INDSET: A DECOMPOSITION TECHNIQUE FOR CSPS USING MAXIMAL INDEPENDENT SETS AND ITS INTEGRATION WITH LOCAL SEARCH

by

Joel Gompert

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

May, 2005

INDSET: A DECOMPOSITION TECHNIQUE FOR CSPS USING MAXIMAL INDEPENDENT SETS AND ITS INTEGRATION WITH LOCAL SEARCH

Joel Gompert, M.S.

University of Nebraska, 2005

Advisor: Berthe Y. Choueiry

We introduce INDSET, a technique for decomposing a Constraint Satisfaction Problem (CSP) by identifying a maximal independent set in the constraint graph of the CSP. We demonstrate empirically that INDSET reduces the complexity of solving the CSP, and yields compact and robust solutions. We discuss how to integrate this decomposition technique with stochastic local search (SLS), and evaluate SLS+INDSET, which combines the two. We discuss the benefit of identifying dangling components of the decomposed constraint graph, and evaluate SLS+INDSET+DANGLES, a strategy that exploits this structural improvement. We explore the possibility of applying INDSET recursively. We explore the capabilities and limitations of INDSET, and provide insights on the combination of INDSET with backtrack search and the detection of local interchangeability.

ACKNOWLEDGEMENTS

I express gratitude to my advisor, Dr. Berthe Y. Choueiry, and the rest of my committee, Dr. Matthew Dwyer, Dr. Steve Goddard, and Dr. Sylvia Wiegand for their time and insightful comments.
I also thank Professor Stephen D. Kachman in the Department of Biometry and Mr. Bradford Danner in the Department of Statistics for their guidance. Thanks to Dr. Jeremy Frank of NASA Ames Research Center for the suggestion of static bundling on I.
I also thank my colleagues: Mr. Anagh Lal, Mr. Ryan Lim, Mrs. Yaling Zheng, Ms. Cate Anderson, Mr. Christopher Bourke, Mr. Praveen Guddeti, and Mr. Andrew Breiner.
I also give a special thanks to my parents for their encouragement.

This research is supported by NSF CAREER Award #0133568. Experiments were conducted on the Research Computing Facility of the University of Nebraska-Lincoln.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Solving techniques for difficult problems can often be improved by identifying types of structure that exists in the problem. This structure often allows us to decompose the problem into smaller pieces in such a way that the problem becomes easier to solve. We present a new conjunctive decomposition technique INDSET that exploits the structure of a binary Constraint Satisfaction Problem (CSP) to boost performance of solving the CSP while yielding multiple solutions. A CSP has a set of variables, and each variable has a domain of values. A binary CSP also has binary constraints limiting which combinations of values are allowed. A binary constraint is an arbitrary relation on the domains of any two variables. The problem is how to assign a value to every variable in such a way that all of the constraints are satisfied. This problem is, in general, in **NP**-complete. A binary CSP can be represented as a *constraint graph*, where each variable is a vertex, and two vertices are adjacent (i.e., have an edge connecting them) if and only if there is a constraint on the two variables.

INDSET, which first appeared in [Gompert, 2004, Gompert and Choueiry, 2005], is based on identifying, in the constraint graph of the CSP, a *maximal independent set* I, which is a set of variables that are pairwise non-adjacent (see Figure 1.1). A graph with

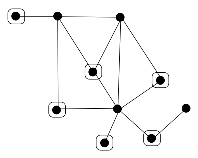

Figure 1.1: Circled vertices form a maximal independent set.

a low number of edges is likely to contain a large independent set. Therefore, INDSET is particularly suited to these types of problems.

1.1 Motivation

Two important factors in solving a CSP are the amount of time that is required to solve the problem and the flexibility of the solution(s) found. Flexibility is desired when a problem has multiple solutions and the user would like some ability to choose between multiple possible solutions, rather than be constrained to one particular solution. Our technique, INDSET, which is used as an enhancement to existing algorithms for solving CSPs, provides improvement in both of these factors, by reducing the time to solve the problem and by returning multiple solutions.

Other factors to consider for solving a CSP are the space required by the solving technique, and the quality of solutions returned. The measurement of the quality of solutions depends on the particular application, so we cannot specify that here. As for the space required, using INDSET with any solving technique does not increase the space complexity.

INDSET partitions the variables of a CSP into two sets, I and its complement $\bar{I}$, and restricts the search to the variables in $\bar{I}$ in order to find a solution to the CSP induced by $\bar{I}$. We then extend the solution found (in $\bar{I}$) to the variables in I by applying directional arc-consistency (DAC, defined in Section 2.2) between the variables in $\bar{I}$ and those in I. DAC can be computed in linear-time in the number of variables in I and the number of constraints between I and $\bar{I}$. When arc-consistency succeeds, it yields multiple solutions to I consistent with the solution found on $\bar{I}$, thus we have multiple solutions to the CSP. The process yields a family of solutions for each of the values remaining in the domain of a variable in I. This property is particularly useful because most search techniques return only one solution. The only solving technique of which we are aware that returns multiple solutions is the dynamic bundling technique of [Choueiry *et al.*, 1995]. We are not aware of any decomposition technique other than INDSET that has been suggested to be used to find multiple solutions simultaneously. Using INDSET returns multiple solutions simultaneously, in addition to being less time consuming. Obtaining multiple solutions makes the results of this approach more robust, by providing flexibility to the user in choosing assignments to the variables in I. INDSET is described in detail in Chapter 3.

While *any* search technique can be used to solve the variables in $\bar{I}$, we have developed and tested a method, SLS/INDSET, for using this approach in combination with stochastic local-search (SLS) with steepest descent. SLS/INDSET integrates information about the constraints between $\bar{I}$ and I in order to find solutions for $\bar{I}$ that can be extended to I. We found that INDSET significantly improves the performance of SLS, and yields robust results by finding multiple solutions and returning them in a compact form. Section 2.4 describes SLS in further detail, and Chapter 4 explains SLS/INDSET. In Section 5.1 we introduce an additional structural technique for finding dangling trees, which further improves INDSET, and in Section 5.2 we recall the definition of neighborhood interchangeability and examine its use to further improve solving techniques in combination with INDSET.

1.2 Contributions

The main contributions of this thesis are as follows:

- We present the new decomposition technique INDSET.

- We describe one method to incorporate INDSET into local-search techniques, and discuss several heuristics that can be used in this combination.

- We demonstrate that the combination of INDSET with local search yields an improvement in the time required to solve a problem, and yields multiple solutions.
- We describe how to identify and utilize trees dangling off the graph of the CSP.
- We report preliminary evidence that neighborhood interchangeability techniques can enhance our search techniques.
- We analyze the INDSET decomposition structure and identify additional information that can be gained from knowledge of the structure, such as being able to compute a lower bound on the number of solutions of the CSP.
- We analyze the technique of applying INDSET recursively on the constraint graph of a CSP, and explain its potential and drawbacks.

1.3 Outline of the thesis

This thesis is organized as follows. Section 2 reviews background information and related work. Chapter 3 describes our basic decomposition technique, INDSET and highlights its benefits. Chapter 4 discusses how to exploit INDSET in local search. Chapter 5 points out some further enhancements that can be made with INDSET, such as finding ‘dangling’ trees in the constraint graph and applying neighborhood interchangeability, as well some other uses for independent sets, such as finding cycle-cutsets. Chapter 6 analyzes the features of INDSET and discusses the additional information gained from this approach, such as computing a lower bound on the number of solutions in the CSP, comparing INDSET to CYCLE-CUTSET, analyzing runtime variance, and performing experiments on clustered constraint graphs. Chapter 7 presents a recursive independent-set decomposition (RECINDSET) and describes the results of its use to obtain a variable ordering for systematic search.

Chapter 8 summarizes our contributions and results and identifies directions for future research. The appendices provide documentation for the source code that we developed for the experiments in this thesis.

Chapter 2

Background and Related Work

This chapter provides the background for the rest of the thesis and identifies some related prior work. We recall the definition of a Constraint Satisfaction Problem (CSP), Arc Consistency (AC), Directional Arc Consistency (DAC), phase transition, robust solutions, and independent sets. We explain the concept of local search, describe the generation of random problem instances. We give a brief overview of the concept of CSP decomposition and list some additional related work.

2.1 The Constraint Satisfaction Problem (CSP)

A Constraint Satisfaction Problem (CSP) is a tuple $\mathcal{P} = \{\mathcal{V}, \mathcal{D}, \mathcal{C}\}$, where $\mathcal{V}$ = {V_1, V_2, …, V_N} is a set of N variables, $\mathcal{D}$ = {$D_{V_1}, D_{V_2}, \ldots, D_{V_N}$} is a set of domains for these variables (a domain D_{V_i} is a set of values for the variable V_i). In this thesis we work only with finite, discrete domains. $\mathcal{C}$ is a set of relations on subsets of $\mathcal{D}$, which specify the allowed combinations of values for variables. Each row of a relation (i.e., an assignment of values to the variables of the constraint) is called an allowed *tuple*.

Solving a CSP requires assigning to each variable V_i a value chosen from D_{V_i} such that all constraints are satisfied. We denote by $\mathcal{P}_X$ the CSP induced on $\mathcal{P}$ by a set of variables

$X \subseteq \mathcal{V}$. In this thesis we focus on *binary* CSPs: each constraint is a relation on at most two variables. Every CSP has an equivalent binary CSP. Converting a non-binary CSP to its binary equivalent is straightforward–we replace a constraint involving k variables by a clique on those variables. The resulting graph is called the *primal graph*.

An alternative graph representation has each variable-value pair as a vertex. Edges either represent pairs of values that the constraints allow, or pairs that the constraints disallow. This graph is called the *microstructure* of the CSP.

- The *tightness* t of a constraint is the ratio of the number of tuples forbidden by the constraint to the number of all possible tuples. We performed experiments using tightness 10%, 30%, and 50%. When we were interested in results at higher tightnesses, we experimented with 58% (a result of following experiments of [Dechter, 2003]) and/or 60%. Tighter random problems are unlikely to be solveable.

- The *constraint ratio* r is the ratio of the number of constraints e in the CSP to the number of possible constraints in the CSP $r = \frac{e}{N*(N-1)/2}$. We assume that there is at most one constraint for each pair of variables.

- The *constraint graph* of a CSP is a graph G where each vertex in G represents a variable in the CSP, and each (binary) constraint in the CSP is represented by an edge in G, connecting the two corresponding vertices.

- A *neighbor* of a vertex (variable) V_i is any vertex that is adjacent to V_i (i.e., shares a constraint with V_i). The set of all such variables is the *neighborhood* of V_i.

Systematic search is a category of complete solving techniques, where, one at a time, variables are assigned values that are consistent with the constraints, backtracking if any conflicts occur, until a solution is found or all combinations of assignments have been eliminated. At any point in the search, variables already assigned values are considered *past* variables, and variables not yet assigned values are considered *future* variables. *Forward checking* is a 'lookahead strategy' that can be used during systematic search where, each time a variable V_i is assigned a value, the domains of all of the future variables in the neighborhood of V_i are reduced, in order to eliminate all values that conflict with the value assigned to V_i. Forward checking provides a significant performance increase.

2.2 Arc Consistency and Directional Arc Consistency

Arc Consistency (AC) is a technique that removes inconsistent values from the domains of the variables. A binary constraint on variables V_i and V_j is arc consistent if and only if, for every value a in D_{V_i}, there exists a value b in D_{V_j} such that the tuple (a, b) is allowed by the constraint. A CSP is arc consistent if and only if every constraint is arc consistent. A procedure called REVISE(V_i, V_j) is useful for enforcing arc consistency. REVISE(V_i, V_j) deletes from the domain of V_i any value that does not have a corresponding consistent value in V_j. A constraint can be made arc consistent by executing REVISE(V_i, V_j) and REVISE(V_j, V_i) alternately until no more changes are made to the domains. The CSP is arc consistent when no domains are changed by applying REVISE(V_i, V_j) to any pair of variables. Various algorithms have been proposed, and are commonly used, for enforcing AC efficiently.

Directional Arc Consistency (DAC), a weaker form of AC, takes into account a strict ordering $<$ of the variables. REVISE(V_i, V_j) is executed only if $V_i < V_j$ in the ordering. In DAC, it is not necessary to execute REVISE(V_i, V_j) multiple times on each constraint. Instead, we can proceed from the largest variable (according to the ordering) to the smallest, executing REVISE(V_i, V_j) once on each constraint. DAC is a necessary, but not sufficient, condition for AC. DAC is more efficient to compute than AC.

An assignment to a CSP is a solution if and only if it is also a solution to the resulting problem after enforcing AC or DAC. Thus the resulting problem can be considered equivalent to the original, but can be solved more efficiently. It may be the case that AC or DAC annihilates the domain of a variable, which is sufficient (but not necessary) to let us know that the original has no solutions.

2.3 Phase transition

Cheeseman et al. [1991] presented empirical evidence for the existence of a phase transition phenomenon at a critical value of certain parameters for some combinatorial problems. They showed a significant increase in the cost of solving these problems around the critical value. Figure 2.1 illustrates this situation. In the case of CSPs, tightness and constraint

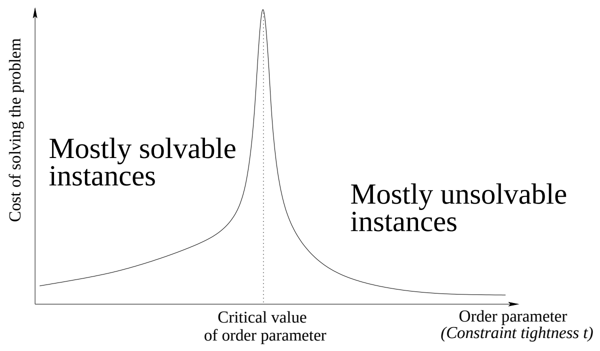

Figure 2.1: Cost of problem solving.

ratio are often used as order parameters. Also, in CSPs, the phase transition is often associated with the transition between problems being solvable and being unsolvable. For example, problems with low constraint ratio are likely to be solvable and problems with high constraint ratio are likely to be unsolvable. The phase transition usually corresponds to the same value of constraint ratio where the majority of problems transition from being solvable to being unsolvable. Problems near this transition point are usually the most difficult to solve. For this reason, experiments comparing algorithms are usually conducted in this area.

2.4 Local search

Local search refers to search algorithms that start with a complete random assignment of values to the variables and make incremental changes to it in an attempt to reach a solution. These algorithms are not guaranteed to find a solution if one exists, however they are useful because they can sometimes find solutions in large problems faster than a systematic search. The basic local search technique is to make greedy incremental changes that yield the largest improvement to the current assignment according to some evaluation function. Each such *move* yields an assignment to the variables with a better evaluation than the previous assignment. This process is analogous to climbing a hill. A *global optimum* of the evaluation function yields a solution to the CSP. As we climb a hill we may reach a *local optimum* that may not be a solution to the CSP, yet we cannot climb any higher from that point, and, as a result, the local search technique is said to be trapped. Many techniques for recovering from, or avoiding, local optima have been proposed. One of the simplest ways to deal with encountering a local optimum is a *random restart*, i.e. to restart the local search process with a new, random assignment. One common evaluation function for CSPs is the number of constraints broken (i.e., violated) by the assignment. At each iteration, an incremental change is made that most reduces the number of *broken* constraints (i.e., constraints that are violated by the current assignment).

We consider Algorithm 1, the the local search algorithm SLS as described in [Dechter, 2003]. This algorithm starts with a random initial assignment to the variables, and then makes incremental changes until it finds a consistent assignment. At each step, SLS evaluates for each variable-value pair the number of broken constraints resulting from changing the assignment of the variable to the value in the pair. SLS chooses the pair that yields the minimum number of broken constraints, breaking ties lexicographically. This heuristic is called ‘steepest descent’ [Galinier and Hao, 1997]. This atomic step is repeated until either

Algorithm 1 SLS.

```
for $i = 1$ to MAX_TRIES do
  Make a random assignment to all variables.
  Count $B =$ broken constraints
  repeat
    if $B = 0$ then
      Return current assignment as a solution
    end if
    for each variable-value pair $< x_i, a_i >$ not in the current assignment do
      Suppose we swap the pair for the current assignment of $x_i$
      Compute $B$ for this assignment.
    end for
    Choose variable-value pair $< x_i, a_i >$ that minimizes $B$.
    $x_i \leftarrow a_i$
  until no further changes occur
end for
```

a solution is found or no improvement is possible, in which case the search is restarted from a new random assignment. The process is repeated for a specified number of restarts or a given time duration unless a solution is found. We call each iterative improvement a *move*. We denote by RUN or TRIAL the process of running a search algorithm until it either finds a solution or is terminated by a cutoff criterion.

Chapter 4 describes in greater detail how local search can be focused on the variables in $\bar{I}$ of an independent-set decomposition.

2.5 Random problems

Random problems are often used to empirically evaluate new CSP techniques. Typically these problems are generated with a specified number of variables and constraint ratio, the same domain size for each variable, and the same tightness for every constraint. There are several models for generating such problem instances, of which we primarily use the common *Model B* [Achlioptas *et al.*, 1997].

The above models distribute the constraints among the variables with a uniform random distribution. A different technique for distributing the constraints yields *clustered graphs* [Hogg, 1996]. In this technique, variables are assigned to leaves of a balanced binary tree. Two variables that are connected at a deeper depth are more likely to have a constraint generated between them than two variables that are connected higher in the tree. The probability of generating a constraint decreases exponentially with this depth of the tree. We evaluate the performance of INDSET on both Model B and clustered graphs.

We conduct our experiments on *connected* problems only. If the problem generation procedure produces a problem with a disconnected constraint graph, then we discard the problem. Generating sparse problems is difficult, since there is a high probability of a random graph being disconnected.

2.6 Independent sets

An *independent set* in a graph $G = (V, E)$ is a set of vertices $I \subseteq V$ such that the subgraph induced by I has no edges (i.e., a set of pairwise non-adjacent vertices). See Figure 1.1. A *vertex cover* is a set of vertices such that every edge in the graph is incident to a vertex in the set. The complement of an independent set is always a vertex cover. A *maximal* independent set is one that is not a subset of any larger independent set. A maximal independent set is to be distinguished from a *maximum* independent set, which is a largest independent set in the graph. In this thesis we use the CLIQUEREMOVAL algorithm, which runs in polynomial time in the number of variables [Boppana and Halldórsson, 1990].

2.7 Decomposition

Decomposition is a natural strategy for reducing the cost of solving a CSP. Freuder and Hubbe introduced two general schemas for decomposing CSPs: *conjunctive* and *disjunc-*

tive decomposition [1995a]. In a conjunctive decomposition schema, a solution to the problem is obtained from 'putting' together solutions of the subproblems (e.g., using articulation points [Freuder, 1982] and cycle-cutset method [Dechter and Pearl, 1987]). In a disjunctive decomposition schema, a solution to any subproblem is a solution to the initial problem (e.g., Inferred Disjunctive Constraint [Freuder, 1993], and Factor Out Failure [Freuder and Hubbe, 1995b]). INDSET is a conjunctive decomposition that partitions the CSP variables with an independent set (see Figure 3.1).

2.7.1 Backdoor variables

INDSET, first reported in [Gompert, 2004], may be considered as one of the techniques that exploit 'strong backdoors' [Williams *et al.*, 2003]. These are techniques that divide a problem into two sets of variables, search is performed on one of the sets (i.e., the backdoor), and the resulting partial solution can be expanded to a full solution (or be shown to be inconsistent) in polynomial time. In our case, the complement $\bar{I} = \mathcal{V} \setminus I$ of our independent set I forms a 'strong backdoor.' Indeed, any instantiation of $\bar{I}$ leads to a linear-time solvable sub-problem $\mathcal{P}_I$, because $\mathcal{P}_I$ has no edges.

2.7.2 CYCLE-CUTSET

Another example of a backdoor is a cycle-cutset. In the cycle-cutset decomposition technique, CYCLE-CUTSET, one chooses a set A of variables such that removing A from the constraint graph leaves the graph acyclic [Dechter and Pearl, 1987, Dechter, 2003]. An example is shown in Figure 2.2. Thus, given any assignment for A, the remaining tree can be solved, or be proved unsolvable, in linear time with backtrack-free search, assuming the CSP is arc-consistent [Freuder, 1982]. Like INDSET, CYCLE-CUTSET is also more beneficial when the constraint graph has a low constraint ratio, because finding a smaller

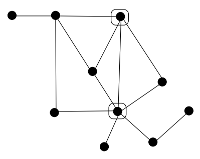

Figure 2.2: Circled vertices form a cycle-cutset.

cycle-cutset increases the benefit of the technique, and small cycle-cutsets are less likely to exist in dense graphs, just as large independent sets are less likely to exist in dense graphs. We compare and contrast CYCLE-CUTSET and INDSET in further detail in Section 6.2.

2.7.3 Other related work

A related work, carried out in the context of SAT, partitions the set of Boolean variables in a SAT problem into 'independent' and 'dependent' variables and exploits this distinction in local search [Kautz *et al.*, 1997]. The technique is heavily dependent on SAT encodings and its application to CSPs is not straightforward.

Finally, except for [Choueiry *et al.*, 1995], none of the decomposition techniques reported in the literature discuss the 'production' of multiple solutions, a by-product of our technique. We argue that this feature of INDSET, shared to a lesser extent by CYCLE-CUTSET, sets our approach apart from the rest.

2.8 Robust solutions

Using INDSET in combination with a search technique yields robust solutions, in the sense that the solutions provide flexibility to the user. Figure 2.3 shows an example CSP. Table 2.1 shows an example of a single solution to the CSP in the left column, where each variable is assigned a single value. In contrast, the robust solution assigns a set of values to each variable. In the context of INDSET, a *robust solution* is an assignment of a set of values to each variable, such that every element of the Cartesian product of the sets of values is a

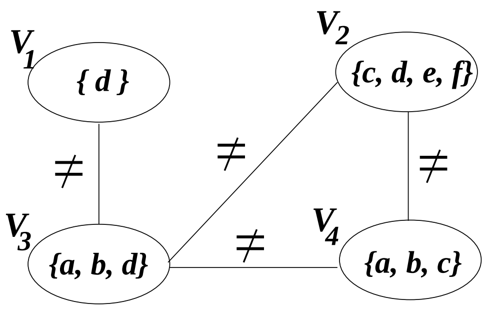

Figure 2.3: Example CSP.

Table 2.1: Single solution vs robust solution.

Single solution	**Robust solution**
V_1: d	V_1: $\{d\}$
V_2: e	V_2: $\{d, e, f\}$
V_3: a	V_3: $\{a\}$
V_4: c	V_4: $\{b, c\}$

solution to the CSP. Thus, the robust solution in Table 2.1 represents 6 solutions. INDSET yields a single values to variables in $\bar{I}$ and sets of values to variables in I, and thus a robust solution to the entire CSP.

Summary

In this chapter, we reviewed the definition of a CSP and the basic concepts that will allow us to introduce our techniques. We also summarized the main related techniques in the literature.

Chapter 3

INDSET

We present here a decomposition method, which we denote INDSET, using maximal independent sets of the constraint graph of a CSP. We explain how INDSET can not only improve the performance of search but enable it to return multiple solutions. We also describe in general how this technique can be incorporated into stochastic local search.

3.1 Decomposition

INDSET partitions the variables of a CSP into two sets I and $\bar{I}$, such that I is a maximal independent set of the constraint graph (see Figure 3.1). By definition, no two variables in I are neighbors. Because I is maximal, every variable U in $\bar{I}$ has a neighbor in I. Otherwise, we could move U from $\bar{I}$ to I to obtain a larger independent set. Incidentally, $\bar{I}$ forms a minimal vertex cover on the constraint graph. We denote as $\mathcal{P}_I$ the subgraph of the constraint graph induced by I, and $\mathcal{P}_{\bar{I}}$ the subgraph induced by $\bar{I}$.

3.2 Solving a CSP with IndSet

IndSet allows us to focus our search on $\bar{I}$. After decomposition, we perform search on $\bar{I}$, and then propagate that solution to I. Figure 3.1 shows an example CSP decomposed into

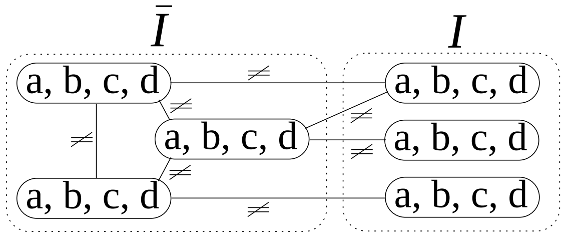

Figure 3.1: Step 1: Decompose into $\bar{I}$ and I.

I and $\bar{I}$, where I is an independent set. We can then solve $\bar{I}$ using any solving technique, which will yield an assignment to $\bar{I}$, as shown in Figure 3.2.

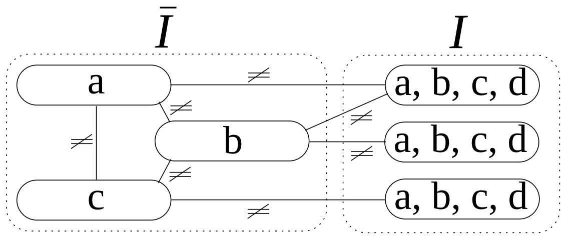

Figure 3.2: Step 2: Solve $\mathcal{P}_{\bar{I}}$, using any technique.

Once we have an assignment to $\bar{I}$, we can propagate that solution to I, using Directional Arc Consistency (DAC). The result of this is shown in Figure 3.3. DAC will remove any

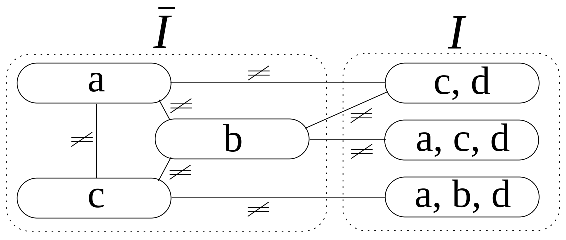

Figure 3.3: Step 3: Propagate to I using Revise(I, $\bar{I}$).

inconsistent values from the domains of the variables in I. In this final result we have a robust solution (i.e., every element of the Cartesian product of the remaining domains is a solution to the CSP).

For a given assignment of $\bar{I}$, and for each variable V_i in I, let d_i be the set of values in the domain of V_i that are consistent with the current assignment of $\bar{I}$. For any two variables V_i, V_j in I, we have $|d_i| \cdot |d_j|$ consistent combinations, because there is no constraint between V_i and V_j. In fact, every element of the Cartesian product $d_1 \times d_2 \times \cdots \times d_{|I|}$ is an expansion of the assignment of $\bar{I}$ to a full solution. Thus, for any assignment of $\bar{I}$ we can quickly obtain all $\prod_i |d_i|$ solutions possible with that assignment. Furthermore, because the result is the Cartesian product of subsets of domains, this possibly large set of solutions is represented in a compact form.

In Figure 3.3 the domains in I become $\{c, d\}$, $\{a, c, d\}$, and $\{a, b, d\}$. Consequently, the set of all possible remaining solutions consists of the Cartesian product of these three sets, along with the instantiation on $\bar{I}$. Thus, we have found $2 \times 3 \times 3 = 18$ solutions.

Again, in INDSET, we restrict the search to the variables in $\bar{I}$, and, for a solution of $\mathcal{P}_{\bar{I}}$, we can find all the resulting solutions using directional arc-consistency. This process reduces the search space by a factor exponential in the size of I. Consequently, we would like to choose an independent set I as large as possible. Finding the maximum independent-set of a graph is **NP**-hard [Garey and Johnson, 1979]. However, we do not need the maximum independent-set for this technique to be beneficial. Fortunately, many efficient approximation algorithms exist for finding independent sets [Boppana and Halldórsson, 1990]. Choosing an approximation algorithm depends on how much time one is willing to spend finding a large independent set.

For finding independent sets, we used the CLIQUEREMOVAL algorithm, which runs in polynomial time in the number of variables [Boppana and Halldórsson, 1990]. For the problem instances we used in our experiments, CLIQUEREMOVAL takes negligible time to execute (less than the clock resolution of the system on which we run our experiments, which is 10 ms). This algorithm incorporates some randomness in building the independent set. Running it multiple times may therefore yield different independent sets. Figure 3.4 shows the average size of the independent set found on random graphs of 80 variables using

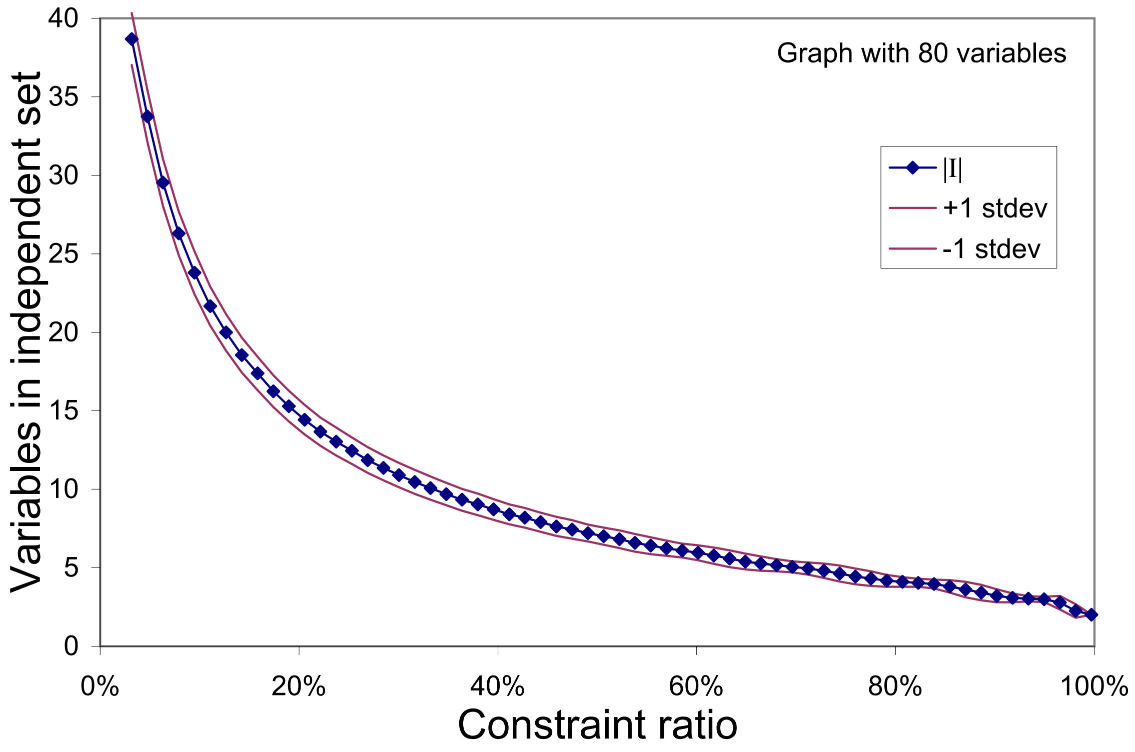

Figure 3.4: Independent set sizes using CLIQUEREMOVAL.

CLIQUEREMOVAL while varying the constraint ratio. It also shows lines above and below showing ±1 standard deviation. For each constraint ratio, we generated 100 graphs and ran CLIQUEREMOVAL 100 times on each graph.

Summary

The independent sets of the constraint graph can be used to focus the efforts, and improve the performance, of the search. It also provides the additional unique feature of producing multiple solutions instead of a single solution. This technique can be incorporated into any kind of strategy for solving the CSP induced by $\bar{I}$. In Chapter 4 we explain how to incorporate it with local search.

Chapter 4

Using INDSET with Local Search

Exploiting independent sets is straightforward for systematic backtrack search, because we can obtain a performance benefit by searching over the variables of $\bar{I}$ before those of I. Once we reach the variables of I, then the search is already complete, because we have considered all the constraints, since there are no constraints in $\mathcal{P}_I$. However, in general, it is less clear how decomposition and/or structural information can be used to improve stochastic local search [Kautz *et al.*, 1997]. Therefore, we focus our investigations on how to use INDSET in conjunction with local search. Our solution is to guide the local search on $\bar{I}$ with information from the constraints between I and $\bar{I}$.

Section 4.1 explains how INDSET can be incorporated into local search techniques. Section 4.2 provides several heuristics for use in this search technique. And Section 4.3 reports the results of empirical evaluation of the technique.

4.1 SLS/INDSET

We extend SLS to SLS/INDSET, which performs SLS on $\bar{I}$ and is guided by the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$. We incorporate, in the mechanism for solving $\bar{I}$,the information about the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$ by modifying the method by which SLS counts broken constraints (i.e., constraints violated by the current assignment). Algorithm 1, presents SLS/INDSET in detail.

Algorithm 2 SLS/INDSET.

```
Find an independent set $I$ and its complement $\bar{I}$.
for $i = 1$ to MAX_TRIES do
    Make a random assignment to $\bar{I}$.
    Filter the domains of $I$.
    Count $B$ = broken constraints in $\bar{I}$ + heuristic measurement of broken constraints
    between $I$ and $\bar{I}$.
    repeat
        if $B = 0$ then
            Return current assignment as a solution
        end if
        for each variable-value pair $< x_i, a_i >$ of $\bar{I}$ not in current assignment do
            Suppose we swap the pair for the current assignment of $x_i$.
            Filter the domains of $I$.
            Compute $B$ for this assignment.
        end for
        choose variable-value pair $< x_i, a_i >$ that minimizes $B$.
        $x_i \leftarrow a_i$
    until no further changes occur
end for
```

When counting broken constraints, SLS/INDSET includes not only the constraints in $\mathcal{P}_{\bar{I}}$, but also some measurement of the number of broken constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$. The question is how to define whether constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$ are broken. For two variables in $\bar{I}$, the situation is clear: they have values assigned to them and these values either do or do not violate a constraint. However, whether or not a particular constraint between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$ is broken is ambiguous. Each variable V in I participates in a set of constraints. If filtering the domain of V with all of the constraints does not annihilate the domain (i.e., leave it with no remaining values), then there is no problem. If however the domain is annihilated, it may be the case that filtering the domain with some subset of the constraints will not annihilate the domain. Because every subset of constraints may or may not annihilate the domain, we cannot say whether or not a particular constraint is broken.

As a result of this imprecision, there is more than one way to consider these constraints to be broken or not. For a constraint between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$, we have an assignment of the variable U in $\bar{I}$, but not to the variable V in I. There may be multiple values for V that support the assignment to U. Even if there are multiple values in V supported by every constraint on V, it is still possible for directional arc-consistency on V to annihilate its domain, due to other constraints. In fact, it may be the case that no single constraint on V will annihilate its domain, but certain combinations of the constraints may do so. In Section 4.2 we discuss ways to measure the brokenness of the constraints on V.

For example, in Figure 4.1, both values **B** and **C** in the variable in I are consistent

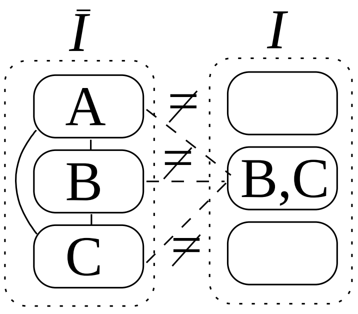

Figure 4.1: Example where a value in I will have its domain annihilated.

with the top-most constraint. In fact, taking any one of the constraints individually will not annihilate the variable's domain. However, when all of the constraints are considered simultaneously, neither value is consistent with the assignment to $\mathcal{P}_{\bar{I}}$, which means that the variable's domain will be annihilated.

Due to this repeated filtering of the domains of I, SLS/INDSET could be thought of as using a procedure analogous to backtracking in systematic search. We make an assignment to $\mathcal{P}_{\bar{I}}$. This assignment is checked for consistency with I. If it is not consistent, then we, in some sense, 'backtrack' to $\mathcal{P}_{\bar{I}}$ to obtain a new assignment to $\mathcal{P}_I$.

A possible modification for future investigation would be to move the finding of the independent set inside the first **for** loop in Algorithm 1, which would mean that a new independent set would be used for each restart.

4.2 Counting broken constraints

Recall that in SLS, incremental improvements are made to the current assignment. In order to provide an improvement, we need a measurement of the quality of assignments. A common such evaluation function is the number of constraints that are not satisfied (i.e., broken) by the assignment. As we discussed, this is measurement is not well defined for the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$. In this section, we explore different ways of defining this evaluation function for these constraints.

When designing a criterion for determining whether the constraints are broken or not, we would like to maintain the property that the number of broken constraints is zero if and only if we have at least one consistent solution to the CSP. Once we have no broken constraints, then, given the assignment on $\bar{I}$ and the filtered domains of the variables in I, we obtain at least one solution, and usually a large number of solutions. We implemented and tested five ways to count the number of broken constraints after an assignment is made: `None`, `Zero-domain`, `Some`, `All`, and `PrefRelax`. Note that the heuristics are applicable not only for the combination of INDSET with SLS but also for that of INDSET with other iterative-repair algorithms. Empirical evaluation of the heuristics are provided in Section 4.3.

`None`: The simplest approach is to simply ignore $\mathcal{P}_I$ and the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$. Figure 4.2 illustrates the heuristic `None`, showing an example where a variable in I will have its domain annihilated. This heuristic counts none of these constraints as broken. We perform search solely to find a solution to $\mathcal{P}_{\bar{I}}$, and then we check whether this partial solution extends to a full solution. If not, then we restart search again on $\mathcal{P}_{\bar{I}}$ with a new random initial assignment. Note that this heuristic does not

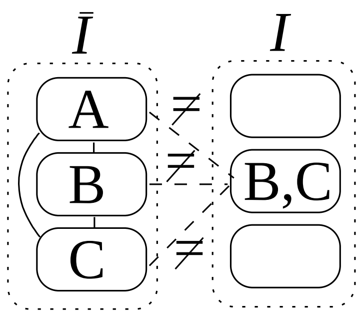

Figure 4.2: Heuristic `None`. No constraints are considered to be broken.

maintain the property outlined above (i.e., the number of broken constraints may be zero even when we have not found a solution to the CSP). One would expect that this approach performs poorly. We include this heuristic in our experiments solely as a frame of reference.

`Zero-domain`: In this heuristic, after making an assignment, we simply add the resulting number of variables with annihilated domains to the number of broken constraints in $\mathcal{P}_{\bar{I}}$. In Figure 4.2, the variable in I will have its domain annihilated, and this is counted as one broken constraint. On random problems, this method performed worse than SLS alone. In special cases SLS/INDSET(`Zero-domain`) did outperform SLS. In a star graph[1] [West, 2001], for example, it is obvious that any use of independent set information will yield an improvement, because it is easy to find an independent set containing $n - 1$ of the variables. INDSET allows us to focus the search on the single, center variable that really affects the problem. SLS alone will spread its search across the entire star graph, wasting much of its effort. In trivial cases like this one, even the poor-performing `zero-domain` significantly outperforms SLS. Because this heuristic performed poorly in preliminary experiments on random CSPs, we will not discuss it further.

`All`: In this next method, we filter the domains in $\mathcal{P}_I$ and then, for each variable left with

[1] A graph in which there exists a variable that participates in every constraint.

an empty domain, we consider *all* of the constraints on that variable to be broken. In Figure 4.3, illustrating heuristic `All`, the domain of the variable in I is annihilated,

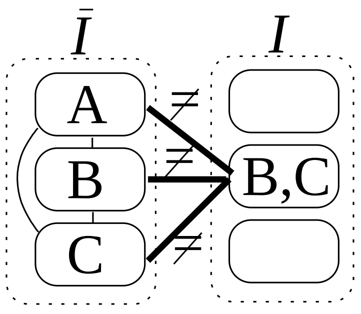

Figure 4.3: Heuristic `All`. All the constraints on the variable are considered to be broken.

so we count 3 broken constraints. Thus, we include in the count of broken constraints the sum of the degrees of the variables of I left with empty domains.

`Some`: In this method, we iterate through each of the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$. In Figure 4.4, illustrating heuristic `Some`, the constraints are considered from top to bottom. Consider one such constraint $C_{u,v}$ with V being the variable in I and U

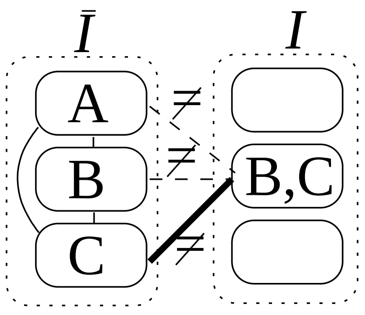

Figure 4.4: Heuristic `Some`. The constraints in bold are considered to be broken.

being the variable in $\bar{I}$. We reduce the domain of V to those values allowed by the constraint, given the value currently assigned to U. Each successive constraint on V may further reduce the domain of V. For each constraint, if this filtering annihilates the domain of V, then we consider the constraint to be broken. Any other constraint on V that we consider afterwards will also be considered to be broken. Note that the value returned by this heuristic depends on the order in which we examine the constraints on V.

`PrefRelax`: We also attempted a heuristic using preferred relaxations of Junker [2004]. Given a set of constraints that cannot all be satisfied simultaneously, a relaxation is a subset of those constraints that have a satisfying assignment. For each variable V in I, we found the preferred relaxation R of the constraints on V, given the lexicographical order of the constraints. We used $|D_u| - |R|$ as the measurement of the number of broken constraints on V. `PrefRelax` is a more accurate measurement of the number of broken constraints than `Some`. An even better consideration would be to find the *maximum* relaxation of the constraints. However, computing the maximum relaxation requires exponential time in the number of constraints in the worst case. For the problem sets in our experiments, `PrefRelax` did not provide a significant improvement over `Some`, likely due to the sparseness of the problems. See Section 4.3.3 for the experimental results. Further investigations may reveal an improvement of `PrefRelax` on denser graphs, or more specifically, on variables in I with a large degree. Also, additional heuristics for determining the order in which to consider the constraints on V may improve performance of `PrefRelax` as well as `Some`.

4.3 Empirical evaluation of SLS/INDSET

We empirically compared the performance of SLS with that of SLS/INDSET with the heuristics described in Section 4.2. We evaluate the ability of the algorithms to solve CSPs, and their runtime.

4.3.1 Experimental setup

We tested the algorithms on random CSP instances (Model B), with 80 variables, 8 values per variable, a constraint tightness of 58%[2], on connected graphs, varying the constraint ratio. In this and all experiments in this thesis that vary the constraint ratio, we varied the constraint ratio up to (and just beyond) the point at which problem instances are no longer solved within the given cutoff criterion. Beyond this point, the algorithms never return solutions and always terminate because of the cutoff criterion, and thus are not interesting to study with local-search algorithms. In each case, the graph shown displays the relevant and/or significant portion of the data. We used two different cutoff criteria in this thesis. The first stops the algorithm after a specified number of restarts have occurred. The second criterion stops the algorithm after a specified amount of time has elapsed. We perform arc-consistency before executing each algorithm, which is a common procedure for solving CSPs. Because arc-consistency itself can find a problem instance to be unsolvable, in our experiments we used only randomly generated problems that can be made arc-consistent.

For the problems used in these experiments, refer to Section 3.2 for the average independent set size found. For each value of constraint ratio, we tested the algorithms on over 1000 instances with a cutoff of 200 restarts per instance.

We selected the number of 1000 instances for the experiment because 1000 instances was sufficient to stabilize the mean. A larger sample size results in a more reliable estimate of various population statistics. A common way to determine what sample size is ‘large enough,’ is to compute the cumulative mean after every trial. This is called the moving average. Figure 4.5 shows an example of a moving average of runtime over 1000 instances. As the number of trials (and thus the sample size) increases, the mean begins to stabilize. A sample size of 1000 yielded an adequately stable mean.

[2]58% is a remnant from our following experiments of [Dechter, 2003].

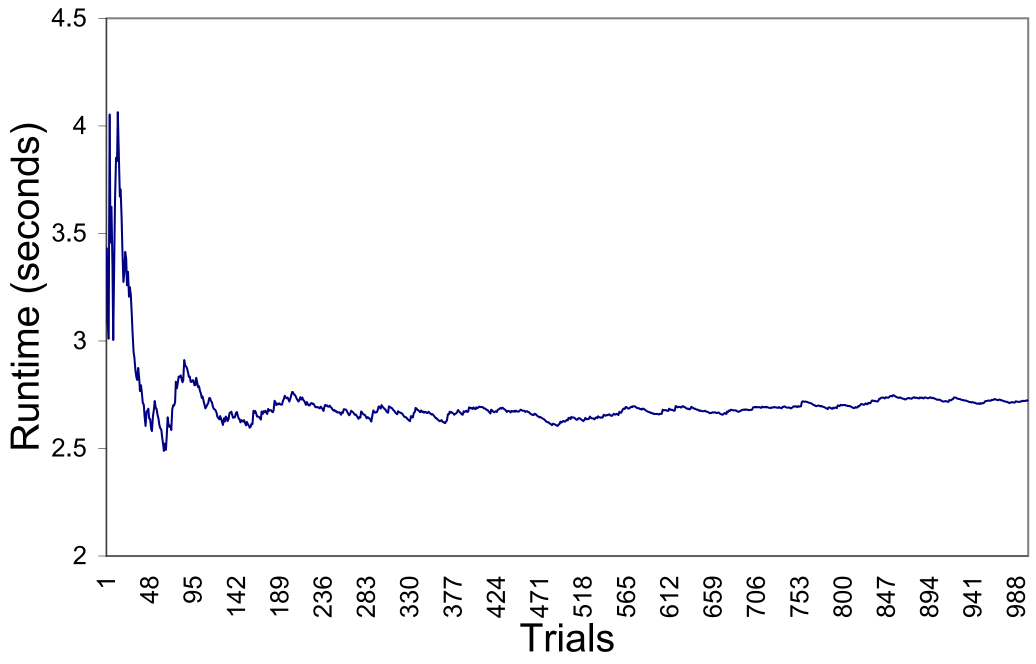

Figure 4.5: Example of moving average over 1000 trials.

4.3.2 Experiment 1: Ability to solve

We report in Figure 4.6 the percentage of instances each algorithm solved comparing SLS alone and SLS/INDSET with heuristics `Some`, `All`, and `None`. The results for `PrefRelax` are given below. For each curve, the algorithm solves 100% of problems with constraint ratios to the left of the curve and 0% of problems with constraint ratios to the right. Note that the curves for the different algorithms/heuristics have similar shapes but appear shifted to the left or right. The curves further to the right indicate that the algorithm can solve problems with more constraints. As a curve shifts to the right, it approaches the similar curve representing the percentage of problems that are actually solvable, corresponding to the phase transition. It is not feasible to compute the actual phase-transition curve for large size problems because a complete solver is needed, which requires an exponential amount of time (i.e., weeks of CPU time), and thus is infeasible for large problems. Consequently, we cannot easily determine how closely the algorithms tested approach the

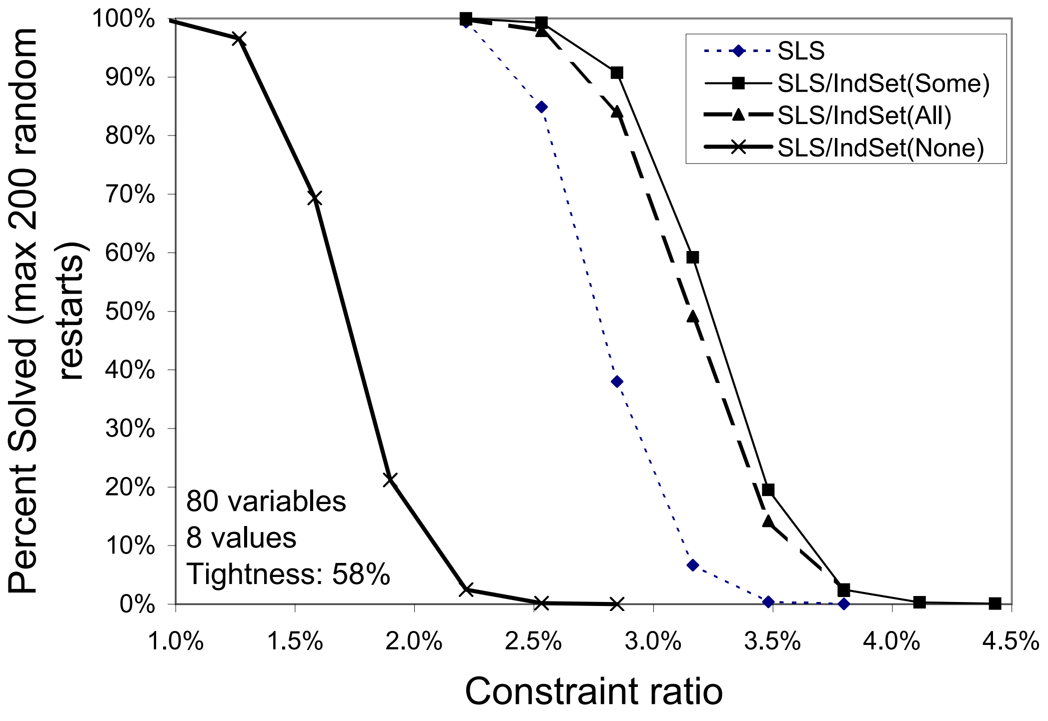

Figure 4.6: Percentage solved for SLS and SLS/INDSET.

actual curve.

It is clear from the graphs that `None`, as expected, performs poorly. It is surprising however that it performs as well as it does, considering that it is merely stumbling around in the dark by ignoring a large number of variables (those in I) and constraints (those between I and $\bar{I}$. The best-performing algorithm is SLS/INDSET using `Some`, although `All` is not far behind.

Finally, SLS/INDSET, with the various heuristics, returns a large number of solutions on average. In general, the algorithm returns an average of about three (out of eight) values per variable in I, over the tested range of constraint ratio. If the independent set contains 30 variables, then the number of solutions obtained is approximately 3^{30}. Figure 4.7 gives the average domain size for the variables in I in the robust solutions returned by SLS/INDSET, over varying constraint ratio. Figure 4.8 shows the estimated $\log_{10}$ of the number of solutions found, over varying constraint ratio. The estimate was computed as

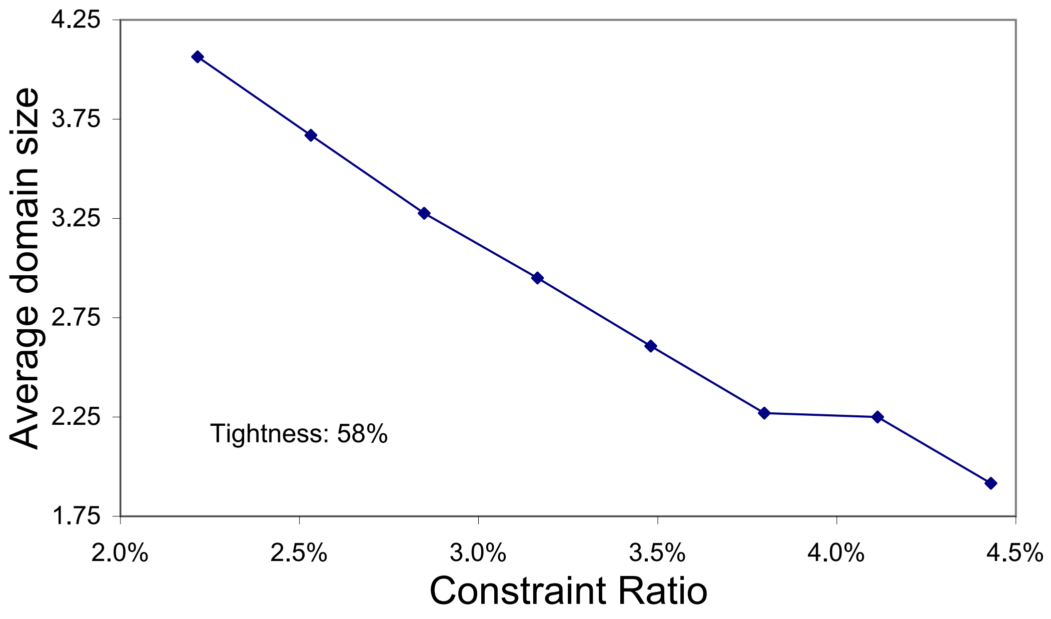

Figure 4.7: Average domain size.

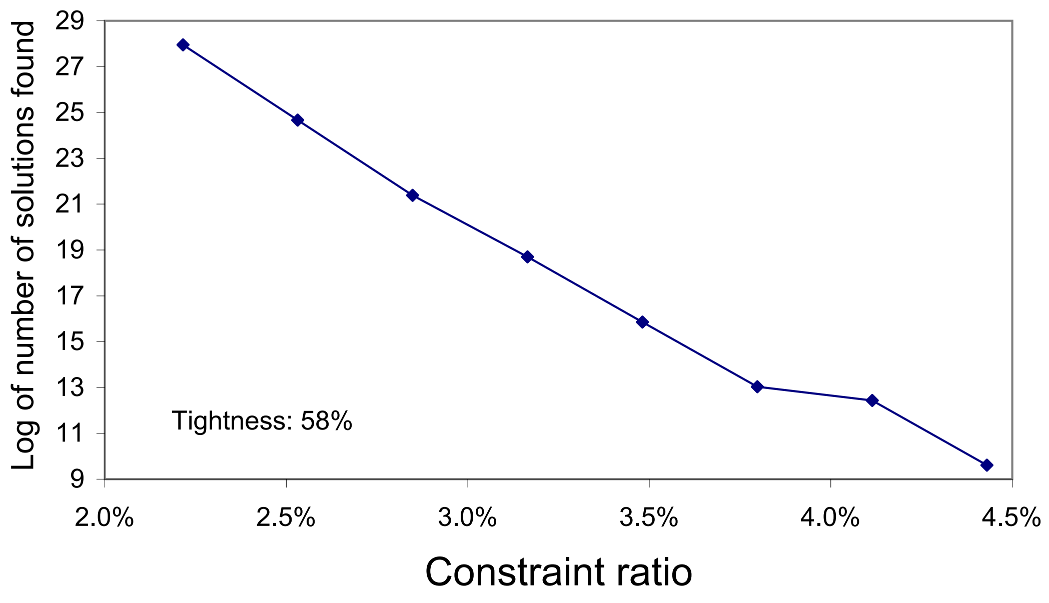

Figure 4.8: Estimated $\log_{10}$ of number of solutions found.

$\log_{10}(\prod_{i,V_i\in I} |S_{V_i}|) = \sum_{i,V_i\in I} (\log_{10}|S_{V_i}|) \approx I \cdot log_{10}(\bar{\mu}_{S_{V_i}})$, where $|S_{V_i}|$ is the number of values assigned to variable V_i in the solution, and $\bar{\mu}_{S_{V_i}}$ is the average number of values assigned to a variable in I.

4.3.3 Experiment 2: Runtime

The runtime for SLS and SLS/INDSET are shown in Figures 4.9, 4.10, and 4.11, for different tightnesses over varying constraint ratio. We used a cutoff time of 120 seconds. Each graph increases constraint ratio until SLS becomes unable to solve most of the problems (i.e., the median is no longer meaningful). Note that SLS alone sometimes runs faster on problems with low tightness and constraint ratio. Solving these problems is usually simple enough that the overhead of SLS/INDSET is unnecessary. INDSET provides greater benefits as constraint ratio increases until the problems are no longer solvable by SLS[3]. Overall it appears that `Some` has the best runtime for most of the range.

Figures 4.12, 4.13, and 4.14 show the results of comparing the runtime of `PrefRelax` and `Some`, over varying constraint ratio. For 10% tightness, `PrefRelax` does have a little better runtime than `Some` as the constraint ratio increases. This evidence supports our conjecture that `PrefRelax` would be more beneficial for higher constraint ratios. For 30% tightness, the two heuristics are approximately equal in performance, and for 50% tightness, `Some` dominates.

4.3.4 Experiment 3: Effect of independent-set size

We also performed experiments regarding the effect that the size of the independent set has on SLS/INDSET. We ran CLIQUEREMOVAL[4] repeatedly on the same graphs to obtain independent sets of different sizes. For example, we ran CLIQUEREMOVAL repeatedly

[3]Because these are not complete solvers, if the algorithm does not return a solution, we do not know whether the problem is solvable.

[4]Introduced in Section 3.2.

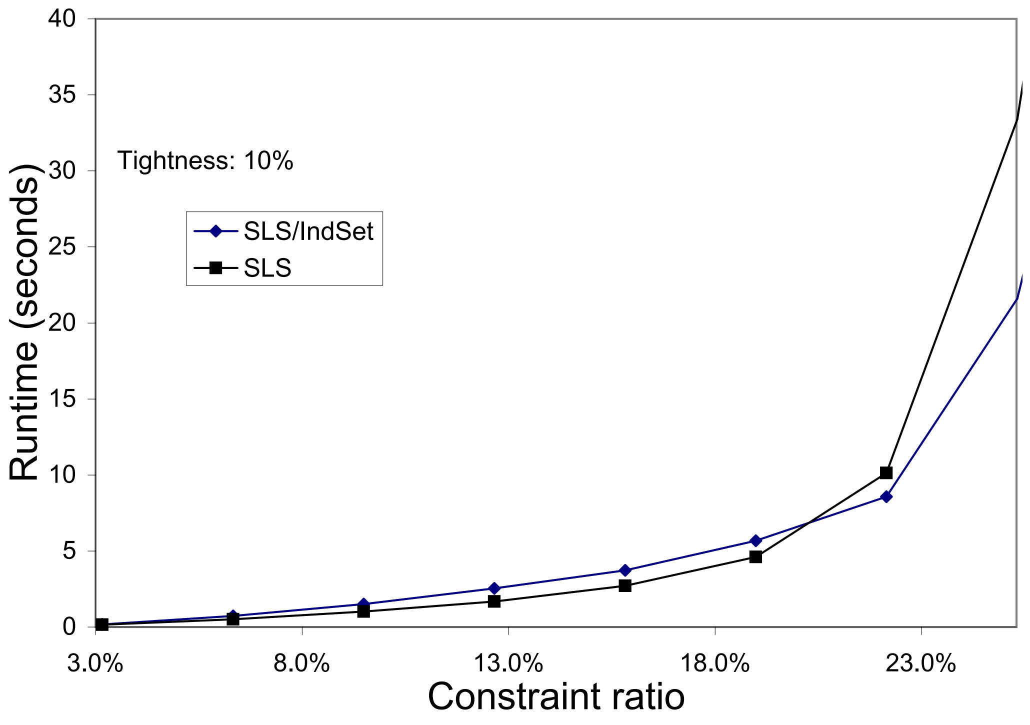

Figure 4.9: Runtime for tightness $= 10\%$.

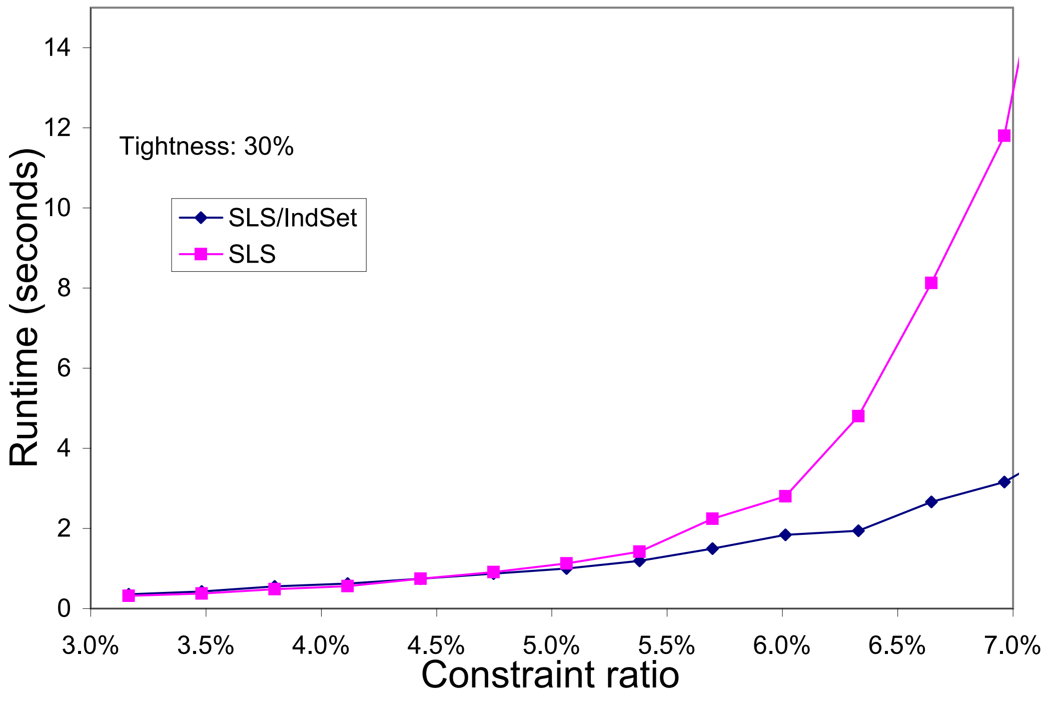

Figure 4.10: Runtime for tightness $= 30\%$.

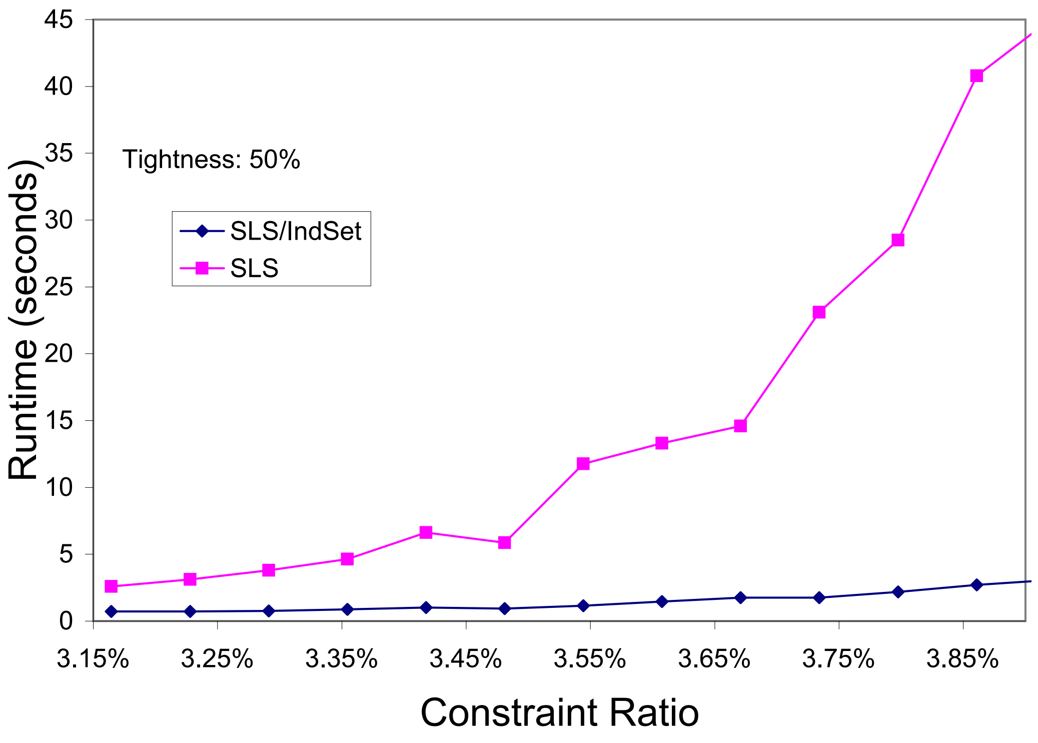

Figure 4.11: Runtime for tightness $= 50\%$.

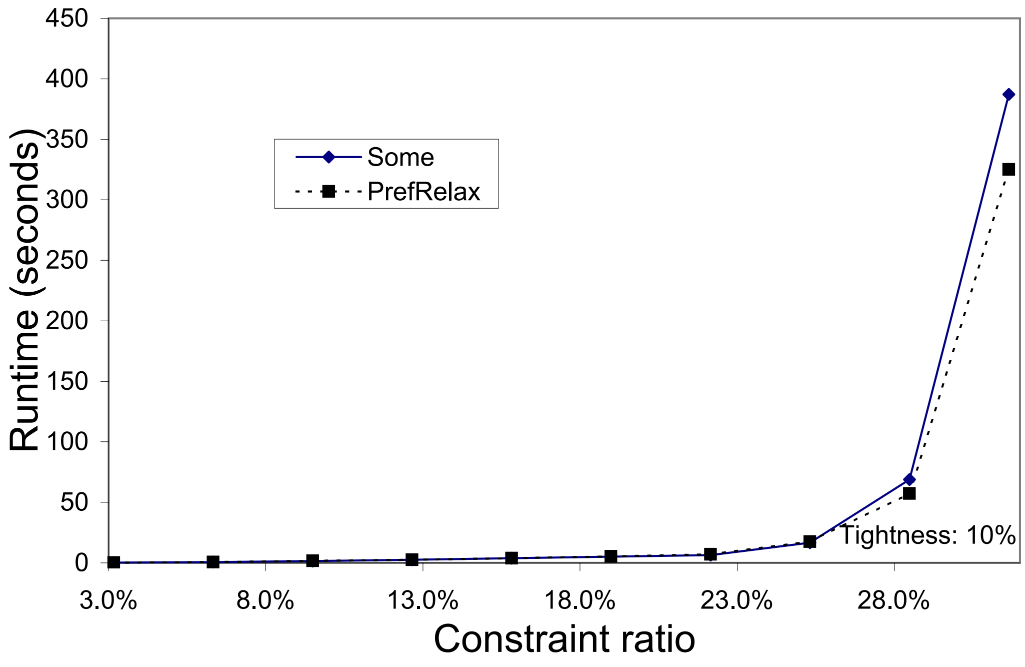

Figure 4.12: `PrefRelax` Runtime for tightness $= 10\%$.

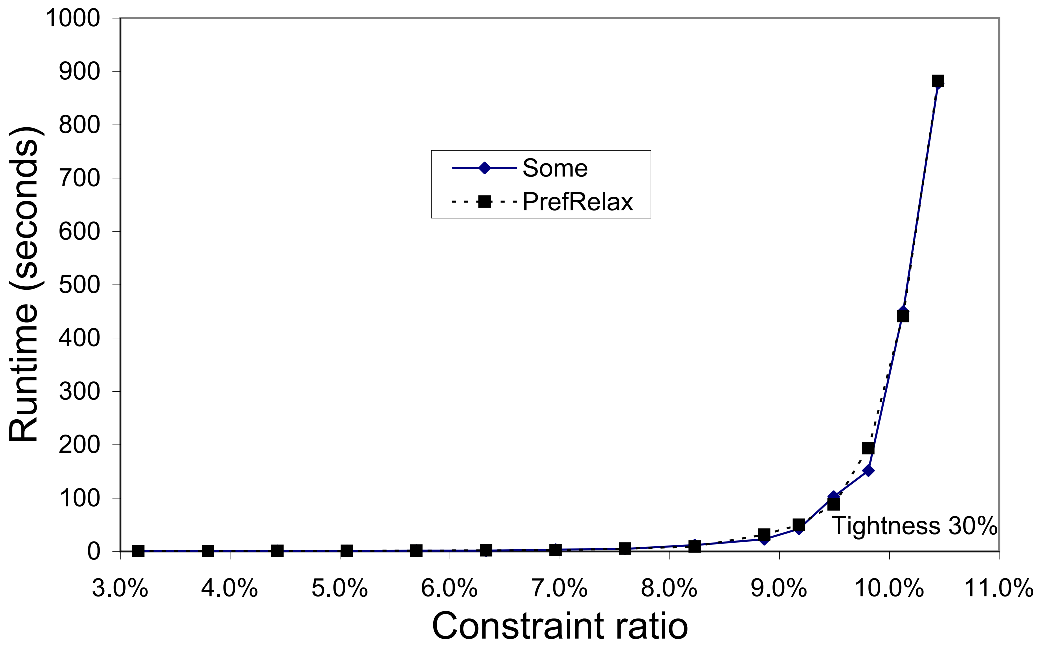

Figure 4.13: `PrefRelax` Runtime for tightness $= 30\%$.

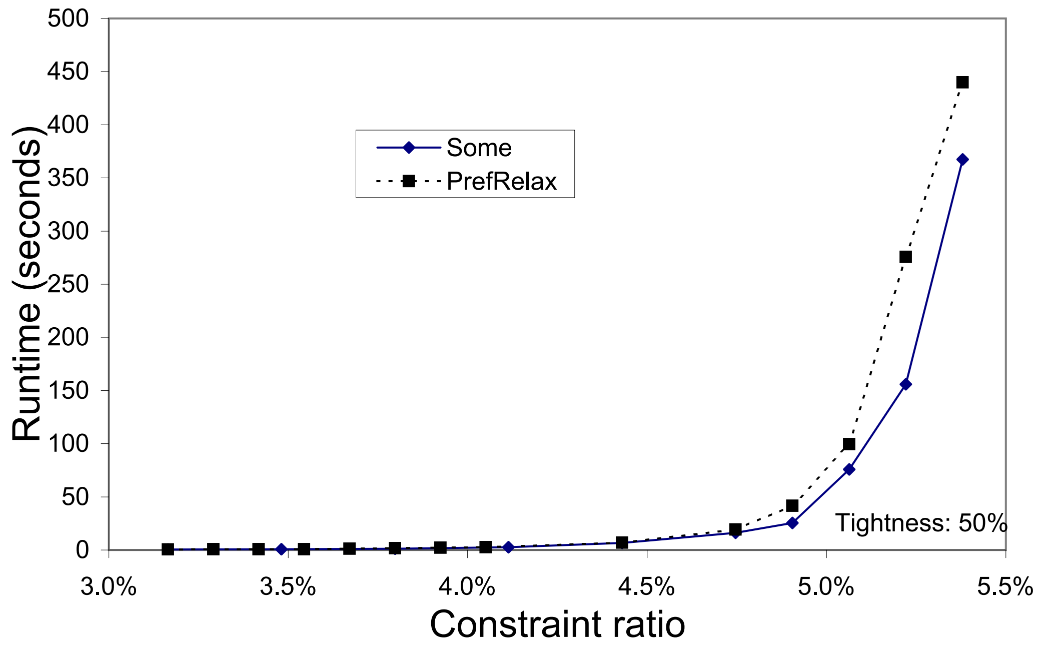

Figure 4.14: `PrefRelax` Runtime for tightness $= 50\%$.

on a graph of 80 vertices and 104 edges. Within a few minutes it returned independent sets ranging from 32 vertices to 41 vertices. Continuing to repeat CLIQUEREMOVAL over the following 24 hours did not yield any independent sets larger or smaller. After 1000 runs, with a tightness of 60% (which yielded a dramatic result for purposes of illustration), this graph was solved in an median of 147 seconds with SLS (effectively SLS/INDSET with empty independent set). Using the largest independent set found (41), SLS/INDSET solved in an median of 1.625 seconds, and using the smallest independent set found (32), SLS/INDSET solved in an median of 4.855 seconds. Table 4.1 summarizes these results.

Table 4.1: SLS/INDSET median runtime vs. size of independent set.

Size of I	SLS/INDSET runtime (seconds)
0	147.20
32	4.86
41	1.64

Clearly a larger independent set can greatly reduce the time to solve. The dramatic difference in running time seen in this example is partly due to the high tightness of the constraints. Problems with lower tightness yield a smaller difference, as was seen by comparing Figures 4.9, 4.10, and 4.11 (because SLS is SLS/INDSET with an empty independent set). SLS/INDSET with a non-empty independent set still yields more solutions than SLS. SLS/INDSET returns a multitude of solutions and does it in less time than SLS requires to return only one.

Summary

We presented and evaluated SLS/INDSET, one possible combination of INDSET with a local search algorithm. We discussed and compared heuristics for incorporating the decomposition into local search. The empirical results show a performance improvement in search when using the decomposition and how a large number of solutions were returned on average.

Chapter 5

Improving INDSET

Additional processing can be done to enhance and extend the benefit of the independent-set decomposition. For example, Section 5.1 examines the benefit of identifying ‘dangling trees’ in the constraint graph. Section 5.2 shows the results of using static bundling with INDSET, providing preliminary evidence that Neighborhood Interchangeability techniques can benefit INDSET. Also, Section 5.3 shows that INDSET can be used to find small cycle cut-sets.

5.1 Identifying dangling trees

We discuss additional processing to enhance INDSET and extend its benefits. We do so by detecting trees that ‘dangle’ off the variables in I. For a variable V in I, we find connected, acyclic, induced subgraphs that become disconnected from the rest of the graph if V is removed. These subgraphs are trees said to ‘dangle’ from V. An example of extracting these dangling trees is shown in Figure 5.1. The graph on the left shows an example of a CSP decomposed using INDSET, and the graph on the right shows the result of finding dangling trees.

We can find these dangling trees quickly, using a linear-time breadth-first search. Algo-

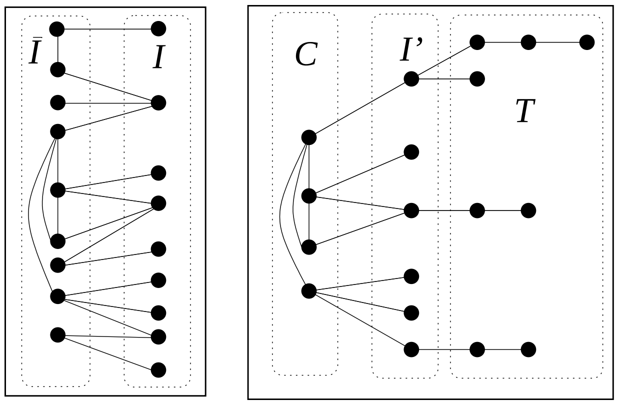

Figure 5.1: Dangling trees.

rithm 3, FINDDANGLESONVAR, finds the set of variables in all trees dangling off a given

Algorithm 3 FINDDANGLESONVAR(v).

```
result ← ∅
for each neighbor of $v$ do
    result ← result ⋃ GETTREE($v$, neighbor)
end for
return result
```

variable. Note that some of the vertices in the dangling trees were removed from I and some from $\bar{I}$, yielding $I' \subseteq I$ and $C \subseteq \bar{I}$ respectively. Let T be the set of variables in the trees dangling off I'. Algorithm 4, GETTREE, performs a breadth-first search starting with v_2, without searching past v_1, and stopping if a cycle is found. In the worst case, this algorithm requires time linear in the number of variables in the subgraph containing v_2 after v_1 is removed. Thus, in the worst case, it requires time linear in the number of variables in the graph (i.e., $O(n)$). FINDDANGLESONVAR requires time $O(n)$, because we can have GETTREE mark nodes as it searches, to ensure that we never visit a node twice.

5.1.1 Benefits of identifying dangles

Finding dangles has two benefits. It slightly reduces the runtime and increases the number of solutions returned. Suppose we enforce directional arc-consistency (DAC), from the

Algorithm 4 GETTREE(v_1, v_2).

```
Require: $v_1$ and $v_2$ are adjacent variables.
Ensure: Return set of variables of the tree rooted at $v_2$ dangling off $v_1$; otherwise return $\emptyset$
  result $\leftarrow v_1$
  stack $\leftarrow v_2$
  while NotEmpty(stack) do
    nextvar $\leftarrow$ Pop(stack)
    $N \leftarrow$ NeighborsOf(nextvar)
    $I \leftarrow$ result $\bigcap N$
    if SizeOf($I$) $\neq 1$ then
      return $\emptyset$ {We found a cycle}
    end if
    $N \leftarrow N \setminus I$
    result $\leftarrow$ result $\bigcup$ nextvar
    stack $\leftarrow$ stack $\bigcup N$
  end while
  return result $\setminus\{v_1\}$
```

leaves of the dangling trees towards the variables in I'. If DAC annihilates any domains, then we know immediately, and *before doing any search*, that the problem is not solvable.

Proposition 1 *When a CSP is decomposed with* INDSET+DANGLES, *and DAC is applied from the leaves of T to I', if any domain is annihilated, then the CSP has no solution.*

The proof is straightforward.

Also, any selection of a value remaining in the domain of a variable in I' can be extended to a solution of the trees dangling off that variable. Now, given an assignment on $\mathcal{P}_C$, because any two variables in I' (and their respective dangling trees) are disconnected from each other, we can select their values independently of each other.

Furthermore, we can completely ignore the nodes in T during search, because we know that any value that remains in the domain of a variable in I has a support. Thus, if we find a partial solution for the variables in C and if directional arc-consistency can successfully extend this partial solution to the variables I', then we know that this partial solution can necessarily be extended to at least one full solution in a backtrack-free manner [Freuder, 1982]. We can also determine a lower bound on the number of solutions as discussed in Section 6.1.

We perform search using an algorithm like SLS/INDSET on C and I' just as we did before on $\bar{I}$ and I, respectively, and ignore the vertices in T. Reducing the size of the cutset $\bar{I}$ to C has two advantages. First it reduces the search space, and secondly it yields more solutions.

Finding dangling trees reduces our search space because $|C| \leq |\bar{I}|$ and because it reduces the cost of the filtering step at each iteration of the inner-loop of Algorithm 2. This is because $|I'| \leq |I|$, and the number of constraints between C and I' is smaller than the number of constraints between I and $\bar{I}$.

Dangling trees also yields more solutions. Recall that every variable in $\bar{I}$ has a fixed value assigned to it, while the other variables can have a flexible assignment of multiple values. Because $|C| \leq |\bar{I}|$, the number of variables with a fixed, single assignment is reduced, and the number of variables that can have a flexible assignment is increased, thus increasing the total number of solutions returned.

5.1.2 Evaluation

Figure 5.2 shows the effect of dangle identification on reducing the number of variables that search needs consider. The vertical axis shows the ratio of the cutset size to the number of variables N, while the horizontal axis varies constraint ratio. For low constraint ratio, identifying dangles reduces the subgraph ($\bar{I}$ or C) over which we perform search. This benefit is negligible above 9%.

We implemented SLS/INDSET+DANGLES based on SLS/INDSET. Using the dangling trees requires a preprocessing step of directed arc-consistency from the leaves of the trees to the roots. Since we apply full arc-consistency as a preprocessing step to our experiments, this is not an issue. We considered 1000 instances for each set of problem parameter values,

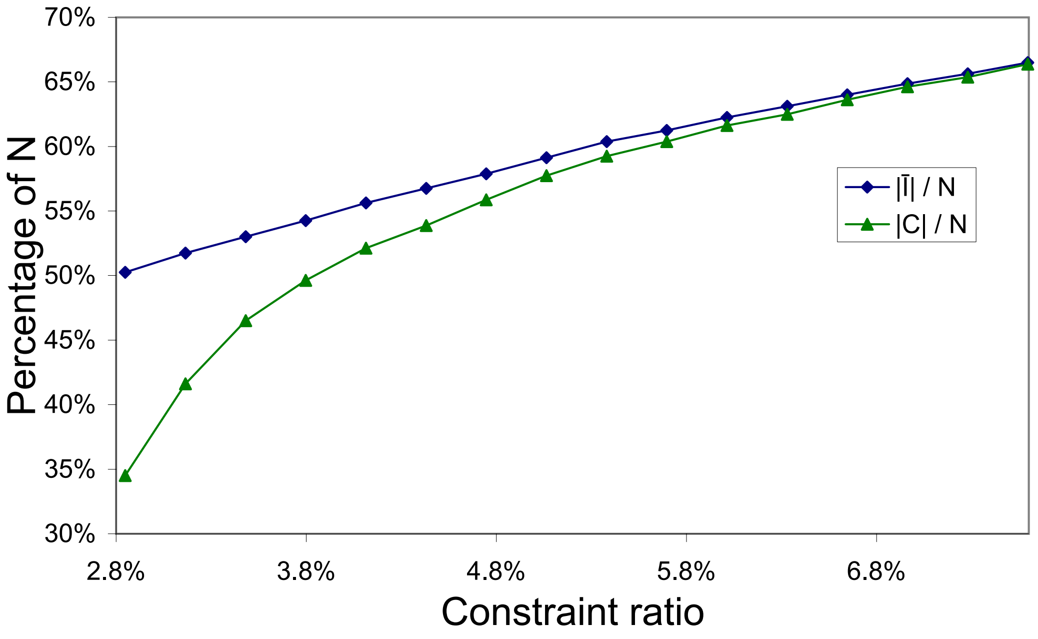

Figure 5.2: Ratios of sizes of C and $\bar{I}$ to the number of variables N.

with a cutoff criterion of 200 restarts.

Figures 5.3 and 5.4 show the results of comparing SLS, SLS/INDSET, and SLS/INDSET+DANGLES using the heuristic `Some` and at a high values of tightness and a low value of tightness. We show the results for `Some` because it yields the best results. The vertical axis shows the percentage of problems that were solved, while the horizontal axis varies constraint ratio.

For the lower tightness (i.e., Figure 5.3), the fall-off curve appears with a larger number of constraints, and the distinction between SLS/INDSET, and SLS/INDSET+DANGLES disappears. The improvement on percentage of problems solved of SLS/INDSET+DANGLES over SLS/INDSET is more visible at the higher tightness (Figure 5.4).

Figures 5.5, 5.6, and 5.7 show the runtime comparison for SLS, SLS/INDSET+DANGLES, and SLS/INDSET, over varying constraint ratio. Even in cases where the use of INDSET+DANGLES requires the same runtime as INDSET, it is still beneficial, since it yields more solutions than INDSET, in the same amount of time, as was explained in Section 5.1.1.

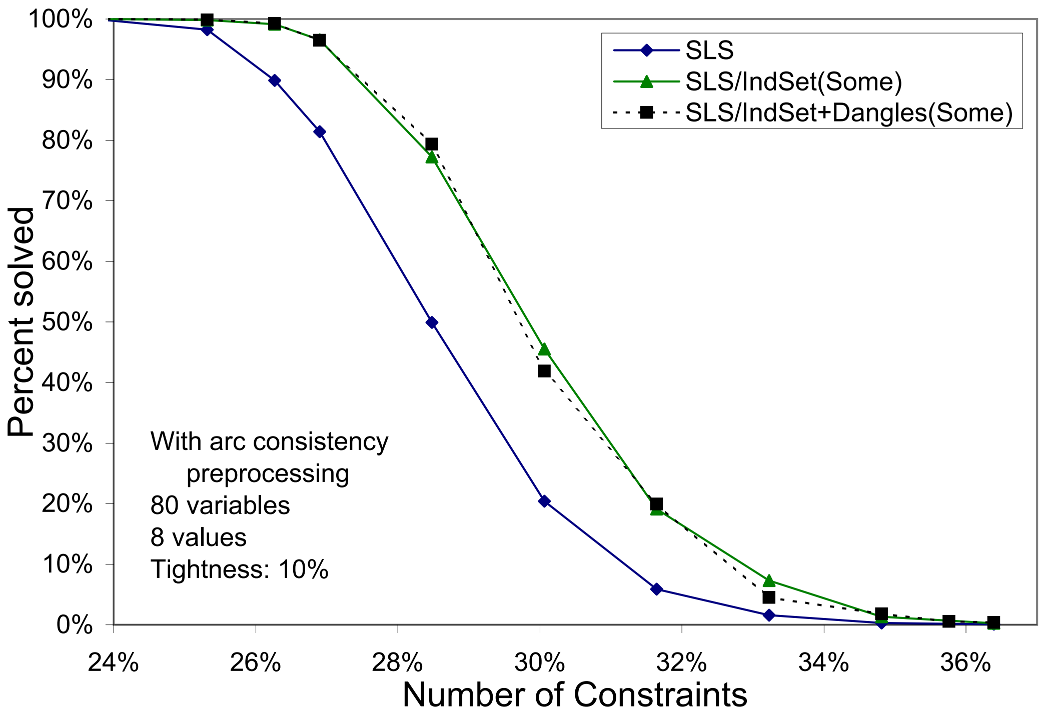

Figure 5.3: Percentage solved for SLS/IndSet+Dangles.

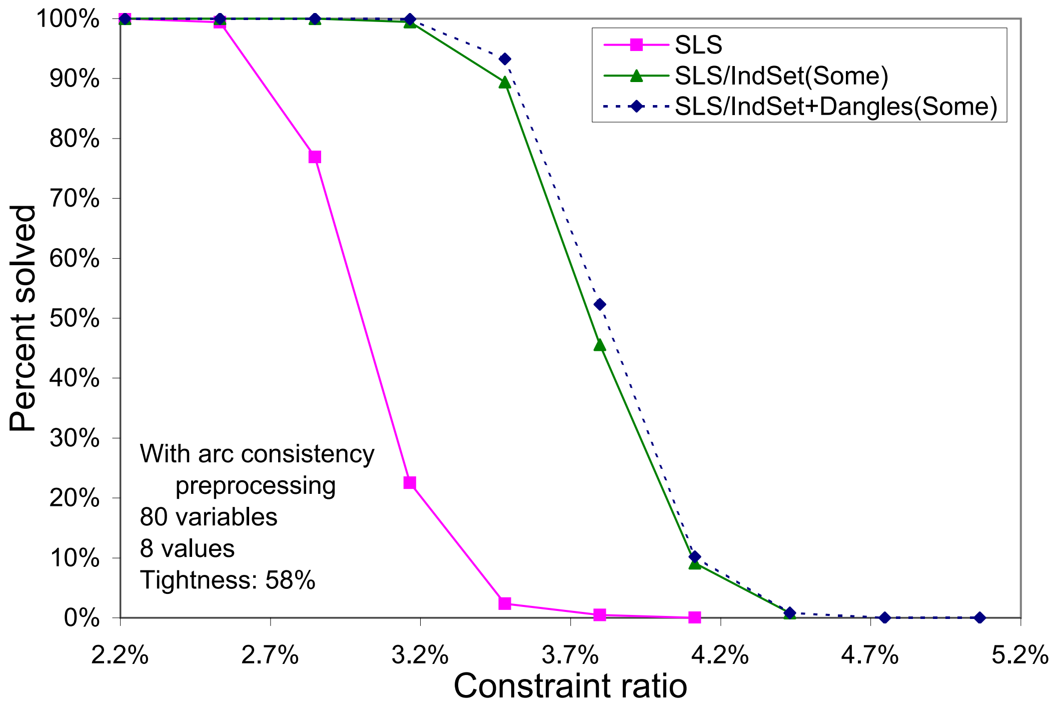

Figure 5.4: Percentage solved for SLS/IndSet+Dangles.

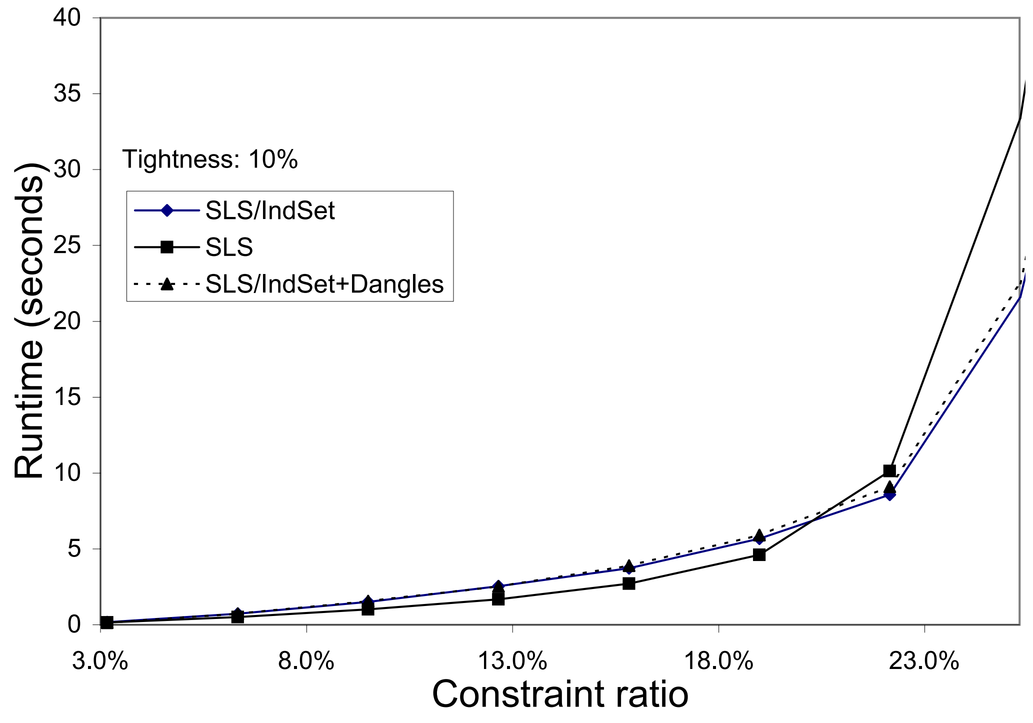

Figure 5.5: Runtime for tightness $= 10\%$.

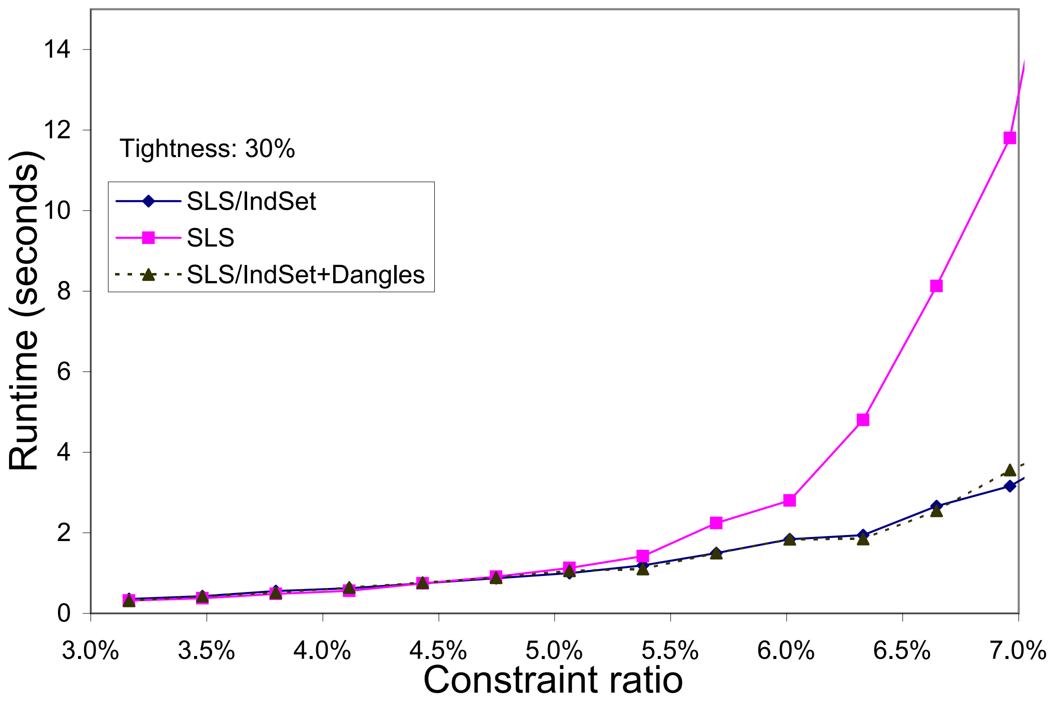

Figure 5.6: Runtime for tightness $= 30\%$.

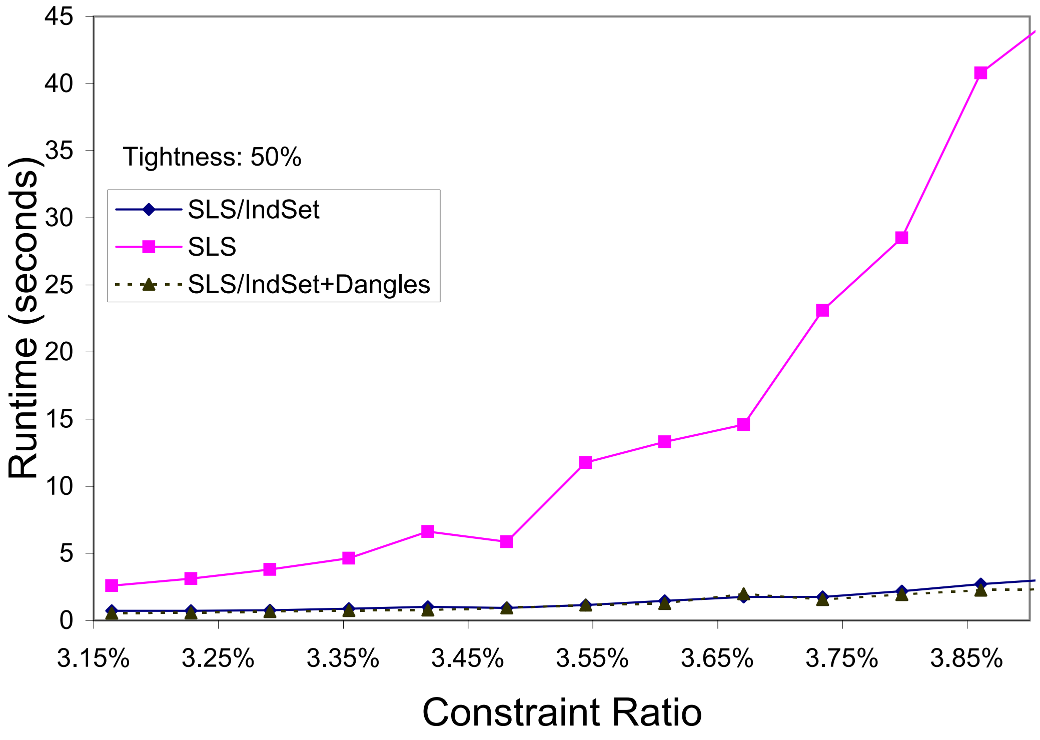

Figure 5.7: Runtime for tightness $= 50\%$.

A difficulty arises when attempting to count the number of solutions. Counting the solutions is easy with INDSET because it is the product of the domain sizes of the variables in I. Counting the solutions in INDSET+DANGLES is more difficult, because it requires performing a search on the trees rooted in I' and iterating through all of the solutions, which may be infeasible because the number of solutions is potentially exponential in the number of variables. Techniques for computing a loose lower bound on the number of solutions in the dangling trees are suggested in Section 6.1.

5.1.3 Performance near phase transition

Like SLS/INDSET, SLS/INDSET+DANGLES has a greater benefit closer to the phase transition area. Intuitively, it also has a greater benefit for lower-constraint-ratio graphs, which have a higher probability of having larger dangles. Indeed, we find that SLS/INDSET+DANGLES provides the greatest benefit when the phase transition exists at a small constraint ratio, which occurs as tightness increases. An example is shown in Figure 5.8, where the runtime cumulative distribution shows SLS/INDSET+DANGLES dominating SLS/INDSET on runtime for these tight, sparse problems.

5.1.4 Effect of $|I|$

We also performed experiments regarding the effect the size of the independent set has on the performance of SLS/INDSET+DANGLES. We used the same independent sets (by running CLIQUEREMOVAL repeatedly) and experimental setup used in Section 4.3.4. We used the same graphs and independent sets with SLS/INDSET+DANGLES and measured its runtime over 1000 runs. Using the largest independent set found, SLS/INDSET+DANGLES solved in an median of 1.16 seconds, and using the smallest independent set found, SLS/INDSET solved in an median of 2.79 seconds. Table 5.1 summarizes these results. A larger inde-

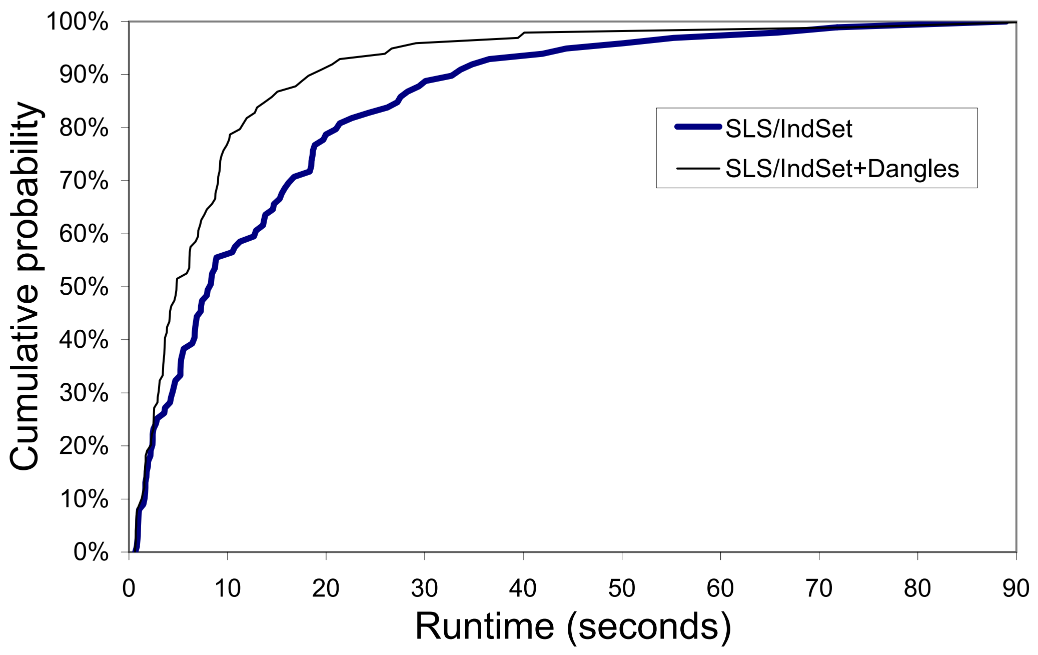

Figure 5.8: Runtime Cumulative Distribution for tightness $= 60\%$ and constraint ratio $= 3.5\%$.

Table 5.1: SLS/INDSET+DANGLES running time (seconds) vs. size of independent set for N=80.

Size of I	SLS/INDSET SECONDS	SLS/INDSET+DANGLES seconds	$\|C\|$	$\|T\|$
32	4.855	2.79	40	8
41	1.635	1.16	33	10

pendent set can significantly reduce the time to solve with SLS/INDSET+DANGLES.

5.1.5 Heuristics and DANGLES

Using the experimental data from experiments in Section 5.1, we compared the effect of using the different heuristics (of applying INDSET to local search) to the effect of using DANGLES. The runtime comparison is shown in Figure 5.9. SLS/INDSET+DANGLES(`All`) performs worse on average than SLS/INDSET(`Some`).

From Section 4.2, we know that `Some` performs better than `All`, and we know that using DANGLES performs better than not using DANGLES, thus, from Figure 5.9 it appears

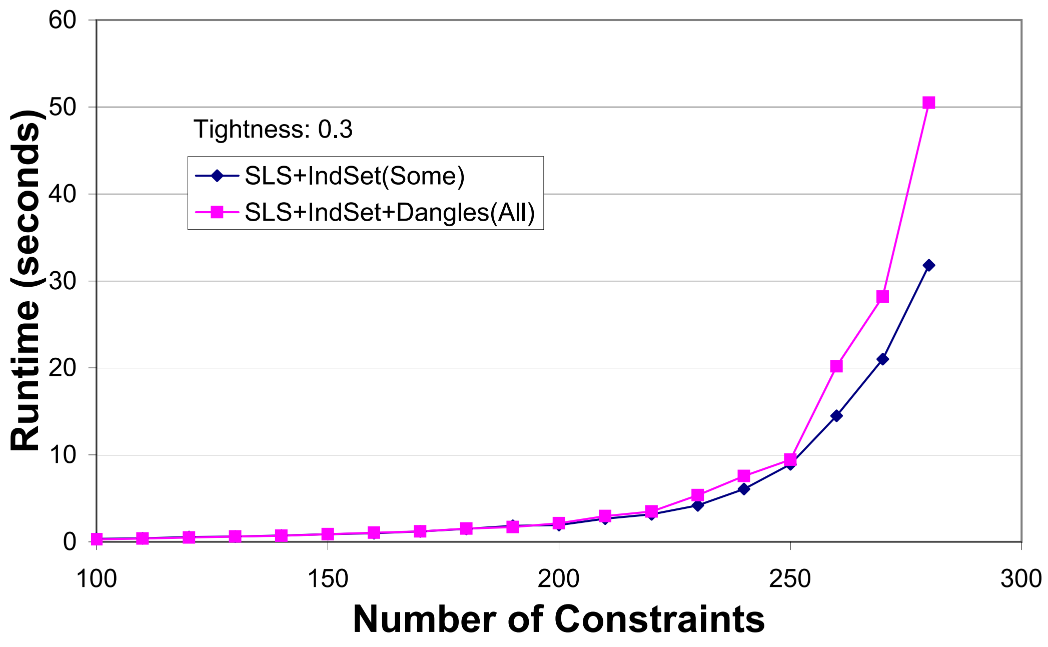

Figure 5.9: Comparison of heuristic to use of DANGLES.

that the benefit of `Some` over `All` is greater than the benefit of DANGLES. Consequently, the choice of heuristic has a greater influence on runtime than whether or not we use DANGLES. However DANGLES increases the number of solutions found.

5.1.6 Finding dangles first

As described earlier, dangles are identified off variables in I, that is, we find them after decomposing the CSP with INDSET. Further experimentation has lead us to advise

1. to apply the dangle identification procedure before applying the decomposition, and
2. to do so for all the variables in the CSP.

We call this technique DANGLES+INDSET. Note also that finding dangles first may result in having trees that dangle off variables in C as well as dangling off variables in I. We designed SLS/DANGLES+INDSET to ignore these trees after directional arc-consistency in the same way SLS/INDSET+DANGLES ignores trees in T assuming no domain was annihilated. Note that identifying dangles, which can be done efficiently, could benefit the performance of any search procedure (i.e., not only SLS) for solving a CSP, regardless of whether or not we use INDSET. We can find dangling variables in a CSP using Algorithm 5.

Algorithm 5 FINDDANGLES.

```
result ← $\emptyset$
stack ← set-of-all-variables
while NOTEMPTY(stack) do
    $V_i$ ← POP(stack)
    temp ← FINDDANGLESONVAR($V_i$)
    stack ← stack $\setminus$ temp
    result ← result $\bigcup$ temp
end while
return result
```

FINDDANGLESONVAR in Algorithm 5 is a depth-first search procedure that returns the set of variables that are in trees dangling off the specified variable, and runs in $O(N)$ time in the worst case. Because, in the worst case, FINDDANGLES calls FINDDANGLESONVAR for every node in the graph, FINDDANGLES is $O(N^2)$. However, this worst case is not likely to occur, because it requires specially structured graphs (such as when a triangle is attached to a leaf of a tree), and then only when we consider the nodes in a particular order. FINDDANGLES finds the set of nodes in the graph that do not participate in any cycles[1].

We tested DANGLES+INDSET (which extracts dangles first then finds an independent set) and INDSET+DANGLES (which finds an independent set first and then extracts dangles). We measured the sizes of C and T for Model B random CSPs over a range of densities on problems with 100 variables. Because the sets of variables identified by INDSET and DANGLES are derived from only the structure of constraint graph, the domain sizes and constraint tightness are irrelevant. Figure 5.10 shows that finding dangles first does result in a larger set of dangles than finding dangles last. (Obviously, it can never result in

[1]It does not fi nd isolated nodes, but we we consider only connected graphs. Typically, if a CSP is disconnected, then each connected component is treated as a separate CSP and is solved independently.

a smaller set of dangles.)

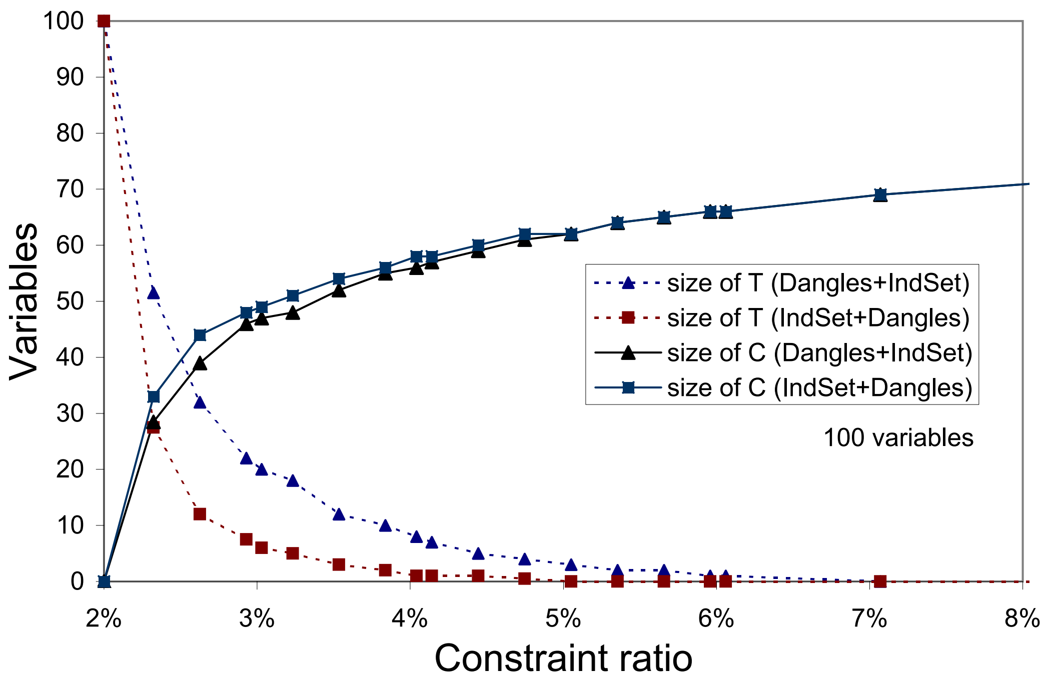

Figure 5.10: Comparing $|T|$ and $|C|$ in DANGLES+INDSET and INDSET+DANGLES.

The increase of the size of T must come from a reduction of the size of I' or that of C. Note that, in Figure 5.10, because the resulting reduction of $|C|$ is smaller than the increase of $|T|$, most of the newer dangles must come from I'. Thus, most of the added benefit from finding dangles first lies in reducing the time required in the domain filtering step of SLS on the decomposition, because I' has fewer variables, and there are consequently fewer constraints between I' and C. We show in Figure 5.11 the runtime cumulative distributions for SLS/DANGLES+INDSET and SLS/INDSET+DANGLES on 1000 instances of Model B random CSPs, with 100 variables, 10 values per variable, 50% tightness and 2.6% constraint ratio. This figure shows that SLS/DANGLES+INDSET outperforms SLS/INDSET+DANGLES. Incidentally, the ‘hump’ between 1 and 2 seconds, on both curves, corresponds to the first time SLS restarts due to encountering a local opti-

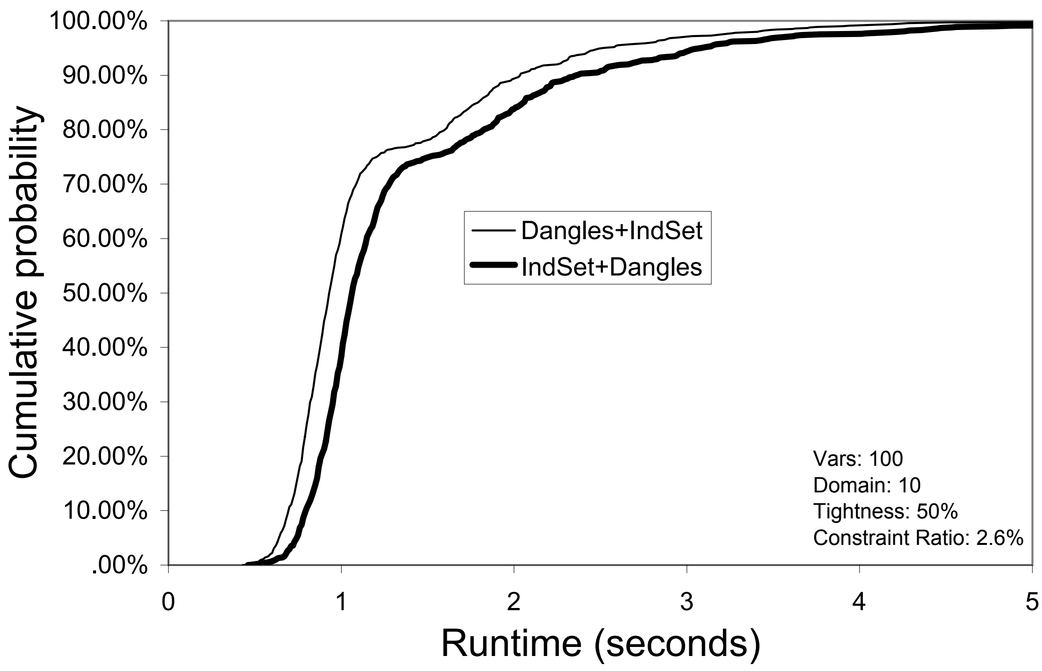

Figure 5.11: Runtime evaluation of finding dangles first.

mum[2].

5.2 Neighborhood interchangeability

We also attempted to exploit interchangeability techniques in INDSET. Two values of a CSP variable are said to be neighborhood interchangeable (NI) if they are consistent with the same values of the neighbors of the variable in the CSP. Freuder [1991] provided a mechanism that partitions the domain of a CSP variable into sets of equivalent values according the notion of neighborhood interchangeability. Choueiry and Noubir [1998] modified this mechanism to (1) apply to any restricted neighborhood of the variable and to (2) determine how the variable can be disconnected from this restricted neighborhood. The latter is indeed possible when there is one or more values in the domain of the variable that are found consistent with all the values in the restricted neighborhood because the variable can be safely assigned any of these values and disconnected from the specified neighborhood. For example, in Figure 5.12, the value f for the variable V_2 is consistent with all of its

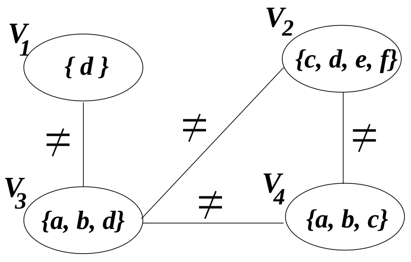

Figure 5.12: Example CSP with potential bundling.

neighbors' values. Thus we can make the assignment of f to V_2 and disconnect V_2 from the rest of the problem. We use the mechanism of [Choueiry and Noubir, 1998] by applying it to each variable in I' using each variable's neighbors in C as a restricted neighborhood.

We exploit the mechanism in two ways. First, when we find that a variable $V_i \in I'$ can

[2]The restart time varied among the 1000 runs, but they were clustered around the area of the 'hump.'

be disconnected from the variables in C, we restrict the domain of V_i to the values consistent with all the values of V_i's neighbors in C, and then remove V_i and its dangling tree from the problem. Naturally, this pruning should be done only after performing directional arc-consistency on the decomposed CSP, from the leaf nodes in T towards nodes I', in order to ensure that V_i has only values that are consistent with the dangles. This variable, restricted to its values consistent with C and together with its dangles, forms a subproblem that we solve independently. Because it is a tree, it can be solved in linear time by backtrack-free search [Freuder, 1982].

Our second application of neighborhood interchangeability was to use the restricted NI-sets found for the variables remaining in I'. We replaced the values in each equivalent set by a single representative. For example, in Figure 5.12, $\{e, f\}$ is an NI-set. Because e and f are equivalent, we can replace those two values with one that represents both of them. This operation reduces the effective domain size of the corresponding variable and can reduce the cost of solving the CSP.

We tested the impact of the above two ideas on the performance of SLS/INDSET+DANGLES by running experiments with and without NI. Figure 5.13 shows the resulting runtimes for Model B random CSPs with 80 variables, domain size 10, and 10% constraint tightness, and a cutoff criterion of 120 seconds. (The results on clustered graphs are shown in Figures 6.5 and 6.6.) As it can be seen, the improvement due to NI is not significant, but increases as we approach the region of the phase transition, where the cost of the problem solving drastically increases. Thus, this research direction may benefit from further investigations.

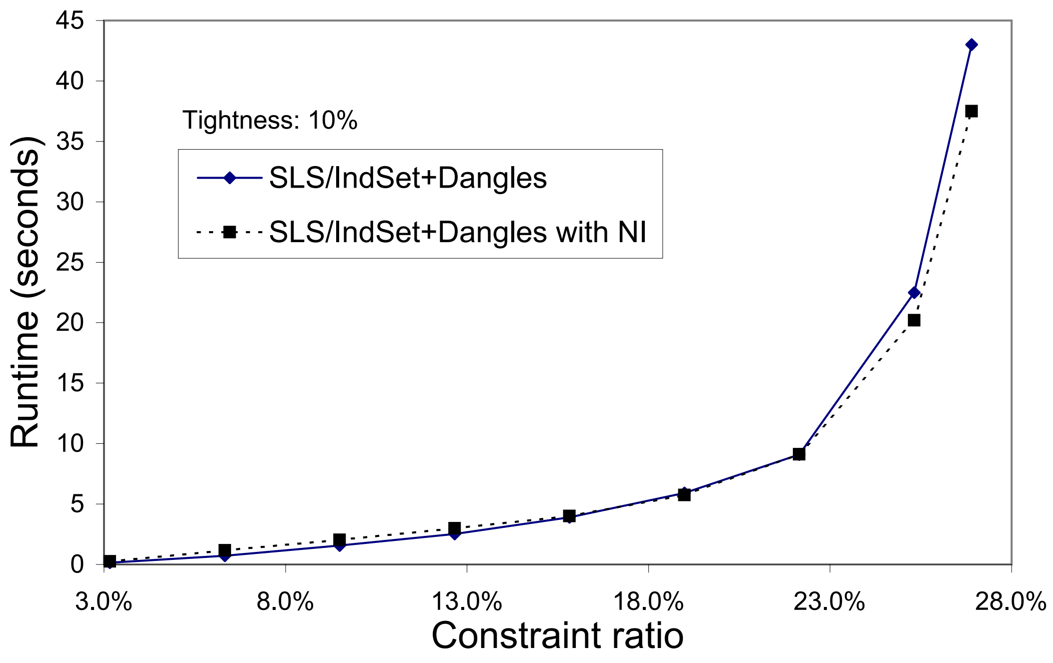

Figure 5.13: Results of using neighborhood interchangeability.

5.3 New technique for finding cycle-cutsets

This study on independent sets and the dangling trees has also led us to propose a new algorithm for finding cycle cutsets. Here are some of the previous heuristics for finding cycle cutsets [Barták, 1998]:

1. Remove variables from the graph in order of decreasing degree (DD) until no cycles remain.

2. Improve the results of (1) by, before removing a variable, first checking whether that variable participates in any cycle.

3. Determine, dynamically, for each variable, the number of cycles in which it participates, and take the variable that participates in the most cycles.

The technique we propose is to use the independent sets to perform a pre-processing step on the nodes of the graph before searching for a cycle-cutset. Finding an independent set I and the dangling trees, as in Section 5.1 leaves us with a set of variables C. C is a cycle-cutset, because removing C from the graph will leave only trees remaining. C is likely *not* a minimal cycle cutset, thus some subset of C may also be a cycle-cutset). We propose finding this subset of variables C as a preprocessing step to searching for a cycle cutset, which can save time by eliminating the need to consider the variables outside C when looking for a cycle cutset. We can search for a cycle-cutset within C using whatever technique we wish, such as heuristic (2) above, checking whether each variable participates in any cycle (because C may still contain variables that do not participate in any cycle).

We performed experiments comparing this IndSet method, DD (heuristic 1 above), and DD-WITH-CHECK (heuristic 2), across the range of constraint ratio. Cutset sizes were found for 1000 instances for each value of constraint ratio. The average cutset sizes are shown in Figure 5.14. Note that, for most of the range, INDSET outperforms the others

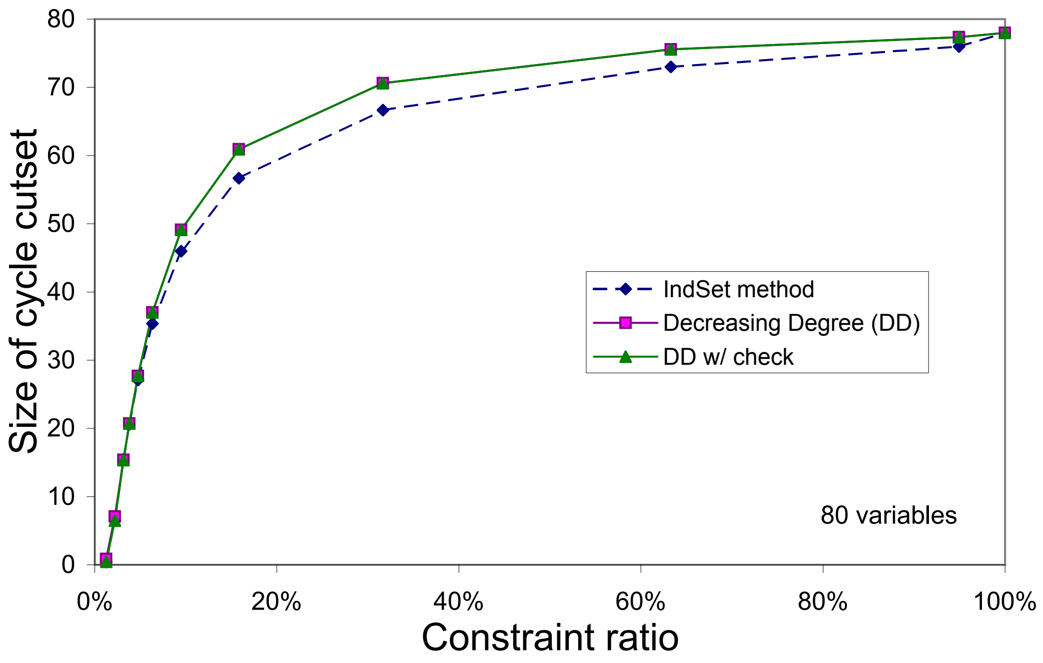

Figure 5.14: Comparing cutset sizes for three techniques.

in cutset size. And, for most of the range, DD and DD-WITH-CHECK return identical cutsets. To show the comparison between INDSET and DD-WITH-CHECK more clearly, Figure 5.15 shows the ratio of size of cutsets obtained by the former to the size of cutsets obtained by the latter. Only for a small range of graphs with low constraint ratio, does

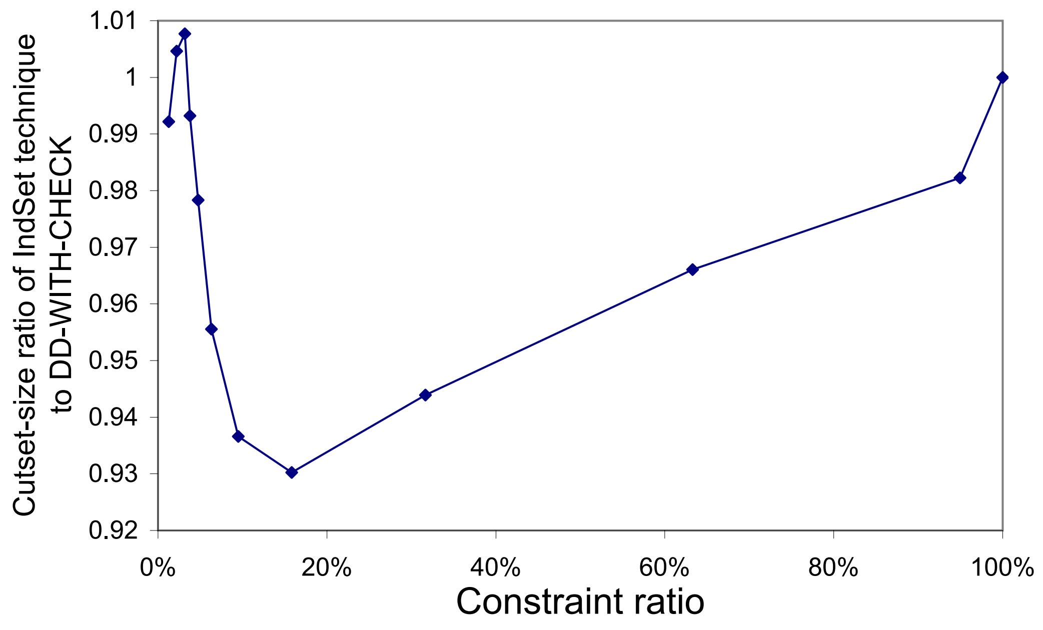

Figure 5.15: Ratio of cutset sizes of INDSET and DD-WITH-CHECK.

DD-WITH-CHECK return better cutsets.

5.3.1 Runtime

To return these smaller cutsets, the INDSET method does take longer to run. Figure 5.16 shows the ratio of runtimes of INDSET to DD-WITH-CHECK, over varying constraint ratio. For most of the range, INDSET requires a small increase in time. Note that if the runtime for solving the problem is exponential in the size of the cutset, then a small amount of extra time to find a smaller cutset will be more than offset by the reductions in search cost. The rapid increase at the left end of the graph results as the runtime for DD-WITH-CHECK falls below the resolution of our system timer, and the runtime of the INDSET technique becomes dominated by the time to find the independent sets. This data point corresponds to a runtime of about 200 ms for INDSET and 10 ms for DD-WITH-CHECK.

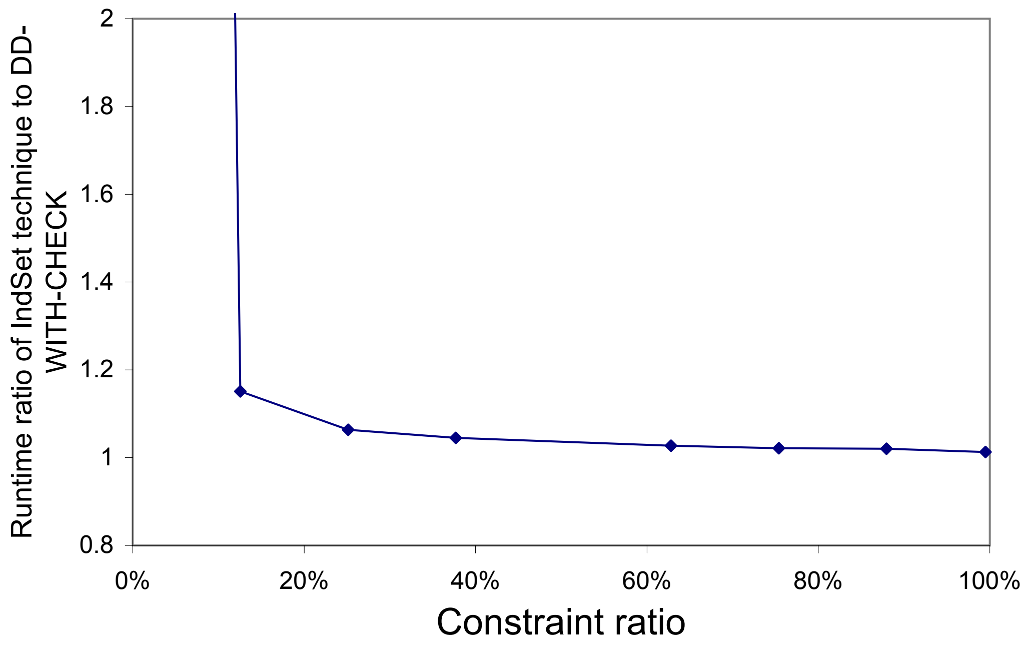

Figure 5.16: Ratio of runtime of INDSET and DD-WITH-CHECK.

Work has also been done on finding cutsets that leave more connectivity in the graph [Bidyuk and Dechter, 2004]. We will discuss this further in Section 6.2.

Summary

By identifying trees dangling off the independent-set variables, the efforts of search can be focused further, slightly reducing the time to solve, and increasing the number of solutions returned. We obtained preliminary evidence for the benefit of NI techniques on INDSET, and further investigation was suggested. It was also shown that INDSET may be useful as a preprocessing step when attempting to find small cycle cutsets.

Chapter 6

Analysis

In this chapter, we report additional qualitative and quantitative analysis of INDSET. First, Section 6.1 explains how INDSET allows us to compute lower bounds on the number of solutions extendible from a given robust solution. Section 6.2 compares and contrasts INDSET with CYCLE-CUTSET of Dechter [1987, 2003]. Section 6.3 identifies potential sources of runtime variance. Section 6.4 evaluates the performance of INDSET+DANGLES on clustered graphs.

6.1 Computing lower bounds of the number of solutions

Dangling trees increases the difficulty of counting the number of solutions found, when the solving process has completed. Counting the solutions is easing with INDSET because it is the product of the domain sizes. Counting the solutions in INDSET+DANGLES is more difficult, because it requires performing a search on the trees and iterating through all of the solutions, which may be infeasible because the number of solutions is potentially exponential in the number of variables. We discuss here techniques for computing a loose lower bound on the number of solutions in the dangling trees.

After instantiating the variables C, the nodes in I' become the roots of a forest, as shown in Figure 6.1. Each of the trees in this forest can be solved independently of the

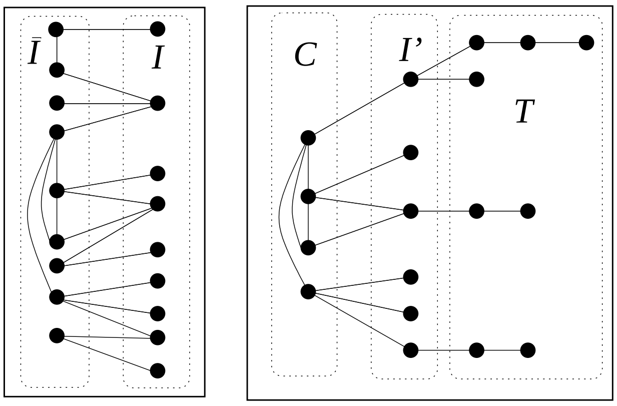

Figure 6.1: Dangling trees.

others, which increases the robustness of our solutions because modifying the solution to one tree does not affect the others. Note that CYCLE-CUTSET does not necessarily yield such independent trees, and when it does, recognizing them requires additional effort.

In addition to improving the performance of problem solving and returning multiple solutions, INDSET allow us to compute a lower bound of the number of solutions that can be extended from the partial assignment.

After instantiating the variables in C, when performing directed arc-consistency (DAC) on I' does not annihilate the domain of any of the variables in I', then the number of solutions to any of the trees rooted in I' is bounded from below by the size of the largest domain of the tree. Indeed, in each tree, we know that any value of each domain is part of some solution. Thus, we can choose any value for any of the variables in a tree and extend it to a full solution for the tree (and thus for the CSP) using backtrack-free search. Consequently, the number of solutions must be at least the size of the largest domain in the tree. Furthermore, because each of the trees can be solved independently, the product of the maximum domains of all the trees gives us a lower bound on the number of solutions we have obtained for the entire CSP: $\prod\limits_{\forall t_i, trees} \textsc{MaxDomain}(t_i)$. Figure 6.2 shows an example comparing the number of solutions found by SLS/INDSET with the lower bound of the number of solutions found by SLS/INDSET+DANGLES. Although the two lines appear approximately the same, note that the line for SLS/INDSET+DANGLES is a loose lower bound, and the actual value is larger.

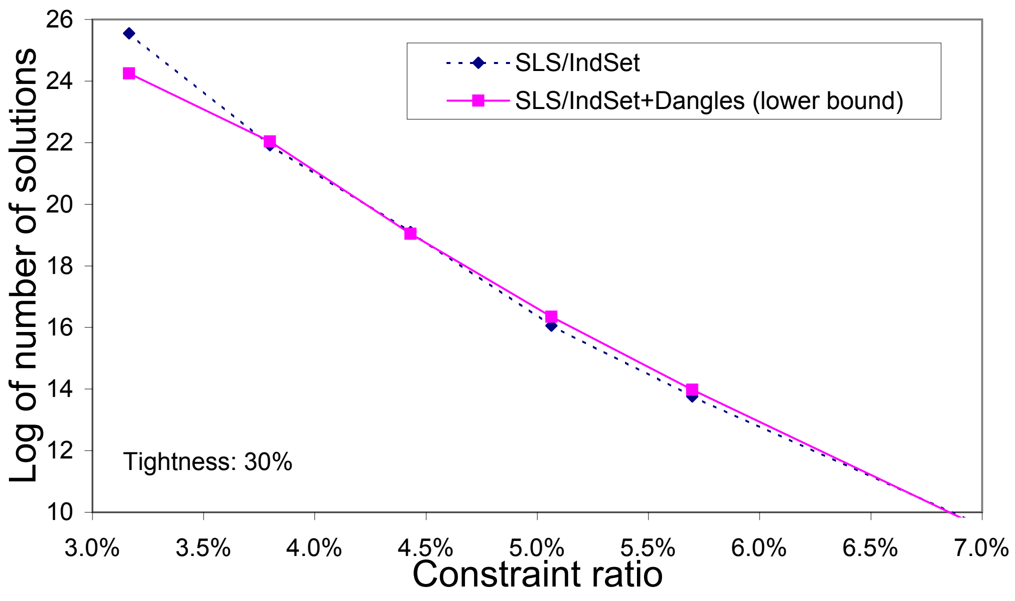

Figure 6.2: Estimated $\log_1 0$ of number of solutions found.

Another possible method for obtaining perhaps an even better lower bound is to perform backtrack-free search on each tree, using bundling [Haselböck, 1993][1]. Bundling is a technique for obtaining a solution to a CSP in which some variables are assigned a bundle of equivalent values. At each step in a backtrack-free search, we choose the largest bundle. Each element of the Cartesian product of the domain bundles gives a solution to the tree. Thus, the product of the sizes of the bundles chosen gives us a lower bound on the number of solutions in the tree. The product of these lower bounds of each tree again gives us a lower bound on the number of solutions for the original CSP.

[1]Note that static [Haselböck, 1993] and dynamic [Beckwith *et al.*, 2001] bundling on trees yield the same result.

6.2 Comparison to CYCLE-CUTSET decomposition

INDSET and INDSET+DANGLES are both special cases of CYCLE-CUTSET of Dechter [1987, 2003], which identifies a subset of vertices that, when instantiated, leave the rest of the constraint graph acyclic. However, both of these decompositions go further than the general CYCLE-CUTSET. INDSET goes further than leaving the graph acyclic, it leaves the graph with no edges at all. The particularity of INDSET+DANGLES with respect to CYCLE-CUTSET is a little less obvious.

In CYCLE-CUTSET, removing the cutset nodes leaves the constraint graph acyclic. The remaining graph may have multiple components, each a tree. A given tree may have more than one vertex adjacent to the cycle-cutset. For example, Figure 6.3 shows a cutset inside

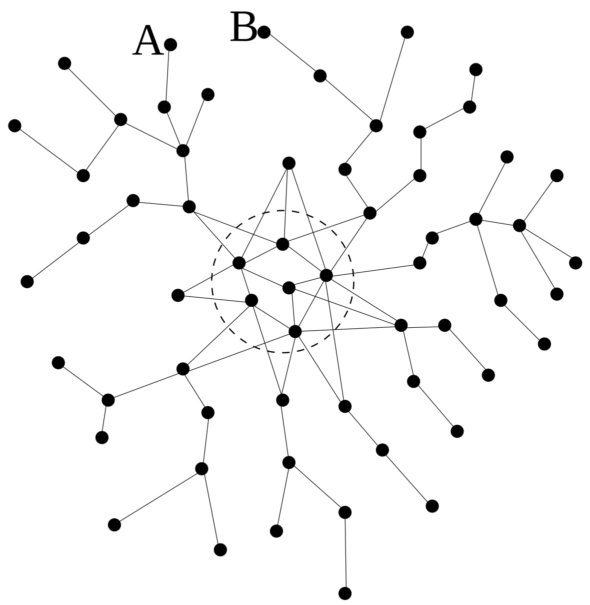

Figure 6.3: Vertices inside the circle form a cutset.

the circle. This could be a cutset found in C of the INDSET+DANGLES decomposition. Suppose an edge is added between vertices **A** and **B**. The tree containing **A** and **B** would then have two vertices adjacent to the cutset. In that case, the circled vertices still form a cycle-cutset, but not a INDSET+DANGLES cutset C.

In contrast, INDSET+DANGLES leaves trees, each of which has *at most* one vertex adjacent to the cutset. This fact results in the advantage of allowing us to ignore all but one vertex of the tree when performing search. On the other hand, in CYCLE-CUTSET, a tree that has even two vertices adjacent to the cutset greatly complicates matters.

The extra edge creates a cycle that causes difficulty, because we can no longer afford to only look at those vertices adjacent to the cutset, because there may be a cycle in the graph extending further out. In this case, *arc-consistency is not sufficient* to guarantee our independent choices for values of the neighborhood of the cutset.

Another issue to consider is that a tradeoff exists between connectivity and cutset size. INDSET leaves a graph with less connectivity than CYCLE-CUTSET, while the cutsets of INDSET are larger. Although we would like both small cutsets and low connectivity, we have a tradeoff between the two. It is worth exploring this tradeoff to determine the best balance between the two. In fact, work has been conducted exploring the other direction from CYCLE-CUTSET, i.e. finding and using cutsets that leave the graph with more connectivity with the benefit of having smaller cutsets (e.g., w-cutsets [Bidyuk and Dechter, 2004]). We are not aware of prior work that explores the tradeoff for connectivity below that of CYCLE-CUTSET.

6.3 Runtime variance

In this section we attempt to justify one aspect of our experimental procedures by examining the sources of runtime variation. Previously in this thesis, we have performed experiments primarily by generating a set of problem instances and running each algorithm (e.g., SLS and SLS/INDSET+DANGLES) once on each problem instance. That approach has the potential to hide the true source of runtime variation. Indeed, while the stochastic nature of the search method clearly causes runtime variation, the probability distribution of the runtime may also vary between problem instances.

To determine what amount of the runtime variation arises from the stochastic nature of the algorithm and what amount arises from differing problem instances, we conducted the following experiment. We generated 100 instances of Model B CSPs, with 80 variables, domain size = 10, constraint ratio = 22%, and tightness = 10%. These are solvable problems, and we measured the median runtime for SLS on these problems at 8.5 seconds. We executed each of SLS and SLS/IndSet+Dangles 100 times on each problem instance. Thus we obtain, for each algorithm, a matrix of runtimes, where each row contains 100 runtimes for a particular problem instance, as illustrated in Figure 6.4. We then computed

	trial 1	trial 2	trial 3	trial 4	$\cdots$
Instance 1	8.74	9.02	4.77	38.71	
Instance 2	27.01	27.89	4.97	4.63	$\cdots$
Instance 3	8.99	8.98	4.48	8.70	
Instance 4	4.10	53.60	17.16	8.72	
$\vdots$		$\vdots$			

Figure 6.4: 100×100 matrix of runtimes, in seconds.

the variances of the rows and the columns. Table 6.1 shows the average row variance and the average column variance for SLS and SLS/IndSet+Dangles.

Table 6.1: Average row and column variances.

	SLS	SLS/IndSet+Dangles
Row variance	67.83	24.02
Column variance	68.45	24.12

If the runtime within a row does not vary much, then we can conclude that the algorithm is consistent across runs (the runtimes of a deterministic algorithm would not vary at all within a row). Conversely, variation of the runtime between the rows is a result of both the stochasticity of the algorithm and of the differing problem instances. Therefore, we should expect the column variance to be greater than the row variance, which is, indeed, what Table 6.1 shows. Furthermore, the difference between the row variance and the column variance represents the variance due to the problem instances. From the table we can see that this difference is small. Therefore we conclude that runtime variation due to differing problem instances, while present, is insignificant compared to the variation due to the stochastic nature of the algorithm. Thus running each algorithm once on each problem instance does give a good approximation of running each algorithm multiple times on each instance. We performed this experiment for other values of constraint tightness and constraint ratio, which yielded similar results.

Additionally, because SLS with and without INDSET+DANGLES yielded approximately the same results in this regard (at different scales), we can conjecture that the above conclusion about the runtime variance is a property of SLS and is not affected by INDSET+DANGLES.

6.4 Clustered graphs

In order to verify that the decomposition technique is beneficial in the case of problems other than Model B random instances, we tested SLS/INDSET+DANGLES against SLS on structured problems. We used the *clustered graphs* of Hogg where "the search costs [is] typically much larger than for random graphs" [Hogg, 1996]. Hogg showed that the phase transition occurs at a lower constraint ratio than Model B problems. We generated problems with 80 variables and 10 values per variable, while varying constraint tightness and constraint ratio. We show the median runtimes in Figures 6.5 and 6.6, using a cutoff criterion of 120 seconds.

In general, the combination of SLS and INDSET+DANGLES results in an improvement on clustered graphs. The improvement tends to increase as the number of constraints in-

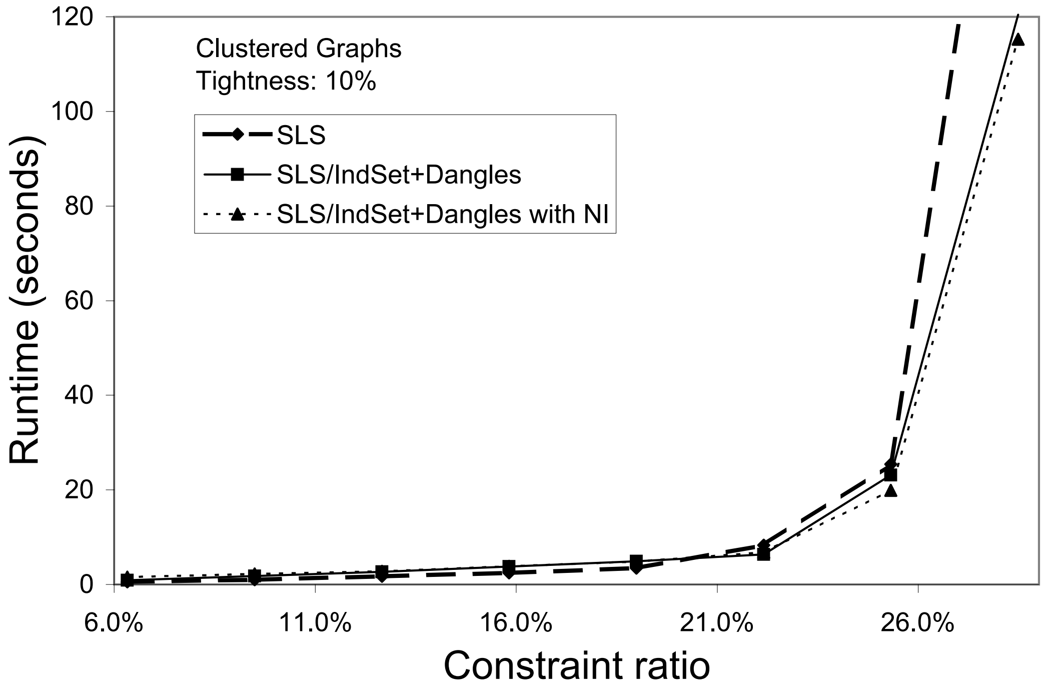

Figure 6.5: Runtime on clustered problems with 10% tightness.

creases to the right (i.e., towards the region of the phase transition). In fact, Figure 6.5 shows improvement only for the higher constraint-ratio values. Otherwise, and for the lower constraint-ratio values, SLS alone sometimes performs slightly better than in combination with INDSET+DANGLES. Note that these problems with lower constraint-ratio are relatively easy to solve, and thus do not matter as much as those in the region of the phase transition.

As stated above, CSPs with clustered graphs are more difficult to solve than Model B problems for the same parameters. For 50% tightness we were not able to generate sparse *connected* clustered problems. The were only able to produce connected graphs at a higher constraint-ratio at which SLS never returned a solution.

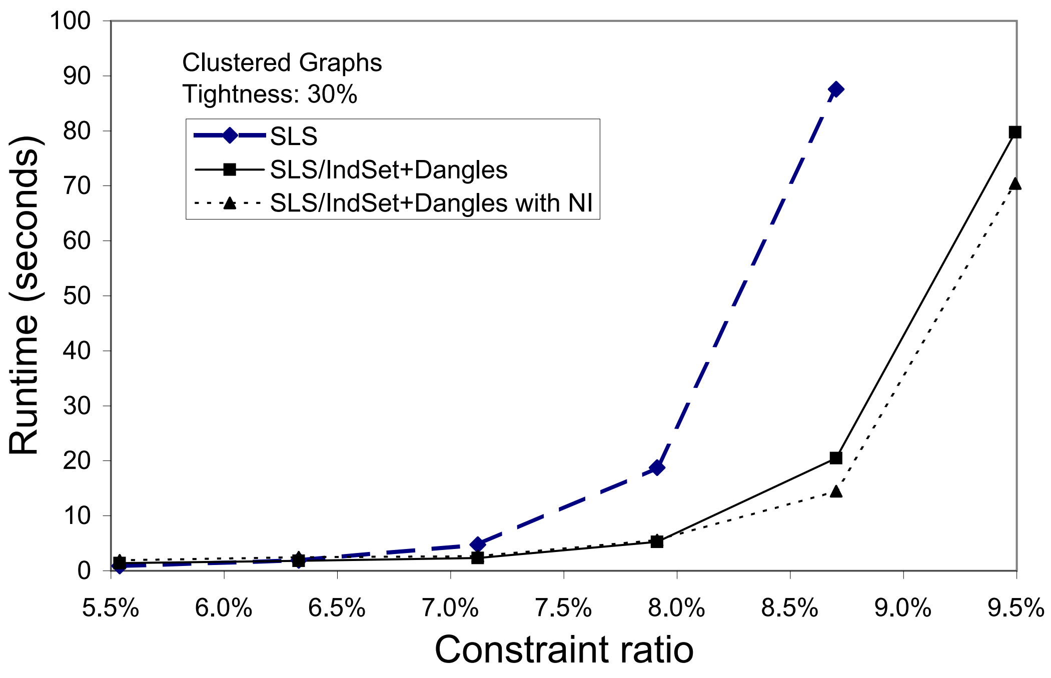

Figure 6.6: Runtime on clustered problems with 30% tightness.

Summary

Useful information can be gained by examining the independent set and dangling tree structure of the constraint graph of a CSP instance. For example, lower bounds on the number of solutions may be computed. Also, we argued that the independent-set decomposition has a close relationship with the cycle-cutset decomposition, but results in stronger cutsets at the cost of having larger cutsets. The runtime variance was found to be due primarily to the stochastic nature of SLS. And INDSET was found to be beneficial on CSPs with clustered graphs.

Chapter 7

Recursive Decompositions

We noticed that the subproblem $\bar{I}$ identified by INDSET sometimes consists of more than one connected component as shown in Figure 7.1. This observation lead to the idea of

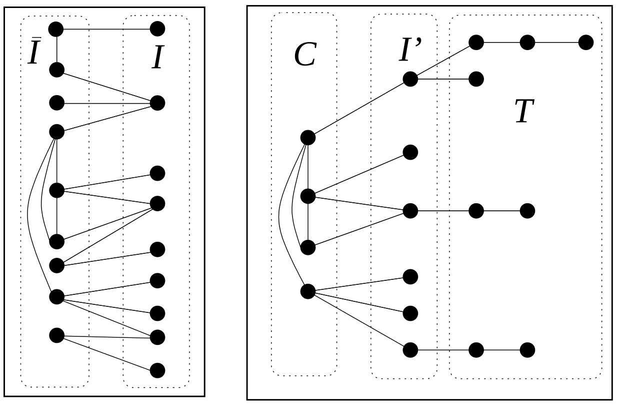

Figure 7.1: Dangling trees.

applying INDSET recursively on connected components of C. Below we introduce this mechanism and a similar one that operates on maximal cliques. While both of these decompositions themselves appear to be interesting, we are as of yet unable to determine a practical use for them. Further investigation may prove fruitful.

7.1 Recursive INDSET

At first glance, the notion of applying INDSET recursively, which we denote RECINDSET, appears as a promising one. We perform this decomposition by first finding an independent set I in the constraint graph of a CSP, then removing I from the graph. The remaining subgraph may contain a number of connected components. Figure 7.2 shows such a situation schematically, where we find an independent set I and the remaining graph has three components A, B, and C.

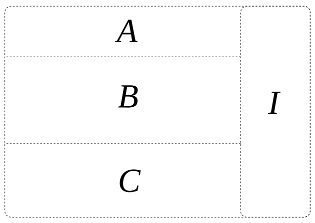

Figure 7.2: First step.

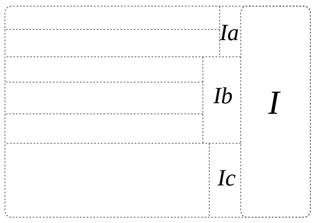

Figure 7.3: Second step.

We then apply RECINDSET to each of the connected components A, B, and C, first identifying, in each of them, a maximal independent set, as shown in Figure 7.3. We repeat this process recursively, thus building a tree such as the one shown in Figure 7.4. The

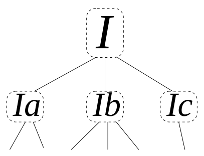

Figure 7.4: RECINDSET tree.

stopping criterion is when no independent set with more than one vertex remains in the graph, which occurs in the case of a clique. Each branching node in the resulting tree 'contains' the independent set found at a given step of RECINDSET, and each child node contains a connected component remaining after removing the independent set from the graph. The tree yields a partition of the set of variables of the CSP. The branching nodes are maximal independent sets and leaf nodes are cliques.

For example, Figure 7.5 shows an example of RECINDSET. The process proceeds from

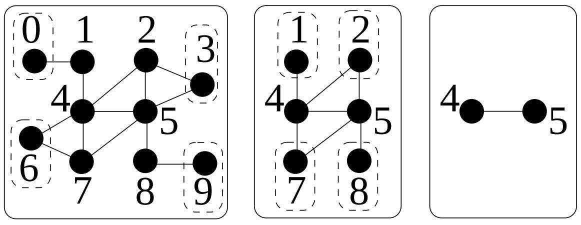

Figure 7.5: RECINDSET example.

left to right in the figure. The leftmost graph shows the original. In each graph, vertices of a maximal independent set are circled, which are removed from the graph immediately to the right. This is repeated until we are left with a clique in the rightmost graph. Figure 7.6 shows the result by redrawing the original graph so that the clique is on the left. In this

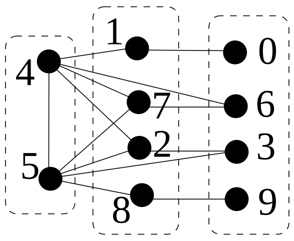

Figure 7.6: RECINDSET example result.

example, the RECINDSET tree has no branching.

We examined the possibility of using RECINDSET in the same way we used INDSET in combination with local search. Intuitively, we could solve the decomposition tree at the leaves, and propagate upward. Recall that in SLS/INDSET there is a kind of backtracking as search moves from $\bar{I}$ to I. This same type of backtracking will occur as we perform search on the RECINDSET tree from the leaves up. For every assignment of values to the variables in the leaves, each level of the tree will have to propagate to all the levels above it, starting with the deepest level. If any domain becomes annihilated in this process, then the domains must be restored and another local-search move will begin. This processing overhead (quadratic in the depth of the tree) is too great to gain a performance increase from this use of RECINDSET. Further investigation may uncover a way to utilize RECINDSET in this manner.

We tested the possibility of using this tree to provide a (static) variable ordering for backtrack search. First, we sort the set of variables at each RECINDSET tree node by decreasing degree (breaking ties lexigraphically). We then build the variable ordering by performing a post-order traversal of the tree. As we visit each node, we add the node's variables to the end of the variable ordering. As a result of this ordering, search first instantiates the variables in the cliques, then works its way to variables in the first independent set I. For the example of Figure 7.6, the order would be: $\{4, 5, 2, 7, 1, 8, 3, 6, 0, 9\}$. Note that search, using forward checking, will be finished after considering variable 8 because all of the constraints will have been considered.

We compared the performance of backtrack search with forward checking using this variable ordering (BT/RECINDSET) to that of the same search mechanism using the static, decreasing-degree ordering, which we denote BT. We conducted the tests on CSP instances with 24 variables, 12 values, and varying constraint tightness and constraint ratio around the region of the phase transition. We show the runtime cumulative-distributions for 30% tightness at the phase transition, using 1000 problem instances for Model B problems (Figure 7.7) and clustered problems (Figure 7.8). These figures show that BT clearly outperforms BT/RECINDSET. The difference is more pronounced for clustered graphs. The poor performance of BT/RECINDSET is likely due to the fact that the ordering resulting from RECINDSET places all the variables of an independent set consecutively in the search tree. Having strings of mutually non-adjacent variables in the ordering is likely to cause unnecessary thrashing. This problem, though obvious to us now, was not immediately clear to us when first examining the potential benefits of RECINDSET. Also, further investigation may reveal that other search techniques do benefit from this type of variable ordering.

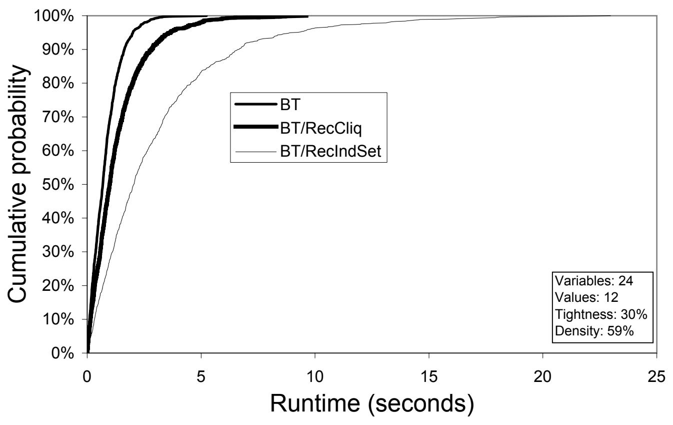

Figure 7.7: Recursive decompositions as variable orderings on Model B problems.

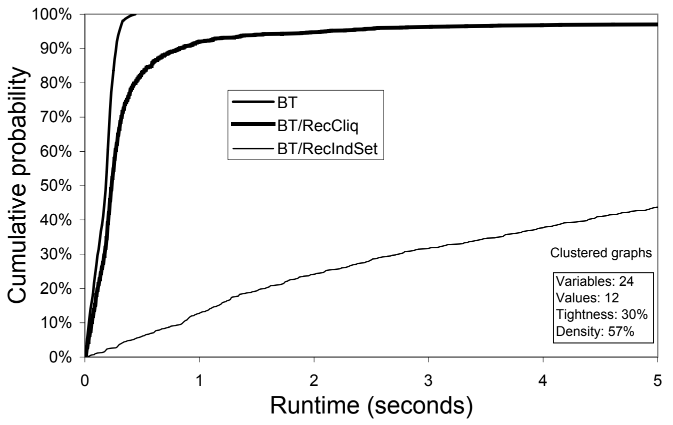

Figure 7.8: Recursive decompositions as variable orderings on clustered problems.

7.2 Recursive decomposition using cliques

The above negative result led to the idea of applying the same recursive decomposition in a complementary way. Instead of repeatedly finding and removing independent sets, we repeatedly find and remove cliques. We denote this decomposition as RECCLIQ. Figure 7.9 provides an example of RECCLIQ. The process proceeds from left to right in the figure. The leftmost graph shows the original. In each graph, vertices of a maximal clique are circled, which are removed from the graph immediately to the right. This is repeated until we are left with an independent set in the rightmost graph. We obtain the cliques as a by-product of CLIQUEREMOVAL of [Boppana and Halldórsson, 1990], which we used to identify max-

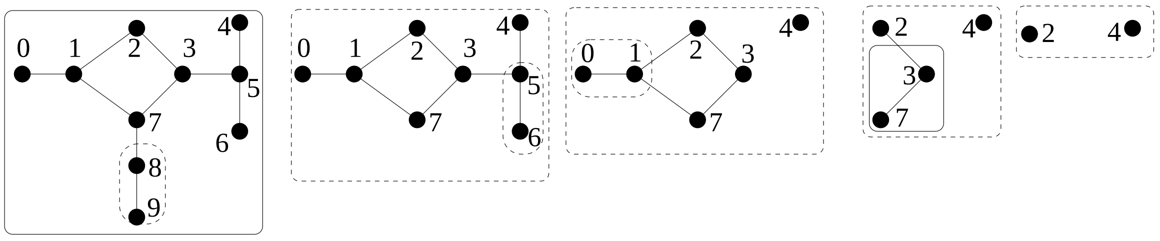

Figure 7.9: RECCLIQ example.

imal independent sets. Some leaves of the tree from RECCLIQ contain independent sets, and all the other nodes of the tree contain cliques. Also, the union of all the independent sets in the leaves forms an independent set of the original graph. We obtain the variable ordering in the same manner as before, except by performing a pre-order traversal of the tree, and then moving the independent set to the end of the ordering. As shown in Figure 7.7, this ordering performed significantly better than the one obtained with RECINDSET, but still did not perform as well as the simple, and widely used, decreasing-degree heuristic.

Finally, note that in BT/RECINDSET, after making an assignment to the cliques, the remaining graph consists of interconnected independent sets (i.e., it is a k-partite graph). An interesting area for future research is to determine whether solving bipartite (or, better, k-partite) graphs can be efficiently done, or at least to design a strategy that heuristically exploits such a topology during search.

Summary

In this chapter, we presented methods for recursive decomposition, one by applying INDSET recursively and a complementary decomposition by recursively extracting cliques. Practical uses for these decompositions are yet to be found. We suggest that further investigation may provide significant results if a method for solving k-partite CSPs is discovered.

Chapter 8

Conclusions

Our results demonstrate that finding a large independent set in the constraint graph and using it to decompose the CSP improves the performance of solving the CSP. An additional benefit of our approach over other decomposition techniques is that it inherently provides us with many solutions, at no extra cost beyond that incurred by the search process, and the multiple solutions are represented in a compact form. We can gain further improvement by identifying and extracting trees that dangle off the independent set. Additional information can be reaped from these trees, such as estimating a lower bound on the number of solutions obtained.

We provided analysis of INDSET and explored some new avenues for improving it. We determined that runtime does not vary significantly due to changing between problems with the same parameters. The use of clustered graphs revealed that INDSET is beneficial not only for uniform random problems, but also for problems having clustered graphs. We generalized the application of DANGLES. We can find dangles on any variable. Searching for dangles before applying INDSET can be done efficiently and can only yield more dangles. Furthermore, DANGLES can be applied to any search technique. Some other avenues yielded negative results. Two applications of using recursive decomposition as a variable ordering for systematic search yielded results not as good as simple decreasing-degree ordering.

Future investigations include the following:

- Explore the use of INDSET on CSPs with non-binary constraints.
- A possible modification of SLS/INDSET for future investigation would be to move the finding of the independent set inside the first **for** loop of Algorithm 2, which would mean that a new independent set would be used for each restart.
- Evaluate the search techniques presented in this thesis on more types of problem instances, such as real-world problems, other structured problems, or k-regular graphs.
- Compare the 'percent solved' curves (from local search) with the curve of actual percentage of instances solvable, on small problems using a complete systematic search. While it is infeasible to do make this comparison on large problems, it may be interesting to see the comparison on small problems.
- Explore ways of counting the number of solutions in a tree-structured CSP, and/or ways to represent the set of all solutions of a tree.
- Explore the possibility of increasing the robustness of a solution, regarding the variables in $\bar{I}$ (or C). The current technique leaves $\bar{I}$ with an inflexible solution. On a related theme, it may be possible to 'fatten up' the domains of I in a solution during search. At present, we stop as soon as we find the first set of solutions.
- INDSET is concerned with only the structure of the constraint graph. Perhaps additional similar structure can be found when the contents of the constraints are considered, such as finding independency in the micro-structure of the problem.
- Explore the possibilities of developing more sophisticated ways to integrate INDSET with search techniques to provide further improvements.

- Explore the tradeoff between stronger and weaker cutsets.
- Further investigations may likely reveal an dominance of the heuristic `PrefRelax` on denser graphs. Also, heuristics for ordering the constraints may improve performance of `PrefRelax` as well as `Some`.
- As we have shown, the improvement due to NI increases as we approach the region of the phase transition, hence this research direction may benefit from further investigations.
- While both of the recursive decompositions (RECINDSET and RECCLIQ) appear to be quite interesting in themselves, we are as of yet unable to determine a practical use for them. Further investigation may prove fruitful. For example:
- We suggest future work to find search techniques that exploit a k-partite graph structure, which would be of great benefit in combination with RECINDSET.

This thesis has furthered the current progress in the field of constraint programming. We have provided enhancements to solving techniques that reduce the cost of solving CSPs, and, at the same time, return multiple solutions.

Appendix A

BinaryCSP Class Index

These apendices provide a documentation for the C++ source code that was developed for this thesis.

A.1 BinaryCSP Class List

Below are the classes, structs, unions and interfaces with brief descriptions. A full description of each is given in Appendix B at the page numbers shown on the right.

A.2 Directory Reference

Files

Below are the files with brief descriptions. Full documentation for the files is given in Appendix C.

- file **BinaryCSP.cpp**

  Binary CSP implementation.

- file **BinaryCSP.h**

  A binary CSP.

- file **JDT.cpp**

  Joint Discrimination Tree implementation.

- file **JDT.h**

  A generic Joint Discrimination Tree.

- file **main.h**

  General definitions.

Appendix B

BinaryCSP Class Documentation

This appendix contains the documentation for all of the C++ classes developed in the implementation for this thesis.

B.1 BinaryCSP Class Reference

A binary CSP.

```
#include <BinaryCSP.h>
```

Public Member Functions

- **BinaryCSP** (int num_vars)

  Creates a CSP with specified number of variables and no constraints.

- **BinaryCSP** (unsigned int num_vars, unsigned int domain_size, float density, float tightness)

  Creates a random (model-b) CSP.

- void **generate_rand** (unsigned int num_vars, unsigned int domain_size, float density, float tightness)

  Creates a random (model-b) CSP.

- void **generate_rand** (unsigned int num_vars, unsigned int domain_size, unsigned int num_constraints, float tightness)

  Creates a random (model-b) CSP.

- void **generate_clustered** (unsigned int num_vars, unsigned int domain_size, unsigned int num_constraints, float tightness, float p)

  Creates a random clustered CSP.

- void **generate_empty** (int num_vars, int domain_size)

  Creates a CSP with no constraints.

- bool **generate_constraint** (**variable_t** var1, **variable_t** var2, float tightness)

  Attempts to create a new constraint between 2 vars.

- unsigned int **get_num_vars** () const

  Returns number of variables in the CSP.

- unsigned int **numConstraints** () const

  Returns number of constraints in the CSP.

- unsigned int **degree** (**variable_t** var) const

Returns the number of constraints on a variable.

- unsigned int **degree_exclude** (**variable_t** var, const std::vector< **variable_t** > &exclude_-list) const

  Returns the number of constraints on a variable in a subgraph.

- template<typename T> void **get_variables** (T &result) const

  Fills result with a list of all variables (i.e., 0 through N-1).

- void **get_neighbors** (const std::vector< **variable_t** > &S, std::vector< **variable_t** > &result) const

  Returns neighbors of a set of variables S.

- void **get_neighbors** (**variable_t** var, std::vector< **variable_t** > &result) const

  Returns neighbors of a variable.

- void **get_neighbors** (**variable_t** var, const std::vector< **variable_t** > &S, std::vector< **variable_t** > &result)

  Returns neighbors of a variable intersected with set S.

- void **get_non_neighbors** (**variable_t** var, const std::vector< **variable_t** > &S, std::vector< **variable_t** > &result)

  Returns non-neighbors of a variable intersected with S.

- void **get_domain** (**variable_t** var, std::vector< **value_t** > &result) const

Returns domain of a variable.

- void **set_domain** (**variable_t** var, const std::vector< **value_t** > &dom)

  Replaces the domain of a variable.

- unsigned int **get_domain_size** (**variable_t** var) const

  Returns a variable's domain size.

- bool **is_consistent** (**variable_t** var1, **value_t** val1, **variable_t** var2, **value_t** val2) const

  Returns whether the given tuple is allowed.

- bool **duplicateConstraints** () const

  Checks whether there is more than one constraint between any pair of variables.

- void **removeTuple** (**variable_t** var1, **variable_t** var2, **value_t** val1, **value_t** val2)

  Causes the given tuple to be forbidden by the CSP.

- bool **is_supported** (**variable_t** var1, **value_t** val1, **variable_t** var2) const

  Checks whether the val1 in var1 is supported by any value in var2.

- void **inequality_constraint** (**variable_t** var1, **variable_t** var2)

  Adds mutex constraint between two variables.

- bool **is_solution** (const std::vector< **value_t** > &solution)

Verifies a solution.

- int **get_broken_constraints** (const std::vector< **value_t** > &solution, std::vector< **VarPair** > &result)

  Returns constraints broken by the "solution".

- int **numBrokenConstraints** (const std::vector< **value_t** > &solution, **variable_t** var)

  Returns the number of broken constraints participated in by a specific variable.

- int **numBrokenConstraints** (const std::vector< **value_t** > &solution)

  Returns the number of broken constraints given a "solution".

- int **numBrokenConstraintsIS_countsome** (const std::vector< **value_t** > &solution, const std::vector< **variable_t** > &independent_set, const std::vector< **variable_t** > &vertex_cover)

  Returns the number of broken constraints using heuristic SOME.

- int **numBrokenConstraintsIS_countall** (const std::vector< **value_t** > &solution, const std::vector< **variable_t** > &independent_set, const std::vector< **variable_t** > &vertex_cover)

  Returns the number of broken constraints using heuristic ALL.

- int **numBrokenConstraintsIS_PrefRelax** (const std::vector< **value_t** > &solution, const std::vector< **variable_t** > &independent_set, const std::vector< **variable_t** > &vertex_cover)

Returns the number of broken constraints using heuristic PrefRelax.

- int **numBrokenConstraintsIS_ignore** (const std::vector< **value_t** > &solution, const std::vector< **variable_t** > &independent_set, const std::vector< **variable_t** > &vertex_-cover)

  Returns the number of broken constraints using heuristic NONE.

- void **tree** (const std::vector< **variable_t** > &cycle_cutset, const std::vector< **variable_-t** > &tree_vars, const std::vector< **value_t** > &partial)

  Tree algorithm described in Dechter's book.

- void **sort_tree_vars** (std::vector< **variable_t** > &tree_vars)

  Helper function for **tree()**(p. 83).

- int **solveFC** (std::vector< **value_t** > &solution, bool count_solutions=false)

  Solves the CSP using Forward Checking (FC).

- int **solveFC** (const std::vector< **variable_t** > &order, std::vector< **value_t** > &o_-solution, bool count_solutions=false)

  Solves the CSP using Forward Checking (FC) with static variable ordering.

- int **solveFC_DynamicOrder** (std::vector< **value_t** > &solution, bool count_solutions=false)

  Solves the CSP using Forward Checking (FC) with dynamic variable ordering.

- int **solveFC_IS** (const std::vector< **variable_t** > &order, int count, std::vector< **value_t** > &o_solution, bool count_solutions=false)

  Solves the CSP using Forward Checking and IndSet.

- int **solveWalkSAT** (std::vector< **value_t** > &solution)

  Solve the CSP using WalkSAT (Min-conflict random walk).

- bool **solveSLS** (std::vector< **value_t** > &solution, double fMaxTime, unsigned int &o_nFlips)

  Solve the CSP using SLS.

- int **solveWalkSatIS** (const std::vector< **variable_t** > &independent_set, std::vector< list< **value_t** > > &solution) const

  Solve the CSP using SLS/IndSet.

- bool **solveSLS_IS_dangles** (int(BinaryCSP::∗numBroken)(const std::vector< **value_-t** > &solution, const std::vector< **variable_t** > &independent_set, const std::vector< **variable_t** > &vertex_cover), const std::vector< **variable_t** > &cutset, const std::vector< **variable_t** > &independent_set, std::vector< list< **value_t** > > &solution, double fMaxTime, unsigned int &o_nFlips, unsigned int &o_nRestarts)

  Solve CSP using SLS/IndSet+Dangles.

- void **get_independent_set_Ramsey** (const std::vector< **variable_t** > &G, std::vector< **variable_t** > &result, std::vector< **variable_t** > &clique)

  Obtains an independent set from the subgraph G using the Ramsey algorithm.

- void **getIndependentSet_CliqueRemoval** (const std::vector< **variable_t** > &G, std::vector< **variable_t** > &result)

  Obtains an independent set from the subgraph G using CliqueRemoval.

- void **getClique_IndependentSetRemoval** (const std::vector< **variable_t** > &G, std::vector< **variable_t** > &result)

  Obtains a clique from subgraph G using IndependentSetRemoval algorithm.

- bool **isIndependentSet** (const std::vector< **variable_t** > &S)

  Determines whether S is an independent set in the constraint graph.

- unsigned int **expandSolution** (const std::vector< **variable_t** > &independent_set, const std::vector< **variable_t** > &vertex_cover, const std::vector< **value_t** > &partial_-solution, std::vector< list< **value_t** > > &solution) const

  Expands a partial solution on I-bar to the variables in I.

- bool **getSubproblem** (const std::vector< **variable_t** > &S, **BinaryCSP** &result) const

  Obtains the sub-problem induced by the subgraph S.

- void **getSubproblem** (const std::vector< **variable_t** > &S, const std::vector< **value_-t** > &solution, **BinaryCSP** &result)

  Obtains the sub-problem induced by the subgraph S.

- int **freedom** (**variable_t** var, const std::vector< **value_t** > &solution)

  Returns number of values in the variable that are consistent with the rest of the solution.

- void **print** ()

  Outputs the CSP to standard out.

- void **printshort** ()

  Outputs the CSP to standard out without listing the constraint tuples.

- bool **getTree** (**variable_t** var1, **variable_t** var2, std::vector< **variable_t** > &result)

  Returns a tree containing var2 and dangling off var1.

- void **getDanglingTrees** (**variable_t** var, std::vector< **variable_t** > &result)

  Get set of variables consisting of subtrees "dangling off" var.

- void **getDangles** (const std::vector< **variable_t** > &ind_set, std::vector< **variable_t** > &o_cutset, std::vector< **variable_t** > &o_Iprime)

  Returns the variables dangling off ind_set.

- void **getDangles** (std::vector< **variable_t** > &result)

  Returns the dangles in entire graph.

- void **sortDecreasingDegree** (std::vector< **variable_t** > &varlist)

  Sorts variables in order of decreasing degree.

- void **getCycleCutset_DecreasingOrder** (std::vector< **variable_t** > &result)

  Finds a cycle cutset using decreasing-degree heuristic.

- void **getCycleCutset_DecreasingOrder_checkParticipate** (std::vector< **variable_-t** > &result)

  Finds a cycle cutset, checking whether a var participates in a cycle.

- void **getCycleCutset_RemoveMostCycles** (std::vector< **variable_t** > &result)

  Finds a cycle-cutset by repeatedly removing the variable that participates in the most cycles.

- void **getCycleCutset_ISMethod** (std::vector< **variable_t** > &result)

  Finds a cycle-cutset by finding an Independent Set.

- void **getComponent** (**variable_t** var, std::vector< **variable_t** > &result)

  Returns the component of the graph containing the variable.

- void **getComponent_exclude** (**variable_t** var, std::vector< **variable_t** > &result, const std::vector< **variable_t** > &exclude_list)

  Returns the component containing the variable, in the subgraph induced by removing the exclude_list.

- void **getComponent_subgraph** (**variable_t** var, std::vector< **variable_t** > &result, const std::vector< **variable_t** > &subgraph)

Returns the component containing the variable, in the induced subgraph.

- int **getComponents_subgraph** (const std::vector< **variable_t** > &subgraph, std::vector< std::vector< **variable_t** > > &result)

  Returns all components in the induced subgraph.

- unsigned int **nComponents** ()

  Returns the number of components in the graph.

- unsigned int **nComponents_exclude** (const std::vector< **variable_t** > &exclude_-list)

  Returns the number of components in the subgraph induced by removing exclude_list.

- bool **bParticipatesInACycle** (**variable_t** var)

  Does var participate in any cycle.

- bool **bParticipatesInACycle_exclude** (**variable_t** var, const std::vector< **variable_-t** > &exclude_list)

  Does var participate in any cycle in the subgraph induced by removing exclude_list.

- unsigned int **nCyclesParticipated** (**variable_t** var)

  Returns number of cycles containing var.

- bool **isAcyclic_exclude** (const std::vector< **variable_t** > &exclude_list)

  is the graph (excluding the vars in the exclude list) acyclic

- bool **isConnected** ()

  is the graph connected

- unsigned int **nEdges** (const std::vector< **variable_t** > &varlist)

  Returns the number of edges in the subgraph varlist.

- const **BinaryCSP::Constraint** ∗ **getConstraintBetween** (**variable_t** var1, **variable_-t** var2)

  Returns a pointer to the constraint between var1 and var2 (if one exists).

- bool **Revise** (**variable_t** var1, **variable_t** var2)

  reduces the domain of var1, removing values not supported by var2

- bool **AC3** ()

  Enforce arc consistency.

- void **save** (string outputfile)

  Write the CSP to a file.

- bool **load** (string inputfile)

  Write a CSP from a file.

- void **loadChineese** (string inputfile, unsigned int nVars, unsigned int domainsize, unsigned int nConstraints, unsigned int nNogoods)

Loads a Chineese problem.

- void **loadChineeseConstraint** (istream &fin, unsigned int nNogoods, unsigned int nValues)

  Loads a constraint from a Chineese problem.

- void **loadChineeseTuple** (istream &fin, **BinaryCSP::Constraint** &C)

  Loads a constraint tuple from a Chineese problem.

- void **adjustForBundling** (const std::vector< **variable_t** > &cutset, std::vector< **variable_t** > &independent_set, unsigned int &o_vars_eliminated, unsigned int &o_-num_bundles, float &o_ave_bundle_size)

  Optimizes the CSP using static bundling with the independent set.

- void **recursiveDecompose** (const std::vector< **variable_t** > &subgraph, **recDec-Node** &node)

  Performs RecIndSet and returns the tree through the argument node.

- int **sortRecDec** (std::vector< **variable_t** > &result)

  Obtains RecIndSet variable ordering.

- void **recursiveCliqueDecompose** (const std::vector< **variable_t** > &subgraph, **rec-DecNode** &node)

  Obtains RecCliq decomposition.

- int **sortCliqueRecDec** (std::vector< **variable_t** > &varlist)

  Obtains RecCliq variable ordering.

Classes

- struct **Constraint**

  A binary constraint.

- struct **Variable**

  A variable in the Binary CSP.

B.1.1 Detailed Description

A binary CSP.

In order to minimize memory storage, each constraint is stored in the first of the two variables, lexicographically

Difference between **Variable**(p. 103) and variable_t

Definition at line 183 of file BinaryCSP.h.

B.1.2 Member Function Documentation

B.1.2.1 void BinaryCSP::adjustForBundling (const std::vector< variable_t > & *cutset*, std::vector< variable_t > & *independent_set*, unsigned int & *o_vars_eliminated*, unsigned int & *o_num_bundles*, float & *o_ave_bundle_size*)

Optimizes the CSP using static bundling with the independent set.

NOTE: may adjust the independent set

Definition at line 3795 of file BinaryCSP.cpp.

References JDT::build(), get_domain_size(), and JDT::getBundles().

B.1.2.2 unsigned int BinaryCSP::degree_exclude (variable_t *var*, const std::vector< variable_t > & *exclude_list*) const

Returns the number of constraints on a variable in a subgraph.

Does not count the constraints with variables contained in the exclude list.

Definition at line 500 of file BinaryCSP.cpp.

References degree().

Referenced by bParticipatesInACycle_exclude().

B.1.2.3 unsigned int BinaryCSP::expandSolution (const std::vector< variable_t > & *independent_set*, const std::vector< variable_t > & *vertex_cover*, const std::vector< value_t > & *partial_solution*, std::vector< list< value_t > > & *solution*) const

Expands a partial solution on I-bar to the variables in I.

Precondition:

- independent_set is an independent set of the CSP
- partial_solution is a std::vector of length equal to the number of variables in the CSP, with a value for each variable not in the independent set. These values must be consistent.

Postcondition: result will be filled with an expanded solution giving all the possibilities for the variables inside the independent set

Definition at line 2501 of file BinaryCSP.cpp.

References get_neighbors(), and is_consistent().

Referenced by solveWalkSatIS().

B.1.2.4 int BinaryCSP::freedom (variable_t *var*, const std::vector< value_t > & *solution*)

Returns number of values in the variable that are consistent with the rest of the solution.

i.e., given a solution, if we vary the variable var, how many values don't conflict with the rest of the solution

Definition at line 2397 of file BinaryCSP.cpp.

References is_consistent().

B.1.2.5 bool BinaryCSP::generate_constraint (variable_t *var1*, variable_t *var2*, float *tightness*)

Attempts to create a new constraint between 2 vars.

Constraint(p. 101) created with specified tightness

Returns false on failure. Will fail if a constraint between the vars already exists.

Definition at line 116 of file BinaryCSP.cpp.

References BinaryCSP::Constraint::tuples, and BinaryCSP::Constraint::var.

Referenced by generate_clustered().

B.1.2.6 void BinaryCSP::get_independent_set_Ramsey (const std::vector< variable_t > & *G*, std::vector< variable_t > & *result*, std::vector< variable_t > & *clique*)

Obtains an independent set from the subgraph G using the Ramsey algorithm.

Returns the set in result.

Definition at line 2844 of file BinaryCSP.cpp.

References get_neighbors(), and get_non_neighbors().

Referenced by getClique_IndependentSetRemoval(), and getIndependentSet_Clique-Removal().

B.1.2.7 void BinaryCSP::getComponent_exclude (variable_t *var*, std::vector< variable_t > & *result*, const std::vector< variable_t > & *exclude_list*)

Returns the component containing the variable, in the subgraph induced by removing the exclude_list.

exclude_list must not contain var

Definition at line 3001 of file BinaryCSP.cpp.

References get_neighbors().

Referenced by getComponent_subgraph(), isAcyclic_exclude(), and nComponents_-exclude().

B.1.2.8 void BinaryCSP::getComponent_subgraph (variable_t *var*, std::vector< variable_t > & *result*, const std::vector< variable_t > & *subgraph*)

Returns the component containing the variable, in the induced subgraph.

exclude_list must contain var

Definition at line 3042 of file BinaryCSP.cpp.

References get_variables(), and getComponent_exclude().

Referenced by getComponents_subgraph().

B.1.2.9 void BinaryCSP::getDangles (const std::vector< variable_t > & *ind_set*, std::vector< variable_t > & *o_cutset*, std::vector< variable_t > & *o_Iprime*)

Returns the variables dangling off ind_set.

Input: ind_set - an independent set Output: o_cutset - the resulting cutset minus the dangles o_Iprime - the resulting independent set minus the dangles

Definition at line 1307 of file BinaryCSP.cpp.

References get_neighbors(), get_variables(), and getDanglingTrees().

B.1.2.10 void BinaryCSP::getSubproblem (const std::vector$<$ variable_t $>$ & *S*, const std::vector$<$ value_t $>$ & *solution*, BinaryCSP & *result*)

Obtains the sub-problem induced by the subgraph S.

Precondition, 1) solution is a valid solution of this CSP, 2) S is a subset of the variables of this CSP

Postcondition: result is the subproblem containing the variables in S, with values and constraints modified by the restriction that the resulting CSP is allowed only those solutions that do not conflict with the other variables (not in S) in this original CSP with the given solution.

Definition at line 2658 of file BinaryCSP.cpp.

References is_consistent(), and vars.

B.1.2.11 bool BinaryCSP::getSubproblem (const std::vector$<$ variable_t $>$ & *S*, BinaryCSP & *result*) const

Obtains the sub-problem induced by the subgraph S.

Precondition: S is a subset of the variables of this CSP

Postcondition: result is the subproblem containing the variables in S, NOTE that the variables are renumbered. If S = {7, 3, 19} then they will be variables {1, 2, 3}, respectively, in the resulting sub-problem. Returns false if any variable's domain is annihilated.

Definition at line 2564 of file BinaryCSP.cpp.

References get_variables(), is_supported(), and vars.

Referenced by solveWalkSatIS().

B.1.2.12 bool BinaryCSP::getTree (variable_t *var1*, variable_t *var2*, std::vector< variable_t > & *result*)

Returns a tree containing var2 and dangling off var1.

Input: two adjacent variables var1 and var2.

Output: if the removal of var1 from the graph leaves var2 in a component which is a tree, then result returns the set of variables in that tree and getTree returns true. Otherwise, getTree returns false. this function is used to detect trees that are "dangling" off var1.

Definition at line 571 of file BinaryCSP.cpp.

References get_neighbors().

Referenced by getDanglingTrees().

B.1.2.13 int BinaryCSP::numBrokenConstraintsIS_ignore (const std::vector< value_t > & *solution*, const std::vector< variable_t > & *independent_set*, const std::vector< variable_t > & *vertex_cover*)

Returns the number of broken constraints using heuristic NONE.

Ignores constraints between I and I-bar

Definition at line 713 of file BinaryCSP.cpp.

References is_consistent(), and BinaryCSP::Constraint::var.

B.1.2.14 int BinaryCSP::numBrokenConstraintsIS_PrefRelax (const std::vector< value_t > & *solution*, const std::vector< variable_t > & *independent_set*, const std::vector< variable_t > & *vertex_cover*)

Returns the number of broken constraints using heuristic PrefRelax.

Uses the prefered relaxation (using lexicographic order) according to paper by Ulrich Junker(2004).

Definition at line 845 of file BinaryCSP.cpp.

References is_consistent(), and BinaryCSP::Constraint::var.

B.1.2.15 bool BinaryCSP::Revise (variable_t *var1*, variable_t *var2*)

reduces the domain of var1, removing values not supported by var2

used for arc-consistency.

Definition at line 3390 of file BinaryCSP.cpp.

References is_supported().

Referenced by AC3().

B.1.2.16 int BinaryCSP::solveFC (const std::vector< variable_t > & *order*, std::vector< value_t > & *o_solution*, bool *count_solutions* = `false`)

Solves the CSP using Forward Checking (FC) with static variable ordering.

Returns solution through the parameter solution. If count_solutions is false, return value true: solved, false: no solution. If count_solutions is true, return value indicates number of solutions to the CSP

Definition at line 2149 of file BinaryCSP.cpp.

References is_consistent().

B.1.2.17 int BinaryCSP::solveFC (std::vector< value_t > & *solution*, bool *count_solutions* = `false`)

Solves the CSP using Forward Checking (FC).

Returns solution through the parameter solution. If count_solutions is false, return value true: solved, false: no solution. If count_solutions is true, return value indicates number of solutions to the CSP

Definition at line 1853 of file BinaryCSP.cpp.

References is_consistent().

B.1.2.18 int BinaryCSP::solveFC_DynamicOrder (std::vector$<$ value_t $>$ & *solution*, bool *count_solutions* = `false`)

Solves the CSP using Forward Checking (FC) with dynamic variable ordering.

Returns solution through the parameter solution. If count_solutions is false, return value true: solved, false: no solution. If count_solutions is true, return value indicates number of solutions to the CSP

Definition at line 1995 of file BinaryCSP.cpp.

References DegreeGreater(), get_variables(), and is_consistent().

Referenced by testChineeseData().

B.1.2.19 int BinaryCSP::solveFC_IS (const std::vector$<$ variable_t $>$ & *order*, int *count*, std::vector$<$ value_t $>$ & *o_solution*, bool *count_solutions* = `false`)

Solves the CSP using Forward Checking and IndSet.

count is the number of variables to consider (excludes the independent set)

Definition at line 2275 of file BinaryCSP.cpp.

References is_consistent().

B.1.2.20 bool BinaryCSP::solveSLS_IS_dangles (int(BinaryCSP::∗)(const std::vector< value_t > &solution, const std::vector< variable_t > &independent_set, const std::vector< variable_t > &vertex_cover) *numBroken*, const std::vector< variable_t > & *cutset*, const std::vector< variable_t > & *independent_set*, std::vector< list< value_t > > & *solution*, double *fMaxTime*, unsigned int & *o_nFlips*, unsigned int & *o_nRestarts*)

Solve CSP using SLS/IndSet+Dangles.

numBroken is a pointer to a function that implements a heuristic, returning the number of broken constraints (e.g., SOME, ALL, NONE).

B.1.2.21 int BinaryCSP::solveWalkSatIS (const std::vector< variable_t > & *independent_set*, std::vector< list< value_t > > & *solution*) const

Solve the CSP using SLS/IndSet.

solution here is a std::vector. each element of the std::vector corresponds to a variable. The vector for that variable is a subset of the domain of that variable. The cartisian product of all these vectors forms a set of solutions to the CSP.

Definition at line 1661 of file BinaryCSP.cpp.

References expandSolution(), get_broken_constraints(), get_variables(), getSubproblem(), and vars.

B.1.2.22 int BinaryCSP::sortRecDec (std::vector< variable_t > & *result*)

Obtains RecIndSet variable ordering.

Returns size of the topmost independent set.

Definition at line 3965 of file BinaryCSP.cpp.

References recDecNode::bLeaf, recDecNode::children, recursiveDecompose(), sortDecreasing-Degree(), recDecNode::user, and recDecNode::vars.

The documentation for this class was generated from the following files:

- Desktop/BinaryCSP/**BinaryCSP.h**
- Desktop/BinaryCSP/**BinaryCSP.cpp**

B.2 BinaryCSP::Constraint Struct Reference

A binary constraint.

```
#include <BinaryCSP.h>
```

Public Member Functions

- bool **is_satisfi ed** (int val1, int val2) const

  Returns whether the constraint is specified by the given tuple.

- void **removeTuple** (**value_t** val1, **value_t** val2)

  Removes a tuple from the constraint.

Friends

- std::ostream & **operator**<< (std::ostream &, const **Constraint** &)

  Serializes the constraint to an output stream.

- std::istream & **operator**>> (std::istream &, **Constraint** &)

  Reads a constraint from an input stream.

B.2.1 Detailed Description

A binary constraint.

Lists allowed tuples.

The first variable is the one that contains this constraint. The second variable is the one referenced by this structure.

Definition at line 233 of file BinaryCSP.h.

B.2.2 Member Function Documentation

B.2.2.1 bool BinaryCSP::Constraint::is_satisfi ed (int *val1*, int *val2*) const `[inline]`

Returns whether the constraint is specified by the given tuple.

Note that the order of the tuple is important!

Definition at line 241 of file BinaryCSP.h.

B.2.2.2 void BinaryCSP::Constraint::removeTuple (value_t *val1*, value_t *val2*) `[inline]`

Removes a tuple from the constraint.

Note that the order of the tuple is important!

Definition at line 256 of file BinaryCSP.h.

Referenced by BinaryCSP::removeTuple(), and BinaryCSP::Variable::removeTuple().

The documentation for this struct was generated from the following file:

- Desktop/BinaryCSP/**BinaryCSP.h**

B.3 BinaryCSP::Variable Struct Reference

A variable in the Binary CSP.

```
#include <BinaryCSP.h>
```

Public Member Functions

- **BinaryCSP::Constraint** ∗ **hasConstraintWith** (**variable_t** var)

  Returns whether this variable contains a constraint with a given variable.

- bool **hasConstraintWith** (**variable_t** var) const

  Returns whether this variable contains a constraint with a given variable.

- void **removeTuple** (**variable_t** var2, **value_t** val1, **value_t** val2)

  Removes a tuple from a constraint (if it exists).

Friends

- std::ostream & **operator**<< (std::ostream &, const **Variable** &)

  Serializes the variable to an output stream.

- std::istream & **operator**>> (std::istream &, **Variable** &)

  Reads a variable from an input stream.

B.3.1 Detailed Description

A variable in the Binary CSP.

This variable is implicitly considered to be the first variable in each of the constraints it contains.

Definition at line 269 of file BinaryCSP.h.

B.3.2 Member Function Documentation

B.3.2.1 bool BinaryCSP::Variable::hasConstraintWith (variable_t *var*) const `[inline]`

Returns whether this variable contains a constraint with a given variable.

NOTE: Since constraints are only stored with the lower numbered variable, this function only checks variables with higher indices. Thus: var1.has_constraint_with(var2) should only be called if var1 is before var2 in the CSP list.

Definition at line 297 of file BinaryCSP.h.

B.3.2.2 BinaryCSP::Constraint$*$ BinaryCSP::Variable::hasConstraintWith (variable_t *var*) `[inline]`

Returns whether this variable contains a constraint with a given variable.

NOTE: Since constraints are only stored with the lower numbered variable, this function only checks variables with higher indices. Thus: var1.has_constraint_with(var2) should only be called if var1 is before var2 in the CSP list.

Definition at line 280 of file BinaryCSP.h.

Referenced by removeTuple().

The documentation for this struct was generated from the following file:

- Desktop/BinaryCSP/**BinaryCSP.h**

B.4 Bundle Struct Reference

A bundle, i.e. a set of equivalent variable-value pairs.

```
#include <JDT.h>
```

Public Attributes

- vector< **Var_Val** > **contents**

  The equivalent pairs.

- unsigned int **depth**

  The number of (neighbor) var-val pairs the bundle is consistent with.

B.4.1 Detailed Description

A bundle, i.e. a set of equivalent variable-value pairs.

Definition at line 20 of file JDT.h.

The documentation for this struct was generated from the following file:

- Desktop/BinaryCSP/**JDT.h**

B.5 JDT Class Reference

The Joint Discrimination Tree.

`#include <JDT.h>`

Public Member Functions

- void **build** (const vector< **variable_t** > &S)

  Builds the JDT with respect to set S.

- void **build** (const vector< **variable_t** > &S, const vector< **variable_t** > &neighbors)

  Builds the JDT for S, using a restricted neighborhood.

- int **get_num_bundles** (**variable_t** var)

  Returns number of bundles found in the domain of a given variable.

- void **print** ()

  Prints the JDT to standard out.

- void **getBundles** (vector< **Bundle** > &o_result)

  Obtains a vector of all the bundles found.

B.5.1 Detailed Description

The Joint Discrimination Tree.

Definition at line 60 of file JDT.h.

B.5.2 Member Function Documentation

B.5.2.1 void JDT::build (const vector$<$ variable_t $>$ & *S*)

Builds the JDT with respect to set S.

i.e., find bundles in the variables in S, with respect to the neighbors of S, ignoring the constraints inside S

Definition at line 18 of file JDT.cpp.

References get_domain(), get_neighbors(), is_consistent(), Var_Val::val, and Var_Val::var.

Referenced by BinaryCSP::adjustForBundling().

B.5.2.2 int JDT::get_num_bundles (variable_t *var*)

Returns number of bundles found in the domain of a given variable.

This function is defined only after the JDT has been built The return value is undefined before building a JDT. var must be a variable in the set S.

Definition at line 211 of file JDT.cpp.

The documentation for this class was generated from the following files:

- Desktop/BinaryCSP/**JDT.h**
- Desktop/BinaryCSP/**JDT.cpp**

B.6 recDecNode Struct Reference

Node for RecIndSet tree (Recursive IndSet decomposition).

B.6.1 Detailed Description

Node for RecIndSet tree (Recursive IndSet decomposition).

This struct is for internal use only

Definition at line 71 of file BinaryCSP.cpp.

The documentation for this struct was generated from the following file:

- Desktop/BinaryCSP/**BinaryCSP.cpp**

B.7 Var_Val Struct Reference

A variable-value pair.

```
#include <JDT.h>
```

B.7.1 Detailed Description

A variable-value pair.

Definition at line 13 of file JDT.h.

The documentation for this struct was generated from the following file:

- Desktop/BinaryCSP/**JDT.h**

Appendix C

BinaryCSP File Documentation

This appendix contains documentation for the C++ source files for the implementation for this thesis.

C.1 Desktop/BinaryCSP/BinaryCSP.cpp File Reference

Binary CSP implementation.

```
#include <ctime>
#include <cmath>
#include <iostream>
#include <cassert>
#include <algorithm>
#include <fstream>
#include <deque>
#include "main.h"
#include "BinaryCSP.h"
#include <windows.h>
#include "JDT.h"
```

Namespaces

- namespace **std**

Defines

- #define **IS_WINDOWS** 1

  Change this define to 0 for compiling under UNIX and 1 for Windows.

Functions

- DegreeGreater_t **DegreeGreater** (const **BinaryCSP** &csp)

  A function object for comparing variables by their degree.

- DegreeGreater_t **DegreeGreater** (const **BinaryCSP** &csp, const vector< **variable_t** > &excl)

  A function object for comparing variables by their degree.

- void **get_neighbors** (const vector< **variable_t** > &S, vector< **variable_t** > &result)

  Obtains the neighbors of a set of nodes in the graph.

- void **get_domain** (**variable_t** var, vector< **value_t** > &result)

  Obtains the domain of a variable.

- bool **is_consistent** (**variable_t** var1, **value_t** val1, **variable_t** var2, **value_t** val2)

Performs a binary constraint check.

- void **testChineeseData** (int argc, char $**$argv)

  Tests search algorithm with chineese data sets.

- void **save_set** (const char $*$filename, const vector$<$ **variable_t** $>$ &set)

  Saves a set of variables to a file.

- void **load_set** (const char $*$filename, vector$<$ **variable_t** $>$ &o_set)

  Loads a set of variables from a file.

- void **fi nd_min_max_IS** ()

  Repeatedly finds random independent sets, recording the largest and smallest sets found.

C.1.1 Detailed Description

Binary CSP implementation.

Author: Joel Gompert 2003-2005

Definition in file **BinaryCSP.cpp**.

C.1.2 Function Documentation

C.1.2.1 void get_domain (variable_t *var*, vector$<$ value_t $>$ & *result*)

Obtains the domain of a variable.

Returns the result in the argument: result Define this function for a particular CSP in order for other classes to interface with it.

Definition at line 2798 of file BinaryCSP.cpp.

References BinaryCSP::get_domain().

Referenced by JDT::build().

C.1.2.2 void get_neighbors (const vector< variable_t > & *S*, vector< variable_t > & *result*)

Obtains the neighbors of a set of nodes in the graph.

Returns the result in the argument: result Define this function for a particular CSP in order for other classes to interface with it.

Definition at line 2793 of file BinaryCSP.cpp.

References BinaryCSP::get_neighbors().

Referenced by JDT::build().

C.1.2.3 bool is_consistent (variable_t *var1*, value_t *val1*, variable_t *var2*, value_t *val2*)

Performs a binary constraint check.

Returns the result in the argument: result Define this function for a particular CSP in order for other classes to interface with it.

Definition at line 2803 of file BinaryCSP.cpp.

References BinaryCSP::is_consistent().

Referenced by JDT::build().

C.2 Desktop/BinaryCSP/BinaryCSP.h File Reference

A binary CSP.

```
#include <vector>
#include <list>
#include <string>
#include <ostream>
#include <iterator>
#include <algorithm>
```

Typedefs

- typedef std::pair< **value_t**, **value_t** > **Tuple**

  A constraint tuple.

- typedef std::pair< **variable_t**, **variable_t** > **VarPair**

  A pair of variables.

Functions

- template<typename T> void **append** (std::vector< T > &S1, const std::vector< T > &S2)

  Modifies S1, adding the contents of S2 to the end of S1.

- std::ostream & **operator**<< (std::ostream &fout, const **BinaryCSP::Constraint** &C)

  Serializes the constraint to an output stream.

- std::istream & **operator**>> (std::istream &fin, **BinaryCSP::Constraint** &C)

  Reads a constraint from an input stream.

- std::ostream & **operator**<< (std::ostream &fout, const **BinaryCSP::Variable** &V)

  Serializes the variable to an output stream.

- std::istream & **operator**>> (std::istream &fin, **BinaryCSP::Variable** &V)

  Reads a variable from an input stream.

C.2.1 Detailed Description

A binary CSP.

Author: Joel Gompert 2003-2005

Definition in file **BinaryCSP.h**.

C.3 Desktop/BinaryCSP/main.h File Reference

General definitions.

```
#include <vector>
```

Typedefs

- typedef unsigned int **variable_t**

  A handle ’pointing’ to a variable.

- typedef unsigned int **value_t**

  The value type for the variable domains.

Functions

- void **get_neighbors** (const vector< **variable_t** > &S, vector< **variable_t** > &result)

  Obtains the neighbors of a set of nodes in the graph.

- void **get_domain** (**variable_t** var, vector< **value_t** > &result)

  Obtains the domain of a variable.

- bool **is_consistent** (**variable_t** var1, **value_t** val1, **variable_t** var2, **value_t** val2)

  Performs a binary constraint check.

C.3.1 Detailed Description

General definitions.

Author: Joel Gompert 2003-2005

Definition in file **main.h**.

C.3.2 Typedef Documentation

C.3.2.1 typedef unsigned int variable_t

A handle 'pointing' to a variable.

This is to be distinguished from a **BinaryCSP::Variable**(p. 103) The implementation may treat variable_t as an index into the array of varaibles

Definition at line 16 of file main.h.

C.3.3 Function Documentation

C.3.3.1 void get_domain (variable_t *var*, vector< value_t > & *result*)

Obtains the domain of a variable.

Returns the result in the argument: result Define this function for a particular CSP in order for other classes to interface with it.

Definition at line 2798 of file BinaryCSP.cpp.

References BinaryCSP::get_domain().

Referenced by JDT::build().

C.3.3.2 void get_neighbors (const vector< variable_t > & *S*, vector< variable_t > & *result*)

Obtains the neighbors of a set of nodes in the graph.

Returns the result in the argument: result Define this function for a particular CSP in order for other classes to interface with it.

Definition at line 2793 of file BinaryCSP.cpp.

References BinaryCSP::get_neighbors().

Referenced by JDT::build().

C.3.3.3 bool is_consistent (variable_t *var1*, value_t *val1*, variable_t *var2*, value_t *val2*)

Performs a binary constraint check.

Returns the result in the argument: result Define this function for a particular CSP in order for other classes to interface with it.

Definition at line 2803 of file BinaryCSP.cpp.

References BinaryCSP::is_consistent().

Referenced by JDT::build().

Bibliography

[Achlioptas *et al.*, 1997] Dimitris Achlioptas, Lefteris M. Kirousis, Evangelos Kranakis, Danny Krizanc, Michael S.O. Molloy, and Yannis C. Stamatiou. Random Constraint Satisfaction: A More Accurate Picture. In *Proc. of CP*, volume 1330 of *LNCS*, pages 107–120. Springer, 1997.

[Barták, 1998] Roman Barták. On-Line Guide to Constraint Programming. kti.ms.mff.cuni.cz/~bartak/constraints/nosearch.html, 1998.

[Beckwith *et al.*, 2001] Amy M. Beckwith, Berthe Y. Choueiry, and Hui Zou. How the Level of Interchangeability Embedded in a Finite Constraint Satisfaction Problem Affects the Performance of Search. In *Advances in Artificial Intelligence, LNAI 2256*, pages 50–61. Springer, 2001.

[Bidyuk and Dechter, 2004] Bozhena Bidyuk and Rina Dechter. On Finding Minimal w-Cutset Problem. In *Proceedings of the Conference on Uncertainty in AI (UAI 04)*, 2004.

[Boppana and Halldórsson, 1990] Ravi Boppana and Magnús M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. In *2nd Scandinavian Workshop on Algorithm Theory*, volume 447, pages 13–25, 1990.

[Cheeseman *et al.*, 1991] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proc. of the 12 th IJCAI*, pages 331–337, Sidney, Australia, 1991.

[Choueiry and Noubir, 1998] Berthe Y. Choueiry and Guevara Noubir. On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems. In *Proceedings of AAAI*, pages 326–333, 1998.

[Choueiry *et al.*, 1995] Berthe Y. Choueiry, Boi Faltings, and Rainer Weigel. Abstraction by Interchangeability in Resource Allocation. In *Proceedings of IJCAI*, pages 1694–1701, 1995.

[Dechter and Pearl, 1987] Rina Dechter and Judea Pearl. The Cycle-Cutset Method for Improving Search Performance in AI Applications. In *Third IEEE Conference on AI Applications*, pages 224–230, 1987.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Freuder and Hubbe, 1995a] Eugene C. Freuder and Paul D. Hubbe. A Disjunctive Decomposition Control Schema for Constraint Satisfaction. In Vijay Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, pages 319–335. MIT Press, 1995.

[Freuder and Hubbe, 1995b] Eugene C. Freuder and Paul D. Hubbe. Extracting Constraint Satisfaction Subproblems. In *Proceedings of IJCAI,* pages 548–555, 1995.

[Freuder, 1982] Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1):24–32, 1982.

[Freuder, 1991] Eugene C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings of AAAI,* pages 227–233, 1991.

[Freuder, 1993] Eugene C. Freuder. Using Inferred Disjunctive Constraints to Decompose Constraint Satisfaction Problems. In *Proceedings of IJCAI*, pages 254–260, 1993.

[Galinier and Hao, 1997] Philippe Galinier and Jin-Kao Hao. Tabu search for maximal constraint satisfaction problems. In *CP 97*, pages 196–208, 1997.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W J Freeman and Co., 1979.

[Gompert and Choueiry, 2005] Joel M. Gompert and Berthe Y. Choueiry. A Decomposition Technique for CSPs Using Maximal Independent Sets and Its Integration with Local Search. In *International FLAIRS conference (FLAIRS'05)*, page 8 pages, 2005.

[Gompert, 2004] Joel M. Gompert. Local Search With Maximal Independent Sets. In *Proceedings of CP*, volume 3258 of *LNCS*, page 795. Springer, 2004.

[Haselböck, 1993] Alois Haselböck. Exploiting Interchangeabilities in Constraint Satisfaction Problems. In *Proceedings of IJCAI*, pages 282–287, 1993.

[Hogg, 1996] Todd Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, 81:127–154, 1996.

[Junker, 2004] Ulrich Junker. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. *AAAI*, pages 167–172, 2004.

[Kautz *et al.*, 1997] Henry Kautz, David McAllester, and Bart Selman. Exploiting variable dependency in local search. *Poster session abstract, IJCAI*, 1997.

[West, 2001] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, second edition, 2001.

[Williams *et al.*, 2003] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *IJCAI 03*, 2003.