A Graph Based Backtracking Algorithm for Solving General CSPs

Wanlin Pang[1] and Scott D. Goodwin[2]

[1] QSS Group Inc., NASA Ames Research Center, Moffett Field, CA 94035
[2] School of Computer Science, University of Windsor
Windsor, Ontario, Canada N9B 3P4

Abstract. Many AI tasks can be formalized as constraint satisfaction problems (CSPs), which involve finding values for variables subject to constraints. While solving a CSP is an NP-complete task in general, tractable classes of CSPs have been identified based on the structure of the underlying constraint graphs. Much effort has been spent on exploiting structural properties of the constraint graph to improve the efficiency of finding a solution. These efforts contributed to development of a class of CSP solving algorithms called decomposition algorithms. The strength of CSP decomposition is that its worst-case complexity depends on the structural properties of the constraint graph and is usually better than the worst-case complexity of search methods. Its practical application is limited, however, since it cannot be applied if the CSP is not decomposable. In this paper, we propose a graph based backtracking algorithm called ω-CDBT, which shares merits and overcomes the weaknesses of both decomposition and search approaches.

1 Introduction

Many AI tasks can be formalized as constraint satisfaction problems (CSPs), which involve finding values for variables subject to constraints. While constraint satisfaction in its general form is known to be NP-complete, many CSPs are tractable and can be solved efficiently. Much work has been done to identify tractable classes of CSPs based on the structure of the underlying constraint graphs and many deep and insightful results have been obtained in this direction [12, 1, 15, 8, 6, 9, 28, 29, 3, 17, 18, 21, 10, 20, 24, 25]. A serious practical limitation of this research, however, has been its focus on backtrack-free conditions. Obviously, a CSP which has backtrack-free solutions is tractable, but a tractable CSP does not necessarily have backtrack-free solutions. In practice, many researchers have tried to improve the efficiency of finding a solution by exploiting the structural properties of the constraint graph. A class of structure-based CSP solving algorithms, called decomposition algorithms, has been developed [14, 16, 4, 7]. Decomposition algorithms attempt to find solutions by decomposing a CSP into several simply connected sub-CSPs based on the underlying constraint graph and then solving them separately. Once a CSP is decomposed into a set of sub-CSPs, all solutions for each sub-CSP are found. Then a new CSP is formed where the original variable set in each sub-CSP is taken as a singleton variable. Usually

the technique aims at decomposing a CSP into sub-CSPs such that the number of variables in the largest sub-CSP is minimal and the newly formed CSP has a tree-structured constraint graph. In this way, the time and space complexity of finding all solutions for each sub-CSP is bounded, and the newly formed CSP has backtrack-free solutions. The complexity of a decomposition algorithm is exponential in the size of the largest sub-CSP. The class of CSPs that can be decomposed into sub-CSPs such that their sizes are bounded by a fixed number k is tractable and can be solved by decomposition in polynomial time. This is the strength of CSP decomposition. A fatal weakness of CSP decomposition, however, is that the decomposition is not applicable to solving a CSP that is not decomposable, that is, its decomposition is itself. A secondary drawback of CSP decomposition is that, even if the CSP is decomposable, finding all solutions for all the sub-CSPs is unnecessary and inefficient.

In this paper, we propose a graph based backtracking algorithm, called ω-CDBT, which shares the strength of CSP decomposition and overcomes its weaknesses. As with CSP decomposition, ω-CDBT decomposes the underlying constraint hypergraph into an acyclic graph. Unlike CSP decomposition, however, ω-CDBT only tries to find one solution for a chosen sub-CSP, which is not separated from other sub-CSPs, and then tries to extend it to other sub-CSPs. The ω-CDBT algorithm uses a constraint representative graph called ω-graph [24, 22, 25]. The complexity of ω-CDBT is exponential in the degree of cyclicity of the ω-graph. Nevertheless, the significant contributions of this research on combining search with constraint structure are: 1) The class of CSPs with the property that the degree of cyclicity of the associated ω-graph is less than a fixed number k is tractable. As shown in [24, 22, 25], given a constraint hypergraph, the degree of cyclicity of an ω-graph is less than or equal that of the constraint hypergraph. Therefore, the class of CSPs that is ω-CDBT solvable in polynomial time includes the class of CSPs that is solvable in polynomial time by other decomposition algorithms such as hinge decomposition [16]. 2) For CSPs that do not have the above mentioned property, ω-CDBT still has a better worst-case complexity bound than other decomposition algorithms such as hinge decomposition [16], which in turn has a better worst-case complexity bound than search algorithms that do not exploit constraint structure. In both cases, ω-CDBT also has advantage over decomposition algorithms in that it finds only one solution for each sub-CSP which saves space and time. 3) In cases where CSPs are not decomposable, decomposition algorithms are not applicable whereas ω-CDBT degenerates to CDBT [23] which is still a practical CSP solving algorithm.

The paper is organized as follows. We first give definitions of constraint satisfaction problems and briefly overview constraint graphs and CSP decomposition. We then present the ω-CDBT algorithm, analyze its complexity, and compare it with decomposition algorithms.

2 Preliminaries

2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem (CSP)* is a structure (X, D, V, S). Here, $X = \{X_1, X_2, \ldots, X_n\}$ is a set of variables that may take on values from a set of domains $D = \{D_1, D_2, \ldots, D_n\}$, and $V = \{V_1, V_2, \ldots, V_m\}$ is a family of ordered subsets of X called *constraint schemes*. Each $V_i = \{X_{i_1}, X_{i_2}, \ldots, X_{i_{r_i}}\}$ is associated with a set of tuples $S_i \subseteq D_{i_1} \times D_{i_2} \times \ldots \times D_{i_{r_i}}$ called *constraint instance*, and $S = \{S_1, S_2, \ldots, S_m\}$ is a family of such constraint instances. Together, an ordered pair (V_i, S_i) is a *constraint* or *relation* which permits the variables in V_i to take only value combinations in S_i.

Let (X, D, V, S) be a CSP, $V_K = \{X_{k_1}, X_{k_2}, \ldots, X_{k_l}\}$ a subset of X. A tuple $(x_{k_1}, x_{k_2}, \ldots, x_{k_l})$ in $D_{k_1} \times D_{k_2} \times \ldots \times D_{k_l}$ is called an *instantiation* of variables in V_K. An instantiation is said to be *consistent* if it satisfies all constraints restricted in V_K. A consistent instantiation of all variables in X is a *solution to the CSP* (X, D, V, S). The task of solving a CSP is to find one or more solutions.

A constraint (V_h, S_h) in a CSP (X, D, V, S) is *minimal* if every tuple in S_h can be extended to a solution. A CSP (X, D, V, S) is *minimal* if every constraint is minimal.

A *binary CSP* is a CSP with unary and binary constraints only, that is, every constraint scheme contains at most two variables. A CSP with constraints not limited to unary and binary is referred to as a *general CSP*.

We will also use some relational operators, specifically, *join* and *projection*. Let $C_i = (V_i, S_i)$ and $C_j = (V_j, S_j)$ be two constraints. The *join* of C_i and C_j is a constraint denoted by $C_i \bowtie C_j$. The *projection* of $C_i = (V_i, S_i)$ on $V_h \subseteq V_i$ is a constraint denoted by $\Pi_{V_h}(C_i)$. The *projection* of t_i on V_h, denoted by $t_i[V_h]$, is a tuple consisting of only the components of t_i that correspond to variables in V_h.

2.2 Graph Theory Background

In this section, we review some graph theoretic terms we will need later and we define constraint representative graphs, namely, the line graph, the join graph, and the ω-graph.

A *graph* G is a structure (V, E), where V is a set of *nodes* and E is a set of *edges*, with each edge joining one node to another.

A *subgraph of* G *induced by* $V' \subset V$ is a graph (V', E') where $E' \subset E$ contains all edges that have both their endpoints in V'. A *partial graph of* G *induced by* $E' \subset E$ is a graph (V, E').

A *path* or a *chain* is a sequence of edges $E_1, E_2, \ldots, E_q$ such that each E_i shares one of its endpoints with E_{i-1} and the other with E_{i+1}. A *cycle* is a chain such that no edge appears twice in the sequence, and the two endpoints of the chain are the same node. A graph is *connected* if it contains a chain for each pair

of nodes. A *connected component* of a graph is a connected subgraph. A graph is *acyclic* if it contains no cycle. A connected acyclic graph is a *tree*.

Let $G = (V, E)$ be a connected graph. A node V_i is called a *cut node* (or *articulation node*) if the subgraph induced by $V - \{V_i\}$ is not connected. A *block* (or *nonseparable component*) of a graph is a connected component that contains no cut nodes of its own. An $O(|E|)$ algorithm exits for finding all the blocks and cut nodes [11].

Let $G = (V, E)$ be a connected graph. The *degree of cyclicity of* G is defined as the number of nodes in its largest block. A graph is *k-cyclic* if its degree of cyclicity is at most k.

A *hypergraph* is a graph with *hyper edges*; that is, an edge in a hypergraph may contain more than two nodes. The graph notations reviewed above can be extended to hypergraph, such as *sub-hypergraph*, *partial hypergraph*, *path*, *connected component*, *block*, and so on. These definitions can be found in [2].

A graph $G = (V, E)$ can be decomposed into a tree of blocks $T_B = (V_B, E_B)$: 1) choose a block $V_{B_i} \in V_B$, which contains at least one non-cut node, as the root node of T_B and mark it; 2) for each unmarked block X_{B_j} that has a node in common with block X_{B_i}, connect X_{B_j} as a child node of X_{B_i} with an edge (X_{B_i}, X_{B_j}) and mark it; 3) take each child node of X_{B_i} as the root node of a subtree, repeat 2) and 3); 4) stop when every block is marked.

For example, give a graph $G = (V, E)$ as shown in Figure 1 (A), we can have a block tree as in Figure 1 (B), where $B_1 = \{V_1, V_2, V_3, V_4\}$, $B_2 = \{V_2, V_5, V_6\}$, $B_3 = \{V_5, V_7, V_8\}$, $B_4 = \{V_6, V_9, V_{10}\}$, $B_5 = \{V_3, V_{11}, V_{12}\}$, $B_6 = \{V_3, V_{13}, V_{14}\}$, $B_7 = \{V_4, V_{15}, V_{16}\}$. The cut nodes in this graph are V_2, V_3, V_4, V_5, and V_6

A block tree determines an order on the block set. For example, block set $B = \{B_1, B_2, B_3, B_4, B_5, B_6, B_7\}$ is in the depth-first order. For each block B_k $(2 \leq k)$ there is a cut node V_{a_k} of the graph that separates this block from its parent block, and there is a node V_{a_1} in B_1 which is not in any other blocks. These nodes are called *separating nodes*. For example, the separating nodes of the graph in Figure 1 (A) are V_1, V_2, V_3, V_4, V_5, and V_6

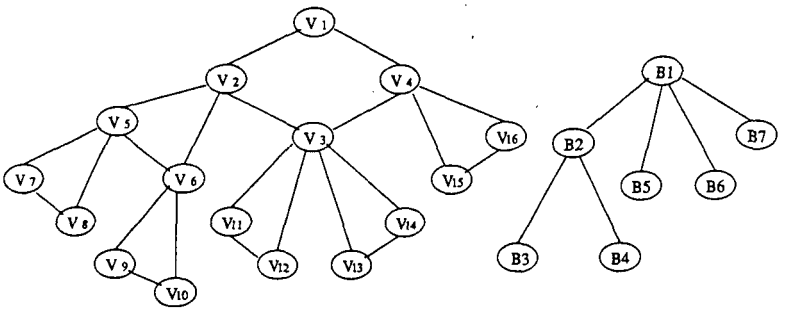


Fig. 1. A graph and its block tree

A binary CSP is associated with a simple constraint graph, which has been well studied and widely used for analyzing and solving binary CSPs [13, 7, 5]. A general CSP is associated with a constraint hypergraph, but the topological properties of the hypergraph have not been well studied in the area of constraint satisfaction problems. Instead, constraint representative graphs such as the *line graph*, the *join graph*, and the *ω-graph* have been studied and used to analyzing and solving general CSPs [20, 19, 16, 24, 25, 26, 27].

Given a CSP (X, D, V, S) and its hypergraph $H = (X, V)$, the *line-graph* [3] is a simple graph $l(H) = (V, L)$ in which nodes in V are hyperedges of the hypergraph and with two nodes joined with an edge in L if these two nodes share common variables. A *join graph* $j(H) = (V, J)$ is a partial linegraph in which some *redundant* edges are removed. An edge in a linegraph is redundant if the variables shared by its two end nodes are also shared by every nodes along an alternative path between the two end nodes. An ω-graph $\omega(H) = (W, F)$ is another constraint representative graph. The node set W of an ω-graph is a subset of nodes V in the line graph such that any node in $V - W$ is *covered* by two nodes in W; that is, if $V_k \in V - W$, then there exist V_i an V_j in V, such that $V_k \subset V_i \cup Vj$. There is an edge joining two nodes if either the two nodes share common variables or they cover a node that is not in W.

For example, given a hypergraph $H = (X, V)$ as in Figure 2 (A) with node set $X = \{X_1, X_2, X_3, X_4, X_5, X_6, X_7\}$ and edge set $V = \{V_1, V_2, V_3, V_4, V_5, V_6\}$, where $V_1 = \{X_1, X_2\}$, $V_2 = \{X_1, X_4, X_7\}$, $V_3 = \{V_2, V_3\}$, $V_4 = \{X_2, X_4, X_7\}$, $V_5 = \{X_3, X_5, X_7\}$, $V_6 = \{X_3, X_6\}$. Its line graph $l(H) = (V, L)$ is in Figure 2 (B). There is an edge, for example, between V_1 and V_2 because these two nodes share a common variable X_1. Edge (V_5, V_6) is redundant because the variable X_3 shared by V_5 and V_6 is also shared by every nodes on an alternative path between V_5 and V_6, that is, path (V_5, V_3, V_6). A join graph resulting from removing redundant edges is in Figure 2 (C), and an ω-graph is in (D) in which there is only 4 nodes, since node V_1 is covered by V_2 and V_4, and node V_3 by V_5 and V_6.

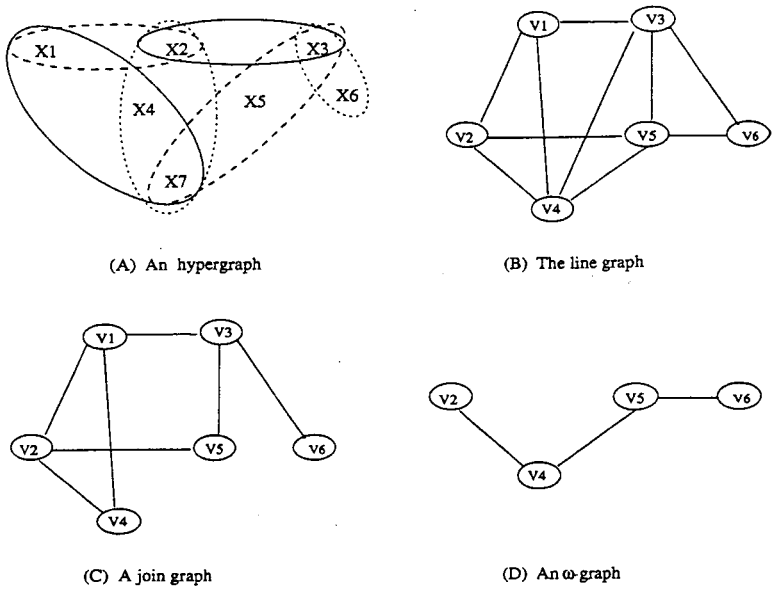
Since constraint representative graphs are simple graphs, all of those graph concepts mentioned previously are applicable. For example, an ω-graph (or a join graph) is *k-cyclic* if the number of nodes in its largest block is at most k. An ω-graph can be decomposed into a block tree.

Notice that the line graph or a join graph is also an ω-graph, but in general, an ω-graph is simpler than the line or join graph in terms of the number of nodes, the degree of cyclicity and the width. In particular, [22] gives an $O(|V|^3)$ algorithm for constructing an ω-graph for a hypergraph with the following property:

Proposition 1. *Given a hypergraph $H = (X, V)$, there exists an ω-graph whose degree of cyclicity is less than or equal the degree of cyclicity of any join graph.*

Note that the *degree of cyclicity of a hypergraph* is defined in [16] as the degree of cyclicity of its minimal join graph. The above proposition indicates that a hypergraph has an ω-graph whose degree of cyclicity is less than or equal that of the hypergraph.

[3] A line graph is also called an inter graph in [19] and a dual-graph in [7].



(A) An hypergraph (B) The line graph

(C) A join graph (D) An ω-graph

Fig. 2. A hypergraph and its representative graphs

2.3 CSP Decomposition

Decomposition algorithms attempt to find solutions more efficiently by decomposing a CSP into a set of sub-CSPs such that these sub-CSPs form a tree and the size of the largest sub-CSP is minimized. In general, a decomposition algorithm works as follows:

1. decompose the constraint hypergraph into a tree;
2. find all solutions to each sub-CSP associated with each node in the tree;
3. form a new CSP where the original variable set in each tree node is taken as a singleton variable;
4. find one solution to the new CSP.

Many decomposition algorithms have been developed [14, 16, 7, 4] and they usually differ in the first step. A comparison of most notable decomposition algorithms can be found in [14]. As pointed out in [14], each decomposition method defines a parameter as a measure of cyclicity of the underlying constraint hypergraph such that, for a fixed number k, all CSPs with the parameter bounded by k are solvable in polynomial time.

ω-graph fits well into this decomposition scheme in that we first construct an ω-graph from a given constraint hypergraph and then decompose the ω-graph into a tree. It is obvious that many graph decomposition methods can be used to decompose an ω-graph. For simplicity, however, we choose the block tree method to decompose ω-graphs in this paper.

The problem with the decomposition methods is that they cannot be applied if a given CSP does not possess some required properties (for example,

non-decomposible). Moreover, even if the underlying constraint graph is decomposable, finding all solutions for every sub-CSP is inefficient and unnecessary. In the following, we propose a graph based backtracking algorithm called ω-CDBT that overcomes these weaknesses.

3 ω-Graph Based Backtracking

Let (X, D, V, S) be a CSP and $C = \{C_i = (V_i, S_i) | V_i \in V, S_i \in S\}$ a set of constraints. Let $\omega(H) = (W, F)$ be an ω-graph and $B = \{B_1, B_2, \ldots, B_l\}$ a set of blocks of $\omega(H)$ which is ordered in the depth-first manner according to the block tree, and each block $B_k = \{V_{k_1}, V_{k_2}, \ldots, V_{k_{|B_k|}}\}$ a set of nodes in which the first one is the separating node. Let $cks_a(V_i, V_j)$ denote the set of constraints on $V - W$ covered by V_i and V_j in W, that is, $cks_a(V_i, V_j) = \{V_k \in V - W | V_k \subset V_i \cup V_j\}$. The idea of the ω-CDBT algorithm is to search for a consistent assignment to variables involved in a block and then extend it to the child blocks. If at a block where no consistent assignment can be found, ω-CDBT backtracks to the parent block, reinstantiates variables in that block, and starts from there. Within a block, ω-CDBT uses a CDBT-like strategy [23] to find a consistent assignment to the variable subset in the block, which may involve backtracking within the block. The algorithm stops when a solution is found or when it proves that no solution exists.

3.1 Algorithm

The ω-CDBT performs backtrack at two nested levels which we call *outer-BT* and *inner-BT*. The inner-BT finds a consistent instantiation of variables involved in a block $B_k = \{V_{k_1}, V_{k_2}, \ldots, V_{k_{|B_k|}}\}$. The outer-BT calls inner-BT to find consistent instantiations for all blocks in the depth-first order. If a consistent instantiation of variables in the current block is found then outer-BT calls inner-BT again to find consistent instantiations of variables in its child blocks; otherwise, the outer-BT moves backward to the parent block and calls the inner-BT to find another consistent instantiation of variables in that block.

The function *DFS* corresponding to the outer-BT, the inner-BT, which is based on *CDBT* [23], consisting of two recursive functions *forward* and *goback*, and an auxiliary function *test* are given below. In these functions, some notations are explained as follows: tup_{B_k} is a consistent instantiation of variables in B_k; sol_{T_k} is a consistent instantiation of variables in the subtree rooted at B_k; $child_{B_k}$ is a set of child blocks of B_k; $changed_{B_k}$ is a flag indicating if the instantiation of variables in the separating node of B_k has changed; idx_{B_k} is the index of the separating node of B_k in the parent block; t_{B_k} is a current instantiation of variables in the separating node of B_k, initialized as a nil-tuple.

ω-*DFS*(B_k, V_I, tup_I)

1. **begin**
2. $tup_{B_k} \leftarrow \omega$-*forward*$(B_k, V_I, tup_I)$;

3. if $tup_{B_k} = \emptyset$ then return $\emptyset$;
4. for each $B_j \in child_{B_k}$ do
5. if $changed_{B_j}$ then
6. $sol_{T_j} \leftarrow \omega\text{-}DFS(B_j, V_{j_1}, t_{B_j})$;
7. if $sol_{T_j} = unsatisfiable$ then return *unsatisfiable*;
8. if $sol_{T_j} = \emptyset$ then
9. delete t_{B_j} from S_{j_1} and $S^*_{j_1}$;
10. if $S_{j_1} = \emptyset$ then return *unsatisfiable*;
11. $V' \leftarrow \cup_{i=1}^{idx_{B_j}-1} V_{k_i}$; $tup' \leftarrow tup_{B_k}[V']$;
12. return $(\omega\text{-}DFS(B_k, V', tup'))$;
13. $changed_{B_j} \leftarrow false$;
14. end for
15. return $tup_{B_k} \bowtie (\bowtie_{B_j \in child_{B_k}} sol_{T_j})$;
16. end

forward(B_k, V_I, tup_I)

1. begin
2. if $V_I = \cup B_k$ then return tup_I;
3. $cks(V_{I+1}) \leftarrow \cup_{j=1}^{i} cks_a(V_{k_j}, V_{k_{i+1}})$;
4. $S^*_{k_{i+1}} \leftarrow \{tup | tup \in S_{k_{i+1}}, tup[V_I \cap V_{k_{i+1}}] = tup_I[V_I \cap V_{k_{i+1}}]\}$;
5. while $S^*_{k_{i+1}} \neq \emptyset$ do
6. $tup \leftarrow$ one tuple taken from $S^*_{k_{i+1}}$;
7. $tup_{I+1} \leftarrow tup_I \bowtie tup$;
8. if $test(tup_{I+1}, cks(V_{I+1}))$ then
9. for each $B_j \in child_{B_k}$
10. if $V_{k_{i+1}} \in B_j$ and $tup \neq t_{B_j}$ then $t_{B_j} \leftarrow tup$; $changed_{B_j} \leftarrow true$;
11. return *forward*$(B_k, V_{I+1}, tup_{I+1})$;
12. end while
13. return *goback*(B_k, V_I, tup_I);
14. end

goback(B_k, V_I, tup_I)

1. begin
2. if $V_I = V_{k_1}$ then return $\emptyset$;
3. while $S^*_i \neq \emptyset$ do
4. $tup \leftarrow$ one tuple taken from S^*_i;
5. $tup_I \leftarrow tup_{I-1} \bowtie tup$;
6. if $test(tup_I, cks(V_I))$ then
7. for each $B_j \in child_{B_k}$
8. if $V_{k_{i+1}} \in B_j$ and $tup \neq t_{B_j}$ then $t_{B_j} \leftarrow tup$; $changed_{B_j} \leftarrow true$;
9. return *forward*(B_k, V_I, tup_I);

10. **end while**
11. **return** $goback(B_k, V_{I-1}, tup_{I-1})$;
12. **end**

$test(tup_I, cks))$
1. **begin**
2. **for** each $C_h = (V_h, S_h)$ in cks **do**
3. **if** $tup_I[V_h] \notin S_h$ **then return** *false*;
4. **return** *true*;
5. **end**

For a current block $B_k = \{V_{k_1}, V_{k_2}, \ldots, V_{k_{|B_k|}}\}$ and a partial instantiation tup_I of variables in $V_I = \cup_{j=1}^{i} V_{k_j}$, *DFS* tries to extend tup_I to a consistent instantiation sol_{T_k} of variables involved in the subtree rooted at B_k. Firstly, it tries to extend tup_I to a consistent instantiation tup_{B_k} of variables in V_{B_k}. This can be done by function *forward*. If *forward* succeeds, that is, if a consistent instantiation tup_{B_k} is found, then *DFS* moves forward to those of its child blocks that have not been instantiated at all or their instantiations have been changed due to backtracking to the parent block. *DFS* calls itself recursively for each of the subtrees, and then returns the joined tuple $tup_{B_k} \bowtie (\bowtie_{B_j \in child_{B_k}} sol_{T_j})$. If *forward* fails, that is, if it does not find a consistent instantiation tup_{B_k} of variables in V_{B_k} such that $tup_{B_k}[V_{k_1}] = tup_{k_1}$, then *DFS* reports that the tuple tup_{k_1} has no consistent extension to variables in V_{B_k}. Before the algorithm backtracks to the parent block, the tuple tup_{k_1} is deleted from S_{k_1}, since it will not be in any solution which will be explained in Section 3.3 and S_{k_1} is checked if it is empty. If it is empty then there is no solution to the problem, so *DFS* stops and reports *unsatisfiable*. If S_{k_1} is not empty then *DFS* moves up to the parent block and starts from there.

Within block B_k, suppose that we have already found a consistent instantiation tup_I of variables in $V_{k_1}, V_{k_2}, \ldots, V_{k_i}$ (their union is denoted by V_I), function *forward* extends this instantiation by appending to it an instantiation of variables in $V_{k_{i+1}}$ which is a node in the ω-graph. *forward* chooses a tuple tup from $S^*_{k_{i+1}}$ as an instantiation of variables in $V_{k_{i+1}}$ and joins tup and tup_I to form a new tuple tup_{I+1}, which is tested to see if it is consistent. Notice that the subset $S^*_{k_{i+1}}$ contains those tuples in $S_{k_{i+1}}$ that are compatible with tup_I. If tup_{I+1} is consistent, then *forward* is called recursively to extend tup_{I+1}; otherwise, another tuple from $S^*_{k_{i+1}}$ is tried. If no tuples are left in $S^*_{k_{i+1}}$, *goback* is called to re-instantiate variables in variable set V_{k_i}. Function *goback* tries to re-instantiate variables in V_{k_i} and to form another consistent instantiation of variables in $V_I = \cup_{j=1}^{i} V_{k_j}$. It first chooses another tuple from $S^*_{k_i}$ and forms a new tuple tup_I which is tested to see if it is consistent. If tup_I is consistent, then *forward* is called to extend tup_I; otherwise, another tuple from $S^*_{k_i}$ is tried. If $S^*_{k_i}$ is empty, then *goback* is called recursively to re-instantiate variables in variable set $V_{k_{i-1}}$. Note that *goback* does not re-instantiate variables in the separating node V_{k_1}. The tuple t_{B_k} for variables in V_{k_1} was chosen when the parent block

was dealt with. If t_{B_k} cannot be extended to variables in B_k, *goback* returns $\emptyset$ and passes the control to *DFS* which deletes t_{B_k} from S_{k_1}. Backtracking across blocks occurs.

Function $test(tup_K, cks)$ returns *true* if tuple tup_K satisfies all the constraints in *cks*, and *false* otherwise.

To find a solution to a given CSP $\mathbb{P} = (X, D, V, S)$, we need a main program such as the one given below to call *DFS* repeatedly until a solution is found or unsatisfiability is verified.

ω-*CDBT*($\mathbb{P}$, *sol*)

1. **begin**
2. **for** each $tup \in S_{1_1}$ **do**
3. $sol \leftarrow DFS(B_{k_1}, V_{k_1}, tup)$;
4. **if** sol = *unsatisfiable* **then return** *unsatisfiable*;
5. **if** $sol \neq \emptyset$ **then return** *sol*;
6. **end for**
7. **return** *unsatisfiable*;
8. **end**

Algorithm *DFS* instantiates the variables in block B_j only if the values assigned to the variables in the separating node of B_j have been changed; that is, only if $changed_{B_j}$ is true (line 5 in the algorithm). At the first time of visiting B_j, $changed_{B_j}$ is true since the variables in the separating nodes have been instantiated when the algorithm visits B_k, the parent block of B_j. However, when the algorithm goes back to re-instantiates the variables in B_k, the variables in the separating node may not be affected, in which case the assignment to the variables in B_j will be retained. Needless to say, this saves time of repeatedly finding values for the variables in the subtree rooted at B_j. However, an immediate question to ask is whether this causes the algorithm to miss any solutions. The answer is *no* because even if the algorithm does re-instantiate variables in B_j, the instantiation will be the same if the values assigned to the variables in the separating node of B_j have not been changed.

3.2 Example

We consider a CSP $\mathbb{P} = (X, D, V, S)$ which has an ω-graph in Figure 1 (A) and we use this example to illustrate how the ω-graph based backtracking works.

We are given an ordered block set $B = \{B_1, B_2, B_3, B_4, B_5, B_6, B_7\}$, where each block is an ordered set of constraint schemes, that is: $B_1 = \{V_1, V_2, V_3, V_4\}$, $B_2 = \{V_2, V_5, V_6\}$, $B_3 = \{V_5, V_7, V_8\}$, $B_4 = \{V_6, V_9, V_{10}\}$, $B_5 = \{V_3, V_{11}, V_{12}\}$, $B_6 = \{V_3, V_{13}, V_{14}\}$, $B_7 = \{V_4, V_{15}, V_{16}\}$. Let V_{B_i} be the subset of variables involved in B_i.

We start from ω-*CDBT*($\mathbb{P}$, *sol*), choose a tuple $t_{B_1} \in S_1$ and call $DFS(B_1, V_1, t_{B_1})$. *DFS* first calls $forward(B_1, V_1, t_{B_1})$ to extend t_{B_1} to variables in V_{B_1}. If it fails, then it will choose another tuple from S_1 and start again. Suppose that it succeeds and it returns a tuple tup_{B_1} as a consistent instantiation of variables in

V_{B_1}, then *DFS* will be called recursively for each child block B_2, B_5, B_6 and B_7. Recall that t_{B_i} is the instantiation of variables in the separating node. For the first child block B_2, $DFS(B_2, V_2, t_{B_2})$ is called to extend t_{B_2} to variables involved in the subtree rooted at B_2, which include the variables in V_5, V_6, V_7, V_8, V_9 and V_{10}. At first, $forward(B_2, V_2, t_{B_2})$ is called to extend t_{B_2} to variables in V_5 and V_6. Suppose that it succeeds and it returns a tuple tup_{B_2} as a consistent instantiation of variables in V_{B_2}, then *DFS* will be called for the child blocks of B_2. Again, suppose that they all succeeds, that is, tuples sol_{T_3} and sol_{T_4} are returned. So, $DFS(B_2, V_2, tup_{V_2})$ returns a tuple $sol_{T_2} = sol_{B_2} \bowtie sol_{T_2} \bowtie sol_{T_4}$ as a consistent instantiation of variables involved in the subtree rooted at B_2. For the second child block B_5, $DFS(B_5, V_3, t_{B_5})$ is called to extend t_{B_5} to variables involved in the subtree rooted at B_5. It calls $forward(B_5, V_3, t_{B_5})$ to extend t_{B_5} to variables in V_{B_5}. If it succeeds, then $DFS(B_5, V_3, t_{B_5})$ will return a tuple sol_{T_5}. However, suppose that $forward(B_5, V_3, t_{B_5})$ fails, which means that t_{B_5} cannot be extended to a consistent instantiation of variables in B_5, then $DFS(B_5, V_3, t_{B_5})$ returns sol_{T_5} which is a nil-tuple. Since sol_{T_5} is empty, tuple t_{B_5} is deleted from S_3, and S_3 is checked to determine if it is empty now. If it is, then there is no solution to the problem, $DFS(B_1, V_1, t_{B_1})$ will return *unsatisfiable* and $CDBT(I\!P, sol)$ will return *unsatisfiable*. We suppose that S_3 is not empty. Then $DFS(B_1, V', tup')$ is called, where $V' = V_1 \cup V_2$ and $tup' = tup_{V_1} \bowtie tup_{V_2}$. This time, $forward(B_1, V', tup')$ is called to extend tup' to a consistent instantiation of variables in V_{B_1}. If it finds one without re-instantiating variables in V_2 and V_1, then the instantiation of variables involved in the subtree rooted at B_2 is retained, and *DFS* is called for child nodes B_5, B_6, B_7. If variables in V_2 are re-instantiated, then the variables involved in the subtree rooted at B_2 may have to be re-instantiated. However, whether or not variables in V_{B_3} and V_{B_5} need to be re-instantiated depends on whether or not V_5 and V_6 are re-instantiated. If $forward(B_1, V', tup')$ goes back to V_1, then $DFS(B_1, V', tup')$ returns empty tuple, we will choose another tuple from S_1 and start from there.

3.3 Analysis

For analysis, we define *minimal constraint* and give a few technical lemmas, which have been proven in [22].

Let $I\!P = (X, D, V, S)$ be a CSP and $C = \{C_i = (V_i, S_i) | V_i \in V, S_i \in S\}$ a set of constraints. Let $H = (X, V)$ be the associated hypergraph, $\omega(H) = (W, F)$ an ω-graph, $B = \{B_1, B_2, \ldots, B_l\}$ a set of blocks ordered in the depth-first manner, and $A = \{V_{a_1}, V_{a_2}, \ldots, V_{a_l}\}$ a set of separating nodes. Let V_{B_i} denote the subset of variables involved in block B_i.

Definition 2. Let (X, D, V, S) be a CSP, let V' be a subset of V and C' a subset of C restricted on V'. A *sub-CSP induced by* V' is a CSP (X', D', V', S') where $X' = \bigcup V'$, D' is a subset of the domains of variables in X', and S' is a set of constraint instances corresponding to V'. A constraint $C_i \in C'$ is said to be *minimal relative to* (X', D', V', S') if every tuple in S_i can be extended to a consistent instantiation of variables in X'. A constraint $C_i \in C$ is said to be

minimal if it is minimal relative to (X, D, V, S). A CSP is said to be *minimal* if every constraint is minimal.

Let $C_A = \{(V_{a_i}, S_{a_i}) | V_{a_i} \in A\}$ be a subset of constraints on A.

Lemma 3. *If every constraint in C_A is minimal, then every consistent instantiation of variables involved in each block can be extended to a solution.*

This lemma suggests that if every constraint on those articulation nodes is minimal, then the relation represented by the sub-CSP corresponding to each block is minimal. The following lemma indicates that minimizing the constraints on articulation nodes can be done by minimizing them relative to each block.

Lemma 4. *If every constraint (V_{a_i}, S_{a_i}) in C_A is minimal relative to the sub-CSP induced by the block to which V_{a_i} belongs, then they are also minimal.*

Lemma 5. *Let B_i be a block and V_{a_i} the separating node. If a tuple in S_{a_i} has no consistent extension to variables in V_{B_i}, then it cannot be extended to a solution.*

Based on this lemma, if a tuple in a constraint corresponding to an articulation node has no consistent extension to the variables involved in the block to which the articulation node belongs, it can be safely deleted. If every tuple in such a constraint is so, then there is no solution to the problem.

Lemma 6. *Let B_i be the parent block of B_j, and V_{a_j} the separating node in B_j. Let tup_{B_i} and tup_{B_j} be consistent instantiations of variable in V_{B_i} and V_{B_j} respectively. If $tup_{B_i}[V_{a_j}] = tup_{B_j}[V_{a_j}]$, then $tup_{B_i} \bowtie tup_{B_j}$ is a consistent instantiation of variables $V_{B_i} \cup V_{B_j}$.*

This lemma suggests that if we have a consistent instantiation tup_{B_i} of variables in a parent block B_i, extending tup_{B_i} to the variables in the child block B_j can be done by extending $tup_{B_i}[V_{a_j}]$ to the variables in B_j, so the consistent checking is restricted within the child block.

Theorem 7. *The ω-CDBT is correct.*

Proof. We prove that ω-CDBT is sound, complete, and it terminates. The CDBT algorithm has been proven to be correct in [23], the inner-BT consisting of *forward* and *goback* is correct with respect to each block.

Based on Lemma 6, when a consistent instantiation of variables in the parent blocks is extended to a child block, the new instantiation to variables including variables in the child block is consistent. In particular, when a consistent instantiation is successfully extended to variables in the last block, we have a whole assignment which is a solution. This proves the soundness.

The completeness follows from Lemma 5 and the fact that the inner-BT is complete.

The search space of *ω-CDBT* can be seen as a $|W|$-level tree, in which each level corresponds to a $V_i \in W$, and *ω-CDBT* visits every node in the search

space at most once, it terminates. □

Suppose that the ω-graph is k-cyclic and has l blocks. Let $|s|$ be the size of the maximal constraint relations.

Lemma 8. *If every constraint in C_A is minimal, then any backtracking performed in* ω-CDBT *is restricted within each block, and the complexity of using* ω-CDBT *to solve a CSP with minimum constraints in C_A is $O(l|s|^k)$.*

Proof. Suppose that the algorithm has found a consistent instantiation of variables in V_{B_i} and it moves forward to a child block B_j. Finding a consistent instantiation of variables in V_{B_j} may require backtracking but it will not backtrack to the parent block B_i, since, according to Lemma 3, the existing consistent instantiation of variables in V_{B_i} can be extended to a solution.

The time complexity of finding a consistent instantiation of variables involved in a block is $O(|s|^k)$, and there are l blocks, so the time complexity of ω-*CDBT* is $O(l|s|^k)$. □

Theorem 9. *The time complexity of* ω-CDBT *is $O(l|s|^k)$.*

Proof. Based on the Lemma 4, minimizing constraints in C_A can be done by minimizing them relative to each block. The complexity of minimizing a constraint relative to a block is $O(|s|^k)$, so the complexity of minimizing constraints in C_A is $O(l|s|^k)$. In the worst case (that is equivalent to using ω-*CDBT* to find all solutions), every constraint in C_A will be minimized, which takes $O(l|s|^k)$ time, and then finding solutions with minimized constraint in C_A takes another $O(l|s|^k)$ time. Together, the time complexity of ω-*CDBT* is $O(l|s|^k)$. □

Since the complexity of solving the CSP by using ω-CDBT is exponential in k, a class of CSP where k is less than a fixed number is ω-CDBT solvable in polynomial time. Following directly from Proposition 1, this class of tractable CSPs includes the class of CSPs solvable by the hinge decomposition [16].

4 Comparison with Decomposition and Other Search Methods

A general decomposition scheme is given in Section 2.3 and an ω-graph based decomposition algorithm can be easily constructed. To compare ω-CDBT with decomposition algorithms including ω-graph based decomposition, we argue that ω-CDBT shares the virtue of tree search algorithms in that it finds only one consistent assignment to variables in each block which corresponding to a sub-CSP in the decomposition scheme. Finding one solution is more cost-effective than finding all solutions.

Furthermore, the ω-CDBT algorithm has an additional two advantages over other search methods: 1) when a tuple t_{B_i} cannot be extended to a consistent instantiation of variables in V_{B_i}, it is deleted from the constraint on the separating

node of B_i; then all the sub-regions of search space rooted at nodes containing t_{B_i} will be ruled out to avoid further exploration; 2) when the algorithm backtracks from a child block to the parent, the instantiation of the variables in the sibling blocks preceding this block may be retained, so that this sub-region of the search space does not need to be searched repetitively.

Another advantage of ω-CDBT is its ability to overcome the failure of decomposition methods when a given CSP is not decomposable. In this case, the decomposition method degenerates into whatever method is used to find all solutions which is expensive. ω-CDBT, on the other hand, degenerates to the original CDBT algorithm which is still a practical CSP solving algorithm.

5 Conclusion

Constraint satisfaction in its general form is known to be NP-complete, yet many CSPs are tractable and can be solved efficiently. Every CSP has an associated constraint graph. The key idea is that the efficiency of finding a solution can be improved by exploiting structural properties of the constraint graph. The contributions of this paper are both theoretical and practical. First, we have identified a new tractable class of CSPs that contains previously identified tractable classes. This extends the known set of CSPs that are solvable in polynomial time. Second, we have provided an algorithm that solves CSPs in this class in polynomial time, whereas other known algorithms cannot guarantee polynomial time solutions for this class. Third, even outside of this class, the provided algorithm has a better worst case complexity. This extends the limits of what is solvable in practice. Future empirical study is required to evaulate the actual improvement of the ω-CDBT algorithm against other decompostion and search algorithms.

References

1. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):497–513, 1983.
2. C. Berge. *Graphs and Hypergraphs.* North-Holland, New York, 1973.
3. M. Cooper, D. A. Cohen, and P. G. Jeavons. Characterizing tractable constraints. *Artificial Intelligence*, 65:347–361, 1994.
4. R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
5. R. Dechter. Constraint networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 276–285. Wiley-Interscience, 2nd edition, 1992.
6. R. Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–102, 1992.
7. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.
8. R. Dechter and J. Pearl. Directed constraint networks: A relational framework for causal modeling. In *Proceedings of IJCAI-91*, pages 1164–1170, Sydney, Australia, 1991.

9. R. Dechter and P. van Beek. Local and global relational consistency. In *Proceedings of the 1st International Conference on Principles and Practices of Constraint Programming*, pages 240–257, Cassis, France, September 1995.
10. Y. Deville and P. Van Hentenryck. An efficient arc consistency algorithm for a class of CSPs. In *Proceedings of IJCAI-91*, pages 325–330, Sydney, Australia, 1991.
11. S. Even. *Graph Algorithms.* Computer Science Press, Potomac, Maryland, 1979.
12. E. Freuder. A sufficient condition for backtrack-free search. *J. of the ACM*, 29(1):25–32, 1982.
13. E. Freuder. Backtrack-free and backtrack-bounded search. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 343–369. Springer-Verlag, New York, 1988.
14. G. Gottlob. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124:243–282, 2000.
15. M. Gyssens. On the complexity of join dependencies. *ACM Transactions on Database Systems*, 11(1):81–108, 1986.
16. M. Gyssens, P. Jeavons, and D. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–89, 1994.
17. P. Jeavons. Tractable constraints on ordered domains. *Artificial Intelligence*, 79:327–339, 1995.
18. P. Jeavons, D. Cohen, and M. Gyssens. A test for tractability. In *Lecture Notes in Computer Science*, volume 1118, pages 267–281, Cambridge, MA, 1996. CP'96.
19. P. Jegou. On some partial line graphs of a hypergraph and the associated matroid. *Discrete Mathematics*, 111:333–344, 1993.
20. P. Jegou. On the consistency of general constraint satisfaction problems. In *Proceedings of AAAI-93*, pages 114–119, 1993.
21. L. M. Kirousis. Fast parallel constraint satisfaction. *Artificial Intelligence*, 64:174–160, 1993.
22. W. Pang. *Constraint Structure in Constraint Satisfaction Problems.* PhD thesis, University of Regina, Canada, 1998.
23. W. Pang and S. D. Goodwin. Constraint-directed backtracking. In *The 10th Australian Joint Conference on AI*, pages 47–56, Perth, Western Australia, December 1997.
24. W. Pang and S. D. Goodwin. A revised sufficient condition for backtrack-free search. In *Proceedings of 10th Florida AI Research Symposium*, pages 52–56, Daytona Beach, FL, May 1997.
25. W. Pang and S. D. Goodwin. Characterizing tractable CSPs. In *The 12th Canadian Conference on AI*, pages 259–272, Vancouver, BC, Canada, June 1998.
26. W. Pang and S. D. Goodwin. Consistency in general CSPs. In *The 6th Pacific Rim International Conference on AI*, pages 469–479, Melbourne, Australia, August 2000.
27. W. Pang and S. D. Goodwin. Binary representation for general CSPs. In *Proceedings of 14th Florida AI Research Symposium (FLAIRS-2001)*, Key West, FL, May 2001.
28. P. van Beek. On the minimality and decomposability of constraint networks. In *Proceedings of AAAI-92*, pages 447–452, 1992.
29. P. van Beek and R. Dechter. On the minimality and global consistency of row-convex constraint networks. *Journal of the ACM*, 42:543–561, 1995.