Partner Units Revisited [⋆]

Conrad Drescher[2], Gerhard Friedrich[1], Martin Gebser[3], Konstantinos Koiliaris[2], Anna Ryabokon[1], Marius Schneider[3]

1 Institut für Angewandte Informatik, Alpen-Adria-Universität Klagenfurt
2 Department of Computer Science, University of Oxford
3 Institute of Computer Science, Potsdam University

Abstract. The Partner Units Problem is a challenging combinatorial search problem that has applications in the domains of security and surveillance. In this work we look at problem-specific implied constraints, search strategies and symmetry breaking approaches. We discuss the modelling and solving techniques offered by SAT solvers and answer set programming on the one hand, and constraint solvers on the other hand.

1 Introduction

In this work we revisit the Partner Units Problem (PUP), a graph partitioning problem with applications in the areas of security and traffic monitoring [10]. Informally the PUP can be described as follows: Consider a set of sensors that are associated to 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).

Formally, the PUP consists of partitioning the vertices of the bipartite instance graph $G = (\mathcal{S}, \mathcal{Z}, E)$ into a set $\mathcal{U}$ of 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 $(s, z) \in E$.

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 $\mathcal{S} \cup \mathcal{Z} \cup \{v_{U_i} \mid U_i \in \mathcal{U}\}$ 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*. Throughout this paper

[⋆] Work and cooperation with Siemens Austria 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.

assume $\mathit{InterUnitCap} \geq 2$; for $\mathit{InterUnitCap} \in \{0, 1\}$ the problem is similar to classical bin-packing [22]. We also assume the underlying instance graph G to be connected.

We are especially interested in finding solutions that use a minimum number of units — the rationale behind this optimisation criterion is that (a) units are expensive, and (b) connections are cheap. This problem can be recast as a decision problem: Is there a solution with a specified number of units?

The problem is NP-complete if $\mathit{InterUnitCap} = 0$ and $\mathit{UnitCap}$ are part of the input [2]. In the same work it has been shown that the case where $\mathit{InterUnitCap} = 2$ and $\mathit{UnitCap} = k$ is tractable for fixed k via the NLOGSPACE algorithm DECPUP. Intuitively, this case is easier because the unit graph can be fixed to a simple cycle. The complexity of the case where both $\mathit{InterUnitCap}$ and $\mathit{UnitCap}$ are arbitrary fixed constants is still unknown. But it can be shown that the problem is tractable for instance graphs of bounded treewidth and suitably restricted choices for the sensor and zone assignments [22].

The minimum number of units needed for a solution is $lb = \lceil \frac{\max(|\mathcal{S}|, |\mathcal{Z}|)}{\mathit{UnitCap}} \rceil$. If $\mathit{InterUnitCap} = 2$ a solution using at most $\max(|\mathcal{S}|, |\mathcal{Z}|)$ units can be found if one exists. If $\mathit{InterUnitCap} > 2$ the best such bound known is the trivial $|\mathcal{S}| + |\mathcal{Z}|$ [2].

In [1] basic encodings of the PUP as SAT, CSP, integer and answer set programming (ASP) have been presented and evaluated. Both the DECPUP algorithm and the integer programming encoding did not perform well.

In this work in Section 2 we identify problem properties that may help prune the search space. Then in Section 3 we look at search strategies and in Section 3.3 we address symmetry breaking. Next in Section 4 we present our experiments with CDCL-solvers, local search and classical constraint solvers. Finally, in Section 5 we conclude.

2 Implied Constraints

The following problem properties may help prune the search space.

2.1 Shared Neighbours

If two vertices share a neighbour in the instance graph, then their respective units have a maximum distance of two in the unit graph. For a higher number of shared neighbours a stronger condition holds:

Proposition 1 (Common Neighbours). *If two vertices v_1 and v_2 of the instance graph have $n \geq (\mathit{InterUnitCap} * \mathit{UnitCap}) + 1$ common neighbours then in the solution graph they will be in the same unit or in directly connected units.*

Proof. Assume to the contrary that v_1 and v_2 are assigned to some units U_1 and U_2 that have distance two. Now all their common neighbours have to be assigned to units between U_1 and U_2. However, there can not be more than $\mathit{InterUnitCap}$ units between U_1 and U_2, and hence the common neighbours won't fit.

This result is not very strong as in solvable instances any two vertices can share at most $(\mathit{InterUnitCap}+1)*\mathit{UnitCap}$ neighbours.[2] But let $\mathit{InterUnitCap}=2$ and assume furthermore $|\mathcal{S}|>4*\mathit{UnitCap}$ or $|\mathcal{Z}|>4*\mathit{UnitCap}$: Then the solution contains at least five units and there can be at most one unit between the units that v_1 and v_2 are assigned to. Thus v_1 and v_2 have to be on the same unit or on directly connected units if they share at least $\mathit{UnitCap}+1$ neighbours.

2.2 Cyclic Solutions

If $\mathit{InterUnitCap}=2$ and the instance graph contains a cycle of a certain length then the unit graph is cyclic and of bounded length.

Proposition 2 (Cycle Property). *If the instance graph contains a cycle of length n, where $n>(4*\mathit{UnitCap})$, then the unit graph of the solution is cyclic of length m with $\lceil \frac{n}{2*\mathit{UnitCap}} \rceil \leq m \leq n$.*

Proof. Observe that a cycle of length $n>(4*\mathit{UnitCap})$ in the input graph requires at least three units in the unit graph. Hence the unit graph has to be cyclic as otherwise the Partner Units condition would be violated. Now let us go through the bounds of m: ($m \leq n$) In a unit graph of length n each unit contains only one vertex from the cycle — this is the longest we can stretch the cycle. ($\lceil \frac{n}{2*\mathit{UnitCap}} \rceil \leq m$) This is the smallest number of units that the vertices on the cycle fit on.

Assume an instance contains cycles of length n_1 and n_2 such that $n_1 < \lceil \frac{n_2}{2*\mathit{UnitCap}} \rceil$. We can use at most n_1 units and need at least $\lceil \frac{n_2}{2*\mathit{UnitCap}} \rceil$: The instance is unsolvable.

2.3 Maximum Joint Vertex Degree

Instances that contain a vertex of degree $d>(\mathit{InterUnitCap}+1)*\mathit{UnitCap}$ are unsolvable [2]. Assume that during search we have assigned sensors to some unit that together have more than $(\mathit{InterUnitCap}+1)*\mathit{UnitCap}$ adjacent zones: Such a partial solution can never be extended to a full solution. This idea generalises to cliques of units: Given a clique of size $1 \leq n \leq \mathit{InterUnitCap}+1$ in a unit graph, each of the clique elements has $n-1$ partner units in the clique, so that the number of partner units outside the clique is at most $\mathit{InterUnitCap}-(n-1)$. That is, the number of the clique elements' partner units cannot exceed $n*(\mathit{InterUnitCap}-(n-1))+n = n*((\mathit{InterUnitCap}+2)-n)$.

Proposition 3 (Maximum Joint Vertex Degree (MJVD)). *If a partial solution contains a clique of units of size n, then the sensors (or zones) assigned to the clique elements must not have more than $[n*((\mathit{InterUnitCap}+2)-n)]*\mathit{UnitCap}$ adjacent zones (or sensors).*

Cliques of size $n=1$ (a vertex) and $n=2$ (an edge) have a simple structure, making them good candidates for the exploiting the MJVD. Of all the implementations described in [1] only the DECPUP algorithm catches a violation of MJVD for single units; note that such a violation cannot occur on the benchmarks from [1].

[2] If they do they have to be assigned to the same unit.

3 Search Strategies

Next we look at search strategies for the PUP.

3.1 Strategies for Finding Optimal Solutions

In [2] 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).
(2) Start with lower bound many units; if no solution can be found increase the number of units by one (iterative deepening search).

Strategy (2) has been very successful: Solvable instances typically have solutions with lower bound many units; only on crafted instances does this not hold. Moreover, the known upper bounds (cf. Section 1) are very weak (they do not depend on *UnitCap*).

Here we propose the following hybrid optimisation strategy (3): Start with (2) for lower bound many units. If that fails the instance probably has no solution. Via (1) we can prove this or find an optimal solution in one step instead of many steps using (2).

3.2 Variable Orderings

Good variable orderings are known to be crucial for the performance of CP solvers. But they can also be used for static symmetry breaking (see below).

A basic idea is to order the variables corresponding to sensors and zones such that each but the first variable is connected in the instance graph to a predecessor in the ordering. This can e.g. be done by a greedy algorithm that (1) fixes one sensor and (2) at each step picks all vertices that are adjacent to already ordered vertices but not yet part of the ordering. For $\mathit{InterUnitCap} = 2$ this variable ordering (called `adjbfs` below) resulted in very good performance for the CSP-encoding in [1].

Next assume an instance contains more sensors than zones: It should be easier to fit the zones into the gaps after assigning the sensors than the other way around.

Then there are two fundamental ideas underlying variable orderings in general: Either try to find a most-constrained variable that is likely to lead to failure soon or try to find a variable that activates as many new constraints as possible. The former goal is called contention and the latter simplification in [33].

The following variable orderings are variations on the above ideas:

- First fix all sensors, then all zones (`sensorszones`), or the other way around (`zonessensors`).
- Pick a zone, pick all adjacent sensors, pick an adjacent zone, etc. (`topological`).
- Pick the vertex that has the most connections to already ordered vertices, use the degree of the vertices as a tie-breaker (`maxadjdeg`).
- Pick an adjacent vertex with max-degree, use the greater number of connections to already ordered vertices as a tie-breaker (`maxdegadj`).
- The same as `maxdegadj`, but prefer the least number of connections to already ordered vertices (`maxdegminadj`).
- Simply order the vertices according to their degrees(`maxdeg`,`mindeg`).
- Use a random ordering (`random`).

3.3 Symmetry Breaking

The PUP exhibits a lot of symmetry. Symmetry breaking is of great practical importance for unsolvable instances and also in subtrees of the search tree that contain no solutions.

Assigning Sensors and Zones to Units First we can do symmetry breaking by restricting the possible assignments from sensors and zones to units. This can be done in a static and in a dynamic way.

(Statically) For $\mathit{InterUnitCap} = 2$ in combination with iterative deepening search symmetry can be broken by (1) assigning some vertex to unit one and (2) enforcing that some other vertex appears on the first half of the cycle formed by the units [1].

For the case of $\mathit{InterUnitCap} > 2$ in [1] the following idea has been proposed: Sensor one has to be on some unit, say unit one; sensor two might be on the same unit or a fresh one, say unit two; and so on. If we have $|\mathcal{S}| + |\mathcal{Z}|$ units at our disposal this static symmetry breaking method can cover all sensors and zones.

(Dynamically) We can do dynamic symmetry breaking via a precedence ordering: For some orderings of (1) the units and of (2) the vertices from $\mathcal{S} \cup \mathcal{Z}$, whenever v_i is assigned to U_j, enforce that $v_k, k < i$ is assigned to unit $U_l, l \leq j$.

Connecting Units to Units Instead of breaking the symmetry on (sensor-or-zone)-to-unit-connections we can also do it on unit-to-unit-connections.

(Statically) Static symmetry breaking analogous to the idea proposed for sensor/zone-to-unit assignments results in very weak symmetry breaking.

(Dynamically) If we use a problem model that might contain more units than actually needed we can disallow "gaps": For some ordering of the units U_1 to U_n require that all $U_i, i < m$ are non-empty if U_m is non-empty [1]. This can safely be combined with the static symmetry breaking on sensor/zone-to-unit-connections.

Dynamic symmetry breaking can also be achieved by a stair function of units that are tabu as partners: After the first unit not connected to U_1 no later unit can be connected to U_1; after the first later unit not connected to U_2 unit U_2 is disallowed; etc.

In many models a $\mathit{NoOfUnits} \times \mathit{NoOfUnits}$ matrix of Boolean variables represents the inter-unit connections (every unit is connected to itself). Here we can require that Row_i is lexicographically greater than or equal to Row_{i+1} for $i = 1..\mathit{NoOfUnits} - 1$.

These last two proposals cannot safely be combined with static symmetry breaking on sensor/zone-to-unit-connections.

4 Encodings

In this section we study the strengths and weaknesses of contemporary solving methodologies on the Partner Units Problem. In particular we look at

- CDCL-solving on propositional problem representations; and
- constraint propagation in combination with programmable search.

We include lazy clause generation in constraint propagation, although technically it is a combination of the above two approaches.

As discussed in Section 3.1 instances are typically either solvable using lower bound many units or unsolvable. Hence, even though we are interested in optimal solutions, the PUP is more of a search than an optimisation problem. This is one of the reasons why integer programming is not among the approaches considered.

In the sequel, we discuss how the different frameworks offer different search algorithms for the PUP: Which variables are selected for branching, whether there is a variable/value ordering and which choices lead to the propagation of new information. Then most constraints in the PUP are about counting up to some fixed threshold ($\mathit{UnitCap}$, $\mathit{InterUnitCap}$). So we look at the overarching question of how the counting is done in the different frameworks.

4.1 The Instances

We have evaluated our encodings on a set of benchmark instances that we received from our partners in industry, including the instances used in [1]. Let us describe in some detail the structure of our instances. In all of our instances $\mathit{UnitCap} = 2$ and there are more sensors than zones. All instances are built atop an underlying grid of rooms.[3]

The double-family of instances for $\mathit{InterUnitCap} = 2$ is built from two rows of rooms stacked on top of each other; cf. Figure 1. Each room is a zone and is connected to all sensors on its boundary; there are sensors on all doors. The length of the rows increases with problem size. In the doublev-* instances the columns are zones, too; in the doublehv-* instances there are overlapping zones of size three (cf. Figure 1).

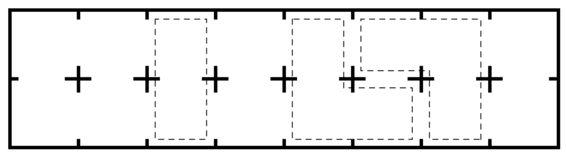

Fig. 1. The basic layout of the various double-* instances

The n4triple-* instances for $\mathit{InterUnitCap} = 4$ are built from blocks depicted in Figure 2: A grid-like layout with exits to the north and south at the corners. The northern- and southernmost rows are connected, whereas in the interior of the grid most east-west connections are blocked. The actual instances consist of multiple such blocks stacked on top of each other. There is a sensor on every door and each room is a zone. There are larger zones consisting of two to six adjacent rooms, growing from the boundary of the grid. In the n4triplemjvd-* instances the overlapping zones are of size eight to eleven. Again, each zone is connected to all sensors that lie on its boundary.

[3] The benchmarks and all encodings will be made publicly available.

The triple-* instances from [1] are like the n4triple-* instances, only that the interior walls have no holes and there are fewer overlapping zones. Smaller triple-* instances with few overlapping zones are solvable for *InterUnitCap* $= 2$ as they do not violate the cycle property described in Section 2.2.

We use the hand-crafted n4triplemjvd-* and doublehv-* instances to test the MJVD property and the shared-neighbours condition from Proposition 1. Both properties do not appear in our other instances.

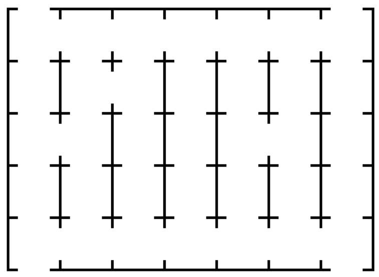

Fig. 2. The basic building blocks of the n4triple-* instances

Finally, concerning the different search strategies described in Section 3.1: All of our benchmarks are unsolvable or require only lower bound many units. Hence, if *InterUnitCap* > 2, the combined strategy (3) is always fastest: The search steps in (3) are identical to the first and last step of (2).

4.2 Implied Constraints

In Section 2 we have identified a cycle property for *InterUnitCap* $= 2$, a common neighbours property and the maximum joint vertex degree (MJVD) as implied constraints for pruning the search space.

The cycle property is difficult to automatically check for. However, we could use it to show that none of the n4triple-8 instances can be solved with *InterUnitCap* $= 2$ (cf. Section 4.1 above). The common neighbours property does not show in most of our benchmarks. Where it does show for all solvers the small gain is neutralised by the small overhead of checking the condition: If two nodes share one neighbour in the instance they have a maximum distance of two in the solution; if they share sufficiently many neighbours that distance is reduced by one. This leaves the MJVD as the sole implied constraint with some promise.

4.3 The Matrix of Inter-Unit-Connections

If no fixed cyclic layout of the units is assumed the inter-unit-connections have to be modelled as variables. This is done via a matrix of Boolean variables UU_{ij} corresponding to a connection from unit i to unit j, where we may assume a unit to be connected to itself. In all the encodings presented in [1] this matrix is symmetric, i.e. we have $UU_{ij} = UU_{ji}$. Here an obvious improvement is to fold the matrix along the diagonal.

4.4 Conflict Driven Clause Learning Solvers

We next look at conflict driven clause learning (CDCL) solvers on ground propositional problem representations. In particular we look at encodings of the PUP in answer set programming and SAT.

Modern CDCL solvers work on a propositional ground problem representation via clauses, i.e. disjunctions of (possibly negated) Boolean variables. For inference they use unit-propagation: If the truth value of a variable is known all occurrences of its complement are removed from all clauses [7]. Clause learning [27] incrementally adds clauses to the problem representation that are sufficient to explain a conflict discovered during search. Learnt clauses allow the solver to backtrack over multiple levels of the search tree in one step ("backjumping"). The technique of watched literals ensures that a clause is only selected for propagation once all but one literals are instantiated [24]. CDCL solver mostly differ with regard to the heuristics they employ for selecting variables, deletion of learnt clauses, branching, etc. These parameters can be improved for specific problem classes via automatic parameter tuning.

Answer set programming differs from plain SAT in that it

- offers a rich modelling language via logical rules that are automatically grounded; and
- performs minimal model reasoning only whereas SAT consider all classical models.

Propositional Satisfiability Testing A main challenge in developing propositional encodings of the PUP in plain SAT is the representation of *atmost* constraints in order to impose upper bounds (*UnitCap*, *InterUnitCap* and, possibly, MJVD thresholds). In the SAT encoding presented in [1] the constraint that each sensor/zone is assigned to at most one unit was encoded "naively": $\bigwedge \neg vU_i \vee \neg \mathrm{vU}_j$, where vU_i means sensor/zone v is on unit i. The other *atmost* constraints were encoded using Sinz' sequential counter [30]. While the number of clauses in this naive encoding is, in general, exponential in the upper bound k, Sinz' sequential counter provides a representation with $O(nk)$ clauses.[4] We now have four direct SAT encodings in total, varying in whether naive encoding or Sinz' sequential counter is used for atmost constraints and in whether inter-unit connections are represented symmetrically or not.

We have used these encodings for experiments with Stochastic Local Search (SLS) methods for solving the PUP; for this we have used the UBCSAT 1.1 [31] solver. However, contrary to CDCL-solvers, SLS algorithms could not find a single solution for these encodings within the given time-frame. These results are in line with chapter 2.3.4 of [5] where it is argued that SLS performs poorly on SAT problems with a lot of variable interaction.

It has been convenient to use ASP for encoding the advanced PUP conditions propositionally; CDCL-solvers on the above SAT encodings are evaluated below, together with these advanced ASP encodings.

[4] A survey on and evaluation of different SAT encodings of counting up to k can be found in [12].

Answer Set Programming Problems are represented in ASP as a combination of data, describing an instance in terms of ground facts, and a uniform first-order encoding, invariant under instances. Solutions are then computed in two phases, first, (deterministically) grounding the encoding w.r.t. data and, second, (non-deterministically) searching for particular models, so-called answer sets [16], of a propositional logic program obtained in the first phase. Below we use the grounder GRINGO [14] and the solver CLASP [15]. Automated grounding allows us to rapidly develop encoding variants, and CLASP performs CDCL-style search on the propositional level.

Our ASP encoding makes use of *atmost* constraints for enforcing the upper bounds given by $\mathit{UnitCap}$, $\mathit{InterUnitCap}$ and MJVD thresholds; traditionally these constraints are called "cardinality" constraints [29] in ASP. Native ASP solvers implement propagators for such constraints, which makes a compilation to simpler constructs such as SAT clauses unnecessary. In particular, CLASP applies a generalised form of unit propagation to such constraints and lazily extracts clauses upon conflict resolution [13].

In our ASP encoding of the PUP the following features are configurable via command-line switches of GRINGO:

MJVD Dynamic checks of this condition can be activated independently for cliques of size one (vertices) and two (edges) in the unit graph. To this end, predicates for identifying the zones (or sensors) connected to the sensors (or zones) in a single unit or a pair, respectively, are introduced in the grounding phase and evaluated w.r.t. partial assignments constructed during search. Moreover, dedicated counting constraints similar to those for checking $\mathit{UnitCap}$ and $\mathit{InterUnitCap}$, are posted in the grounding phase to enforce that the threshold from in Proposition 3 is not exceeded.

Symmetry Breaking Each of the approaches described in Section 3.3 can optionally be activated. For $\mathit{InterUnitCap} = 2$ the unit graph is fixed to a cycle by asserting the existence of respective edges (in the grounding phase). Restrictions on the assignment of sensors and zones to units rely on a variable order (cf. Section 3.2) provided as part of an instance. Static restrictions are applied in the grounding phase by limiting the range of units to which a sensor or zone can be assigned; the condition that there are no gaps between assigned units is dynamically checked during search. Furthermore, our encoding offers (dynamic) restrictions on inter-unit connections: Step-wise recognition and exclusion of tabu partner units has an easy implementation (just four lines in our encoding). Symmetry breaking by ordering whole the rows in the matrix of inter-unit-connections is stronger and guarantees a unique representation of isomorphic unit graphs. This can further be strengthened by requiring that the variables assigned to units with identical rows to be in order, and our encoding also offers this optional extension.

Empirical Evaluation Let us now turn to the empirical evaluation of our ASP encoding. Automated translations from ASP to SAT via LP2SAT2 [20] allow us to run SAT solvers on CNF formulas that are semantically equivalent (modulo auxiliary variables) to the grounded propositional ASP programs. As our programs do not involve positive recursion (implication cycles via positive preconditions and consequents) they can be translated to SAT via a linear translation [9], known as Clark's completion [6]. The main hurdle is the compilation of *atmost* constraints into clauses. LP2SAT2 offers four different compilations for this purpose: adder circuit, binary counter, counting grid, and Sinz' sequential counter [20].

In our experiments we use CLASP (version 2.0.4), LINGELING [4] (version 587f), and MINISAT [8] (version 2.2.0). Furthermore, we used GRINGO (version 3.0.3) for grounding and LP2SAT2 (version 1.15) for converting ground logic programs to CNF. Runtime results provided below exclude grounding and translation to SAT, which were accomplished offline.

LINGELING and MINISAT are plain SAT solvers whereas CLASP can be run both as an ASP solver (on GRINGO output) and as a SAT solver (on LP2SAT2 output). For CLASP, we switched from its default BerkMin-like decision heuristic (cf. [18]) to VSIDS (cf. [3]) which is also used by LINGELING and MINISAT; the latter heuristics works better on the PUP for $\mathit{InterUnitCap} > 2$ [1]. We switched off occasional random decisions in LINGELING to ensure reproducible behaviour.

For assessing encoding options we divide the instance families described in Section 4.1 into the following three groups: The polynomially solvable "N2", "N4Triple" instances, and finally "N4TriplesMJVD". We consider the following encodings:

- the two "legacy" encodings for $\mathit{InterUnitCap} = 2$ and $\mathit{InterUnitCap} > 2$ from [1];
- the "plain" new encoding without symmetry breaking and MJVD checks and its "ring" variant for $\mathit{InterUnitCap} = 2$; and
- SAT-encodings generated from "plain" via LP2SAT2.

For "plain" we consider variants checking MJVD for cliques of size one and two and the different combinations of static and dynamic symmetry breaking.

Assuming a fixed cyclic unit graph as in "legacy" and "ring" proved far more effective on "N2" than any other method. "Ring" improves on "legacy" in that it does not include both directions for the edges between units.

Checking MJVD for single units introduced only little overhead for "N4Triples" and "N4TriplesMJVD"; it only really helped on the latter instances. Checking the condition for pairs of units had detrimental effects.

Static symmetry breaking always helped. Of the various variable orderings that could be used for this `maxadjdeg` and `maxdegadj` performed best. We observe that disallowing "gaps" in the sequence of units throughout search hurt performance on "N4Triples" whereas it helped on "N4TriplesMJVD".

Using MJVD, a folded inter-unit-connection matrix and appropriate symmetry breaking the new ASP encoding can solve one-and-a-half as many instances as "legacy" in a quarter of the time.

We have then tested plain SAT encodings obtained via LP2SAT2 from

- "ring" for "N2";
- MJVD on single units plus `maxdegadj` for "N4Triples"; and
- the same plus disallowing gaps for "N4TriplesMJVD".

We have run CLASP on these encodings where it internally compiles *atmost* constraints to Sinz' sequential counter. This also was clearly the best of the various encodings of *atmost* constraints available in LP2SAT2. On "N2" the differences between the different solvers were not very significant, with LINGELING having slightly the edge. On "N4TriplesMJVD" CLASP was the clear winner, either run on the output of LP2SAT2 or internally compiling the *atmost* constraints; on "N4Triples" CLASP was again the clear winner, but only on the output of LP2SAT2.

On "N4Triples" CLASP using a propagator instead of Sinz' sequential counter was about four times faster; on "N4TriplesMJVD" the effect was less pronounced, but CLASP was still in front.

Our attempts to further improve CLASP via PARAMILS [19], i.e. parameter tuning via local search on CLASP's configuration space, did not bear any fruit. Overall, CLASP in its default configuration using VSIDS instead of Berk-Min using the "plain" encoding extended by MJVD and suitable symmetry breaking is our recommended approach to solving PUP instances where $\mathit{InterUnitCap} > 2$.

We have also run CLASP, MINISAT and LINGELING on a relatively small set of instances using the different crafted SAT encodings described above: Interestingly, here using a non-folded, symmetric matrix helped all solvers. This contrasts starkly with ASP where, due to minimal-model reasoning, a symmetric matrix incurs a relatively expensive well-foundedness check.

In [28] it has been shown that a CDCL SAT solver can become faster by ignoring the auxiliary variables from Sinz' encoding of "at most one" constraints; we have also tested their modified version of MINISAT (version 2.0). However, the auxiliary variables from "atmost one" constraints constitute only a very small part of a SAT-encoded PUP instance and our results were inconclusive. We have also considered crafted instances with $\mathit{InterUnitCap} = 2$ and $\mathit{UnitCap} = 1$ — i.e. all counting is up to one. These instances, however, are too easy to solve for meaningful conclusions.

Search Strategies For $\mathit{InterUnitCap} = 2$ the search strategy () from Section 3.1 worked best: Assuming a fixed cyclic unit graph (does not) provide enough additional information to outperform an encoding using general symmetry breaking methods. As all our instances are either unsolvable or solvable using lower bound many units the support available in CLASP for optimisation does not help.

4.5 Constraint Programming

In this section we first look at counting constraints in CP. We then discuss our various basic models for the PUP. We have modelled the problem for the solver MINION [17] and in the MINIZINC modelling language [25], in order to use the FDX lazy clause generation solver [11]. We still consider the basic model from [1], written in ECL^{i}PS^e [32].

In Constraint Programming (CP) [26] a problem is represented via variables with discrete finite domains. The constraints specify which combinations of values can be used together. Compared with the explicit ground propositional representation employed by CDCL solvers a constraint solver maintains an implicit problem representation. If the domain of a variable shrinks during search then the constraint solver tries to *propagate* changes to the domains of other variables occurring in the same constraint. The propagation algorithm used depends on the semantics of the constraint; often there are multiple competing propagation algorithms for the same constraint. The order in which variables and values are selected for instantiation is usually programmed by hand. These orderings are well known to have a drastic impact on the time needed to find a solution.

MINION is a modern pure CP solver aiming at raw propagation speed and incorporating suitably adapted watched-literal techniques from SAT solvers. ECL^{i}PS^e is an older Constraint Logic Programming (CLP) system, i.e. the constraint solving is embedded in a Turing-complete logic programming environment. MINIZINC is a high-level problem specification language that is automatically transformed into low-level input for a number of CP solvers. In FDX a CDCL SAT solver is embedded into a classical constraint solver. The CP solver translates explanations for the propagations it has made into SAT clauses and incrementally (lazily) adds these clauses to the SAT representation of the problem. It then exploits clause learning and backjumping available in the SAT solver to prune the search space it considers.

Classical Constraint Programming We first consider classical constraint programming in ECL^{i}PS^e and MINION, i.e. without the no-good learning capabilities of lazy clause generation.

Counting in Constraint Programming There are three natural candidates for modelling the counting needed for the PUP in CP:

- An $\mathit{atmost}(n, \mathit{vars}, \mathit{val})$ constraint enforces that at most *n* variables take the fixed value *val*, where *n* is fixed.
- An $\mathit{occurrences}(\mathit{val}, \mathit{vars}, \mathit{var})$ constraint enforces that the number of occurrences of the fixed value *val* in *vars* is in the domain of the variable *var*.
- A global cardinality constraint $\mathit{gcc}(\mathit{vars}, \mathit{vals}, \mathit{cvars})$, where *vals* and *cvars* are vectors of the same length, enforces that val_i occurs cvar_i times in *vars*. Here val_i is fixed, but cvar_i may be a domain variable.

Basic Model for $\mathit{InterUnitCap} = 2$ We ported our ECL^{i}PS^e model directly to MINION. The units form a ring $(1, 2, \ldots, \mathit{NoOfUnits}, 1)$, and we use iterative deepening on their number. For each sensor and zone we introduce a variable with domain $1..n$. We post constraints $\mathit{atmost}(m, \mathcal{S}, i)$, for $m = \mathit{UnitCap}$ and all $1 \leq i \leq n$, and repeat the same for zones. For each connection (s, z) we introduce a variable *sz* with domain $\{-(\mathit{NoOfUnits} - 1), -1, 0, 1, (\mathit{NoOfUnits} - 1)\}$ and post a constraint $s - z = sz$. Using the same symmetry breaking and variable ordering as in ECL^{i}PS^e MINION is two to three orders of magnitude faster.

Basic Model for $\mathit{InterUnitCap} > 2$ The modelling of sensors and zones as unit-valued variables and the *UnitCap*-constraints are as before. Using a single *gcc* constraint in MINION proved superior to multiple *atmost* constraints — *gcc* is not available in ECL^{i}PS^e. The *InterUnitCap* constraints are posted as *atmost* constraints on the matrix of Boolean variables representing the inter-unit-connections. Folding the matrix did not have significant effects in ECL^{i}PS^e whereas in MINION a non-folded matrix is two to three times slower.

The following is an interesting difference between the two CP models with regard to the handling of connections between units. In ECL^{i}PS^e we have written a custom propagator that behaves as follows:

– If either s or z is assigned to U_i (say s is) then for every value v in the current domain of z during search we add the hand-coded constraint $UU_{iv} = 0 \Rightarrow z \neq v$.
– This constraint $UU_{iv} = 0 \Rightarrow z \neq v$ is checked only when UU_{iv} is instantiated — changes to the domain of z cannot result in new information.
– If both s and z are assigned the respective Boolean is set.

In MINION we have tried the following two encodings:

(1) Post constraints $((s-1) \times \mathit{NoOfUnits}) + z = i$ and $\mathit{element}(\mathit{matrix}, i, 1)$.
(2) Write the map from units to matrix indices as a tuple list $\mathit{sztuple} = (\mathit{Unit}_k, \mathit{Unit}_l, i)$, all k, l, and then post constraints $\mathit{table}([s, z, i], \mathit{sztuple})$ and $\mathit{element}(\mathit{matrix}, i, 1)$.

Approach (2) avoids the expensive propagation associated with the arithmetic in approach (1). Contrary to ECL^{i}PS^e in both approaches shrinking a variable domain without instantiating it also leads to a propagator being woken. Due to the more economical model used ECL^{i}PS^e has the edge over MINION.

Symmetry Breaking Let us now look at symmetry breaking:

$\mathit{InterUnitCap} = 2$ Here we fix one sensor to be on the first unit and some other sensor to be on the first half of the cycle.

$\mathit{InterUnitCap} > 2$ The lexicographic symmetry breaking described in Section 3.3 is straightforward in the CSP model, given that the connections between units are specified by a (folded-up version of) a $\mathit{NoOfUnits} \times \mathit{NoOfUnits}$ matrix of Boolean variables. Advanced methods for symmetry breaking on Boolean matrices are described in [21] and [34]. With the basic lexicographic method we can solve e.g. `triple-34` that cannot be solved without symmetry breaking. However, there is a lot of negative interaction with the search strategy on instances that are otherwise easily solvable.

If we instead restrict the units that sensors and zones can be assigned to no such negative interaction can occur. Static restrictions are straightforward to implement. Strong dynamic symmetry breaking on the assignment of sensors and zones to units could be achieved by the global constraints described in [23] — however, these constraints are not available in MINION and ECL^{i}PS^e. We used multiple $\leq$ constraints between pairs of variables instead which results in weaker propagation. The strength of the symmetry breaking strongly depends on the chosen variable orderings. Dynamic symmetry breaking hurt performance whether used as a stand-alone or in combination with static symmetry breaking, whereas static symmetry breaking always helped.

Variable Orderings Let us now look at the merits of the various proposed variable ordering heuristics for CP.

$\mathit{InterUnitCap} = 2$**:** As the `adjbfs` variable ordering yielded good results for CP, one might have expected even better results using more refined `adj` orderings. This, however, was not the case: With `adjbfs` MINION could solve a double-3000 in 85 seconds, whereas most other `adj` orderings reported a time-out on double-100. Of the different variable orderings from Section 3.2 `adjbfs` worked by far the best.

This can be explained as follows: Call a variable ordering *non-decomposing* if for no sub-sequence of the ordering the instance graph minus the sub-sequence decomposes into multiple connected components. Call it *almost-non-decomposing* if for every subsequence all but one of the remaining components contain only a single vertex — consider e.g. an overlapping zone that is preceded in the ordering by all of its sensors.

On our instances the `adjbfs` ordering always starts from a corner of the underlying grid (that is where sensor one is). If the instance contains overlapping zones it is almost-non-decomposing, and non-decomposing otherwise. For various reasons the other orderings frequently decompose the instance graph into several large components, interleaving vertices from different remaining components in the ordering. Now the standard increasing value ordering frequently assigns vertices from different remaining components to the same units. This in turn leads to frequent failures because of too many neighbouring sensors and zones. Whether `adjbfs` is non-decomposing crucially depends on the first node chosen — after permuting the vertex names in the instance graph MINION with `adjbfs` will time-out on double-40.

Note that on general instance graphs almost-non-decomposing orderings do not always exist. Our bipartite graphs are obtained from grid-like instances where overlapping zones are contiguous. Here walking the columns (or rows) and adding overlapping zones when all sensors have been covered yields an almost-non-decomposing ordering. Empirically row- or column-wise orderings perform as well as `adjbfs`.

$\mathit{InterUnitCap} > 2$**:** Here results were less conclusive. Assigning the inter-unit-connections before the sensors and zones is hopeless; doing it the other way around is quite good already. In our experiments we used the same variable ordering for both search and symmetry breaking. `maxdeg`, `mindeg`, `zonessensors`,[5] and `random` were hopeless; `maxdegminadj` did better but still bad. Instance-specific (almost-)non-decomposing variable orderings did not do particularly well. With the exception of `sensorszones` all good variable orderings follow the topology of the instance graph.

The best ordering we found was `maxadjdeg` with inter-unit-connection variables interspersed every n steps, where n is a little less than $2 * \mathit{UnitCap}$. Setting inter-unit variables to "false" before "true" worked slightly better. We have not been able to identify a strikingly superior variable ordering as in the case of $\mathit{InterUnitCap} = 2$.

Search Strategies We have evaluated the search strategies described in Section 3.1 for $\mathit{InterUnitCap} = 2$: Pure iterative deepening (2) is fastest even on unsolvable instances.

Maximum Joint Vertex Degree Assume we use iterative deepening search. Now for $\mathit{InterUnitCap} = 2$ and a connection (s, z) with s assigned we know the three possible positions z can take. Hence in MINION we can identify the MJVD for single units by posting single *gcc* constraints on the vectors of sensors and zones, respectively, instead of multiple *atmost* constraints. On most instances MJVD does not show and runtimes double because of the more expensive propagation.

Next consider the case of $\mathit{InterUnitCap} > 2$. In the pure CSP framework of MINION we introduce for every unit U Booleans U_s for every $s \in \mathcal{S}$ and U_z for $\mathcal{Z}$ respectively. These vectors represent the sensors/zones that have to be placed on a unit or its neighbours. Assume $\mathcal{Z}_s$ is the set of neighbours of a sensor s. We post reified constraints $\mathit{reify}(s = U, S_U)$ and $\mathit{reifyimply}(\mathit{watched\text{-}and}(\{U_z = 1\}), S_U)$ for all $z \in \mathcal{Z}_s$, and

[5] We assume $|\mathcal{S}| > |\mathcal{Z}|$.

repeat for every pair $z, \mathcal{S}_z$. Now S_U is set to "true" if and only if sensor s is on unit U which in turn sets all the U_z needed.

In the CLP framework offered by ECL^{i}PS^e we create for each sensor/zone a hash containing all of its neighbours. Next we maintain throughout search for every unit two lists of sensors and zones that have to be assigned to this unit or one of its neighbours. We add elements to this list whenever we assign a sensor/zone and check that the lists do not grow too long.

On instances that exhibit the MJVD both approaches pay. On the other instances the overhead incurred doubles the runtimes in MINION but is negligible in ECL^{i}PS^e.

Lazy Clause Generation We know from our experiments with CDCL solvers that clause learning and backjumping work well for the PUP if *InterUnitCap* > 2. Hence we were curious to see what a LCG solver could achieve for a CP model. MINIZINC translates the matrix-index in a constraint like

$$\mathit{Zones}[i] = u_1 \wedge \mathit{Sensors}[j] = u_2 \rightarrow \mathit{InterUnits}[u_1, u_2] = 1$$

into a linear arithmetic expression on $\mathit{Zones}[i], \mathit{Sensors}[j], u_1$ and u_2 analogous to one of our MINION models. We regard this handling of the inter-unit-connections as a likely explanation for why FDX cannot keep up with MINION or ECL^{i}PS^e.

4.6 Cross-Evaluation

Let us briefly compare the different encoding frameworks. All experiments data were done on the same 3 GHz dual core machine with 4 GB RAM running Fedora Linux used in [1] using a ten minute time limit. For *InterUnitCap* $= 2$ CDCL-solvers don't scale, whereas CP solvers do: E.g. CLASP solves a double-80 in $50s$ and times out on double-100, whereas MINION solves a double-3000 in $80s$. For *InterUnitCap* $= 2$ the situation is reversed: Most instances from "N4Triples" and "N4TriplesMJVD" are beyond the reach of CP. In fact, we had to use much smaller instances from these families for the above evaluation.

5 Conclusions

In this work we have contrasted classical constraint programming with CDCL-solvers on the PUP. We have seen that native support for *atmost* constraints is hugely beneficial for CDCL-solvers on the PUP. Likewise, in CP user-defined control over constraint propagation proved to have a big impact on performance. Observe that unit propagation-based CDCL-solvers don't have this issue.

For the case of *InterUnitCap* $= 2$ we have seen that a good programmed search strategy easily outperforms the search heuristics employed by CDCL-solvers. On the other hand, for *InterUnitCap* > 2, we have not been able to find a CP search strategy that matches CDCL-solvers. It would be interesting to see whether our best CP model ported to a lazy clause generation solver (i.e. adding learning) closes this gap.

Thanks We wish to thank Chris Jefferson, João Marques-Silva and Andreas Schutt for hints on MINION, the modified MINISAT and MINIZINC/FDX, respectively.

References

1. Aschinger, M., Drescher, C., Gottlob, G., Friedrich, G., Jeavons, P., Ryabokon, A., Thorstensen, E.: Optimization Methods for the Partner Units Problem. In: Proceedings of CPAIOR'11 (2011)
2. Aschinger, M., Drescher, C., Gottlob, G., Jeavons, P., Thorstensen, E.: Tackling the Partner Units Configuration Problem. In: Proceedings of IJCAI'11 (2011)
3. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Kleine Büning, H., Zhao, X. (eds.) Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08). Lecture Notes in Computer Science, vol. 4996, pp. 28–33. Springer-Verlag (2008)
4. Biere, A.: Lingeling and friends at the SAT competition 2011. Technical Report FMV 11/1, Institute for Formal Models and Verification, Johannes Kepler University (2011)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press (2009)
6. Clark, K.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press (1978)
7. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3) (1960)
8. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03). Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer-Verlag (2004)
9. Fages, F.: Consistency of Clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science 1, 51–60 (1994)
10. Falkner, A., Haselböck, A., Schenner, G.: Modeling Technical Product Configuration Problems. In: Proceedings of the Configuration Workshop at ECAI'10 (2010)
11. Feydy, T., Stuckey, P.J.: Lazy clause generation reengineered. In: Principles and Practice of Constraint Programming - CP 2009. Lecture Notes in Computer Science, vol. 5732. Springer (2009)
12. Frisch, A., Giannoros, P.: SAT Encodings of the At-Most-k Constraint. In: Proceedings of MODREF'10 (2010)
13. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: On the implementation of weight constraint rules in conflict-driven ASP solvers. In: Hill, P., Warren, D. (eds.) Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09). Lecture Notes in Computer Science, vol. 5649, pp. 250–264. Springer-Verlag (2009)
14. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In: Delgrande, J., Faber, W. (eds.) Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11). Lecture Notes in Artificial Intelligence, vol. 6645, pp. 345–351. Springer-Verlag (2011)
15. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Veloso, M. (ed.) Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). pp. 386–392. AAAI Press/MIT Press (2007)
16. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
17. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: Proceedings of ECAI'06 (2006)
18. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT solver. In: Proceedings of the Fifth Conference on Design, Automation and Test in Europe (DATE'02). pp. 142–149. IEEE Press (2002)

19. Hutter, F., Hoos, H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267–306 (2009)
20. Janhunen, T., Niemel, I.: Compact translations of non-disjunctive answer set programs to propositional clauses. In: Balduccini, M., Son, T. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning, Lecture Notes in Computer Science, vol. 6565, pp. 111–130. Springer Berlin / Heidelberg (2011)
21. Katsirelos, G., Narodytska, N., Walsh, T.: On the complexity and completeness of static constraints for breaking row and column symmetry. In: Proceedings of CP'10 (2010)
22. Koiliaris, K.: Complexity in Constraint Satisfaction and Automated Configuration. Master's thesis, University of Oxford, UK (2011)
23. Law, Y.C., Lee, J.H.M.: Global constraints for integer and set value precedence. In: Proceedings of CP'04 (2004)
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of DAC'01 (2001)
25. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Principles and Practice of Constraint Programming - CP 2007. Lecture Notes in Computer Science, vol. 4741. Springer (2007)
26. Rossi, F., van Beek, P., Walsh, T. (eds.): The Handbook of Constraint Programming. Elsevier (2006)
27. Silva, J.P.M., Sakallah, K.A.: GRASP: A new search algorithm for satisfiability. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design. pp. 220–227. IEEE Computer Society Press (1996)
28. Silva, J.P.M., Lynce, I.: Towards robust CNF encodings of cardinality constraints. In: Proceedings of CP'07 (2007)
29. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
30. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: Proceedings of CP 2005. Springer (2005)
31. Tompkins, D.A.D., Hoos, H.H.: Ubcsat: An implementation and experimentation environment for sls algorithms for sat & max-sat. In: SAT (2004)
32. Wallace, M., Schimpf, J., Novello, S.: ECLiPSe: a platform for constraint logic programming. ICL systems journal 12, 159–200 (1997)
33. Wallace, R.J.: Determining the principles underlying performance variation in CSP heuristics. International Journal on Artificial Intelligence Tools 17(5), 857–880 (2008)
34. Yip, J., Hentenryck, P.V.: Symmetry breaking via lexleader feasibility checkers. In: Proceedings of IJCAI'11 (2011)