Constraint-Directed Backtracking Algorithm for Constraint-Satisfaction Problems

Wanlin Pang
Scott D. Goodwin

Technical Report CS-96-05

May 1996

Department of Computer Science
University of Regina
Regina, Saskatchewan
S4S 0A2

ISSN 0828-3494
ISBN 0-7731-0322-8

Constraint-Directed Backtracking Algorithm for Constraint-Satisfaction Problems

Wanlin Pang Scott D. Goodwin
Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada S4S 0A2
Email: pang@cs.uregina.ca goodwin@cs.uregina.ca

Abstract

We propose a new backtracking method called *constraint-directed backtracking (CDBT)* for solving constraint-satisfaction problems (CSPs). *CDBT* and chronological backtracking (BT) share a similar style of instantiating variables (forward) and re-instantiating variables (backward). They differ in that *CDBT* searches instantiations of variables in a variable set from a given constraint posed on that variable set and appends it to a partial solution, whereas BT searches the instantiation of one variable from its domain. The search space of *CDBT* is much more limited than that of chronological backtracking. The similarity between *CDBT* and *BT* enables us to incorporate other tree search techniques, such as *BJ, CBJ, FC*, into *CDBT* to improve its performance further.

1 Introduction

Backtracking search is one of the most popular methods for solving constraint satisfaction problems ([4, 15, 17]). The original backtracking *BT* [9, 2] (often referred to as chronological or generic backtracking) suffers from the *thrashing* problem, as well as explosive search space. Many new search techniques have been developed to deal with these problems. For instance, backjumping (BJ) [8] and conflict-directed backjumping (CBJ) [16] jump to the culprit point when a dead-end is encountered. Forward checking (FC) [11] and backmarking (BM) [7] detect dead-ends before they occur. There are also hybrid ones [16] that improve BT in both ways. In this paper, we propose a new algorithm called *constraint-directed backtracking (CDBT)* for solving CSPs. *CDBT* and *BT* share a similar style of instantiating variables (forward move) and re-instantiating variables (backward move). They differ in that *CDBT* searches instantiations to variables in a variable set from a given constraint posed on that variable set and appends it to a partial solution. When a partial solution cannot be extended, i.e., when a dead-end is encountered, *CDBT* backtracks to a previously instantiated variable set, re-instantiates variables in that set, and continues from there. In this way, *CDBT* has a much more limited search space than BT has. The similarity between *CDBT* and

BT enables us to incorporate the existing advanced techniques into *CDBT* to improve its performance further.

In the following sections, we describe the *CDBT* algorithm, analyze its complexity, prove its correctness, and report our preliminary experimental results.

2 Definitions

A **constraint satisfaction problem** is a structure $< X, D, C >$ where $X = \{X_1, X_2, \ldots, X_n\}$ is a set of variables that may take on values from a set of domains $D = \{D_1, D_2, \ldots, D_n\}$, and $C = \{C_I, C_J, \ldots, C_K\}$ is a set of constraints. Each constraint C_I of C is in the form of $< V_I, S_I >$, where $V_I = \{X_{i_1}, X_{i_2}, \ldots, X_{i_{mi}}\}$ is an ordered subset of X and S_I is a subset of $D_{i_1} \times D_{i_2} \times \ldots \times D_{i_{mi}}$. In other words, C_I is a constraint posed on V_I to limit the values that they can take on.[1] The problem is to find one (or all) solution(s).

A **solution** to the CSP is an n_ary tuple sol from $D_1 \times D_2 \times \ldots \times D_n$ such that sol satisfies all constraints; i.e., for all C_I in C, the projection of sol on V_I is an element of S_I. A **partial solution** to a subset of variables $V_H = \{X_{h_1}, X_{h_2}, \ldots, X_{h_{mh}}\}$ is an mh_ary tuple sol_H from $D_{h_1} \times D_{h_2} \times \ldots \times D_{h_{mh}}$ such that for all $C_I \in C$ where $V_I \subset V_H$ the projection of sol_H on V_I is an element of S_I.

Let V_H and V_K be subsets of variables, sol_H a partial solution to V_H. Partial solution sol_K to V_K is **combinable** with sol_H if either $V_H \cap V_K = \emptyset$, or $V_H \cap V_K = V_{HK} \neq \emptyset$ and the projection of sol_K on V_{HK} is the same as the projection of sol_H on V_{HK}.

Example 1 We consider the 4-queens problem where we need to place 4 queens in a 4 by 4 chess board such that they do not attack each other. This can be formulated as a binary CSP with four variables $V = \{X_1, X_2, X_3, X_4\}$. Each variable corresponds to a row, and its value represents which column to place a queen. The domain of each variable is $\{1, 2, 3, 4\}$. The constraints specified in the problem description exist between every pair of variables:

$C = \{C_{34}, C_{23}, C_{12}, C_{24}, C_{13}, C_{14}\}$, where

$V_{34} = \{X_3 X_4\}$, $V_{23} = \{X_2 X_3\}$, $V_{12} = \{X_1 X_2\}$,
$V_{24} = \{X_2 X_4\}$, $V_{13} = \{X_1 X_3\}$, $V_{14} = \{X_1 X_4\}$.

$S_{34} = S_{23} = S_{12} = \{(31), (41), (42), (13), (14), (24)\}$,

$S_{24} = S_{13} = \{(21), (41), (12), (32), (23), (43), (14), (34)\}$,

$S_{14} = \{(21), (31), (12), (32), (42), (13), (23), (43), (24), (34)\}$.

Let $V_{234} = \{X_2, X_3, X_4\}$ and $V_{12} = \{X_1, X_2\}$ be two subsets of variables, $sol_{234} = (241)$ a partial solution to V_{234}. Partial solution $sol_{12} = (42)$ to V_{12} is combinable with $sol_{234} = (241)$ to V_{234}, but partial solution $sol_{12} = (41)$ to V_{12} is not combinable with $sol_{234} = (241)$. Note that although $sol_{12} = (42)$ is combinable with $sol_{234} = (241)$, the combined tuple (4241) is not a partial solution to V_{1234}, since the constraint C_{13} is violated.

[1] For simplicity, we assume that $\forall I, J (C_I \in C \wedge C_J \in C \wedge I \neq J \Rightarrow V_I \neq V_J \wedge V_I \not\subset V_J \wedge V_J \not\subset V_I)$.

3 CDBT Algorithm

The *CDBT* algorithm is defined by two recursive procedures, *forward* and *goback*. Suppose that we have already found a partial solution sol_I to variable set V_I. Procedure *forward* extends this partial solution by appending to it instantiations of variables in another selected variable set on which there exists a given constraint. It first selects a $C_J =< V_J, S_J >$ from the given constraint set C, then it chooses a tuple tup from S^*_J containing those tuples in S_J, which are combinable with sol_I as instantiations of variables in V_J, and appends tup to sol_I to form a tuple tup_K, which is tested to see if it is a partial solution to variable set $V_K = V_I \cup V_J$. If tup_K is a partial solution to V_K, *forward* is called recursively to extend tup_K. If tup_K is not a partial solution to V_K, another tuple from S^*_J is chosen and appended to form another tup_K, which is tested again. If no tuples are left in S^*_J to be chosen, *goback* is called to re-instantiate variables in a selected variable set, which were previously instantiated.

Procedure *goback* first selects a constraint $C_J =< V_J, S_J >$ from constraint set C_0 containing constraints $C_L =< V_L, S_L >$, where V_L has already been instantiated. Then it re-instantiates variables in V_J by choosing another tuple from S^*_J and forms a new tup_K which is tested to see if it is a partial solution to variable set V_K. If tup_K is a partial solution to V_K, *forward* is called to extend tup_K. If tup_K is not a partial solution to V_K, another tuple from S^*_J is chosen and appended to form another tup_K, which is tested again. If S^*_J is empty, *goback* is called recursively to re-instantiate variables in another selected variable set.

$forward(V_I, sol_I)$:

begin
    if $|V_I| = n$ **then return** sol_I;
    select $C_J =< V_J, S_J >$ from C;
    $V_K \leftarrow V_I \cup V_J$;
    compute $cks(V_K) = \{C_H | C_H \in C, V_H \neq V_J, V_H \not\subset V_I, V_H \subset V_K\}$;
    compute $S^*_J = \{tup | tup \in S_J, tup$ is combinable with $sol_I\}$;
    while $S^*_J \neq \emptyset$ **do**
        $tup \leftarrow$ one tuple taken from S^*_J;
        $join(sol_I, tup, tup_K)$;
        if $test(tup_K, cks(V_K))$ **then**
            move C_J from C to C_0;
            $forward(V_K, tup_K)$;
    end while
    $goback(V_I, sol_I)$;
end

$goback(V_K, sol_K)$:

begin
    if $|C_0| = 0$ **then return** $failure$;
    select C_J from C_0;
    move those C'_J (moved to C_0 after C_J) from C_0 to C;
    $V_I \leftarrow V_K - \cup V'_J$;
    $sol_I \leftarrow proj(sol_K, V_I)$
    while $S^*_J \neq \emptyset$ **do**
        $tup \leftarrow$ one tuple taken from S^*_J;

        $join(sol_I, tup, tup_K)$;
        if $test(tup_K, cks(V_K))$ **then** $forward(V_K, tup_K)$;
    end while
    move C_J from C_0 to C;
    $goback(V_I, sol_I)$;
end

Let tup_I and tup_J be instantiations of variables in V_I and V_J respectively. The procedure $join(tup_I, tup_J, tup_K)$ produces a tuple tup_K, an instantiation of the variables in $V_K = V_I \cup V_J$, such that $proj(tup_K, V_I) = tup_I$ and $proj(tup_K, V_J) = tup_J$.

The function $test(tup_K, cks(V_K))$ returns $true$ if the tuple tup_K satisfies all the constraints in $cks(V_K)$ and $false$ otherwise. It is defined as follows:

$test(tup_K, cks(V_K))$

begin
    for each C_H in $cks(V_K)$ **do**
        if $proj(tup_K, V_H) \notin S_H$ **then return** *false*;
    return *true*;
end

The function $proj(tup, V_K)$ returns a $|V_K|\_ary$ tuple, which is the projection of tup on the variable subset V_K.

To find a solution to a given $CSP < V, U, C >$, *CDBT* first selects a constraint C_I from C, moves C_I from C to C_0, takes a tuple sol_I from $S_I^* = S_I$, and then calls $forward(V_I, sol_I)$.

Example 2 For the 4-queens problem in *Example 1*, let *CDBT* choose the constraints of the order: $\{C_{34}, C_{23}, C_{12}\}$ in the *forward* procedure and backtrack to the most recently instantiated variable set. The process of *forward* and *goback* can be illustrated by the backtrack tree in Figure 1, where the up-down arrows indicate the forward moves and the down-up arrows the backward moves.

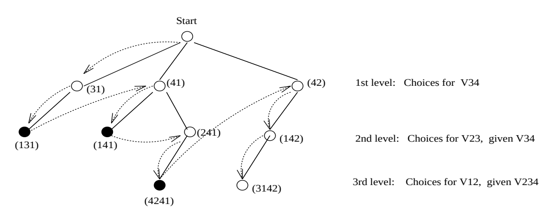

Figure 1: CDBT backtrack tree for the 4-queens problem

So far we have been assuming that constraints such as $C_I =< V_I, S_I >$, posed on a subset of variables $V_I = \{X_{I_1}, X_{I_2}, \ldots, X_{I_{mi}}\}$, are given in the form of a relation, i.e., $S_I \subset D_{I_1} \times D_{I_2} \times \ldots \times D_{I_{mi}}$. To handle constraints expressed in other forms, a procedure is needed to generate the relational form from the original form. Suppose we are given a CSP $< V, U, C^0 >$ where C^0 is the set of constraints such that S_I^0 are expressed in a non-relational form. A procedure $generate(S_I^0, S_I)$ is needed to generate a relation S_I on $D_{I_1} \times D_{I_2} \times \ldots \times D_{I_{mi}}$ from the given non-relational constraint $C_I^0 =< V_I, S_I^0 >$. We can modify the *CDBT* algorithm to deal with non-relational constraints by preprocessing the constraints C^0 using the procedure *generate* to produce the set of relational constraints C.

Notice that in both *forward* and *goback*, the selection of C_J is arbitrary. This feature gives *CDBT* the flexibility to adopt other tree search methods, such as *BJ, CBJ, FC* and so on, to improve its efficiency. However, if the order of variable sets is decided before the call of *forward* procedure and the most recently instantiated variable set is always chosen to be backtracked to in *goback*, *CDBT* has the same style of *forward* and *backward* moves as the chronological BT has.

4 The Search Space of CDBT

Given a constraint satisfaction problem $< V, U, C >$, where $V = \{X_1, X_2, \ldots, X_n\}$, $U = \{D_1, D_2, \ldots, D_n\}$, $C = \{C_I, C_J, \ldots, C_K\}$, and each constraint C_I is in the form of $< V_I, S_I >$, where $V_I = \{X_{i_1}, X_{i_2}, \ldots, X_{i_{mi}}\}$ is a subset of V and S_I is a subset of $D_{i_1} \times D_{i_2} \times \ldots \times D_{i_{mi}}$. The chronological backtracking algorithm searches the instantiation of a variable from its domain, and its search space is of the size $\prod_{j=1}^{n} |D_j|$. To find a solution to a given problem, *CDBT* selects a subset of constraints $C' = \{C_{H_1}, C_{H_2}, \ldots, C_{H_l}\}$ such that $\bigcup_{j=1}^{l} V_{H_j} = V$. *CDBT* searches the instantiations of variables in the variable set V_{H_i} from the given constraint posed on that variable set, that is, from S_{H_i}. Therefore, the search space can be seen as a l level search tree. For example, a search tree for 4-queens problem is shown in Figure 2.

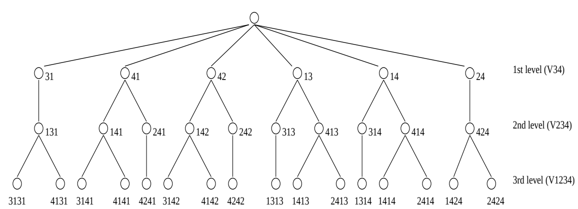

Figure 2: The search tree of CDBT for the 4-queens problem

Let N_i be the number of nodes at the ith level, V_I the set of variables that have been instantiated up to the ith level, and Sol_i^j the partial solution corresponding to the jth node at the ith level. We have $N_1 = |S_{H_1}|$, $N_i = \sum_{j=1}^{N_{i-1}} |S_{H_i}^{*j}|$ for $i = 2, 3, \ldots, l$, where $S_{H_i}^{*j} = \{t | t \in S_{H_i}, proj(t, V_I \cap V_{H_i}) = proj(Sol_i^j, V_I \cap V_{H_i})\}$. In the worst case, it can be shown that $N_l = \prod_{j=1}^{n} |D_j|$; that is, *CDBT* has the same size search space as standard backtrack has. However, for almost all the problems, $N_l = \sum_{j=1}^{N_{l-1}} |S_{H_l}^{*j}|$ is much less than $\prod_{j=1}^{n} |D_j|$.

Considering the *n-queens* problem ($n > 3$) where $C_{i,i+1}$, $i = 1, 2, \ldots, n-1$ are chosen in the *forward* procedure. The number of nodes in the search tree can be calculated as follows. At the first level, the number of nodes is $N_1 = (n-1)(n-2)$. At ith level, where $i = 2, 3, \ldots, n-1$, the number of nodes is $N_i = 2G_{i-1} + (n-3)N_{i-1}$ where G_i can be calculated by $G_i = N_{i-1} - G_{i-1} + \sum_{j=1}^{i-1} (-1)^j G_{i-j} + (-1)^{i-1}$ and $G_1 = n-2$. It can be shown that N_{i-1} is much less than N^i.

As another example, let us look at *ZEBRA* problem [16, 3]. It has 25 variables: five (house) colors, five nationalities, five brands of cigarettes, five pets, and five drinks.

Each of the variables has a domain of $\{1, 2, 3, 4, 5\}$ corresponding to the house number. Suppose we choose the variable sets of the basic constraints (i.e., each of the houses is a different color, inhabited by a different nationality, who smokes a different brand of cigarettes, owns a different pet, and prefers a different drinks) to be instantiated in the *forward* procedure; then the search space of *CDBT* is $(5!)^5$ instead of 5^{25}.

In general, if the average tightness of constraints is α $(0 < \alpha < 1)$, N_l will be at least $(\frac{1}{\alpha})^l$ times less than $\prod_{j=1}^{n} |D_j|$.

5 Correctness of CDBT

An algorithm is correct if it is sound (finds only solutions), complete (finds all solutions), and terminates ([14]). In this section, we show that *CDBT* is correct.

When a solution is found by *CDBT* , *CDBT* traverses an l level backtrack search tree. Along the path from the root to the solution node, the ith level node corresponds to an instantiation to variables in variable set $V^i = \bigcup_{j=1}^{i} V_{H_j}$. It is guaranteed, by performing *test* at line 10 in procedure *forward* and at line 10 in procedure *goback*, that the instantiation satisfies all those constraints C_J such that $V_J \subset V^i$. In other words, the ith node corresponds to a partial solution to variables in V^i. Consequently, the final l level node corresponds to a partial solution to variables in $V^l = \bigcup_{j=1}^{l} V_{H_j} = V$. It is a solution to the given problem.

To prove the completeness of *CDBT* , we suppose that *CDBT* is used repeatedly to search all the solutions. Let $(d_1, d_2, \ldots, d_n)$ be a solution. For any selected constraint subset $C' = \{C_{H_1}, C_{H_2}, \ldots, C_{H_l}\}$ where $\bigcup_{j=1}^{l} V_{H_j} = V$, we permute $(d_1, d_2, \ldots, d_n)$ into $(d'_1, d'_2, \ldots, d'_n)$ such that $(d'_1, d'_2, \ldots, d_{|V_{H_1}|'})$ is an instantiation to variables in V_{H_1}, $(d'_1, d'_2, \ldots, d'_{|V_{H_1} \cup V_{H_2}|})$ is an instantiation to variables in $V_{H_1} \cup V_{H_2}$, $(d'_1, d'_2, \ldots, d'_{|V^i|})$ is an instantiation to variables in $V^i = \bigcup_{j=1}^{i} V_{H_j}$, and so on. If *CDBT* is used repeatedly, since $(d'_1, d'_2, \ldots, d'_{|V^i|})$ satisfies all constraints on V^i, the node corresponding to this tuple will be visited and found consistent. Therefore, $(d'_1, d'_2, \ldots, d'_n)$ will be found by *CDBT* as a solution.

CDBT terminates when $|V_K| = n$ turns to be true at line 11 in procedure *forward* (a solution to the given problem is found) or when $|C_0| = 0$ is true at line 13 in procedure *goback* (no solution exists to the given problem is discovered). In both cases, the number of recursive calls of *forward* and *goback* is bounded by $\prod_{i=1}^{l} |S_{H_i}|$.

6 Experiments

To evaluate the performance, we tested *CDBT* on the *n-queens* problem, where n varies from 3 to 10, and compared it with the chronological backtracking program. The *n-queens* problem is chosen since it has been widely used to rank tree search methods, although *CDBT* 's advantages are more pronounced when it is used to solve general CSPs. We implemented the basic *CDBT* for *n-queens* problem, where the variable pairs $\{V_{n-1,n}, V_{n-2,n-1}, \ldots, V_{2,1}\}$ are chosen in the *forward* procedure and it backtracks to the most recently instantiated variable pair when a dead-end occurs.

The Prolog implementations of both *CDBT* and *BT* do not include any heuristics. The order of variables to be instantiated and the order of values to be chosen are the same for both programs, so that they give the same output. We first use the programs to find the first solution to *n-queens* problems and then to find all the solutions. The number of nodes visited and the number of consistency checks are recorded and summarized in Table 1. This shows that the number of nodes and checks recorded from *CDBT* is significantly smaller than that from *BT* (with one exception for finding the first solution to 7-queens problem).

no. of	First Solution				All Solutions				
var-	BT		CDBT		no. of	BT		CDBT	
ables	nodes	checks	nodes	checks	solutions	nodes	checks	nodes	checks
3	18	17	4	11	0	18	17	4	11
4	26	31	9	23	2	68	84	22	34
5	15	22	6	33	10	270	453	104	157
6	171	314	89	181	4	918	1754	488	856
7	42	87	21	95	40	3864	8791	2180	4527
8	876	2205	564	1394	92	16456	42296	10266	24634
9	333	935	229	684	352	75546	216149	49856	134146
10	975	2987	692	2113	724	355390	1115840	249976	743816

Table 1: The result of solving n-queens problem

Further experiments are being conducted on *n-queens* problem, where *CDBT* is embedded with the technique of *BJ, CBJ, or FC*. We intend to compare its performance with the similar ones (e.g., *CDBT* embedded with BJ to BJ). Unfortunately, no result can be reported at this time.

7 Related Work

There are a few decomposition techniques developed for solving CSPs [5, 10, 13] which we consider are related to *CDBT* in that they all treat particular variable sets as singleton variables and apply backtracking to search for solutions.

To see the similarity and difference among them, we borrow some definitions from graph theory [1, 12].

Let $H =< X, E >$ be a hypergraph where $X =< X_1, X_2, \ldots, X_n >$ is a finite node set and E a family of subsets of X (hyper-edges of H). A *partial hypergraph* of H is a hypergraph $H' =< X, E' >$ such that $E' \subseteq E$. A *line graph* of H is the graph $GR(H) =< E, F >$, where $F = \{(E_i, E_j) | i \neq j, E_i \in E, E_j \in E, E_i \cap E_j \neq \emptyset\}$. An intergraph of H is a graph $G(H) =< E, K >$, where $K \subseteq F$ and $\forall E_i, E_j \in E$, if $E_i \cap E_j \neq \emptyset$, there exists in $G(H)$ a chain $(E_i = E_1, E_2, \ldots, E_q = E_j)$ such that $\forall l, 1 \leq l < q, E_i \cap E_j \subseteq E_l \cap E_{l+1}$. A minimal intergraph of H is an intergraph $G_m(H) =< E, K_m >$, where K_m is minimal w.r.t. inclusion (i.e. there is no $K'_m \subset K_m$ such that $G(H) =< E, K'_m >$ is an intergraph).

Given a CSP $< X, D, C >$, where $C = \{C_I, C_J, \ldots, C_K\}$, and each C_I is in the form $< V_I, S_I >$ (S_I may not be explicitly given). Let $V = \{V_I, V_J, \ldots, V_K\}$ (also called scheme of constraints) and $S = \{S_I, S_J, \ldots, S_K\}$. We have a hypergraph $H^c =< X, V >$ (called constraint hypergraph), where nodes are variables and hyper-edges are defined by constraint scheme.

Let Φ be a family of subsets of V that covers V, Ψ be a family of subset of X such that $\Psi_i = \cup\Phi_i$. The existing decomposition methods are, first, to find a Φ such that the hypergraph $H^\psi =< V, \Psi >$ has an acyclic minimal intergraph $G_m(H^\psi) =< \Psi, V^\psi >$, then, to translate the original CSP to a binary CSP $< \Psi, D^\psi, C^\psi >$. The scheme of the binary CSP $< \Psi, D^\psi, C^\psi >$ is V^ψ. Similarly and also differently, the *CDBT* algorithm is, first, to find a partial hypergraph of H_c, $H^p =< X, P >$ where $P \subseteq V$ and $\cup P = X$, then to translate the original CSP $< X, D, C >$ to a binary CSP $< P, D^p, C^p >$. The scheme of the binary CSP $< P, D^p, C^p >$ is

$V^p = \{(V_J, V_K) | V_J \cap V_K \neq \emptyset \text{ or } \exists V_H \in V s.t. V_H \subseteq V_J \cup V_K\}$.

Informally, *CDBT* can be described as follows:

- *CDBT* algorithm:
  1. selecting a subset P of V such that $\cup P = X$,
  2. constructing a sub-partial graph induced by P,
  3. solving the given problem by searching the constructed graph.

Let us compare *CDBT* with the other two well-referred decomposition schemes:

- The *Tree Clustering Scheme* (TC) described in [5]:
  1. identifying all the maximal cliques of variables,
  2. ordering cliques by constructing a *join tree* of cliques (as nodes),
  3. finding all solutions to each subproblem represented by each node in the join tree,
  4. solving the given problem by searching the join tree.
- The *Hinge Decomposition Scheme* (HD) described in [10]:
  1. generating minimal hinges,
  2. constructing a hinge tree T of minimal hinge (as nodes),
  3. finding all solutions to each subproblem represented by each node in the hinge tree,
  4. solving the given problem by searching the hinge tree.

In both *TC* and *HD* schemes, the set of maximal cliques (nodes of the join tree) and the set of minimal hinges (nodes of hinge tree) are the family of subset of V (the scheme of the original constraints). In the case that the cardinality of maximal clique (or minimal hinge) is huge, (e.g., it is close to $|X|$ for hard 3SAT problems), finding all solutions to the subproblems is quite inefficient and unnecessary. For those CSPs (e.g., n-queens problem) with only one maximal clique or only one minimal hinge (i.e., the whole variable set X), both *TC* and *HD* degenerate into any method that is used in the third step and lose any of the advantage completely. On the one hand, *CDBT* is a general method that can be applied to any kind of problems without losing its advantages. On the other hand, the join tree or the hinge tree can serve as an excellent constraint ordering heuristics for selecting C_J in procedure *forward* so that backtracking (if it has to backtrack) can be limited within certain areas of the search tree. Without finding all solutions to subproblems, *CDBT* can be modified to perform tree search at different level and may be implemented parallelly. However, this will be the future work.

8 Future Work and Conclusion

We presented a new backtracking algorithm *CDBT* for CSPs, which has a much more limited search space than generic backtracking algorithm does and which has the flexibility to embed other tree search methods to improve the performance further.

Current work is being carried out on the following aspects:

- Select C_J in procedure *forward* (forward move for instantiation) in order to limit the number of backtracks. The criterion is that if the given *CSP* has a tree structure hypergraph, the selected constraint subset should be in such an order that a solution can be found with at most 2-bounded backtracks ([6]). For a *CSP* of general dual constraint graph, the selected constraint subset should be in such an order that the backtracking should be limited in the smallest subsets of constraints.
- Define consistency for general CSPs and develop efficient method for enforcing such consistency.
- Implement *CDBT* incorporated with *BJ, CBJ* and *FC* and evaluate the performance.

References

[1] C. Berge. *Hypergraphs*. North-Holland, New York, 1989.

[2] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[3] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[4] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[5] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[6] E. Freuder. Backtrack-free and backtrack-bounded search. In L. Kanal and V. Kumar, editors, *Search in Artificail Intelligence*, pages 343–369. Springer-Verlag, New York, 1988.

[7] J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of the 3rd International Joint Conference on AI*, page 457, Cambridge, MA, 1977.

[8] J. Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the 2nd Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 268–277, Toronto, Ont., July 1978.

[9] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the Association for Computing Machinery*, 12(4):516–524, 1965.

[10] M. Gyssens, P. G. Jeavons, and D. A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–89, 1994.

[11] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[12] P. Jegou. On some partial line graphs of a hypergraph and the associated matroid. *Discrete Mathematics*, 111:333–344, 1993.

[13] P. Jegou. On the consistency of general constraint satisfaction problems. In *Proceedings of AAAI-93*, pages 114–119, 1993.

[14] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In *Proceedings of IJCAI-95*, pages 541–547, Montreal, Canada, August 1995.

[15] B. A. Nadel. Tree search and arc consistencey in constraint satisfaction algorithm. In L. Kanal and V. Kumar, editors, *Search in Artificail Intelligence*, pages 287–341. Springer-Verlag, New York, 1988.

[16] P. Prosser. Hybrid algorithms for the constrain satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[17] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, San Diego, CA, 1993.