Metaheuristics as Generic Search Procedures for Constraint Programming

Georg Ringwelski[1], Ole Boysen[2] and Kathleen Steinhöfel[2]

[1] 4C, UCC, Cork, Ireland. g.ringwelski@4c.ucc.ie
[2] Fraunhofer FIRST, Kekuléstraße 7, 12489 Berlin, Germany.
boysen|kathleen@first.fhg.de

Abstract. We introduce a generic approach to incorporate metaheuristic search procedures into constraint programming. This allows the individual combination of both paradigms enabling us to exploit their respective advantages. Arbitrary constraint satisfaction and optimization problems can be solved from a declarative constraint model by means of propagation and both (discrete and heuristic) search paradigms. We show the run time generation of penalty functions for the supplementary algorithms and give an overview of our current results and the different search components.

1 Integrating Metaheuristics and Constraint Programming

This paper addresses a generic scheme for solving constraint satisfaction problems with constraint programming (CP) and integrated metaheuristics (MH). During the last decade many propagation methods have been developed to speed up the complete search performed by constraint programming. For problems where complete search is inefficient such as real-life combinatorial optimization problems recent research has been directed to metaheuristics, a class of general, non-deterministic optimization algorithms comprising local search procedures like Simulated Annealing and Tabu Search as well as constructive heuristics like Genetic Algorithms and others. To take advantage of thoroughly developed propagation procedures of constraint programming and local search methods we propose a hybrid system that incorporates both in a generic manner. Constraint satisfaction or optimization problems are modelled, as in CP, using variables and a conjunction of constraints or optimization goals while the algorithm to solve the problem can be individually defined in every application. Metaheuristics as well as complete labeling algorithms can be used to cooperatively find variable assignments in different parts of the search space. We thus extend constraint programming with new generic search algorithms that use a set of variables and

This work was partially funded by the Embark Initiative of the Irish Research Council for Science, Engineering and Technology under grant PD/2002/21. The Cork Constraint Computation Center is supported from Science Foundation Ireland under Grant 00/PL.1/C075.

constraints as input and produce a variable assignment as output. In difference to standard labeling this assignment does not have to be a solution to the global CSP such that our approach can also be used for over-constrained problems. On the other hand additional optimization criteria can be passed to the metaheuristics to find "good" solutions in under-constrained or optimization problems.

Approaches to combine propagation and local search have been presented e.g. in [5, 10, 13, 11]. In the latter paper local probing is used to utilize simulated annealing subsequently in a global complete search procedure. This constitutes one possible cooperation of complete and local search as we propose it in this paper. Moreover, propagation methods are often used to guide local search procedures, cf. [12, 10, 15]. The latter paper presents a simulated annealing procedure that takes advantage of propagation to avoid unfeasible variable assignments and therefore it performs optimization in the feasible search space only. In our approach we can also perform such pruning of the neighborhood during local search by using so-called propagators. These derive values for some variables from the current assignment of others.

Most of the proposed approaches have been tailored for the considered problem setting, e.g. [11, 15]. Instead of providing a generic interface to CSP modelling, they use domain specific implementations to compute the neighborhood and penalty of current variable assignments. Thus, they cannot be considered as general search procedures for CSP to be used within constraint programming. We propose a general, domain independent, generic scheme to combine constraint programming with local search. Our system enables metaheuristics and local search procedures to be used in the same way as classical labelling procedures during the incremental constraint processing of CP. Since the interplay is not predefined it is possible to obtain efficient heuristics to solve constraint satisfaction/optimization problems by passing all necessary information to the MH. Therefore, we extend the classical constraint programming while preserving all features of local search.

2 Runtime Generation of Penalty Functions

Our approach to a hybrid constraint processing system is the integration of metaheuristics as labeling procedures in incremental constraint programming systems. A CSP can be iteratively modelled by variables with associated domains and constraints in a programming system that is based on the CLP scheme [8] such as CHIP od SICStus Prolog. To find solutions to the thus defined CSP we propose the use of a non-deterministic search algorithm instead of backtracking and labelling. In order to remain the incremental handling of knowledge in the CP system, the search algorithm has to adapt itself to changing models. In difference to other approaches that use metaheuristics for constraint satisfaction (e.g. [2]), we do not define penalty functions for the metaheuristic in compile time, but generate them dynamically during runtime.

In a constraint logic programming system for example, we allow thus for the use of local search for dynamically defined (sub-)problems of arbitrary CSPs. In one of our current projects with a major German railway company, we investigated that the best solutions for very large CSPs can be found by first running Simulated Annealing for the subproblem that contains all the information to be optimized and then using the efficient and powerful (forward-checking) propagation of CLP to make the remaining arithmetic evaluations. In order to allow this generic integration of MH and CP we use as a basic penalty function a total mapping of variable assignments to integers, representing the number of violated constraints.

Definition 1. *Let a $CSP = (V, C)$ be given, where $C = \{c_1, ..., c_m\}$ is a set of constraints that have as actual parameters a list of variables*[3] $\nu(c_i) = \langle v_1, ..., v_n\rangle \dot{\subseteq} V$ *and a list of constants $\kappa(c_i) \dot{\subseteq} D$ of a previously defined constraint domain D. The variables are associated to their domains $\delta(v_i) \subseteq D$ and the declarative semantics of every constraint is given by the relation $\models \subseteq C \times \mathcal{P}(\delta(v_1) \times ... \times \delta(v_n) \times D \times ... \times D)$. We define the* **basic penalty function** $\phi_0 : (V \to D) \to \mathbb{N}$, *as a mapping of variable assignments to a natural number such that $\phi_0(ass) := |\{c \in C \mid c \not\models (\langle ass(v_1), ..., ass(v_n)\rangle \circ \kappa(c))$ with $\langle v_1, ..., v_n\rangle = \nu(c)\}|$.*

This basic target function returns the number of violated constraints, as referred to MAX-CSP in [6]. In Section 4 we will discuss more specialized penalty functions that support optimization and soft constraints. However, this basic target function has proven to be quite effective when used with Simulated Annealing, i.e. Codognet and Diaz used it when obtaining immense performance for some benchmark CSPs in [2].

The main contribution of this paper is a method to generate such penalty functions for arbitrary sets of variables and constraints. In our setting, the MH uses a CSP as input and evaluates a variable assignment as output. From the input a penalty function is generated that is used by the search algorithm to find "good" variable assignments, which are in most cases solutions of the CSP. The challenge of this approach is to find an efficient algorithm that encodes a CSP from a set of variables and constraints to have as little system overhead as possible. This is a non-trivial problem due to the complex structure of constraint networks (hyper graphs) and the requirement to encode well-defined ($\forall c \in C : \nu(c) \dot{\subseteq} V$) CSPs. We developed and implemented an algorithm for this encoding and the penalty function generation that has a time (and memory) complexity that is linear with the size of the CSP. In Figure 1 we show the absolute values for the computation overhead in some benchmark programs, which is negligible compared to the overall runtime of search.

We start from a constraint program that defines a set of constraints C and a set of variables V in a constraint domain D (i.e. $\forall v_i \in V : \delta(v_i) \subseteq D$). Furthermore the set of allowed constraints in D must be specified as the set AC.

Example 1. For example in clp(FD) [4] the constraint domain is $D = \mathbb{Z}$ such that allowed constraints AC are (among others) mathematical relations and the

[3] With $\dot{\subseteq}$ we denote the inclusion of list elements in sets: $\langle v_1, ..., v_n\rangle \dot{\subseteq} V \Leftrightarrow \forall i \in \{1, .., n\} : v_i \in V$.

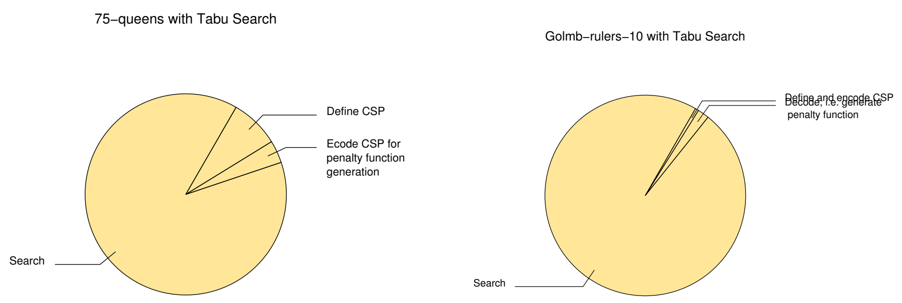

Fig. 1. The runtime consumption of the different tasks to solve CSPs with Tabu Search in CHIP.

pertinent global constraints. The constraint $X \leq 5$ is then represented by the allowed constraint type $' \leq '$ together with its associated variables $\nu(X \leq 5) = \langle X \rangle$ and constant arguments $\kappa(X \leq 5) = \langle 5 \rangle$ as actual parameters.

A CSP $(\{v_1, ..., v_n\}, \{c_1, ..., c_m\})$ in such a constraint program with variables $\{v_1, ..., v_n\} \subseteq V$ and constraints $\{c_1, ..., c_m\} \subseteq C$ is encoded in two steps:

1. Traverse the variables $\{v_1, ..., v_n\}$ to
   (a) create a list of variable domains $DL = \langle \delta(v_1), ..., \delta(v_n) \rangle$
   (b) create a partial mapping of variables to indices in DL:
   $$\iota : V \nrightarrow \{i_1, ..., i_n\} \text{ with } \iota(v_j) = \begin{cases} i_j, & \text{if } 1 \leq j \leq n, \\ \bot, & \text{otherwise.} \end{cases}$$
   by a pointwise definition.
2. Traverse the constraints $\{c_1, ..., c_m\}$ to create a list CL of pairs $(\rho(c), \alpha(c))$ for every $c \in \{c_1, ..., c_m\}$ with $\nu(c) \dot{\subseteq} \{v_1, ..., v_n\}$, by using
   (a) a bijective mapping $\rho : AC \to \mathbb{N}$ to encode the kind of c,
   (b) a function $\alpha : C \to \mathcal{P}(\{i_1, ..., i_n\} \cup D)$ that creates a list of actual parameters for each constraint, such that $\alpha(c) = \alpha'(\nu(c)) \circ \kappa(c)$ with $\alpha'(\langle v'_1, ..., v'_k \rangle) = \langle \iota(v'_1), ..., \iota(v'_k) \rangle$,
   (c) the domain of ι to decide, if $\nu(c) \dot{\subseteq} \{v_1, ..., v_n\}$:
   $\forall c \in \{c_1, ..., c_m\} : (\forall v \in \nu(c) : \iota(v) \neq \bot) \Rightarrow ((\rho(c), \alpha(c)) \in CL)$

Example 2. Let the constraint encoding $\rho = \{(\leq, 1), (alldiff, 2)\}$ be given in a clp(FD) system. Let further be the variables $\{v_1, v_2, v_3\}$ be defined in a CLP program such that their domains are $\delta(v_i) = \{0, ..., i\}$. The CSP $(\{v_1, v_2, v_3\}, \{v_2 \leq v_1 \wedge alldiff([v_2, v_1, 3, v_3, 5])\}$ will then be encoded by our algorithm as $DL = \langle \{0, 1\}, \{0, 1, 2\}, \{0, 1, 2, 3\} \rangle$ and $CL = \langle (1, \langle i_2, i_1 \rangle), (2, \langle i_2, i_1, 3, i_3, 5 \rangle) \rangle$ by traversing the constraints and the variables only once.

The metaheuristic or local search algorithm is then using the list DL of variable domains to select values for solution transitions. An initial variable assignment (in terms of CP) respectively solution (in terms of MH) can be defined as a mapping of variables to randomly selected values from every domain:

$$sol_0(v_i) := d_i \text{ with } d_i \in \delta(v_i) \tag{1}$$

During search, any of these values can be modified to traverse the search space and eventually find improved solutions w.r.t. to the applied penalty function. The neighborhood function used defines which variable and which new value for it are selected during this non-deterministic search process.

The basic penalty function (Def. 1) for arbitrary CSPs uses a cost function ϕ_c for each constraint c and combines all of these functions from the list CL by summation into a single penalty function during runtime:

$$\phi_0(sol) = \sum_{c \in CL} \phi_c(\alpha(c), sol) \tag{2}$$

The cost function ϕ_c for every constraint is created during compile time. This function uses the constants $\kappa(c)$ and the variable indices $\alpha'(\nu(c))$ to select the current values from sol to evaluate the total number of violated constraints:

$$\phi_c(\langle \iota(v_1), ..., \iota(v_n) \rangle \circ \kappa(c), sol) = \begin{cases} 0, \text{ if } c \models (sol(\iota(v_1)), ..., sol(\iota(v_n))) \circ \kappa(c), \\ cost_c, \text{ else.} \end{cases} \tag{3}$$

For primitive constraints, such as simple arithmetic equations or inequations, a violation will normally have fixed costs: $cost_c = 1$. Whereas violations of global constraints may produce higher and more various costs. Using $cost_c$-values that are related to the degree of constraint violation, e.g. the number of violations of implied primitive constraints, has also shown to be effective in [2].

Example 3. The cost function for an alldifferent constraint in our C-implementation counts the values that are used more than once in the current solution sol:

```
static float alldiff(struct constraint con,int sol[]){
  int i,j;
  float res = 0.0;

  for(i=0; i < con.arity; i++)
    for(j=i+1; j < con.arity; j++){
      if(value(con.args[i],sol) == value(con.args[j],sol))
        res = res + 1.0;
  return res;
}
```

The function `value` returns the current value (w.r.t. `sol`) of its arguments `con.args[x]`, which is a number or a variable index. In the latter case the `value` is evaluated as `sol[con.args[x]]`.

3 Metaheuristics for CSP

Numerous combinatorial problems occurring e.g. in engineering or business can be modeled as CSP's. This very general class of problems includes such popular problems like the job shop scheduling and the travelling salesman problem. Therefore, intensive research has been directed towards solving CSP's as a super class of problems. Nevertheless, especially in context with large and complex real world optimization problems exact methods including CLP become prohibitively time consuming. In most real world applications, good solutions obtained in reasonable computational time are more desirable than an optimal solution in long or even infinite time. Consequently, the use of approximate methods is commanded to solve such problems and have been studied intensively in recent years. Metaheuristics is a class of such approximate algorithms that has been applied to all kinds of combinatorial optimization problems very successfully in research as well as industrial settings. The term metaheuristics subsumes amongst others local search algorithms like Simulated Annealing and Tabu Search, population-based heuristics like Genetic Algorithms and Ant Colony Optimization, and their hybrids. Usually, simple heuristical search algorithms, like greedy search, suffer from their incapability of overcoming local optima: Once they have found a local optimum, they are stuck and search is terminated. Metaheuristics are more systematic or intelligent approaches to guide such simple heuristics so that this kind of premature search termination is avoided and the entire search space is considered sufficiently to find a good local or ideally the global optimum. The great success of metaheuristics is partly due to their simplicity and flexibility. In order to allow the exploitation of the respective advantages of CLP and MH in different application areas of solving CSP's, we envision the constructed hybrid CLP-MH constraint processing system as a more powerful and versatile tool.

Basically, the search space for our MH algorithms is given by all possible assignments of variables to values from within their domains. A "solution" for a MH is any such complete variable assignment regardless if the constraint set C is satisfied or not. To find new variable assignments during search, MH algorithms change or combine values for one or several variables of the current assignments. Different means of traversing the search space, i.e. transforming one solution state into another, are used by metaheuristical paradigms. For example, population based-methods, like for instance Genetic Algorithms , usually transform one search state into the next by combining characteristics of the current set of solutions into a new set of solutions. In contrast, in trajectory-methods, e.g. Simulated Annealing, each search state consists of a single solution which is transformed into a new one by modifying a few value assignments. All solutions reachable by a single transformation, as described e.g. in (4), constitute the neighborhood of the actual solution. How search states are transformed, variables and values for modification are chosen, local minima are overcome, and memory is used for these purposes is a design feature of the particular search algorithm.

Other approaches to the CSP usually take advantage of specific problem knowledge not available in context with our very general approach, i.e. we cannot use very specialized penalties as surrogate functions or specialized neighborhoods. Nevertheless, we have identified some types of general neighborhoods reoccurring in several problem classes. For out test-CSP's we used three types of basic, general neighborhoods for use in the local search algorithms, which are applied according to the specific problem. The first one ist the shift move: One variable is chosen and assigned to a new value from within its domain. Starting from a random assignment like presented in (1), the selection process for each local move is as follows:

1. Select the index j of the variable to be changed.
2. Select the new value $d \in \delta(v_j)$ to be assigned.

The resulting assignment is then the mapping

$$sol_n(v_i) = \begin{cases} d, & \text{if } i = j, \\ sol_{n-1}(v_i), & \text{else.} \end{cases} \tag{4}$$

The shift move is applicable to any type of problem but may be inefficient. In case of the Magic Squares problem for example, we can use the knowledge that any solution is a complete ordered set, i.e. a permutation of a given set of different values. Here, we apply a swap move neighborhood where the values of two variables are exchanged. The third type is used for incomplete ordered sets like the Golomb Ruler problem: Any solution is a permutation of a given set of different values but not all values appear in the solution vector. Then we use a mixed neighborhood where sometimes swap moves are applied and sometimes values leave and enter the solution.

For the development of a prototype system we considered three popular MH algorithms, namely genetic algorithms, simulated annealing, and tabu search. **Simulated Annealing** (SA) is a search procedure that has its origin in a simulation model for describing the physical annealing process for condensed matter. The identifying feature for SA algorithms is a temperature parameter T which controls the probability of accepting a deteriorating moves with objective function deterioration Δ. In each search step SA chooses a move randomly and executes it if it improves the current solution. Otherwise, it accepts the move with probability $e^{-\Delta/T}$ depending on the level of deterioration Δ. The temperature T is cooled down during the solution process. SA is described in detail e.g. in [7].

In Section 4.2 we will describe one heuristic that worked well with many CSP's. Several heuristics can be used to define the best moves during search. The overall goal of such transitions is to reduce the result of of the penalty function (2). However, often transitions have to be made that increase this value in order to escape from local minima.

Tabu Search (TS) allows, similar to SA, all possible moves during each search step: It tentatively chooses the best move available, i.e. the most improving or the least deteriorating one, but uses some type of temporary memory or "tabu list" to avoid the revisiting of previous solutions and thereby hopefully escapes the attraction of a local minimum and moves on through the search space.

Usually, the reassignment of a variable to a value that it took during some recent period is forbidden by the tabu list. A brief tutorial on TS can be found e.g. in [9].

While Simulated Annealing is based on principles of physical science, **Genetic Algorithms** (GA) are search techniques based on an abstract model of natural evolution. GAs, view solutions as individuals, which are members of a population. Each individual is characterized by its fitness which is associated with the objective function. The procedure works iteratively and each iteration is referred to as a generation. Only surviving individuals of the previous generation and new solutions or children from the previous generation are members of the current generation. The population size usually remains constant during the procedure. Children are generated through reproduction and mutation of individuals, parents. A mutation is similar to a neighborhood function a local operation on part of a solution. In each generation only the fittest parents reproduce. For a thorough introduction to GA cf. [14].

In our approach, every individual is given by a variable assignment. Consequently a generation is set of assignments and the fitness is evaluated by our penalty function (2). Holding several different variable assignments, this algorithm makes several local moves with the creation of every new generation. These local moves are all defined by "crossover" and "mutation".

1. The crossover in a generation n of individuals x and y, which we denote as the assignments $sol_{n,x}$ and $sol_{n,y}$, can be defined as a set I_x of indices of values to be selected from x

$$crossover(sol_{n,x}, sol_{n,y})(v_i) = \begin{cases} sol_{n,x}(v_i), & \text{if } i \in I_x \\ sol_{n,y}, (v_i), & \text{else.} \end{cases} \tag{5}$$

2. The mutation is executed on every individuum after being created by a crossover. It is given by one or more local moves like they are presented in Equation (4).

Again, several heuristics can be used to define the set I_x in every pair of individuals and to define the mutation. As in simulated annealing these local moves aim on decreasing the result of the penalty function in the long term and can increase it to escape from local minima.

3.1 Combining Penalty Functions

In many applications, a measure of quality is given among the different solutions of (underconstrained) CSPs. For such cases, we combined a penalty function ϕ_{opt} for this quality measure with the basic penalty function (2) in one search algorithm.

The basic approach for this was the use of a hard upper bound as a constraint, which is gradually decreased in order to find better solutions. This approach simulates the `min_max` constraint, which is supported in many CP systems to enable optimization with discrete CP methods.

As another approach we used the sum of both penalty functions to choose the local moves and thus used a real optimization function instead of a hard constraint. Using this heuristic, we could improve the runtime, because the optimization objectives can be kept separate in the MH (see Fig. 2). We could even improve this heuristic by using a weighted sum where the wights adapt themselves to the progress of the search process. This simple heuristic has turned out to be very effective and is yet very easy to implement and to apply in CP programs.

In a second approach to obtain a neighboring assignment for such problems, we implemented two nested optimization routines. The inner procedure optimizes using our basic penalty function ϕ_0 and in a outer procedure solutions that are "good" w.r.t. ϕ_0 are optimized according to ϕ_{opt}. If in the inner procedure optimal assignments ($\phi_0(ass) = 0$) are found, the outer procedure will optimize among solutions of the CSP. This setting, however, was found to be very time consuming and can thus probably only be used if a very good solution is to be found in a long computation time. If less time is available, the outer routine can also optimize among suboptimal ($\phi_0(ass)$ close to 0) assignments which are not solutions of the CSP. This setting would lead to good results much faster than enforcing the configuration space for the outer routine to consist only of solutions.

4 Benefits of the Hybrid Approach

4.1 New Constraints and Optimization

The integration of new types of constraints, respectively their propagators, can be very difficult in incremental CP systems [3]. The reason for many problems arising here is that propagation operates on variable domains and not just on values. If a constraint uses an operation from the algebra of the constraint domain (e.g. $\mathbb{Z}$), it may be hard to "lift" this operation to the algebra of the variable domains (e.g. interval arithmetic).

Since metaheuristics do not operate on variable domains but only on values from within these domains to determine constraint violations such problems are avoided. The declarative semantics which are defined by the algebra of the constraint domain can be implemented with the operations and relations of this algebra and do not have to be "lifted" to the algebra of the variable domains. In our framework, new constraints can thus be added to the list of allowed constraints (AC) and to the constraint encoding (ρ) to allow the implementation of their declarative semantics on the "value"-level in a corresponding primitive cost function. Such new constraints will then have no operational semantics in CP, i.e. they do not propagate. Nevertheless, our penalty function will consider them such that they tend to be satisfied during the MH solution process. Any feasible solution returned by MH satisfies these additional constraints.

This integration of new constraints cannot only be done for hard constraints, that are not available in the actual CP system, but also for soft constraints. In many application areas of CP soft constraints are used to distinguish between several feasible solutions and to determine the "best" among those [1]. Thus, we can use soft constraints to express optimization goals and thus use them to integrate COPs in CSPs. Such soft constraints for optimization purposes ca be integrated in the proposed approach via special optimization penalty functions. In 3.1, we have described some possible combinations of the thus arising different optimization objectives.

There are various ways of taking multiple soft constraints resp. objectives into account. In the benchmark tests presented in Figure 2, we used a weighted sum penalty function with heavily weighted hard constraint violations which aggregates all penalty functions to one function: The penalty of inconsistencies is just multiplied and added to the penalty of the optimization function. But other methods as proposed in the operations research literature, like for instance the deviation sum strategy, can be easily adopted by our approach.

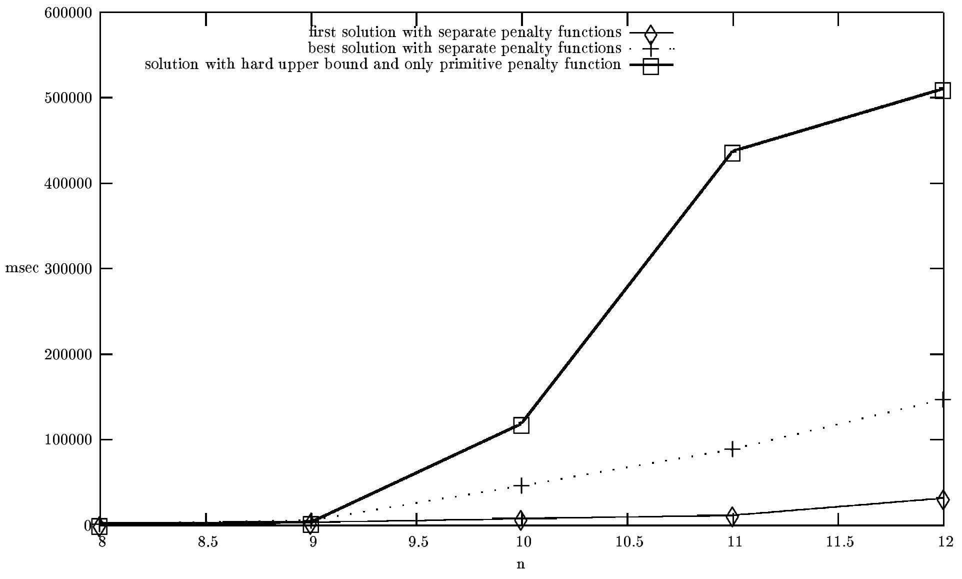

Fig. 2. The golomb-rulers n benchmark using separate penalty functions for constraint satisfaction and optimization. All values are means of 10 runs of Tabu Search with mixed-neighborhood in milliseconds. As "hard" upper bound for the test without separate optimization function we used $1.1 * opt$, where opt is the known optimal value. In the other tests, the best solution found was always around $1.1 * opt$ and the quality of the first solution was in average, approximately $1.6 * opt$

4.2 Violation Depending Local Moves

In our approach we provide the number of violated constraints a variable is involved in as a heuristic for variable selection. A heuristic that selects always the most violating variable has been applied very successfully in [2].

For this purpose, the penalty function defines as a side effect a mapping of variables to natural numbers. This mapping assigns to every variable v the number of violated constraints c with $v \in \nu(c)$. The computational effort caused is constant and thus there is no significant effect on the performance of the algorithm. This mapping can then be used as a heuristic to select the variable which is likely to improve the solution if its value is changed.

In Figure 3 we show some runtime results of our Tabu Search applied to the well know n-queens problem to evaluate different instances of this heuristic. We tried to explore the tradeoff between near to optimal local moves, which are expensive to compute, and more random moves wich can be generated quickly but can be worse w.r.t. global optimization. Thus we tested, how important for the local moves it is to change the variables with many violations prior to those with one very little, but not 0, violations. As can be seen from the figure, none of the tested heuristics is constantly dominant over all instances. However, using no heuristic leads to runtime results far worse (several minutes for 20 queens), thus we did not include these results. Recapitulating on this, we claim that using violations to select neighbors in local moves is very useful, but the information on the number of violations is unlikely to provide better heuristics for local moves.

4.3 Propagators for Metaheuristics

Another method to improve the performance of the search algorithms we have investigated is the use of propagators. Forward-checking is implemented in almost all CP systems, because it can significantly increase search performance. In metaheuristics, usually no such explicit propagation is used. However, in experiments we have determined, that the application of forward-checking to some kinds of constraints is very fruitful for the efficiency of a metaheuristic in the context of solving general CSPs (or COPs). It allows to exclude inconsistent variable assignments in the local moves. This is done by only permuting some values of the assignment and by evaluating all the others using forward-checking. For example, to find a variable assignment that does not violate a simple arithmetic constraint such as $X + Y = Z$, it is much easier to assign Z to the current value of $X + Y$ than to guess three values which satisfy the constraint. In Figure 4 we show the runtime results of the all-intervals benchmark with the use of propagators.

Such assignments to variables that are driven by propagation and not by the normal Local Search methods can be integrated in our approach as new constraints. The primitive cost function for such constraints will always return zero and, as a side-effect, assign values in a forward-checking manner. This propagation is done before the evaluation of the cost for an assignment, which can be implemented in the penalty function: first execute all propagators, then execute the “regular” constraints.

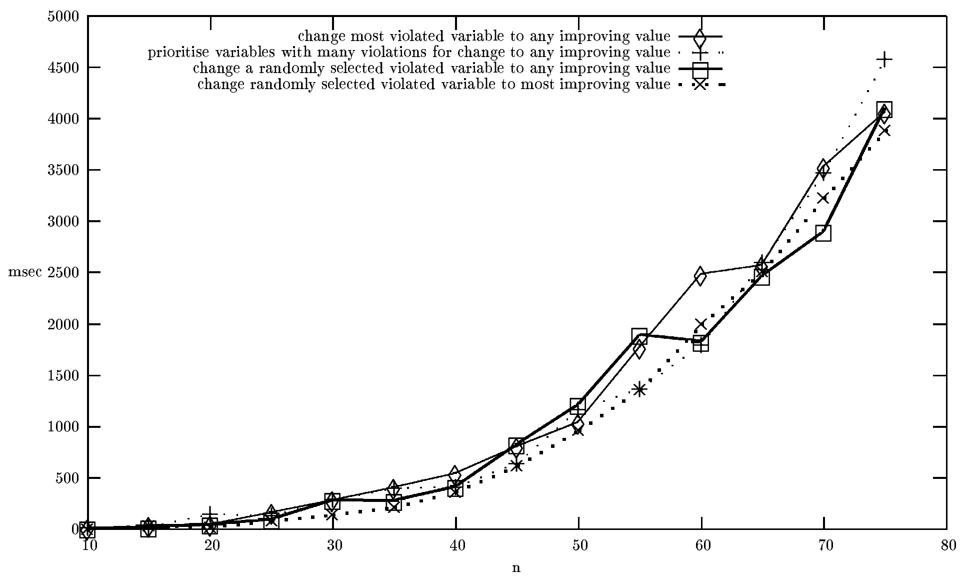

Fig. 3. The effect of using the violations to determine local moves. The n-queens problem solved by Tabu Search and shift-move neighborhood with different neighborhood functions . Ordinate depicts the median of 25 runs in milliseconds.

5 Conclusion

In this paper we give an overview of our current research on a generic interface between Constraint Programming and Metaheuristics. Different metaheuristic search procedures are incorporated as (non-backtrack-able) labeling predicates in CLP. The MH procedures use penalty functions to solve constraint satisfaction problems, that are generated during runtime from the specified constraint model. If applicable, also special heuristics can be integrated to improve the search performance. Thus we allow for any efficient and specialized optimization method from operations research to be integrated in specialized search algorithms. On the other hand the pure CSP model, as it is defined in a CP language, can be used as a basis for the metaheuristic search procedures, such that no specials search strategies have to be implemented to use arbitrary MH algorithms.

We have implemented and analyzed different general heuristics that apply to CSP, such as violation depending local moves or forward-checking procedures for the generation of the neighborhood. Some of these heuristics when applied to CSP have already shown to be quite effective w.r.t. search performance and optimization capabilities. Thus we expect, that we can find configurations for our generic approach that combine the good features of both programming paradigms very soon. With this integration, we preserve the declarative modeling

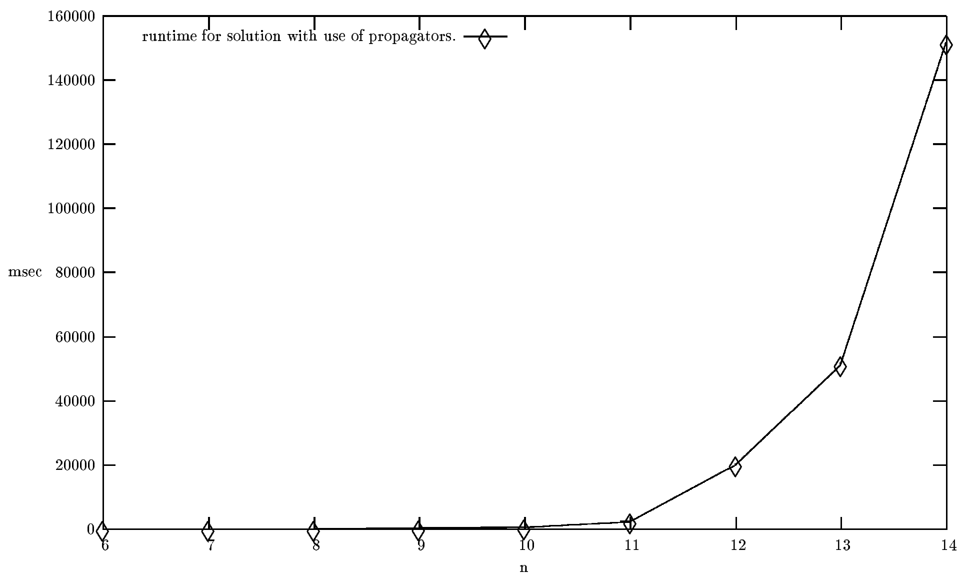

Fig. 4. The mean runtime of 10 runs of Tabu Search with propagators and shift-move neighborhood on the all-intervals n benchmark. The same algorithm would use more than 5 minutes for $n = 6$ if used without propagators.

features and the strong propagation methods of CP while gaining the efficient optimization properties from the MH used.

References

[1] Stefano Bistarelli and Ugo Montanari and Francesca Rossi *Semiring-based Constraint Satisfaction and Optimization* Journal of the ACM 44(2), pp.201–236, 1997.

[2] Philippe Codognet and Daniel Diaz *Yet Another Local Search Method for Constraint Solving* In K. Steinhöfel, editor, SAGA 2001, LNCS 2264, pp. 73–89, Springer 2001.

[3] Alain Colmerauer *Solving the multiplication constraint in several approximation spaces* Invited talk at CP 2001, Paphos, November 2001.

[4] M. Dincbas and P. van Hentenryck and H. Simonis and A. Aggoun and T. Graf and F. Berthier *The Constraint Logic Programming Language CHIP* Proceedings of the International Conference on Fifth Generation Computer Systems, 1988.

[5] F. Focacci, F. Laburthe and A. Lodi *Local Search and Constraint Programming* In F. Glover, G. Kochenberger, Eds., Handbook of Metaheuristics, Kluwer Academic Publishers, 2003.

[6] Eugene Freuder and Richard Wallace *Partial Constraint Satisfaction* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89, Detroit, USA, pp.278–283, 1989.

[7] D. Henderson, S.H. Jacobson and A.W. Johnson *The Theory and Practice of Simulated Annealing* In F. Glover, G. Kochenberger, Eds., Handbook of Metaheuristics, Kluwer Academic Publishers, 2003.

[8] Pascal van Hentenryck *Constraint Satisfaction in Logic Programming* MIT Press, 1989.

[9] A. Hertz, E. Taillard and D. de Werra *A Tutorial on Tabu Search.* Proc. of Giornate di Lavoro AIRO'95 (Enterprise Systems: Management of Technological and Organizational Changes), pp. 13–24, Italy, 1995.

[10] Narendra Jussien and Oliver L'Homme *Local search with constraint propagation and conflict-based heuristics* Artificial Intelligence 139(1), pp. 21–45, 2002.

[11] Olli Kamarainen and Hani El Sakkout *Local Probing Applied to Scheduling* In P. van Hentenryck, editor, Principles and Practice of Constraint Programming - CP2002, LNCS 2470, pp. 155–171. Springer 2002.

[12] S.Minton, M. Johnson and P. Laird *Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems* Artificial Intelligence 58, pp. 161 – 206, 1992.

[13] Gilles Pesant and Michel Gendreau *A view of local search and constraint programming* In E. Freuder, editor, Principles and Practice of Constraint Programming - CP1996, LNCS 1118. Springer 1996.

[14] C. Reeves *Genetic Algorithms* In F. Glover, G. Kochenberger, Eds., Handbook of Metaheuristics, Kluwer Academic Publishers, 2003.

[15] K. Steinhöfel, A. Albrecht and C.K. Wong *Two Simulated Annealing-Based Heuristics for the Job Shop Scheduling Problem* European Journal of Operational Research, 118(3), pp.524–548, 1999.