Synthesis of Constraint-Based Local Search Algorithms from High-Level Models*

Pascal Van Hentenryck
Brown University

Laurent Michel
University of Connecticut

Abstract

The gap in automation between MIP/SAT solvers and those for constraint programming and constraint-based local search hinders experimentation and adoption of these technologies and slows down scientific progress. This paper addresses this important issue: It shows how effective local search procedures can be automatically synthesized from models expressed in a rich constraint language. The synthesizer analyzes the model and derives the local search algorithm for a specific meta-heuristic by exploiting the structure of the model and the constraint semantics. Experimental results suggest that the synthesized procedures only induce a small loss in efficiency on a variety of realistic applications in sequencing, resource allocation, and facility location.

Introduction

Ease of use is becoming increasingly prominent in constraint satisfaction and optimization. Indeed, in contrast to MIP and SAT solvers, most existing tools for constraint programming (CP) and constraint-based local search (CBLS) require significant user expertise and, in particular, the programming of effective search procedures. Puget (2004) cites simplicity of ease as the next challenge for CP and advocates black-box systems. Gomes and Selman (2006) suggest that CP systems such as GECODE and CBLS systems such as COMET be also available in black-box versions. It is thus a fundamental challenge to transfer the degree of automation traditionally found in MIP/SAT solvers to the rich languages of constraint programming and constraint-based local search.

This paper represents a step in this direction: It shows how effective local search algorithms can be synthesized from high-level models, exploiting the structure of the models and the semantics of the constraints and objectives. The input of a CBLS synthesizer is a model in a constraint-based language in which constraints and objectives capture combinatorial substructures of the application. Moreover, the modeling language offers the ability to specify both hard and soft constraints, explicitly recognizing the distinct roles of constraints in many applications. Given a model, a CBLS synthesizer derives a local search (LS) algorithm for a specific meta-heuristic in two steps. It first analyzes the model and the instance data and extracts its structure. The synthesizer then derives the neighborhood, as well as any other component required by the meta-heuristic (e.g., a diversification in tabu search). The synthesizer may also perform Lagrangian relaxations for models involving (hard and soft) constraints and an objective function; it can rewrite an optimization application into a sequence of decision problems. The synthesizer also exploits the distinction between hard and soft constraints during the derivation. In particular, the same model, with different choices of hard and soft constraints, will lead to local search algorithms with fundamentally different neighborhoods.

To assess the practicability of the approach, CBLS synthesizers were implemented on top of COMET and evaluated on a variety of applications including car sequencing, resource allocation, and facility location. The experimental results show that, on these applications, effective LS algorithms can be synthesized from high-level models, inducing only a small overhead over state-of-the-art procedures.

The main contributions of this research are as follows:

1. It demonstrates how effective local search algorithms can be synthesized from concise and natural high-level models for a variety of realistic applications.
2. It shows how the synthesizers exploit the constraint semantics to derive the neighborhoods.
3. It shows how the constraint roles (their hard/soft statuses) drive the synthesis to different local search algorithms.
4. It indicates that, on these applications, the synthesis process only induces a small loss in performance compared to *tailored* search procedures.

The rest of this paper presents high-level models for a wide variety of applications, discusses the synthesis process for each of them, shows skeletons of the synthesized search algorithms whenever appropriate, and reports experimental results.

Satisfaction Problems

Getting Started Figure 1 illustrates the approach on the n-queens problem. Lines 2–7 specify the model: they declare the decision variables (line 3) and the *soft* combinatorial constraints specifying that the queens cannot be placed

*Partially supported by NSF awards DMI-0600384 and IIS-0642906 and ONR Award N000140610607.

```
1.  range Size = 1..8;
2.  model m {
3.      var{int} queen[Size](Size);
4.      soft: alldifferent(queen);
5.      soft: alldifferent(all(i in Size) (queen[i] + i));
6.      soft: alldifferent(all(i in Size) (queen[i] - i));
7.  }
8.  TabuSearch search(m);
9.  search.apply();
```

Figure 1: A Model For The Queens Problem.

```
1.  int n = 15;
2.  model m {
3.      var{int} v[1..n](0..n-1);
4.      hard: alldifferent(v);
5.      soft: alldifferent(all(k in 1..n-1) abs(v[k+1]-v[k]));
6.  }
7.  MinConflictSearch search(m); search.apply();
```

Figure 2: A Model For The All-Interval Series.

on the same row and diagonals. The model m is used by the tabu-search synthesizer to derive a search procedure (line 8) which then look for a feasible solution (line 9).

Since all the constraints are soft, the tabu search minimizes the sum of the constraint violations and the derived neighborhood consists of assigning a value to a variable. By default, the tabu search first selects the most violated variable and assigns the value that is not tabu and minimizes the overall violations. We illustrate later how more complex tabu searches are derived for richer models. Note the complete separation of modeling and search. The tabu-search synthesizer only receives a model containing decision variables, constraints, and (possibly) an objective function. It includes restarts, intensification (i.e., returning to the best solution after a number of iterations with no improvement), and diversification (e.g., randomly assigning some variables). Observe also that line 9 can be replaced by

```
9.  MinConflictSearch search(m);
```

in which case a min-conflict search, with the same neighborhood, is synthesized.

All Interval Series Figure 2 depicts a concise model for the all-interval series, a benchmark often used to evaluate SAT solvers. The problem consists of finding a series $(v_1, \ldots, v_n)$ such that all v_i are different and the differences $(v_{k+1} - v_k)$ $(1 \leq k < n-1)$ are also all distinct. The model expresses these two constraints directly in lines 4–5. It also states that the alldifferent constraint on the series is hard, while the constraint on the differences is soft.

The derived min-conflict search is driven by the hard constraint. The synthesizer analyzes the model, deducing that

1. all variables appear in the hard constraint h;
2. all variables have the same domain;
3. the hard constraint is an alldifferent constraint;
4. the alldifferent is tight (bijection from variables to values).

Since the model is initialized with an assignment satisfying constraint h, fact (3) ensures that swapping two variables from h maintains its feasibility. Facts (1) and (4) guarantee that these swaps provide a connected neighborhood: Every solution can be reached from any assignment using these moves. Fact (2) ensures that all these swaps satisfies the domain constraints. The initialization of a hard alldifferent constraint, at the beginning of the search or during a restart, amounts to finding a (random) maximal matching in the bipartite variable-value graph of the constraint. Figure 3 depicts part of the search procedure synthesized for this model. The hard constraint and its variables are retrieved from the model (lines 1–2). The search then consists of a number of restarts (line 5), each consisting of a maximum number of stable iterations (line 7). The neighborhood consists of swapping variables occurring in the hard constraint. The move is selected in two steps by selecting first a variable x[c] in conflict (line 8) and then choosing another variable x[v] which, when swapped with x[c], decreases the violations the most (line 9). The swap is performed in line 10. When restarting, the model is re-initialized (line 14), finding a new feasible solution for the hard constraint. The skeleton omits the maintenance of the best solution, and the intensification and diversification for brevity. Finding a solution for a series of size 15 typically takes 0.1 second.

```
1.  Constraint hardCstr = m.getHard();
2.  var{int}[] x = hardCstr.getVariables();
3.  range X = x.getRange();
4.  ...
5.  while (restarts < maxRestarts && !found) {
6.     int it = 0;
7.     while (it++ < maxIterations && !found) {
8.        select(c in X: S.violations(x[c]) > 0)
9.           selectMin(v in X)(S.getSwapDelta(x[c],x[v]))
10.             x[c] :=: x[v];
11.       ...
12.    }
13.    ...
14.    m.initialize();
15. }
```

Figure 3: The Derived Skeleton For the All-Interval Series.

It is worth mentioning that traditional CBLS algorithms for this problem (e.g., in (Van Hentenryck & Michel 2005)) omit the hard constraint, whose feasibility is implicitly maintained by the local moves. By making it explicit, the model cleanly separates the constraints (*what a solution is*) from their roles (*how to use them during the search*). As a consequence, modelers can easily explore different neighborhoods by changing the constraint roles. For instance, a modeler may turn the first alldifferent into a soft constraint. This (less effective) synthesized procedure chooses a variable in conflict but now assigns to the selected variable the value that minimizes the number of violations.

Car Sequencing Figure 4 presents a model for car sequencing. In this application, n cars must be sequenced on an assembly line of size n. Cars may require different sets of options, while capacity constraints on the production units restrict the possible car sequences. For a given option o, these constraints are of the form *k outof m* meaning

```
1.  model m {
2.     var{int} car[Line](Configs);
3.     hard: cardinality(demand,car);
4.     forall(o in Options)
5.        soft: sequence(car,options[o],lb[o],ub[o]);
6.  }
7.  TabuSearch search(m); search.apply();
```

Figure 4: A Model For Car Sequencing.

```
1.  range R = 1..9;
2.  model m {
3.     var{int} S[R,R](R);
4.     forall(t in FixedPositions) hard: S[t.r,t.c] == t.v;
5.     forall(i in 0..2,j in 0..2)
6.        hard: alldifferent(all(r in i*3+1..i*3+3,
                                 c in j*3+1..j*3+3) S[r,c]);
7.     forall(i in R) soft: alldifferent(all(j in R) S[i,j]);
8.     forall(j in R) soft: alldifferent(all(i in R) S[i,j]);
9.  }
10. TabuSearch search(m); search.setMaxIterations(2000);
11. search.apply();
```

Figure 5: A Model For Sudoku.

that, out of m sucessive cars, at most k can require $\circ$. The model declares the decision variables specifying which type of car is assigned to each slot in the assembly line (line 2). It specifies a hard constraint specifying which cars must be produced (line 3) and then states the soft capacity constraints for each option (lines 4–5).

Like in the all-interval series, the synthesis is driven by the hard constraint. All variables appear in the cardinality constraint and swapping two of its variables is feasibility-preserving. Moreover, the cardinality constraint is tight, meaning that there is a bijection from variable to value occurrences. Finally, a hard cardinality constraint can be satisfied using a (randomized) feasible flow algorithm. It is important to emphasize that the diversification component of the tabu search randomly swaps variables as well, thus maintaining the feasibility of the hard constraint at all times. This is in contrast with the diversification of the queens model, which randomly reassigns variables, illustrating how the constraint roles affect all components of a meta-heuristic.

Sudoku The previous two models contain a single hard constraint. Figure 5 presents a model for sudoku containing many hard constraints. Lines 4–6 specify the hard constraints: simple equalities for the fixed positions (line 4) and alldifferent constraints for each of the 3×3 squares (lines 5–6). Lines 7–8 state the soft alldifferent constraints specifying that the numbers of each row and each column must be distinct. Lines 10–11 request to synthesize a tabu search with restarts after 2,000 (non-improving) iterations. Other parameters of the CBLS synthesizer can be specified similarly. For simplicity, we omit them in the rest of the paper.

The synthesizer analysis deduces the following facts:

1. all variables appears in the hard alldifferent constraints;
2. the hard alldifferent constraints have different variables;
3. the hard alldifferent constraints are tight.

```
1. MinNeighborSelector N();
2. forall(k in hardSystem.getRange()) {
3.    var{int}[] x = hardSystem.getConstraint(k).getVariables();
4.    range X = x.getRange();
5.    forall(i in X,j in X: legalSwap(x[i],x[j]))
6.       neighbor(S.getSwapDelta(x[i],x[j]),N) x[i] :=: x[j];
7. }
8. if (N.hasMove()) call(N.getMove());
```

Figure 6: The Derived Neighborhood Skeleton For Sudoku.

```
1.  model m {
2.     var{int} boat[Guests,Periods](Hosts);
3.     forall(g in Guests)
4.        soft(2): alldifferent(all(p in Periods) boat[g,p]);
5.     forall(p in Periods)
6.        soft(2): knapsack(all(g in Guests) boat[g,p],crew,cap);
7.     forall(i in Guests, j in Guests : j > i)
8.        soft: atmost(1,all(p in Periods) boat[i,p] == boat[j,p]);
9.  }
10. TabuSearch search(m); search.apply();
```

Figure 7: A Model For The Progressive Party Problem.

Fact (2) ensures that swapping two variables in one alldifferent constraint maintains the feasibility of all of them. Facts (1) and (3) guarantee that such swaps result in a connected neighborhood. Not all these swaps are legal however, since they may violate some of the unary hard constraints. The synthesizer then derives a tabu search driven by the hard alldifferent constraints, the remaining hard unary constraints being solved once for all by updating the variable domains. The initial solution is generated by finding random matchings for the hard alldifferent constraints that satisfy the domain constraints. The neighborhood maintains feasibility of hard constraints by only swapping variables occurring in the same hard constraint and by respecting the domain constraints. The best such swap is selected at each step.

Figure 6 depicts the neighborhood skeleton derived for the sudoku model. The skeleton only depicts the move selection, omits the tabu-search elements for brevity and uses advanced control structures of COMET. Line 1 declares a neighbor selector, lines 2–7 explore all the neighbors, and line 8 performs the move to the selected neighbor (if any). The moves are generated by iterating over all hard constraints (lines 2–3) and considering all legal swaps between variables occurring in the same hard constraints (line 5). These swaps are submitted with their evaluations to the neighbor selector (line 6). Note that, since the hard constraint system is feasible at all times, a swap of `x[i]` and `x[j]` is legal whenever

```
x[i] in domain(x[j]) && x[j] in domain(x[i]) &&
hardSystem.getSwapDelta(x[i],x[j]) == 0;
```

Progressive Party The previous models show how hard constraints and their semantics drive the synthesis process. The progressive party problem, a standard benchmark in combinatorial optimization, illustrates how soft constraints may also be instrumental in defining the synthesized neighborhood. The goal in this problem is to assign guest parties to boats (the hosts) over multiple time periods. Each guest

```
1.  MinNeighborSelector N();
2.  selectMax(i in X)(S.violations(x[i])) {
3.     forall(v in x[i].getDomain())
4.        neighbor(S.getAssignDelta(x[i],v),N) x[i] := v;
5.     forall(c in SwapCstr[i]) {
6.        var{int}[] y = cstr[c].getVariables();
7.        forall(j in y.getRange())
8.              neighbor(S.getSwapDelta(x[i],y[i]),N)
9.                 x[i] :=: y[i];
10.    }
11. }
12. if (N.hasMove()) call(N.getMove());
```

Figure 8: The Derived Skeleton for the Progressive Party.

```
1.  model m {
2.     var{bool} open[Warehouses]();
3.     minimize: sum(w in Warehouses) fcost[w] * open[w] +
                  sumMinCost(open,tcost);
4.  }
5.  VNSearch search(m); search.apply();
```

Figure 9: A Model For Warehouse Location.

```
1.  model m {
2.     var{bool} open[Warehouses]();
3.     hard: exactly(p,open);
4.     minimize: sumMinCost(open,cost);
5.  }
6.  VNSearch search(m); search.apply();
```

Figure 10: A Model For the K-Median Problem.

can visit the same boat only once and can meet every other guest at most once over the course of the party. Moreover, for each time period, the guest assignment must satisfy the capacity constraints of the boats.

Figure 7 depicts the model for this problem. The decision variable `boat[g,p]` specifies the boat visited by guest `g` in period `p`. Lines 3–4 specify the alldifferent constraints for each guest, lines 5–6 specify the capacity constraints, and lines 7–8 state that two guests meet at most once. All constraints are soft but the first two sets are weighted.

Since all constraints are soft, the core of the synthesized search procedure consists of reassigning a variable. However, the knapsack constraints have a significant impact in hard instances with many time periods. There are configurations in which no single assignment can decrease the violations of a knapsack, although a single swap would. The model analysis recognizes the presence of knapsack constraints and generates a neighborhood consisting of the union of variable assignments and of the swaps of variables arising in violated knapsack constraints, an idea first articulated by Van Hentenryck (2006). The skeleton for the move selection, which omits the tabu-search aspects, is depicted in Figure 8 and it uses the fact that all variables have the same domains. Line 1 defines a neighbor selector. Line 2 selects the variable `x` with the most violations. Lines 3-10 generate the possible moves for variable `x[i]`: an assignment of a value `v` (lines 3–4) or a swap with a variable `y[j]` appearing in a common knapsack constraint. The moves are inserted in the neighbor selector in line 4 and lines 8–9 respectively. The best move is selected and executed in line 12. To obtain the swaps, the synthesized search uses the constraints identified by the model analysis (line 5 for the variable `x[i]`). It then retrieves the variables of each such constraint (line 6) and considers all swaps of `x[i]` and `y[j]` (line 7). Observe that swapping variables in alldifferent or cardinality constraints cannot decrease their violations: these constraints are not identified for possible swaps by the model analysis. As shown later, the synthesized search outperforms all published results on this problem.

Optimization Problems

Warehouse Location Uncapacitated warehouse location is a standard optimization problem which consists of choosing warehouse locations to minimize the fixed costs of the warehouses and the transportation costs to the customers. Figure 9 presents a really concise model for this problem for which a variable-neighborhood search (VNS) is synthesized. The decision variables represent whether to open a warehouse (line 2) and an objective function minimizing the fixed and transportation costs (line 3). The objective function features a combinatorial objective `sumMinCost(open,tcost)` which maintains an assignment of minimal cost from the customers to the open warehouses. The synthesized variable neighborhood search iterates two steps: (1) a greedy local search whose moves select the variable assignment decreasing the objective function the most; (2) a shaking procedure that randomly reassigns an increasing number of variables. Combined with restarts, the VNS algorithm is probably the most effective search procedure for large-scale uncapacitated facility location (Harm & Van Hentenryck 2005). Note that the VNS procedure can easily be replaced by a tabu search: just modify line 5. The resulting algorithm is also quite effective but is typically dominated by the VNS.

K-Median The k-median problem is a variation of the uncapacitated warehouse location in which exactly k warehouses must be opened. Figure 10 depicts the model for this problem: It uses the same decision variables and the same combinatorial objective for the transportation cost but it adds a hard constraint (line 3) to specify the number of warehouses to open. This hard cardinality constraint drives the VNS synthesizer, very much like in the satisfaction problems. Indeed, the cardinality constraint involves all problem variables and is tight, so that only swaps must be considered.

The greedy component of VNS selects the best swap. When all swaps degrade the objective function, the shaking procedure is applied. It randomly selects a variable and applies the best swap involving the selected variable (possibly degrading the objective) (Hansen & Mladenovic 1997). As is typical in VNS, the number of variables considered in shaking increases each time the greedy component does not produce an improvement. Observe how the hard constraint drives both the greedy and shaking components of the VNS.

Scene Allocation Figure 11 features a model for the scene allocation problem, sometimes used to compare CP and MIP

```
1.  int occur[Days] = 5;
2.  model m {
3.     var{int} day[Scenes](Days);
4.     hard: atmost(occur,day);
5.     minimize: sum(a in Actor) pay[a]*
6.        (sum(d in Days) or(s in Appear[a]) (day[s]==d));
7.  }
8.  TabuSearch search(m); search.apply();
```

Figure 11: A Model For Scene Allocation.

```
1.  MinNeighborSelector N();
2.  selectMax(v in X)(O.decrease(x[v])) {
3.     select(c in X)
4.        neighbor(O.getSwapDelta(x[c],x[v]),N) x[c] :=: x[v];
5.     select(c in x[v].getDomain(): legalAssign(x[v],c))
6.        neighbor(O.getAssignDelta(x[v],c),N) x[v] := c;
7.  }
8.  if (N.hasMove()) call(N.getMove());
```

Figure 12: The Synthesized Skeleton For Scene Allocation.

solvers since it is highly symmetric. The problem consists of assigning specific days for shooting scenes in a movie. There can be at most 5 scenes shot per day and all actors of a scene must be present. Each actor has a fee and is paid for each day she/he plays in a scene. The goal is to minimize the production cost. The decision variable `day[s]` (line 3) represents the day scene `s` is shot. The hard cardinality constraint (line 5) specifies that at most 5 scenes a day can be shot. The objective function minimizes the sum of each actor compensation, which is the actor fee times the number of days he/she appears in a scene shot on that day.

The synthesized tabu search is driven by the hard constraint, but with a significant difference. In all the earlier models, the hard constraints were tight, a fact detected by the model analysis. For instance, the cardinality constraint in car sequencing is tight because the assembly line has as many slots as the number of cars to produce. This is not the case in the scene allocation: there are typically fewer scenes than the number of slots in which they can be scheduled and thus the hard constraint is not a bijection between variables and value occurrences. As a result, restricting the search only to swaps preserves the feasibility of the cardinality constraint but leads to a significant decrease in solution quality (or an increase in time). This is not surprising since the neighborhood is not connected any more. The model analysis however recognizes that the atmost constraint is not tight and also considers feasibility-preserving assignments.

The synthesized skeleton, once again omitting the tabu-search management for space reasons, is depicted in Figure 12. The variable `x[v]` with the best gradient for the objective function is selected in line 2. The swaps are considered in lines 3–4 and the assignments in lines 5–6. Note that an assignment is legal if it does not violate the hard constraints.

Spatially Balanced Magic Squares We now consider a model for the totally spatially balanced squares introduced in (Gomes *et al.* 2004) for the design of experiments. The model, depicted in Figure 13, features multiple hard constraints, multiple soft constraints, and an objective function

```
1.  int bal = size * (size + 1)/3;
2.  model m {
3.     var{int} pos[R,R](R);
4.     forall(i in R) hard: alldifferent(all(j in R) pos[i,j]);
5.     forall(j in R) soft: alldifferent(all(i in R) pos[i,j]);
6.     ObjectiveSum S();
7.     forall(v in R, w in R: v ¡ w)
8.        S.post(((sum(i in R) (abs(pos[i,v] - pos[i,w])))-bal)ˆ2);
9.     minimize: S;
10. }
11. TabuSearch search(m); search.apply();
```

Figure 13: A Model For Spacially Balanced Magic Squares.

```
1.  range Colors = 1..nb;
2.  model m {
3.     var{int} col[Vertices](Colors);
4.     forall(i in Vertices, j in Vertices: j > i && adj[i,j])
5.        soft: col[i] != col[j];
6.     minmax: col;
7.  }
8.  TabuSearch search(m); search.apply();
```

Figure 14: A Model For the Coloring Problem.

to minimize the imbalance. The hard constraints express that variables on the same row must have distinct values (line 4). The soft constraints express similar constraints for the columns (line 5). The objective function is a sum (line 6), each term of which accumulates the imbalance between a pair of values (v, w) over the rows (lines 7–8). The search synthesis proceeds in two steps. It first applies a Lagrangian relaxation of the soft constraints to obtain a new objective $O^* = w_1 * O + w_2 * Soft$ where O is the original objective, *Soft* is the system of soft constraints, and w_1, w_2 are weights to scale the two components. The soft constraints are then transformed into an objective representing their violations. The search procedure can now be driven by the hard constraints, using O^* to guide the search. The search procedure thus swaps variables in each of the hard constraints. With such a high-level and explicit model, some algorithmic choices become more apparent and easy to experiment with. One may wonder what would happen if the constraint roles are permuted. The results are surprising as shown later.

Graph Coloring Figure 14 presents a model for graph coloring. The model uses soft constraints for the not-equal constraints and a `minmax` objective to minimize the maximum value of the variables in array `col`. The search procedure synthesized for a `minmax` objective reduces an optimization problem into a sequence of feasibility problems, decreasing the maximum value in the domains each time a new solution is found. The search for the feasibility problems is synthesized as discussed earlier in the paper. The interesting point in this model is the ability of the synthesizer to exploit the specific shape of the objective function.

Experimental Results

This section reports some experimental results to indicate the feasibility of black-box constraint-based local search for some problem classes. It compares synthesized and tailored

Prob.	A	T	I	I/T
interval (15)	S	0.10	2370.06	24606.10
	T	0.06	1721.52	27526.70
interval (50)	S	4.67	54445.70	11647.04
	T	14.16	223048.96	15755.98
car (4-72)	S	11.48	67610.67	5888.47
	T	6.89	40643.17	5903.12
sudoku	S	0.50	1235.08	2465.52
progressive (5-7)	S	0.90	1913.63	2136.89
	T	1.13	2360.35	2088.55
progressive (6-7)	S	2.35	5353.05	2284.34
	T	2.91	7343.08	2519.93

Table 1: Experimental Results: Satisfaction Problems

Prob.	A	O	T	I	I/T
1032GapAS	S	36127.00	0.60	408.4	7315.9
1032GapAS.	T	36127.33	1.62	396.4	1322.4
10FPP11S	S	36240.89	0.68	1402.4	5045.6
10FPP11S.	T	36257.68	2.21	1631.6	1198.2
10FPP17S	S	54570.58	2.51	2562.6	1930.1
10FPP17S.	T	57608.96	7.87	2882.6	426.5
1331GapBS	S	43536.46	0.64	2694.5	7105.5
1331GapBS	T	45026.51	1.71	2633.7	1749.4
pmed10	S	1255.00	1.81	113.8	62.8
pmed10	T	1255.00	1.66	99.4	60.0
pmed13	S	4374.00	1.81	61.1	33.7
pmed13	T	4374.14	1.93	10.4	32.1
pmed18	S	4809.36	9.96	194.1	17.5
pmed18	T	4809.36	11.19	200.9	15.7
scene	S	334144	0.20	345.2	1737.4
scene	T	334144	0.21	406.5	1930.1
balanced	S	0.00	24.96	27476.1	1100.6
balanced	T	0.00	14.85	16121.5	1085.8
balanced-R	S	0.00	2.42	3471.5	1431.7

Table 2: Experimental Results: Optimization Problems

search procedures for the same models. The tailored procedures were typically obtained from (Van Hentenryck & Michel 2005) and some recent papers. It is not always easy to make such comparison, since the searches behave differently. As a result, the results describe not only the CPU times (on an 2.16 GHz Intel processor) but also the number of iterations per seconds, which provides an interesting measure of speed. Some of the benchmarks exhibited some surprising behaviors as will be discussed. All numbers are averages over 100 runs.

Table 1 reports the results for satisfaction problems. The second column specifies whether a synthesized (S) or tailored (T) algorithm is used. The next three columns give the average times, iterations, and iterations per second. Series of size 15 and 50 are tested for the all-interval model, the larger instance using a value-weighting scheme. The car-sequencing model is tested on instance 4-72 and the progressive party problem on the hardest configurations 5-7 and 6-7. The experimental results show that the synthesized procedures are often comparable to tailored algorithms. They produce better results on all-interval series (50), most likely due to a restarting strategy not used in the tailored program. They are dominated on car-sequencing because of a less aggressive intensification. The average number of iterations per second is close in all instances, indicating the small overhead typically induced by the synthesis.

Table 2 reports the results for optimization problems. and also gives the average solution quality in column O. The warehouse location and median models are evaluated on some standard benchmarks containing up to 400 warehouses. The spacially balanced latin squares are of size 9 by 9. Some of the results were surprising here. On warehouse location, the synthesized procedure outperforms the state-of-the-art algorithm: It is a pure VNS procedure with restarts, while Harm and Van Hentenryck (2005) proposed an hybrid tabu-VNS search (also with restarts). Also, their algorithm maintains the best move incrementally, which does not seem to pay off on these instances. The results on the k-median problem are really comparable. The scene allocation results are interesting because the tailored procedure makes the cardinality constraint tight by exploiting the problem semantics. Yet the synthesized procedure is clearly competitive. Note also the excellent results obtained on the balanced latin squares with different constraint roles.

Conclusion

This paper showed that, for some application classes, local search algorithms can be synthesized from high-level CBLS models. The synthesis is driven by the structure of the model, the constraint roles and the constraint semantics and produce search algorithms that appear competitive with state-of-the-art procedures. Such synthesizer fundamentally increase the ease of use for CBLS, and provide a first step toward a black-box version of COMET as suggested by Gomes and Selman (2006). There are obviously many research opportunities and challenges ahead. They include the synthesis of other meta-heuristics such as guided local search, simulated annealing, and hybrid evolutionary algorithms, and the integration of instance-based performance-tuning techniques (e.g., for restarts or tabu management). The resulting synthesizers can then be integrated in a modeling system exploiting a wide portfolio of algorithms.

References

Gomes, C., and Selman, B. 2006. The Science of Constraints. http://www.cs.cornell.edu/gomes/TALKS/next10-cp06.pdf.

Gomes, C.; Sellmann, M.; van Es1, C.; and van Es, H. 2004. The Challenge of Generating Spatially Balanced Scientific Experiment Designs. In *CP-AI-OR'04*, 387–394.

Hansen, P., and Mladenovic, N. 1997. Variable Neighborhood Search for the P-Median Problem. *Location Science* 5(4).

Harm, G., and Van Hentenryck, P. 2005. A MultiStart Variable Neighborhood Search for Uncapacitated Warehouse Location. In *MIC-2005*.

Puget, J.-F. 2004. Constraint programming next challenge: Simplicity of use. In *CP-04*, 5–8.

Van Hentenryck, P., and Michel, L. 2005. *Constraint-Based Local Search*. Cambridge, Mass.: The MIT Press.

Van Hentenryck, P. 2006. Constraint Programming as Declarative Algorithmics. (http://www.cs.brown.edu/people/pvh/acp.pdf).