Composing Global Constraints for Local Search

Markus Bohlin

Swedish Institute of Computer Science
Kista, Sweden
markus.bohlin@sics.se

Abstract. In this paper we propose a new *compositional* approach for global constraint design and implementation for local search. Global constraints in local search can be seen as high-level modeling components, designed to reduce evaluation time and space complexity. Traditionally, they have been implemented as monolithic entities, often using a low-level language and requiring in-depth knowledge of the constraint system itself. In this work we propose a *compositional model* of global constraints, in which we use graph structures, filters and cost components to create global constraints using a high-level C++ framework called *Composer*. The composed constraints can then be used for constraint solving in a generic, domain-independent local search solver. We show how to compose several well-known global constraints, and also show by experimental results that a compositional approach for global constraint modeling is not only possible in practice, but also highly competitive with existing low-level constraint-based local search implementations.

1 Introduction

The constraint satisfaction problem (CSP) is a well-known problem in computer science. Informally, a CSP can be defined as follows: Given a set of constraints, is there an assignment to the variables in the constraints such that all constraints are satisfied? A large amount of scientific activity has been spent on solving generic CSP's. One such approach is the *constraint programming* approach, which has evolved from generic systematic search methods since the late 70's.

Another approach for combinatorial problem solving is to use incomplete search methods such as local search. Heuristic methods based on local search for solving constraint problems have evolved during the last fifteen years into becoming one of the most powerful techniques for solving large combinatorial problems, often outperforming complete algorithms. The classical approach for generic constraint solving in local search is to provide a set of *primitive* constraints, which in turn can be used to form more complex combinatorial substructures [9]. A natural choice is to restrict the user to binary constraints only. Unfortunately, for several common combinatorial structures there simply do not exist a decomposition into binary constraints that is acceptable in terms of space and/or time complexity. Global constraints exist in local search systems to provide time- and space efficient high-level components capturing common combinatorial substructures.

One such constraint is the well-known `alldiff` constraint, restricting all variables present to take disjoint values. A natural decomposition of this constraint reduces to $n(n-1)/2$ binary inequality constraints, where n is the number of variables. Unfortunately, the evaluation of the effect of a change of one variable in such a set of constraints has a time complexity of $O(n)$ as well as a space complexity of $O(n^2)$, where a global constraint implementation of the `alldiff` constraint could be implemented using $O(n)$ space and $O(1)$ time.

In this work we propose a *compositional* approach for modeling of global constraints themselves, which gives us a high-level tool for *invention* and *modification* of global constraints. This process is very common in real-life applications, where the provided global constraints often do not fit the problem exactly. In our approach, we quantify global constraints over key properties of their structure. To do this we use a *generic graph model*, an approach which has previously been used successfully to model a large number of global constraints [2]. The main benefit of a quantified model of global constraints is that practitioners can very easily experiment with different cost functions and different structures, and also compose completely new global constraint with a minimum of work. Also, composition of constraints is not only simple to model, but the resulting constraints are evaluated very efficiently using highly-optimized incremental algorithms. The user does not have to care about keeping costs and conflict levels updated, which is handled by the *Composer* system automatically.

The implementation of *Composer* as a template library has two main purposes. First, it provides the generality needed for such a component - it is easy to extend the library with more objects which will interact with the system, and second, it maintains the efficiency that is absolutely crucial in local search constraint solving. Using templates allows us to inline and eliminate function calls that would otherwise affect performance substantially. To use inheritance for component construction would require a virtual method call for each component evaluation, something that could be fatal for the solver performance. For example, in a single typical iteration it is not uncommon that a filter constraint is evaluated 10,000 times. Because the nature of the filter is to compute a simple expression, the actual virtual method call overhead would be a larger amount of the total execution time than the filter evaluation itself. This is of course unacceptable in a local search library, where main focus is on speed of evaluation. In addition, templates makes inlining of these crucial method calls possible, something that further increase the performance of our method.

The rest of the article is organized as follows. First, Sect. 3 gives an introduction to constraint composition in *Composer*. Section 2 describes the basic architecture of the *Composer* local search system, and Sect. 4 and 5 presents two example applications, showing that constraint composition is indeed useful in practice. Section 6 gives some concluding remarks.

2 The *Composer* Architecture

The basis of *Composer* is *incremental evaluation* of basic expressions of the form

```
X = f(Y1,Y2,Y3,...,Yn)
```

where `f` is a generic function and `Y1,Y2,...,Yn` are integer variables. A resulting acyclic graph of functions on the form above has been shown previously to be wellsuited to incremental evaluation [10, 1, 6]. In general, we assume that there exist an efficient algorithm to compute `X` once a single input variable `Yi` has changed. *Composer* provide many such functions useful for expressing relationship between variables in the problem model. Also, observe that the expression `X = f()` denotes that `X` should take the constant value computed by `f` (which take no arguments). More complex expressions can also be used and will be flattened into basic expressions on the form above. Only acyclic expressions are allowed. All basic expressions, constraints and variables as well as modifications are posted into a *model*, which keeps track of expression evaluation and updates, and also provides support methods for local search.

In *Composer* we differ between *state* and *non-state* variables. A state variable is subject to change from the search component. Non-state variables are simply entities whose values are entirely dependent on the values of the state variables in the model. In any incremental expression, the left-hand side variable must be a non-state variable, and the right-hand variables can be either state or non-state variables. The reason to differ between state and non-state variables is that *Composer* automatically computes a *conflict level* for each state variable `X`, indicating in how many conflicts `X` is present.

As an example of an incremental function, *Composer* provide the `vecOp` operator, which applies a given operator $\oplus$ on all its arguments $\mathtt{Y1}, \mathtt{Y2}, \ldots, \mathtt{Yn}$ and gives the result $\mathtt{Y1} \oplus \mathtt{Y2} \oplus \ldots \oplus \mathtt{Yn}$. The folding of any such associative and commutative binary operator $\oplus$ over a set of variables can be incrementally evaluated in time $O(f_{\oplus} + f_{\oplus^{-1}})$ where $f_{\oplus}$ and $f_{\oplus^{-1}}$ is the time of computing $r = x \oplus y$ and $x = r \oplus^{-1} y$ respectively. For many common operators, such as addition and multiplication, this time is $O(1)$ giving such functions nice incremental properties. For other functions such as `max`, `argmax` and relatives, *Composer* provides efficient $O(\lg n)$ incremental implementations.

The usefulness of incremental expression can be shown by an example. In train scheduling [5] a train traversing a sequence of stations can be elegantly modeled using variables denoting departure d_i from station i, traversal time from station i to station $i+1$, arrival a_{i+1} at station $i+1$, and stop time s_{i+1} at station $i+1$, and so on. However, there exists a very tight relationship between these variables. Consider the case where the traversal time is known in advance and constant. We get the following equations.

$$\forall i > 1 : d_i = a_{i-1} + s_{i-1}$$
$$\forall i . a_i = d_i + t_i$$

Because t_i is constant, we get a single departure variable d_1 and a set of stop time variables s_i on which all the other variables depend. Using incremental expressions, these relationships can be declaratively modeled and automatically used. The *Composer* system takes care of the handling of the dependent variables and the distribution of conflicts, so that state variables automatically gets assigned a proper conflict level. The relationships above can be modeled in *Composer* as follows.

```
a[0] = d[0] + t[0] ;
for(uint i=1;i<k;i++) {
  d[i] = a[i-1] + s[i-1] ;
  a[i] = d[i]   + t[i] ;
}
```

The incremental back-bone of *Composer* also gives the global constraints an environment which helps in modeling many combinatorial problems. In fact, composed global constraints are compiled into special incremental functions and are fully integrated in the incremental framework of *Composer*. Because of this it also becomes very easy to post global constraints on non-state variables, which are efficiently computed by the system during run-time. For example, consider the train scheduling application above. We can for example impose that all trains on a single track should be serialized in time, which can be done as follows.

```
vector<vector<uint> > start(Tracks);
vector<vector<int> >  dura(Tracks);
for(uint i=0;i<Traversals;i++) {
  start[track[i]].push_back(d[i]);
  dura[track[i]].push_back(t[i]);
}
for(uint i=0;i<Tracks;i++)
  serialized(model,start[i],dura[i]);
```

The code fragment above first creates two vectors for each track i, one with the start time and the other with the duration of all traversals on track i. The, a `serialized` constraint is posted on the tasks represented by the start and duration of the traversals on that track. Observe that `start[i]` is composed of non-state variables as shown in the previous example. *Composer* will now automatically assign a violation level to the state variables that `start[i]` depends on, making guided local search possible.

3 Constraint Composition Using *Composer*

In this section we will discuss how to compose global constraints using the *Composer* template library. In the *Composer* model, global constraints are quantified over its *graph structure*, its *filter constraint*, and its *cost*. A global constraint is composed by selecting three components corresponding to the quantified properties above. The structure component uses the variables and parameters of the global constraint to construct an *arc set*. Although we often use binary arcs and constraints in *Composer* we are not limited to this and can use any uniform arity we want.

The filter component is then used to select a final set of *critical arcs* – the arcs that satisfy the filter constraint. We can then apply our *cost component* to the set of critical arcs to form the total cost of the constraint. The cost components themselves are also composed of a *property* and a *cost function*. The property samples a trait of the arc set, returning an integer, and the cost function is then applied to the integer, giving us the full cost of the constraint.

As an example of how to compose a constraint using *Composer*, we now consider the binarization of the `alldiff` constraint as shown in Fig.1. The `typedef`:s are used to handle the verbose types resulting from using templates in C++. The first type definition provides a shortcut for the cost function, in this case constructed by using the `AtmostC<0>` cost function on the `Card` property, the cardinality of the edges produced by the structure. The cost obtained from `AtmostC`<k> is $\max(0, n-k)$ where

```
typedef Card<AtmostC<0> > Cost;
typedef AllUnordBin<VarEqVar,Cost > AllDiff;

AllDiff * con = new AllDiff(X,&model);
con->setCost(Cost(con));
con->setFilter(VarEqVar(con));
```

Fig. 1. The primitive `alldiff` constraint.

n is in this case the cardinality of the arc set. *Composer* provides many cost functions and properties, although we will mostly see variations of the ones above.

The second type definition states that the `AllDiff` type is a composition of the `AllUnordBin` graph structure, a `VarEqVar` filter, and the previously specified cost component `Cost`. The `AllUnordBin` component forms arcs between all subsets $\{x,y\}$ of variables, and the `VarEqVar` constraint filters away all edges for which the variables have disjoint values. The rest of the code instantiates the constraint with the composition classes. In this case we need to tell that the model in where the constraints should be posted is `model` and that the variables of the constraints are in `X`. The constraint above will act exactly as the binarization of the `alldiff` constraint – *Composer* will automatically compute the constraint cost and the conflict level of each present variable. In this case the `Card` component will compute a conflict level $c(\mathtt{X})$ for a variable `X` and a critical arc set A as $c(\mathtt{X}) = \frac{2v|S|}{|A|}$ where $S = \{(\mathtt{X},y) \in A\} \cup \{(z,\mathtt{X}) \in A\}$ and v is the cost of the constraint.

Although the model for `alldiff` shown in Fig.1 is brief and natural, it has one drawback. It is essentially a decomposition of the constraint into binary inequalities, and the space complexity of $O(n^2)$ and incremental time complexity of $O(b \cdot n)$, where b is the number of variable changes, is naturally the same as for the decomposition. We can however do better than this, given that a natural implementation of the `alldiff` constraint would have a space complexity of $O(n+v)$ and an incremental time complexity of $O(b)$, where v is the number of values for all variables. In *Composer* we can use the concept of *subcosts* to partition the cost of a constraint into smaller, more manageable parts. A subcost is simply a cost computed on a *subset* of the final graph. Using all subcosts of the model we can form the total cost of the constraint by a *combination* operator. At present we require that the combination operator must be a commutative and associative binary operator. Typically we use addition for this purpose.

We have found that using subcosts on all edges entering or leaving a subset of the nodes of the graph is often useful as we will see later. We therefore provide the two subcosts `Entering` and `Leaving`, which each take another cost component C as a template argument. The subcost components apply C on each subset A_i of edges entering or leaving a single node respectively, and compute a total cost by addition.

For the `alldiff` constraint a natural representation is a bipartite graph with the variables as one node partition and the possible values as the other one. The edges of the graph corresponds to an assignment of a variable to a value. In such a model we can update the cost very efficiently by keeping track of how many variables are connected to a single value. In *Composer* we can model the `alldiff` constraint as an efficient bipartite structure with subcosts as shown below. The composition of the `alldiff` constraint below uses $O(n)$ space and $O(1)$ time for a single variable change.

```
typedef Entering<Card<AtmostC<0> > > Cost;
typedef Bipartite<VarEqC,Cost>     AllDiff;

AllDiff *con = new AllDiff(X,&model);
Card<QuadAtmostC<1> > subcost(con,QuadAtmostC<1>());
con->setCost(Cost(con,subcost));
con->setFilter(VarEqC(con));
```

Fig. 2. A bipartite `alldiff` constraint with subcosts.

4 The *n*-Queens Problem

In this section we will investigate a model and algorithm for solving the *n*-queens problem, a classical combinatorial puzzle that has been used extensively as an example in the constraints community. In this problem, n queens are to be placed on a chessboard of dimensions $n \times n$ so that no queen can be attacked by another queen. The simplest instance of the *n*-queens problem with a solution is the 4-queens problem.

4.1 A Constraint Model

```
Model model;
statevector queen(Queens,"Queen_",0,Queens-1);
vector<uint> X(fromto(0,Queens-1));
vector<int> pos(fromto(0,Queens-1)), neg(fromto(Queens-1,0));
alldiff(model,&X);
alldiff(model,&X,pos);
alldiff(model,&X,neg);
model.randomize();
model.close();
```

Fig. 3. A first model of the *n*-queens problem.

We have solved the *n*-queens problem in two different ways – one using a simple 1-assignment neighborhood, and one using a more elaborate 2-swap neighborhood. Both models are similar in that each queen is represented by a state variable with domain $0..n-1$. In the first model, we use three `alldiff` constraints to model that no two queens on the board may attach each other as shown in Fig.3. The constraint

```
alldiff(model,&X,M)
```

states that for all variable pairs (x_i, x_j) in X where $i \neq j$, the constraint $x_i + m_i \neq x_j + m_j$ should hold where $m_i, m_j \in$ M. The *Composer* system automatically computes a cost and a conflict level for each composed constraint.

```
typedef Entering<Card<QuadAtmostC<1> > > Cost;
typedef Bipartite<VarPlusEqC,Cost> AllDiff;

AllDiff *con = new AllDiff(X,&model);
Card<QuadAtmost<1> > subcost(con,QuadAtmostC<1>());
con->setCost(Cost(con,subcost));
con->setFilter(VarPlusEqC(con,M));
```

Fig. 4. A bipartite `alldiff` constraint with an offset vector M.

A generic `alldiff` constraint with an offset vector is composed in *Composer* as shown in Fig.4. The difference from the `alldiff` constraint in Fig.2 is highlighted, and is simply a replacement of the trivial equality constraint $v(X) = k$ with an equality constraint with offset, $v(X) + c(X) = k$. This shows quite clearly the expressiveness of the global constraint composition approach of *Composer*. This new composition of the `alldiff` constraint also uses $O(n)$ space and $O(1)$ time for a single variable change.

4.2 A First Local Search Algorithm

The local search algorithm used for the first model of the n-queens problem is shown in Fig.5. Line 1 declares place-holders for a variable and a value, Q and V. The next line states that the search should continue until no constraints are violated. Remember that *Composer* will update the constraint violations and conflict levels automatically using incremental algorithms. The next line

```
Q = model.getMaxConflicting();
```

fetches a random variable with maximum conflicts, and the line

```
V = model.getMinCostValue(Q);
```

finds the value V for Q which minimizes the violation of assigning V to Q. Finally,

```
model.assign(Q,V);
```

places queen Q on row V and updates all conflict levels and the violation of `model`.

4.3 Results

In Table 1 we show the best results obtained by running n-queens 11 times for each instance using our first model on a 1 GHz Pentium III running Linux kernel 2.4.20. The results are clearly competitive with those found in [7] and [3]. It is important to keep in mind that *Composer* is a fully generic global constraint composition library – in fact, *Composer* maintains a lot of data structures that are necessary to keep this flexibility. The reader should keep this in mind when comparing the results. We noted that the search was prone to becoming trapped in local minima and was not very robust in terms of consistency of results, which led us to try another method of solving the problem.

```
int Q,V;
while(model.violation()!=0) {
  Q = model.getMaxConflicting();
  V = model.getMinCostValue(Q);
  model.invAssign(Q,V);
}
```

Fig. 5. Search component for the n-queens puzzle using assignment transitions.

Size n	Init (s)	Solve (s)	Iter.
8	0.06	0.06	30
16	0.05	0.02	465
32	0.05	0.02	152
64	0.06	0.04	218
128	0.06	0.05	193
256	0.08	0.21	368
512	0.08	0.87	671
1024	0.10	2.14	834
2048	0.12	7.81	1346
4096	0.21	27.35	2336
8192	0.36	100.39	4259
16384	0.76	386.06	8179

Table 1. Experimental results of 11 runs of the n-Queens Problem.

4.4 A Swap-based Local Search Algorithm

For the second algorithm we decided to use variable swaps as the local transition instead of assignments. This approach has previously been shown to be very successful for several hard combinatorial problems [3, 4, 8]. If we initialize the queens as a random permutation of 0..`Queens` − 1 and always use swaps between two variables as the transition, it is easy to see that the first `alldiff` constraint becomes redundant since it will be satisfied in all visited states. Thus, the second model only differs from the first in that we remove the first constraint, and use a random permutation instead of the fully random initialization.

```
int Qi,Qj,oldV,iter;
Tabuvector tabu(model.arity(),&iter);
for(iter=0; model.violation()!=0; iter++) {
  Qi = model.getMaxConflicting(tabu);
  Qj = model.getMinCostSwapWith(Qi,tabu);
  oldCost = model.violation();
  model.swap(Qi,Qj);
  if(model.violation() >= oldCost)
    tabu.insert(Qi,Tenure);
}
```

Fig. 6. Search component for the n-queens puzzle using value swaps.

In the new algorithm, in each iteration we first select at random one of the queens s with maximum number of conflicts. We then find the set of queens t which minimizes the resulting cost of swapping queen s and t. We then randomly select one of these queens and swap the positions of s and t. We also use a tabu component from *Composer* to diversify the search on plateaus and to escape local minima. In this case the tabu component consists of a vector T of size n in which element T_i stores the iteration count when queen i was marked as tabu. This arrangement makes it possible to check if a queen q_i is tabu in constant time, simply by checking the current iteration count against T_i.

The code for solving the second model is shown in Fig.6. The line

```
Tabuvector tabu(model.arity(),&iter);
```

declares a tabu vector with size equal to the arity of the problem, which use the integer variable `iter` as the iteration count. The next line

```
for(iter=0; model.violation()!=0; iter++) {
```

initializes the iteration count, loops until the violation of the constraints in `model` is zero, and increases the iteration count by one each iteration. The lines

```
Qi = model.getMaxConflicting(tabu);
Qj = model.getMinCostSwapWith(Qi,tabu);
```

first fetches a randomly chosen variable `Qi` with maximum number of conflicts, disregarding the variables which are tabu in `tabu`, and then selects a second random variable `Qj` for which swapping `Qi` and `Qj` yields the least violation possible, once again disregarding variables declared as tabu. Finally, the lines

```
oldV = model.violation();
model.swap(Qi,Qj);
if(model.violation() >= oldV)
  tabu.insert(Qi,Tenure);
```

first saves the previous violation of the model, and then commits to the value swap between `Qi` and `Qj`. If the new violation is greater or equal to the previous one, we then declare `Qi` tabu. We use a tabu tenure of 10 which we fixed with minor experimentation.

Previous work on the n-queens puzzle has been successful in using a first-descent strategy instead of a conflict minimizing one [8]. We therefore modified our search procedure to select an improvement immediately if one existed. The new procedure is equal to the one shown in Fig.6 except that we use the line

```
Qj = model.getImprovingSwapWith(Qi,tabu);
```

to select a second queen. This method works exactly as the one above, except that as soon as a swap which improves the current violation is found, the method returns with this one immediately.

	Composer				Adaptive search	
Size n	Init (s)	Solve (s)	Total (s)	Iter.	Solve (s)	Iter.
8	0.03	0.00	0.03	2	0.00	6
16	0.03	0.00	0.03	11	0.00	9
32	0.04	0.00	0.04	40	0.00	14
64	0.04	0.00	0.04	25	0.00	22
128	0.04	0.00	0.04	29	0.00	36
256	0.04	0.01	0.05	54	0.01	63
512	0.04	0.04	0.09	121	0.06	116
1024	0.06	0.16	0.22	213	0.22	216
2048	0.08	0.84	0.93	421	0.82	409
4096	0.15	4.48	4.62	806	3.23	805
8192	0.27	20.82	21.09	1593	13.12	1577
16384	0.52	89.96	90.48	3153	59.25	3118
32768	1.01	371.89	372.9	6279	276.09	6215

Table 2. Experimental results of 11 runs of the n-Queens Problem using the second model and conflict minimization.

4.5 Results

The results of the swap-based steepest-descent algorithm is shown in Table 2 together with results obtained by using the *Adaptive Search* algorithm from [3]. The computer used for both methods was a 1 GHz Pentium III running Linux kernel 2.4.20. The results reported are median results of 11 runs. As can be seen, the results show that using a generic framework for global constraints is not only feasible but also highly competitive with those results previously obtained in [7] and [3]. The results of the modified first-improvement algorithm shown in Table 3 also support this. Note that the number of iterations has increased but the runtime of the new algorithm is generally smaller than the conflict minimizing algorithm. Also, keep in mind that *Adaptive Search* uses a low-level encoding of the problem in contrast with *Composer*.

5 The Progressive Party Problem

The progressive party problem is a well-known benchmark problem in the constraint programming community, and have been used in several generic constraint-based local search methods as well [9, 4, 7]. The problem can be described informally as follows. An evening party is to be organized in the setting of a yachting rally. The organizers have decided that the visiting boats will visit the organizers' boats (the *host boats*) in turn, where the crew of the host boat stays aboard and serves the guests on the *guest boat*. Every 30 minutes, each guest boat will move to a different host boat. This will go on for a given number of time periods. The organizers have also decided that no guest boat have to visit the same host boat twice, and that two crews must never meet more than once. In Table 4 the different boats and their crews and capacities are shown. We also show the *spare capacities* of each boat, which is simply the number of guests that can safely be taken aboard.

Size n	*Composer*			
	Init (s)	Solve (s)	Total (s)	Iter.
8	0.02	0.00		4
16	0.02	0.00		8
32	0.02	0.00		59
64	0.02	0.00		30
128	0.02	0.00		65
256	0.02	0.00		119
512	0.02	0.01		253
1024	0.03	0.02		498
2048	0.04	0.10		1014
4096	0.07	0.39		1975
8192	0.13	1.51		3975
16384	0.25	7.35		7942
32768	0.51	35.74		15906

Table 3. Experimental results of 11 runs of the n-Queens Problem using the second model and conflict improvement.

i	k_i	c_i	s_i	i	K_i	c_i	s_i	i	k_i	c_i	s_i	i	K_i	c_i	s_i
1	6	2	4	12	10	3	7	23	7	4	3	33	6	2	4
2	8	2	6	13	8	4	4	24	7	4	3	34	6	2	4
3	12	2	10	14	8	2	6	25	7	2	5	35	6	2	4
4	12	2	10	15	8	3	5	26	7	2	5	36	6	2	4
5	12	4	8	16	12	6	6	27	7	4	3	37	6	4	2
6	12	4	8	17	8	2	6	28	7	5	2	38	6	5	1
7	12	4	8	18	8	2	6	29	6	2	4	39	9	7	2
8	10	1	9	19	8	4	4	30	6	4	2	40	0	2	-2
9	10	2	8	20	8	2	6	31	6	2	4	41	0	3	-3
10	10	2	8	21	8	4	4	32	6	2	4	42	0	4	-4
11	10	2	8	22	8	5	3								

Table 4. Configuration of the Progressive Party Problem. Each boat i has a total capacity k_i (the total number of people allowed on board at the same time), and a crew size c_i. The spare capacity s_i of a boat is formed by subtracting the crew size from the total capacity, $s_i = k_i - c_i$.

5.1 The Model

We have solved the progressive party problem using the *Composer* system and composed global constraints. The problem instances we have considered are listed in Table 5, where the first 6 instances are well-known and previously studied in a local search context [7, 4]. Instance 7 to 11 are new and has not been investigated as far as we know. We will see that these instances become increasingly harder to solve. Also, we have tried to solve the progressive party problem for as many time periods as possible, making the problem increasingly hard.

The model we have used is shown in Fig.7. We use a matrix of states `boat` of dimension $p \times g$, where p is the number of periods to run the problem, and g is the number of guests available. `boat(i, j)` represent which host boat guest `j` is assigned

Instance	Host boats	h	g	ratio
1	1-12,16	100	92	0.92
2	1-13 (orig.)	98	94	0.96
3	1,3-13,19	96	92	0.96
4	3-13,25,26	98	94	0.96
5	1-11,19,21	95	93	0.98
6	1-9,16-19	93	91	0.98
7	1-11,21,22	94	92	0.98
8	1-9,16-18,22	92	90	0.98
9	1-9,15-17,22	91	89	0.98
10	1-11,22,23	93	92	0.99
11	1,3-11,21-23	91	90	0.99

Table 5. Progressive Party Problem analysis for different host configurations.

to in period `i`. We also use the auxiliary vectors `size` which hold the crew size of the guests, and `spare` which hold the spare capacities of the host boats. The model consists of three types of composed global constraints. First, the `capa` constraint states that for all time periods, the capacity of the host boats should not be exceeded. The `capa` constraint is actually a restricted version of a general weighted cardinality constraint (which is also easily expressed using *Composer*), and is shown in Fig.8. This constraint is equivalent with the inequalities

$$\forall v. \sum_{\mathtt{x}=v} w_{\mathtt{x}} \leq c_v$$

The `nbdiff` constraint states that in the two vectors `X` and `Y`, at most $k = 1$ pairs X_i, Y_i may be equal. The constraint is equivalent to the following.

$$|S| \leq k, S = \{i \mid \mathtt{X}[i] = \mathtt{Y}[i]\}$$

```
statematrix boat(Periods,Guests);
vector<uint> X(Periods),Y(Periods);
for(int p=0;p<Periods;p++)  capa(model,boat(p),size,spare);
for(int i=0;i<Guests;i++) {
  for(int j=0;j<Periods;j++)  X[j] = boat(j,i);
  alldiff(model,&X);
}
for(int i=0;i<Guests;i++) {
  for(int j=i+1;j<Guests;j++) {
    for(int k=0;k<Periods;k++)  X[k]=boat(k,i), Y[k]=boat(k,j);
    nbdiff(model,&X,&Y,1);
  }
}
```

Fig. 7. A model of the progressive party problem.

```
typedef Entering<WSum<VarAtmost,WVector> > Cost;
typedef Bipartite<VarEqC,Cost>             Capa;
Capa *con = new Capa(X,&model);
WSum<VarAtmost,WVector> subcost(con,VarAtmost(K),C);
con->setCost(Cost(con,subcost));
con->setFilter(VarEqC(con));
```

Fig. 8. A bipartite `capa` constraint with subcosts.

5.2 A Local-Search Algorithm

```
int G,H,iter,bestsofar,oldV
Tabumatrix(model.arity(),Hosts,&iter,&bestsofar);
for(iter=0; model.violation()!=0; iter++) {
  oldV = model.violation();
  G = model.getMaxConflicting();
  H = model.getMinCostValue(tabu);
  model.assign(G,H); tabu.insert(G,H,Tenure);
  if(model.violation() < oldV && Tenure > 2) Tenure --;
  if(model.violation() >= oldV && Tenure < TenureLimit)   Tenure ++;
  if(model.violation() < bestsofar) {
    model.hist_clear(); model.hist_store();  nonimproving=0;
  } else {
    nonimproving++;
    if(model.violation()==bestsofar) model.hist_store();
  }
  if(nonimproving>Backtrack) model.hist_restoreProportional()
  if(iter>MaxTries) model.randomize();
}
```

Fig. 9. The search component for the progressive party problem.

The local search procedure used is shown in Fig.9. In short, we use a simple hill-climbing modification in which the best move is always selected, and where ties are broken randomly. As before we use the single-assign neighborhood and use a tabu structure, this time over the variables *and* values, to diverge the search. As parameters we use a tabu tenure of initially 2. This tenure is increased or decreased as the search proceeds in order to not hinder the exploration of new states. We also use a limited backtracking mechanism to improve performance on very hard problem instances. Every time an new best state is found, a state memory is cleared and initialized by the line

```
model.hist_clear(); model.hist_store();
```

The state history component also keeps track of the frequency of the states, and whenever `Backtrack` transitions have been done, a state in the state memory is chosen with probability inversely proportional to the frequency of the state, and the search backtracks to this state. This will ensure that the search restarts from a good state and also that all currently best states are kept in memory for future backtracking reference.

To improve performance even more, we also use a restarting mechanism which restarts the search from scratch whenever `MaxTries` iterations have passed. We added this feature when we noted that the search got stranded in areas where no improvement seemed possible. With the restarting mechanism, we were able to get the local search to terminate with a solution in every sample run of every instances of the problem tested.

5.3 Results

Hosts/Periods	1	2	3	4	5	6	7	8	9	10
1-12, 16	0.01	0.02	0.02	0.04	0.06	0.10	0.18	0.31	0.69	5.23
1-13	0.01	0.03	0.05	0.09	0.16	0.27	0.58	2.51	14.9	
1, 3-13, 19	0.01	0.03	0.05	0.10	0.15	0.27	0.57	2.28	22.2	
3-13, 25, 26	0.01	0.03	0.06	0.11	0.19	0.31	0.66	2.52	20.8	
1-11, 19, 21	0.02	0.05	0.11	0.20	0.40	1.14	6.94			
1-9, 16-19	0.02	0.06	0.13	0.26	0.55	1.81	18.3			
1-11, 21, 22	0.02	0.05	0.11	0.22	0.47	1.41	46.1			
1-9, 16-18, 22	0.02	0.06	0.13	0.24	0.61	2.26	118			
1-9, 15-17, 22	0.02	0.06	0.14	0.31	1.11	49.5				
1-11, 22, 23	0.04	0.20	0.51	4.46						
1, 3-11, 21-23	0.04	0.18	0.62	3.03						

Table 6. Median CPU time of 101 runs of the Progressive Party Problem.

Hosts/Periods	1	2	3	4	5	6	7	8	9	10
1-12, 16	14	37	62	103	165	256	456	832	1862	13248
1-13	26	72	135	249	430	729	1572	6634	37226	
1, 3-13, 19	25	70	144	266	405	736	1570	6206	55262	
3-13, 25, 26	34	91	164	284	508	841	1812	6609	51681	
1-11, 19, 21	52	148	300	559	1143	3256	17853			
1-9, 16-19	53	156	372	735	1540	4927	46298			
1-11, 21, 22	50	138	298	595	1332	3929	116239			
1-9, 16-18, 22	47	169	348	665	1765	6511	291119			
1-9, 15-17, 22	50	165	398	885	3162	124017				
1-11, 22, 23	104	570	1479	11532						
1, 3-11, 21-23	110	543	1780	8090						

Table 7. Median iterations of 101 runs of the Progressive Party Problem.

The problem was solved on a 1 GHz Pentium III running Linux kernel 2.4.20. The resulting median CPU times and iterations of 101 sample runs of PPP are shown in Table 6 and 7. For the first 6 host configurations we see a clear improvement over the results reported in [9, 4, 7] using similar neighborhoods. It is important to note that the local search algorithm does not differ in any substantial way from those used in [4, 7], which clearly shows the feasibility of high-level constraint modeling using global-constraint composition.

6 Conclusions

In this paper we have proposed a compositional approach for global constraint modeling and implementation, and shown that such an approach is not only feasible but also highly competitive with existing low-level, non-compositional approaches for constraint solving using local search. The main point of a compositional approach for constraint modeling is that it allows great flexibility in experimentation with modification and creation of new constraints at a very low run-time overhead. In compositional constraint design, global constraints are quantified over three main properties of their structure; their *graph structure*, their *filter constraint* and their *cost components*. Using this quantification we have composed several useful global constraints using *Composer*. We have also integrated the global constraint composition mechanism into a generic local search library, *Composer*, which provide much help in designing local search algorithms for combinatorial problems. We also showed that a compositional approach is very competitive with existing methods on two hard problems.

References

1. Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, San Francisco, California, January 1990.
2. Nicolas Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Principles and Practice of Constraint Programming*, pages 52–66, 2000.
3. Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In *SAGA*, pages 73–90, 2001.
4. Philippe Galinier and Jin-Kao Hao. A general approach for constraint solving by local search. In *Proc. CP-AI-OR'00*, Paderborn, Germany, March 2000.
5. Per Kreuger, Mats Carlsson, Thomas Sjöland, and Emil Åström. Sequence dependent task extensions for trip scheduling. Technical Report ISRN:SICS-T–2001/14-SE, Swedish Institute of Computer Science, May 2001.
6. Laurent Michel and Pascal Van Hentenryck. Localizer: A modeling language for local search. In *Principles and Practice of Constraint Programming*, pages 237–251, 1997.
7. Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, November 2002.
8. R. Sosic and J. Gu. 3,000,000 queens in less than one minute. In *SIGART Bulletin*, volume 2,2, pages 22–24, April 1991.
9. Joachim Paul Walser. *Integer Optimization by Local Search*, volume 1637 of *Lecture Notes in Artificial Intelligence*. Springer, 1999.
10. Daniel M. Yellin and Robert E. Strom. Inc: a language for incremental computations. *ACM Trans. Program. Lang. Syst.*, 13(2):211–236, 1991.