The Partner Units Problem
A Constraint Programming Case Study

Conrad Drescher, *Department of Computer Science, University of Oxford*

***Abstract*—The Partner Units Problem is a challenging combinatorial search problem that originates in the domain of security and surveillance. Technically it consists of partitioning a bipartite graph under side conditions. In this work we describe how constraint programming technology can be leveraged to tackle the problem. We address problem modelling, symmetry breaking and problem-specific search strategies. We introduce the best search strategy known to date as well as a powerful new implied constraint for pruning the search space. Finally, we present implementations in ECL^{i}PS^e Prolog and the MINION constraint solver and compare these to a state-of-the-art dedicated algorithm.**

I. The Partner Units Problem

Consider a museum where we want to keep track of the number of visitors occupying certain parts of the building. To this end we divide the rooms into a set of (possibly overlapping and non-contiguous) zones $\mathcal{Z}$ and place a set of sensors $\mathcal{S}$ on the doors between different zones. The Partner Units Problem (PUP) [1] now 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. 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). See Figure 1 for an illustration.

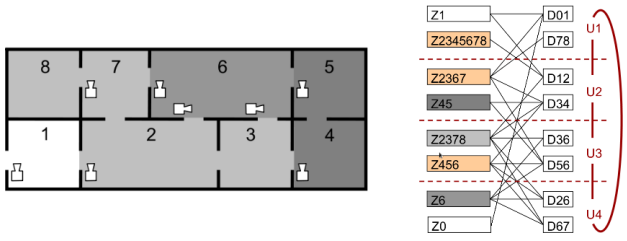

Fig. 1. A floor plan is divided into zones with sensors on the doors between zones (left). In the bipartite graph (right) zones are connected to the respective sensors. Zones are named after the rooms covered and sensors after the zones connected. The graph partitioning shown (U1-U4) constitutes an optimal solution for $\mathit{InterUnitCap} = \mathit{UnitCap} = 2$.

Clearly the PUP is not limited to the museum domain, but may arise in manifold security and surveillance applications. See [2] for a description of how the PUP arises in traffic monitoring for railways or in peer-to-peer networks.

Work and cooperation with Siemens Austria funded by EPSRC Grant EP/G055114/1.

In this work we show how constraint programming technology can be leveraged to tackle the PUP. Constraint programming (CP) [3] is a powerful tool for solving combinatorial search problems that has its roots in fundamental AI research. Today CP together with integer programming and Boolean satisfiability solving (SAT) constitutes the standard in combinatorial optimisation [4]. Contrary to SAT and integer programming, where search is typically conducted autonomously, in a typical CP toolkit *problem-specific* search strategies can be programmed by hand.

The remainder of this paper is organised as follows: After introducing the formal basics in Section II we review the literature on the problem in Section III. We then present our CP model, including search strategies, techniques for symmetry breaking and a novel implied constraint. For the empirical evaluation in Section V we consider problem models in ECL^{i}PS^e Prolog and the MINION constraint solver as well as a state-of-the-art dedicated algorithm. Finally, in Section VI we conclude with a short discussion.

II. Preliminaries

A. The Partner Units Problem

Formally the Partner Units Problem is defined as follows:

Definition 1 (Partner Units Problem)**.** *Given a connected bipartite instance graph $G = (\mathcal{S}, \mathcal{Z}, \mathcal{E})$ on sensors $\mathcal{S}$ and zones $\mathcal{Z}$ as well as concrete values for the parameters* UnitCap[1] *and* InterUnitCap*, a solution to the Partner Units Problem is a partitioning of the graph into a set $\mathcal{U}$ of non-empty bags such that each bag*

- *contains at most* UnitCap *vertices from $\mathcal{S}$ and $\mathcal{Z}$ each; and*
- *has at most* InterUnitCap *adjacent bags, where the bags U_1 and U_2 are adjacent whenever $s \in U_1$ and $z \in U_2$ and $(s, z) \in \mathcal{E}$.*

To every solution of the PUP we associate a *solution graph*: The nodes are the bags of the solution and the edges represent the adjacency relation between bags.

We are especially interested in finding optimal solutions; that is, solutions that use a minimum number of units. The minimum number of units needed for a solution is $lb = \lceil \frac{\max(|\mathcal{S}|, |\mathcal{Z}|)}{\mathit{UnitCap}} \rceil$; unless there are empty units the number of units used cannot be greater than $|\mathcal{S}| + |\mathcal{Z}|$ [5].

Let us recapitulate the known complexity results [5] for deciding whether there is a solution using at most k units. The

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

problem is NP-complete if *InterUnitCap* $= 0$ and *UnitCap* is a parameter. The case where *InterUnitCap* $= 2$ is tractable for fixed *UnitCap*. Intuitively, this case is easier because, w.l.o.g., on a given number of units the solution graph can be fixed to a simple cycle. The complexity for the case of a fixed *InterUnitCap* > 2 is still open. Here, however, the number of different possible solution layouts grows exponentially with the number of units: Recall that a graph is k-regular if each vertex has exactly k neighbours: Hence *InterUnitCap*-regular graphs are most general solution topologies [6].

B. Constraint Programming

We briefly recall the basic ideas of constraint programming over finite domains; for an in-depth treatment we refer the reader to [3]. In CP we consider a finite set of variables, where each variable is associated with a finite domain of admissible values. A constraint is a relation on a subset of the variables that specifies the admissible combinations of values. A simple constraint is e.g. the equality constraint $x = y$. So-called global constraints accept an arbitrary finite number of variables. An example is *alldifferent*$(\vec{x})$ which constrains all the variables in $\vec{x}$ to take different values. A CP solver finds values for all variables such that all constraints hold.

The fundamental concept in CP is constraint propagation. This can very roughly be described as follows: Given a change to the domain of one of the variables constraint propagation tries to reduce the domains of the other variables. E.g. in the case of *alldifferent* after setting one of the $\vec{x}$ to a value V this value V can be removed from the domain of all the other variables. The propagation algorithm actually employed depends on the constraint's semantics, but typically has a low polynomial runtime. Nevertheless there is a trade-off between the cost for constraint propagation and its benefit in the form of a pruned search space. As propagation alone does not suffice to find solutions CP tool-kits interleave it with a search procedure that instantiates variables.

The two CP solvers considered in this work, ECL^{i}PS^e and MINION, both use a standard backtracking search procedure. In ECL^{i}PS^e the constraint solving is embedded into Prolog as a host language [7], [8]. The platform comes with a plethora of libraries e.g. for real-valued arithmetic or hybrids of CP and integer programming and is very good for rapid prototyping. ECL^{i}PS^e adopts a "glass-box" model: It is relatively easy to implement new custom propagators or to modify propagator behaviour. MINION on the other hand is a pure constraint solver and adopts a "black box" model for constraint solving [9]. This means the user is not supposed to extend the solver and has access to only a limited number of search strategies. However, MINION is carefully engineered to offer scalable constraint solving by sheer speed of propagation. For example, it can detect and avoid attempts at constraint propagation that cannot result in new information [10].

III. PREVIOUS WORK

A first set of encodings for the PUP in different formalisms such as SAT or constraint handling rules as well as heuristic search methods have been presented in [1]. Better, albeit still basic, encodings as SAT and as a constraint, integer and answer set program have then been presented in [6], together with experimental results on the polynomial algorithm for the case of *InterUnitCap* $= 2$ from [5]. Here, for the case of *InterUnitCap* > 2, the conflict-driven clause learning solvers used for the SAT encoding and the answer set program performed best. However, some instances were still beyond the reach of these methods. Interestingly, for the case of *InterUnitCap* $= 2$, the polynomial algorithm performed only poorly; it was the CP model that performed best.

Due to space constraints no explicit CP model was given in [6]. The models described in the present work are improved and extended versions thereof.

Next the recent QUICKPUP and QUICKPUP* algorithms from [2] constituted a real breakthrough, capable of quickly finding solutions for all our instances. Let us briefly recapitulate here what these algorithm do: Prior to search the sensors and zones are ordered via a breadth-first traversal of the instance graph starting from some zone. The sensors and zones are assigned in that order. For example, on the instance graph from Figure 1 we obtain the ordering $\langle Z_1, D_{01}, D_{12}, Z_0, Z_{2345678}, Z_{2367}, Z_{2378}, D_{26}, D_{34}, D_{36}, D_{56}, D_{67}, D_{78}, Z_{45}, Z_{456}, Z_6\rangle$, starting at zone Z_1 and breaking ties lexicographically. This ordering has first been used in [6].

Next, both algorithms employ restarts: The available computation time is distributed evenly between the vertex orderings obtained by starting from different zones.

Now, whenever QUICKPUP tries to assign a sensor or a zone to a unit it first tries a fresh unit; only if that fails all previously used units are tried successively (in decreasing order). Finally, after having constructed a large solution, QUICKPUP tries to contract this into a smaller one in a simple greedy fashion. In the QUICKPUP* variant the previously used units are tried first before finally a fresh unit is used. Contrary to QUICKPUP the solutions found by QUICKPUP* are always optimal.

The PUP constraints are enforced as follows: An object-oriented representation of the current partial solution is maintained throughout search.[2] There is no constraint propagation: For example, attempts to place sensors or zones on already full units are not precluded — they simply fail. This makes for an interesting comparison with CP: The respective search strategies are easily emulated, but constraint propagation may offer some benefit.

IV. THE CP MODEL

We next describe the CP model. We address modelling, search strategies and symmetry breaking as well as the respective implementation techniques in ECL^{i}PS^e. Finally we discuss how to port this implementation to MINION.

A. The Instances

Let us first describe our benchmark instances. Most have been provided by our partners in industry; some have been crafted to highlight model properties. For all our instances

[2] QUICKPUP is implemented in JAVA.

InterUnitCap $\in \{2, 4\}$ and *UnitCap* $= 2$. The industrial instances are built from an underlying grid of rooms, where each room is a zone. There are sensors on all doors and there may be larger zones covering multiple rooms.

1) InterUnitCap $= 2$*:* We use the same industrial instances as in [6], [2]. The "triple" instances are similar to the grid instance depicted below in Figure 3. As the grids or the overlapping zones grow larger these instances become unsolvable for *InterUnitCap* $= 2$. The "double" instances consist of two rows of rooms stacked on top of each other; cf. Figure 2. In the "doublev" instances the columns form overlapping zones.

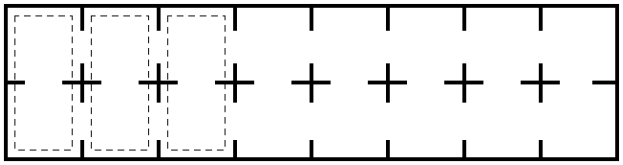

Fig. 2. The layout of the "double"- and "doublev"-instances.

2) InterUnitCap > 2*:* Here we use two sets of solvable instances, *Old* and *New*. A typical example is depicted in Figure 3. Instances may contain multiple such blocks stacked on top of each other.

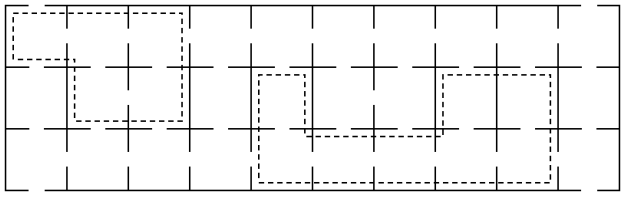

Fig. 3. A typical instance with overlapping zones (dotted).

a) Old: These are the instances also used in [6], [2]. All twenty instances are built atop a grid of rooms, with some doors blocked and some overlapping zones. All these instances have solutions using a maximum number of units.

b) New: On the one hand, these also contain grid-like problems, but with more and larger overlapping zones. On the other hand, we include some hand-crafted instances that admit only solutions of minimum size. For this we use the construction for enforcing arbitrary solution topologies from [6].

B. A Basic ECL^{i}PS^e Model for InterUnitCap $= 2$

We next describe a specialised model for the case of *InterUnitCap* $= 2$, which is of some interest for our industrial partners. The case of *InterUnitCap* $= 2$ deserves a separate treatment because of the following: For a fixed number n of units we may assume without loss of generality that the units form a cycle $\langle U_1, U_2, \ldots, U_n \rangle$; the solution topology does not have to be inferred. The CP model is shown in Figure 4.

Assuming we use n units for each sensor s (or zone z) we introduce a variable s_v (or z_v) with domain $1..n$; if s_v is assigned to i this means that s is to be placed on the i-th unit. A constraint of the form $\mathit{atmost}(\mathit{UnitCap}, \mathcal{S}_v, i)$ enforces that the value i occurs at most *UnitCap* times in the set of variable $\mathcal{S}_v$ [11]. Repeat for all i and for the zones $\mathcal{Z}_v$. For the "partner units" constraint $\mathit{pu}(s_v, z_v)$ we use the following observation: If for an edge $(s, z) \in \mathcal{E}$ sensor s is assigned to unit U_i there are at most three possible units for zone z (and vice versa). We have considered two models:

$$
\begin{array}{l}
|\mathcal{S}_v| = |\mathcal{S}|, |\mathcal{Z}_v| = |\mathcal{Z}| \\
s_v \in \{1, \ldots, n\}, \text{ all } s_v \in \mathcal{S}_v \\
z_v \in \{1, \ldots, n\}, \text{ all } z_v \in \mathcal{Z}_v \\
\mathit{atmost}(\mathit{UnitCap}, \mathcal{S}_v, i), \text{ all } 1 \leq i \leq n \\
\mathit{atmost}(\mathit{UnitCap}, \mathcal{Z}_v, i), \text{ all } 1 \leq i \leq n \\
\mathit{pu}(s_v, z_v), \text{ all } (s, z) \in \mathcal{E}
\end{array}
$$

Fig. 4. The CP model for n units and an instance graph $G = (\mathcal{S}, \mathcal{Z}, \mathcal{E})$.

- Using a custom propagator that (1) waits until either s_v or z_v are instantiated and (2) then restricts the other variable's domain [6].
- Using an arithmetic constraint with a variable sz_{diff} for the expressing the distance between s and z:

  $$
  \begin{array}{l}
  |\mathcal{SZ}_{\text{diff}}| = |\mathcal{E}| \\
  sz_{\text{diff}} \in \{-(n-1), -1, 0, 1, (n-1)\}, \text{ all } sz_{\text{diff}} \in \mathcal{SZ}_{\text{diff}} \\
  s_v - z_v = sz_{\text{diff}}, \text{ all } (s, z) \in \mathcal{E}
  \end{array}
  $$

  This model will react to any change to the domain of s_v or z_v, not just to instantiation.

The *atmost* constraints on the sensors (or the zones) can be combined into a single global cardinality constraint [12]: For two vectors *vals* and *cvars* of variables of the same length, the constraint $\mathit{gcc}(\mathit{vars}, \mathit{vals}, \mathit{cvars})$ enforces that each *val* occurs *cvar* times in *vars*. Here *val* is fixed, but *cvar* may be a domain variable. In our case we have e.g. $\mathit{gcc}(\mathcal{S}_v, [1, \ldots, n], \mathit{cvars})$, with each *cvar*'s domain set to $\{0, \ldots, \mathit{UnitCap}\}$.

C. A Basic ECL^{i}PS^e Model for InterUnitCap > 2

In this setting we have to infer the layout of the solution. For the respective parts of the ECL^{i}PS^e model used in [6] see Figure 5; the remainder of the model is as in Figure 4.

$$
\begin{array}{l}
|\mathcal{UU}| = n^2 \\
uu_{i,j} \in \{0, 1\}, \text{ all } uu_{i,j} \in \mathcal{UU}, i, j \in \{1, \ldots, n\} \\
uu_{i,i} = 1, uu_{i,j} = uu_{j,i}, \text{ all } i, j \in \{1, \ldots, n\} \\
\mathit{atmost}(\mathit{InterUnitCap} + 1, \mathcal{UU}[i, \_], 1) \text{ all } 1 \leq i \leq n \\
\mathit{pu}(s_v, z_v, \mathcal{UU}), \text{ all } (s, z) \in \mathcal{E}
\end{array}
$$

Fig. 5. The CP model for n units and an instance graph $G = (\mathcal{S}, \mathcal{Z}, \mathcal{E})$.

Connections between units are modelled via a symmetric $n \times n$ matrix $\mathcal{UU}$ of Boolean variables uu_{ij} corresponding to

a connection from unit i to unit j. We set the diagonal UU_{ii} to "one". The *InterUnitCap* constraints are then enforced as $\mathit{atmost}(\mathit{InterUnitCap}+1, \mathcal{UU}[i], 1)$, were $\mathcal{UU}[i,\_]$ denotes the i-th column of the matrix.

Finally, we need to propagate the "Partner Units" constraint. To this end we have written a custom propagator $\mathit{pu}(s_v, z_v, \mathcal{UU})$ that operationally behaves as follows: (1) If either s or z is assigned to unit u_i (say s is) then we add the constraint $\mathit{uu}_{iv} = 0 \Rightarrow z \neq v$ for every value v in the current domain of z. (2) This implicational constraint is checked only when uu_{iv} is instantiated — ignoring all changes to the domain of z. (3) If both s and z are assigned the respective Boolean is set to "one".

Note that we can safely set an inter-unit-connection Boolean UU_{ij} to "zero" only when for all pairs of variables s_v, z_v, $(s,z) \in \mathcal{E}$, the combination i,j does not appear in the respective domains. We consider this not worth detecting.

Let us describe two model variants:

(M1) The matrix can be folded along the diagonal, removing the redundant variables.

(M2) The "partner units" constraint can also be modelled via an $\mathit{element}(\mathcal{UU}, i, 1)$ constraint: On a list representation of $\mathcal{UU}$ this constraint enforces that the i-th element is a "one", where i is a domain variable ranging over the list indices [13]. For constraining the index variable i we have considered two models: (a) We can post the arithmetic constraint $((s_v - 1) * n) + z_v = i$ for each edge $(s,z) \in \mathcal{E}$ in the instance graph. (b) We can precompute all possible solutions to the above arithmetic constraint, that is, the tuples $\langle j,k,l\rangle$ such that $((j-1)*n)+k = l$ for all $j,k \in \{1,\ldots,n\}$. Storing these extensionally we can post a so-called table constraint $\mathit{table}(s_v, z_v, i)$ for all $(s,z) \in \mathcal{E}$, ensuring that the tuple of variables $\langle s_v, z_v, i\rangle$ is instantiated to one of the tuples listed. In both approaches the constraint will be checked whenever the domain of one of the variables changes. The *element* constraint will also be checked whenever the domain of i changes.

D. Variable and Value Orderings

The order in which variables and values are selected for instantiation is well known to have dramatic impact on the performance of CP solvers.

1) Variable Orderings: The variables *uu* for the inter-unit-connections always come last in our variable orderings. Once all sensor and zone variables are instantiated the remaining uninstantiated *uu* variables can all be set to "zero". This means in our CP model a variable $uu_{i,j}$ will only be set to

- "one" if there are s_v, z_v on units i,j such that $(s,z) \in \mathcal{E}$;
- "zero" if the maximum number of "ones" has been reached in row i or column j.

For $\mathit{InterUnitCap} = 2$ the breadth-first ordering of the sensor and zone variables[3] resulted in good runtimes for the CP model in [6]. However, a closer look revealed the following: All the results were obtained by variable orderings starting from a corner of the underlying grid (zone one). Redoing the experiments we observed that with other orderings (starting zones) the same model frequently timed out.

[3]Below we will simply speak of "variables".

This observation led to the following idea: Consider e.g. a large "double" instance as depicted in Figure 2. If we order the variables in a breadth first manner starting from somewhere in the middle of the instance then the instance is decomposed into two large remaining connected components. But the sensor and zone variables from these connected components will be interleaved in the ordering and they will share the same possible values/units. The solver has to discover the graph decomposition by costly trial and error.

As all our instances are built from an underlying grid (possibly containing overlapping zones) we can use the following idea for a variable ordering:

(1) Walk the underlying grid in such a way that it is not decomposed.
(2) Add in the overlapping zones as soon as all their neighbours are covered.

In such a variable ordering the remainder of the graph will consist of one large component (the remainder of the underlying grid) and possibly singleton vertices (overlapping zones with their neighbours covered). Call such a variable ordering almost-non-decomposing. Let us point out that depending on the entry point a breadth first ordering can be almost-non-decomposing, too. Note that for general bipartite PUP instance graphs such orderings do not always exist.

2) Value Orderings: For the old CP-model from [6] the default increasing value ordering of most CP solvers was used. The search strategies of QUICKPUP and QUICKPUP* can be captured by value orderings (1) and (2), respectively, where we remember throughout search the current "fresh" unit:

(1) Try the "fresh" unit first, then try the preceding units.
(2) Try the preceding units first before trying the fresh unit.

Values other than the fresh unit are tried in decreasing order (more recent units are preferred). The respective value orderings are computed from a variable's domain prior to instantiating it for the first time.

Next we propose a new search strategy (PRED):

(3) For each variable remember which units those variables preceding it in the ordering were assigned to; but do this only for variables which are up to two edges away in the instance graph. Try these units first. If that fails, try the fresh unit. If that fails, try the remaining preceding units.

The idea is to place sensors and zones on the same units as those neighbours in the instance graph that precede them in the ordering; this should require fewer new connections between units to be made. If that fails a fresh unit may be more promising than more distant predecessors. In order to implement (3) prior to search we compute for each variable the set of its successors in the ordering that are at most two edges away in the instance graph. We then maintain throughout search for each variable the set of units that its predecessors have been assigned to: Whenever a variable is instantiated, we update this set for each of its successors. We compute the PRED value ordering from the variable's current domain whenever a variable is instantiated for the first time.

Observe that these value orderings are not compatible with the CP model for the case $\mathit{InterUnitCap} = 2$: With the solution topology fixed in this model for a neighbour of the first

variable we have to consider the first, the second and the last unit, two of which are "fresh". In this setting we simply use the default ascending value ordering.

E. Restarts

Restarts of the search have been employed with great success in [2], where up to $|\mathcal{Z}|$ restarts have been used, i.e. one per zone. A new variable ordering is computed at every restart, starting from the respective zone. The idea is that each potential "bottleneck" zone is located at the top of the search tree once. Here we adopt the same restart model.

F. Finding Optimal Solutions

In [6] two basic strategies for finding optimal solutions have been proposed:

(1) Make a model that contains upper bound many units and try to maximise the number of empty units (real optimisation in a single CSP).
(2) Start with lower bound many units; if no solution can be found increase the number of units by one (iterative deepening search on a sequence of CSPs).

Strategy (2) has been very successful. Almost all our instances are solvable and the solvable ones can all be solved using just lower bound many units: We only have artificially created instances where this does not hold; cf. Figure 6. Moreover, the strongest upper bound on the number of units required for a solution that is currently known ($|\mathcal{S}|+|\mathcal{Z}|$) must be considered weak as it does not depend on *UnitCap*. Strategies (1) and (2) are implemented by adjusting the sensors' and zones' domain size. QUICKPUP is similar to (1), whereas QUICKPUP* and PRED also work well with (2).

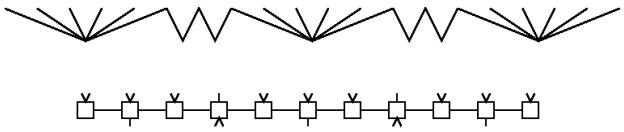

Fig. 6. A PUP instance (top) that forces a solution with two units not filled to the maximum ($\mathit{UnitCap} = \mathit{InterUnitCap} = 2$): The $K_{1,(\mathit{InterUnitCap}+1)*\mathit{UnitCap}}$ subgraphs in the instance admit only solutions with $\mathit{InterUnitCap}+1$ units. We can then trap single vertices between these $K_{1,6}$ subgraphs. Clearly the pattern can be extended to the right or to $\mathit{InterUnitCap} > 2$. The construction shown is the core of the non-minimal constructions we know to date.

G. Symmetry Breaking

The PUP exhibits a lot of symmetry. Breaking (some of) these symmetries allows to avoid the repeated exploration of identical subtrees of the search tree containing no solution.

1) InterUnitCap = 2: There are two types of rotational symmetry: Rotating a cyclic solution like a wheel gives a new solution as does flipping it like a coin. These symmetries are broken as follows: (1) Fix some sensor or zone to the first unit and (2) require that some other sensor or zone appears on the first half of the cycle formed by the units [5].

2) InterUnitCap > 2: In [6] the following two basic ideas for symmetry breaking have been proposed:

(1) For some ordering of the sensor and zone variables, variable v_1 has to be on some unit, say unit one; variable v_2 might be on the same unit or on a fresh one, say unit two; and so on.
(2) For some ordering of the units, if unit j is non-empty then unit i must not be empty for all $i < j$. This only applies if there are more units than the bare minimum.

Part (2) is particularly important because of the following: Assume we have a partial solution using units $U_1, \ldots, U_n$, and that extending this solution by using the fresh unit U_{n+1} fails. Now, in the absence of (2), trying to extend the solution using unit U_m will fail, too, all $m > n+1$, leading to so-called thrashing of the search engine.

Here we make the observation that (1) and (2) together enforce *precedence*$(U_i, U_j, \vec{v})$ constraints in the sense of [14] on the sensor and zone variables for all $U_i < U_j$:[4] Let $\vec{v}$ be the sequence of ordered sensor and zone variables and let the units be numbered $1..n$. Then in any assignment to $\vec{v}$ we have that, whenever unit j occurs in the assignment, then there is an earlier occurrence of unit i.

Originally, we have implemented (1) via statically restricting the variables' domains, and (2) via reified constraints, similar to the "if-then" constraints described in [14], as the *precedence* constraint is not available in ECL^{i}PS^e: Assume that $\mathit{vars} = \langle v_1, v_2, \ldots, v_n \rangle$ is an ordering of the sensor and zone variables and that the units are ordered $\langle U_1, U_2, \ldots, U_m \rangle$. For each pair U_i, U_{i+1} we post constraints

$$v_k = U_{i+1} \Rightarrow \bigvee_{l<k} v_l = U_i,$$

for every k such that $\lceil \frac{i+1}{\mathit{UnitCap}} \rceil \leq k \leq i+1$. This was found to be computationally even more expensive than not using symmetry breaking at all [6]. Here we observe that a *precedence* constraint on the sensor and zone variables is also enforced if we use one of the QUICKPUP, QUICKPUP* or PRED value orderings: The value orderings achieve the symmetry breaking, there is no need to post additional constraints.

Symmetry breaking can also be done by restricting the possible inter-connections between units instead of the possible assignments from sensors and zones to units. In its simplest form symmetry breaking on Boolean matrices like the one for the inter-unit-connections is done via lexicographic ordering constraints $\mathrm{Row}_i >=_{\mathrm{lex}} \mathrm{Row}_j, i < j$; more advanced methods are described in [15], [16]. All these methods, however, are not compatible with restricting the potential assignments from sensors and zones to units.

H. Checking Maximum Joint Vertex Degree

We next describe a novel condition that can be exploited to detect dead-ends earlier during search. Instances that contain a vertex of degree $d > (\mathit{InterUnitCap}+1) * \mathit{UnitCap}$ are unsolvable [5]: The neighbours require more neighbouring units than allowed. Now assume that during search we have

[4] We assume (1) and (2) use the same ordering of the units.

assigned two sensors to some unit that together have more than $(\mathit{InterUnitCap}+1)*\mathit{UnitCap}$ adjacent zones: Such a partial solution can also never be extended to a full solution. We refer to this condition as a violation of the *Maximum Joint Vertex Degree (MJVD)*. Such a violation can either be detected by exhaustively trying all possibilities for placing the respective neighbours on units, or by simple counting.

In ECL^{i}PS^e we have implemented the counting check as follows: For every unit we keep track of those sensors/zones that have to be assigned to this unit or one of its neighbours, using two duplicate-free lists. We update the respective list whenever we assign a sensor/zone; we fail as soon as some list grows too long. Alternatively, instead of lists we can introduce vectors of Booleans of length $|\mathcal{S}|$ and $|\mathcal{Z}|$, respectively. An entry i in the vector is set to one if vertex i has to be on this unit or on one of its direct neighbours. Finally we constrain the vectors to contain at most $(\mathit{InterUnitCap}+1)*\mathit{UnitCap}$ "ones"; cf. Figure 7. The advantage of using lists is that their length is bounded by a constant.

$$
\begin{array}{l}
|U^i_{\mathcal{S}}| = |\mathcal{S}|, |U^i_{\mathcal{Z}}| = |\mathcal{Z}|, \text{ all } U^i \in \mathcal{U} \\
U^i_{\mathcal{S}}, U^i_{\mathcal{Z}} \in \{0,1\}, \text{ all } U^i \in \mathcal{U} \\
s_j = i \Rightarrow \bigwedge_k U^i_{\mathcal{Z}}[k] = 1, \text{ all } s_j \in \mathcal{S}, (s_j, z_k) \in \mathcal{E} \\
z_j = i \Rightarrow \bigwedge_k U^i_{\mathcal{S}}[k] = 1, \text{ all } z_j \in \mathcal{Z}, (s_k, z_j) \in \mathcal{E} \\
\mathit{atmost}((\mathit{InterUnitCap}+1)*\mathit{UnitCap}, U^i_{\mathcal{S}}, 1), \text{ all } 1 \le i \le n \\
\mathit{atmost}((\mathit{InterUnitCap}+1)*\mathit{UnitCap}, U^i_{\mathcal{Z}}, 1), \text{ all } 1 \le i \le n
\end{array}
$$

Fig. 7. A pure CP model for checking the MJVD.

I. Porting the ECL^{i}PS^e Model to MINION

Next we describe issues arising when porting the ECL^{i}PS^e model to MINION. MINION employs specialised algorithms for variables of different types. We use *DISCRETE* variables for $\mathcal{S}$ and $\mathcal{Z}$.

1) InterUnitCap $= 2$: The ECL^{i}PS^e model from Figure 4 is fully supported by MINION. For the "partner units" constraint in MINION we use the arithmetic formulation as MINION does not support custom propagators.

2) InterUnitCap > 2: Here we also use *DISCRETE* variables for the index variables for the *element* constraint and *BOOL* for the Booleans for the inter-unit-connections.

Next, like ECL^{i}PS^e MINION does not support the *precedence* constraint. However, it also does not allow the user to program value orderings like QUICKPUP or PRED. Hence the precedence constraints have to be encoded using logical combinations of (in-)equalities.

For checking the MJVD in MINION we use the pure CP model from Figure 7. Folding the matrix for the inter-unit-connections in MINION cuts run times in half.

V. EXPERIMENTAL RESULTS

All experiments were done on the same 3 GHz dual core machine with 4 GB RAM running Fedora Linux as in [6].

A. Comparing Search Strategies

For search strategies with restarts we use a fixed timeout per zone (ordering): If the timeout is one second an instance with ten zones gets a maximum of ten seconds computation time. We start from a timeout of one second, with gradual increases in steps of half a second. The maximum time considered per zone/ordering is ten seconds. The time-outs given below are the smallest such that a given number of instances can be solved. Note that higher time-outs may lead to smaller overall runtimes. By tuning the time-out parameter to properties of the instances as described in [17] better results should be attainable. For QUICKPUP we haven't implemented the minimising of solutions; the times reported are just for finding (usually very large) solutions.

Table I contains results for instances with $\mathit{InterUnitCap} = 4$. An entry of the form $x@y$ reads: "x instances solved in y seconds". Three instances of the *Old* instance set could not be solved using a timeout of at most ten seconds — which means up to 1460 s total computation time per instance. The PRED strategy generally does best. QUICKPUP struggles the most on the *New* instances that do not allow large solutions.

Instances	Timeout	QP	QP*	PRED	PRED$^-$
Old (20)	1 s	3@1511	5@1307	13@780	13@775
	2 s	3@3021	7@2262	**17@1026**	
	4.5 s		**17@2465**		
	6.5 s	**17@2395**			
New (9)	1 s	4@195	5@211	**9@65**	
	4 s	**9@83**	8@231		
New (6)	2 s	3@472	**6@65**	**6@25**	1@529
	5 s	4@860			

TABLE I
STRUCTURED PROBLEMS WITH $\mathit{InterUnitCap} = 4$ AND $\mathit{UnitCap} = 2$

As explained above in Section IV-D2 for $\mathit{InterUnitCap} = 2$ we only consider the default increasing value ordering.

B. Checking MJVD

In Table I above PRED$^-$ denotes a model with the check for violations of the MJVD turned off. On the *Old* instances these never occur as there are no vertices of sufficiently high degree. On the crafted New(6) instances, however, they occur frequently. The overhead incurred by the check appears to be negligible, while the potential benefits are big.

C. Variable Orderings and Restarts

The instance-specific idea of almost-non-decomposing variable orderings work very well on the "double" and "doublev" instances for $\mathit{InterUnitCap} = 2$. Figure 8 shows typical results. On these instance families restarts are not needed.

On less simply structured instances, however, running times are less predictable; cf. Figure 9. In particular, almost-non-decomposing orderings can not be seen to be superior. Hence restarts will remain a powerful tool until we gain a better understanding of the relationship between the instance graph and good variable orderings.

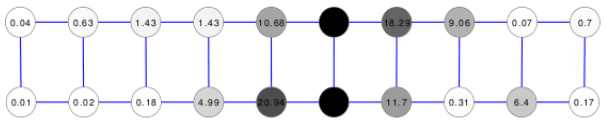

Fig. 8. The running times obtained by different entry points for the bfs ordering on "double-20". Nodes coloured from 0 s (white) to 30 s (black). Only zones are shown; there are doors/sensors on all edges.

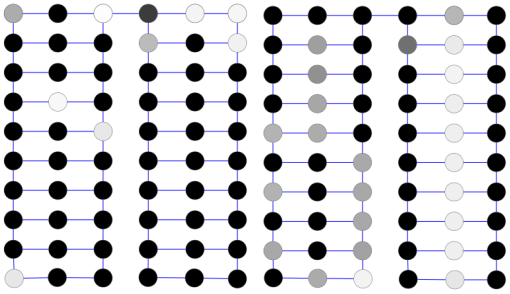

Fig. 9. The running times obtained by different entry points for the bfs ordering on "triple-60". Nodes coloured from 0 s (white) to 10 s (black). The left figure is for *InterUnitCap* $= 2$, the right for *InterUnitCap* $= 4$.

D. Symmetry Breaking

For *InterUnitCap* > 2 we have also built a model where we lexicographically order the matrix for inter-unit-connections, using $\text{Row}_i >=_{\text{lex}} \text{Row}_j, i < j$. Hence no precedence constraint on the sensor/zone variables can be used. On our instances this model was almost three times slower than a model with no symmetry breaking at all. The latter model could solve five instances in 4130 s with a time-out of 3.5 s per zone; with a time-out of 10 s it couldn't solve more than those five instances.

E. Model Variants

ECL^{i}PS^e features a prototypical implementation of Régin's algorithm for the *gcc* constraint [18], where the *cvar* variables are represented by lower and upper bounds, i.e. their domain may not contain holes. Using *gcc* constraints instead of *atmost* contraints increases the running times by more than one order of magnitude in ECL^{i}PS^e.

1) InterUnitCap $= 2$*:* We have first compared the two different models of the "partner units" constraint. In Table II ECL^{i}PS^e I denotes the custom propagator and ECL^{i}PS^e II the arithmetic constraint: ECL^{i}PS^e II does much better. All running times are for almost-non-decomposing bfs variable orderings starting from "zone one" without restarts. We have also run our general model with the PRED search strategy on the same instances. Here an entry $x@y$ reads: Solved the instance in x seconds using a timeout of y seconds per variable ordering. A "–" denotes a timeout. The variable orderings and the order in which they are considered are the same in all three models. Fixing the solution topology as in the specialised model clearly performs better.

2) InterUnitCap > 2*:* Here we consider the model variants (M1) and (M2) from Section IV-C:

Instance	$\lvert\mathcal{U}\rvert$	ECL^{i}PS^e I	ECL^{i}PS^e II	PRED
double-20	14	0.47	0.02	2.8@0.5s
double-200	149	14.40	2.36	–@10s
doublev-30	15	0.03	0.03	1.1@1.5s
doublev-180	90	3.77	3.02	–@10s
triple-30	20	0.54	0.03	–@10s
triple-60	40	45.98	0.37	–@10s

TABLE II
ECL^{i}PS^e RESULTS FOR *InterUnitCap* $=$ *UnitCap* $= 2$

(M1) Folding the matrix has no significant impact on performance in ECL^{i}PS^e.

(M2) The *element*$(list, i, value)$ constraint in ECL^{i}PS^e does not allow variables in *list*; hence we have developed an appropriate version ourselves. It (1) removes from *index*'s domain those values corresponding to elements of *list* such that *value* is not in the element's domain; and (2) sets the respective *list* entry to *value* when *index* gets instantiated. When the domain of one of the list values is modified it repeats (1). This could be done much more efficiently if ECL^{i}PS^e reported which values were removed from the respective domains. The combination of *element* and arithmetic constraints is orders of magnitude slower than using our custom propagator. Using a timeout of two seconds per variable ordering PRED solves only three instances in 2807 seconds.

F. Comparing ECL^{i}PS^e and MINION

In Table III we report runtimes for MINION and ECL^{i}PS^e on a representative instance for *InterUnitCap* $= 4$, using identical variable orderings, but no restarts or precedence constraints. In the absence of a custom propagator for the "partner units" constraint MINION is slightly slower than ECL^{i}PS^e. It is better to constrain the *index* of the *element* constraint via *table* constraints than via arithmetic ones (MINION $^+$), which timed out after ten minutes. Just like in ECL^{i}PS^e encoding the precedence constraint on the sensor and zone variables via Boolean combinations of (dis-)equalities increases running times by orders of magnitude (results not shown).

It can also be seen that interestingly in MINION for *InterUnitCap* $= 4$ using single *gcc* constraints is superior to using multiple *atmost* constraints,[5] with the *gccweak* variant having slightly the edge. MINION's highly optimised implementation of the *gcc* constraints [19] here makes all the difference to ECL^{i}PS^e's prototypical implementation.

ECL^{i}PS^e	MINION	MINION ^{g}	MINION ^{gw}	MINION $^+$
1.50	19.83	6.55	6.39	–

TABLE III
RUNNING TIMES FOR "TRIPLE-90" WITH *InterUnitCap* $= 4$.

As illustrated by Figure 10 for *InterUnitCap* $= 2$ MINION is much faster than ECL^{i}PS^e. It can also be seen that now in MINION using multiple *atmost* constraints is superior to single *gcc* or *gccweak* constraints. The overhead incurred by the pure CP check for the MJVD in MINION is so big that even on the *New(6)* instances we obtain no gain.

[5]The latter are called *occurrencesleq* in MINION.

G. Comparison with the QUICKPUP Implementation

For *InterUnitCap* $= 4$ we have run the JAVA implementation of QUICKPUP* on the *Old* and *New* instance sets. It takes about seven seconds for each set, at a timeout of one second per variable ordering. For *InterUnitCap* $= 2$ in Figure 10 we report results on larger "double"-instances that show how all the different approaches scale. ECL^{i}PS^e here denotes the CP model with the PRED search strategy.

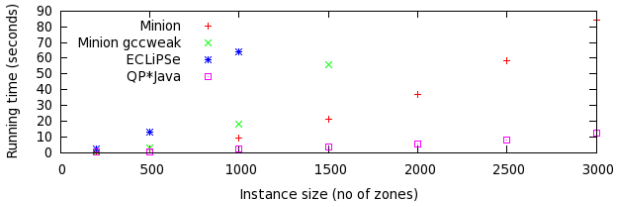

Fig. 10. Running times for larger "double" instances.

We can see that — at least in the proposed model/solver combinations — the CP models can not yet compete with the dedicated implementation of the QUICKPUP* algorithm.

VI. DISCUSSION

We believe that the somewhat disappointing runtimes of the CP models stem from different sources: In the case of ECL^{i}PS^e employing propagation-avoiding techniques like those used in MINION should prove beneficial. MINION on the other hand does not admit a lightweight propagator of the "partner units" constraint for *InterUnitCap* $= 4$. In the case of *InterUnitCap* $= 2$ it is almost on a par with the JAVA implementation of QUICKPUP*. Our work shows that for a CP solver to be truly competitive both is needed: Carefully implemented propagators that avoid futile attempts at propagation and the support for user-defined lightweight propagators. We have also seen that the choice of propagator depends on implementation details in the solver.

Our work also provides a better understanding of the good performance of QUICKPUP and QUICKPUP*: This has to be attributed foremost to (1) powerful yet inexpensive symmetry breaking via value orderings that enforce precedence constraints and (2) a highly efficient implementation for checking the PUP constraints.

On the positive side this paper has contributed the best search strategy known to date for the PUP in the case of *InterUnitCap* > 2. We note that by considering neighbours more than two edges away the PRED strategy can gradually be turned into the QUICKPUP* strategy. Finally, the previously unknown MJVD condition provides an inexpensive and potentially very useful check for pruning the search space. The current state-of-the-art, the QUICKPUP* implementation in JAVA, should benefit from incorporating both these contributions.

Thanks We wish to thank Andreas Falkner, Gerhard Friedrich and Erich Teppan for providing us with their paper on, and implementation of, the QUICKPUP and QUICKPUP* algorithms. Special thanks also for making the paper available to the reviewers. We also wish to thank Chris Jefferson for helpful comments on MINION.

REFERENCES

[1] A. Falkner, A. Haselböck, and G. Schenner, "Modeling Technical Product Configuration Problems," in *Workshop on Configuration at the 19th European Conference on Artificial Intelligence ECAI 2010, Lisbon, Portugal*, L. Hotz and A. Haselböck, Eds., 2010.

[2] E. Teppan, A. Falkner, and G. Friedrich, "QuickPup: A Heuristic Backtracking Algorithm for the Partner Units Configuration Problem," in *Proceedings of the 24th Conference on Innovative Applications of Artificial Intelligence, IAAI 2012, Toronto, Canada*, M. Fromherz and H. Munoz-Avila, Eds. AAAI press, 2012, Preprint available at http://proserver3-iwas.uni-klu.ac.at/restricted/iaai.pdf. User: restricted, Pwd: D0wnL04d. **Please treat confidentially (patent pending)**.

[3] F. Rossi, P. van Beek, and T. Walsh, Eds., *The Handbook of Constraint Programming*. Elsevier, 2006.

[4] J. N. Hooker, *Integrated Methods for Optimization*. New York: Springer, 2006.

[5] M. Aschinger, C. Drescher, G. Gottlob, P. Jeavons, and E. Thorstensen, "Tackling the Partner Units Configuration Problem," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011, Barcelona, Catalonia, Spain*, T. Walsh, Ed. AAAI press, 2011, pp. 497–503.

[6] M. Aschinger, C. Drescher, G. Friedrich, G. Gottlob, P. Jeavons, A. Ryabokon, and E. Thorstensen, "Optimization Methods for the Partner Units Problem," in *Proceedings of the 8th International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2011, Berlin, Germany*, ser. Lecture Notes in Computer Science, T. Achterberg and J. C. Beck, Eds., vol. 6697. Springer, 2011, pp. 4–19.

[7] M. Wallace, J. Schimpf, and S. Novello, "ECLiPSe: A Platform for Constraint Logic Programming," *ICL Systems Journal*, vol. 12, pp. 159–200, 1997.

[8] J. Schimpf and K. Shen, "ECLiPSe - From LP to CLP," *Theory and Practice of Logic Programming*, vol. 12 (1-2), pp. 127–156, 2012, http://eclipseclp.org/.

[9] I. P. Gent, C. Jefferson, and I. Miguel, "Minion: A Fast Scalable Constraint Solver," in *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI'06, Riva del Garda, Italy*, ser. Frontiers in Artificial Intelligence and Applications, 2006, pp. 98–102, http://minion.sourceforge.net/.

[10] ——, "Watched Literals for Constraint Propagation in Minion," in *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming, CP'06, Nantes, France*, ser. Lecture Notes in Computer Science, F. Benhamou, Ed., vol. 4204. Springer, 2006, pp. 182–197.

[11] "The *atmost* constraint," http://www.emn.fr/z-info/sdemasse/gccat/Catmost.html.

[12] "The *global cardinality* constraint," http://www.emn.fr/z-info/sdemasse/gccat/Cglobal_cardinality.html.

[13] "The *element* constraint," http://www.emn.fr/z-info/sdemasse/gccat/Celement.html.

[14] Y. C. Law and J. H.-M. Lee, "Global Constraints for Integer and Set Value Precedence," in *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, CP 2004, Toronto, Canada*, ser. Lecture Notes in Computer Science, M. Wallace, Ed., vol. 3258. Springer, 2004, pp. 362–376.

[15] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh, "Breaking Row and Column Symmetries in Matrix Models," in *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, CP 2002, Ithaca, NY, USA*, ser. Lecture Notes in Computer Science, P. van Hentenryck, Ed., vol. 2470. Springer, 2002, pp. 462–476.

[16] G. Katsirelos, N. Narodytska, and T. Walsh, "On the Complexity and Completeness of Static Constraints for Breaking Row and Column Symmetry," in *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming, CP 2010, St. Andrews, Scotland, UK*, ser. Lecture Notes in Computer Science, D. Cohen, Ed., vol. 6308. Springer, 2010, pp. 305–320.

[17] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An Automatic Algorithm Configuration Framework," *Journal of Artificial Intelligence Research (JAIR)*, vol. 36, pp. 267–306, 2009.

[18] J.-C. Régin, "Generalized Arc Consistency for Global Cardinality Constraint," in *Proceedings of the 13th National Conference on Artificial Intelligence, AAAI 96, Portland, Oregon*, W. J. Clancey and D. S. Weld, Eds. AAAI Press, 1996, pp. 209–215.

[19] P. Nightingale, "The Extended Global Cardinality Constraint: An Empirical Survey," *Artificial Intelligence*, vol. 175, no. 2, pp. 586–614, 2011.