University of Windsor

Scholarship at UWindsor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

1-1-2007

Dynamic backtracking for general CSPs.

Kan Yu
University of Windsor

Dynamic Backtracking for General CSPs

by

Kan Yu

A Thesis
submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2007

Your file Votre référence
ISBN: 978-0-494-35062-1
Our file Notre référence
ISBN: 978-0-494-35062-1

Abstract

There are two categories of CSPs: binary CSPs and general CSPs. A binary CSP has only unary and binary constraints. A unary constraint restricts the value of one variable while a binary constraint restricts the values of two variables. A general CSP may have constraints that restrict more than two variables. Many algorithms have been developed to solve CSPs. Dynamic Backtracking and Constraint-directed Backtracking algorithms (CDBT) are two of them. This thesis introduces a new general CSP-solving algorithm – Constraint-directed Dynamic Backtracking (CDDBT) that combines the advantages of Dynamic Backtracking and CDBT.

Dedication

To

my father and mother

Acknowledgements

I would like to thank my supervisor Dr. Scott Goodwin, for his guidance, for his academic advice, and for his support. I am also grateful for the scholarships provided by the University of Windsor.

I also extend my appreciation to the members of my committee – Dr. Dan Wu and Dr. Myron Hlynka, for their feedback and academic advice. I would like to thank Dr. Richard Frost, the chair of my committee, for his guidance and academic advice when I wrote my 510 survey.

I would like to thank Dr. Liwu Li, for his guidance. His sudden and unexpected death was a shock to everyone. He is sincerely missed.

I would like to thank Robert George Price, for his academic advice and help. I would also like to thank Robert Effinger, for his comments on Dynamic Backtracking.

I want to especially thank my parents for their love and full support.

Table of Contents

List of Tables

List of Figures

1 Introduction

In Artificial Intelligence (AI), a lot of problems can be represented as Constraint Satisfaction Problems (CSPs). We can find them in many fields of AI such as machine vision, belief maintenance, scheduling problems, temporal reasoning, graph-coloring problems, bioinformatics, and so on.

1.1 Definition of CSP

One definition of CSP from (Russell & Norvig, 2003) is:

"A constraint satisfaction problem (or CSP) is defined by a set of variables, X_1, X_2, . . . , X_n, and a set of constraints, C_1, C_2, . . . , C_m. Each variable X_i has a nonempty domain D_i of possible values."

Another definition from (Tsang, 1993) is:

"A constraint satisfaction problem is a triple: (Z, D, C)

where Z = a finite set of variables { $X_1, X_2, \ldots, X_n$ }.

D = a function which maps every variable in Z to a set of objects of arbitrary type.

C = a finite (possibly empty) set of constraints on an arbitrary subset of variables in Z."

Other researchers have definitions with different representations, but they all contain the three key elements of CSP: variables, domains, and constraints. The arity of a constraint is the number of involved variables. A solution to a CSP is an assignment of values to all variables that does not violate any constraints. A CSP may have one solution, more than one solution, or no solution.

There are two categories of CSPs: binary CSPs and general CSPs. A binary CSP has only unary and binary constraints. A unary constraint restricts the value of one variable while a binary constraint restricts the values of two variables. A general CSP may have constraints that restrict more than two variables. Many algorithms have been developed to solve CSPs. Dynamic Backtracking (Ginsberg, 1993) and Constraint-directed Backtracking algorithms (CDBT) (Pang, 1998) are two of them.

1.2 Examples of CSP

There are many CSPs in different areas. For example, one well-known CSP is the 8-queens problem. A chess player named Max Bezzel originally proposed this problem in 1848. Over the years, many mathematicians and Computer Scientists have worked on the problem. The problem is to put eight queens on an 8×8 chessboard such that no two queens can attack each other. The 8-queens problem has 92 distinct solutions (12 solutions if not counting symmetry operations).

One formalization of the 8-queens problem makes each row a variable {V1, V2, . . . , V8}. The domain of each variable is one of eight columns {1, 2, . . . , 8}. The constraint of the 8-queens problem is "no two queens can attack each other", which means that no two queens are on the same row, column, or diagonal. If we set V1=1, we cannot set V2=1 or V2=2. Another formalization can be made by representing the problem with the same number of variables but a different domain {1, 2, . . . , 64}, which stands for 64 positions on the 8×8 chessboard.

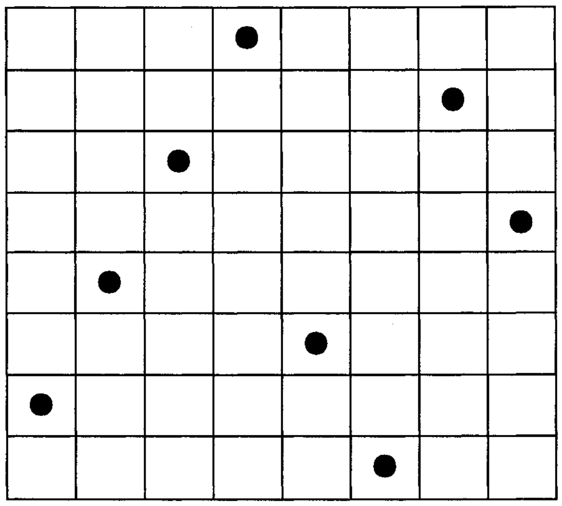


Figure 1-1 8-queens problem

Another well-known but harder CSP problem is the car sequencing problem. The goal of the problem is to find an optimal arrangement of cars along a production line, given production requirements, option requirements and capacity constraints. The detailed description can be found in (Tsang, 1993). Other famous examples are Crossword Puzzles, Map-Coloring problems, and so on.

1.3 Motivation

1. Dynamic Backtracking for binary constraints continues to be a focus of research (Effinger & Williams, 2006) and (Zivan, Shapen, Zazone, & Meisels, 2006). Its concept of "eliminating explanation" can be applied to both binary and non-binary CSPs. However, Dynamic Backtracking with non-binary cases has not been completely investigated.

2. The power of the constraint-directed mechanism, the core of CDBT, has not been sufficiently mined. The constraint-directed mechanism can be added to many CSP approaches such as Forward Checking, Backjumping, and so on, as mentioned in (Pang & Goodwin, 1996).

3. We wish to determine whether the performance improvements provided by

Dynamic Backtracking in the binary case can be carried over to the non-binary case.

2 Background

2.1 Basic Concepts

2.1.1 Unary, Binary and General Constraints

There are two categories of CSPs: binary CSPs and general CSPs. General CSPs are also called non-binary CSPs. A binary CSP has only unary and binary constraints. A unary constraint restricts the value of one variable while a binary constraint restricts the values of two variables. A general CSP may have constraints that restrict more than two variables. In (Rossi, Petrie, & Dhar, 1990), the authors claim that it is possible to convert any non-binary CSP to a binary CSP having the same solutions. However, the efficiency of converting and then applying a binary CSP-solving algorithm may not be as good as simply applying a non-binary CSP-solving algorithm directly.

2.1.2 Density and Tightness

$$Density = \frac{the\_number\_of\_constraints}{the\_number\_of\_all\_possible\_constraints}$$

For example, if a CSP has three variables $\{V_1, V_2, V_3\}$, we have seven all possible constraints, which are $\{V_1\}$, $\{V_2\}$, $\{V_3\}$, $\{V_1, V_2\}$, $\{V_1, V_3\}$, $\{V_2, V_3\}$, and $\{V_1, V_2, V_3\}$. The number of all possible constraints is defined by $2^{the\_number\_of\_variables} - 1$. If the CSP has only one constraint $C_1 = \{V_1, V_2\}$, the density of the CSP is $\frac{1}{7} \approx 0.14$.

$$Tightness\_of\_a\_constraint = \frac{the\_number\_of\_valid\_tuples\_of\_a\_constraint}{the\_number\_of\_all\_possible\_tuples\_of\_a\_constraint}$$

For example, if the domains of the above CSP are $D_1 = \{1, 2\}$, $D_2 = \{1, 2, 3\}$, $D_3 = \{1,$

2, 3, 4}, all possible tuples of C_1 are (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), and (2, 3). The number of all possible tuples of C_1 is defined by $|D_1| \times |D_2| = 2 \times 3 = 6$. If the CSP has three valid tuples (2, 1), (2, 2), and (2, 3), the tightness of C_1 is $\frac{3}{6} = 0.5$.

Some researchers use an opposite definition:

$$Tightness\_of\_a\_constraint = \frac{the\_number\_of\_invalid\_tuples\_of\_a\_constraint}{the\_number\_of\_all\_possible\_tuples\_of\_a\_constraint}$$

2.1.3 Constraint Graphs

A binary CSP can be represented as an undirected graph. In the graph, the nodes stand for variables and the edges stand for binary constraints. A General CSP can be represented as a hypergraph. Graph theory has a significant influence on CSP research. A CSP can be unconnected (Figure 2.1).

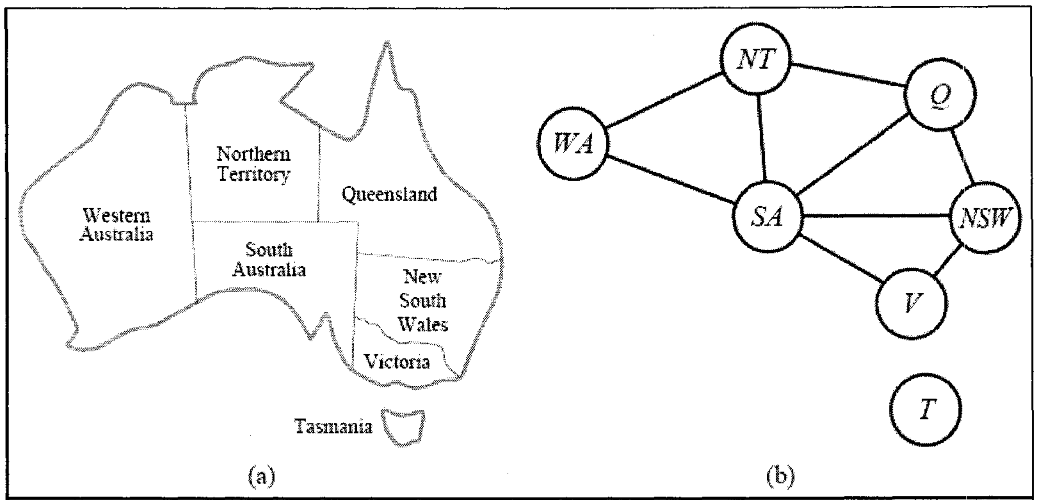


Figure 5.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Figure 2-1 A constraint graph (Russell & Norvig, 2003)

2.1.4 Satisfiability and Consistency

Two fundamental concepts in CSP are satisfiability and consistency. In (Tsang, 1993), the author introduces the concept of compound label. A compound label is an assignment of values to variables like (<Variable1, value1>, <Variable2, value2>, . . . , <VariableX, ValueX>). A constraint can also be viewed as a set of legal compound labels. He also introduces a simple definition of satisfiability, which is "a compound label X satisfies a constraint C if and only if X is an element of C". Based on this simple definition of satisfiability, related concepts are built such as satisfiable, k-satisfies, and k-satisfiable (Tsang, 1993).

Consistency is another essential concept in CSP. According to (Tsang, 1993), "a CSP is 1-consistent if and only if every value in every domain satisfies the unary constraints on the subject variable. A CSP is k-consistent, for k greater than 1, if and only if all $(k-1)$ compound labels which satisfy all relevant constraints can be extended to include any additional variable to form a k-compound label that satisfies all the relevant constraints".

Satisfiability and consistency have a close relationship. They support many other important concepts and theorems in CSP research, for example, the concepts of node consistency (NC, same as 1-consistency), arc consistency (AC, same as 2-consistency), and path consistency (PC, same as 3-consistency in binary CSP).

2.1.5 Search Ordering

Search ordering is one of the most fundamental factors that affect the efficiency of CSP-solving algorithms (Tsang, 1993). Search ordering includes ordering of both variables and values in their domains. For example, in (Russell & Norvig, 2003), the authors introduce three heuristics: the minimum remaining values (MRV) heuristic, the degree heuristic, and the least-constraining-value heuristic. The MRV heuristic picks a variable that has fewer remaining values. The degree heuristic picks a variable that is "involved in the largest number of constraints on other unassigned variables". The least-constraining-value heuristic picks a value that "rules out the fewest choices for the neighboring variables in the constraint graph."

For example, we have a CSP, which has ten variables $\{V_1, V_2, \ldots, V_{10}\}$. Each variable has the same domain $\{1, 2, \ldots, 100\}$. After we assign 1 to V_1, we are going to pick the next variable. V_2 to V_9 have more than one remaining value. V_{10} has one only value left, which is 5. Instead of picking V_2, the MRV heuristic will pick V_{10}. If it cannot assign 5 to V_{10}, we need backtrack. It may save time since we don't need to assign values to variables between V_2 and V_9.

2.2 CSP-solving Techniques

2.2.1 General Discussion

Modeling or representing a problem as a CSP is one area in CSP research. How to solve a CSP is another important area. Over thirty years, CSP researchers have developed different kinds of methods or algorithms that can solve CSPs.

(Tsang, 1993) classifies techniques in CSP-solving into three categories: *problem reduction*, *search*, and *solution synthesis*. Each category corresponds to one chapter in his book. In the problem-reduction chapter, the author mainly talks about NC, AC, and PC algorithms. In (Russell & Norvig, 2003) problem-reduction methods are classified as constraint propagation methods. In the search chapter, Tsang introduces three categories of search strategies: general search strategies, lookahead strategies, and gather-information-while-searching strategies. Most of the CSP-solving algorithms can be found in this chapter such as Backtracking, Forward Checking, Backjumping, Backchecking, Backmarking, and so on. In the solution-synthesis chapter, the author mainly talks about GENET. In the rest of the chapters, the author introduces other important techniques like stochastic search.

In this section, firstly, two CSP algorithms are used as examples: backtracking and AC-3. Secondly, a lot of work about systematic and non-systematic search is introduced. Thirdly, the performance of CSP-solving techniques is discussed. The approach of Dynamic Backtracking (Ginsberg, 1993) and CDBT (Pang, 1998) are two other CSP-solving algorithms. As these are of prime importance in this thesis, they will be described separately in later sections.

2.2.2 Backtracking

The Backtracking algorithm is a fundamental CSP-solving algorithm, which is the basis of many other algorithms. It was first formally introduced by (Bitner & Reingold, 1975). However, the basic idea of Backtracking can be traced back to the 19th century. Furthermore, it is often compared with other algorithms to evaluate their performance. Backtracking, or backtracking search, is a depth-first search. It is shown in Figure 2.2.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment according to CONSTRAINTS[csp] then
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

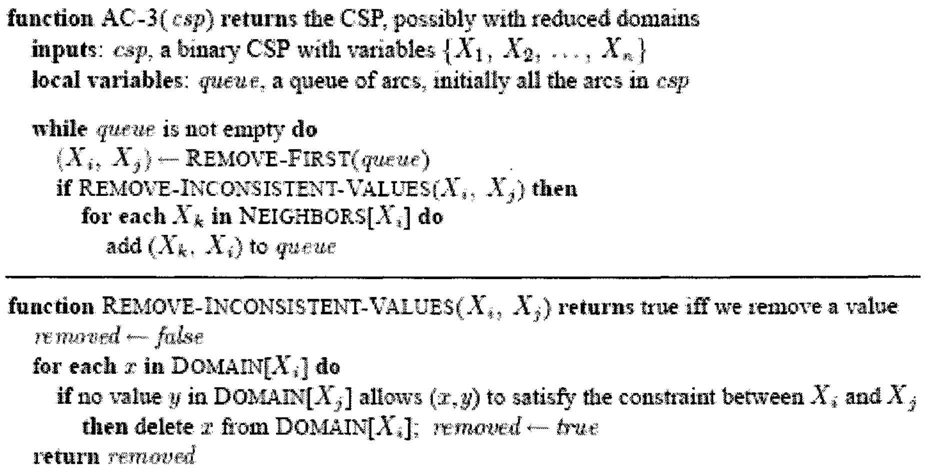
Figure 2-2 Backtracking search (Russell & Norvig, 2003)

Backtracking tries to assign a value to a variable. If it does not violate any constraints, it will assign a value to the next variable. If it fails to assign a value, it will backtrack to the previous variable.

2.2.3 AC-3 algorithm

One important class of CSP-solving algorithms is called "arc consistency" algorithms (Mackworth, 1977a). Achieving consistency is also called problem reduction (Tsang, 1993), problem relaxation, or constraint propagation. In (Montanari, 1974), the author introduces the concept of constraint networks and propagation using path consistency. This approach was popularized by (Waltz, 1975). By achieving certain consistency (NC, AC, or PC), the problem is reduced by eliminating redundant information from domains and constraints. In other words, "an arc consistency algorithm can be thought of as a simplification algorithm which transforms the original problem into a simpler version that has the same solutions" (Nadel, 1989). Consistency concepts are so defined to guarantee it. Another property of arc consistency mentioned in (E. C. Freuder, 1982) is that in any binary CSP, if its constraint graph can be represented as a tree, a backtrack-free search can be obtained if node and arc consistency are obtained.

NC, AC, and PC are different levels of consistency. In (Nadel, 1989), the author classifies AC algorithms into two categories: partial arc consistency algorithms ($AC^{1/5}$, $AC^{1/4}$, $AC^{1/3}$, and $AC^{1/2}$) and full arc consistency algorithms (AC1, AC2, and AC3). In (Tsang, 1993), the author lists another AC algorithm: AC4. AC-3 (Mackworth, 1977a) is a widely-used algorithm:



```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {$X_1$, $X_2$, ..., $X_n$}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    ($X_i$, $X_j$) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES($X_i$, $X_j$) then
      for each $X_k$ in NEIGHBORS[$X_i$] do
        add ($X_k$, $X_i$) to queue

function REMOVE-INCONSISTENT-VALUES($X_i$, $X_j$) returns true iff we remove a value
  removed ← false
  for each $x$ in DOMAIN[$X_i$] do
    if no value $y$ in DOMAIN[$X_j$] allows ($x$,$y$) to satisfy the constraint between $X_i$ and $X_j$
      then delete $x$ from DOMAIN[$X_i$]; removed ← true
  return removed
```

Figure 2-3 AC-3 (Russell & Norvig, 2003)

For example, we have a CSP, which has ten variables {V_1, V_2, ..., V_{10}}. Each variable has the same domain {1, 2, ..., 100}. AC-3 tries to maintain arc consistency. If we assign 1 to V_1, we find that no value can be chosen from V_2's domain that satisfies the constraint between {V_1, V_2}. Then 1 will be removed from V_1's domain. If we assign 3 to V_1, we find that no value can be chosen from V_8's domain that satisfies the constraint between {V_1, V_8}. Then 3 will be removed from V_1's domain.

2.2.4 Systematic and Non-systematic Search

Systematic search (global search) and non-systematic search (local search) are two categories of CSP-solving methods. In (F. Freuder, Dechter, Ginsberg, Selman, & Tsang, 1995), Dechter credits the work done in (Pearl, 1984): "Systematic algorithms have two properties (1) Do not leave any stone unturned (completeness), and (2) do not turn any stone more than once (efficiency)." Dechter claims that greedy non-systematic algorithms may "leave many stones unturned and may also turn the same stone multiple times". Here, efficiency does not mean performance. She also claims that systematic search can beat non-systematic search sometimes, and vice versa. The following papers discuss systematic search and/or non-systematic search. Others can be found in later Sections.

In (Minton, Johnston, Philips, & Laird, 1990), the problem addressed by the authors is meaningful progress on how to solve large-scale constraint satisfaction and scheduling problems. Three previous papers referred to by the authors are (Stone & Stone, 1987), (Johnston & Adorf, 1989), and (Adorf & Johnston, 1990). The authors develop a new heuristic called the min-conflicts heuristic that captures the idea of Guarded Discrete Stochastic (GDS) Network. The main idea of the min-conflicts heuristic is to minimize the number of conflicts by assigning a new value to the variable, which is in conflict. The authors do experiments by employing three search strategies (hill-climbing, informed backtracking, and best-first search) with the min-conflicts heuristic. They claim that min-conflicts hill-climbing and min-conflicts backtracking perform much better than basic backtracking on the n-queens problem. They also claim that the min-conflicts heuristic is less effective on problems like coloring sparsely-connected graphs. They state that these problems have a few highly-critical constraints and many less important constraints. This paper has been cited by many researchers such as (Minton, Johnston, Philips, & Laird, 1992) and (Ginsberg, 1993).

After two years, four authors presented another paper (Minton et al., 1992). They analyzed the min-conflicts heuristic. They state that (Johnston & Adorf, 1989) and (Adorf & Johnston, 1990) inspired their heuristic. Adorf and Johnston developed a neural network called GDS network. Minton et al. raise a question "why does the GDS network perform so well". They state both a non-systematic search hypothesis and an informedness hypothesis. They claim that the informedness hypothesis is the reason. By capturing the idea of GDS, the authors state the min-conflicts heuristic. The heuristic assigns a value of a variable in conflict while the value minimizes the number of conflicts. The authors also claim that many search strategies can use the method of repairing an inconsistent assignment except the hill-climbing strategy. This paper has been cited by many researchers such as (Davenport, Tsang, Zhu, & Wang, 1994) and (F. Freuder et al., 1995).

In (Davenport et al., 1994), the authors introduce a new connectionist architecture - GENET that solves CSPs using iterative improvement methods. One previous work referred to by the authors is (Minton et al., 1992). The authors state that the GENET network is similar to the GDS network. One significant difference from GDS is that GENET has a learning procedure. In order to escape local minima, they introduce a rule for adjusting the weights of the connections. The authors introduce two specific constraints: illegal constraints and atmost constraints, in addition to general constraints. They do experiments on the Graph Coloring problem, random general constraint satisfaction problems, and the Car Sequencing Problem. They test five different algorithms namely MCHC, MCHC2, GENET, GENET2, and GENET3. The authors claim that GENET outperforms other existing iterative improvement techniques. This paper has been cited by many researchers such as (F. Freuder et al., 1995).

2.2.5 Performance of CSP Algorithms

In (Nadel, 1988), the author evaluates some CSP-solving algorithms on n-queens and confused n-queens problems. He claims that Forward Checking (FC) performs best among these algorithms. In (Kumar, 1992), the author lists three schemes of CSP-solving techniques: backtracking, constraint propagation, and constraint propagation inside backtracking. The author claims that the drawbacks for backtracking are thrashing (Gaschnig, 1979) and redundant work. For example, algorithms using a backtracking mechanism may keep backtracking for the same reason. Kumar (1992) claims that there are two possible reasons for thrashing: node inconsistency and arc inconsistency (Mackworth, 1977a). On the other hand, he also states that constraint propagation is more expensive than simple backtracking in most cases. So the author raises a question - "how much constraint propagation is useful." In (Mackworth & Freuder, 1993), the authors compare and analyze the complexity of many finite CSP (FCSP) algorithms such as AC-1, AC-2, AC-3, and AC-4. They state that it is important to identify tractable problem classes that are specific classes with tractable solution techniques.

2.3 Dynamic Backtracking

2.3.1 Problem Addressed

In (Ginsberg, 1993), the problem addressed by the author is that meaningful progress is sometimes removed in existing backtracking methods. For example, Backtracking suffers from thrashing. Two previous papers referred to by the author are Dependency-directed backtracking (Stallman & Sussman, 1977) and Backjumping (Gaschnig, 1979). They both suffer from this problem. In (Ginsberg, 1993), the author introduces a new algorithm called Dynamic Backtracking that can solve this problem.

2.3.2 Definitions

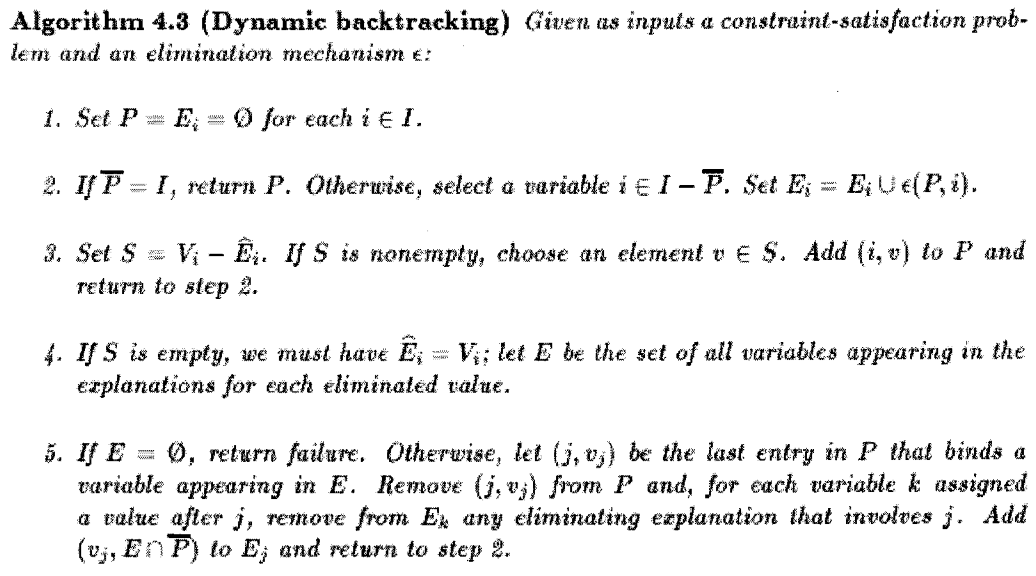
Ginsberg uses another definition of the CSP. He defines a CSP as "a set I of variables; for each $i \in I$, there is a set of V_i of possible values for the variable i. k is a set of constraints, each a pair (J, P) where $J= (j_1, \ldots, j_k)$ is an ordered subset of I and P is a subset of $V_{j1} \times \cdots \times V_{jk}$". Because i is unique, the author uses it to indicate both a variable and the index of a domain.

The most important concept the author introduced is the concept of an eliminating explanation. "Given a partial solution P^1 to a CSP, an eliminating explanation for a variable i is a pair (v, S) where v $\in V_i$ and $S \subseteq \overline{P}$." $\overline{P}$ is the corresponding set of variables for P. The underlying meaning of eliminating explanation is that i cannot be set to v because of the values that are already set by P to the variables in S. An eliminating mechanism ε is a function. It takes two inputs: a partial solution P and a variable $i \notin \overline{P}$. It outputs an eliminating explanation set $\varepsilon(P, i)$ for i.

[1] Note this P is different from the one in the previous paragraph.

2.3.3 The Algorithm

The author reconstructs the depth-first search algorithm and the Backjumping algorithm with his notations of CSP and the concept of eliminating explanation. Then he gives the algorithm of Dynamic Backtracking:



Algorithm 4.3 (Dynamic backtracking) *Given as inputs a constraint-satisfaction problem and an elimination mechanism ϵ:*

1. *Set $P = E_i = \emptyset$ for each $i \in I$.*
2. *If $\overline{P} = I$, return P. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = E_i \cup \epsilon(P, i)$.*
3. *Set $S = V_i - \widehat{E}_i$. If S is nonempty, choose an element $v \in S$. Add (i, v) to P and return to step 2.*
4. *If S is empty, we must have $\widehat{E}_i = V_i$; let E be the set of all variables appearing in the explanations for each eliminated value.*
5. *If $E = \emptyset$, return failure. Otherwise, let (j, v_j) be the last entry in P that binds a variable appearing in E. Remove (j, v_j) from P and, for each variable k assigned a value after j, remove from E_k any eliminating explanation that involves j. Add $(v_j, E \cap \overline{P})$ to E_j and return to step 2.*

Figure 2-4 Dynamic Backtracking (Ginsberg, 1993)

The essential difference from previous methods is that the author saves nogood information based on the current assignment. A nogood is dropped if it depends on old information. The author compares Dynamic Backtracking with Backjumping by the experiment of generating nineteen puzzles of different sizes. Similar work has been done in (Ginsberg, Frank, Halpin, and Torrance, 1990). The author claims that Dynamic Backtracking has better performance than Backjumping. He claims that, in nineteen tests, Dynamic Backtracking beats Backjumping in six and obtains the same performance as Backjumping in the other thirteen. Future work suggested by the author is backtracking to older culprits and dependency pruning.

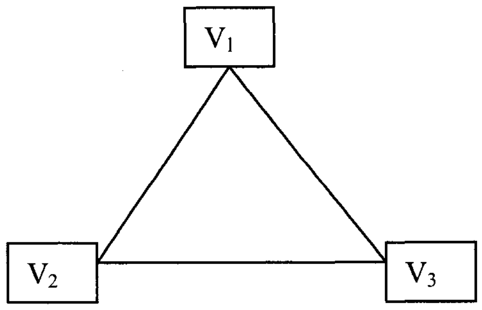
2.3.4 Example

Given a simple CSP:

Variables: V_1, V_2, V_3

Domains: $D_1=D_2=D_3= \{1, 2\}$

Constraints: $V_1 \neq V_2$, $V_2 \neq V_3$, $V_3 \neq V_1$



Initially:

	Eliminating Explanations	Assigned Value
V_1		
V_2		
V_3		
P		
$\overline{P}$		
E		

Iteration 1:

Select the first variable V_1 --> calculate E_1, the set of eliminating explanations for V_1. Because partial solution $P = \emptyset$, E_1 is $\emptyset$ --> assign the first valid value to V_1, which is 1 --> add $(V_1, 1)$ to P

	Eliminating Explanations	Assigned Value
V_1		1
V_2		

V_3		
P	{(V_1, 1)}	
$\overline{P}$	{V_1}	
E		

Iteration 2:

Select next variable V_2 --> calculate E_2, which is {(1, {V_1})} --> assign the first valid value to V_2, which is 2 --> add (V_2, 2) to P

	Eliminating Explanations	Assigned Value
V_1		1
V_2	(1, { V_1})	2
V_3		
P	{(V_1, 1), (V_2, 2)}	
$\overline{P}$	{V_1, V_2}	
E		

Iteration 3:

Step1:

Select next variable V_3 --> calculate E_3, which is {(1, {V_1}), (2, {V_2})}

	Eliminating Explanations	Assigned Value
V_1		1
V_2	(1, { V_1})	2
V_3	(1, { V_1}), (2, { V_2})	
P	{(V_1, 1), (V_2, 2)}	
$\overline{P}$	{V_1, V_2}	
E		

Step2:

Assign the first valid value to V_3, but no valid value can be found

	Eliminating Explanations	Assigned Value
V_1		1
V_2	(1, { V_1})	2
V_3	(1, { V_1}), (2, { V_2})	cannot assign a value
P	{(V_1, 1), (V_2, 2)}	
$\overline{P}$	{V_1, V_2}	
E		

Step3:

E is the set of all variables appearing in the explanations for each eliminated value. It needs to be calculated. Then {V_1, V_2} is assigned to *E*.

	Eliminating Explanations	Assigned Value
V_1		1
V_2	(1, { V_1})	2
V_3	(1, { V_1}), (2, { V_2})	cannot assign a value
P	{(V_1, 1), (V_2, 2)}	
$\overline{P}$	{V_1, V_2}	
E	{V_1, V_2}	

Step 4:

Unlike Backtracking, which will backtrack directly to the previous variable, Dynamic Backtracking removes the last entry in *P* while the variable of this entry is in *E*. However, in this example, the entry happens to be (V_2, 2). Then, for every variable after V_2, we remove all eliminating explanations that involve V_2.

	Eliminating Explanations	Assigned Value
V_1		1
V_2	(1, { V_1})	

V_3	(1, { V_1})	
P	{(V_1, 1)}	
$\overline{P}$	{V_1}	
E	{V_1}	

Step 5:

Add $(2, E \cap \overline{P})$ to E_2

	Eliminating Explanations	Assigned Value
V_1		1
V_2	(1, { V_1}), (2, { V_1})	
V_3	(1, { V_1})	
P	{(V_1, 1)}	
$\overline{P}$	{V_1}	
E	{V_1}	

Iteration 4:

Step1:

We select next variable. Here we select V_2 again. --> calculate E_2, which is still {(1, {V_1}), (2, {V_1})}

	Eliminating Explanations	Assigned Value
V_1		1
V_2	(1, { V_1}), (2, { V_1})	
V_3	(1, { V_1})	
P	{(V_1, 1)}	
$\overline{P}$	{V_1}	
E	{V_1}	

Step2:

Assign the first valid value to V_2, but no valid value can be found

	Eliminating Explanations	Assigned Value
V_1		1
V_2	(1, { V_1}), (2, { V_1})	cannot assign a value
V_3	(1, { V_1})	
P	{(V_1, 1)}	
$\overline{P}$	{V_1}	
E	{V_1}	

Step3:

E needs to be calculated. Then {V_1} is assigned to *E*.

	Eliminating Explanations	Assigned Value
V_1		1
V_2	(1, { V_1}), (2, { V_1})	cannot assign a value
V_3	(1, { V_1})	
P	{(V_1, 1)}	
$\overline{P}$	{V_1}	
E	{V_1}	

Step 4:

Dynamic Backtracking removes the last entry in *P* while the variable of this entry is in *E*. The entry is (V_1, 1), which is the only one left. Then, for every variable after V_1, we remove all eliminating explanations that involve V_1.

	Eliminating Explanations	Assigned Value
V_1		
V_2		
V_3		
P		

$\overline{P}$	
E	

Step 5:

Add $(1, E \cap \overline{P})$ to E_1. Because E and $\overline{P}$ are both $\emptyset$, $E \cap \overline{P}$ is Φ. So we add (1, $\emptyset$) to E_1. $E_1 = \{(1, \emptyset)\}$ means V_1 cannot be assigned to 1 whatever assignments of other variables are.

	Eliminating Explanations	Assigned Value
V_1	(1, Ø)	
V_2		
V_3		
P		
$\overline{P}$		
E		

Iteration 5 to the end:

We select V_1 again, and we assign 2 to V_1. Following the similar steps, (2, $\emptyset$) has been added to E_1. We backtrack to V_1 again. At this time, no value is valid for V_1. Then we have $E = \emptyset$. The algorithm terminates and returns failure, which means there is no solution for this CSP.

	Eliminating Explanations	Assigned Value
V_1	(1, Ø), (2, Ø)	
V_2		
V_3		
P		
$\overline{P}$		
E		

In this specific CSP, the performance of Dynamic Backtracking may not be as good as Backtracking because it just backtracks to the previous variable as Backtracking does. In other CSPs (Ginsberg, 1993), Dynamic Backtracking may backtrack to some variable other than the previous variable.

2.3.5 Related Work

Dynamic Backtracking is a systematic search technique. In (Jonsson & Ginsberg, 1993), the authors make a comparison between systematic and non-systematic search techniques. They compare the performance of depth first search and three new search methods, which are Dynamic Backtracking (Ginsberg, 1993), Minimum Conflicts hill climbing (Minton et al., 1990) and GSAT (Selman, Levesque, & Mitchell, 1992). The authors do experiments mainly on the graph-coloring problem because they state that it is the best problem to evaluate these methods' performance among graph-coloring problem, n-queens problem, and crossword puzzles. The authors claim some results. For example, they claim that Dynamic Backtracking performs better than the non-systematic methods in graph coloring problem. Future work suggested by the authors is that people can compare their work with similar work done at the AT&T Bell Laboratories.

In (Ginsberg & McAllester, 1994), the authors introduce a new algorithm that combines both systematic and non-systematic approaches. Two previous works referred to by the authors are Dynamic Backtracking (Ginsberg, 1993) and GSAT (Selman et al., 1992). The authors use the notation of nogoods instead of constraints in standard definition of CSP. The new algorithm is called Partial-order Dynamic Backtracking (PDB). In this algorithm, they also introduce two new concepts: safety conditions and weakening. In experiment (3-SAT problem), the authors compare PDB with WSAT and TABLEAU. They claim that PDB performs the best among these three algorithms. Two type of future work are suggested by the authors. First, more problems need to be tested. Second, there are a few untouched questions about the flexibility of PDB.

In (F. Freuder et al., 1995), the problem addressed by the authors is systematic and stochastic control in CSP. Two previous works referred to by the authors are (Minton et al., 1992) and (Ginsberg & McAllester, 1994). Freuder states a lot of questions that relate to this problem. Dechter claims that, between systematic algorithms and stochastic greedy, the main job is how to exploit identified class-superior algorithms. Ginsberg states two observations about systematic and non-systematic search. Selman claims that it is better to formulate problems using model-finding rather than theorem proving. Tsang claims that stochastic search is more important in practical applications. This paper has been cited by many researchers such as (Gomes & Selman, 1997).

2.4 General CSPs

2.4.1 Introduction

More research has been done on binary CSPs than on general CSPs. One reason is that "new ideas/techniques are usually much simpler to present/elaborate by first restricting them to the binary case" (Bessiere, 1999). The other reason is that all CSP problems can be transformed into binary CSPs with some cost (Tsang, 1993). However, many researchers have done significant work on general CSPs.

2.4.2 Early Research

(Mackworth, 1977b) is one of the early works on general CSPs. The purpose of this paper is to describe a program, called MAPSEE, which interprets sketch maps. One previous work referred to by the author is (E. C. Freuder, 1976). Mackworth states that, first, there is a phase called the initial partial segmentation. Then the second phase addressed by the author is achieving consistency. In this period, he provides a new algorithm NC, an n-ary Relation Consistency Algorithm. He claims that NC is a generalized version of AC-3, which is more efficient than AC-3. In the end, the author states that there is some room for refining the initial segmentation. Future work suggested by the author includes the integration of segmentation and interpretation phases, the problem of automatically generating primary cue interpretation catalogue, and the use of schemata. This paper has been cited by many researchers such as (Bessiere, Meseguer, Freuder, & Larrosa, 1999) and (Bacchus, Chen, van Beek, & Walsh, 2002).

2.4.3 Later Research

In (Rossi et al., 1990), the problem addressed by the authors is that the old definition of equivalence of CSPs is limited. One previous work referred to by the authors is (Montanari, 1974). Two CSPs are equivalent based on the old definition of equivalence if they share the same solutions. The authors develop a new and more general definition of equivalence - extended equivalence. The authors introduce the concept of mutual reducibility as the base of extended equivalence. They claim to prove binary and non-binary CSPs are equivalent using a new definition of equivalence. The authors also introduce two algorithms for transforming non-binary CSPs into equivalent binary CSPs. They claim that one algorithm of them can produce an equivalent binary CSP and the other one can successfully transform with some cost. Future work suggested by the authors is that it is possible to generalize the new definition to other types of problems. This paper has been cited by many researchers such as (Bessiere et al., 1999) and (Bacchus et al., 2002).

In (Bacchus & van Beek, 1998), the problem addressed by the authors is that few theoretical and experimental works have been done on performance of non-binary CSPs and their binary representations. Two previous theoretical works referred to by the authors are (Mackworth, 1977b) and (Van Hentenryck, 1989). One previous experimental work referred to by the authors is (Ginsberg, 1993). The authors introduce a new algorithm called FC+ that is a modification of FC. In addition to pruning the domains of h-variables, FC+ also prunes the domains of corresponding uninstantiated variables. The authors claim that FC+ sometimes performs better than FC on non-binary CSPs. They also claim that the number of satisfying tuples may be the most important factor when we decide to translate or not. Future work suggested by the authors is to investigate the relationship between binary translations. This paper has been cited by many researchers such as (Bessiere, 1999).

In (Bessiere et al., 1999), the problem addressed by the authors is the problem of solving non-binary CSPs by extending binary search algorithms. One previous work referred to by the authors is (Rossi et al., 1990). In (Bessiere et al., 1999), the authors extend FC for non binary constraints. Depending on different alternatives of constraints involving past, current, and future variables, the authors introduce six algorithms (nFC0, nFC1, nFC2, nFC3, nFC4, and nFC5). The authors prove some results on the six algorithms. For example, they prove that these algorithms are all correct (soundness and completeness). To compare FC+, nFC0, nFC1, nFC2, nFC3, nFC4, and nFC5, they do three experiments on random problems, Schur's lemma, and the car sequencing problem. The authors claim that their performance has very close relationship with the tightness and arity of constraints. They also claim that their performance depends on the use of the semantics of constraints. Future work suggested by the authors is how to find a criterion to choose an appropriate nFCx algorithm. This paper has been cited by many researchers such as (Stergiou, 2001).

2.4.4 Current Research

In (Bacchus et al., 2002), the authors compare binary constraints and non-binary constraints. Two major previous works are (Dechter & Pearl, 1989) and (Rossi et al., 1990). The authors compare the dual transformation and the hidden transformation. The forward checking and maintaining arc consistency algorithms are used in the comparison. The two algorithms are two variations of the chronological backtracking algorithm. At every node in the search tree, they maintain a local consistency property. The authors prove some results from the comparison. For example, they prove that enforcing arc consistency on the original CSP is the same as its hidden transformation. They claim that their results can help users who want to apply the two transformations to a CSP model. This paper has been cited by (Stergiou & Walsh, 2006).

2.4.5 CDBT

In (Pang, 1998), Pang introduces an algorithm to solve non-binary CSPs. One previous work referred to by the author is (Pang & Goodwin, 1996). The algorithm is called constraint-directed backtracking algorithm (CBDT). He claims that a shortcoming of traditional backtracking is that all given constraints are as criterion functions when we check consistency. The most significant feature of CBDT is that it assigns values to the variables from some constraint simultaneously. However, other CSP-solving algorithms usually assign one value to one variable. The author claims that CBDT has a more limited search space than Backtracking and other tree search algorithms.

Pang gives his own definition of CSP: "A constraint satisfaction problem is a structure (X, D, V, S). X={X_1, X_2, ..., X_n} is a set of variables, D={ D_1, D_2,..., D_n} is a set of domains where each domain D_i is a set of possible values for variable X_i, and V={V_1, V_2,..., V_m} is a family of ordered subsets of X called constraint schemes. Each V_i = {X_{i1}, X_{i2},..., X_{iri}} is associated with a set of tuples $S_i \subseteq D_{i1} \times D_{i2} \times ... \times D_{iri}$ called a constraint instance, and S={ S_1, S_2,..., S_m} is a family of such constraint instances. Together, a pair (V_i, S_i) is a constraint (or relation) which permits the variable in V_i to take only the value combinations in S_i."

The CDBT algorithm (Pang, 1998) is described as the following three functions:

```
forward(IP, $V_I$, $tup_I$)
1. begin
2.    if | $V_I$ | = n then return $tup_I$;
3.    select $C_{i+1}$=($V_{i+1}$, $S_{i+1}$) from C s.t. $V_{i+1} \not\subset V_I$;
4.    cks'() $\leftarrow$ {$C_h$ | $C_h \in C$, $V_h \neq V_{i+1}$, $V_h \not\subset V_I$, $V_h \subset V_{I+1}$ };
5.    $S^*_{i+1}$ $\leftarrow$ {tup | tup $\in S_{i+1}$, tup[$V_I \cap V_{i+1}$] = $tup_I$[$V_I \cap V_{i+1}$]};
6.    while $S^*_{i+1} \neq 0$ do
```

```
7.        tup ← one tuple taken from $S^*_{i+1}$;
8.        $tup_{I+1}$ ← $tup_I \bowtie tup$;
9.        if test($tup_{I+1}$, cks'($V_{I+1}$)) then return forward(IP, $V_{I+1}$, $tup_{I+1}$)
10.    end while
11. return goback(IP, $V_I$, $tup_I$)
12. end
```

```
goback(IP, $V_I$, $tup_I$)
1.  begin
2.     if $| V_I | = 0$ then return unsatisfiable;
3.     while $S^*_i \neq 0$ do
4.        tup ← one tuple taken from $S^*_i$;
5.        $tup_I$ ← $tup_{I-1} \bowtie tup$;
6.        if test($tup_I$, cks'($V_I$)) then return forward(IP, $V_I$, $tup_I$);
7.     end while
8.  return goback(IP, $V_{I-1}$, $tup_{I-1}$);
9.  end
```

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

Here, IP is a CSP. cks'(V_I) is a constraint check-set including the constraints which need checking for variable set V_I. V_I is all variables involved in constraints from all constraints that have been selected so far, whereas V_i is just the variables involved in the *ith* selected constraint. More description can be found in (Pang, 1998).

The key point of CDBT can be illustrated in the following example:

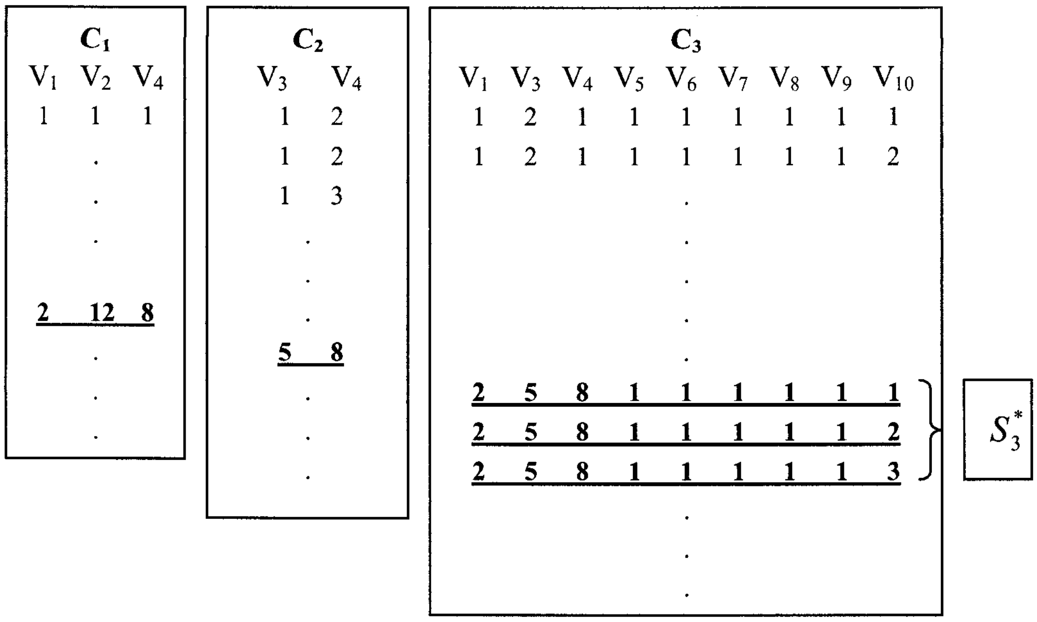
Given a simple CSP:

Variables: $V_1, V_2, \ldots, V_{20}$

Domains: $D_1=D_2=\ldots=D_{20}=\{1, 2, \ldots, 30\}$

Constraints: $C_1, C_2, \ldots, C_9$

Suppose we have already selected C_1 and C_2. So V_1, V_2, V_3, and V_4 have been assigned values. Next, we select C_3 and only consistent tuples can be considered. In other words, the tuples must include (V_1, 2), (V_3, 5), and (V_4, 8). Suppose we obtain 3 tuples (see C_3 in Figure 8). We put them into S_3^*. Then all the 3 tuples need to check consistency using constraint check-set. If we find such a tuple, we pick this tuple to build a partial solution. Then, we select next constraint. If we cannot find such a tuple, we need to backtrack and consider other tuples in S_2^*.



C_1

V_1	V_2	V_4
1	1	1
	.	
	.	
	.	
2	**12**	**8**
	.	
	.	
	.	

C_2

V_3	V_4
1	2
1	2
1	3
.	
.	
.	
5	**8**
.	
.	
.	

C_3

V_1	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
1	2	1	1	1	1	1	1	1
1	2	1	1	1	1	1	1	2
				.				
				.				
				.				
				.				
				.				
2	**5**	**8**	**1**	**1**	**1**	**1**	**1**	**1**
2	**5**	**8**	**1**	**1**	**1**	**1**	**1**	**2**
2	**5**	**8**	**1**	**1**	**1**	**1**	**1**	**3**
				.				
				.				
				.				

3 CDDBT

3.1 Methodology

Constraint-directed Dynamic Backtracking (CDDBT) is built on the basic structure of Dynamic Backtracking with some modifications. Three main modifications are:

1. Use the key mechanism of CDBT: constraint-directed. CDDBT chooses one constraint each time instead of variable.
2. Use a different eliminating explanation (t, C): Given a partial solution P, t is a tuple for a constraint i. C is a set of constraints. We are going to select i, which has not been selected before. V_i is the set of variables involved in i. (t, C) means that V_i cannot take the tuple t because of the tuples already assigned to some constraints in C. Different from the definition in (Ginsberg, 1993), C may have constraints that don't appear in P. This definition is more general than Ginsberg's. Ginsberg uses the first of his three assumptions to support his definition of eliminating explanation. Our definition has no restrictions.
3. Use a different eliminating mechanism ε from the ε of Dynamic Backtracking. In (Ginsberg, 1993), the author points out that his definition of elimination mechanism is "somewhat flexible with regard to the amount of work done by the elimination mechanism - all values that violate completed constraints might be eliminated, or some amount of lookahead might be done." There are two main rules for CDDBT's eliminating mechanism to guarantee the partial solution satisfies all related constraints. First, when a tuple needs to be eliminated, all reasons that cause elimination must be given. The eliminating explanation of the same tuple and the same reason is eliminated only once. Second, if a tuple for a constraint is found consistent with the partial solution, no more consistency checks are needed for this constraint.

3.2 The Algorithm

Notation:

csp	a CSP
vars	variables of *csp*, | *vars* | is the size of *vars*
I	constraints of *csp*
i	the new selected constraint
pse	a partial solution element (*c*, *t*). *c* is a constraint and *t* is one of valid tuples of *c*
P	a partial solution. It is a list of partial solution elements. | *P* | is the size of *P*
$\overline{P}$	constraints involved in *P*
(*t*, *C*)	an eliminating explanation (*t*, *C*): Given a partial solution *P*, *t* is a tuple for a constraint *c*. *C* is a set of constraints.
E_i	eliminating explanation set for *i* (because *i* is unique, it can be used as index also)
$\hat{E}_i$	tuples that are eliminated in E_i
$\varepsilon(P, i)$	elimination mechanism $\varepsilon(P, i)$ returns eliminating explanations for *i* when the partial solution is *P*. The tuples of *i* that are inconsistent with *P* are going to be eliminated until one consistent tuple is found.
cks (*pses*, *i*)	a constraint check set for *pses* and *i*. *pses* is a list of partial solution elements and $pses \subseteq P$. For example constraints C_1, C_2, ..., and C_j, are involved in *pses*, now we are going to choose constraint *i*. $cks = \{C_h \mid C_h \in I, C_h \neq i, V_h \not\subset VX_j, V_h \not\subset VX_j, V_h \subseteq VX_{j+1}\}$. Here, C_h is a constraint; V_h is the variables involved in C_h; VX_j is the variables involved in C_1, C_2, ..., and C_j; VX_{j+1} is the variables involved in C_1, C_2, ..., C_j, and *i*.
S	If only one solution is required, *S* contains the first tuple that is

	consistent with *P*. If all solutions are required, *S* contains the all tuples that are consistent with *P*.
E	*E* is the set of constraints appearing in the eliminating explanations for each eliminated tuple

Table 3-1 Notation of CDDBT

Constraint-directed Dynamic Backtracking (CDDBT)

Input: a CSP

Output: the first solution

```
1. $P \leftarrow \emptyset$
2. $E_i \leftarrow \emptyset$ for each $i$ in $I$
3. WHILE ($P$ doesn't covers $vars$ AND (at least one constraint has not been
           chosen AND it contains a variable that is not assigned a value))
4.     Choose a constraint $i$ from $I$ where $P$ doesn't cover $i$'s involved variables
5.     $E_i \leftarrow E_i \bigcup \varepsilon(P,i)$, $S$ is obtained when calculating $\varepsilon(P,i)$
6.     IF ($S$ <> $\emptyset$)
7.         Choose a tuple $t$ from $S$
8.         Add ($i$, $t$) to $P$
9.     ELSE
10.        IF ($E = \emptyset$ OR $P = \emptyset$)
11.            RETURN No Solution
12.        ELSE
13.            Let ($c$, $t$) be the last entry in $P$ that binds a constraint appearing
                   in $E$ if $E$ <> $\emptyset$; if not found, choose the last entry in $P$
14.            Remove ($c$, $t$) from $P$
15.            For each constraint $k$ in $P$ after $c$ or $k$ not in $P$,
                 Remove from $E_k$ any elimination explanation that involves $c$
16.            Add $(t, \overline{P})$ to $E_c$
17.        END OF IF
18.    END OF IF
19. END OF WHILE
20. Output $P$
```

Elimination mechanism $\varepsilon(P,i)$**:**

```
1.  tupleCanBeAdded ← false
2.  WHILE (tupleCanBeAdded = false AND at least one valid tuple cl in i has not
            been chosen)
3.      tupleAlreadyInEliminationExplanation ← false
4.      tupleIsPermanentlyEliminated ← false
5.      IF (cl = t and (t, C) is an eliminating explanation of $E_i$)
6.          tupleAlreadyInEliminationExplanation ← true
7.          IF (C = Ø)
8.              tupleIsPermanentlyEliminated ← true
9.          END OF IF
10.     END OF IF
11.     tupleNeedsEliminating ← false
12.     IF (tupleIsPermanentlyEliminated = false)
13.         IF (P = Ø)
14.             WHILE (at least one $c_l$ in cks(P, i) has not been chosen )
15.                 IF (cl violates $c_l$)
16.                     Add (cl, $c_l$) to $E_i$
17.                     tupleNeedsEliminating ← true
18.                 END OF IF
19.             END OF WHILE
20.         ELSE
21.             pseList ← Ø, constraintList ← Ø
22.             WHILE (at least one (c, t) in P has not been chosen)
23.                 Add (c, t) to pseList
24.                 Add c to constraintList
25.                 checkset ← cks(pses, i)
26.                 Add t to pscl
```

```
27.   IF ($cl$ violates $t$)
28.       Add ($cl$, $c$) to $E_i$
29.       tupleNeedsEliminating ← true
30.   ELSE
31.       tup ← $t \cup cl$
32.       WHILE (at least one $c_l$ in checkset has not been
               chosen )
33.           IF (tup violates $c_l$)
34.               Add ($cl$, $c_l$) to $E_i$
35.               tupleNeedsEliminating ← true
36.           END OF IF
37.       END OF WHILE
38.   END OF IF
39.   IF (($c$, $t$) is not the first element in pseList)
40.       IF ($cl$ violates pscl)
41.           Add ($cl$, constraintList) to $E_i$
42.           tupleNeedsEliminating ← true
43.       ELSE
44.           nppscl ← pscl $\cup$ cl
45.           WHILE (at least one $c_l$ in checkset has not been
                   chosen )
46.               IF (nppscl violates $c_l$)
47.                   Add ($cl$, constraintList) to $E_i$
48.                   tupleNeedsEliminating ← true
49.               END OF IF
50.           END OF WHILE
51.       END OF IF
52.   END OF IF
53.   WHILE (at least one $c_l$ in checkset has not been chosen )
54.       IF ($cl$ violates $c_l$)
```

```
55.                     Add (cl, Ø) to $E_i$
56.                     tupleNeedsEliminating ← true
57.                 END OF IF
58.             END OF WHILE
59.         END OF WHILE
60.     END OF IF
61. END OF IF
62. IF (tupleAlreadyInEliminationExplanation = false AND
       tupleNeedsEliminating = false)
63.     Add cl to S
64.     tupleCanBeAdded ← true
65. END OF IF
66. END OF WHILE
```

3.3 Proof

Theorem 3.1: If a CSP is solvable, CDDBT can always return a solution.

Proof. Suppose we have a simple CSP:

Variables: $V_1, V_2, \ldots, V_n$

Domains: $D_1=D_2=\ldots=D_n=\{d_1, d_2, \ldots, d_h\}$

Constraints: $C_1, C_2, \ldots, C_m$

We choose C_1 first. We must find at least one tuple t_1 from C_1 that satisfies constraints $C_2, \ldots, C_m$ because this CSP is solvable and elimination mechanism of CDDBT eliminates any tuple before t_1 that violates related constraints. If t_1 involves all variables form V_1 to V_n, t_1 is a solution. If not, we choose the next constraint C_k that involves at least one new variable. We look for a tuple t_2 from C_k that satisfies t_1 and related constraints. If we can not find it, we backtrack and look for another t_1. We must find at least one tuple t_2 from C_k that satisfies t_1 and related constraints because this CSP is solvable and elimination mechanism of CDDBT eliminates any tuple before t_k that violates t_l and related constraints. Then $t_l \cup t_k$ is a partial solution. If $t_l \cup t_k$ involves all variables form V_l to V_n, $t_l \cup t_k$ is a solution. If not, we choose the next constraint that involves at least one new variable. Following this procedure, we must find a solution.

3.4 An CDDBT Example

We use the same example as the example in CDBT:

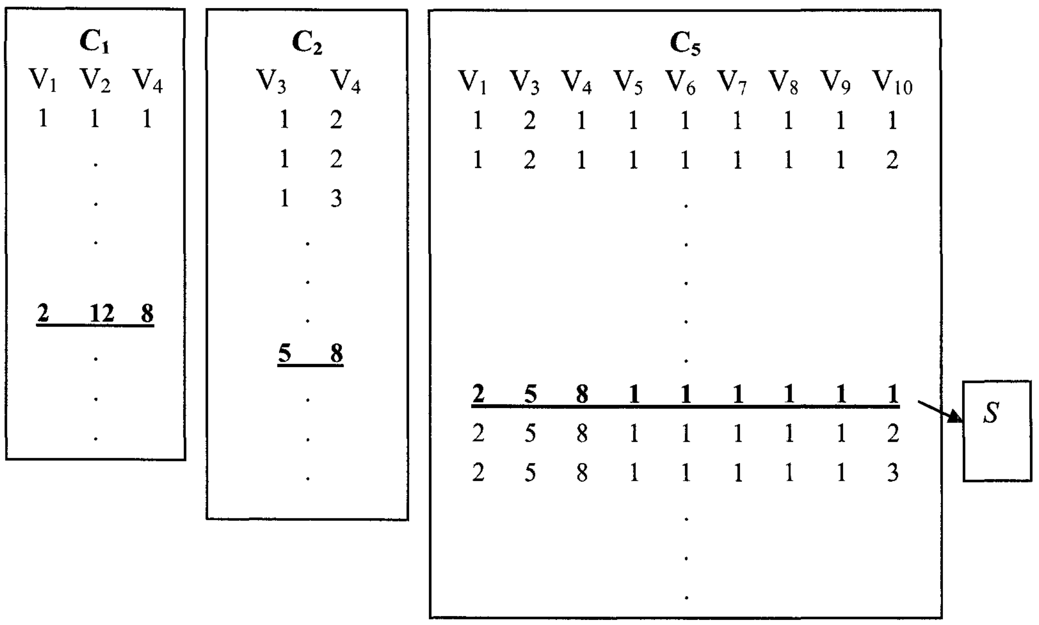
Given a simple CSP:

Variables: $V_1, V_2, \ldots, V_{20}$

Domains: $D_1=D_2=\ldots=D_{20}=\{1, 2, \ldots, 30\}$

Constraints: $C_1, C_2, \ldots, C_9$

Suppose we have already selected C_1 and C_2. So V_1, V_2, V_3, and V_4 have been assigned values. We select the next constraint whose involved variables are not a subset of $\{V_1, V_2, V_3, V_4\}$. Next, we eliminate the tuples that violate partial solution P and related constraints until we find one tuple in C_5 which is consistent. If we find that tuple, we put it into S and select the next constraint. If not, we put this inconsistent tuple with the reason (suppose C_1 and C_2) into E_5 and select the next constraint.


C₁
V₁ V₂ V₄
1 1 1
2 12 8
C₂
V₃ V₄
1 2
1 2
1 3
5 8
C₅
V₁ V₃ V₄ V₅ V₆ V₇ V₈ V₉ V₁₀
1 2 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1 2
2 5 8 1 1 1 1 1 1
2 5 8 1 1 1 1 1 2
2 5 8 1 1 1 1 1 3
S

4 Experiments

4.1 Random General CSP Generator

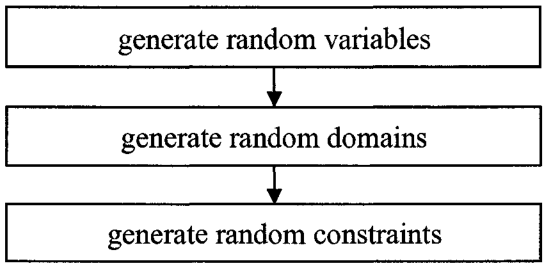
4.1.1 Methodology

At present, most random CSP generators are binary generators. Further more, they usually generate random CSPs with the same domain, the same arity, and the same tightness of each constraint. They only generate random tuples of each constraint. My random CSP generator is a general CSP generator and generates more random features. It generates a CSP with random variables, random domains, and random constraints.

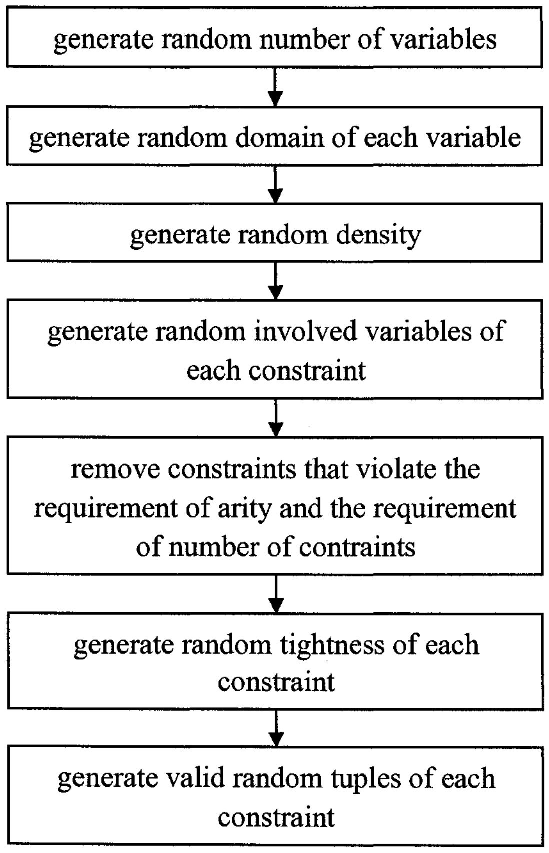
In (Gent, MacIntyre, Prosser, Smith, & Walsh, 2001), the authors state that "many models of random binary constraint satisfaction problems become trivially insoluble as problem size increases." They claim that one reason for the problem is the appearance of "flawed variables". Their definition of "flawed" is "A value for a variable is flawed if, when the value is assigned to the variable, there exists an adjacent variable in the constraint graph that cannot be assigned a value without violating the constraint between the two variables." They also cite an early work by (Achlioptas et al., 1997), in which the authors prove that if tightness is larger than some value (related to domain size), as the problem size increases, the generated random binary CSP may have a flawed variable. My random general CSP generator may also suffer from this problem. Our concern in this thesis is CSP-solving algorithms, not random CSP models.

Another problem of random CSP generators is: when density is large, we may generate the same involved variables of constraints again and again. The same applies to tightness of each constraint. When tightness is large, we may generate the same valid tuples of a constraint again and again. We call this the "unsuccessful hit" problem. We solve it using the following strategy. For example, if tightness *ti* is larger than 0.5, we first generate all possible tuples, then we generate invalid tuples with 1-*ti*, finally we can easily obtain valid tuples.

My random general CSP generator has three modules:



If we make this procedure more specific, that is:


generate random number of variables
generate random domain of each variable
generate random density
generate random involved variables of each constraint
remove constraints that violate the requirement of arity and the requirement of number of contraints
generate random tightness of each constraint
generate valid random tuples of each constraint

4.1.2 A random CSP example

The output of my random general CSP generator is a CSP. The following is an example:

```
 Arguments:

    numberOfCSPsNeedsToBeGenerated=1

    maximumNumberOfVariables=7

    maximumDomainSize=4

    maximumDensity=0.5

    maximumArity=7

    maximumTightnessOfConstraint=0.5

    connected=true

    randomLevel=00000



*****************CSP 0 has been generated****************

numberOfVariables=5

Because argument maximumArity>numberOfVariables, maximumArity is

assigend to 5

V0's domain(1 elements): 0

V1's domain(2 elements): 0 1

V2's domain(3 elements): 0 1 2

V3's domain(3 elements): 0 1 2

V4's domain(2 elements): 0 1

density=0.2242043106667143

numberOfAllPossibleConstraints=31

expected numberOfConstraints=6

cspIsConnected=true

numberOfConstraints=6,after remove from constraints that violate arity

requirement
```

```
C0's involved variables:  V4

this constraint's tightness=0.43176073273825644

numberOfAllPossibleTuples=2

numberOfTAllowedTuples=1

 <V4,1>


C1's involved variables:  V1 V3

this constraint's tightness=0.38900738448987093

numberOfAllPossibleTuples=6

numberOfTAllowedTuples=2

 <V1,0>  <V3,0>

 <V1,1>  <V3,0>


C2's involved variables:  V0 V2 V3 V4

this constraint's tightness=0.05772896162759178

numberOfAllPossibleTuples=18

numberOfTAllowedTuples=1

 <V0,0>  <V2,1>  <V3,1>  <V4,1>


C3's involved variables:  V1 V2 V4

this constraint's tightness=0.2511751603623239

numberOfAllPossibleTuples=12

numberOfTAllowedTuples=3

 <V1,0>  <V2,0>  <V4,0>

 <V1,1>  <V2,2>  <V4,1>

 <V1,1>  <V2,1>  <V4,0>


C4's involved variables:  V0 V1 V2 V4

this constraint's tightness=0.263554641222876
```

```
numberOfAllPossibleTuples=12

numberOfTAllowedTuples=3

 <V0,0>  <V1,1>  <V2,2>  <V4,1>

 <V0,0>  <V1,0>  <V2,1>  <V4,1>

 <V0,0>  <V1,1>  <V2,0>  <V4,0>


C5's involved variables:  V0 V3 V4

this constraint's tightness=0.44465671147816943

numberOfAllPossibleTuples=6

numberOfTAllowedTuples=2

 <V0,0>  <V3,2>  <V4,1>

 <V0,0>  <V3,0>  <V4,0>
```

4.2 Experiment Data

We are interested in how some CSP properties would influence the algorithms. These properties are number of variables, size of each domain, density (or number of constraints), arity, and tightness of each constraint. Three criteria (CPU runtime, node checks, and consistency checks) are used to evaluate the performance of CDBT and CDDBT.

criterion	definition
CPU runtime	the time between an algorithm starts and ends (milliseconds, approx.).
node checks	the number of constraints that an algorithm has visited (the same constraint may be visited more than once).
consistency checks	if we check consistency between two tuples, the number of consistency checks add one.

Table 4-1 Three criteria for comparing the performance of CDDBT and CDBT

The generator can generate both connected and unconnected CSPs, but in this thesis only connected and solvable CSPs are used. Thirty CSPs are generated for each configuration (a configuration consists of a given number of variables, a given domain size, density, arity, and tightness.) Then the average is put into the data tables. In the last three rows of each table, AVG is average number. STDEV is sample standard deviation, which is defined by $\sqrt{\frac{\sum(X-\overline{X})}{n-1}}$, where X is individual value, $\overline{X}$ is sample mean, and n is sample size (Bluman, 2001). *Total Wins* indicates the number that an algorithm has fewer CPU runtime, node checks, or consistency checks.

4.2.1 Target Property: number of variables

domain size=10, density =0.1, arity=3, tightness of constraint=0.5					
Configuration#	number of variables	CPU runtime			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	5	2.8	3.6	7.2	18.8
2	6	3	3.3	35.7	50.6
3	7	4.6	5.4	110.1	196.9
4	8	10.8	10.8	523	921.8
5	9	34.3	73	6207.8	17824.5
6	10	131.8	206.6	63186.2	194173
7	11	87.2	320.9	31021.8	109827.6
8	12	13.2	10.2	1041.5	1665.2
9	13	7.6	7.4	233.8	305.9
10	14	5.9	5.9	409.5	589.4
AVG	9.5	30.1	64.7	10277.7	32557.4
STDEV	3.0	44.1	110.3	20919.6	66238.0
Total Wins		10		0	

Table 4-2 Experiment #1(number of variables vs. CPU runtime)

domain size=10, density =0.1, arity=3, tightness of constraint=0.5					
Configuration#	number of variables	node checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	5	2.1	0.3	2.1	0.3
2	6	2.7	0.7	2.7	0.7
3	7	3.3	0.5	3.3	0.5
4	8	4.6	1.4	4.6	1.4

5	9	8.8	9.3	8.8	9.3
6	10	43	108.7	43	108.7
7	11	50.6	166.3	50.6	166.3
8	12	6.3	1.8	6.3	1.8
9	13	6.5	0.8	6.5	0.8
10	14	6.8	0.9	6.8	0.9
AVG	9.5	13.5	29.1	13.5	29.1
STDEV	3.0	17.8	58.8	17.8	58.8
Total Wins		0		0	

Table 4-3 Experiment #1(number of variables vs. node checks)

domain size=10, density =0.1, arity=3, tightness of constraint=0.5					
Configuration#	number of variables	consistency checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	5	656.5	293.3	815.8	1922.3
2	6	1339.7	780.6	18276.5	36621.5
3	7	1951.7	1256.9	27280.4	49838.8
4	8	5668.4	6141.8	230923.8	371988
5	9	33560.4	102820.1	2642506.8	7736933.1
6	10	147367.7	245254.7	30174343.8	86505843.5
7	11	89690.9	363024.5	11903416.1	44358703.4
8	12	7307.2	11113.2	198955.1	302261.8
9	13	3079	2189.2	72852.2	110486.1
10	14	2922.1	1142.6	67348.6	112551.7
AVG	9.5	29354.4	73401.7	4533671.9	13958715.0
STDEV	3.0	49867.4	128582.1	9739536.8	28987769.2
Total Wins		10		0	

Table 4-4 Experiment #1(number of variables vs. consistency checks)

4.2.2 Target Property: domain size

number of variables=10, density =0.1 , arity=3, tightness of constraint=0.5					
Configuration#	domain size	CPU runtime			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	4	20.2	39.8	456.9	1084.8
2	5	48.3	92.5	2368.9	3751
3	6	34.9	60.7	3307.8	9458.1
4	7	37.8	99.1	6789.4	21325.2
5	8	56.1	147.4	10453.6	36129.3
6	9	296.1	1494.7	27593.9	132988.9
7	10	303.7	1320.3	46453.8	185863.8
8	11	57.1	94.7	20870.8	35017.1
9	12	62.9	75.6	33578.1	73733.5
10	13	101.2	196	55652.3	95430.5
AVG	8.5	101.8	362.1	20752.6	59478.2
STDEV	3.0	106.6	554.3	19565.0	61904.4
Total Wins		10		0	

Table 4-5 Experiment #2(domain size vs. CPU runtime)

number of variables=10, density =0.1 , arity=3, tightness of constraint=0.5					
Configuration#	domain size	node checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	4	71.1	154.2	71.1	154.2
2	5	105.5	168.2	105.5	168.2

3	6	68.9	179.7	68.9	179.7
4	7	30.8	80.9	30.8	80.9
5	8	37	106.2	37	106.2
6	9	45.3	194.3	45.3	194.3
7	10	35.9	124.5	35.9	124.5
8	11	11.3	9.6	11.3	9.6
9	12	10.5	10.2	10.5	10.2
10	13	12.8	17.4	12.8	17.4
AVG	8.5	42.9	104.5	42.9	104.5
STDEV	3.0	30.9	72.0	30.9	72.0
Total Wins		0		0	

Table 4-6 Experiment #2(domain size vs. node checks)

number of variables=10, density =0.1 , arity=3, tightness of constraint=0.5					
Configuration#	domain size	consistency checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	4	5873.6	12235.5	150423.6	307210.7
2	5	32356.2	71230.4	1149792.9	2102456.2
3	6	28680.7	49400.7	1419459.8	3824804.4
4	7	38654.3	102066.7	4357396.4	14909110.4
5	8	63316.3	196480.6	3821391.5	11825787
6	9	391443.4	2027349.4	19702135	100137020.1
7	10	372913.1	1659342.6	19167865.7	79628910
8	11	57099.4	125810.4	7178282	13529756
9	12	59107.8	88559	9886455.4	21588321
10	13	146727.9	368773.3	19830199.1	39540189.4
AVG	8.5	119617.3	470124.9	8666340.1	28739356.5
STDEV	3.0	143333.1	735582.9	8055244.4	34484281.6

Total Wins		10		0	

Table 4-7 Experiment #2(domain size vs. consistency checks)

4.2.3 Target Property: number of constraints

domain size=10, density =0.1, arity=3, tightness of constraint=0.5					
Configuration#	number of constraints	CPU runtime			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	4	5.2	6.9	4.2	6.4
2	5	5.1	4.8	99.8	149.3
3	6	6.3	5.8	237.1	227.4
4	7	6.4	5.7	320.4	467.7
5	8	12.7	23.4	1001.2	1545.9
6	9	9.6	7.1	520.3	490.8
7	10	18.1	20.6	4181.6	10143.3
8	11	26.2	44.1	6441.9	22629.9
9	12	29.2	30.8	8233.6	15452.6
10	13	37	40.6	5484.4	13206.4
AVG	8.5	15.6	19.0	2652.5	6432.0
STDEV	3.0	11.5	15.3	3125.9	8282.6
Total Wins		9		1	

Table 4-8 Experiment #3(number of constraints vs. CPU runtime)

domain size=10, density =0.1, arity=3, tightness of constraint=0.5					
Configuration#	number of constraints	node checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	4	4	0	4	0

2	5	4.6	0.5	4.6	0.5
3	6	5.1	0.7	5.1	0.7
4	7	5.1	0.9	5.1	0.9
5	8	5.9	1.9	5.9	1.9
6	9	5.4	0.8	5.4	0.8
7	10	7.7	6.2	7.7	6.2
8	11	10.3	17.2	10.3	17.2
9	12	11.6	13.2	11.6	13.2
10	13	8.4	7.6	8.4	7.6
AVG	8.5	6.8	4.9	6.8	4.9
STDEV	3.0	2.6	6.1	2.6	6.1
Total Wins		0		0	

Table 4-9 Experiment #3(number of constraints vs. node checks)

domain size=10, density =0.1, arity=3, tightness of constraint=0.5					
Configuration#	number of constraints	consistency checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	4	1000	0	84.8	48.5
2	5	1595.2	525.1	11712.6	24870.4
3	6	2300.9	607.9	53685.8	98081.5
4	7	3147.2	1147.2	65535.4	92798.8
5	8	5293.8	4126.2	293218.7	462158.2
6	9	4941.7	3218.7	186416.3	181796.5
7	10	11719.7	11484.8	1148759.9	2290542.6
8	11	27706.6	62381.7	2032266.8	6295508.4
9	12	29155.7	35264.5	2891293.2	5208255.9
10	13	39954.6	57597.9	1983216.2	3280283.6

AVG	8.5	12681.5	17635.4	866619.0	1793434.4
STDEV	3.0	14198.6	24729.1	1072910.8	2377384.7
Total Wins		9		1	

Table 4-10 Experiment #3(number of constraints vs. consistency checks)

4.2.4 Target Property: arity

number of variables=7, domain size=5, density =0.1, tightness of constraint=0.5					
Configuration#	arity	CPU runtime			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	2	3.1	7	14.2	53.2
2	3	2.9	3.6	54.2	128.1
3	4	4.7	5.1	204.5	429
4	5	8.4	7.2	588	1145.5
5	6	21.9	14.3	16232.8	21260
AVG	4.0	8.2	7.4	3418.7	4603.2
STDEV	1.6	8.0	4.1	7166.9	9321.4
Total Wins		5		0	

Table 4-11 Experiment #4(arity vs. CPU runtime)

number of variables=7, domain size=5, density =0.1, tightness of constraint=0.5					
Configuration#	arity	node checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	2	14.8	49.6	14.8	49.6
2	3	7.1	8.4	7.1	8.4

3	4	3.5	1.5	3.5	1.5
4	5	2.3	0.5	2.3	0.5
5	6	2.2	0.6	2.2	0.6
AVG	4.0	6.0	12.1	6.0	12.1
STDEV	1.6	5.3	21.2	5.3	21.2
Total Wins		0		0	

Table 4-12 Experiment #4(arity vs. node checks)

number of variables=7, domain size=5, density =0.1, tightness of constraint=0.5					
Configuration#	arity	consistency checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	2	114.1	275.5	1510.7	6344.5
2	3	1059.5	2910.1	19650.4	60248.9
3	4	1697.6	1749.9	57523.4	141205.5
4	5	4267.7	5001.9	191181.9	420071.5
5	6	12528.3	11115	3986688	9048845.1
AVG	4.0	3933.4	4210.5	851310.9	1935343.1
STDEV	1.6	5045.5	4228.7	1754300.3	3979758.2
Total Wins		5		0	

Table 4-13 Experiment #4(arity vs. consistency checks)

4.2.5 Target Property: tightness of constraint

number of variables=7, domain size =5 , density =0.1, arity=3			
Configuration#	tightness of constraint	CPU runtime	
		CDBT	CDDBT

		AVG	STDEV	AVG	STDEV
1	0.1	3.9	6.2	12.6	16.1
2	0.2	6.4	15.9	29.1	66.5
3	0.3	4	6.6	25.4	31.8
4	0.4	3.6	4.1	46.9	90.8
5	0.5	2.5	2.9	18.4	36.5
6	0.6	2.6	4.3	7.7	10.5
7	0.7	2.3	3.7	9.8	16.6
8	0.8	2.2	3.1	9.5	10.5
9	0.9	2.5	3.9	7.8	9.6
AVG	0.5	3.3	5.6	18.6	32.1
STDEV	0.3	1.3	4.1	13.2	28.7
Total Wins		10		0	

Table 4-14 Experiment #5(tightness of constraint vs. CPU runtime)

number of variables=7, domain size=5 , density =0.1, arity=3					
Configuration#	tightness of constraint	node checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	0.1	22.9	30.7	22.9	30.7
2	0.2	22.6	53.2	22.6	53.2
3	0.3	9	8	9	8
4	0.4	9.7	14.1	9.7	14.1
5	0.5	4.2	2.1	4.2	2.1
6	0.6	3.5	0.7	3.5	0.7
7	0.7	3.5	0.7	3.5	0.7
8	0.8	3.4	0.6	3.4	0.6
9	0.9	3.3	0.5	3.3	0.5

AVG	0.5	9.1	12.3	9.1	12.3
STDEV	0.3	8.1	18.3	8.1	18.3
Total Wins		0		0	

Table 4-15 Experiment #5(tightness of constraint vs. node checks)

number of variables=7, domain size=5 , density =0.1, arity=3					
Configuration#	tightness of constraint	consistency checks			
		CDBT		CDDBT	
		AVG	STDEV	AVG	STDEV
1	0.1	331.5	434.8	1502.1	2579.9
2	0.2	1102.4	2628.7	7956.5	21036.3
3	0.3	1237.4	3639	8176.4	12548.4
4	0.4	976.5	1599.1	17393.6	43733
5	0.5	379.4	499.2	4787.7	10798.7
6	0.6	290.1	177.3	1983.7	4645
7	0.7	307.2	166.5	3456.2	9351.6
8	0.8	293.9	104.4	1803.7	2399.3
9	0.9	299.3	179.4	1987.7	3167.8
AVG	0.5	579.7	1047.6	5449.7	12251.1
STDEV	0.3	400.5	1292.9	5168.0	13275.2
Total Wins		9		0	

Table 4-16 Experiment #5(tightness of constraint vs. consistency checks)

5 Results and Analysis

1. For random general CSPs, the constraint-directed mechanism cuts a lot of the search space. Both CDBT and CDDBT need only a few backtracks or even no backtracks. The search for a solution is pretty smooth. The constraint-directed mechanism avoids a lot of backtracks.

2. For random general CSPs, CDBT usually performs better than CDDBT.
   There are four reasons. First, both the constraint-directed mechanism and the Dynamic Backtracking mechanism cut the search space using different strategies. If we combine both, there is some redundant work. In CDDBT, we have to use a more complicated eliminating mechanism to guarantee their cooperation. Second, the constraint-directed mechanism reduces the number of backtracks, on the other hand, it weakens one strength of the Dynamic Backtracking mechanism, which is it can save a lot backtracks. Third, the random general CSP generator, by its nature, generates "fair" CSPs, which means, for example, all variables have the same opportunity to involve in all constraints. However, in cases such as Crossword Puzzles and Map-Coloring problems, some variables involve in fewer constraints and other variables involve more constraints. Fourth, most experiments are done under tightness=0.5, which means in each constraint half the tuples out of all possible tuples are valid. It is difficult for the random general CSP generator to generate a solvable CSP under low tightness.

3. For random general CSPs, the number of variables does not affect the performance of CDBT and CDDBT greatly. This is a good feature for the constraint-directed mechanism because the number of variables is the main

factor that affects the size of CSP. In the following figures, values along y-axis are the base-10 logarithms of corresponding values.

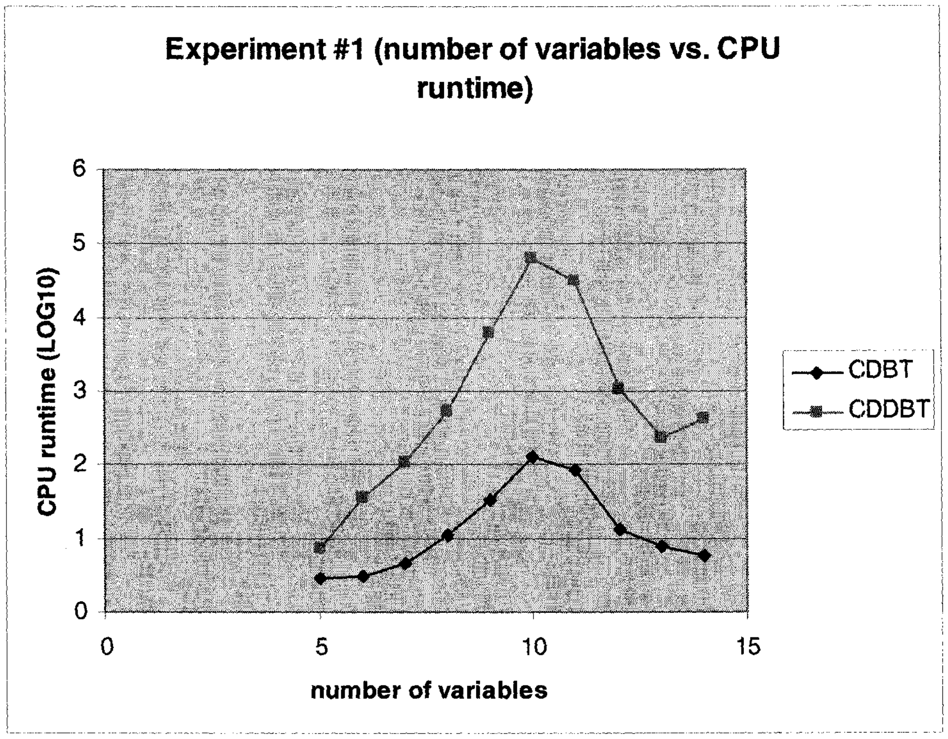


Figure 5-1 Experiment #1(number of variables vs. CPU runtime)

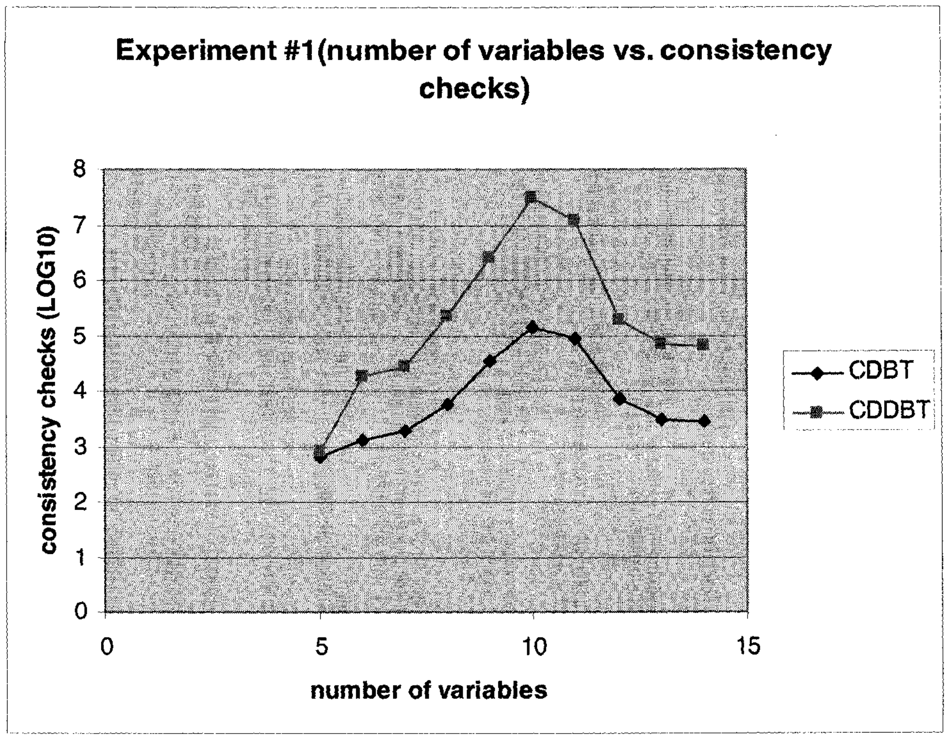


Figure 5-2 Experiment #1(number of variables vs. consistency checks)

4. For random general CSPs, with domain size, density, and arity increasing, CDBT performs well and is stable, but CDDBT performs poorly and is unstable. For example,

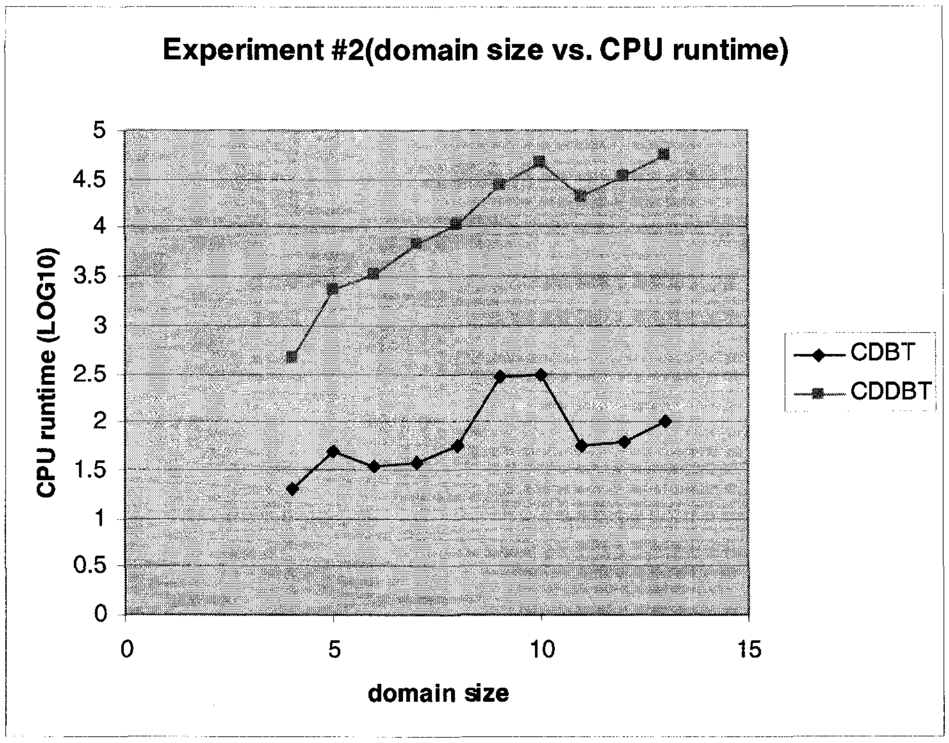


Figure 5-3 Experiment #2(domain size vs. CPU runtime)

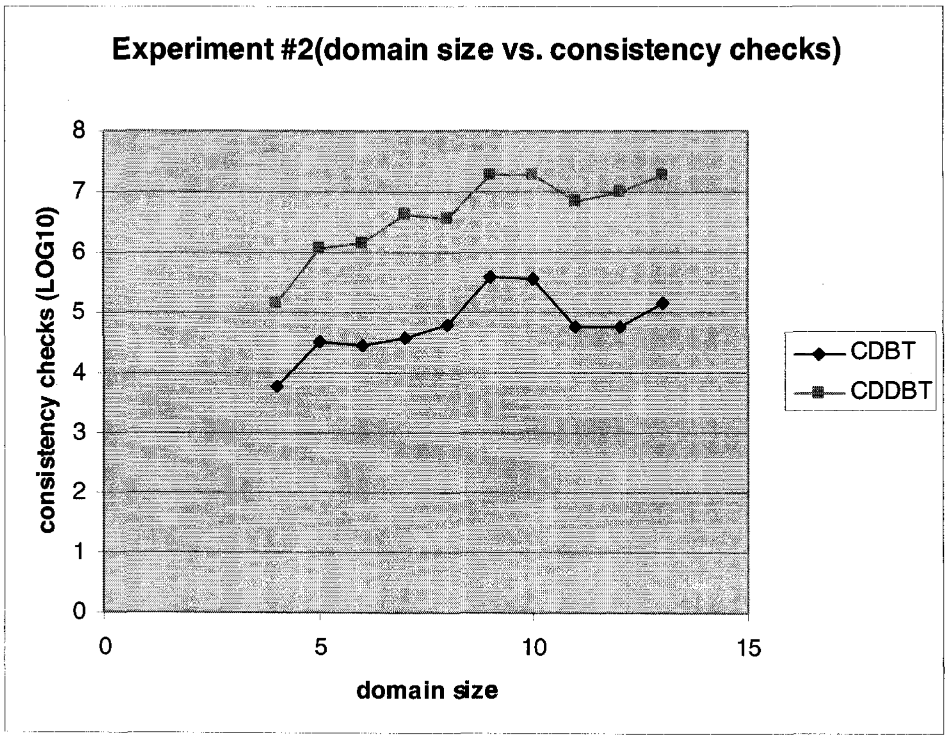


Figure 5-4 Experiment #2(domain size vs. consistency checks)

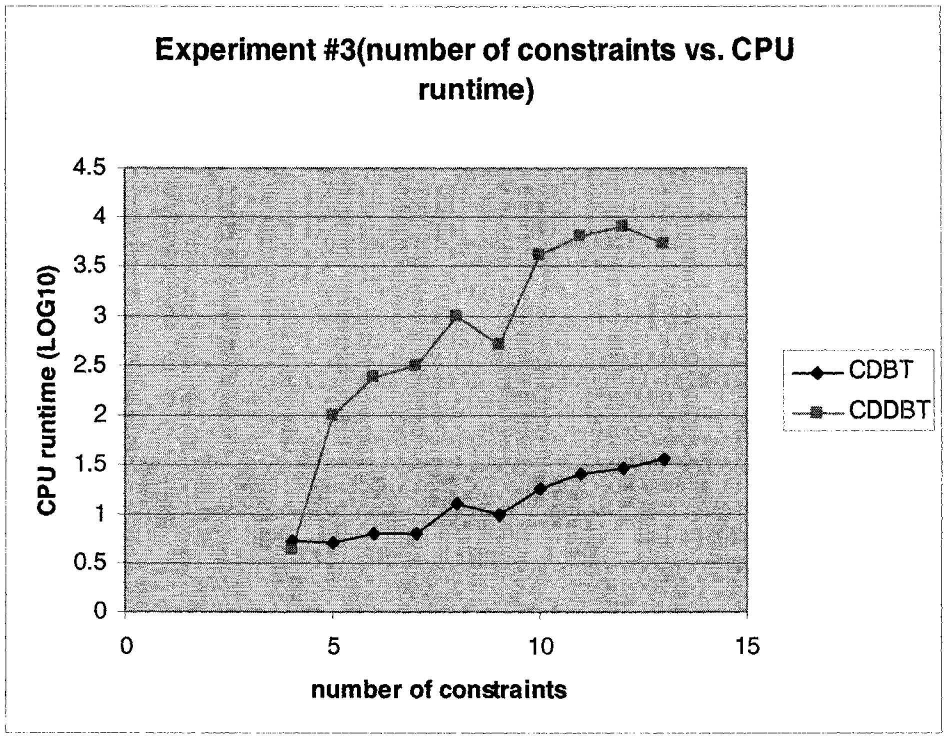


Figure 5-5 Experiment #3(number of constraints vs. CPU runtime)

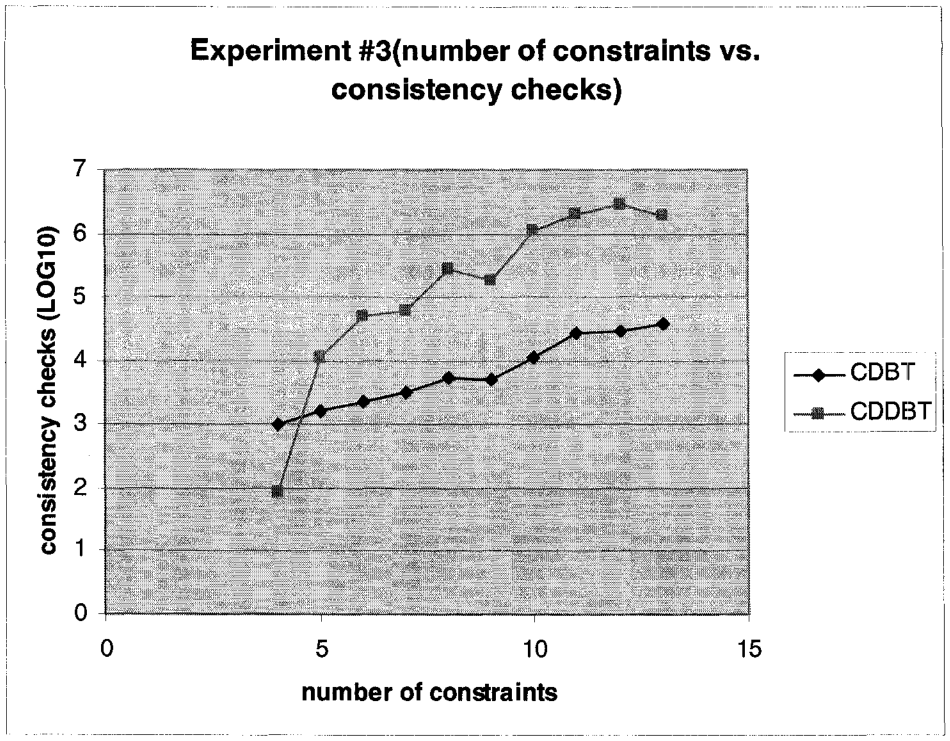


Figure 5-6 Experiment #3(number of constraints vs. consistency checks)

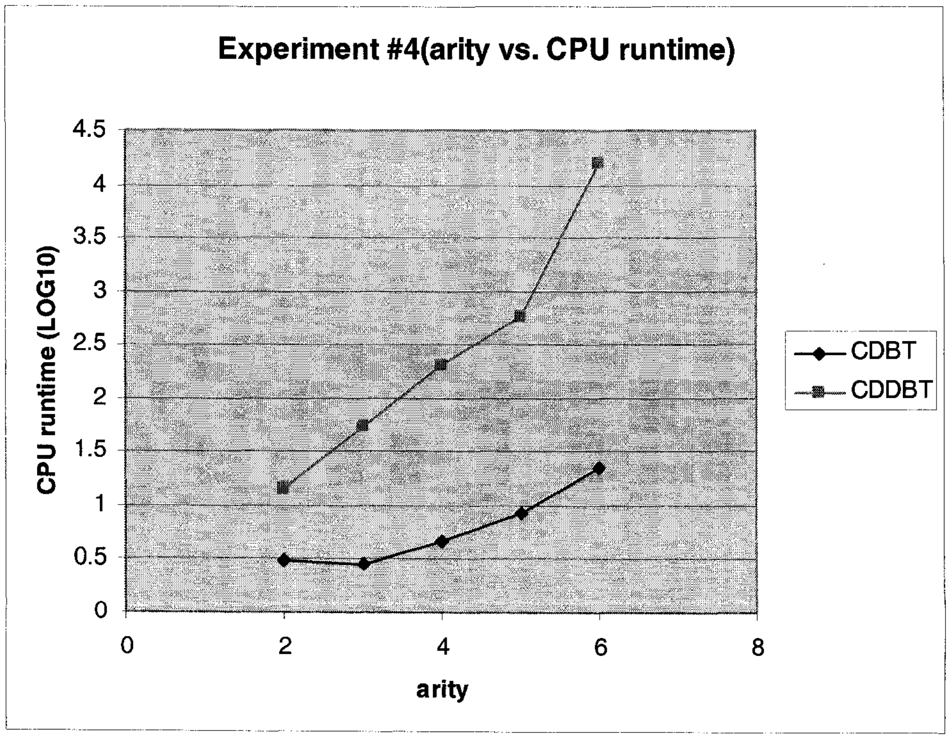


Figure 5-7 Experiment #4(arity vs. CPU runtime)

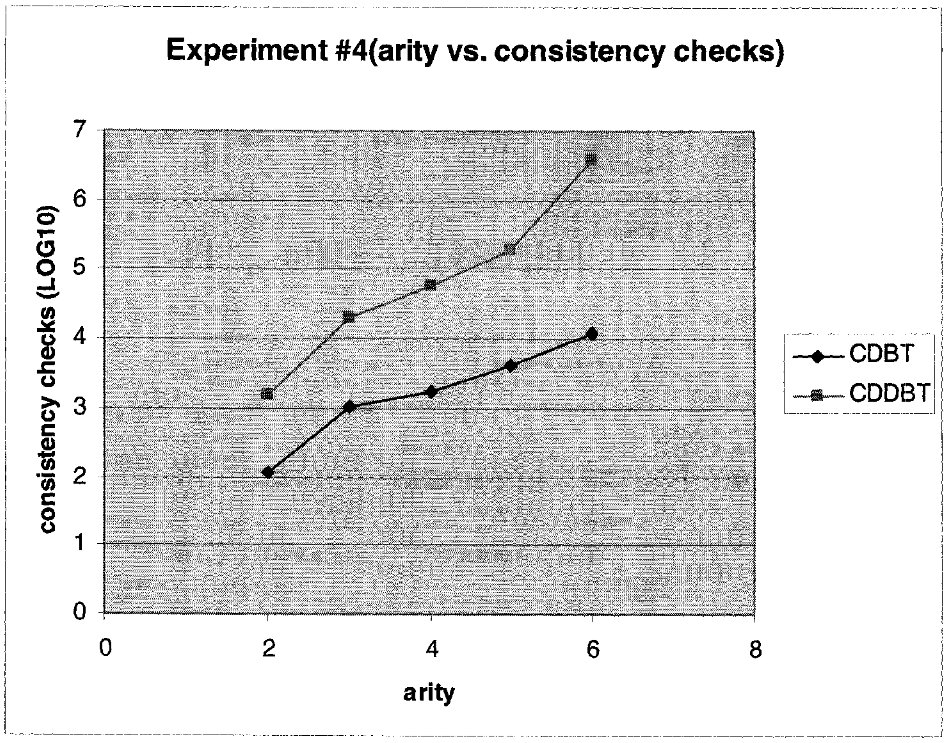


Figure 5-8 Experiment #4(arity vs. consistency checks)

5. For random general CSPs, the tightness of each constraint does not affect the performance of CDBT and CDDBT greatly.

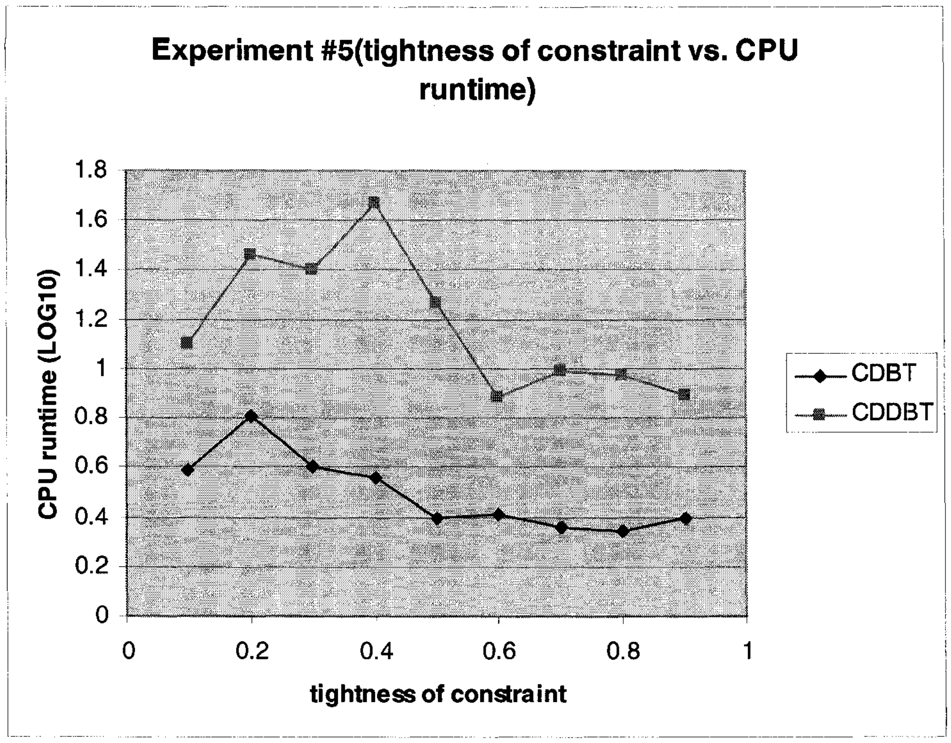


Figure 5-9 Experiment #5(tightness of constraint vs. CPU runtime)

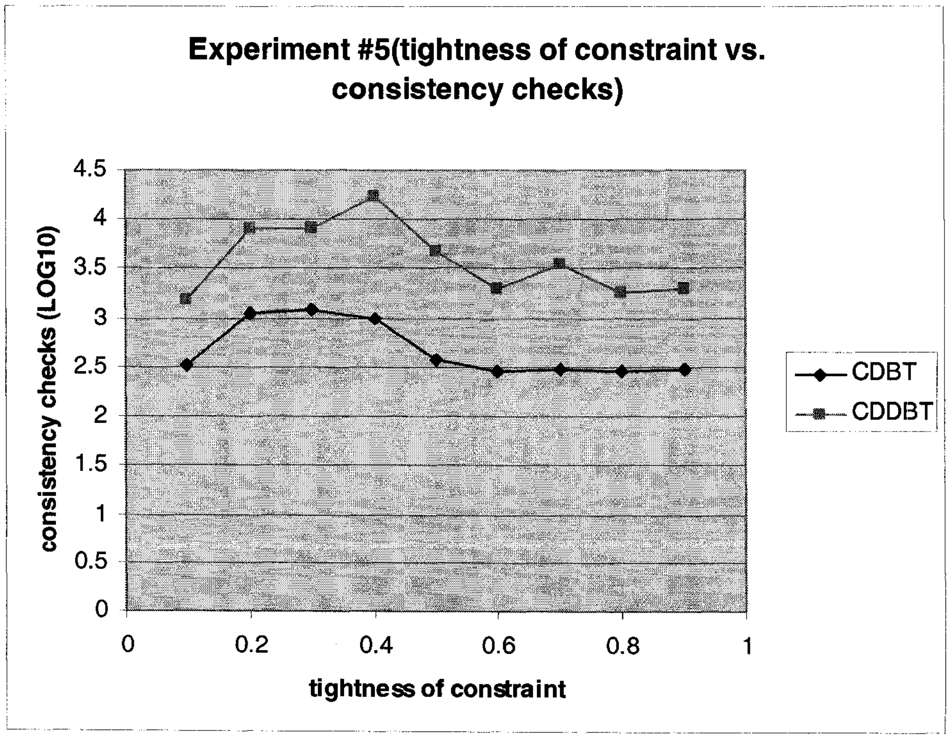


Figure 5-10 Experiment #5(tightness of constraint vs. consistency checks)

6. For random general CSPs, CDDBT performs better than CDBT in some cases.

number of variables=7, domain size=5 , density =0.1, arity=3, tightness of constraint=0.9		
CSP#	consistency checks	
	CDBT	CDDBT
1	259	1306
2	336	1100
3	263	124
4	171	2952
5	1055	7644
6	361	1078
7	112	55
8	251	80
9	211	210
10	238	71
11	523	11394
12	395	2441
13	497	3729
14	112	85
15	173	1734
16	151	3393
17	448	12655
18	377	1341
19	276	948
20	220	139
21	336	343
22	299	1784
23	340	1546

24	131	51
25	184	2258
26	224	14
27	336	444
28	157	148
29	337	302
30	206	263
AVG	299.3	1987.7
STDEV	179.4	3167.8
Total Wins	19	11

Table 5-1 detailed experiment data on configuration (number of variables=7, domain size=5, density=0.1, arity=3, tightness of constraint=0.9)

6 Conclusion and Future Work

In (Mackworth & Freuder, 1993), the authors state that it is important to identify tractable problem classes that are specific classes with tractable solution techniques. From research done on CSP-solving algorithms so far, there is no single best algorithm for CSP solving. Usually, one algorithm can beat another algorithm in some class of CSPs, and vice versa. This is also what we found for CDDBT and CDBT. For random general CSPs, CDBT usually performs better than CDDBT, but CDDBT performs better than CDBT in some cases. If the conditions under which CDDBT outperforms CDBT can be identified in the future, we could then choose to use it in those cases and use CDBT in the other cases. At present we only know that such cases exist and have not discovered a means to identify them.

Currently, most significant papers on CSPs appear in the journal of *Artificial Intelligence* and the journal of *Constraints*. The primary conference in this area is called the International Conference on Principles and Practice of Constraint Programming (CP). The International Joint Conference on Artificial Intelligence (IJCAI) and AAAI Conference on Artificial Intelligence are other important ones. From these sources, some but not all CSP research can be identified in following:

1. Consistency: (Li, 2006) and (de Givry, Heras, Zytnicki, & Larrosa, 2005)
2. Constraints in Bioinformatics: (Backofen & Will, 2006)
3. Distributed CSPs: (Zivan & Meisels, 2006) and (Hirayama & Yokoo, 2005)
4. Dynamic Backtracking: (Effinger & Williams, 2006) and (Zivan et al., 2006)
5. Multi-agent: (Liu, Jing, & Tang, 2002)
6. Non-binary CSPs: (Butaru & Habbas, 2005)
7. Quantified Constraint Satisfaction Problems: (Gent, Nightingale, & Stergiou, 2005)and (Gottlob, Greco, & Scarcello, 2005)
8. Symmetry: (Cohen, Jeavons, Jefferson, Petrie, & Smith, 2006) and (Puget, 2005)
9. Uncertainty Reasoning: (Tarim, Manandhar, & Walsh, 2006)

Dynamic Backtracking and general CSP are still two popular research areas in constraint research. We believe that CCDBT provides a new perspective for solving general CSPs. Some future works are:

1. Classify the CSPs that CDDBT has good performance.
2. Do experiments on real-world CSPs, for example, N-queens problem, Crossword Puzzles, scheduling problems, temporal reasoning, and graph-coloring problems. Constraint researchers usually do experiments either on random cases or practical cases; we choose the former in this thesis. One result can be anticipated is that it is very likely CDDBT would perform better than CDBT in the similar problems, where Dynamic Backtracking outperforms Backtracking such as graph-coloring problems.
3. Compare Dynamic Backtracking and CDDBT.
4. Do research on random general CSP generator.

Appendix: testing environment

Hardware:

Dell Dimension 5150

Base: Intel® Pentium® 4 Processor 521 w/HT Technology (2.8GHz,800FSB)

Memory: 1GB Dual Channel DDR2 SDRAM at 400MHz (4x256M)

Software:

Windows XP Professional SP2

JDK 6u1

Eclipse SDK 3.2.2.

Because there are many processes running simultaneously including XP itself, the CPU runtime value is slightly different every time we run an algorithm. Furthermore, because of the limitation of hardware and software condition, for example, CPU, memory, Java heap space, Java stack, and so on, the maximum number of constraints is set to 100 and the maximum number of allowed tuples of each constraint is set to 20000. Consequently, in following experiment data, if *density* does not match *number of constraints*, number of constraints is real; if *tightness* does not match *number of allowed tuples*, *number of allowed tuples* is real.

References

Achlioptas, D., Kirousis, L. M., Kranakis, E., Krizanc, D., Molloy, M. S. O., & Stamatiou, Y. C. (1997). Random constraint satisfaction: A more accurate picture. *Principles and Practice of Constraint Programming - CP97, Third International Conference,* Linz, Austria. 107-120.

Adorf, H. M., & Johnston, M. D. (1990). A discrete stochastic neural network algorithm for constraint satisfaction problems. *The International Joint Conference on Neural Networks,* San Diego, CA., *3*, 917-924.

Bacchus, F., Chen, X., van Beek, P., & Walsh, T. (2002). Binary vs. non-binary constraints. *Artificial Intelligence, 140*(1-2), 1-37.

Bacchus, F., & van Beek, P. (1998). On the conversion between non-binary and binary constraint satisfaction problems. *AAAI-98,* Madison, Wisconsin. 311-318.

Backofen, R., & Will, S. (2006). A constraint-based approach to fast and exact structure prediction in three-dimensional protein models. *Constraints, 11*(1), 5-30.

Bessiere, C. (1999). Non-binary constraints. *Principles and Practice of Constraint Programming (CP-99),* 24-27.

Bessiere, C., Meseguer, P., Freuder, E. C., & Larrosa, J. (1999). On forward checking for non-binary constraint satisfaction. *Principles and Practice of Constraint Programming (CP-99),* New York. 88-102.

Bitner, J. R., & Reingold, E. M. (1975). Backtrack programming techniques. *Communications of the ACM, 18*(11), 651-656.

Bluman, A. G. (2001). *Elementary statistics: A step by step approach* (4th ed.). Boston: McGraw-Hill.

Butaru, M., & Habbas, Z. (2005). Solving the car-sequencing problem as a non-binary CSP. *Principles and Practice of Constraint Programming - CP 2005,* Sitges, Spain. 840.

Cohen, D. A., Jeavons, P., Jefferson, C., Petrie, K. E., & Smith, B. M. (2006). Symmetry definitions for constraint satisfaction problems. *Constraints, 11*(2-3), 115-137.

Davenport, A., Tsang, E. P. K., Zhu, K., & Wang, C. J. (1994). GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. *AAAI,* Seattle, WA, USA. 325-330.

de Givry, S., Heras, F., Zytnicki, M., & Larrosa, J. (2005). Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. *International Joint Conference on Artificial Intelligence (IJCAI),* Edinburgh, Scotland, UK. 84-89.

Dechter, R., & Pearl, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence, 38*, 353-366.

Effinger, R. T., & Williams, B. C. (2006). Extending dynamic backtracking to solve weighted conditional CSPs. *The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference,* Boston, Massachusetts, USA.

Freuder, E. C. (1976). *Synthesizing constraint expressions.* M.I.T., Cambridge, Mass: A. I. Memo 370.

Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM, 29*(1), 24-32.

Freuder, F., Dechter, R., Ginsberg, M., Selman, B., & Tsang, E. (1995). Systematic versus stochastic constraint satisfaction. *IJCAI,* Montreal, Canada. 2027-2032.

Gaschnig, J. (1979). Performance measurement and analysis of certain search algorithms. (Ph.D. Diss., Carnegie-Mellon University).

Gent, I. P., MacIntyre, E., Prosser, P., Smith, B. M., & Walsh, T. (2001). Random constraint satisfaction: Flaws and structure. *Constraints, 6*(4), 345-372.

Gent, I. P., Nightingale, P., & Stergiou, K. (2005). QCSP-solve: A solver for quantified constraint satisfaction problems. 138-143.

Ginsberg, M. L. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research (JAIR), 1,* 25- 46.

Ginsberg, M. L., & McAllester, D. A. (1994). GSAT and dynamic backtracking. *The 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94),* Bonn, Germany. 226-237.

Gomes, C. P., & Selman, B. (1997). Problem structure in the presence of perturbations. *AAAI-97,* Providence, RI. 221-226.

Gottlob, G., Greco, G., & Scarcello, F. (2005). The complexity of quantified constraint satisfaction problems under structural restrictions. *IJCAI 2005,* Edinburgh, Scotland, UK. 150-155.

Hirayama, K., & Yokoo, M. (2005). The distributed breakout algorithms. *Artificial Intelligence, 161*(1-2), 89-115.

Johnston, M. D., & Adorf, H. M. (1989). Learning in stochastic neural networks for constraint satisfaction problems. *NASA Conference on Space Telerobotics,* Pasadena, CA., *2* 367-376.

Jonsson, A. K., & Ginsberg, M. L. (1993). Experimenting with new systematic and nonsystematic search techniques. *The AAAI Spring Symposium on AI and*

NP-Hard Problems, Stanford, California.

Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey. *AI Magazine, 13*(1), 32-44.

Li, S. (2006). On topological consistency and realization. *Constraints, 11*(1), 31-51.

Liu, J., Jing, H., & Tang, Y. Y. (2002). Multi-agent oriented constraint satisfaction. *Artificial Intelligence, 136*(1), 101-144.

Mackworth, A. K. (1977a). Consistency in networks of relations. *Artificial Intelligence, 8*(1), 99-118.

Mackworth, A. K. (1977b). On reading sketch maps. *The Fifth International Joint Conference on Artificial Intelligence,* Cambridge, Mass. 598-606.

Mackworth, A. K., & Freuder, E. C. (1993). The complexity of constraint satisfaction revisited. *Artificial Intelligence, 59*, 57-62.

Minton, S., Johnston, M. D., Philips, A. B., & Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence, 58*(1-3), 161-205.

Minton, S., Johnston, M. D., Philips, A. B., & Laird, P. (1990). Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method. *The 8th National Conference on Artificial Intelligence (AAAI 1990),* Boston, Mass. 17-24.

Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information Science, 7*(2), 95-132.

Nadel, B. A. (1988). Tree search and arc consistency in constraint satisfaction algorithms. In L. Kanal, & V. Kumar (Eds.), *Search in artificial intelligence* (1st ed., pp. 287-342). New York: Springer-Verlag.

Nadel, B. A. (1989). Constraint satisfaction algorithms. *Computational Intelligence, 5,* 188-224.

Pang, W. (1998). Constraint structure in constraint satisfaction problems. (PhD thesis, University of Regina, Canada).

Pang, W., & Goodwin, S. D. (1996). Application of CSP techniques to scheduling problems. *The 2Nd International Symposium on Operations Research and its Applications (ISORA '96),* Gulin, China.

Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Puget, J. (2005). Symmetry breaking revisited. *Constraints, 10*(1), 23-46.

Rossi, F., Petrie, C., & Dhar, V. (1990). On the equivalence of constraint satisfaction problems. *9th European Conference on Artificial Intelligence (ECAI '90),* Stockholm, Sweden. 550-556.

Russell, S. J., & Norvig, P. (2003). *Artifical intelligence: A modern approach* (2nd ed.). Englewood Cliffs, NJ: Prentice Hall.

Selman, B., Levesque, H. J., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. *AAAI 1992,* San Jose, CA. 440-446.

Stallman, R. M., & Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence, 9*(2), 135-196.

Stergiou, K. (2001). Representation and reasoning with non-binary constraints. (PhD thesis, University of Strathclyde).

Stergiou, K., & Walsh, T. (2006). Inverse consistencies for non-binary constraints. *ECAI 2006,* Riva del Garda, Italy. 153-157.

Stone, H., & Stone, J. (1987). Efficient search techniques - an empirical study of the n-queens problem. *IBM Journal of Research and Development, 31*, 464-474.

Tarim, S., Manandhar, S., & Walsh, T. (2006). Stochastic constraint programming: A scenario-based approach. *Constraints, 11*(1), 53-80.

Tsang, E. (1993). *Foundations of constraint satisfaction.* San Diego: Academic Press.

Van Hentenryck, P. (1989). *Constraint satisfaction in logic programming* (1st ed.). Cambridge, MA: MIT Press.

Waltz, D. (1975). Understanding line drawings of scenes with shadows. In P. H. Winston (Ed.), *The psychology of computer vision* (1st ed.,). New York: McGraw-Hill.

Zivan, R., & Meisels, A. (2006). Dynamic ordering for asynchronous backtracking on DisCSPs. *Constraints, 11*(2-3), 179-197.

Zivan, R., Shapen, U., Zazone, M., & Meisels, A. (2006). Retroactive ordering for dynamic backtracking. *Principles and Practice of Constraint Programming - CP 2006,* Nantes, France. 766-771.

Vita Auctoris

Name:	Kan Yu
PLACE OF BIRTH:	Shanghai, China
YEAR OF BIRTH:	1974
EDUCTION:	Qibao High school, Shanghai, China
	1990-1993
	Shanghai University, Shanghai, China
	1993-1997 BEng
	University of Windsor, Windsor, Ontario, Canada
	2004-2007 M.Sc.