On Solving Complex Rack Configuration Problems using CSP Methods

Wolfgang Mayer and **Marc Bettex** and **Markus Stumptner**
University of South Australia
Advanced Computing Research Centre
Adelaide, Australia
Firstname.Lastname@unisa.edu.au

Andreas Falkner
Siemens AG Österreich
Vienna, Austria
Andreas.A.Falkner@siemens.com

Abstract

Constraint Satisfaction Techniques have been widely applied to the configuration of complex systems in various domain. While much progress has been achieved on algorithms and theoretical characterisations of special cases that can be solved efficiently, many of these results make strong assumptions. In this paper we investigate different modelling and search techniques to assess their suitability for on-line configuration of a complex prototypical configuration problem. We found that CSP solving techniques perform poorly on our problem, while local search and repair techniques may achieve good performance while permitting simpler constraint formalisms.

1 Introduction

Assembly and configuration of larger systems from individual modular parts has been a central topic of interest, leading to a number of different technical solutions to address this problem in a variety of domains (Stumptner and Soininen; 2003). Constraint satisfaction based search techniques have been particularly successful in solving this task in certain domains, such as the telecommunications sector (Fleischanderl et al.; 1998), where entire switching systems are assembled from racks and modules.

Similar techniques have recently been employed to address other domains. For example, the design and planning of railway systems and the underlying infrastructure also requires that the results adhere to legal requirements and technical constraints. Similarly, the composition and configuration of distributed service processes can be phrased as a configuration problem (Thiagarajan et al.; 2009).

Typically, modular ("object-oriented") knowledge representation techniques are applied to capture the relevant properties of components and their related constraints, which may then be used to check configurations for compliance and extend partial configurations with additional components (Fleischanderl et al.; 1998). Modular knowledge representation is convenient for knowledge acquisition and is complemented with a custom-built constraint solving implementation that may be used to reason about configurations (Fleischanderl et al.; 1998).

While custom constraint frameworks are easily implemented to check a (partial) configuration for consistency with given constraints, completing a partial configuration ("*solving*") or altering a configuration to repair constraint violations is a challenging problem. Since some of the newer application areas have significantly more complex constraints than earlier applications, traditional solving approaches that are based only on backtracking or back jumping algorithms are no longer effective and may lead to inefficient exploration of the space of possible configurations. Instead, advanced Constraint Satisfaction techniques would be desired that can handle complex arithmetic expressions, relations, graphs, and other approximations. In the configuration context, a combination of advanced domain reduction and search procedures allows a solver to detect infeasible configurations early and focus search. For example, using integer linear programming techniques to better estimate the number of required components may allow to prune the search space more efficiently. Similar combinations of multiple CSP techniques has already led to significant results in other domains (Van Hentenryck and Michel; 2005).

While advanced CSP technology has great potential to improve configuration efficiency, it remains challenging to implement the necessary algorithms within the custom configuration system. Rather, ways to exploit existing implementations, for example the ECLiPSe suite (Apt and Wallace; 2007), would be desired. Hence, methods to express selected aspects of the configuration task as standard CSP problem have been developed (Kızıltan and Hnich; 2001; Narodytska and Walsh; 2007). In particular, the elimination of equivalent solutions due to symmetries —in our context typically caused by interchangeable components or permutations of components— has been thoroughly researched in the CSP community.

However, most representations of configuration problems address only a small subset of the entire configuration do-

main and must be extended to cover complex configuration problems. Moreover, existing symmetry reduction techniques typically require strong assumptions that may not hold for configuration problems in practice.

In this paper, we investigate different ways to encode a prototypical configuration problem as a standard CSP to assess which techniques are suitable for semi-interactive configuration. Our sample problem revolves around a toy example that is derived from a real-world configuration problem and includes complex constraints that are challenging to handle for most constraint solvers. Complex global constraints together and global cost function that should be optimised render this problem interesting from both a theoretical and a practical perspective.

Our results indicate that constraint technology is no silver bullet and must be applied carefully. Unsurprisingly, standard CSP heuristics alone are often insufficient to derive useful solutions and must be complemented with domain-specific heuristics. More surprisingly, symmetry reduction techniques that have shown significant improvements on other benchmarks exhibit poor performance when applied to our configuration domain. In contrast to our expectations, advanced symmetry reduction strategies designed to focus search lead to performance that is significantly worse than solvers not exploiting this information. We also report that traditional backtracking solvers scale poorly on any encoding, while incremental refinement-based solvers can quickly arrive at solutions that are close (within 25%) to the (estimated) optimum. Hence, we advocate the use of heuristics combined with iterative repair-based search procedures to attack semi-interactive configuration where configurations need not be optimal but must be delivered timely. We also show that replacing complete search procedures with incomplete heuristics does not necessarily sacrifice the quality of the resulting configurations.

The remaining paper is organised as follows: in Section 2 we introduce the problem used throughout our case study. In Section 3 we evaluate matrix-based translations of our problem, followed by a tailored CSP representation that more directly encodes the modular KB constraints in Section 4. We then discuss local search procedures in Section 5 before concluding the paper by summarising our findings and avenues for further work in Section 6.

2 Showcase Problem

Our investigations are based on the "Showcase House" configuration problem where a given list of items ("Things") must be allocated to Cabinets in different Rooms while respecting certain constraints and preference criteria (Falkner and Schreiner; 2008). While the problem seems trivial at first, it is interesting because it incorporates constraints and properties that can also be found in more technical domains like design and assembly of complex systems and software processes.

A conceptual model of the problem is shown in Figure 1. In this domain, a house (of given size) contains a number of Rooms where each can accommodate multiple Persons and Cabinets. Cabinets are used to store "Things" and have a fixed capacity. Cabinets may be either low or high, where low Cabinets may be stacked on other (low) Cabinets. Each

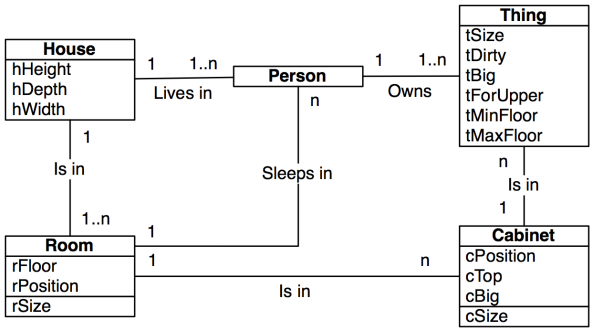


Figure 1: House Configuration Model

Thing is characterised by its size, whether it is clean or dirty, and whether it can be stored in an upper Cabinet. Cabinets may contain only clean or only dirty Things to ensure dirty Things cannot soil clean ones, and lower Cabinets cannot accommodate big Things. Furthermore, restrictions on the floor level where a thing can be stored apply. Each thing belongs to a Person. Since floor space and Cabinets are expensive, Things should be allocated to as few Cabinets as possible, while respecting the allocation constraints.

In addition to the mandatory constraints, a good solution not only minimises the number of Cabinets used, but also adheres to certain preference criteria. In particular, a thing should be stored in the Room of its owner, and, should this be impossible, in a vacant Room. The option where a Thing is stored in another Person's Room is least preferred, since this is likely to cause many disruptions.

The requirements and preferences can be expressed formally in a CSP framework. For example, assume a model where each entity is uniquely identified by some identifier and where properties of entities as well as relations between entities are represented as functions. Then, the constraint that a Cabinet may contain either only clean or only dirty Things can be expressed as a logical sentence as follows (Bettex; 2009):

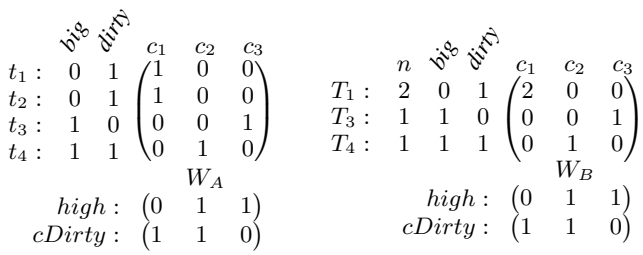
$$\forall t_1, t_2 \in T : t_1 \neq t_2 \wedge dirty(t_1) \neq dirty(t_2) \Rightarrow cabinet(t_1) \neq cabinet(t_2)$$

The remaining requirements can be formalised similarly. While this logical representation can be derived directly from the problem statement, it may may not be best-suited to the configuration task. Since the logical model can only express "hard" constraints, it is difficult (or at best inefficient) to handle the preference criteria. Here, (soft) CSP satisfaction methods may be more suitable.

In the following sections we investigate different constraint-based modelling approaches that have been proposed for similar configuration tasks to assess which approaches are suitable to capture complex domain constraints like the above.

3 Configuration as Matrix CSP

A variety of constraint satisfaction techniques have been proposed to address the configuration problem of modular electronic systems, where modules of different types must be consistently assembled into a (the smallest) set of racks of different kinds. In particular, Kızıltan and Hnich (2001) compare



(a) Model A (b) Model B

Figure 2: Two Matrix Models

and combine different complementary models in order to focus the search for solutions. Racks and modules correspond to Cabinets and Things in our problem. In this section, we adapt two of their models to our problem domain and summarise our empirical results.

We consider a simplified variant of the configuration problem, where the number of Cabinets are minimised but the preference constraints related to Persons and associated Rooms are left aside. This simplification may lead to solutions that are not optimal in the full problem. We refrain from extending the matrix CSPs to the more detailed representation since our experiments showed that even the simplified version is computationally intractable. The poor performance of the matrix CSP for the simplified representation discouraged us from attempting to formalise the problem in full; this would require a multi-dimensional matrix, which, we conjecture, would prohibit the solving efficiency required for (semi-)interactive configuration of large problem instances. The full problem is addressed in Section 4.

3.1 Model

Matrix-like CSP models are a special class of CSP models that have been proposed to model relationships between entities (Kızıltan and Hnich; 2001). In matrix models, the properties of entities are typically expressed as variables indexed by entity type (or even entity instance), while a matrix $W = (w_{ij})$ represents the associations between entities. In this framework, a given configuration problem is translated into a matrix CSP problem and is subsequently solved using a standard CSP solver; the solution of the CSP problem determines the configuration.

Kızıltan and Hnich (2001) discuss two different modelling approaches:

(A) Rows and columns of W represent single entity *instances*, where the elements of W range over $\{0, 1\}$ to indicate a relationships between entity instances i and j.

We adapt this models for our house configuration problem as follows; a similar representation was also described in Falkner (2009):

Each row $w_{.j}$ of W represents a single Thing, whose properties are represented as constants. Figure 2a shows an example, where four things ($t_{1...4}$) are placed in three cabinets ($c_{1...3}$). The properties of concrete things are represented as constant vectors. For our case study, the properties big and $dirty$ are modelled.

Each column $w_{i.}$ of W represents a unique Cabinet, where $w_{ij} = 1$ iff t_i is placed in c_j. A Cabinet's properties are modelled as constraint variables, whose values are constrained by the $w_{i.}$. We use variable $high_j$ to indicate that a Cabinet contains big Things, and $cDirty_j$ to reflect that at least one dirty Thing is placed in c_j. Constraints ensure that each Thing is assigned to one Cabinet

$$\sum_j w_{ij} = 1$$

and that Things with incompatible properties (clean/dirty) are assigned to different Cabinets (columns):

$$cDirty_j = \max_i(w_{ij} dirty_i),$$
$$cDirty_j \neq dirty_i \Rightarrow w_{ij} = 0$$

Other constraints are modelled similarly. Preferences are expressed as minimisation criterion, where the number of Cabinets (columns) where at least one thing is placed is minimal:

$$\arg\min_W \sum_j sign(\sum_i w_{ij}).$$

(B) Rows of W represent entity *types*, where each element $w_{ij} \in \mathbb{N}$ denotes the *number* of entities (modules, Things) i that are related to an instance of the entity (racks, Cabinets) represented by column j.

For example, the first row of matrix W_B shown in Figure 2b represents two indistinguishable things of type T_1. Each row $w_{i.}$ of W represents a set of n_i Things that share the same properties. In contrast to model A, only one row is required to represent multiple instances. The representation of columns (Cabinets) remains the same as in the previous model. The constraint that each Thing must be allocated to some Cabinet, while not exceeding the Cabinet's capacity, is captured as follows:

$$\sum_j w_{ij} = n_i, \quad \sum_i w_{ij} \leq size_j$$

Since each row represents potentially many Things, the constraint model is likely to be smaller than the previous one. The compact representation may also allow to solve more efficiently, since equivalent solutions where rows corresponding to equivalent Things are interchanged need not be explored separately.

3.2 Symmetry Reduction

While the models can be solved using standard finite-domain constraint solvers, both suffer from inefficient search caused by different representations of semantically equivalent variable assignments. This "symmetry" problem is well-known in the CSP literature and a number of techniques have been developed to reduce this effect (Frisch et al.; 2003). Typically, symmetries are eliminated from the search by imposing additional constraints that enforce an ordering on the variable assignments to eliminate equivalent solutions.

Both models introduced in the previous section suffer from many redundancies, stemming from indistinguishable Things and Cabinets. Following Kızıltan and Hnich (2001), we impose restrictions on the Cabinet's assignments such that (in this order) (i) high Cabinets must occupy columns with smaller indexes than low Cabinets, (ii) "dirty" Cabinets must come before "clean" ones, and (iii) adjacent Cabinets of the same type are ordered with increasing free space. For example, the latter constraint is expressed as follows:

$$high_j = high_{j+1} \wedge cDirty_j = cDirty_{j+1} \Rightarrow \sum_i size_i w_{ij} \geq \sum_i size_i w_{i,j+1}$$

In addition, one could enforce a lexicographic ordering on the columns of W (Frisch et al.; 2003). Since this additional constraint slowed search considerably (see next section), we decided to not use it in our experiments. Kızıltan and Hnich (2001) argue that neither model is strictly better than the other and show that a combined approach can significantly improve symmetry elimination (although not all symmetries could be eliminated). Their best model backtracks only $\approx 5K$ times, down from $164M$ for the original Model A. However, their improved model does not directly apply to our problem domain, since different requirements on entity types and their constraints prohibit the same symmetry elimination in our problem domain.

3.3 Results

Similar to the experiments described by Kızıltan and Hnich (2001), Falkner (2009) reported that Model A outlined above performed poorly on all but the most trivial house configuration instances. While a solution for 10 Things could be found in a few seconds, problems for 20 Things could not be solved in 30 seconds.

We reimplemented both models in the ECLiPSe CLP framework (Apt and Wallace; 2007) to investigate whether symmetry reduction techniques could help to increase efficiency, and whether different labelling heuristics had a significant impact on the solving process. We chose the ECLiPSE CLP implementation since it offers an integrated framework where a variety of different CSP solving libraries can simultaneously be combined with logic programming and other search procedures. This was advantageous in particular for the model described in Section 4, where constraints of different type must be considered in unison. Based on earlier experiences with CSP libraries such as Minion and Choco, we are confident that the implementation used in our work is of comparable efficiency as other state-of-the-art generic CSP solvers. Encodings using SAT solvers are also expected to perform poorly due to a large number of global and arithmetic constraints increases clause sizes.

Our evaluation was based on a test suite of over 200 generated problem instances of varying size (from 10–450 Things). We focused our investigations on Model B, which showed better performance on our example problem.

Our results are summarised in Figure 3:[1] It can be seen that without symmetry reduction, *a* solution to problems of size up to 450 Things can be solved in a few seconds. Larger problems could not be investigated due to memory exhaustion (> 256Mb).

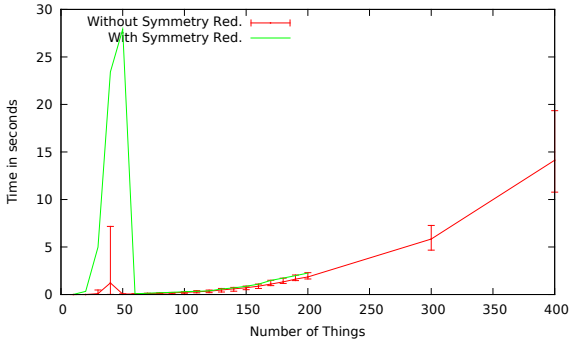


Figure 3: Results for the Matrix CSP model, Approach B

However, we were able to prove for only three of all problems that the solution found by our solver was indeed optimal. In all other cases, optimality could not be established within the set time limit of two minutes per problem. Instead, we estimated the quality of our solutions by comparing each with a lower bound for each solution.[2] We found that the solutions are almost all within factor 1.25 of the lower bound. Hence, for configuration problems where an absolute optimum is not required, matrix representations without symmetry reduction may be an appropriate CSP representation.

From the discussion in Kızıltan and Hnich (2001), we anticipated that adding symmetry reduction constraints would significantly boost the efficiency of our solver. However, our experiments contradicted our expectations: while the symmetry breaking constraints removed much of the redundant assignments, the overhead of evaluating the necessary constraints by far outweighed the benefit for all but the most trivial examples. Most symmetry-breaking constraints are "global" for each column or row and must therefore be evaluated frequently. This overhead by far outweighs the benefit for larger examples. Using symmetry reduction, we could prove optimality in 14 cases (up from 3), but could not find solutions for most problems exceeding 50 Things. Note that Figure 3 only shows times for successfully solved problem instances; cases where no solution could be found within the time limit have been omitted. Hence the graphs for solving with and without symmetry reduction appear similar, while the number of successfully solved cases actually drops to around 15% compared to the "plain" CSP. Hence, in practice, symmetry reduction for this complex constraint optimisation problem does not lead to any improvement.

Furthermore, we confirmed that the labelling strategies used by the CSP solver have considerable impact. In our experiments, we used a variable and value selection strategy that explores variables of W column-wise, with smaller domain values first. This strategy has shown the best results for this problem.[3] This can be explained by the fact that this strat-

[1] The vertical bars represent minimum and maximum values.

[2] This can easily be computed from the total size of all clean (dirty) Things and the fixed Cabinet capacity.

[3] Row-wise labelling produces exactly the same results. Using

egy partly incorporates the symmetry reduction constraints (empty/full Cabinets before full/empty ones) without incurring the same overhead as explicit constraints. In fact, we found this strategy to be backtrack-free for most problems. Surprisingly, the well-known "most-constrained" variable selection heuristic performed poorly; we believe that is due to the large number of global constraints and the additional overhead necessary to search for a suitable variable. Similarly, the "anti-first-fail" variable selection strategy that selects the variable that is least constrained required excessive time and failed to solve many instances.

4 Custom Symmetry Reduction

While rephrasing the configuration problem as a standard CSP has the advantage that available off-the-shelf solvers can be applied, our experiments indicate that this approach may not be the best solution for complex problems.

In this section, we pursue a different approach that is more closely aligned with the conceptual model introduced in Section 2, but applies partial symmetry reduction by transforming the problem. In contrast to the previous section, we include preference constraints into our model.

4.1 Transformed Problem

The case study given in Section 2 exhibits the following symmetries (other than those introduced by the matrix-like representation):

(i) Rooms on the same floor not occupied by a Person are indistinguishable;

(ii) The identity of Cabinets is insignificant. Swapping any two Cabinets (together with their properties, such as low/high, clean/dirty, etc.) results in an equivalent solution;

(iii) Cabinets of the same type (low/high, clean/dirty, upper/lower) within a Room are indistinguishable; their contents can be swapped or mixed without changing the quality of the solution.

We eliminate Symmetries i and ii by restructuring the model into an equivalent representation that does not exhibit such equivalent solutions. We did not attempt to address Symmetry iii, but it is believed that imposing constraints on the order and number of some Things allocated to Cabinets in a Room may eliminate redundancy further.

The structure of our revised conceptual model is shown in Figure 4. The Cabinets and their properties that are explicitly represented in the conceptual model in Figure 1 have been transformed into attributes of Rooms and Things. For each Room, the number of each type of Cabinet is represented as an attribute, with constraints enforcing that a lower Cabinet exists for each upper one, and that the Cabinets can fit within a Room. While the number of Cabinets is represented, their specific positions within a Room are not, hence reducing Symmetry ii.

To address Symmetry i, Rooms that are unoccupied are merged into a single empty Room per floor, with the Room size being the sum of all merged Rooms.

domain splitting or maximal domain values instead also results in the same solutions but requires up to three times as much time.

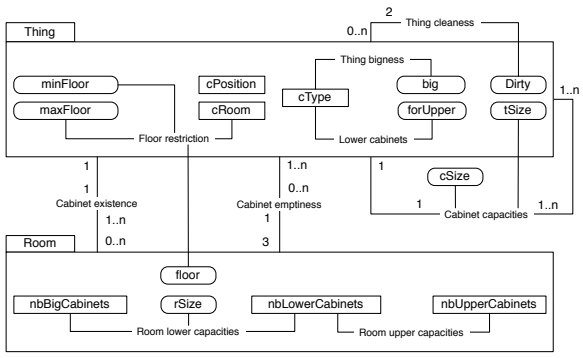


Figure 4: Symmetry-reduced CSP (Bettex; 2009)

Associations between Things and Cabinets are represented as attributes of Thing, capturing the Room, Cabinet type and index within a Room. The constraints are adjusted accordingly. For example, the constraint that clean and dirty Things must be stored in different Cabinets is enforced by a constraint operating on the attributes of Thing:

$$dirty(t_1) \neq dirty(t_2) \Rightarrow cRoom(t_1) \neq cRoom(t_2) \vee cType(t_1) \neq cType(t_2) \vee cPos(t_1) \neq cPos(t_2)$$

Additional constraints are necessary to ensure the Cabinets occurring in attributes of Thing actually exist in the Rooms, and that empty Cabinets should be prohibited (to ensure minimality of the solution). The former constraint for lower Cabinets can be formalised as follows:

$$type(t) = \text{lower} \Rightarrow nbLowerCab(cRoom(t)) \geq cPos(t)$$

Note that this constraint can only be evaluated if attribute *cRoom* of a Thing has been assigned. Our implementation uses techniques that are similar to Constraint Handling Rules (Frühwirth; 1998) to suspend such indirect constraint until they can be evaluated. Our implementation does not apply domain reduction to the index part of such constraints ($cRoom(t)$ in the example), and only checks consistency once variables have been instantiated sufficiently.

We represent the preference criteria for optimal solutions as a lexicographic ordering over the number of Cabinets, the number of Things stored in a Room occupied by someone else than the Thing's owner, and the number of Things stored in vacant Rooms.

Since many of our constraints rely on certain variables being instantiated, variable and value ordering heuristics are critical for solving. The structure of the constraints (Things rely on their Room assignment, and Cabinets in a Room are determined by the Things assigned to them) and initial empirical results lead to a strategy that instantiates the variables belonging to a Thing together without intervening other assignments. Within a Thing, its Room and type of Cabinet are assigned first, where the owner's Room and lower Cabinets are tried first. At this point most constraints relating Things to Rooms can be evaluated to check consistency and prune the possible values of attributes in Room. Once the Things' variables have been assigned, we proceed with the Rooms, with lower values first.

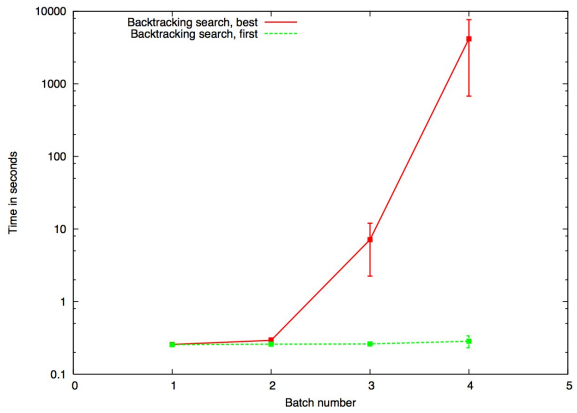


Figure 5: Results for the Custom CSP model

4.2 Results

We attempted to solve our CSP model using a standard backtracking solver using the interval domain and constraint suspension (Rossi et al.; 2006) to obtain a baseline for further evaluation with different search techniques (presented in Section 5). The results are shown in Figure 5. Note that the scale of the ordinate is *logarithmic*.

It can be seen that this approach does not scale; even for trivial problems with four Rooms and twelve Things, problems can take up to several hours to solve optimally. The figure also shows that *some* (possibly sub-optimal) solution is found quickly; however, the problem instances are very small and this observation may not uphold for larger instances. Regarding the quality of the solutions, we find that the (non-exhaustive version of our) backtracking solver produces solutions that are approximately 1.6 times worse than the estimated optimum.

Overall, as informed readers may have expected, complete search techniques based on backtracking are insufficient to devise solutions from our model. This is most likely due to weak consistency enforcement and delayed pruning due to constraint suspension. We investigate alternative search procedures using the same model in the next section.

5 Local Search

The constraint representation outlined in the previous section is effective only for *checking* a candidate solution assignment, where the values of the variables in Thing and Rooms are known. While building such an assignment using finite-domain CSP techniques has proven ineffective (see previous section), it may be possible to use the same constraint representation to *improve* a given solution. In this section, we investigate whether *local search techniques* (Van Hentenryck and Michel; 2005) that iteratively improve an invalid assignment may be suitable to address this problem.

Constraint-based local search and optimisation techniques arrive at an (optimal) solution by altering fully instantiated but possibly invalid assignments to reduce the constraint violations or to improve the overall cost. Many variations of the general scheme exist, with each employing different strategies to navigate from one assignment to the other and to stop the overall process (Van Hentenryck and Michel; 2005).

The variant used in our work is similar to Simulated Annealing (Russel and Norvig; 2003), where probabilistic thresholds are applied to guide exploration. Starting with an initial solution, constraint satisfaction techniques are employed to find minimal sets of constraints that imply infeasibility of a solution ("conflicts"). Once conflicts have been identified, one or more of the involved variables are chosen and assigned different values. Suitable values are determined by domain-specific heuristics. As a result, the previously violated constraint is satisfied, but other constraints may be violated in the new assignment. Hence, the overall quality of a solution may deteriorate rather than improve in a single step.

If the quality of the resulting candidate solution is better than that of the previous assignment it is used as the basis for further exploration. Otherwise, it may be kept (with some probability), or another alternative assignment may be explored. This process continues until all constraints are satisfied and no improvement could be achieved over a number of iterations, or until a threshold on the number of iterations (or time) has been reached.

Since this approach does not require to consider partial assignments, the drawback of delayed and suspended constraint evaluations observed with the backtracking solver are not an issue. The restriction to *complete* assignments also is not a limitation on the problems that can be handled. Partial configurations and dynamically expanding problems can be addressed by using techniques where the relevant scope of variables and constraints is expanded incrementally, as for example in the generative CSP framework (Fleischanderl et al.; 1998).

5.1 Solving Heuristics

To generate the initial solution, we leverage the *Best Fit Decreasing* heuristics from the related class of *Bin Packing Problems* (Lodi et al.; 2002). We consider the Things in order of high before low and in decreasing size. Starting with all Rooms empty, we assign each Thing to a Cabinet in a Room such that all constraints remain satisfied, possibly creating a new Cabinet if no suitable Cabinet can be found. Ties between cost-equivalent Cabinet assignments are resolved by minimising the remaining space in a Cabinet. Otherwise, the Thing is placed in a random location, disregarding any violated constraint. As a result, an initial solution is created that may contain violated constraints and that may not be minimal.

To improve upon the initial solution, heuristics that repair constraint violations are applied iteratively. Conflicts implied by the given candidate assignment are identified using ECLiPSe's conflict monitoring mechanisms (Cheadle et al.; 2003–2006). For each constraint we identify possible repair actions. For example, the constraint that Room capacities must not be exceeded is repaired by selecting and moving cabinets to other Rooms until the capacity constraint is satisfied. Similarly, to ensure only clean or only dirty Things are in a Cabinet, the clean or dirty Things must be moved to another location.

Since local repair choices may critically influence the further progress of the overall optimisation process, we allow our solver to spend considerable time choosing between possible repair actions. In particular, we apply local constraint prop-

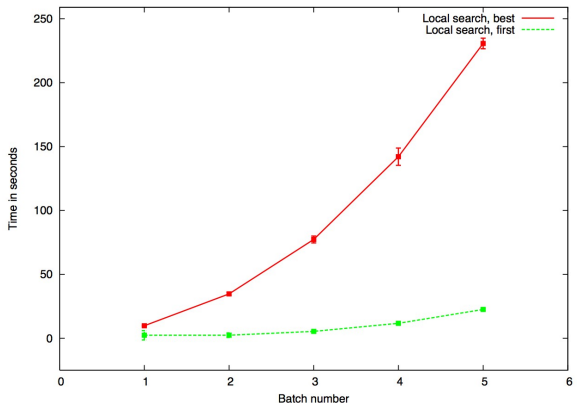


Figure 6: Results for the Custom CSP model

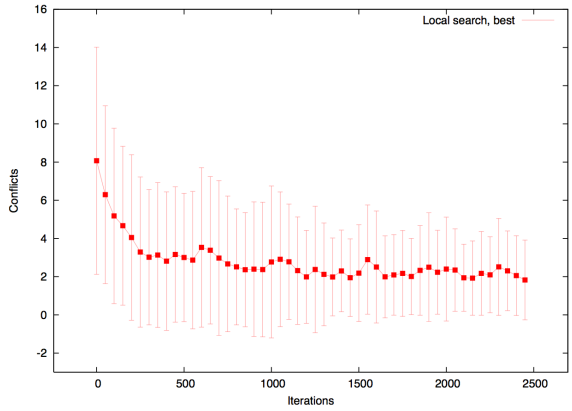


Figure 7: Conflict evolution

agation for different alternative repair assignments to determine the number of conflicts in each assignment. The locally conflict-minimal assignment is then chosen as basis for the subsequent iteration.

While this process is not guaranteed to result in an optimal solution, our results indicate that, typically, good-quality solutions within factor 1.2 of the estimated optimum can be found.

While the repair heuristics incorporate domain-specific knowledge, such as preferred ordering of Rooms, the local minimisation approach allows us to reduce the domain-specific aspects compared to imperative domain-specific solving algorithms. By utilising constraints that are close to the conceptual domain model rather than encodings tailored to low-level finite-domain CSP representations, we are able to largely structure repair actions in terms of domain concepts, which may be an advantage when considering the maintenance of knowledge bases that evolve over time. Using the local repair paradigm, constraints can be used to assess validity of a configuration and repair actions can often be designed and considered without considering the entire minimisation problem. It is not necessary to explicitly factor the search and minimisation strategies into the solving algorithms; this aspect is instead covered by the local exploration strategy that searches for combinations of repair actions that are suitable for achieving a goal or for finding a close assignment.

5.2 Results

Figure 6 summarises the results for our example problem. For this experiment, we selected five batches of example problems from our library, where problems in batch n contained n floors with $6n$ Rooms, $2n$ Persons, and $20n$ Things. We report the averages over five repeated runs.

For each problem an initial candidate assignment was devised using the Bin Packing heuristics introduced in Section 5.1. We then identified conflicts and repeated the repair process until either a conflict-free assignment was found, or up to a maximum of 2500 iterations.

While we always accept assignments with better cost valuation, the probability that an assignment is accepted that contains more conflicts or that has a worse cost estimate is 0.75. Compared to traditional probabilistic optimisation techniques, this value may seem unusually high; however, since we consider only a single repair action at a time, resolving one conflict often requires to create a number of intermediate invalid configuration before another, better configuration can be reached. Figure 7 depicts the evolution (arithmetic mean and variance) of conflicts as the search progresses. It can be seen that while the initial solution may suffer from a considerable number of conflicts, this number drops quickly but keeps oscillating until a solution is found.

Figure 6 summarises the time required for our algorithm. The case where only a valid solution is sought is labelled "*first*", whereas the algorithm that continues for 2500 iterations is labelled "*best*". It is observed that the time required for each solver increases polynomially with the problem size; hence the algorithm scales well to non-trivial problems. Furthermore, since the time required to find the first solution is little and scales well with problem size, the local search strategy can also be used in an "anytime" framework where the solving process can still provide a good solution even if stopped early.

Comparing the absolute time values with those reported in Section 3 is difficult, since the matrix representation of the problem captured only a subset of all constraints. Furthermore, our current implementation is not optimal. We are aware that our local search implementation currently suffers from excessive constraint checking that could be eliminated by using advanced implementation techniques and different constraint systems that handle global constraints more efficiently.

Comparing the quality of the solutions derived by the "best" and the "first" algorithms, we observe that both are virtually indistinguishable, with both being within 10% of the estimated optimum. Both are much better than the non-complete backtracking-based solver (which produced solutions exceeding by as much as 60%). Since differences between the "first" and "best" solution are negligible, the effort required to pursue further optimisation may not be justified in practice.

6 Discussion and Conclusion

We investigated three approaches to solving a prototypical configuration problem by using finite-domain constraint satis-

faction and constraint optimisation techniques. While there is considerable work on efficiently solving particular classes of problems, our configuration domain includes constraints and irregularities that render many of those techniques inapplicable. Our experiments aimed at assessing the suitability of CSP technology for the configuration of complex problems that do not fit squarely in the categories researched previously. Our findings can be summarised as follows:

- The globally optimal solution cannot be found in reasonable time for most non-trivial problem instances. (More precisely, it cannot be proved in reasonable time that the derived solution is indeed optimal.)
- Adding constraints for symmetry reduction seems ineffective for this type of problem. While the backtracking behaviour is indeed much reduced, the overhead caused by non-local constraints that span many variables make this approach slow and unsuitable in practice.
- Heuristic search techniques together with heuristics adapted from Bin Packing problems are able to arrive at good solutions quickly for medium to large-sized problems. However, considerable transition periods between regions of feasible configurations and the evaluation of global constraints remain the critical factors in this process.
- The first valid solution that is found with local search is often close to the optimum, and further search may be limited if time is critical.
- Specifying local repair actions for constraints in combination with generic conflict-driven optimisation procedures can yield an effective configuration framework where complex imperative solving strategies need not be asserted explicitly. Furthermore, this formalism may be more amenable to devising constraint representations that closely follow the domain entities instead of requiring complex translations to structurally different finite-domain CSPs.
- The time required by local search procedures can be predicted more easily than that of conventional CSP search algorithms. Hence, this technique may be preferred for interactive configuration scenarios.

From our experiments we identify the following areas where progress may be most beneficial for future improvements:

- The synthesis of local repair actions from declarative knowledge bases, in particular from domain constraints and optimisation criteria.
- On-line tuning and learning approaches to help choosing suitable algorithm parameters (probabilities, thresholds) and to focus more effectively on promising repairs, depending on non-local contexts and properties of the configuration instance under consideration.
- The identification of criteria which can be used to select a "good" CSP encoding for a problem described by a structured, "object-oriented" knowledge base that is close to the conceptual domain models.

References

Apt, K. R. and Wallace, M. (2007). *Constraint Logic Programming using Eclipse*, Cambridge University Press, New York, NY, USA.

Bettex, M. (2009). *House configuration problem using constraint optimization*, Master's thesis, School of Computer and Information Science, University of South Australia, Adelaide, Australia. Joint work with the Aritificial Intelligence Laboratory, Swiss Federal Institute of Technology.

Cheadle, A. M., Harvey, W., Sadler, A. J., Schimpf, J., Shen, K. and Wallace, M. G. (2003–2006). *ECLiPSe – A Tutorial Introduction*, Cisco Systems, Inc.

Falkner, A. (2009). Choco implementation (partly) of s'upreme showcase house, *Technical report*, Siemens AG, PSE, Vienna, Austria.

Falkner, A. and Schreiner, H. (2008). Two decades' experience in developing product configurators – mastered challenges and remaining issues, *in* J. Tiihonen, A. Felfernig, M. Zanker and T. Männistö (eds), *Proceedings of the Workshop on Configuration at the 18th European Conference on Artificial Intelligence*, Patras, Greece.

Fleischanderl, G., Friedrich, G., Haselböck, A., Schreiner, H. and Stumptner, M. (1998). Configuring large-scale systems with generative constraint satisfaction, *IEEE Intelligent Systems* **13**(4).

Frisch, A. M., Jefferson, C. and Miguel, I. (2003). Constraints for breaking more row and column symmetries, *in* F. Rossi (ed.), *CP*, Vol. 2833 of *Lecture Notes in Computer Science*, Springer, pp. 318–332.

Frühwirth, T. W. (1998). Theory and practice of constraint handling rules, *Journal of Logic Programming* **37**(1-3): 95–138.

Kızıltan, Z. and Hnich, B. (2001). Symmetry breaking in a rack configuration problem, *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, Seattle, WA.

Lodi, A., Martello, S. and Monaci, M. (2002). Two-dimensional packing problems: A survey, *European Journal of Operational Research* **141**(2): 241–252.

Narodytska, N. and Walsh, T. (2007). Constraint and variable ordering heuristics for compiling configuration problems, *in* M. M. Veloso (ed.), *Proceedings International Joint Conf. on AI*, pp. 149–154.

Rossi, F., Beek, P. v. and Walsh, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, Elsevier Science Inc., New York, NY, USA.

Russel, S. and Norvig, P. (2003). *Artificial Intelligence A Modern Approach*, 2 edn, Prentice Hall.

Stumptner, M. and Soininen, T. (2003). Special issue on configuration, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **17**(1).

Thiagarajan, R., Mayer, W. and Stumptner, M. (2009). Generative composition of web services, *CAiSE Forum*, CEUR Workshop Proceedings, Amsterdam, The Netherlands. Forthcoming.

Van Hentenryck, P. and Michel, L. (2005). *Constraint-Based Local Search*, The MIT Press.