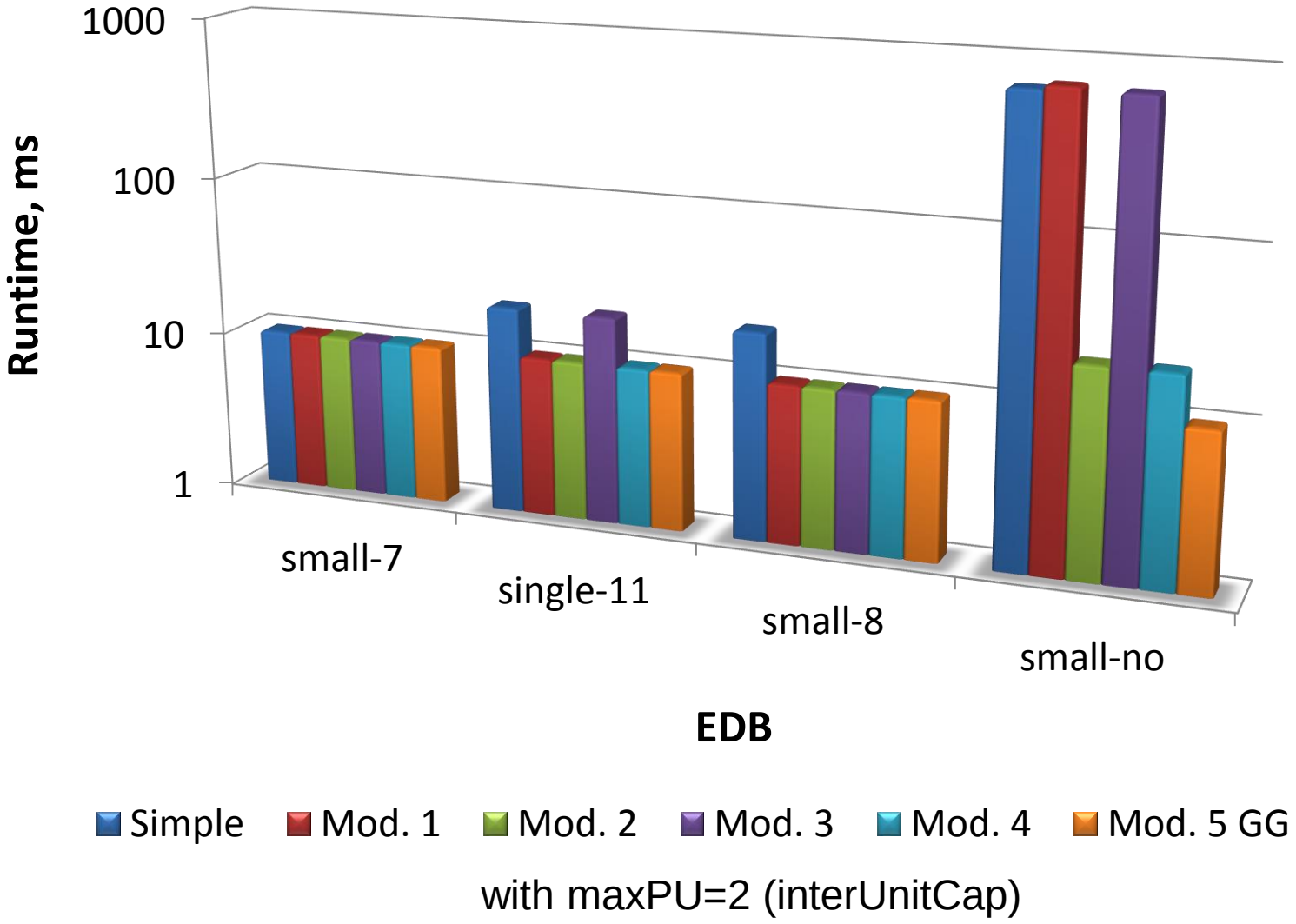
```
% generate a path
pu(U1,U2) :- unit(U1), unit(U2), U1=U2-1.

% create a cycle
pu(lower,1).
```

Example

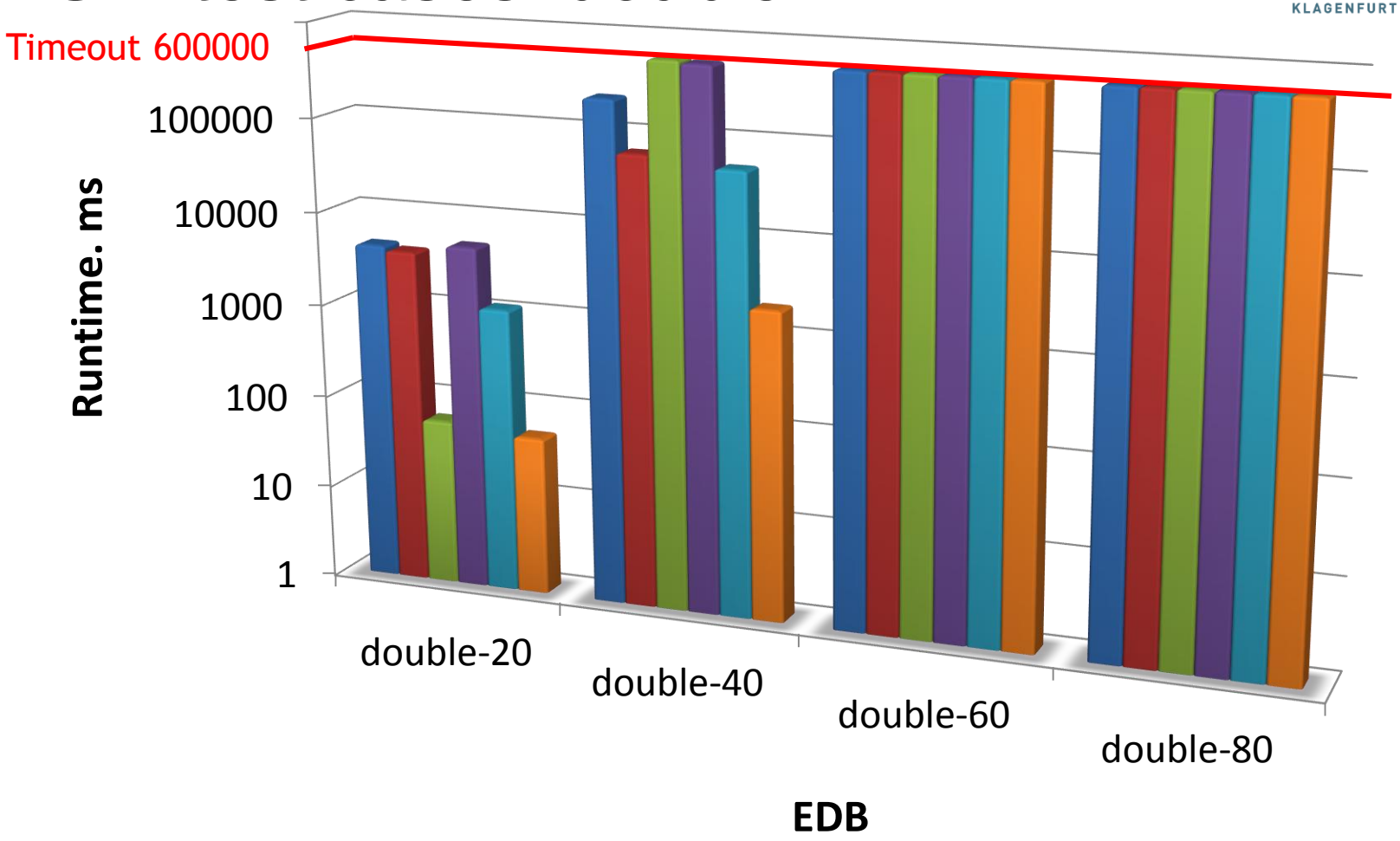



```
#const lower = 4.
#const maxPU = 2.


z2s(0,1). z2s(0,4).
z2s(0,6).

z2s(1,0). z2s(1,1).
z2s(1,2). z2s(1,3).
z2s(1,4). z2s(1,5).
```

SIE test cases: single & small

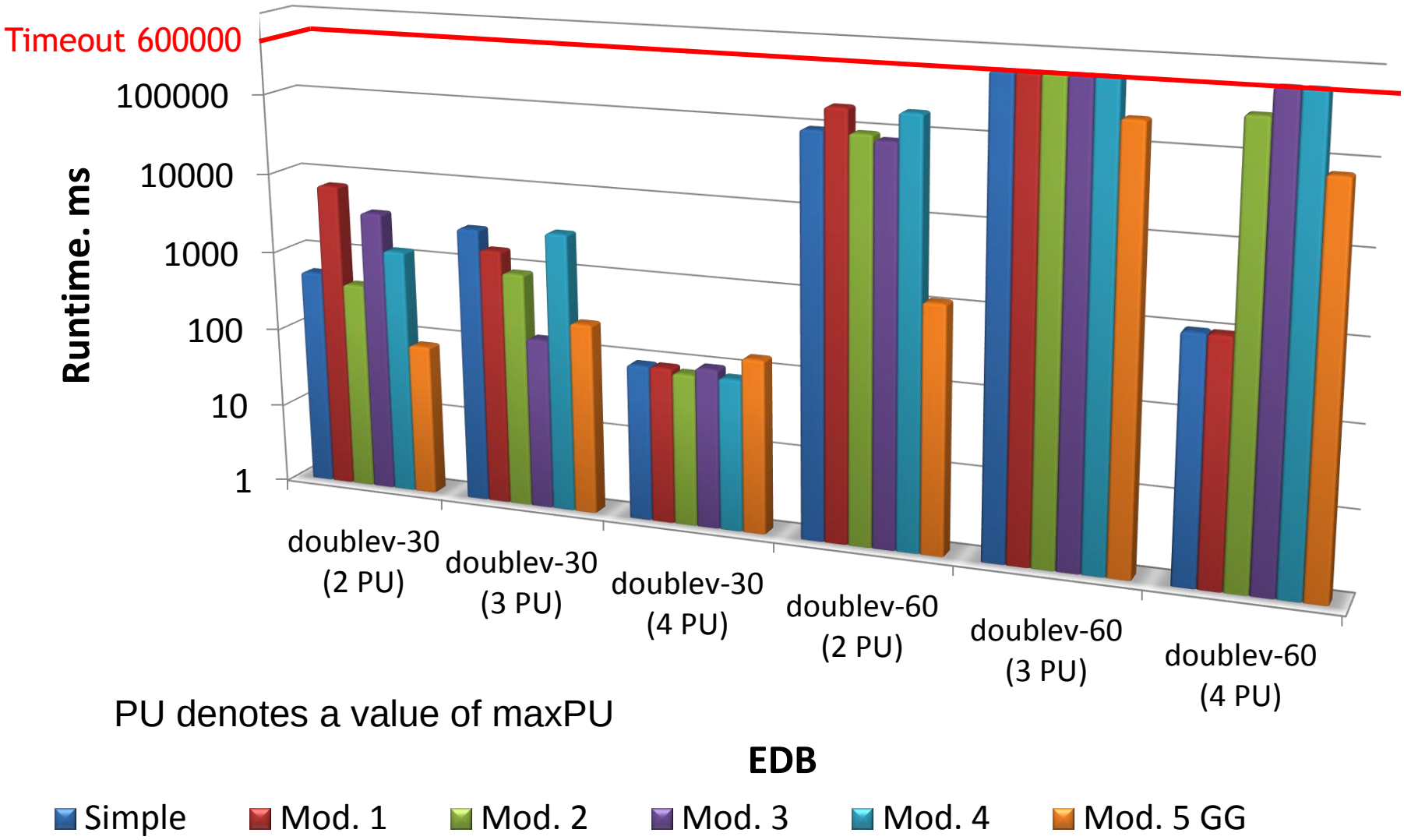


SIE test cases: double



with maxPU=2 (interUnitCap)

SIE test cases: double variant



Evaluation summary

- Simple ASP encoding shows performance comparable with a CP encoding
- Application of local search:
  - not advised if problem is solvable with complete algorithms
  - advised in case complete algorithms fail
- Heuristics help to improve the performance on specific instances