Optimization Methods for the Partner Units Problem*

Markus Aschinger[1], Conrad Drescher[1], Gerhard Friedrich[2], Georg Gottlob[1], Peter Jeavons[1], Anna Ryabokon[2], Evgenij Thorstensen[1]

1 Computing Laboratory, University of Oxford
2 Institut für Angewandte Informatik, Alpen-Adria-Universität Klagenfurt

Abstract. In this work we present the Partner Units Problem as a novel challenge for optimization methods. It captures a certain type of configuration problem that frequently occurs in industry. Unfortunately, it can be shown that in the most general case an optimization version of the problem is intractable. We present and evaluate encodings of the problem in the frameworks of answer set programming, propositional satisfiability testing, constraint solving, and integer programming. We also show how to adapt these encodings to a class of problem instances that we have recently shown to be tractable.

1 Introduction

The Partner Units Problem (PUP) has recently been proposed as a new challenge in automated configuration [8]. It captures the essence of a specific type of configuration problem that frequently occurs in industry. Informally the PUP can be described as follows: Consider a set of sensors that are grouped into zones. A zone may contain many sensors, and a sensor may be attached to more than one zone. The PUP then consists of connecting the sensors and zones to control units, where each control unit can be connected to the same fixed maximum number *UnitCap* of zones and sensors.[1] Moreover, if a sensor is attached to a zone, but the sensor and the zone are assigned to different control units, then the two control units in question have to be directly connected. However, a control unit cannot be connected to more than *InterUnitCap* other control units (the partner units).

For an application scenario consider, for example, a museum where we want to keep track of the number of visitors that populate certain parts (zones) of the building. The doors leading from one zone to another are equipped with sensors. To keep track of the visitors the zones and sensors are attached to control units; the adjacency constraints on the control units ensure that communication between units can be kept simple.

* Work funded by FFG FIT-IT Grant 825071 (Klagenfurt) and EPSRC Grant EP/G055114/1 (Oxford).

[1] For ease of presentation and without loss of generality we assume that *UnitCap* is the same for zones and sensors.

Let us emphasize that the PUP is not limited to this application domain: it occurs whenever sensors that are grouped into zones have to be attached to control units, and communication between units should be kept simple. Typical applications include intelligent traffic management, or surveillance and security applications. The PUP has been introduced as a novel benchmark instance at this year's answer set programming competition [2].

Figure 1 shows a PUP instance and a solution for the case $\mathit{UnitCap} = \mathit{InterUnitCap} = 2$. In this example six sensors (left) and six zones (right), which are completely inter-connected, are partitioned into units (shown as squares) respecting the adjacency constraints. Note that for the given parameters this is a maximal solvable instance; it is not possible to connect a new zone or sensor to any of the existing ones.

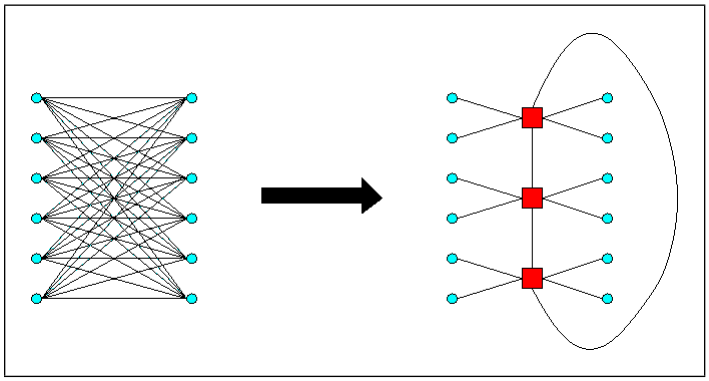

Fig. 1. Solving a $K_{6,6}$ Partner Units Instance — Partitioning Sensors and Zones into Units on a Circular Unit Layout

Very recently, we have shown that the case where $\mathit{InterUnitCap} = 2$ and $\mathit{UnitCap} = k$ for some fixed k is tractable, by giving a specialized NLOGSPACE algorithm that is based on the notion of a path decomposition [1]. While this case is of some importance for our partners in industry, the general case is also interesting: Consider, for example, a grid of rooms, where every room is accessible from each neighboring room, and all the doors are fitted with a sensor. Moreover, assume there are doors to the outside on two sides of the building; the corresponding instance is shown in Figure 2, with rooms as black squares, and doors as circles. It is not hard to see that this instance is unsolvable for $\mathit{InterUnitCap} = 2$ and $\mathit{UnitCap} = 2$. However, it is easily solved for $\mathit{InterUnitCap} = 4$ and $\mathit{UnitCap} = 2$: Every room goes on a distinct unit, together with the sensors to the west and to the north; the connections between units correspond to those between rooms. Clearly this solution is optimal, in the sense of using the least possible number of units.

In this paper we present and evaluate encodings in the optimization frameworks of answer set programming, constraint satisfaction, SAT-solving, and integer programming, that can be used to solve the general version of the PUP. We also show how to adapt these encodings to the special case $\mathit{InterUnitCap} = 2$, and compare the adapted encodings against our specialized algorithm.
It is worth emphasizing that we do not take our encodings/algorithms to be the final answer on the PUP. Instead we hope that our work will spark interest in the problem across the different optimization research communities, eventually resulting in better encodings and better theoretical understanding of the problem.

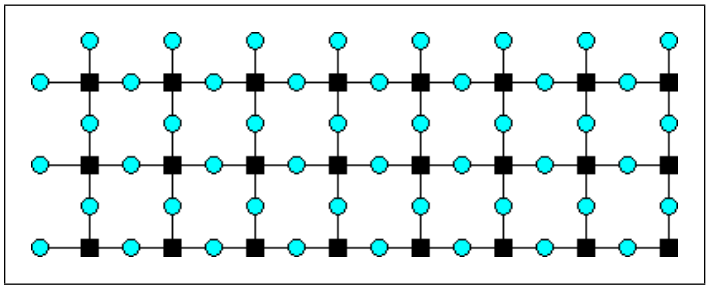

Fig. 2. A Grid-like PUP Instance

The remainder of this paper is organized as follows: In section 2 we give the basic formal definitions, and identify general properties of the PUP. Then, in section 3 we present problem models in the frameworks of answer set programming, propositional satisfiability testing, integer programming, and constraint solving; these problem models can be used for arbitrary fixed values of *InterUnitCap*. In section 4 we briefly recall our recent tractability results for the case $\mathit{InterUnitCap} = 2$, and show how the various PUP encodings presented in this paper can be adapted to this special case. Finally, in section 5 we evaluate the performance of our encodings, and in section 6 we list some directions for future research.

2 The Partner Units Problem

2.1 Formal definition

Formally, the PUP consists of partitioning the vertices of a bipartite graph $G = (V_1, V_2, E)$ into a set U of bags such that each bag

- contains at most *UnitCap* vertices from V_1 and at most *UnitCap* vertices from V_2; and
- has at most *InterUnitCap* adjacent bags, where the bags U_1 and U_2 are adjacent whenever $v_i \in U_1$ and $v_j \in U_2$ and $(v_i, v_j) \in E$.

To every solution of the PUP we can associate a solution graph. For this we associate to every bag $U_i \in \mathcal{U}$ a vertex v_{U_i}. Then the solution graph G^* has the vertex set $V_1 \cup V_2 \cup \{v_{U_i} \mid U_i \in \mathcal{U}\}$ and the set of edges $\{(v, v_{U_i}) \mid v \in U_i \wedge U_i \in \mathcal{U}\} \cup \{(v_{U_i}, v_{U_j}) \mid U_i \text{ and } U_j \text{ are adjacent.}\}$. In the following we will refer to the subgraph of the solution graph induced by the v_{U_i} as the *unit graph*.
The following are the two most important reasoning tasks for the PUP: Decide whether there is a solution, and find an optimal solution, that is, one that uses the minimal number of control units. We are especially interested in the latter problem. For this we consider the corresponding decision problem: Is there a solution with a specified number of units? The rationale behind the optimization criterion is that (a) units are expensive, and (b) connections are cheap.

2.2 The Partner Units Problem is Intractable

By a reduction from BINPACKING, it can be shown that the optimization version of the PUP is intractable when $\mathit{InterUnitCap} = 0$, and $\mathit{UnitCap}$ is part of the input. Observe that clearly the PUP is in NP (cf. also section 3).

Theorem 1 ([1]). *Deciding whether a* PUP *instance has a solution with a given number of units is* NP*-complete, when* $\mathit{InterUnitCap} = 0$*, and* $\mathit{UnitCap}$ *is part of the input.*

In practice, however, the value of $\mathit{UnitCap}$ will typically be fixed.

2.3 Forbidden Subgraphs of the PUP

In solvable instances sensors cannot belong to arbitrarily many zones (and vice versa) [1]:

Lemma 1 (Forbidden Subgraphs of the PUP). *A* PUP *instance has no solution if it contains* $K_{1,n}$ *or* $K_{n,1}$ *as a subgraph, where* $n = ((\mathit{InterUnitCap} + 1) * \mathit{UnitCap}) + 1$.

2.4 K-regular Graphs

There is an interesting connection between most general solutions to the PUP and k-regular unit graphs, where a graph is k-regular if every vertex has exactly k neighbors: In k-regular unit graphs we are exploiting the $\mathit{InterUnitCap}$ capacity for connections between units to the fullest. Hence, k-regular unit graphs are the most general solutions (if they exist). In the case where $k = 2$, there is exactly one k-regular graph, the cycle; we exploit this fact in section 4. In the case where k is odd, k-regular unit graphs only exist if there is an even number of units ("hand-shaking lemma"). Moreover, for $k > 2$ the number of distinct most general unit graphs grows rapidly: E.g. for $k = 4$ there are six distinct graphs on eight vertices, and 8037418 on sixteen vertices; for twenty vertices not all distinct graphs have been constructed [13]. It can be shown that all these solution topologies can be forced:

Observation 1 (PUP instances and k-regular graphs) *For every k-regular graph G_k there exists a* PUP *instance G with InterUnitCap $= k$ such that in every solution of G the unit graph is G_k.*

Proof. Construct the instance G as follows:
1) First connect $2 * UnitCap$ vertices (i.e. sensors and zones) to each node in G_k. Let the set of all sensors (zones) be V_1 (V_2).
2) The instance G contains all edges (v_1, v_2), where $v_1 \in V_1$ and $v_2 \in V_2$ are connected to either the same or adjacent nodes in G_k.

We show that every solution is isomorphic to G_k. We consider two cases:
- $0 \leq k \leq 1$: The result is immediate.
- $k > 1$: Let u_0 be a node in G_k with neighbors $u_j : 1 \leq j \leq k$. Denote by $V_1^{u_i}$ and $V_2^{u_i}$ the sensors and zones created in G for $u_i : 0 \leq i \leq k$. Let G'_k be an optimal solution for G. We need two observations: (1) For each $0 \leq i \leq k$ both $V_1^{u_i}$ and $V_2^{u_i}$ are on the same unit in G'_k. (2) For $0 \leq i < j \leq k$ if $V_1^{u_i} \cup V_2^{u_i}$ and $V_1^{u_j} \cup V_2^{u_j}$ are connected in G then their units are connected in G'_k.

Hence, if $InterUnitCap > 2$, and there are no restrictions on the solution topology in the application domain, then it is practically not feasible to iteratively try all most general solution topologies. The solution topology will have to be inferred instead.

2.5 Bounds on the Number of Units Required

Let us next point out that the number of units used when solving an instance $G = (V_1, V_2, E)$ is bounded from below by $lb = \lceil \frac{\max(|V_1|,|V_2|)}{UnitCap} \rceil$. Clearly it can also be bounded from above by $ub = |V_1| + |V_2|$ — we never need empty units. If $InterUnitCap = 2$ and $UnitCap > 1$ we can show that the stronger $ub = \max(|V_1|, |V_2|)$ holds for connected instances [1]. Now, if there are multiple connected components C_i in the instance with upper bounds ub_i, then we have $ub = \sum ub_i$. We conjecture that this also holds for $InterUnitCap > 2$, but have so far been unable to prove it. These bounds are exploited in the problem encodings below either for keeping the problem model small, or to limit the depth of iterative deepening search. In this approach we first try to find a solution with lb units; if that fails increase lb by one; the first solution found will be optimal. For both approaches better upper bounds are desirable.

2.6 Symmetry breaking

If we don't use iterative deepening search, then in some problem models we might obtain solutions with empty units. Here we can do symmetry breaking, by demanding that whenever unit j has a sensor or zone assigned to it, then every unit $j' < j$ also has some sensor or zone assigned to it.

We can also rule out a lot of the connections between sensors and units (or alternatively, between zones and units) immediately. Consider sensors and units: Sensor 1 must be somewhere, so it might as well be on unit 1. Sensor 2 can either be on unit 1 or on a new unit, let's say 2, and so on. Unfortunately, we cannot do this on both sensors and zones, since the edges may disallow a zone and a sensor on the same unit.

3 Encodings for the General Case

We are next going to outline encodings of the PUP where *InterUnitCap* is an arbitrary fixed constant. Due to cost considerations we are especially interested in the optimization version of the PUP: We want to minimize the number of expensive units used, but do not consider the cost for the cheap connections between them.
In particular, we show how the problem can be encoded in the frameworks of propositional satisfiability testing (SAT), integer programming (IP), and constraint solving (CSP), all of which can be considered as state-of-the-art for optimization problems [11]. In addition we will also describe an encoding in answer set programming (ASP), a currently very successful knowledge representation formalism.

3.1 Answer Set Programming

First, we show how to encode the PUP in answer set programming [9, 12] which has its roots in logic programming and deductive databases. This knowledge representation language is based on a decidable fragment of first-order logic and is extended with language constructs such as aggregation and weight constraints. Already the propositional variant allows the formulation of problems beyond the first level of the polynomial hierarchy. In case standard propositional logic is employed [2], an answer set corresponds to a minimal logical model by definition of [12].
In our encodings a solution (i.e. a configuration) is the restriction of an answer set to those literals that satisfy the defined solution schema.
To encode a PUP instance in ASP we represent the zones and sensors by the unary predicates `zone/1` and `sensor/1`. The edges between zones and sensors are represented by the binary predicate `zone2sensor/2`. The number of available units `lower` $= \left\lceil \frac{\max(|Sensors|,|Zones|)}{2} \right\rceil$, `unitCap` and `interUnitCap` are each specified by a constant. The PUP is then encoded via the following logical sentences employing the syntax described in [3]:

```
(1)  unit(1..lower).
(2)  1 { unit2zone(U,Z) : unit(U) } 1 :- zone(Z).
(3)  1 { unit2sensor(U,S) : unit(U) } 1 :- sensor(S).
(4)  :- unit(U), unitCap+1 { unit2zone(U,Z): zone(Z) }.
(5)  :- unit(U), unitCap+1 { unit2sensor(U,S): sensor(S) }.
(6)  partnerunits(U,P) :- unit2zone(U,Z), zone2sensor(Z,S),
                          unit2sensor(P,S), U!=P.
(7)  partnerunits(U,P) :- partnerunits(P,U), unit(U), unit(P).
(8)  :- unit(U), interUnitCap+1 { partnerunits(U,P): unit(P) }.
```

The first statement generates the required number of units represented as facts: `unit(1). unit(2). ... unit(lower)`. The second and the third

[2] All literals in rules are negation free. $\bot$, $\rightarrow$, $\wedge$, $\vee$ are used to formulate (disjunctive) rules.

clause ensure that each zone and sensor is connected to exactly one unit. The edges between units and zones (rsp. sensors) are expressed by `unit2zone/2` (rsp. `unit2sensor/2`) predicates. We use cardinality constraints [17] of the form $l\ \{L_1, \ldots, L_n\}\ u$ specifying that at least l but at most u literals of $L_1, \ldots, L_n$ must be true. So called *conditions* (expressed by the symbol ":") restrict the instantiation of variables to those values that satisfy the condition. For example, in the second rule, for any instantiation of variable `Z` a collection of ground literals `unit2zone(U,Z)` is generated where the variable `U` is instantiated to all possible values s.t. `unit(U)` is true. In this collection at least one and at most one literal must be true.

The fourth and the fifth rule guarantee that one unit controls at most *UnitCap* zones and *UnitCap* sensors. In these rules the head of the rule is empty which implies a contradiction in case the body of the rule is fulfilled. The last three rules define the connections between units and limit the number of partner units to *InterUnitCap*. Note that rules 4, 5 and 8 can be rephrased by moving the cardinality constraint on the left-hand-side of the rule and adapting the boundaries. We used the depicted encoding because it follows the Guess/Check/Optimize pattern formulated in [12]. Depending on the particular encoding runtimes may vary.

Alternatively, ASP solvers provide built-in support for optimization by restricting the set of answer sets according to an objective function. For example, for minimizing the number of units, the upper bound on the number of units used has to be provided as a constant `upper` $= \max(|Zones|, |Sensors|)$. The unit generation rule of the original program (line 1) then has to be replaced by:

```
(1') unit(1..upper).
(2') unitUsed(U):- unit2zone(U,Z).
(3') unitUsed(U):- unit2sensor(U,S).
(4') lower { unitUsed(X):unit(X) } upper.
(5') unitUsed(X):- unit(X), unit(Y), unitUsed(Y), X<Y.
(6') #minimize[unitUsed(X)].
```

Here, the second and the third rule express the property that a used unit always has to be non-empty. Rule `4'` states that the number of used units must be between `lower` and `upper`. Rule `5'` expresses an ordering on the units: units with smaller numbers should be used first. This statement improves the performance of the solver. The last rule expresses that the optimization criterion is the number of units used in a solution.

3.2 Propositional Satisfiability Testing

We next show how to encode the PUP as a propositional satisfiability problem. We are given sensors $[1, S]$, zones $[1, Z]$, and units $[1, U]$, as well as *UnitCap* and *InterUnitCap*.

Let su_{ij} denote that sensor i is assigned to unit j, and zu_{ij} that zone i is assigned to unit j. First of all, every sensor and zone must belong to a unit, so

$$\forall 1 \leq i \leq S \bigvee_{1 \leq j \leq U} su_{ij} \text{ and } \forall 1 \leq i \leq Z \bigvee_{1 \leq j \leq U} zu_{ij}.$$

Furthermore, every sensor and zone belongs to at most one unit, therefore we have

$$\forall 1 \leq i \leq S. \forall 1 \leq j < j' \leq U. \left(\neg su_{ij} \vee \neg su_{ij'}\right)$$

and the same for zones.
Now we need to count both the number of zones and sensors on a unit, and forbid both numbers to be above *UnitCap*. For this we use a sequential counter, similar to the one presented in [18]. Let sc_{ijk} mean that unit j has k sensors assigned (ignore the i for now). We need to say that every sensor counts as one,

$$\forall 1 \leq i \leq S. \forall 1 \leq j \leq U. \left(su_{ij} \rightarrow sc_{ij1}\right),$$

and also that we increment this number when we see something new:

$$\forall 1 \leq i < i' \leq S. \forall 1 \leq j \leq U. \forall 1 \leq k \leq \mathit{UnitCap}. \\ \left(su_{i'j} \wedge sc_{ijk} \rightarrow sc_{i'j(k+1)}\right)$$

The fact that we keep track of what we have seen (using index i) is to make sure, for example, that sc_{ij5} is only true if there are five distinct sensors on a unit. Finally, we forbid too many sensors on a unit via

$$\forall 1 \leq i \leq S. \forall 1 \leq j \leq U. \neg sc_{ij(\mathit{UnitCap}+1)}.$$

Repeat this trick for zones using zc_{ijk}.
Finally, we need to use the edges. Let sz_{ij} be given, and mean that sensor i has an edge to zone j. Also, let uu_{ij} mean that units i and j are partnered. We need to define this as

$$\forall 1 \leq i \leq S. \forall 1 \leq j \leq Z. \forall 1 \leq k < k' \leq U. \\ \left(\left(\left(su_{ik} \wedge zu_{jk'}\right) \vee \left(su_{ik'} \wedge zu_{jk}\right)\right) \wedge sz_{ij} \rightarrow uu_{kk'}\right)$$

and also, by symmetry,

$$\forall 1 \leq i < j \leq U. \left(uu_{ij} \rightarrow uu_{ji}\right).$$

Now we can count the partnered units like we did before, using pc_{ijk}, and then forbidding $pc_{ij(y+1)}$. Technically, we don't need both uu_{ij} and uu_{ji}, but having both makes the encoding simpler in the definitions above. We may skip uu_{ii} — but we may also leave them in, as the clauses forcing uu_{ij} have $i < j$, and thus uu_{ii} is never forced. Therefore,

$$\forall 1 \leq i \leq j \leq U. \left(uu_{ij} \rightarrow pc_{ij1}\right),$$

and

$$\forall 1 \leq i < i' \leq U. \forall 1 \leq j \leq U. \left(uu_{i'j} \wedge pc_{ijk} \rightarrow pc_{i'j(k+1)}\right).$$

Finally, we forbid too many partners, and we are done:

$$\forall 1 \leq i \leq j \leq U. \neg pc_{ij(\mathit{InterUnitCap}+1)}.$$

3.3 Integer Programming

We next show how the PUP can be encoded into integer programming. If $\mathit{InterUnitCap} = 2$ we set $|\text{Units}| = \max(|\text{Sensors}|, |\text{Zones}|)$; otherwise it is $|\text{Units}| = |\text{Sensors}| + |\text{Zones}|$. Then we make matrices of Boolean variables su_{ij} (and zu_{ij}, respectively) sensor s_i (zone z_i) is assigned to unit u_j. These matrices are constrained to enforce that each sensor/zone is assigned exactly one unit, and that no unit is assigned more than $\mathit{UnitCap}$ sensors/zones:

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

The zone-units matrix looks identical. Next we need a Boolean variable $\mathit{UnitUsed}_i$ that indicates whether u_i is assigned any sensors/zones. This can be achieved by constraints $su_{ji} \leq \mathit{UnitUsed}_i$ and $zu_{ji} \leq \mathit{UnitUsed}_i$, for all j. Observe that in principle even for unused units $\mathit{UnitUsed}_i$ can be set to one — a possibility that will be excluded by the objective function.

For the constraints on the connections between units it is convenient to increase $\mathit{InterUnitCap}$ by one, and stipulate that every unit is connected to itself. We then obtain a symmetric matrix of Boolean uu_{ij} variables, which can be used to indicate whether unit i is connected to unit j:

$$\begin{array}{cccc|c} 1 & uu_{1,2} & uu_{1,3} & \ldots & \sum \leq \mathit{InterUnitCap}+1 \\ uu_{2,1} & 1 & \ldots & \ldots & \sum \leq \mathit{InterUnitCap}+1 \\ uu_{3,1} & \ldots & 1 & \ldots & \sum \leq \mathit{InterUnitCap}+1 \end{array}$$

In addition to enforcing that $\mathit{InterUnitCap}$ is not exceeded, the entries in this matrix are subject to the following constraints:

- $uu_{ij} = uu_{ji}$ (symmetry); and
- $uu_{ij} \geq (su_{ki} + zu_{lj}) - 1$, for all connections (s_k, z_l) between sensors and zones — if a sensor s_k and a zone z_l are connected yet assigned different units u_i, u_j then these units are connected.

This model allows more connections between units than are actually needed, in this case mandating a post-processing step for solutions.

As a last constraint we add that the number of units used is bounded from below:

$$\lceil \frac{\max(|\text{Sensors}|, |\text{Zones}|)}{2} \rceil \leq \sum_j \mathit{UnitUsed}_j.$$

Finally, we add the objective function $\sum_j \mathit{UnitUsed}_j$, subject to minimization. As usual, first a linear relaxation with cost C is solved, and only then is the problem solved over the integers, posting the cost C as a lower bound.

3.4 Constraint Satisfaction Problem

Finally, we model the Pup as a Csp by letting sensors and zones be variables $S = \{s_1, \ldots, s_n\}$ and $Z = \{z_1, \ldots, z_m\}$. For the domains we use (a numbering of) the units $U_1, \ldots, U_n$.
We post a global cardinality constraint `gcc(`U_i`,[`$s_1, \ldots, s_n$`],`C`)` on the sensors for every U_i , where C is a variable with domain $\{0, \ldots, \mathit{UnitCap}\}$, and do likewise for the zones. These constraints ensure that each unit occurs at most *UnitCap* times in any assignment to S and Z.
Tracking connections between units via Boolean variables is done using a matrix of Boolean uu_{ij} variables as in the integer programming model, but using cardinality constraints to count the number of ones.
In addition for each connection (s, z) we post implicational constraints that exclude the value j from the domain of sensor s if z is assigned to unit i and $uu_{ij} = 0$ (and vice versa):

$$(s = U_i \wedge uu_{ij} = 0) \rightarrow z \neq U_j \text{ and } (z = U_i \wedge uu_{ij} = 0) \rightarrow s \neq U_j$$

4 A Special Case: *InterUnitCap* = 2

In this section we focus on the case where *InterUnitCap* = 2. We first briefly recall the fundamental ideas of our recent tractability results for this case; for the details the interested reader is referred to [1]. We then show how the fundamental ideas from this work can be incorporated into the Pup encodings presented above.

4.1 A Specialized Algorithm for *InterUnitCap* = 2

The basic observation in the case *InterUnitCap* = 2 is that the unit graph in a solution of a connected Pup instance is always either a path or a cycle. This holds because the number of neighbors of a unit is bounded by two. Based on this observation we have developed a non-deterministic algorithm DecPup that decides the Pup. Basically, DecPup recursively guesses the contents of the units. It turns out that this can be done in NLogSpace by exploiting the notion of a path decomposition; this non-deterministic algorithm can then be turned into a polynomial backtracking search procedure.
Let us now turn to those of the ideas we use for the DecPup algorithm that can be incorporated into the other problem models: We first observe that cyclic unit graphs are more general solution topologies than paths. Any solution that is a path can be extended to a cycle, but the converse is clearly false. Hence, for a fixed number of units used in the solution, we can assume a fixed cyclic layout of the units throughout the search. By using iterative deepening search (on the number of units used) we can find optimal solutions first.
In this context let us point out that branch-and-bound-search for optimal solutions (again on the number of units used) does not work: e.g. a $K_{6,6}$ graph does not admit solutions with more than three units.
Note also that finding optimal solutions gets more complicated if there are multiple connected components in the input graph. DecPup can then

still be used to compute optimal solutions in polynomial time — but only if there are at most logarithmically many connected components in the input graph. Part of the problem is that any two connected components may either have to be assigned to the same, or to two distinct unit graph(s). A priori it is unclear which of the two choices leads to better results. E.g. if we assume that $\mathit{UnitCap} = 2$ then two $K_{3,3}$ should be placed on one cyclic unit graph, while two $K_{6,6}$ must stand alone.
Note that with cycles for unit graphs there are two kinds of rotational symmetry: For any given solution with unit graph $U_1, \ldots, U_n, U_1$ there also are identical solutions $U_2, \ldots, U_n, U_1, U_2$, etc.; in addition, there is also $U_n, U_{n-1}, \ldots, U_1, U_n$. We can break this symmetry without additional computational cost by requiring that

- the first sensor is assigned to unit U_1; and
- the second sensor appears somewhere on the first half of the cycle.

We have prototypically implemented the DecPup algorithm in Java (for connected graphs), and will use it below in the evaluation of the other encodings for the case $\mathit{InterUnitCap} = 2$. The implementation features memoization of no-good units to avoid the rediscovery of unsolvable subproblems, and two-step forward-checking: Checking whether there is enough space for the open neighbours of the current unit on the current plus the next unit (step one), and doing the same for the open neighbours of the open neighbours (step two).

4.2 Adapting the Encodings to $\mathit{InterUnitCap} = 2$

To some extent the ideas presented above can be incorporated into the other problem models: If we use iterative deepening search, then we can assume a fixed cyclic layout of the units for each depth. Then, the connections between units are given, something that greatly simplifies the problem models. It also allows us to use symmetry breaking as defined in section 4.1 above. For example, in the constraint model we can drop the Boolean matrix that tracks the connections between units, and simplify the implicational constraints for a connection (s, z) to

$$s = U_i \rightarrow z \in \{U_{i-1}, U_i, U_{i+1}\} \text{ and } z = U_i \rightarrow s \in \{U_{i-1}, U_i, U_{i+1}\}.$$

The adaptations for Asp and Sat are similar [1].
If we are not doing iterative deepening search, that is, the maximum available number of units in the model is given by the upper bound, then this does not work, as it is not clear where to close the cycle. Especially for the integer programming model this constitutes a challenge: If we use iterative deepening we lose the objective function.
To guide the search, we can, by a simple greedy algorithm, compute an ordering of the variables that ensures that each sensor (or zone) has some predecessor that has already been assigned to a unit; we assume that an arbitrary sensor (or zone) is fixed initially. If variables are assigned in this order then the number of possible unit choices per zone (or sensor) is bounded by three throughout the search, instead of $\mathit{NoOfUnits}$. However, to the best of our knowledge neither integer programming tools nor Asp- or Sat-solvers usually provide this level of control over variable ordering to the user.

5 Evaluation

We have evaluated our encodings on a set of benchmark instances that we received from our partners in industry. All experiments were conducted on a 3 GHz dual core machine with 4 GB RAM running Fedora Linux, release 13 (Goddard). In general in our experiments we have imposed a ten minute time limit for finding solutions.
For the evaluation of the different encodings of the Pup we use the Sat-solver MiniSat v2.0 [14], the constraint logic programming language ECL^{i}PS^e-Prolog v6.0 [7], and Clingo v3.0 [3] from the Potsdam Answer Set Solving Collection (Potassco). For evaluating the integer programming model we have used Cbc v2.6.2 in combination with Clp v1.13.2 from the COIN-OR project [4], and IBM's Cplex v12.1 [5].
In the Asp, Sat and Csp models, as well as in DecPup, we use iterative deepening search for finding optimal solutions, as this has proven to be the most efficient. We did not try this in the integer programming model, as we would lose the objective function in doing so.
The reader is advised to digest the results presented below with caution: We are using both the Sat and the integer programming solvers out of the box, whereas for the Csp model we employ the variable ordering heuristics outlined in the previous section. Moreover, if $\mathit{InterUnitCap} > 2$, for the Asp model we employ the following advanced feature: a portfolio solver Claspfolio, which is a part of Potassco [3], comes with a machine learning algorithm (support vector machine) that has been trained on a large set of Asp programs. Claspfolio analyzes a new Asp program (in our case the Pup program), and configures Clingo to run with options that have already proved successful on similar programs. It is likely that such machine learning techniques could also be developed and fruitfully applied in the other frameworks.

5.1 Experimental Results

$\mathit{InterUnitCap} = 2$ All instances are based on rectangular floor plans, and all instance graphs are connected. In all instances there is one zone per room, and by default there are sensors on all doors. Only the grid-* and tri-* instances feature external doors. For an illustration see Figure 2, which shows a rectangular 8×3 floor plan with external doors on two sides of the building.
Apart from that, the instances are structured as follows:

- dbl-* consist of two rows of rooms with all interior doors equipped with a sensor.
- dblv-* are the same, only that there are additional zones that cover the columns.
- tri-* are grids with only some of the doors equipped with sensors. There are additional zones that cover multiple rooms.
- grid-* are not full grids, but some doors are missing, and there are no rooms (zones) without doors.

The runtimes we obtained for the various problem encodings described above are shown in seconds in Table 1 (a "*" indicates a timeout). The Cost column contains the number of units in an optimal solution; a slash "/" in that column indicates that no solution exists.

Table 1. Structured Problems with $\mathit{InterUnitCap} = \mathit{UnitCap} = 2$

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	*	*	*
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
grid-90	50	68	97	34	0.04	4.51	0.03	1.53	*	21.19
grid-91	50	63	97	32	0.10	*	*	0.92	*	16.60
grid-92	50	65	97	33	0.49	*	*	0.87	*	17.40
grid-93	50	58	97	29	0.13	2.68	*	1.75	*	13.41
grid-94	50	66	97	33	0.04	3.66	*	1.61	*	*
grid-95	50	60	97	30	0.02	3.90	0.48	0.97	*	18.34
grid-96	50	62	97	31	0.07	3.30	*	0.87	*	13.62
grid-97	50	64	97	32	0.02	3.67	*	0.86	*	17.90
grid-98	50	59	97	30	0.03	*	*	1.19	*	12.30
grid-99	50	65	97	33	0.03	*	202.48	1.16	*	20.35

$\mathit{InterUnitCap} > 2$ For the general case we have also tested our encodings on a set of benchmark instances where $\mathit{InterUnitCap} = 4$ that we obtained from our partners in industry:

- tri-* are exactly as before, only with $\mathit{InterUnitCap} = 4$.
- grid-* are as before, only that a bigger number of doors exists.

5.2 Analysis

Any conclusions drawn from our experimental results have to be qualified by the remark that, of course, in every solution framework there are many different problem models, and there is no guarantee that our problem models are the best ones possible.

Table 2. Structured Problems with *InterUnitCap* = 4 and *UnitCap* = 2

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	*	*

Let us begin our analysis of the results by highlighting a peculiarity of the Pup: While it is possible to construct instances that require more than the minimum number of units, it is not straight-forward to do so, and such instances also appear to be rare in practice: In our experiments in no solution are there more units than the bare minimum required. It is clear that iterative deepening search thrives on this fact, whereas the integer programming model suffers.

InterUnitCap = 2 The combination of assuming a fixed cyclic unit graph together with iterative deepening search resulted in drastic speed-ups for the Asp, Sat, and Csp solvers. Symmetry breaking did not have much effect — except on the unsolvable instances.

The Asp and the Sat encoding show broadly similar behavior: Both Clingo and MiniSat use variations of the DPLL-procedure [6] for reasoning. Oddly, they even both get faster at some point as problem size increases on the dblv-* instances. However, Clingo does significantly better on the grid-like instances. Interestingly, machine learning did not help for the Asp encoding specialized to *InterUnitCap* = 2; hence the results shown were obtained using both solvers out of the box.

For the Csp encoding the variable ordering is the key to the good results: Without the variable ordering the Csp model performs quite poorly. The absence of a similar variable selection mechanism from both Asp and Sat in our experiments might explain the surprising superiority of Csp on most benchmarks.

The inconsistent results for DecPup are particularly striking. On the one hand, DecPup performs excellently on the dbl-* instances. But in general, it disappoints. Possibly this might be due to the following: DecPup

has a "local" perspective on the problem, that is, it only can see the current and past units; the subsequent units are only created at runtime. In all the other encodings all units are present from the beginning, something which, in one way or another, facilitates propagating the current variable assignment to other units.
The IP encoding is not yet fully competitive. It particularly struggles with the dblv-* instances. In general, the commercial CPLEX is at least one order of magnitude faster than the open source CBC.
It is also interesting to compare the dbl-* with the dblv-* instances, as the latter are obtained from the former by adding constraints. Both CLINGO and MINISAT thrive on the additional constraints, contrary to ECL^{i}PS^e, CBC, CPLEX and DECPUP.

InterUnitCap > 2 In this setting, for finding solutions the symmetry breaking methods from section 2.6 did increase computation time for the CSP, the SAT, and the IP model. However, symmetry breaking again does help when proving an instance unsatisfiable. The results in Table 2 were obtained without symmetry breaking.
If CLASPFOLIO's machine learning database is not used to configure options of CLINGO, then the two DPLL-based programs again perform quite similar, with Clingo slightly having the edge (results not shown). With machine learning CLINGO clearly is the winner, with the main benefits stemming from the following: Use the VSIDS heuristics [15] instead of the BerkMin heuristics [10], and exploit local restarts [16]. Note that MINISAT also uses the VSIDS heuristics.
Interestingly, the CSP-encoding now disappoints. Given that the same variable ordering is used, this may have to be attributed to insufficient propagation when tracking the connections between units.
Again our IP encoding is not on par yet. But for this encoding comparing the instances tri-30,32,34 in Tables 1 and 2 is particularly instructive: This is basically the same model in both settings, only that in the latter case there are more variables due to the higher upper bound on the number of required units.

6 Future Work

There is still significant work to be done on the PUP: Almost all interesting complexity questions are still open, and a thorough investigation of these questions should eventually lead to better algorithms and encodings for the PUP. It should also be possible to prove better upper bounds, in particular ones that depend on *UnitCap*; especially the integer programming model would benefit from this. It would be interesting to see what the variable ordering heuristics can do for SAT and ASP. More generally, the major challenge is to find stronger problem models in the various frameworks and to improve the implementation of DECPUP, the only algorithm guaranteed to run in polynomial time.
Acknowledgment. We greatly appreciate the helpful comments from the anonymous reviewers.

References

1. Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., , Ryabokon, A., Thorstensen, E.: Tackling the Partner Units Problem. Tech. Rep. RR-10-28, Computing Laboratory, University of Oxford (2010), available from the authors
2. Third International Answer Set Programming Competition 2011. https://www.mat.unical.it/aspcomp2011/ (2011)
3. The Potsdam Answer Set Solving Collection. http://potassco.sourceforge.net/
4. COIN-OR CLP/CBC IP solver. http://www.coin-or.org/
5. IBM ILOG CPLEX IP solver. http://www.ibm.com/
6. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3) (1960)
7. ECL^{i}PS^e-Prolog. http://eclipseclp.org/
8. Falkner, A., Haselböck, A., Schenner, G.: Modeling Technical Product Configuration Problems. In: Proceedings of the Configuration Workshop at ECAI'10 (2010)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP'88 (1988)
10. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Proceedings of DATE'02 (2002)
11. Hooker, J.N.: Integrated Methods for Optimization. Springer, New York (2006)
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3) (2006)
13. Meringer, M.: Regular Graphs Page. http://www.mathe2.uni-bayreuth.de/markus/reggraphs.html
14. Minisat SAT solver. http://www.minisat.se
15. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of DAC'01 (2001)
16. Ryvchin, V., Strichman, O.: Local restarts. In: Proceedings of SAT'08 (2008)
17. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2) (2002)
18. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: Proceedings of CP 2005. Springer (2005)