# Symmetric Component Caching

**Matthew Kitching** and **Fahiem Bacchus**
Department of Computer Science,
University of Toronto, Canada.
[kitching|fbacchus]@cs.toronto.edu

## Abstract

Caching, symmetries, and search with decomposition are powerful techniques for pruning the search space of constraint problems. In this paper we present an innovative way of efficiently combining these techniques with branch and bound for solving certain types of constraint optimization problems (COPs). Our new method significantly reduces the overhead of performing decomposition during search when dynamic variable orderings are employed. In addition, it supports the exploitation of dynamic symmetries that appear only during search. Symmetries have not previously been combined with decomposition. Finally, we achieve a superior integration of decomposition and caching with branch and bound than previous approaches. We test our methods on the Maximum Density Still Life problem and show that each of our ideas yields a significant gain in search performance.

## 1 Introduction

As the variables of a constraint satisfaction problem (CSP) are assigned during backtracking search the problem can break into disjoint parts. Consider a CSP consisting of the variables $\{A,B,M,X,Y\}$ and two constraints $C_1(A,B,M)$ and $C_2(M,X,Y)$. If we make the assignment $M=m$, then the resulting *reduced CSP* will consist of two disjoint subproblems that share no variables. One subproblem is over the variables $A$ and $B$ with the constraint $C_1^{M=m}(A,B) = C_1(M{=}m,A,B)$ while the other subproblem is over $X$ and $Y$ with the constraint $C_2^{M=m}(X,Y) = C_2(M{=}m,X,Y)$. We call these disjoint subproblems created by variable assignments *components*. The two components created by the assignment $M=m$ can be solved independently: any solution $\langle A=a, B=b\rangle$ to the first, and solution $\langle X=x, Y=y\rangle$ to the second, can be combined with $M=m$ to obtain a solution to the original CSP $\langle A=a, B=b, M=m, X=x, Y=y\rangle$. Since the worst case complexity of solving a CSP is exponential in the number of variables, decomposition into components can yield significant computational gains.

This insight has been exploited to perform search where decomposition is applied recursively, e.g., [Bayardo & Pehoushek, 2000; Darwiche, 2001; Park & Gelder, 2000; Sang *et al.*, 2004; Dechter, 2004; Amir & McIlraith, 2000]. This can yield a reduction in the worst case complexity of search from $2^{O(n)}$ to $n^{O(w)}$, where $n$ is the number of variables of the problem and $w$ is the tree width of the CSP-graph (the graph determined by the constraint scopes) [Darwiche, 2001].

A component produced by decomposition at one point of the search tree might appear at many other nodes of the search tree. Caching allows us to solve each component once and then reuse that solution in the rest of the search. When caching is added to decomposition the worst case complexity of search can be further reduced, from $n^{O(w)}$ to $2^{O(w)}$ [Darwiche, 2001; Bacchus, Dalmao, & Pitassi, 2003]. This can be a significant speedup in practice (e.g., see the results in [Sang *et al.*, 2004]). Of course caching requires space, but in this case the space requirements are completely flexible. In particular, caching is used only to speedup the algorithm—it is not required for correctness. Thus we can cache as many solved components as we have space for—within certain practical limits the more space we have for caching the faster the search will proceed. Furthermore, we can always prune the cache of less useful items if available space is exhausted.

Decomposition and caching impose a significant overhead during search, but these techniques are still very effective for more complex constraint problems, e.g., finding the best solution (optimization) [Marinescu & Dechter, 2005] or counting the number of solutions [Sang *et al.*, 2004]. When a static variable ordering is used much of this overhead can be eliminated using data structures computed prior to search [Darwiche, 2001]. However, dynamic variable orderings can yield significant reductions in the size of the search tree, enough to pay off their added overhead (e.g., [Marinescu & Dechter, 2006]). Nevertheless, methods for reducing this overhead and for making these techniques even more effective are still needed to increase the practical benefits of dynamic decomposition.

In this paper we present a method for making search with decomposition and caching more effective in the context of dynamic variable orderings. Our key contribution involves a method for representing the common structure of an entire set of components in a single data structure we call a *component template*. The individual components that are instances of the template can thus share a single representation of their common information, making caching considerable more space efficient. With templates we can also increase the efficiency of cache lookup—we can access any of the template's instances by simple array indexing. Templates also allow us to perform component detection during search more efficiently.

Another final key contribution of this paper is a method for automatically detecting symmetries between templates. Once we have detected that two templates $T_1$ and $T_2$ are symmetric we can use the discovered symmetry mapping to map all instances of $T_1$ to a symmetric instance of $T_2$. Thus the template symmetry encodes an entire set of individual component symmetries. This makes computing template symmetries much more cost effective than computing symmetries between individual components. Template symmetries allow us to make more effective use of cached information—cached component bounds can now be used to provide bounds for all symmetric components as well. This can yield significant reductions in the size of the explored search space.

Finally, we show how to achieve a better integration between branch and bound and component caching by caching bounds on components rather than requiring that the components be completely solved prior to being stored in the cache (as done, e.g., in [Marinescu & Dechter, 2005]).

In the sequel we first expand on the background behind our approach. Then we present our new technique of component templates and show how they can be exploited so as to achieved the benefits just described. Then we present some initial empirical results on the test problem of Maximum Density Still Life.

## 2 Background

In this paper, we concentrate on solving Constraint Optimization Problems (COPs) that have decomposable objective functions. However, our template technique could be used in other applications of component caching (search with decomposition and caching), e.g., counting solutions.

**A Constraint Optimization Problem (COP)**, $\mathcal{P}$, is specified by a tuple $\langle \mathcal{V}, Dom, \mathcal{C}, O\rangle$, where $\mathcal{V}$ is a set of variables, for each $V \in \mathcal{V}$, $Dom[V]$ is its domain of values, $\mathcal{C}$ is a set of constraints, and $O$ is an objective function that assigns a value to all complete assignments of values to the variables. The typical goal in solving a COP is to find an assignment of values to the variables that maximizes $O$ and at the same time satisfies all of the constraints in $\mathcal{C}$.

Our techniques are effective on COPs that have *decomposable* objective functions and constraints. In particular, we require that $O$ be decomposed into the sum of objective subfunctions $o_i$, $O = \sum_i o_i$, such that each $o_i$ is dependent on only a subset of the variables $scope(o_i) \subset \mathcal{V}$, and that each constraint $C \in \mathcal{C}$ also be dependent on only a subset of the variables $scope(C) \subset \mathcal{V}$.[1] The constraints and objective sub-functions can be unified by treating each constraint as an additional objective sub-function mapping satisfying assignments to 0 and violating assignments to $-\infty$. Thus the overall (unified) objective function becomes $\mathcal{O} = \sum_i o_i + \sum_j C_j$, and the problem is simply to maximize this augmented objective function. (This formulation is like a soft-constraint problem). Hence, in this paper we regard a COP as being the tuple $\langle \mathcal{V}, Dom, \mathcal{O}\rangle$ where $\mathcal{O}$ includes both the original objective sub-functions and the hard constraints encoded as additional objective sub-functions. We use the term *objectives* to denote the sub-functions in $\mathcal{O}$.

[1]If the objective function or constraints cannot be decomposed, our method will still be correct but no decompositions will be generated during search. Often, however, the objective and global constraints can be reformulated in a decomposed form.

Note that we could equally use an objective function that is decomposed into a product rather than a sum of objectives; similarly we could cast the problem as a minimization task rather than a maximization task.

**Components.** In backtracking search each node of the search tree $n$ corresponds to a set of variable assignments, and these assignments might cause the problem to be broken up into disjoint components. A component is a subset of the original problem that has been isolated by a set of assignments. Here we present a formalization of this idea tailored to our subsequent developments.

A *component*, $\rho$, of a COP $\mathcal{P} = \langle \mathcal{V}, Dom, \mathcal{O}\rangle$ is a tuple $\langle \rho.\mathcal{V}, \rho.\mathcal{O}, \rho.\mathcal{A}\rangle$ where $\rho.\mathcal{V} \subseteq \mathcal{V}$, $\rho.\mathcal{O} \subseteq \mathcal{O}$ and $\rho.\mathcal{A}$ is a set of assignments $\{V_i = a_i, \ldots\}$, subject to the following conditions. Let $vars(\rho.\mathcal{A})$ be the set of variables assigned values in $\rho.\mathcal{A}$.

1. $\rho.\mathcal{O} = \bigcup_{V\in\rho.\mathcal{V}}\{o | V \in scope(o)\}$
2. $vars(\rho.\mathcal{A}) \cup \rho.\mathcal{V} = \bigcup_{o\in\rho.\mathcal{O}} scope(o)$
3. $vars(\rho.\mathcal{A}) \cap \rho.\mathcal{V} = \emptyset$
4. p is minimal. That is, there is no other tuple $p' = \langle \rho'.\mathcal{V}, \rho'.\mathcal{O}, \rho'.\mathcal{A}\rangle$ that satisfies conditions 1-3 with $\rho'.\mathcal{V} \subseteq \rho.V$, $\rho'.\mathcal{O} \subseteq \rho.\mathcal{O}$, $\rho'\mathcal{A} \subseteq \rho.\mathcal{A}$ and with at least one of these sets being a strict subset.

That is, a component contains a set of variables and all of the objectives over these variables. Furthermore, these objectives are isolated from the rest of the problem by a set of assignments $\rho.\mathcal{A}$: all of the variables they mention are either variables in the component or are instantiated in $\rho.\mathcal{A}$. Furthermore none of the variables of the component, i.e., the variables in $\rho.\mathcal{V}$, are assigned in $\rho.\mathcal{A}$.

**The Components at a node.** At each node $n$ some set of assignments $\mathcal{A}$ have been made, and some set of variables $\mathcal{U}$ remain unassigned. The components at $n$, $\rho_1, ...\rho_k$, are those components such that (a) $\rho_i.\mathcal{V} \subseteq \mathcal{U}$ (the variables of the components are unassigned), (b) $\rho_i.\mathcal{A} \subseteq \mathcal{A}$ (the current assignments isolate the variables of the component).

**Example 1** *Consider a COP with variables $\{A, B, M, N, X, Y\}$ and constraints $C_1(A, B, M)$, $C_2(X, Y, N)$, and $C_3(M, N)$. At a node n where just the assignment M=a has been made, there will be two components: $\rho_{ab}$ with $\rho_{ab}.\mathcal{V} = \{A,B\}$, $\rho_{ab}.\mathcal{O} = \{C_1\}$, and $\rho_{ab}.\mathcal{A} = \{M = a\}$ and $\rho_{xyn}$ with $\rho_{xyn}.\mathcal{V} = \{X,Y,N\}$, $\rho_{xyn}.\mathcal{O} = \{C_2, C_3\}$, and $\rho_{xyn}.\mathcal{A} = \{M = a\}$.*

**Computing the components** at a node $n$ is easily accomplished by standard algorithms for detecting the CONNECTED COMPONENTS of a graph, e.g., union-find or depth-first search. In particular, we consider an undirected graph $G_n$, that contains a node for every objective in $\mathcal{O}$ and for every **uninstantiated** variable $V$. There is an edge between two nodes in $G_n$ if and only if one of the nodes is a variable node $V$, the other is an objective node $o$, such that $V \in scope(o)$.

**Observation 1** *The CONNECTED COMPONENTS of $G_n$ correspond to the minimal components of $n$. In particular, $\rho$ is a component at $n$ if and only there exists a connected component $c$ of $G_n$ such $\rho.\mathcal{V}$ is the set of variable nodes in $c$, $\rho.\mathcal{O}$ is* *the set of objective nodes of $c$, and $\rho.\mathcal{A}$ are the assignments made to the instantiated variables of these objectives.*

This observation can be verified by realizing that conditions (1) and (2) in the definition of a component (above) can be achieved by incrementally adding connected objectives to $\rho.\mathcal{O}$ and connected variables to $\rho.\mathcal{V}$ stopping when there are no more connections to follow. This is precisely what CONNECTED COMPONENTS algorithms do (these algorithms also compute minimal components). $\rho.\mathcal{A}$ can then computed after $\rho.\mathcal{V}$ and $\rho.\mathcal{O}$ have been finalized. Note also that any objective that has been fully instantiated at the node $n$ will become an isolated node in $G_n$; i.e., a single node component. These fully instantiated objectives form components with no variables, one objective, and the assignments required to fully instantiate that objective.

**Example 2** *Suppose in our example we next assign $N = b$, so that only the two assignments $M = a$ and $N = b$ have been made. This will generate 3 components: $\rho_{ab}$ is unaffected by the new assignment, but $\rho_{xyn}$ is now split into two components $\rho_{xy} = \langle\{X, Y\}, C_2, N = b\rangle$ and $\rho_{mn} = \langle\{\}, C_3, \{N = b, M = a\}\rangle$. Note the $\rho_{mn}$ contains no variables, just a single fully instantiated objective, and that the component $\rho_{xy}$ does not contains $M = a$ in its assignment set even though its parent component $\rho_{xyn}$ did.*

**Computation Benefits of Components.** If $\rho$ is a component then the value of any assignment $\mathcal{A}$ to its variables, $\rho.\mathcal{V}$, is equal to the sum of its objectives $\rho.\mathcal{O}$ evaluated at the set of assignments $\mathcal{A} \cup \rho.\mathcal{A}$. Note that the objectives of $\rho$ are functions only of $\rho.V$ and the assignments in $\rho.\mathcal{A}$, thus any complete assignment to $\rho.\mathcal{V}$ along with $\rho.\mathcal{A}$ is sufficient to fully instantiate all of these objectives (yielding a single numeric value for each objective which we can then sum). The **value** (maximal value) of $\rho$, $value(\rho)$, is the maximum value that can be achieved by any assignment to its variables $\rho.\mathcal{V}$:

$$value(\rho) = \max_{\mathcal{A}:\mathcal{A} \text{ is an assignment to } \rho.\mathcal{V}} \sum_{o \in \rho.\mathcal{O}} o(\mathcal{A} \cup \rho.\mathcal{A}).$$

A *solution* to the component is any assignment to its variables that achieves its (maximal) value. Note that components corresponding to fully instantiated objective have a value equal to the value of the instantiated objective. Note also that the value of a component can be computed by examining assignments to the component's variables only, the rest of the problem can be ignored.

**Proposition 1** *Let $n.\mathcal{A}$ be the set of assignments made at a node $n$, and let $\rho_1, \ldots, \rho_k$ be the set of components at $n$. The maximal value that can be obtain for any complete set of assignments $\mathcal{A}$ to the variables of the COP $\mathcal{P}$ such that $\mathcal{A} \supseteq n.\mathcal{A}$ (i.e., at any leaf node in the subtree below $n$), is $\sum_{i=1}^{k} value(\rho_i)$. Furthermore a complete assignment $\mathcal{A}$ achieves this maximal value if and only if it is equal to $n.\mathcal{A}$ unioned with a solution for each component $\rho_i$.*

This proposition follows from the fact that the values of the components are independent of each other. Computationally, this means that we can solve each component independently, and that we obtain further computational advantage by applying decomposition recursively as we solve the individual components.

## 3 Templates

Now we introduce our new idea of component templates used to represent the shared information of a group of components, each of which then becomes an instance of the template. The basic idea is quite simple, with most of the innovation arising from how templates can be exploited algorithmically.

From the definitions above it can be observed that for any two components $\rho_1$ and $\rho_2$ if $\rho_1.\mathcal{V} = \rho_2.\mathcal{V} \neq \emptyset$ then $\rho_1.\mathcal{O} = \rho_2.\mathcal{O}$: both of these sets are all the objectives mentioning these variables (and no others due to minimal). Furthermore, the variables assigned in $\rho_1.\mathcal{A}$ are identical to the variables assigned in $\rho_2.\mathcal{A}$: both of these sets must assign all variables of the $\rho_i.\mathcal{O}$ not in $\rho_i.\mathcal{V}$. In fact the only difference between two non-equal components containing the same (non-empty) set of variables is that the particular *values* assigned in $\rho_1.\mathcal{A}$ and $\rho_2.\mathcal{A}$ differ, and as a consequence $value(\rho_1)$ and $value(\rho_2)$ might also differ (since the objectives are being maximized subject to the differing values in $\rho_1.\mathcal{A}$ and $\rho_2.\mathcal{A}$).

We use component templates to represent all the components that have an identical set of variables. Formally, a component template $\mathcal{T} = \langle\mathcal{T}.\mathcal{V}, \mathcal{T}.\mathcal{O}, \mathcal{T}.D\rangle$ is a set of variables $\mathcal{T}.\mathcal{V}$, objectives $\mathcal{T}.\mathcal{O}$, and another set of variables $\mathcal{T}.D$ disjoint from $\mathcal{T}.\mathcal{V}$ called the *dependency variables*, such that every set of assignments $\mathcal{A}$ to the variables in $\mathcal{T}.D$ (a) generates an instance of the template $\mathcal{T}\langle\mathcal{A}\rangle$ and (b) every instance $\mathcal{T}\langle\mathcal{A}\rangle$ is a component: $\mathcal{T}\langle\mathcal{A}\rangle = \langle\mathcal{T}.\mathcal{V}, \mathcal{T}.\mathcal{O}, \mathcal{A}\rangle$. That is, the instance is a component with the same variables and objectives as the templates and with $\mathcal{A}$ being the set of assignments that disconnects it from the rest of the problem.

**Example 3** *For instance, consider the component $\rho_{ab} = \langle\{A, B\}, C_1, M = a\rangle$ seen in the previous example. The component template $\mathcal{T}_{ab} = \langle\{A, B\}, \{C_1\}, \{M\}\rangle$ includes $\rho_{ab}$ as one of its instances. In particular $\mathcal{T}_{ab}\langle M = a\rangle = \rho_{ab}$.*

**Using Templates During Search**. As described above, the components at each node $n$ of a backtracking search can be determined by a CONNECTED COMPONENTS algorithm run on the graph $G_n$. Note however that $G_n$ contains only variables and objectives, it does not mention the actual values assigned to the instantiated variables. Hence the algorithm actually identifies a set of templates. The components at the node $n$ are the **particular instances of these templates** determined by the assignments at $n$.

Once a template is detected for the first time we create a data structure to represent the template instance and store this in a template cache. This data structure can then be used to efficiently detect when any of its instances are components of future nodes in the search as follows.

**Observation 2** *Let $\mathcal{A}$ be the set of assignments made at node $n$, and let $\mathcal{T}$ be a component template. If $\mathcal{A}$ instantiates all of the variables in $\mathcal{T}.\mathcal{D}$ and none of the variables in $\mathcal{T}.\mathcal{V}$, then $\mathcal{T}\langle\mathcal{A}\rfloor_{\mathcal{T}.\mathcal{D}}\rangle$ is one of the components at $n$, where $\mathcal{A}\rfloor_{\mathcal{T}.\mathcal{D}} = \{V{=}a | V{=}a \in \mathcal{A} \wedge V \in \mathcal{T}.\mathcal{D}\}$ is the subset of $\mathcal{A}$ that assigns the variables in $\mathcal{T}.\mathcal{D}$.*

We can efficiently detect when all of the variables in $\mathcal{T}.\mathcal{D}$ are instantiated at a node using standard lazy watch techniques [Moskewicz *et al.*, 2001]. Once all of $\mathcal{T}.\mathcal{D}$ has been assigned we test $\mathcal{T}.\mathcal{V}$. If all of these variables are unassigned then $\mathcal{T}$ has been *triggered*, and we know that $\mathcal{T}.\mathcal{V}$ forms a component at the current node. All of the template's variables and objectives can then be removed from $G_n$, further reducing its size. CONNECTED COMPONENTS can then be run on this smaller remaining graph to identify the other components at the current node. Triggering components and reducing the size of $G_n$ in this way can yield a non-trivial improvement in the total time needed to perform component detection.

Once a template has been triggered, we need to access information about the particular instance that exists at the current search node, $\mathcal{T}\langle\mathcal{A}\downharpoonright_{\mathcal{T}.\mathcal{D}}\rangle$. Associated with each template is a *value cache* that is used to store upper and lower bounds on the values of its instances (solutions can also be stored for solved instances). If $\mathcal{T}\langle\mathcal{A}\rangle$ is a template instance, then $\mathcal{T}\langle\mathcal{A}\rangle.lb$ and $\mathcal{T}\langle\mathcal{A}\rangle.ub$ will denote the stored lower and upper bounds on $value(\mathcal{T}\langle\mathcal{A}\rangle)$. If the instance has never been seen before, these bounds are given some default initial values. The search in the subtree below the current node will either compute the value of the instance (making $\mathcal{T}\langle\mathcal{A}\rangle.lb = \mathcal{T}\langle\mathcal{A}\rangle.ub$), compute better bounds on its value, or backtrack without updating these bounds. Once we have the template, accessing an instance's bounds can be very efficient. In particular, each variable in $\mathcal{T}.\mathcal{D}$ has a finite domain of values, and each instance $\mathcal{T}\langle\mathcal{A}\rangle$ is defined by the values in $\mathcal{A}$ it assign to these variables. Thus we can use an instance's defining sequence of values as an index into a multi-dimensional array. Many instances of the template might, however, never be encountered during the search (because of branch and bound pruning), so if such an array would be too large we can use the instance's sequence of values as a hash code to index into a small hash table more suitable for storing sparse data.

**Example 4** *In our previous example, when we set $M = a$ for the first time, we create a new template $\mathcal{T}_{ab}$ with $\mathcal{T}_{ab}.\mathcal{V} = \{A,B\}$, $\mathcal{T}_{ab}.\mathcal{O} = \{C_1\}$, and $\mathcal{T}_{ab}.D = \{M\}$ which represent component $\rho_{ab}$. An instance of the template $\mathcal{T}_{ab}\langle M = a\rangle$ is immediately created. Search proceeds over the variables A and B, returning the upper and lower bounds of component $\rho_{ab}$ under the instantiation $M = a$. These are bounds on the maximal value that can be achieved by the objectives in $\mathcal{T}_{ab}.\mathcal{O}$ over all possible values for A and B subject to $M = a$. These bounds are stored in the template cache as $\mathcal{T}_{ab}\langle M = a\rangle.lb$ and $\mathcal{T}_{ab}\langle M = a\rangle.ub$, i.e., as bounds indexed by the assignment $M = a$.*

*If we assign $M = k$ later in search with A and B still unassigned then $\mathcal{T}_{ab}$ is triggered and the bounds on the new instance $\mathcal{T}_M\langle M = k\rangle$ are retrieved from the template's value cache. If $k = m$ then the cached upper and lower bounds can be reused at this new search node.*

**Template Symmetries** Another key computational use of templates is to perform automatic symmetry detection between components at the abstract level of templates. In particular, we compute symmetries between templates during search (templates are only created during search). A symmetry between two templates can then be applied to all of their instances, thus allowing us to amortize a single symmetry computation over many different components. This is key in making automatic symmetry detection cost effective. To further reduce the cost of symmetry detection we confine ourselves to variable symmetries rather than finer-grain symmetries defined over variable-values.

Formally, we require a symmetry between two templates $\mathcal{T}_1$ and $\mathcal{T}_2$ to be a one-to-one and onto mapping $\sigma$ between the variables $\mathcal{T}_1.\mathcal{D} \cup \mathcal{T}_1.\mathcal{V}$ and $\mathcal{T}_2.\mathcal{D} \cup \mathcal{T}_2.\mathcal{V}$ such that

1. $\mathcal{T}_2.\mathcal{D} = \sigma(\mathcal{T}_1.\mathcal{D})$ and $\mathcal{T}_2.\mathcal{V} = \sigma(\mathcal{T}_1.\mathcal{V})$, where $\sigma$ applied to a set $S$ is $\sigma(S) = \{\sigma(V)|V \in S\}$.
2. for any assignment $\mathcal{A}$ to all of the variables in $\mathcal{T}_1.\mathcal{D} \cup \mathcal{T}_1.\mathcal{V}$, the value of the objectives $\mathcal{T}_1.\mathcal{O}$ evaluated at $\mathcal{A}$ is identical to the value of the objectives of $\mathcal{T}_2.\mathcal{O}$ evaluated at $\sigma(\mathcal{A})$, where $\sigma$ applied to a set of assignments $\mathcal{A}$ is $\sigma(\mathcal{A}) = \{\sigma(V) = a|V = a \in \mathcal{A}\}$.

In otherwords, $\sigma$ keeps the dependency variables and template variables separated, and it preserves the value of the template objectives. Since the value of a template instance $\mathcal{T}_1\langle\mathcal{A}\rangle$ is the maximum of the sum of the objectives $\mathcal{T}_1.\mathcal{O}$ under the fixed assignment $\mathcal{A}$ to the dependency variables $\mathcal{T}_1.\mathcal{D}$ and any assignment to $\mathcal{T}_1.\mathcal{V}$, we have that

**Observation 3** *If $\mathcal{T}_1$ and $\mathcal{T}_2$ are symmetric under the mapping $\sigma$ then for any instance of $\mathcal{T}_1$, $\mathcal{T}_1\langle\mathcal{A}\rangle$ we have $value(\mathcal{T}_1\langle\mathcal{A}\rangle) = value(\mathcal{T}_2\langle\sigma(\mathcal{A})\rangle)$.*

This means that any bounds we have computed for the component $\mathcal{T}_2\langle\sigma(\mathcal{A})\rangle$ can be reused for the component $\mathcal{T}_1\langle\mathcal{A}\rangle$.

To automatically detect symmetries between templates during search we map the problem to a graph isomorphism problem by constructing a graph representation for each template. The graph representation has the property that two templates' graph representations are graph isomorphic if and only if the templates are symmetric in the sense defined above. The graph isomorphism, which maps the vertices of one graph to the other, provides the variable to variable symmetry mapping between the two templates.

We solve the graph isomorphism problem using available graph automorphism software (in our case NAUTY [McKay, 2004]). As shown in [Puget, 2005] such software can be surprising efficient even though graph isomorphism is not known to be of polynomial complexity. To utilize symmetries during search we proceed as follows. When a template $\mathcal{T}$ is first created, we construct its graph representation $G_{\mathcal{T}}$. NAUTY is then used to compute $iso(G_{\mathcal{T}})$, a canonical isomorph of $G_{\mathcal{T}}$. $\mathcal{T}$ is symmetric to some previously cached template $\mathcal{T}'$ if and only if their graph isomorphs are equal, $iso(G_{\mathcal{T}}) = iso(G_{\mathcal{T}'})$. By using hashing techniques on $iso(G_{\mathcal{T}})$ we can find any isomorphic template in near constant time. If an isomorphic template $\mathcal{T}'$ is found we utilize the invertible mappings $G_{\mathcal{T}'} \mapsto iso(G_{\mathcal{T}'})$ and $G_{\mathcal{T}} \mapsto iso(G_{\mathcal{T}})$ produced by NAUTY to compute a symmetry map from $\mathcal{T}$ to $\mathcal{T}'$.

To utilize this symmetry, we avoid allocating a value cache for $\mathcal{T}$'s instances. Instead we mark $\mathcal{T}$ as being symmetric to $\mathcal{T}'$ and store the symmetry map $\sigma$. We continue to use $\mathcal{T}$ to detect when any of its instances are created, but, transparently to the rest of the code, whenever we read or store information about one of $\mathcal{T}$'s instances we instead remap that access to the symmetric instance of $\mathcal{T}'$. Thus, all instances of $\mathcal{T}$ are able to utilize bounds computed for instances of $\mathcal{T}'$. Furthermore any bound computed for instances of $\mathcal{T}$ are stored as information about instances of $\mathcal{T}'$. Since we might have many different templates being symmetric to $\mathcal{T}'$ (symmetries always map to the earliest created template), this means that information computed for an instance of $\mathcal{T}$ can then be utilized by many other symmetric components.

**Example 5** *Consider the same COP described earlier. When we set $N = b$ for the first time, we create a new template $\mathcal{T}_{xy}$ with $\mathcal{T}_{xy}.\mathcal{V} = \{X, Y\}$, $\mathcal{T}_{xy}.\mathcal{O} = \{C_2\}$, and $\mathcal{T}_{xy}.D = \{N\}$ which represent component $\rho_{xy}$. If $C_1$ is the same type of constraint as $C_2$, then the graph representation of $\mathcal{T}_{ab}$ and $\mathcal{T}_{xy}$ will be the same, and an isomorphism between the two templates will be the found, in this case mapping variable N to M. The cache of template $\mathcal{T}_{xy}$ will point to the cache of template $\mathcal{T}_{ab}$ under the mapping $N \mapsto M$. If we create an instance $\mathcal{T}_{xy}\langle N = a\rangle$ we can then use the cached results $\mathcal{T}_{ab}\langle M = a\rangle.lb$ and $\mathcal{T}_{ab}\langle M = a\rangle.ub$.*

*Note that in the original COP (Example 1), if N and M are not exchangable in $C_3(N, M)$, then this symmetry does not exist until both M and N are assigned. That is, our mechanism can detect dynamic symmetries.*

Unfortunately space restrictions prohibit us from presenting the details of our template graphical representation. But we will mention that many choices for this representation exist. Our representation utilizes NAUTY's ability to input coloured graphs to ensure that the computed graph symmetries keep the dependency variables and template variables separated (condition (1) above), it allows exploitation of the fact that some objectives have exchangeable variables,[2] and it uses colours to ensure that only equivalent objectives can map to each other.

## 4 Symmetric Component Caching Search

The search algorithm given in Figure 1 shows how we utilize the above ideas to perform recursive decompositions integrated with branch and bound. CCS+BB attempts to find the value for a single component given as a template instance $\mathcal{T}\langle\mathcal{A}\rangle$. It is also provided with a lower bound LB, and can abort its computation as soon as it discovers that $\mathcal{T}\langle\mathcal{A}\rangle$ cannot achieve a value greater than this bound. Even if the computation is aborted, however, the routine still stores the best bounds it was able to compute before termination, line 14. Storing the bounds produced by a partial computation of a component's value allows a better integration with branch and bound. As described above, these bounds are stored in the template's value cache (or in some symmetric template's value cache).

Branch and bound inevitably implies that the search might attempt to compute the value of a particular template instance many times, aborting each attempt because of the current bound. By updating the component's bounds after each of these attempts work is saved. In particular, each new attempt can proceed more efficiently by utilizing the stronger bounds computed in the previous attempt. This approach is in contrast with that presented in [Marinescu & Dechter, 2005], where the cache is used only to store the values of fully solved components. Hence, default initial bounds are always used when trying to solve an unsolved component, even when the search had computed better bounds in a previous attempt.

[2]Two variables $X_i$ and $X_j$ are exchangeable in a objective if exchanging their assigned values has no effect on the objective.

```
CCS+BB($\mathcal{T}\langle\mathcal{A}\rangle$, LB)
1.  if $\mathcal{T}.\mathcal{V} = \emptyset \vee (\mathcal{T}\langle\mathcal{A}\rangle.ub \leq$ LB$) \vee (\mathcal{T}\langle\mathcal{A}\rangle.lb = \mathcal{T}\langle\mathcal{A}\rangle.ub)$
2.      return
3.  if LB$< \mathcal{T}\langle\mathcal{A}\rangle.lb$    LB := $\mathcal{T}\langle\mathcal{A}\rangle.lb$
4.  $V$ := select variable from $\mathcal{T}.\mathcal{V}$ to branch on
5.  Ts := Find the templates contained in the graph
          consisting of $T.Ob$ and $T.V - \{V\}$.
6.  foreach $d \in Dom[V]$
7.      foreach $\mathcal{T}_i \in$ Ts
8.          $\mathcal{A}_i$ := $(\mathcal{A} \cup \{V = d\})\downarrow_{\mathcal{T}_i.\mathcal{D}}$
9.      foreach $\mathcal{T}_i \in$ Ts while $\sum_i \mathcal{T}_i\langle\mathcal{A}_i\rangle.ub >$ LB
10.         LB$_i$ := LB-$\sum_{j\neq i}\mathcal{T}_j\langle\mathcal{A}_j\rangle.ub$
11.         CCS+BB($\mathcal{T}_i\langle\mathcal{A}_i\rangle$, LB$_i$)
12.     (lb$^d$, ub$^d$) = $(\sum_i \mathcal{T}_i\langle\mathcal{A}_i\rangle.lb, \sum_i \mathcal{T}_i\langle\mathcal{A}_i\rangle.ub)$
13.     LB := max(LB, lb$^d$)
14. $(\mathcal{T}\langle\mathcal{A}\rangle.lb, \mathcal{T}\langle\mathcal{A}\rangle.ub)$ = $(\max_d($lb$^d), \max_d($ub$^d))$
```

Figure 1: Template Component Caching Search with Branch and Bound and Templates. Try to compute $value(\mathcal{T}\langle\mathcal{A}\rangle)$, but give up as soon as we discover that $value(\mathcal{T}\langle\mathcal{A}\rangle) \leq$ LB.

The first thing the algorithm does (line 1) is to check if (a) the template contains no variables (in which case it contains only a single objective function that is fully instantiated by the assignments $\mathcal{A}$, hence $\mathcal{T}\langle\mathcal{A}\rangle.lb$ and $\mathcal{T}\langle\mathcal{A}\rangle.ub$ are both equal to the value of the objective on $\mathcal{A}$); (b) $\mathcal{T}\langle\mathcal{A}\rangle.ub \leq$ LB (in which case the required bound cannot be achieved); or (c) $\mathcal{T}\langle\mathcal{A}\rangle.lb = \mathcal{T}\langle\mathcal{A}\rangle.ub$ (in which case the component has already been fully solved, which actually captures case (a) as well); in any of these cases the procedure can immediately return.

Otherwise we check if it is already known that the component can achieve a higher value than required (LB), line 3. In this case the algorithm is going to compute the component's maximum value (the computation cannot be aborted by bounding), and we can make it more efficient by resetting the lower bound to the higher value. A variable $V$ to split on is then selected, and we determine how the component will decompose when this variable is assigned, line 5. This is done by a combination of template triggering and connected component analysis. Once the triggered templates are removed from the constraint graph each remaining connected component forms a new template. Any objective of the input template $o \in \mathcal{T}.\mathcal{O}$ whose only uninstantiated variable was $V$ forms a template $\mathcal{T}'$ with $\mathcal{T}'.\mathcal{V} = \emptyset$ and $\mathcal{T}'.\mathcal{O} = \{o\}$. Any instance of this template, $T'\langle\mathcal{A}\rangle$, will have upper and lower bounds equal to $o$ evaluated at $\mathcal{A}$. The other components, containing variables, will generate new templates upon which automatic symmetry detection is performed. As $V$ is assigned different values, different instances of the templates in Ts will be created and solved.

Lines 7 and 8, identify the new template instances (components) created by the current value of $V$. These components are solved in lines 9–11. We can abandon the assignment $V{=}d$ (line 9) if at any time the sum of the upper bounds of these components fails to reach the required lower bound (this sum changes as we compute the component values). We need from each component a value large enough so that if the other components reach their upper bound, the sum will exceed the lower bound required for the input component. This is the value $\mathsf{LB}_i$ computed at line 10. When we have finished with $V{=}d$ we compute bounds that can be achieved for the input component under this setting of $V$, line 12. It could be that these bounds exceed the inputed lower bound, i.e., $\mathtt{lb}^d > \mathsf{LB}$. In that case, the algorithm will continue to compute the input component's maximum value, by trying the remaining assignments to $V$. However, since $V = d$ achieves at least $\mathtt{lb}^d$, any new value for $V$ needs to achieve an even higher value, hence we can reset the lower bound (line 13) before attempting the next assignment to $V$. Finally, after all values for $V$ have been tried, we can update the bounds for the input component. Note that if the computation succeeded in solving the component, then for some value $d$ of $V$ achieving the maximum we will have $\mathtt{lb}^d = \mathtt{ub}^d$ and $\mathtt{ub}^{d'} \leq \mathtt{lb}^d$ for all $d' \neq d$. Hence line 14 will set $\mathcal{T}\langle\mathcal{A}\rangle.lb = \mathcal{T}\langle\mathcal{A}\rangle.ub = value(\mathcal{T}\langle\mathcal{A}\rangle)$ as required.

**Constraint Propagation.** The algorithm can also employ constraint propagation over the hard objectives (the original constraints) to prune domain values that would necessarily lead to values of $-\infty$. In particular, domain pruning will simply reduce the iterations we have to perform at line 6, the algorithm requires no other changes. It can be proved that this simple change is all that is needed, but here we provide only the intuition behind the proof.

Since the values pruned by constraint propagation violate hard constraints and thus make the sum of the objectives $-\infty$, it can be seen that the parts of the search space that are avoided by not branching on the pruned values cannot contain an optimal solution to the COP. Thus these pruned parts of the space cannot cause the algorithm to miss an optimal solution.

The other part of the proof is to ensure that the parts of the space that are explored continue to compute the same values when propagation is used, so that overall the search still computes the optimal value. Here the key element is showing that cached template values remain correct even when some the domains of its unassigned variables $\mathcal{T}.\mathcal{V}$ have been reduced by constraint propagation. This can be seen by realizing that the template values correspond to a maximization over all possible assignments of the unassigned variables (subject to some fixed setting of the dependency variables). Since pruned values violate hard constraints, the maximum cannot have arisen from any of the pruned values. That is, the maximization over all possible values of the template variables is equal to the maximization over all unpruned values of the template variables, and the cached template values remain correct.

This highlights a subtely in the template algorithm. A template $\mathcal{T}$ cannot be triggered if any variable $Var \in \mathcal{T}.\mathcal{V}$ is assigned a value. This is because the maximum value stored in $\mathcal{T}$'s value cache might require that $Var$ be assigned a different value. That is, $\mathcal{T}$'s cached values will might not be valid when $Var \in \mathcal{T}.\mathcal{V}$ has been assigned. In contrast, if $Var$'s domain has been reduced to a singleton by propagation, its value is *forced* and as we just explained $\mathcal{T}$'s cached values are still valid. In this case $\mathcal{T}$ can be triggered. Hence, our algorithm must treat variables with singleton domains (after propagation) differently from variables that have been assigned.

## 5 Empirical Results

We have implemented our approach and tried it on the *Maximum Density Still Life* problem (MDSL) [Larrosa, Morancho, & Niso, 2005]. This problem involves finding maximum density stable configurations in Conway's game of Life. In particular, we have a $N \times N$ grid of cells that can be either dead or alive, implicitly surrounded by a boundary of dead cells. A cell is live and stable if it is surrounded by 2 or 3 live cells, it is dead and stable if it is surrounded by any other number of live cells. The goal is to find a stable configuration of the cells that has a maximum number of live cells.

State of the art solvers for this problem [Larrosa, Morancho, & Niso, 2005] employ an extensive amount of problem specific information. However, our solution is entirely generic; our aim is to use the problem to evaluate the effectiveness of our techniques. We use the most basic representation of the problem with $N^2$ boolean variables, each one representing the alive/dead status of a single cell, $N^2$ unary objective functions, one for each cell, assigning 1 to live cells and 0 to dead cells, and $N^2$ hard objective functions, one for each cell, that assigns 0 to stable settings of the cell given its neighbour's values and $-\infty$ to unstable settings.

We solved MDSL using six different algorithms always performing GAC constraint propagation on the hard objectives to prune domain values. All experiments were run on a 2.2 GHz Pentium IV with 6GB of memory. The time limit for each run was set to 10,000 seconds.

The algorithms we tested were. (1) a standard Branch and Bound algorithm (**BB**) which uses the variable ordering heuristic of domain size plus degree (both computed dynamically). The remaining algorithms use a heuristic that is biased toward completing rows or columns that have already been started in the grid; this tends to encourage the generation of components. We chose domain plus degree for **BB** because it performs better with this heuristic. (2) Component Branch and Bound (**C+BB**). This version searches for components and solves them separately, i.e., it performs search with decomposition. However, it does not use templates, nor does it cache already solved components. (3) Template Branch and Bound (**T+BB**) is a template version of **C+BB**. Its only improvement over **C+BB** is the use of templates to improve component detection. There is no template cache to store bounds for the instances. (4) Component Caching Search + Branch and Bound (**CCS+BB**) extends **T+BB** by activating the template cache to store computed bounds for the instances. (5) Symmetric Component Caching Search + Branch and Bound (**SCCS+BB**) extends **CCS+BB** by performing automated symmetric detection between templates. (6) **ASCCS+BB** which extends **SCCS+BB** to exploit automorphisms as well as symmetries. In particular, automorphisms (also computed by NAUTY) tell us that different instances of the *same* template are equivalent, so we can utilize cached instance bounds for all automorphic instances (as well as for instances of symmetric templates).

| *Size* | **ASCCS+BB** | | **SCCS+BB** | | **CCS+BB** | | **T+BB** | | **C+BB** | | **BB** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *nodes* | *time* | *nodes* | *time* | *nodes* | *time* | *nodes* | *time* | *nodes* | *time* | *nodes* | *time* |
| 4 | 486 | 0 | 1076 | 0 | 1646 | 0 | 1646 | 0 | 1646 | 0 | 767 | 0 |
| 5 | 3991 | 0 | 7744 | 0 | 7744 | 0.1 | 8146 | 0.1 | 8146 | 0.24 | 5364 | 0 |
| 6 | 18299 | 0.2 | 41346 | 0.5 | 93335 | 1.1 | 206141 | 2.0 | 177540 | 2.4 | 177522 | 2.0 |
| 7 | 251522 | 3.4 | 492939 | 6.5 | 644175 | 8.5 | $1.5{*}10^6$ | 19.8 | $2.1{*}10^6$ | 41.8 | $4.7{*}10^6$ | 55.1 |
| 8 | $9.7{*}10^6$ | 13.8 | $2.0{*}10^6$ | 29.3 | $6.1{*}10^6$ | 82.3 | $2.4{*}10^7$ | 316.6 | $2.3{*}10^7$ | 408.8 | $3.1{*}10^8$ | 2207 |
| 9 | $1.5{*}10^7$ | 408.7 | $3.1{*}10^7$ | 704.8 | $9.7{*}10^7$ | 4662.4 | NA | $> 10000$ | NA | $> 10000$ | NA | $> 10000$ |
| 10 | $5.8{*}10^7$ | 1540.5 | $1.2{*}10^8$ | 3003 | $5.6{*}10^8$ | 7845.4 | NA | $> 10000$ | NA | $> 10000$ | NA | $> 10000$ |

Table 1: Nodes Searched and Time taken in CPU secs.

Table 1 shows the relative performance of the six algorithms in terms of nodes searched and time taken. We see that decomposition (**C+BB**) yields significant improvements over standard branch and bound (**BB**) decreasing the size of the search tree. For the smaller problems **C+BB** takes more time, due to its larger overhead, but this overhead is quickly recouped as the problems get large and the reduction in the search tree achieved by decomposition becomes more significant. Since **T+BB** does not employ caching, it does not reduce the size of **C+BB**'s search tree (except for heuristic reasons). But it does provide a non-trivial improvement in efficiency. In particular, **T+BB** shows that template triggering does improve the efficiency of detecting components. The times in the table show that the overhead of detecting components is non-trivial. **CCS+BB** makes a further improvement by activating storage of bounds in the template, which results in a significant decrease in the size of the search space over **T+BB**. There is a corresponding decrease in time. **SCCS+BB** makes an improvement by performing symmetric detection between templates providing another significant decrease in the size of the search space. **ASCCS+BB** performs automorphism detection within a single template in addition to symmetries between template and provides another useful performance gain. We also see that on this problem automated symmetry detection is not adding a significant overhead. That is, the nodes/second search rate with symmetry detection is only 5% less than the search rate without symmetry detection on the largest problem. In all of these cases, the size of the cached information was never a problem. So we did not have to implement any strategy for pruning the cache.

## 6 Conclusions

We have presented an algorithm which incorporates search with decomposition, caching, and symmetric use of the cache, while mitigating much of the computational cost associated with such techniques. Component templates reduce the costs associated with dynamically finding components, and also offer effective and efficient caching during search. Templates also allow us to use symmetries detected during search to make more effective use of previous computations.

## References

[Amir & McIlraith, 2000] Amir, E., and McIlraith, S. 2000. Partition-based logical reasoning. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 389–400.

[Bacchus, Dalmao, & Pitassi, 2003] Bacchus, F.; Dalmao, S.; and Pitassi, T. 2003. Algothims and complexity results for sat and bayesian inference. *FOCS 2003* 340–351.

[Bayardo & Pehoushek, 2000] Bayardo, R. J., and Pehoushek, J. D. 2000. Counting models using connected components. In *Proceedings of the AAAI National Conference (AAAI)*, 157–162.

[Darwiche, 2001] Darwiche, A. 2001. Recursive conditioning. *Artificial Intelligence* 126:5–41.

[Dechter, 2004] Dechter, R. 2004. And/or search spaces for graphical models. Technical report, School of Information and Computer Science, University of California, Irvine.

[Larrosa, Morancho, & Niso, 2005] Larrosa, J.; Morancho, E.; and Niso, D. 2005. On the practical applicability of bucket elimination: Still-life as a case study. *Journal of Artificial Intelligence Research* 23:412–440.

[Marinescu & Dechter, 2005] Marinescu, R., and Dechter, R. 2005. Advances in and/or branch-and-bound search for constraint optimization. In *Workshop on Preferences and Soft Constraints*.

[Marinescu & Dechter, 2006] Marinescu, R., and Dechter, R. 2006. Dynamic orderings for and/or branch-and-bound search in graphical models. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*.

[McKay, 2004] McKay, B. D. 2004. Nauty. available at `http://cs.anu.edu.au/people/bdm/nauty`.

[Moskewicz *et al.*, 2001] Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *Proc. of the Design Automation Conference (DAC)*.

[Park & Gelder, 2000] Park, T. J., and Gelder, A. V. 2000. Partitioning methods for satisfiability testing on large formulas. *Information and Computation* 162:179–184.

[Puget, 2005] Puget, J.-F. 2005. Automatic detection of variable and value symmetries. In *International Conference on Principles and Practice of Constraint Programming*, 475–489.

[Sang *et al.*, 2004] Sang, T.; Bacchus, F.; Beame, P.; Kautz, H.; and Pitassi, T. 2004. Combining component caching and clause learning for effective model counting. In *Theory and Applications of Satisfiability Testing (SAT-2004)*.