Synthesis of Search Algorithms from High-level CP Models*

Samir A. Mohamed Elsayed**, Laurent Michel

Computer Science Department, University of Connecticut.

Abstract. The ability to specify CP programs in term of a declarative model and a search procedure is a central feature primarily responsible for the industrial CP successes. However, writing search procedures is often difficult for novices or people accustomed to mathematical programming tools where this step is absent. Several attempts have been made to produce generic black-box searches that would be suitable for the vast majority of benchmarks. This paper offers an alternative viewpoint and argues for the synthesis of a search from the declarative model to exploit the problem instance structures. The intent is not to eliminate the search altogether. Instead, it is to have a default that performs adequately in the majority of cases while retaining the ability to write full-fledged procedures for experts. Preliminary empirical results demonstrate that the approach is viable, delivering search procedures approaching and sometimes rivaling hand-crafted code produced by experts.

1 Introduction

Constraint programming (CP) techniques are successfully used in various industries and quite successful when confronted with hard constraint satisfaction problems. Parts of this success can be attributed to the considerable amount of flexibility that arise from the ability to write completely custom search procedures. Indeed, constraint programming has often been described from the basic belief that

$$CP = Model + Search$$

where the model is responsible for providing a declarative specification of the constraints that solutions of the problem must satisfy and the search is a specification of how to explore the search space to produce such a solution. In a number of languages designed for constraint programming, the search can be quite sophisticated. It can often concisely specify variable and value selection heuristics, search phases [11], restarting strategies [7], large neighborhood search [1], exploration strategies like depth-first-search, best-first search or limited discrepancy search [9] to name just a few.

This capability is in stark contrast with, for instance, mathematical programming where the search is a so-called *black-box* that can only be controlled

* This work is partially supported through NSF award IIS-0642906.
** The author is partially supported by Helwan University, Cairo, Egypt.

through a collection of parameters affecting pre-processing, cut generation or the selection of predefined global heuristics. Users of mathematical programming are accustomed to solely rely on modeling techniques and reformulations to indirectly influence and hopefully strengthen the search process effectiveness.

Unsurprisingly, many users of one technology often bring their baggages, habits and expectations when discovering a new technology like CP. Too often, newcomers overlook the true potential of open (i.e., *white-box*) search specification and fail to exploit it. The observation prompted a number of efforts to rethink constraint programming tools and mold them after mathematical programming tools by *eliminating open search procedures* in favors of intelligent black-box procedures. Efforts of this type include [11] and [4] while others, e.g., [14] elected to provide a number of predefined common heuristics.

Our contention in this paper is that it is possible to get the best of both worlds, retaining the ability to write custom search procedures and *synthesizing* model-specific search procedures that are competitive with procedures handcrafted by experts. The central contribution of this paper is Cp-as, a model-driven automatic search procedure generator. Cp-as is written in Comet [17], an object-oriented programming language with support for constraint-programming at large and finite domains solving in particular. Cp-as analyzes a CP model instance at runtime, examining the variable declarations, the arithmetic and logical constraints, as well as the global constraints, to synthesize a procedure that is likely to perform reasonably well on this model instance. Cp-as is evaluated on a collection of CP models that requires custom search such as scene allocation, progressive party and warehouse allocation. The rest of the paper is organized as follows. Section 2 presents some related work. Section 3 provides details about the synthesis process of Cp-as. Section 4 illustrates the process on some popular CP applications. Experimental results are reported in section 5. Finally, section 6 concludes.

2 Related Work

Some work have been already done on the process of search synthesis from high level models. Minion [4] offers a black-box search and combines it with matrix based modeling, aiming for raw speed alone to produce 'model and run' solutions. The idea of deriving the search from the model appeared in [19] for Constraint-Based Local Search (CBLS) in which search procedures can be automatically synthesized from models expressed in a rich constraint language. Given a model, a CBLS synthesizer derives a local search 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. The experimental results show that effective LS algorithms can be synthesized from high-level models, inducing only a small overhead over state-of-the-art procedures. Aeon [12] is closely related and focuses exclusively on scheduling. Given a scheduling model specified in a high-level modeling language, Aeon recognizes and classifies its structures, and

synthesizes an appropriate search algorithm. The classification result drives the selection of a search template. AEON provides synthesizers for Constraint-based Scheduling (complete search) or Constraint-based local search (incomplete). Experimental results indicate that this approach may be competitive with state-of-the-art search algorithms. This paper extends this line of thinking with synthesis of search for *general non-scheduling* CP models.

3 The Synthesis Process

CP-AS, which is written in COMET, generates search procedures for COMET models. CP-AS is defined in term of an extensible collection of rules meant to recognize features of the model for which good heuristics are known. Each rule can issue a set of *recommendations* where a recommendation is characterized by a score indicating its fitness, a subset of variables to which it applies, and three heuristics to be used for labeling, namely: a variable, value and symmetry-breaking heuristic. CP-AS applies all the rules to a model to obtain a set of recommendations which fully specify the search procedure. Recommendations are applied in decreasing order of scores, hence if several recommendations apply to the same subset of variables, the highest-scoring one will take precedence. If there is an overlapping among different subsets of variables, then the highest-scoring set will be labelled first and will not be considered by lower-score recommendations. This section details this process.

3.1 Preliminaries

A Constraint Satisfaction Problem (CSP) is a triplet <X,D,C>, where X is a set of variables, D is a set of domains, and C is a set of constraints. Each $x_i \in X$ is associated with a domain $D_i \in D$. An assignment α associates a value $v \in D_i$ to each variable x_i, i.e., $\alpha(x_i) = v \in D_i$. A constraint $c \in C$, over variables $x_i, \cdots, x_j$, specifies a subset of the Cartesian product $D_i \times \cdots \times D_j$ indicating mutually-compatible variable assignments. A tuple $v = (v_i, \cdots, v_j)$ satisfies a constraint $c(x_i, \cdots, x_j)$ if $v \in c$. A solution α is a complete assignment that satisfies all the constraints. A Constraint Optimization Problem (COP) <X,D,C,f> is a CSP with an objective function f.

Definition 1. *A variable selection heuristic h_x maps a set of variables of fixed cardinality k to a permutation of those variables: $h_x : 2^X \rightarrow \mathbb{N} \rightarrow X$.*

Example 1. For instance, the familiar first-fail variable selection heuristic over a set of variables X returns a permutation function $\pi : \mathbb{N} \rightarrow X$ of the naturals $0..k-1$ that satisfies

$$\forall i, j \in 0..k-1 : i < j \Rightarrow |D_{x_{\pi(i)}}| \leq |D_{x_{\pi(j)}}|$$

Definition 2. *A value selection heuristic h_v is a function that maps a set of finite values of fixed cardinality k to a permutation: $h_v : 2^{\mathbb{Z}} \rightarrow \mathbb{N} \rightarrow \mathbb{Z}$.*

Example 2. For instance, the min-value heuristic for x with domain D of cardinality k specified as $h_v(D)$ denotes the permutation function π satisfying

$$\forall a, b \in 0..k-1 : a < b \Rightarrow \pi(a) \leq \pi(b)$$

Definition 3. *A value symmetry breaking heuristic h_s is a function that maps a set of finite values of fixed cardinality k to a subset of non-symmetric values: $h_s : 2^{\mathbb{Z}} \rightarrow 2^{\mathbb{Z}}$.*

In the following, we assume that $deg(x)$ denotes the cardinality of the set of constraints referring to x. $vars(c)$ denotes the subset of variables from X that appear in constraint c. $A(X)$ denotes the set of arrays of decision variables (groupings) found among the declarations of X. $type(c)$ denotes the nature of a constraint c. Finally, $G(C)$ denotes the set of the global constraint types found in C, i.e., $G(C) = \{type(c) : c \in C\} \cap G_T$ where G_T is the set of global constraint types supported by the COMET solver.

3.2 Rules and Recommendations

Definition 4. *Given a CSP $M = \langle X, D, C \rangle$, a* CP-AS *rule $r \in Rules$ is a quadruple $\langle \mathcal{S}, \mathcal{P}, \mathcal{V}, \mathcal{H} \rangle$ where $\mathcal{S}$ is a scoring function $\mathcal{S} : M \rightarrow \langle \mathbb{R}^n \rangle$ where $n > 0$ and $\mathcal{P}$ is a priority function $\mathcal{P} : M \rightarrow \langle \mathbb{Z}^n \rangle$. $\mathcal{V} : 2^X \rightarrow \langle 2^{X^n} \rangle$ is a function returning a vector of subsets of variables subjected to the recommended search heuristic. Finally, $\mathcal{H} : M \rightarrow \langle \langle h_x, h_v, h_s \rangle^n \rangle$ is a function returning a vector of triplets of heuristic functions.*

All scores are normalized in the 0..1 range with 1 representing the strongest fit. While priorities are in the $1..\infty$ range with 1 representing the highest priority. In addition, CP-AS uses priorities not only to break ties when rules come up with the same exact score, but also, to scale the score so that the higher priority rules get higher score.

Definition 5. *A set of recommendations $\{\langle S_i, P_i, V_i, H_i \rangle : 0 < i \leq n\}$ resulting from applying a rule $r = \langle \mathcal{S}, \mathcal{P}, \mathcal{V}, \mathcal{H} \rangle$ to a CSP $M = \langle X, D, C \rangle$, where $S = \mathcal{S}(M)$, $P = \mathcal{P}(M)$, $V = \mathcal{V}(X)$, and $H = \mathcal{H}(M)$, produces the four vectors $\langle S, P, V, H \rangle$ all of length n representing the recommendations scores, priorities, variable sets, and heuristics respectively.*

Note that most rules issue a set of recommendations (i.e., $n > 1$). Some might produce a singleton though (i.e., $n = 1$). In addition, currently recommendations share the same priority of the corresponding rule.

3.3 Rules Library

CP-AS rules are meant to exploit modeled structures and properties such as global constraints, domain sizes, or variables degree to name just a few. Currently, all the rules share the same value ordering heuristic, namely, min-value ordering. This temporary limitation will be lifted in the near-future with the addition of other value-selection heuristics and their selection in finer-grained rules.

The symmetry breaking heuristic is a result of a *global analysis* of the model. Even though each rule is endowed with a potentially specialized symmetry-breaking heuristic, all the recommendations share the same symmetry breaking heuristic. The following presents a description for each rule in the rule set:

Degree rule. Intuitively, the rule determines whether the static degrees of variables in the constraint hyper-graph are sufficiently diverse. If the degrees are all very similar, a degree based heuristic is inappropriate. Conversely, a very diverse set of degrees indicates a strong fit. The measure of diversity is based on the relative frequencies of each member of the collection [6]. The degree variable heuristic consists in choosing first the variable with the smallest domain size while breaking ties with the highest static degree. The rule generates a set of recommendations, one for each decision variable array $a \in A(X)$ and scale their scores by a ratio of the highest degree in the array to the highest degree overall. Intuitively, if the array only contains variables with low degrees, the score will be lowered (the ratio is maximum at 1 when the highest degree variable is in the array). Accordingly, the rule consists of the following:

- $\mathcal{S}(< X, D, C >) = \langle S_1, \ldots, S_n \rangle$ where $n = |A(X)|$.
- $\mathcal{V}(X) = \langle V_1, \ldots, V_n \rangle$.
- $\mathcal{H}(< X, D, C >) = \langle H_1, \ldots, H_n \rangle$.

While a recommendation consists of the following:

- $S_i = (1 - \sum_{d=1}^{z} p_d^2) \cdot \frac{\max_{x \in a} deg(x)}{\max_{x \in X} deg(x)}$ where z is the count of the distinct variable degrees, and $p_d = freq_d / |X|$ where $freq_d = |\{deg(x) = d : x \in X\}|$.
- $V_i = a$.
- $h_x \in H_i$ is a combination of the first fail function defined earlier, and the function $\pi : \mathbb{N} \to X$ of the naturals $0..k-1$ that satisfies the following property:
$$\forall i, j \in 0..k-1 : i < j \Rightarrow deg(x_{\pi(i)}) \geq deg(x_{\pi(j)})$$

Global constraints rules. Global constraints are extremely useful in many CP models. They capture common combinatorial substructures and exploit their semantics to obtain effective filtering algorithms [3]. The CP-AS framework is meant to include rules to take advantage of this fact. Currently, it offers rules for the most frequently used global constraints such as `alldifferent` and the `knapsack`. A few interesting observations are in order:

- When the variables $vars(c)$ of a global constraint c coincide with a user-specified *grouping* of variables (e.g., an array from $A(X)$), it is often effective to first branch on this group. Similarly, when the ratio of |vars(c)| to the total variables of the same constraint type tends to 1, it is often effective to first branch on these variables as well.
- High couplings (i.e., a dense connectivity in the constraint graph) among different variable groups resulting from the presence of many different global constraint types (larger $|G(C)|$) tends to adversely affect the effectiveness

of such heuristic. Indeed, conflicting preferences for the different global constraints may be at odds with each other and any one choice may not dominate or might even be counter-productive.
– Whenever several groups in $A(X)$ are covered by global constraints, the array with variables of higher degrees should be labeled first.

Given a global constraint c, a global rule applies to $vars(c)$. Thereby, the rule generates a set of recommendations, one for each global constraint of the same type as c. The fallback variable selection heuristic, in case a specific global rule does not wish to further specialize h_x, is simply the first-fail heuristic. Accordingly, the generic rule consists of the following:

– $\mathcal{S}(< X, D, C >) = \langle S_1, \ldots, S_n \rangle$ where $n = |\{k : k \in C \wedge type(k) = type(c)\}|$.
– $\mathcal{V}(X) = \langle V_1, \ldots, V_n \rangle$.
– $\mathcal{H}(< X, D, C >) = \langle H_1, \ldots, H_n \rangle$.

While a recommendation consists of the following:

– Intuitively, the more similar the constraint types used in the model, the stronger the fit. Once again, the score is scaled by a ratio of the maximum variable degree in the global constraint to the variable degree over the entire model. If none of the global constraints covers an entire variable group, the score is based on the relative size:

$$S_i = \begin{cases} \frac{\max_{x \in vars(c)} deg(x)}{\max_{x \in X} deg(x) \cdot |G(C)|} & if \exists a \in A(X) \text{ s.t. } a \subseteq vars(c) \\ \\ \frac{|vars(c)|}{|\{vars(k) : k \in C \wedge type(k) = type(c)\}| \cdot |G(C)|} & \text{otherwise.} \end{cases}$$

– $V_i = vars(c)$.
– $h_x \in H_i$ is the first fail function.

Knapsack Rule. When there is one `knapsack` constraint $c \equiv \sum_{i \in N} w_i \cdot x_i \leq b$ defined over $vars(c) = \{x_i : i \in N\}$ and this collection of variables coincides exactly with a user-defined variable group, the suggested variable selection heuristic is to consider the variables in decreasing weight w_i order with ties broken by domain sizes. Namely, whenever $\exists a \in A : vars(c) = a$ with $|vars(c)| = k$, the produced variable selection heuristic is a function $\pi : \mathbb{N} \to X$ that satisfies the following property

$$\forall i, j \in 0..k-1 : i < j \Rightarrow w_{\pi(i)} \geq w_{\pi(j)}$$

Otherwise, the variable selection is the default first-fail heuristic.

Pick Value First Rule. If the size of the union of, the domain values of the decision variables, is sufficiently bigger than the size of the decision variables set. Then, the suggested heuristic is to choose the value, rather than the variable, first and then this value is tried on all the variables containing this value inside their domain. The rule generates a set of recommendations, one for each decision variable array $a \in A(X)$ with a score based on the ratio between the variables size to the the domain range. Accordingly, the rule consists of the following:

- $\mathcal{S}(< X, D, C >) = \langle S_1, \ldots, S_n \rangle$ where $n = |A(X)|$.
- $\mathcal{V}(X) = \langle V_1, \ldots, V_n \rangle$.
- $\mathcal{H}(< X, D, C >) = \langle H_1, \ldots, H_n \rangle$.

While a recommendation consists of the following:

- Given an array $a \in A(X)$, the score of the corresponding recommendation tends towards 1 as the number of selectable values becomes substantially larger than the number of variables in the array a. Formally,

$$S_i = \begin{cases} 1 - \frac{1}{\alpha} \; if \alpha = \frac{|\cup_{i \in a} D_i|}{|a|} \wedge \alpha \geq 1 \\ 0 \qquad \text{otherwise} \end{cases}$$

- $V_i = a$.
- $h_x \in H_i$ is a chronological order specified by the identity function $\pi : \mathbb{N} \to X$ defined over the naturals $0..k-1$.

Most constrained variables Rule. This rule follows the first fail dynamic ordering but in a slightly different way. It sorts each decision variables array $a \in A(X)$ according to their degree over the model constraints. Then, it branches on the variables of each array based on their domain sizes. Thereby, the rule generates a different recommendation for each decision variable array $a \in A(X)$ with a score determined by a ratio of the highest degree in the array to the highest degree overall. Accordingly, the rule consists of the following:

- $\mathcal{S}(< X, D, C >) = \langle S_1, \ldots, S_n \rangle$ where $n = |A(X)|$.
- $\mathcal{V}(X) = \langle V_1, \ldots, V_n \rangle$.
- $\mathcal{H}(< X, D, C >) = \langle H_1, \ldots, H_n \rangle$.

While a recommendation consists of the following:

- $S_i = \max_{x \in a} deg(x) / \max_{x \in X} deg(x)$.
- $V_i = a$.
- $h_x \in H_i$ is the first fail function.

First-Fail. This is the default rule in the case of no other rules got a better score. As the name implies, it follows the first fail principle. The score function in this case returns a constant ($\omega \approx 0$). Thereby, the rule generates a single recommendation (i.e., $n = 1$) with a suggested heuristic that sorts all decision variables based on their domain sizes in an ascending order. Accordingly, the rule consists of the following:

- $\mathcal{S}(< X, D, C >) = \langle \omega \rangle$.
- $\mathcal{V}(X) = \langle X \rangle$.
- $\mathcal{H}(< X, D, C >) = \langle H_1 \rangle$.

As for rule priority, the global constraints rule set has mainly the highest priority (i.e., $\mathcal{P} \rightarrow 1$), while the default first fail rule has the lowest (i.e., $\mathcal{P} \rightarrow \infty$). However, among the global constraints rule set itself, one rule's priority can be higher than another. For example, the `knapsack` constraint is more likely to better distinguish the variables, thanks to the weights, whereas an `alldifferent` constraint sees all the variables on equal footing. Hence, the knapsack rule gets a higher priority. In addition, the degree rule gets a higher priority (due to its wide use and effectiveness) than that of the the pick-value-first rule which has in turn higher priority than that of the most constrained variables rule (which is just an enhanced version of the first fail default rule).

3.4 Symmetry Breaking

Symmetries arise in many CP models, due to the fact that either certain variables or some domain values are indistinguishable. Symmetries are worthwhile exploiting for two reasons. First, one is often interested in only one solution and not all its symmetric cousins. Second, the search can be made significantly more efficient by avoiding the repeated explorations of symmetric sub-trees that provably contain no solutions. Variable and value symmetries can be broken either statically with constraints or dynamically through improvements to the search process. While breaking symmetries statically is appealing for its simplicity it can sometimes lead to unexpected behaviors when the filtering of the symmetry-breaking constraint adversely affects the variable and value selection heuristics. Breaking symmetries dynamically through the search avoids that issue. To synthesize a search that implements a form of dynamic value symmetry breaking, it is therefore necessary to first analyze the model to recognize value symmetries and be in a position to dynamically collapse values to their equivalence classes and only consider one value from each equivalence class when labeling.

The automatic derivation of *value* symmetry breaking is a key feature of Cp-as[1]. It follows the work of [16] where the authors propose a compositional approach that detects symmetries in CP models by exploiting the properties of the combinatorial sub-structures explicitly captured in the declarative model with global constraints. They show that, once the symmetries of global constraints are specified, various classes of symmetries can be derived precisely and efficiently in a compositional fashion. Cp-as analyzes the constraints according to the propositions/patterns found in [16,2]. If the compositions of the constraints are value interchangeable, Cp-as breaks the symmetry by only considering values previously selected plus one additional unselected value.

3.5 Rules Composition & Calibration

Recommendations can be composed to derive an effective search procedure. Cp-as sorts the recommendations by score while breaking ties with priorities. The search template is shown in Figure 1. Cp-as iterates over the recommendations

[1] It is still at an early stage of development

in lexicographic order of score and priority. Line 2 invokes the polymorphic labeling method *label* of the recommendation. Once all the variables are bound, the search ends in line 3. The search is complete as the set of variables it labels is $\bigcup_{r \in rec} \left(\cup_{x \in V(r)} \right) = X$.

```
1 forall(r in rec.getKeys()) by (−rec{r}.getScore(), rec{r}.getPriority()) {
2    rec{r}.label();
3    if (solver.isBound()) break;
4 }
```

Fig. 1. A Skeleton for a Synthesized Search Template.

Figure 2 depicts the implementation of *label* method of a *variable first recommendation*, i.e., a recommendation that first selects a variable and then chooses a value as opposed to a *value first recommendation* that selects a value and then chooses the variable to assign it to. Line 10 (respectively 23 for the value recommendation) retrieves the variables the recommendation operates on, and line 12 (respectively 27) selects a variable according to the heuristic h_x of the recommendation. Line 13 (respectively 25) retrieves the set of values to try for the chosen variable. Note that the *getValues* method encapsulates the use of value symmetry-breaking. If there is no exploitable value symmetries, the *getValues* method returns the complete domain for a variable x_i. The value selection is driven by the heuristic h_v on line 14 (respectively 26) which embodies the value permutation adopted by the recommendation.

4 Example Applications

In this section, the CP-AS synthesis process is illustrated on a few representative combinatorial applications known for benefiting from non trivial search, namely: the progressive party problem, scene allocation problem, and the steel mill slab problem. The definitions of the problems are borrowed from [19,18]. The experimental results provide a comparison on a more extensive set of problems and instances which couldn't be covered in details below due to space limitations.

4.1 Progressive Party

The progressive party problem must assign guest parties to boats (the hosts) over multiple time periods while satisfying several constraints. Each guest 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 3 depicts the model for the problem. The decision variable `boat[g,p]` (line 2) specifies the boat visited by guest `g` in period `p`. Lines 4–5 specify every guest must have an assignment in all periods, lines 6–7 specify the capacity constraints on the boat for every time periods, and lines 8–9 state the social mingling constraints, namely: two guests meet at most once during the course of the evening.

```
interface Recommendation {
    var<CP>{int}[] getVars();
    set{int} getValues(var<CP>{int} x);
    set{int} unboundVars(var<CP>{int}[] x);
    int hx(var<CP>{int}[] x,int rank);
    int hv(set{int} vals,int rank);
}
class VariableRecommendation implements Recommendation { ...
    void label() {
        var<CP>{int}[] x = getVars();
        while(!bound(x)) {
            selectMin(i in unboundVars(x))(hx(x,i)) {
                set{int} values = getValues(x[i]);
                tryall<solver>(v in values) by (hv(values,v))
                    solver.label(x[i], v);
                onFailure solver.diff(x[i], v);
            }
        }
    }
}
class ValueRecommendation implements Recommendation { ...
    void label() {
        var<CP>{int}[] x = getVars();
        while(!bound(x)) {
            set{int} values = collect(s in x.getRange():!x[s].bound()) x[s].getMin();
            selectMin(v in values) (hv(values,v)) {
              tryall<solver>(i in unboundVars(x)) by (hx(x,i)) {
                    solver.label(x[i], v);
                onFailure solver.diff(x[i], v);
              }
            }
        }
    }
}
```

Fig. 2. The Variable/Value Recommendation Classes and their Label Methods.

```
Solver<CP> m();
var<CP>{int} boat[Guests,Periods](m,Hosts);
solve<m> {
    forall(g in Guests)
        m.post(alldifferent(all(p in Periods) boat[g,p]),onDomains);
    forall(p in Periods)
        m.post(multiknapsack(all(g in Guests) boat[g,p],crew,cap));
    forall(i in Guests, j in Guests : j > i)
        m.post(sum(p in Periods) (boat[i,p] == boat[j,p]) <= 1);
}
CPAS.generateSearch(m);
```

Fig. 3. A Model for the Progressive Party Problem.

Line 11 shows how the user invokes CP-AS to derive the search. The following are the top rated recommendations generated by CP-AS:

$$\{\langle S_1 = 0.5, P_{knapsack}, V_1 = vars(c)\rangle : type(c) \in G(C) \wedge type(c) = \texttt{knapsack}\}$$
$$\bigcup$$
$$\{\langle S_2 = 0.25, P_{alldiff}, V_2 = vars(c)\rangle : type(c) \in G(C) \wedge type(c) = \texttt{alldifferent}\}$$

The highest scoring recommendations correspond to global constraints, as they cover all the variables of the `boat` matrix. The synthesized search proceeds through a sequence of *phases*, one for each recommendation, and labels each set of variables corresponding to a `knapsack` constraint in the order dictated by the chosen variable selection heuristic. Given that the `knapsack` constraint at hand operates on a single column of the `boat` matrix at a time, it does not cover the entire matrix and, therefore, the selected variable heuristic h_x is first-fail. Note that this synthesized search delivers a variable selection heuristic that coincides with the tailored algorithm [13].

4.2 Scene Allocation

The problem deals with 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 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. Figure 4 depicts the model

```
1  Solver<CP> m();
2  var<CP>{int} shoot[Scenes](m,Days);
3  var<CP>{int} nbd[Actor](m,Days);
4  int up[i in Days] = 5;
5  minimize<m> sum(a in Actor) fee[a] * nbd[a]
6  subject to {
7     forall(a in Actor)
8        m.post(nbd[a]==sum(d in Days) (or(s in which[a]) shoot[s]==d));
9     m.post(atmost(up,shoot),onDomains);
10 }
11 CPAS.generateSearch(m);
```

Fig. 4. A Model for the Scene Allocation Problem.

for this problem. The decision variable `shoot[s]` (line 2) represents the day scene `s` is shot. Whereas, the decision variable `nbd[a]` represents the number of days an actor `a` appears in the scenes. The cardinality constraint (line 9) specifies that at most 5 scenes a day can be shot. The objective function (line 5) minimizes the sum of each actor compensation. The following are the top rated

recommendations generated by Cp-as:

$$\{\langle S_1 = 0.15, P_{degree}, V_1 = \texttt{shoot}\rangle\}$$
$$\bigcup$$
$$\{\langle S_2 = 0.1, P_{cardinality}, V_2 = vars(c)\rangle : type(c) \in G(C) \wedge type(c) = \texttt{cardinality}\}$$
$$\bigcup$$
$$\{\langle S_3 = 0.025, P_{degree}, V_3 = \texttt{nbd}\rangle\}$$

The highest scoring recommendations correspond to the degree rule. Indeed, the scene allocation model has a nice variation among the static degree of its decision variables. Note that the degree rule delivered two different recommendations with different scores. Naturally, the first one with the higher score recommends branching on the variables of the array `shoot` since it is referred by more constraints than the other array `nbd`. The synthesized search proceeds and label the set of variables of the first recommendation. Since all the variables get bound by the first recommendation, the search doesn't need to go beyond that.

[10] suggests that the exact days assigned to the scenes have no importance in this application and are fully interchangeable which is inferred by the symmetry breaking analysis. Thus, value symmetries can be broken by only considering days previously selected plus one additional unselected day.

Cp-as combines the degree heuristic and the value symmetry breaking. It would be interesting to try a dynamic variable degree heuristic in the future. Unlike the synthesized search, the tailored search [15] iterates over the scenes and always chooses to assign first the scene with the smallest domain. Ties are broken by choosing the most costly scene first. It employs the exact same value symmetry breaking technique though.

4.3 Steel Mill Slab Design

The problem deals with producing n orders using a set of slabs. Each order o has a color c_o representing the path the slab takes in the factory and a weight w_o for its size. Each slab has a capacity that must be chosen from the increasing set of capacities $\{u_1, u_2, \ldots, u_k\}$. A solution is an assignment of orders to slabs such that:

- The total weights of the orders in a slab must not exceed the slab capacity.
- Each slab contains at most two colors.

The objective is to minimize the sum of the weights of the slabs used in the solution or, equivalently, the sum of losses (unused capacity) in the slabs used in the solution.

Figure 5 depicts the model for this problem. The decision variable `x[o]` (line 2) specifies the slab assigned to order `o`, while variable `l[s]` (line 3) represents the load of slab `s`. Once the load of a slab is known, it is easy to compute its loss: simply take the smallest capacity supporting the load. Line 5 computes an array of losses for each possible capacity, while the objective function in line 6 uses the element constraint to compute the loss of each slab. The following are

```
Solver<CP> m();
var<CP>{int} x[Orders](m,Slabs);
var<CP>{int} l[Slabs](m,0..maxCap);
var<CP>{int} obj(m,0..nbSlabs*maxCap);
int loss[c in 0..maxCap] = min(i in Caps: capacities[i] >= c) capacities[i] - c;
minimize<m> obj subject to {
  m.post(obj == sum(s in Slabs) loss[l[s]]);
  m.post(multiknapsack(x,weight,l));
  forall(s in Slabs)
    m.post(sum(c in Colors) (or(o in colorOrders[c]) (x[o] == s)) <= 2);
}
CPAS.generateSearch(m);
```

Fig. 5. A Model for the Steel Mill Slab Design.

the top rated recommendations generated by CP-AS:

$$\{\langle S_1 = 1, P_{knapsack}, V_1 = vars(c)\rangle : type(c) \in G(C) \wedge type(c) = \texttt{knapsack}\} \bigcup \{\langle S_2 = 0.1, P_{degree}, V_2 = \texttt{x}\rangle\}$$

The highest scoring recommendation corresponds to the knapsack rule (score =1). The slab model contains only one type of global constraints (i.e, `knapsack`) and the sole constraint fully covers a decision variable array (i.e., `x`). Hence, the variable ordering is based on the knapsack weights. Again, since all the variables get bound by this recommendation, the search doesn't need to go beyond that.

Similar to the scene allocation symmetry property, two empty slabs are equivalent for allocating an order and are fully interchangeable [18]. Again, CP-AS takes advantage of this property and breaks the value symmetries by only considering slabs in which some orders have been placed plus one additional empty slab.The tailored search is just a slightly different [18]. It iterates over the orders, selecting first the orders with the smallest domains and breaking ties by choosing orders with the largest weight in a chronological order.

5 Experimental Results

Preliminary experiments show the practicality of CP-AS on a range of problems. It compares synthesized, tailored, and first fail procedures on the same models. The tailored procedures were taken from the state-of-the art. Table 1 reports the number of choices, the average CPU time (in seconds), and its standard deviation.

The selected benchmarks are well-known CSP and COP problems that use tailored search procedures. The steel mill slab model is tested on the CSPLib [5] instance as well as two more instances (20-10, 18-10) from [8]. Similarly, the car sequencing model is tested on 3 instances (car-1,car-2,car-3) from [5]. In addition, the progressive party problem is tested on different configurations and periods (1-9, 2-8, 6-6) from [5].

All results were obtained from 50 runs with COMET 2.1 on 2.33GHz Intel Core Duo machine with 2GB RAM running Ubuntu 9.10. A time out of 5 minutes (i.e., 300 ms) is employed. The results show that the synthesized search is

Problem	#cp,**t** (FF)	#cp,**t**,σ (CP-AS)	#cp,**t**,σ (tailored)
Slab (csplib)	>300	866,**2.688**,538	1656,**4.755**,666
Slab (20-10)	>300	1093,**3.497**,522	908,**2.451**,343
Slab (18-10)	>300	1672,**7.047**,1034	>300
Scene Allocation	>300	10124,**1.405**,197	4331,**0.621**,88
Car (car-1)	>300	475,**0.529**,74	77,**0.172**,25
Car (car-2)	>300	>300	69,**0.173**,25
Car (car-3)	>300	>300	669,**0.825**,116
Perfect Square	>300	74,**0.191**,27	74,**0.182**,26
Progressive (1-9)	>300	326,**0.156**,22	326,**0.158**,22
Progressive (2-8)	>300	2675,**0.845**,118	2675,**0.798**,112
Progressive (6-6)	>300	185,**0.112**,16	185,**0.107**,15
Warehouse	1645, **0.072**	119,**0.118**,3	70,**0.007**,2
SGP (8-4-7)	168, **0.298**	533,**0.478**,67	146,**0.146**,138
Sports Avenue	>300	11069,**9.721**,1361	11062,**9.471**,1326
Graph Coloring	>300	308676,**28.301**,3969	491736,**22.124**,3101
R-robin Tourn.	>300	10474,**8.614**,1206	10474,**8.318**,1164

Table 1. Experimental Results.

often reasonably close to the tailored search procedures. In particular, the synthesized search of the progressive party is competitive with the tailored search on all instances. For the scene allocation problem, the static-degree selected by the synthesizer is quite competitive with the tailored search. Surprisingly, the synthesized search outperforms the tailored search for the steel slab mill problem (the synthesized search uses a variable heuristic first driven by the weights of the knapsack, then by domain size). Indeed, it outperformed the tailored search substantially on the instance (18-10), the hardest among the 3 used instances, while having good results on the other ones too.

In the case of car sequencing, the synthesized search was decent on the easiest instance. However, it failed to scale to the harder instances. This is probably due to a sophisticated value selection heuristic employed by the tailored search that CP-AS does not support yet. A variety of value selection heuristics will be investigated shortly.

Additional experiments performed on several easier models, which do not require sophisticated search, indicate no significant degradation in the performance compared to the first fail search. The small overhead observed on some instances is due to the generic data structures manipulated by the search template. In a nutshell, the synthesized algorithms are competitive to tailored algorithms offering a reasonable first search with no efforts beyond modeling.

6 Conclusion & Future Work

CP-AS is a framework to automatically generate search algorithms from high-level CP models. Given a COMET CP model, CP-AS recognizes and classifies its structure to synthesize an appropriate algorithm. Preliminary empirical results indicate that the technique appears to be competitive with state-of-the-art procedures on several classic benchmarks.

While the approach demonstrates potential, work remains to augment the rule set. Ideally, rules should capture as many strategies as possible and recognize

when they are applicable. Improvements to the composition mechanism, the symmetry breaking inference capabilities, as well as an ability to handle search strategies are needed. Finally, an in-depth empirical evaluation is absolutely essential.

References

1. Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Appl. Math.*, 123(1-3):75–102, 2002.
2. M. Eriksson. Detecting symmetries in relational models of CSPs. Master's thesis, Department of Information Technology,Uppsala University, Sweden, 2005.
3. F. Focacci, A. Lodi, and M. Milano. Optimization-Oriented Global Constraints. *Constraints*, 7(3-4):351–365, 2002.
4. I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast, scalable, constraint solver. In *ECAI 2006: 17th European Conference on Artificial Intelligence, August 29-September 1, 2006, Riva del Garda, Italy*, page 98, 2006.
5. I.P. Gent and T. Walsh. Csplib: a benchmark library for constraints. In *Principles and Practice of Constraint Programming–CP99*, pages 480–481. Springer, 1999.
6. J.P. Gibbs and W.T. Martin. Urbanization, technology, and the division of labor: International patterns. *American Sociological Review*, 27(5):667–677, 1962.
7. C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1):67–100, 2000.
8. Belgian Constraints Group. Data and results for the steel mill slab problem. available from http://becool.info.ucl.ac.be/steelmillslab. Technical report, UCLouvain.
9. W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 607–615, 1995.
10. Pascal Van Hentenryck, Pierre Flener, Justin Pearson, and Magnus Ågren. Tractable symmetry breaking for csps with interchangeable values. In *IJCAI*, pages 277–284, 2003.
11. SA ILOG. ILOG Concert 2.0.
12. J.N. Monette, Y. Deville, and P. Van Hentenryck. Aeon: Synthesizing scheduling algorithms from high-level models. *Operations Research and Cyber-Infrastructure*, pages 43–59, 2009.
13. B.M. Smith, S.C. Brailsford, P.M. Hubbard, and H.P. Williams. The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared. *Constraints*, 1(1):119–138, 1996.
14. Gecode Team. Gecode: Generic constraint development environment. *Available from http://www.gecode.org*, 2006.
15. P. Van Hentenryck. Constraint and integer programming in OPL. *INFORMS Journal on Computing*, 14(4):345–372, 2002.
16. P. Van Hentenryck, P. Flener, J. Pearson, and M. Ågren. Compositional derivation of symmetries for constraint satisfaction. *Abstraction, Reformulation and Approximation*, pages 234–247, 2005.
17. P. Van Hentenryck and L. Michel. *Constraint-based local search*. The MIT Press, 2005.
18. P. Van Hentenryck and L. Michel. The steel mill slab design problem revisited. In *CPAIOR*, pages 377–381. Springer, 2008.
19. Pascal Van Hentenryck and Laurent Michel. Synthesis of constraint-based local search algorithms from high-level models. In *AAAI'07*, pages 273–278. AAAI Press, 2007.