Knowledge-based (re)configuration: an ASP approach

Kostyantyn Shchekotykhin

Alpen-Adria-Universität Klagenfurt

Outline

- More heuristics
  - Class of “double” instances
  - QuickPUP
- Comparison of different KRRs
- Symmetry breaking
- Portfolio solvers

Evaluation of different KRR approaches

[Aschinger et al 2011]

Applied KRR methods

- Constraint programming: ECLiPSe-Prolog v6.0 (CSP)
- Propositional satisfiability testing: MiniSat v2.0 (SAT)
- Polynomial algorithm (DECPUP)
- Answer set programming: Clingo v3.0 (ASP)
- Integer programming:
  - Cbc v2.6.2 in combination with Clp v1.13.2 (CBC) and
  - Cplex v12.1 (CPLEX)

Integer programming I

- Modelled with matrixes of Boolean variables
- su_{ij} assign sensor i to unit j (same for zones)

$$
\begin{array}{cccc|c}
su_{1,1} & su_{2,1} & su_{3,1} & \dots & \sum \leq 2 \\
su_{1,2} & su_{2,2} & \dots & \dots & \sum \leq 2 \\
su_{1,3} & \dots & \dots & \dots & \sum \leq 2 \\
\dots & \dots & \dots & \dots & \dots \\
\hline
\sum = 1 & \sum = 1 & \dots & \dots &
\end{array}
$$

Integer programming II

- uu_{ij} unit i is a partner of the unit j

$$\begin{array}{cccc|l} 1 & uu_{1,2} & uu_{1,3} & \dots & \sum \leq maxPU + 1 \\ uu_{2,1} & 1 & \dots & \dots & \sum \leq maxPU + 1 \\ uu_{3,1} & \dots & 1 & \dots & \sum \leq maxPU + 1 \end{array}$$

- Boolean variables $unitUsed_i$

$su_{ij} \leq unitUsed_i$ and $zu_{ij} \leq unitUsed_i$

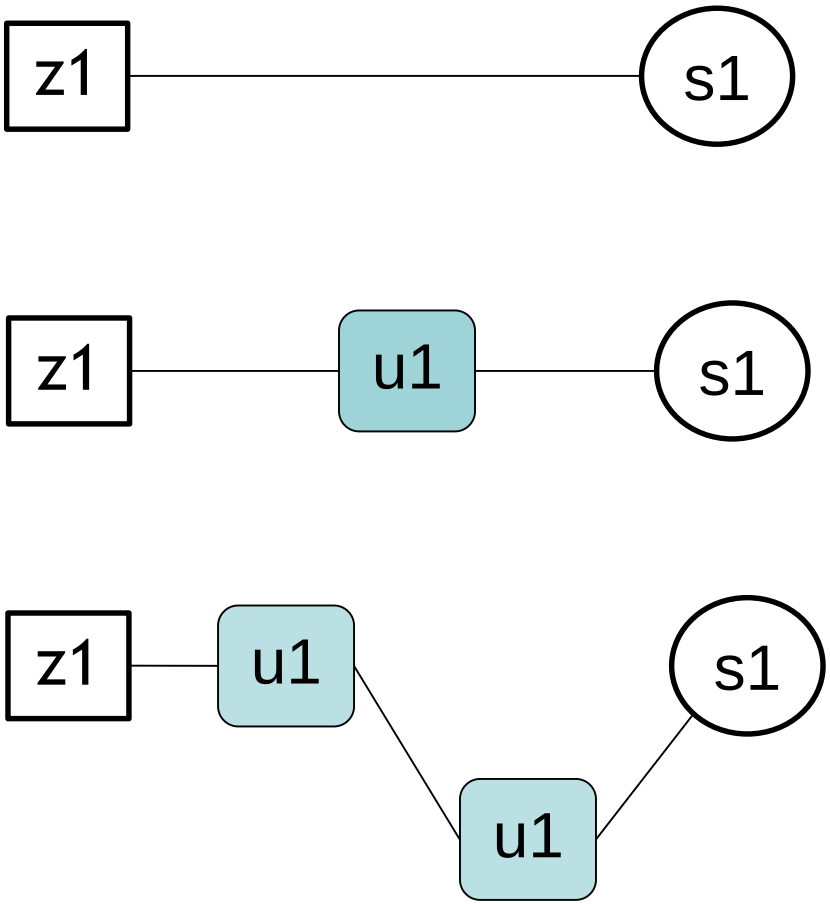
- Objective function

$$\sum_i unitUsed_i$$

DecPUP overview

- Inspired by a hypertree decomposition algorithm [Gottob et al., 2002]
- Runs in NLogSpace – polynomial time
- Instance check:
  - An instance has a solution if every zone or sensor has a degree less or equal to $2(maxPU + 1)$
- Exploits the fact that cyclic unit graphs (Mod.5) are more general solution topologies than paths
- Implements memorization of no-goods and two-step forward checking

DecPUP idea



DecPUP algorithm

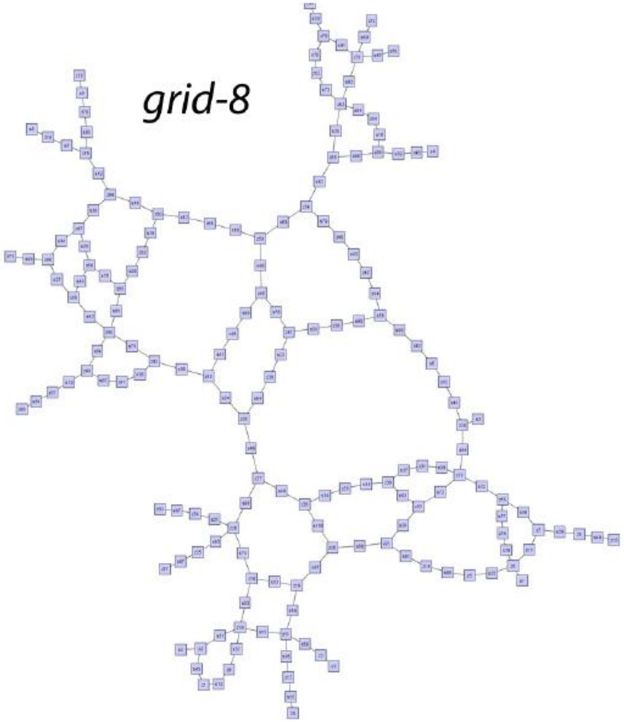
- Given the zone-sensor graph $G = (V_1, V_2, E)$
  1. Guesses two subsets of vertices $U_1, U_2 \subseteq V_1 \cup V_2$ such that $|U_i \cap V_1| \leq 2 \geq |U_i \cap V_2| \quad i = 1..2$
  2. Remove assigned vertices $C_R \leftarrow (V_1 \cup V_2) \setminus (U_1 \cup U_2)$

  Recursive function $C_R, \langle U_1, U_2 \rangle, \langle U_{i-1}, U_i \rangle$

  3. If $C_R = \emptyset$ and requirements hold, then terminate
  4. Guess an additional unit U_{i+1} (as in steps 1 and 2)
  5. Check if all neighbors of $v \in U_{i+1}$ appear in
$$U_{i-1} \cup U_i \cup U_{i+1}$$
  6. Make a recursive call

Evaluation instances

- **double, double variant:**
  - two rows of connected rooms, each room being a zone
  - variant – 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
- **grid:**
  - derived from real interlocking systems



[Teppan et al., 2012]

Evaluation for maxPU = 2 I

dbl-* – double, dblv-* – double variant, tri-* – triple
Cost … number of units
Runtime in sec., timeout 600 sec.

Name	$\|S\|$	$\|Z\|$	Edges	Cost	CSP	SAT	DECPUP	ASP	CBC	CPLEX
dbl-20	28	20	56	14	0.02	0.48	0.01	0.16	14.12	1.53
dbl-40	58	40	116	29	0.28	2.36	0.05	3.93	224.14	13.58
dbl-60	88	60	176	44	0.42	29.74	0.08	*	*	213.58
dbl-80	118	80	236	59	1.14	*	0.16	*	*	522.50
dbl-100	148	100	296	74	1.89	*	0.41	*	*	*
dbl-120	178	120	356	89	3.21	*	0.39	*	*	*
dbl-140	208	140	416	104	5.01	*	0.59	*	*	*
dbl-160	238	160	476	119	13.94	*	0.71	*	*	*
dbl-180	268	180	536	134	20.07	*	0.87	*	*	*
dbl-200	298	200	596	149	14.4	*	1.08	*	*	*

Evaluation for maxPU = 2 II

Name	$\|S\|$	$\|Z\|$	Edges	Cost	CSP	SAT	DECPUP	ASP	CBC	CPLEX
dblv-30	28	30	92	15	0.09	0.42	65.49	0.26	37.18	2.93
dblv-60	58	60	192	30	0.26	3.15	*	1.94	*	*
dblv-90	88	90	292	45	0.82	12.54	*	27.35	*	*
dblv-120	118	120	392	60	1.85	41.65	*	13.92	*	*
dblv-150	148	150	492	75	3.48	20.97	*	29.54	*	*
dblv-180	178	180	592	90	6.20	44.28	*	54.50	*	*
tri-30	40	30	78	20	1.07	0.79	0.50	0.41	45.17	78.75
tri-32	40	32	85	20	0.64	0.74	*	0.26	55.20	4.66
tri-34	40	34	93	/	21.10	22.77	*	0.89	74.78	5.06
tri-60	79	60	156	40	158.49	315.42	114.08	4.40	*	108.01
tri-64	79	64	170	/	*	379.36	*	43.88	*	76.26

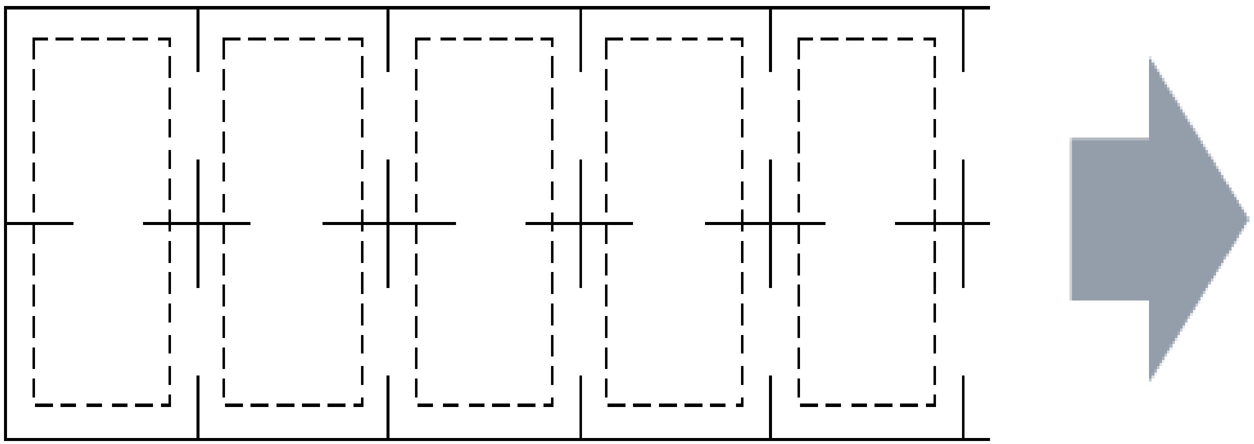
Evaluation for maxPU = 4

Name	$\|S\|$	$\|Z\|$	Edges	Cost	CSP	SAT	ASP	CBC	CPLEX
tri-30	40	30	78	20	0.12	2.40	0.40	182.91	24.79
tri-32	40	32	85	20	0.14	1.91	0.66	270.27	20.84
tri-34	40	34	93	20	*	1.98	0.60	331.29	*
tri-60	79	60	156	40	0.52	*	11.07	*	*
tri-64	79	64	170	40	*	*	7.61	*	*
tri-90	118	90	234	59	1.50	401.44	332.34	*	*
tri-120	157	120	312	79	3.37	*	*	*	*
grid-1	100	79	194	50	*	78.19	31.45	*	*
grid-2	100	77	194	50	*	90.89	18.91	*	*
grid-3	100	78	194	50	*	88.87	25.72	*	*
grid-4	100	80	194	50	*	95.12	24.66	*	*
grid-5	100	76	194	50	*	454.42	48.88	*	*
grid-6	100	78	194	50	*	204.85	9.15	*	*
grid-7	100	79	194	50	*	112.36	12.89	*	*
grid-8	100	78	194	50	*	*	11.89	*	*
grid-9	100	76	194	50	*	91.62	19.71	*	*
grid-10	100	80	194	50	*	545.16	13.54	*	*

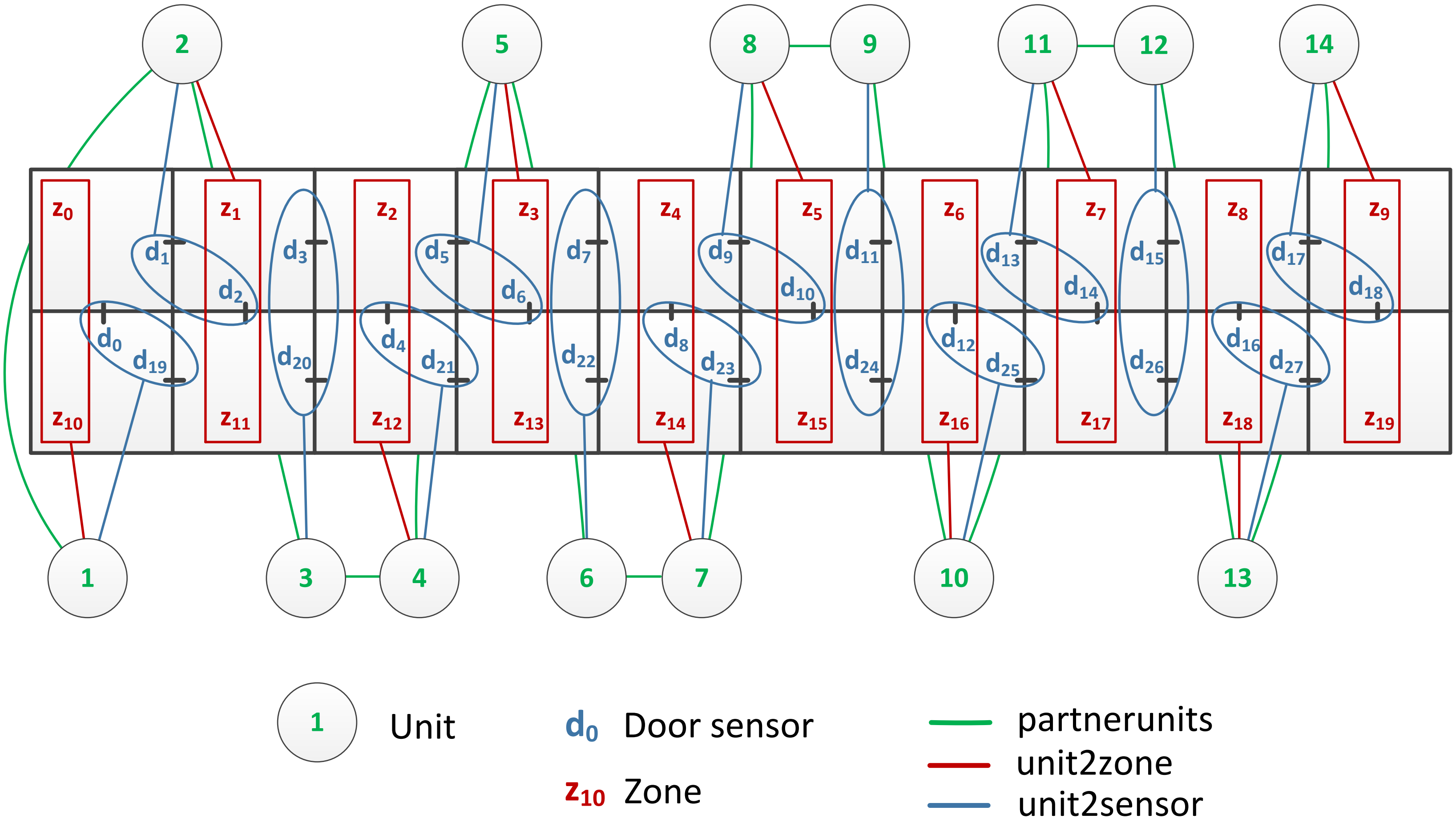
More heuristics

“Double” instances

- 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



Solutions for the double cases



double-80 is solved in 5 seconds (instead TO after 100 min) just by adding definite Horn clauses to the simple program.

Encoding extension I

```
% assign first zone to the first unit
firstZone(Y):- Y = #min[zone(X)=X].
firstUnit(Y):- Y = #min[unit(X)=X].
u2z(U,Z):- firstZone(Z), firstUnit(U).

adj(Z1,D,Z2) :- z2s(Z1,D), z2s(Z2,D), Z1<Z2.

% defines a column/numeration of zones
col(Z1,D,Z2):- zone(Z1), zone(Z2),
               adj(Z1,D,Z2), #abs(Z2-Z1) > 1 .
```

Encoding extension II

```
% assign column on one unit
u2z(U,Z1):- col(Z,_,Z1), u2z(U,Z).

% next column on the next unit
u2z(P,Z2):- u2z(U,Z), zone(Z2), Z2=Z+1,
                      P #mod 3 > 0, pu(U,P).

% every third unit should be free
u2z(X,Z2):- u2z(U,Z), zone(Z2), Z2=Z+1,
 P #mod 3 == 0, partners(U,P), partners(P,X).
```

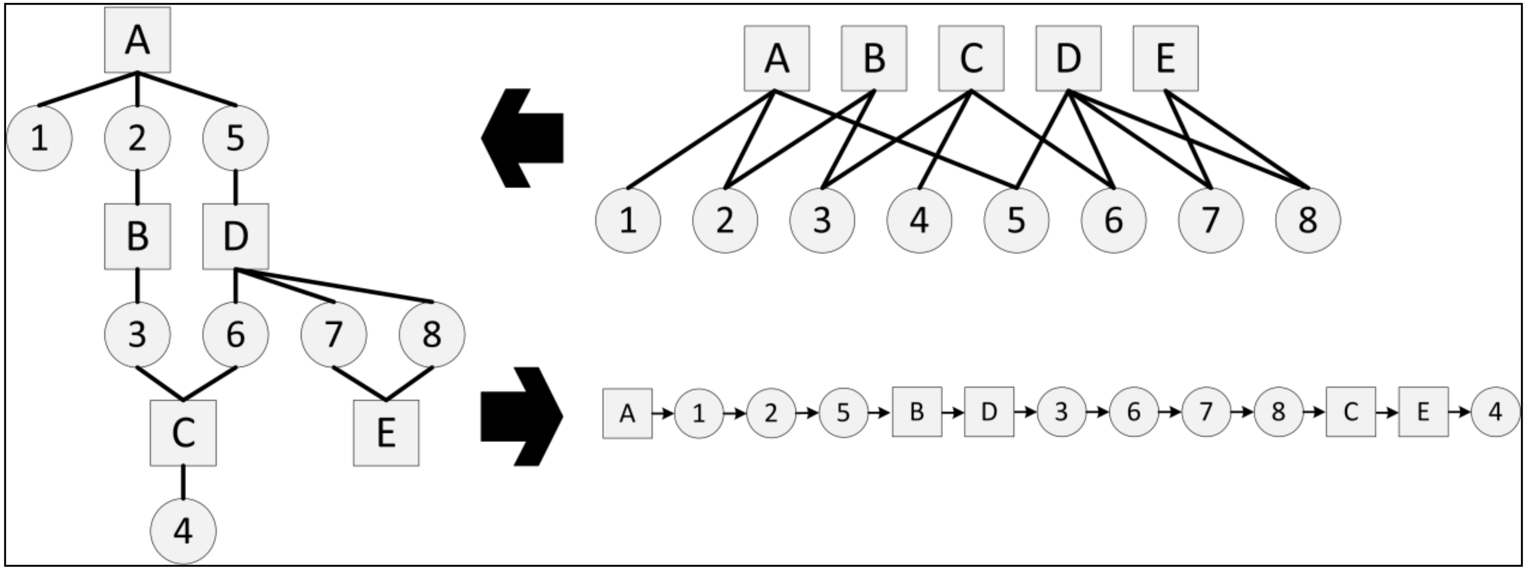
General heuristic

- For `maxPU = 2` there exists a poly-time algorithm [Aschinger et al., 2011]
- For `maxPU >= 3` complexity remains unclear
- Problems of Siemens have `maxPU = 4`
- SAT, MIP, CP or ASP are unable to find solutions for mid- and large-size instances [Aschinger et al, 2011]
- Double works only for one class of instances
- QuickPUP is a general heuristic for solving [Teppan et al., 2012]

QuickPUP overview I

- Recursive backtracking search algorithm
- Three main steps:

1. Order the elements (zones/sensors) in breadth-first order starting from some zone



QuickPUP overview II

2. Assignment step
   - Assign element to a new unit
   - Else, assign the element to some used unit
     - Ordering: from units with less connections to ones more connections
     - Leads to many units not filled to capacity
   - Greedy merging procedure for densifying and merging the units

Variation: Create new units when old units are full

- Densifying is not needed
- Might find a model with minimal number of units

QuickPUP overview

3. Use timeout to restart the search with the goal to test a different starting point
   - Start from step 1 and select the next zone to build the ordering
   - Restart Do 1. and 2. for every zone or until a solution is found

- The algorithm continues until a solution is found

Experimental Setup

- Restart in 1 second
- QuickPup (new unit first)
- QuickPup* (old units first)
- QP and QP* were implemented in Java 1.5.
- Timeout = 600 secs

Results

IUCAP=2, UCAP=2

INSTANCE	UNITS	DP	SAT	CP	ASP	IP	QP*	QP	+UNITS
dbl-20	14	0.01	0.48	0.02	0.16	1.53	0.00	0.02	1
dbl-40	29	0.05	2.36	0.28	3.93	13.58	0.00	0.03	1
dbl-60	44	0.08	29.74	0.42	/	213.58	0.00	0.03	1
dbl-80	59	0.16	/	1.14	/	522.5	0.01	0.04	1
dbl-100	74	0.41	/	1.89	/	/	0.03	0.08	1
dbl-120	89	0.39	/	3.21	/	/	0.02	0.08	1
dbl-140	104	0.59	/	5.01	/	/	0.02	0.09	1
dbl-160	119	0.71	/	13.94	/	/	0.03	0.10	1
dbl-180	134	0.87	/	20.07	/	/	0.04	0.13	1
dbl-200	149	1.08	/	14.40	/	/	0.04	0.15	1
dblv-30	15	65.49	0.42	0.09	0.26	2.93	0.00	0.00	0
dblv-60	30	/	3.15	0.26	1.94	/	0.01	0.00	0
dblv-90	45	/	12.54	0.82	27.35	/	0.01	0.01	0
dblv-120	60	/	41.65	1.85	13.92	/	0.02	0.01	0
dblv-150	75	/	20.97	3.48	29.54	/	0.02	0.02	0
dblv-180	90	/	44.28	6.20	54.50	/	0.03	0.03	0

IUCAP=4, UCAP=2

INSTANCE	UNITS	SAT	CP	ASP	IP	QP*	QP	+UNITS
tri-30	20	2.40	0.12	0.40	24.79	0.00	0.00	0
tri-32	20	1.91	0.14	0.66	20.84	0.00	0.00	2
tri-34	20	1.98	/	0.60	/	0.00	0.00	5
tri-60	40	/	0.52	11.07	/	0.00	0.01	0
tri-64	40	/	/	7.61	/	0.01	0.01	6
tri-90	60	401.44	1.50	332.34	/	2.33	0.01	0
tri-120	79	/	3.37	/	/	8.23	0.02	0
grid1	50	78.19	/	31.45	/	0.18	0.02	0
grid2	50	90.89	/	18.91	/	0.69	0.01	0
grid3	50	88.87	/	25.72	/	0.10	0.01	1
grid4	50	95.12	/	24.66	/	0.00	0.01	0
grid5	50	454.42	/	48.88	/	0.01	0.01	2
grid6	50	204.85	/	9.15	/	0.01	0.01	1
grid7	50	112.36	/	12.89	/	0.05	0.01	2
grid8	50	/	/	11.89	/	1.54	0.01	0
grid9	50	91.62	/	19.71	/	0.01	0.01	0
grid10	50	545.16	/	13.54	/	4.15	0.02	0

Results for real cases: QP << 1 sec – ASP up to 17 minutes

Portfolio solvers

Portfolio solvers I

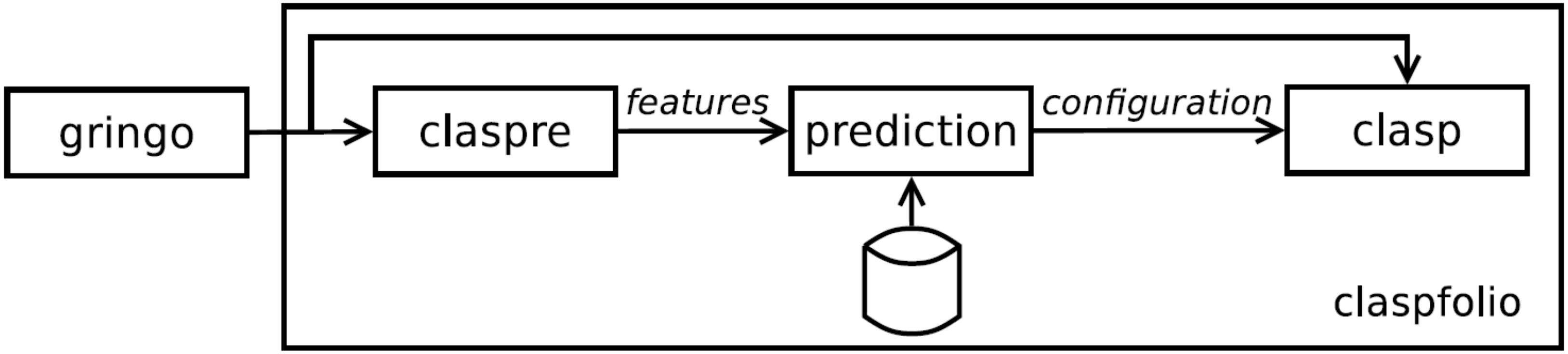
- Modern solvers
  - are highly configurable
  - implement different heuristics and tie breaking strategies
- Which solver/solver configuration works best for my problem?
- Portfolio solvers:
  - Claspfolio http://potassco.sourceforge.net/
  - ME-ASP https://www.mat.unical.it/ricca/me-asp/

Portfolio solvers: an overview I

- Given a representative set of problems and their instances
- Extract features characterizing the problem instances (>100 features)
- Solve each instance with different configurations/solvers
- Apply machine learning to find “empirical hardness” of the problem instance
  - Statistical model predicting runtime of different configurations/solvers

Portfolio solvers: an overview II

- Given a new program
  - extracts a set of features used to classify the program
  - find the best configuration/solver for the instance
- Claspfolio was used for solving double cases which turned out to be the hardest cases for all programs



Claspfolio, evaluation details

- Input: the double case double-20 and simple program
- Output: chosen solver configuration (set of options):

  --heu=VSIDS --del=3,1.1,1000

  --restarts=100,1.5,20000 --local-restarts

  - VSIDS – Variable State Independent Decaying Sum
  - del – fixes the size and growth factor of the dynamic nogood database
  - restarts – parameterizes a restart policy
  - local restarts - exploits local restarts

Time frame 10 minutes

Test case	Input	InterUnitCap=4		InterUnitCap=3		InterUnitCap=2	
		Default opt.	Claspfolio	Default opt.	Claspfolio	Default opt.	Claspfolio
double-20	20Z,28S,14U	00:00,07	00:00,07	00:00,08	00:00,10	00:04,35	00:00,68
double-40	40Z,58S, 29U	00:02,29	00:00,70	01:51,13	00:05,18	03:43,25	05:39,80
double-60	60Z,88S, 44U	01:55,11	00:05,54	timeout	timeout	timeout	timeout
double-80	80Z,118S,59U	timeout	06:41,89	timeout	timeout	timeout	timeout

Time frame 100 minutes

Test case	Input	InterUnitCap=4		InterUnitCap=3		maxPU=2	
		Default opt.	Claspfolio	Default opt.	Claspfolio	Default opt.	Claspfolio
double-20	20Z,28S,14U	0:00.07	0:00.07	0:00.08	0:00.10	0:04.33	0:00.68
double-40	40Z,58S, 29U	0:02.29	0:00.72	1:50.67	0:05.18	3:42.38	5:40.10
double-60	60Z,88S, 44U	1:55.33	0:05.60	37:34.89	22:21.08	timeout	timeout
double-80	80Z,118S,59U	15:41.67	6:44.38	timeout	timeout	timeout	timeout

Symmetry breaking in ASP

Symmetry breaking I

- Symmetry: one solution can be obtained from the other by renaming constants
- House problem (e.g. renaming of cabinets)

```
t2c(1,10), t2c(2,11) -> t2c(1,11), t2c(2,10)
```

- Same for PUP (renaming of units)
- Simple symmetry breaking
  - PUP: assign first sensor to the first unit and the second one to a unit in the first half of the cycle
  - House: assign things with smaller ids to cabinets with smaller ids

Symmetry breaking II

- Problems:
  - Finding symmetries is hard
  - Blocking them is even harder
    - Do all instances have the same symmetries?
- Is automatic detection and blocking of symmetries possible?

Example: House problem

```
cabinet(10..12).
thing(1..3).

{c2t(X,Y):cabinet(X)}1 :- thing(Y).
placed(T) :- c2t(X,T).
:- thing(X), not placed(X).
```

27 Models:

```
c2t(10,3) c2t(10,2) c2t(10,1)
…
c2t(12,3) c2t(12,2) c2t(12,1)
```

Symmetry breaking I

Three types of symmetry breaking for SAT:

- variable (A, B),
- value $(A, \neg A)$ and
- variable-value $(A, \neg B)$

where A and B are propositional symbols, and (A, B) is a permutation that replaces A in all clauses of a CNF with B and vice versa

Symmetry breaking II

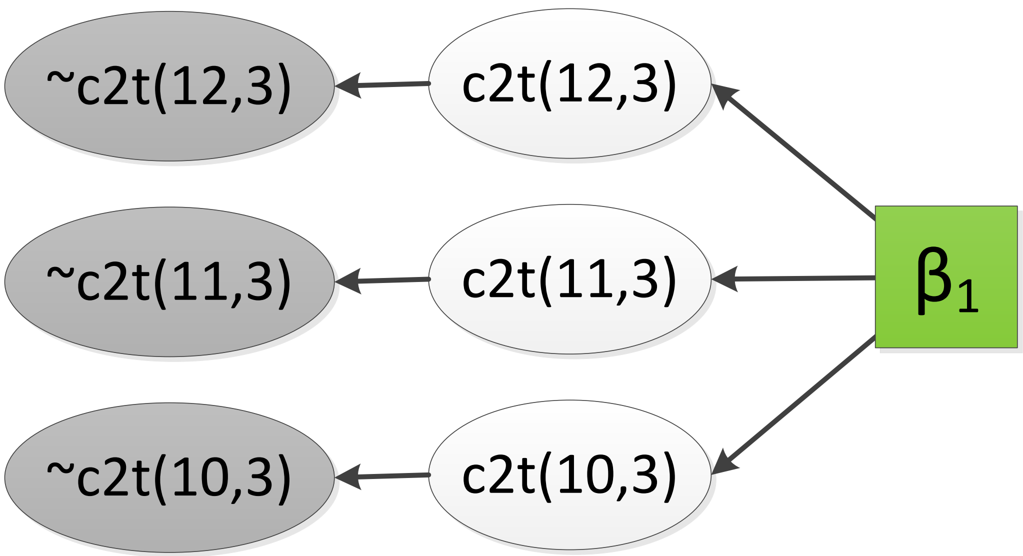
- Permutations in a CNF are generators of symmetric solutions
- Identification of permutations can be reduced to the colored graph automorphism problem
- Automorphism is, in some sense, a way of mapping the object to itself while preserving all of its structure (coloring), i.e. a symmetry of a mathematical object
- Algorithms like **saucy**, **nauty** or **bliss** can be used to find automorphims of a colored graph

Grounded program

```
cabinet(10). cabinet(11). cabinet(12).
thing(1). thing(2). thing(3).
#count{c2t(12,3),c2t(11,3),c2t(10,3)}1.
#count{c2t(12,2),c2t(11,2),c2t(10,2)}1.
#count{c2t(12,1),c2t(11,1),c2t(10,1)}1.
placed(1):-c2t(10,1).
placed(2):-c2t(11,2).
...
:-not placed(2).
:-not placed(1).
```

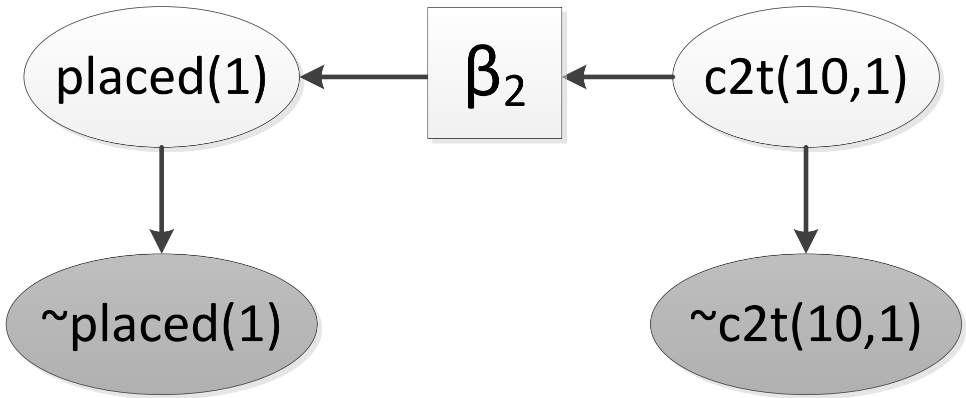
Graph coloring I

```
#count{c2t(12,3),c2t(11,3),c2t(10,3)}1.
```

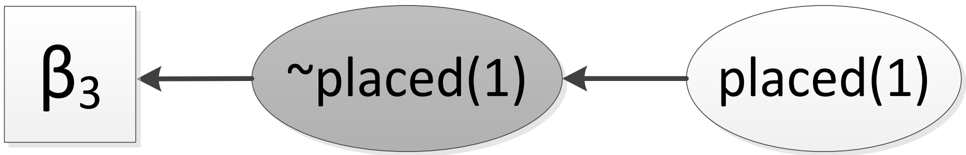


Graph coloring II

- `placed(1):-c2t(10,1).`



- `:-not placed(1).`



Symmetry breaking rules

- SBASS was queried to find 6 permutation sets corresponding to automorphisms of the colored graph [Drescher et al., 2011]

```
(19 20)
1 1 2 1 20 19
:- not c2t(10,1), c2t(11,1).

(18 19)
1 1 2 1 19 18
:- not c2t(11,1), c2t(12,1).
```

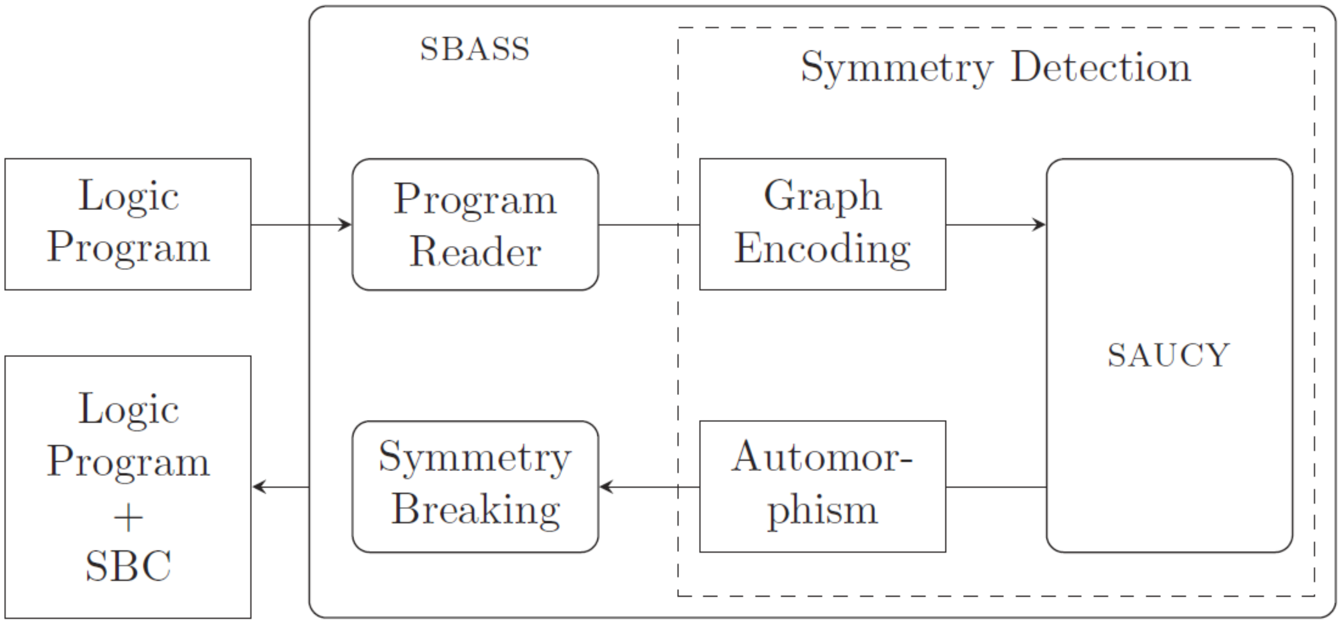
- Other four constraints are defined for the things 2 and 3

Breaking the symmetries

- Grounded program is extended with SBASS SB constraints
- Clasp returns the only model:

```
cabinet(10) cabinet(11) cabinet(12)
thing(1) thing(2) thing(3)
c2t(10,3) c2t(10,2) c2t(10,1)
placed(1) placed(2) placed(3)
```

SBASS architecture



Evaluation

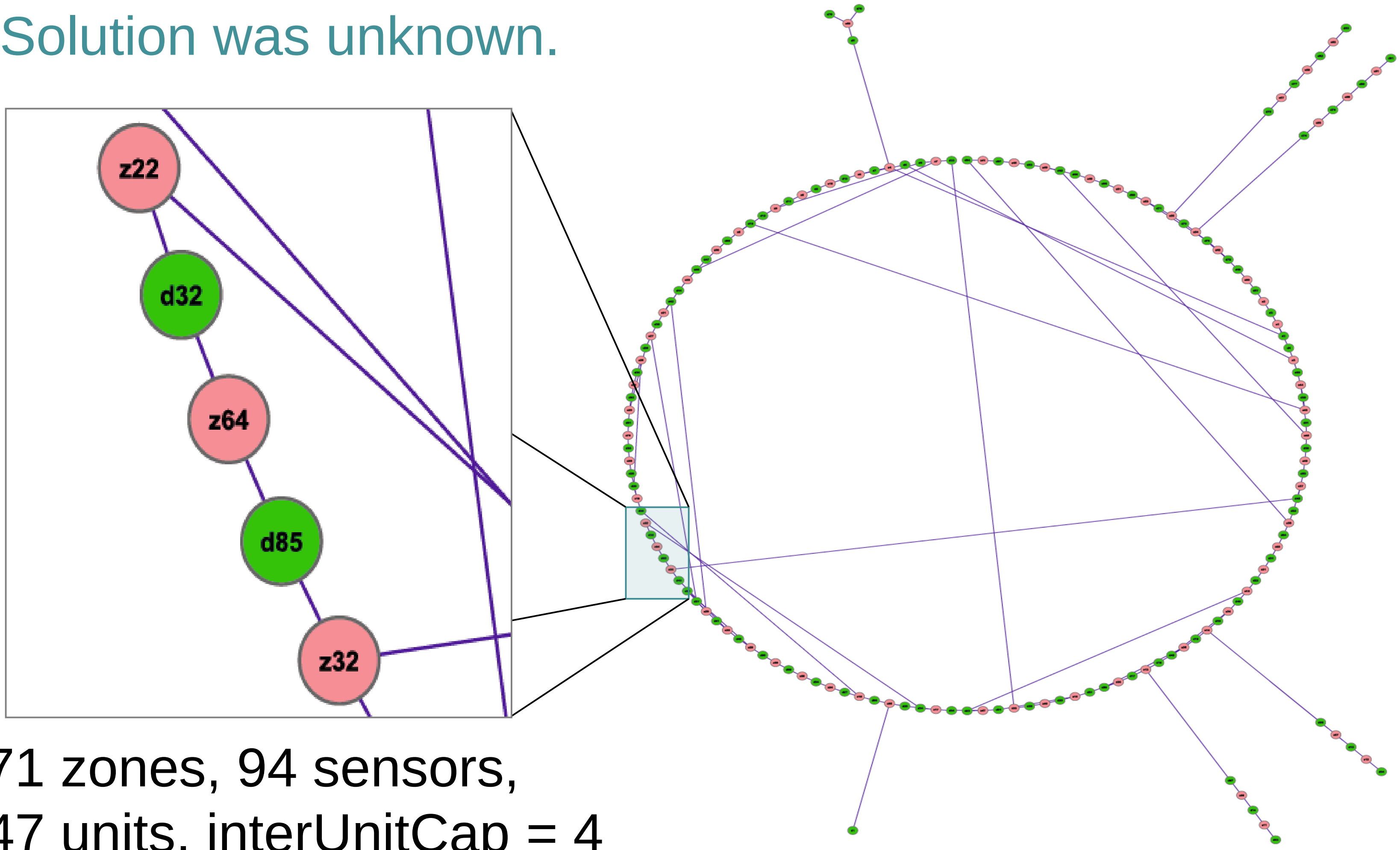
- Limit – number of computed generators
- TO – timeout 600 seconds

Instance	Optimum	No SBASS	SBASS, default	SBASS, limit=5	SBASS, limit=20
empty_p05t025	50	50/0:00.047	50/0:00.079	50/0:00.053	50/0:00.079
empty_p10t050	100	100/0:00.284	100/0:01.049	100/0:00.321	100/0:00.465
empty_p15t075	150	150/0:00.977	150/1:17.148	150/0:01.149	150/0:01.614
empty_p20t100	200	200/0:04.369	200/-	200/0:05.718	200/0:39.575
empty_p25t125	250	250/1:04.125	TO	250/0:58.110	250/-
empty_p30t150	300	300/-	TO	300/-	300/-
empty_p35t175	350	350/-	TO	350/-	350/-
empty_p40t200	400	400/-	TO	400/-	TO
long_2_p02t030c3	0	0/0:00.082	0/0:0.121	0/0:00.083	0/0:00.113
long_2_p04t060c3	0	0/0:00.721	0/0:01.556	0/0:00.786	0/0:01.639
long_2_p06t090c3	0	0/2:07.973	0/0:36.373	0/1:03.695	0/0:12.070
long_2_p08t120c3	0	35/-	35/-	40/-	30/-
long_2_p10t150c3	0	45/-	55/-	55/-	70/-
long_2_p12t180c3	0	90/-	75/-	80/-	80/-
long_2_p14t210c3	0	TO	150/-	TO	170/-
long_2_p16t240c3	0	TO	TO	TO	TO

Some good news :-)

Problem instance 1

Solution was unknown.



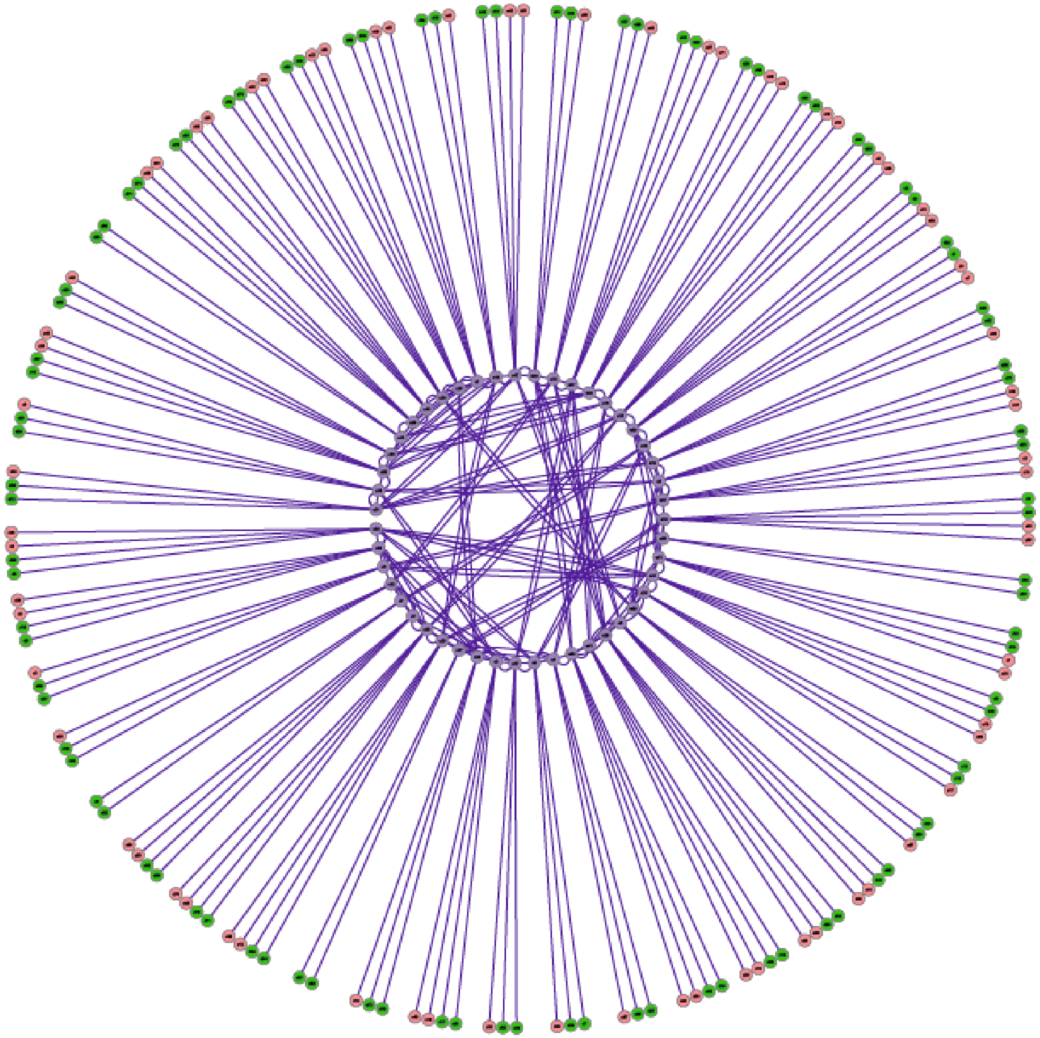
71 zones, 94 sensors,
47 units, interUnitCap = 4

Solution of instance 1

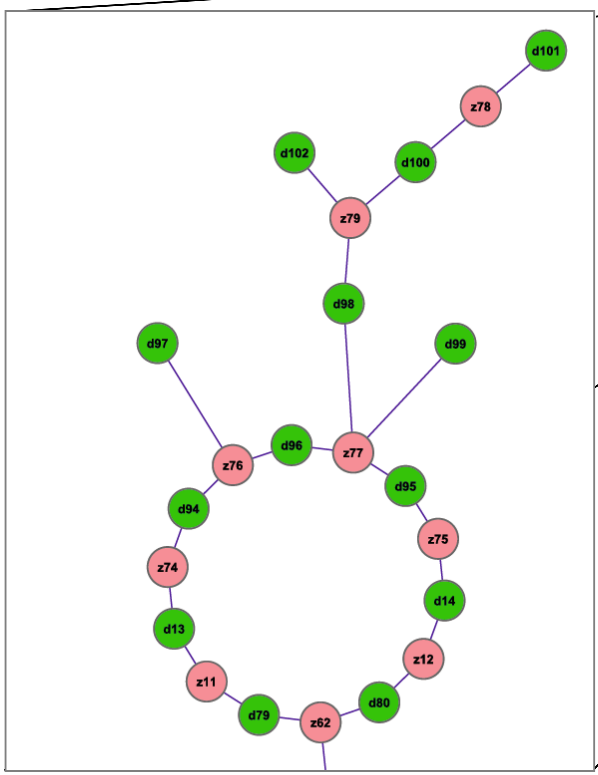
```
unit2zone(35,64).
...
unit2sensor(1,32).
...
partnerunits(1,35)
partnerunits(35,1)
...
```

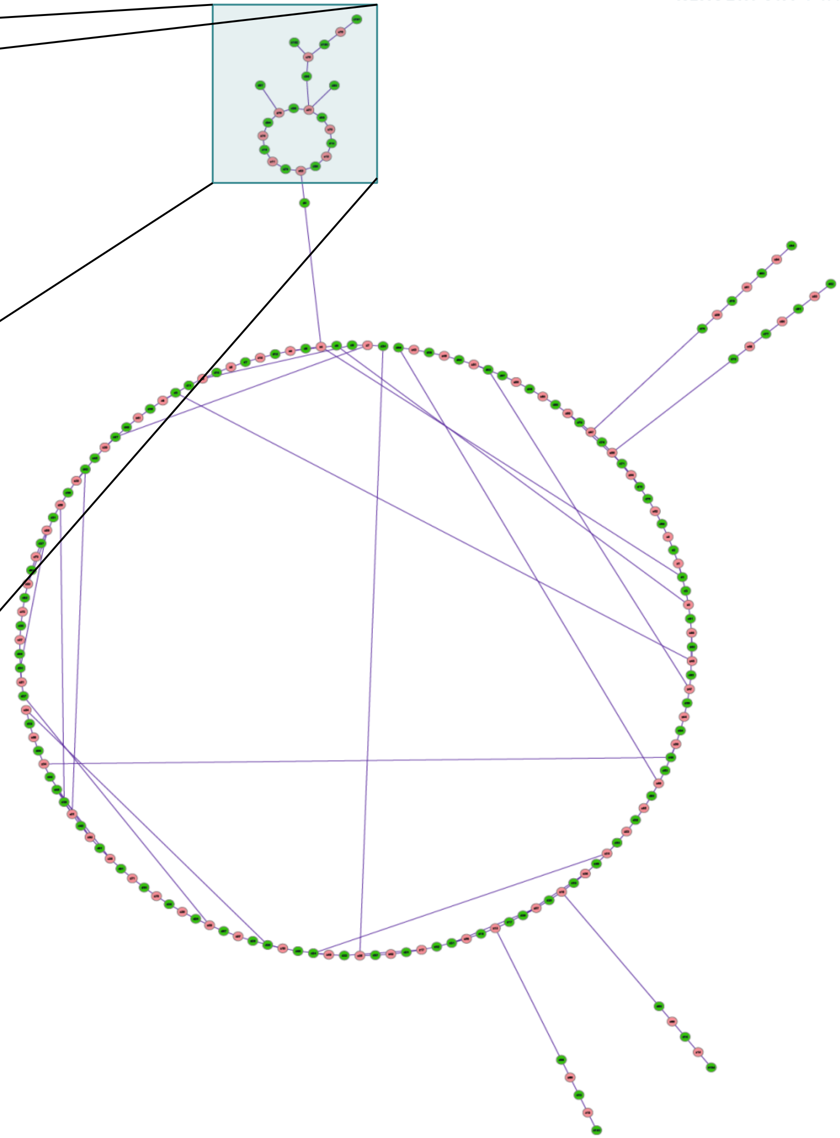
Time:

13 seconds (simple + parameter learning)



Problem instance 2





Solution was unknown.

79 zones, 104 sensors,
52 units, interUnitCap = 4

Solution of instance 2

```
unit2zone(13,79).
...
unit2sensor(47,102).
...
partnerunits(13,47).
partnerunits(47,13).
...
```

Time: 25 seconds

(simple + parameter learning)