Dynamically Detecting and Exploiting Symmetry in Finite Constraint Satisfaction Problems

By

Amy Davis

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, NE

April, 2002

Dynamically Detecting and Exploiting Symmetry
in Finite Constraint Satisfaction Problems

Amy Davis, M.S.

University of Nebraska, 2002

Advisor: Berthe Y. Choueiry

In this thesis we investigate the dynamic detection of symmetry relations in combinatorial problems modeled as constraint satisfaction problems (CSPs). We examine how to exploit these symmetries in order to generate a compact representation of the solution space and overcome the complexity barrier that undermines the efficient solving of these problems.

We assess the combination of these techniques with the best known strategies for improving the performance of search such as dynamic ordering heuristics and full lookahead strategies. We demonstrate that the benefits drawn from our approach are orthogonal to, and benefit from, such combinations.

We thoroughly validate these improvements through both theoretical and empirical means. Our experiments show the utility of dynamic symmetry detection on a full range of problems (i.e., from easy to difficult, for toy, real-world, and randomly generated problems). We demonstrate that these techniques are useful when finding one and all solutions, under static and dynamic ordering heuristics, and using partial and full lookahead strategies. In doing so, we dispel common notions that the dynamic computation of symmetry is too costly to be of practical utility.

We also establish that the dynamic detection and exploitation of symmetries is a powerful, cost-effective tool for dramatically reducing the peak of the phase transition, possibly the most critical phenomenon challenging the efficient processing of combinatorial problems in practice.

Although most of our work focuses on binary CSPs, we show how it can be extended to non-binary problems. We focus on the computational aspects of symmetry detection, and identify directions for future research and their impact on other disciplines, such as AI Planning, visualization, and relational databases.

Acknowledgments

Dr. Berthe Choueiry provided continual motivation and direction in this work.

Dr. Scott Henninger, Dr. Peter Revesz, Dr. Ashok Samal—committee members.

Thanks to Dr. Peter Revesz for a careful reading of the thesis and helpful feedback.

The random generator to control interchangeability that we introduce in Chapter 6 was designed in collaboration with Zou Hui and implemented by him.

The generator of non-binary random problems of Chapter 9 was designed and implemented by Zou Hui.

Xu Lin and Robert Glaubius performed nearly endless data formatting and produced the majority of charts and graphs shown in this thesis.

Special thanks to Hannah Nielsen for proof reading.

Chester Davis provided infinite patience and a large measure of sanity both in the research and the writing of this thesis.

This research was supported in part by NASA Nebraska Space Grant WBS # 26-0511-0025-001 and the Center for Communication and Information Science (CCIS).

Dedication

To my parents,

who have always encouraged me to do my best (scholastically and otherwise)

while honoring Jesus first.

Contents

List of Figures

List of Tables

Chapter 1

Overview

Constraint Satisfaction [Mackworth 1977] has emerged as a central paradigm for modeling and solving various decision problems in computer science, engineering, and management. It is most commonly used in scheduling and resource allocation applications [Fox 1987; Choueiry *et al.* 1995], and has recently been exploited in other areas of artificial intelligence (AI) such as classical AI planning [Kambhampati 1999] and collaborative problem-solving [Petrie *et al.* 1996]. Constraint Satisfaction techniques are also successfully employed in other areas in computer science such as database systems [Kanellakis *et al.* 1990] and bioinformatics [Backofen 2001]. The success of this paradigm likely derives from the flexibility and expressiveness of the modeling primitives that it provides and the effectiveness and efficiency of the techniques for solving constraints.

In its general form, a Constraint Satisfaction Problem (CSP) is **NP**-complete. Backtrack search remains the ultimate technique for solving this problem. Various strategies for improving the performance of search have been investigated in the literature [Meseguer 1989; Kumar 1992]. They are essentially based on:

1. algorithms for constraint propagation and mechanisms for pruning the search space,
2. heuristics for variable-value ordering,
3. mechanisms for intelligent backtracking, and
4. the identification of simplifying properties of a particular CSP, such as its topological structure or the semantics of its constraints.

Another opportunity to enhance the performance of backtrack search is through the identification and exploitation of structure in the problem instance. It is widely acknowledged that real-world problems exhibit an intrinsic non-random structure that makes most instances 'easy' to solve. However, such a structure is difficult to replicate 'faithfully' in modeled problem instances intended to represent real-world problems. An algorithm designed to exploit the structure of one problem will likely fail to generalize to other types of problems. It is also questionable whether a general algorithm that performs well on random problems will continue to perform well in practical settings, due to the lack of such structure in random problems.

We address this dilemma by designing a general technique that can uncover, and benefit from, the intrinsic structure in an *instance* of a problem without restricting ourselves to the declared structure of a particular class of problems. This is achieved through the computation of symmetry relations as interchangeability [Freuder 1991] among the entities of the problem. The exploitation of symmetry in general, and interchangeability in particular, can be used both to reduce the size of the search space and to represent the *solution space*, partially or entirely, in a compact manner by identifying families of qualitatively equivalent solutions, which are useful in practical applications [Choueiry *et al.* 1995] as we justify below.

1.1 Long-term motivations

Our long-term objective is to organize the solution space of a CSP in order to draw, first, a landscape of this space then, to characterize regions of this landscape (e.g., as regions where solutions are numerous or rare, stable or brittle, easy or time-consuming to modify, etc.) Such a map is useful in practical applications as it allows a human user to:

1. Rank regions of solutions with respect to some optimization function or a qualitative property;

2. Use constraints that are hard to model, such as personal preferences, to discriminate among the individual solutions in a given region; and

3. In time-critical applications, quickly retrieve an alternative to a solution that is made inconsistent by an unforeseen event.

Further, in a distributed environment where a problem is run independently through a number of specialized solvers (automated or human), global solutions can be obtained by intersecting compact solution sets of the individual solvers. The alternative strategy of having the distributed solvers collaborate on an individual solution, amending it iteratively or in parallel, is likely to cause loops and undermine the convergence of the problem-solving process.

1.2 Related work

Glaisher [1874], Brown et al. [1988], Fillmore and Williamson [1974], Puget [1993] and Ellman [1993] proposed to exploit *declared* symmetries among values in the problem to improve the performance of search. The first four papers considered *exact* symmetries only, and the latter proposed to include also necessary and sufficient *approximations* of symmetry relations. Freuder [1991] introduced a classification of various types of symmetry, which he called interchangeability. He proposed an efficient algorithm, based on building a discrimination tree, that *discovers* an exact but local form of interchangeability, *neighborhood interchangeability*. Haselböck [1993] simplified neighborhood interchangeability to a weaker form that we call *neighborhood interchangeability according to one constraint* (NI_C). He showed how to exploit NI_C advantageously in backtrack search, with and without forward checking (FC), for finding all the solutions of a CSP. Choueiry and Noubir [1998] introduced *neighborhood partial interchangeability* (NPI) that can be controlled to compute interchangeability anywhere between, and including, neighborhood interchangeability and NI_C, as shown in Figure 1.1. They generalized the discrimination tree [Freuder 1991] into the

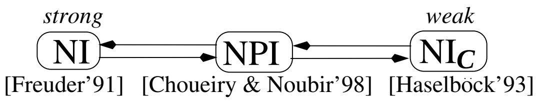


Figure 1.1: *Three types of neighborhood interchangeability.*

joint discrimination tree (JDT) to allow the computation of this new type of interchangeability.

In parallel to the work on interchangeability, Hubbe and Freuder [1989] introduced the Cross Product Representation (CPR) to represent in a compact manner the partial solutions of a CSP during search. They determined that this method significantly improves the performance of forward checking search. In Section 3.3, we show the relationship between CPR and our algorithms.

1.3 Questions addressed

In this thesis, we address the following questions:

1. How can weak interchangeability be exploited in search?

   Answer: We propose dynamic bundling, which integrates weak interchangeability with backtrack search.

2. Is it too costly to recompute interchangeability during search?

   Answer: When looking for all solutions, we establish theoretically that our technique is never more expensive than traditional forward checking algorithms. Further, we establish empirically that it provides a significant cost reduction.

3. How well does dynamic bundling fare with dynamic variable and value ordering heuristics, which are basic techniques for improving the performance of search?

   Answer: We show that dynamic bundling is an orthogonal improvement and combines well with such techniques.

4. When looking for only one solution, is it still worthwhile to bundle (either statically or dynamically)?

   Answer: Surprisingly yes, and we provide empirical evidence on a wide variety of cases.

5. Does the amount of interchangeability embedded in a given instance affect the performance of these new bundling strategies?

   Answer: A lack of interchangeability slows down search; however, we establish that bundling remains superior to non-bundling in these cases and that dynamic bundling is still beneficial.

6. How does bundling behave when combined with state-of-the-art lookahead techniques such as Maintaining Arc Consistency (MAC)?

   Answer: The combination of dynamic bundling and MAC is beneficial when used in the most rudimentary search strategies. When more advanced methods are used, such as dynamic variable ordering, we establish that MAC is not worthwhile and negatively affects performance.

7. How is the phase transition affected by bundling?

   Answer: We show that dynamic bundling significantly reduces the phase transition phenomenon.

8. Can these techniques be extended to non-binary CSPs?

   Answer: Absolutely, we show how to generalize them to non-binary CSPs in a proof of concept.

1.4 Summary of contributions

Our contributions can be organized into the following categories:

1. *Foundational issues:*

   - We introduce a new type of interchangeability, which we call Dynamic Neighborhood Partial Interchangeability (DNPI) [Beckwith and Choueiry 2001].
   - We show how to use this interchangeability to simultaneously find multiple similar solutions to a CSP. These solutions are particularly useful due to the presence of many robust [Ginsberg *et al.* 1998] alternative solutions that are compactly stored in a solution bundle.
   - We show how to obtain forward checking information from the computation of interchangeability using the joint discrimination tree, or JDT.
   - We introduce a method for exploiting DNPI during search, based on the repeated computation of the JDT during search. While NI_C or NPI can be computed in a preprocessing step prior to search, providing *static bundling*, DNPI is computed during search and provides *dynamic bundling*.
   - We extend the idea of interchangeability to non-binary CSPs, introducing a method for computing interchangeability in such situations, and showing the utility of a non-binary CSP solving algorithm that exploits DNPI.

2. *Comparing bundling strategies:*

    - We compare non-bundling search strategies to static bundling and dynamic bundling search strategies, and we demonstrate both theoretically and empirically the superiority of dynamic bundling. In particular, we give certain conditions where dynamic bundling is guaranteed to perform better bundling, visit fewer nodes, and check fewer constraints than both non-bundling or static bundling search strategies.
    - We demonstrate that dynamic bundling continues to perform better than static and non-bundling search strategies when looking for only one solution, rather than all solutions.

3. *Combining with best strategies for search:*

    - We provide an adaptation of the backtrack search procedure to allow dynamic variable-value orderings with interchangeability and show that this combination is useful in practice. We show the additional value of ordering variables dynamically (rather than statically) during search.
    - We integrate full lookahead strategies into our dynamic bundling (DNPI) search strategy, and examine under what conditions full lookahead is beneficial.

4. *Evaluation conditions:*

    - We design and implement a random generator that allows us embed a certain level of interchangeability in a CSP. We then track the behavior of various bundling and non-bundling search strategies with respect to the amount of interchangeability the problem contains.
    - We demonstrate the behavior of dynamic bundling on problems that are known to be adverse to bundling: puzzles, random CSPs, CSPs constructed to contain no opportunities for bundling, and problems residing at the phase transition [Cheeseman *et al.* 1991].

5. Finally, we identify new directions for future research.

1.5 Guide to thesis

This document is structured as follows: We first give background information on CSPs and interchangeability in Chapter 2. In Chapter 3, we show how to exploit various forms of interchangeability in the midst of search to solve a CSP and draw comparisons, both theoretical and empirical, among search without interchangeability, search with static interchangeability, and search with dynamic interchangeability. Then we extend our observations beyond what is theoretically provable to include dynamic variable-value ordering in Chapter 4 and finding the first solution in Chapter 5. We discuss the effects that interchangeability has on our algorithms and how to control and change that interchangeability in Chapter 6. We consider the effects of full lookahead techniques in Chapter 7, and the effects of all the techniques developed on the phase transition in Chapter 8. Finally, Chapter 9 extends these ideas to show that they also hold true for solving non-binary CSPs. Chapter 10 gives a conclusion reviewing our contributions and stating our future work.

Chapter 2

Background

A finite Constraint Satisfaction Problem is defined as $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, where $\mathcal{V} = \{V_1, V_2, \ldots, V_n\}$ is the set of variables, $\mathcal{D} = \{D_{V_1}, D_{V_2}, \ldots, D_{V_n}\}$ is the set of their corresponding domains (the domain of a variable is a set of values that may be assigned to the variable), and $\mathcal{C}$ is a set of constraints that specifies the acceptable combinations of values for variables. A solution to the CSP assigns to each variable a value from its domain such that all constraints are satisfied. The question is to find one or all solutions. A CSP is often represented as a graph in which the variables are represented by nodes, the domains by node labels, and the constraints between variables by edges linking the nodes in the scope of the corresponding constraint, as shown in Figure 2.1. We study CSPs with finite domains and binary constraints to develop our work. Additionally, we prove the concepts developed on non-binary CSPs.

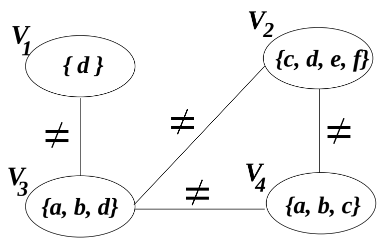


Figure 2.1: *Representation of a binary CSP.*

2.1 Characteristics of CSPs

Parameters used to describe CSPs include n, a, p (or C) , t, and $arity$. These represent:

- the number of variables in the problem (n),
- the size of the domain in each variable (a) (or the maximum domain size of all the variables),
- the constraint probability ($p = \frac{\text{number of constraints}}{\text{all possible constraints}}$) sometimes, we use the number of constraints, C,
- the constraint tightness ($t =$ the average of $\frac{\text{number of forbidden tuples}}{\text{total number of tuples possible}}$ over all constraints), and
- the number of variables to which a constraint applies ($arity$, which may also be the maximum arity of all the constraints).

Notice that low values of p may allow a CSP to be disconnected, and when $p = 1$, the CSP is a complete graph. In the example CSP shown in Figure 2.1, $n = 4$, $a = 4$, $p = \frac{4}{6} = 0.67$, and $t =$ the average of $\frac{1}{3}$ (C_{V_1,V_3}), $\frac{1}{12}$ (C_{V_2,V_3}), $\frac{1}{12}$ (C_{V_2,V_4}), and $\frac{2}{9}$ (C_{V_3,V_4}) $= 0.36$. The arity is obviously 2. Because no constraints have an arity larger than 2, this is called a *binary* CSP.

Each constraint in a CSP enumerates the combinations of variables and values that are permitted by that constraint[1]. For example, we see the constraint between V_1 and V_3 in Figure 2.1 allows V_3 to take either value a or b when V_1 is assigned d. Each constraint contains *variable-value pairs*, or vvps. Combinations of vvps are represented by *tuples*, for example ((V_1, d) (V_3, a)) and ((V_1, d) (V_3, b)). The *definition* of a constraint is a list of such tuples. Any tuple that appears in this definition is a permitted combination.

2.2 CSP benchmarks

For testing the algorithms that we develop, we use a combination of well-known CSP benchmarks and randomly generated problems. Most of our work concerns binary CSPs, so we first describe the binary CSP benchmarks. However, we also include non-binary CSPs in order to show how to extend our algorithms to the non-binary context. We include puzzles, which are known to be particularly resistant to bundling, real-life problems, and various sets of randomly generated problems.

[1] It is possible to define constraints intensionally. In our implementation, we only consider enumerated constraints—such as a table in a relational database.

2.2.1 Puzzles

Notorious puzzles include the N-Queens problem [Tsang 1993a], the Vision problem [Tsang 1993a], and the Zebra problem [Prosser 1993]. In N-Queens, one may picture a chess board, or another NxN board, on which must be placed N queens, each so that none attacks another. One solution to this problem (when N=8) is shown in Figure 2.2. While this problem has a polynomial solu-

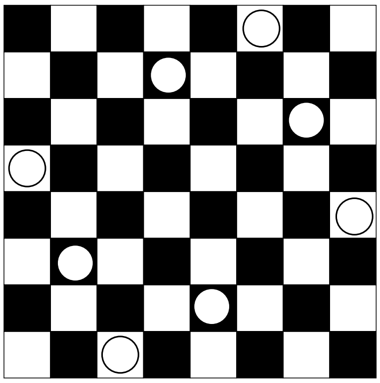


Figure 2.2: *One solution to the 8-Queens problem.*

tion [Abramson and Yung 1989; Sosic and Gu 1990], it remains a classical example for testing CSP techniques because of the large amount of interaction between the variables of the problem.

The Vision problem has the distinction of beginning the research in CSPs. It was introduced in the early 1970's and concerns the three-dimensional interpretation of two-dimensional drawings. Huffman-Clowes labeling uses "+" for a convex edge, "−" for a concave edge, and "→" for an occluding boundary. A line drawing such as the one shown in Figure 2.3 (left) is given a three-dimensional interpretation by labeling it with these Huffman-Clowes labels. The problem is to label each edge such that each corner (or junction of edges) is a possible junction, or to label each junction such that each edge common to two junctions receives the same label. For this example, the possible solutions are shown in Figure 2.3 (right).

Finally, the Zebra problem is an equally famous puzzle, and its specification is shown in Figure 2.4. Different versions of the Zebra problem have different numbers of solutions. The one shown here has one solution—The Zebra lives in the furthest right (fifth) house, and they drink water in the leftmost house (house number 1). However, by relaxing the constraint that says 'The green

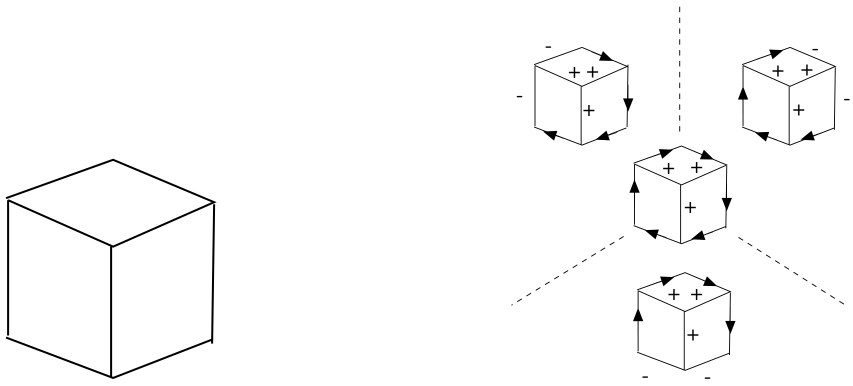


Figure 2.3: *The Vision problem (left) and its solutions (right).*

There are five houses with five different colors, in each house lives a person of different nationality having favorite drinks, cigarettes and pets, the information is:

- The Englishman lives in the red house.
- The Spaniard owns the dog.
- The Norwegian lives in the first house on the left.
- Kools are smoked in the yellow house.
- The man who smokes Chesterfields lives in the house next to the man with the fox.
- The Norwegian lives next to the blue house.
- The Winston smoker owns snails.
- The Lucky Strike smoker drinks orange juice.
- The Ukrainian drinks tea.
- The Japanese smokes Parliaments.
- Kools are smoked in the house next to the house where the horse is kept.
- Coffee is drunk in the green house.
- The green house is immediately to the right (your right) of the ivory house.
- Milk is drunk in the middle house.

The question is: Where does the zebra live, and in which house do they drink water ?

Figure 2.4: *The Zebra problem.*

house is immediately to the right of the ivory house' to 'The green house is to the right of the ivory house' (i.e., not requiring the green house to be next to the ivory house), we have a problem with eleven solutions. Finally, by removing the unary constraints, and letting milk be drunk anywhere and the Norwegian live anywhere, we have a version of the Zebra problem with 210 solutions. We call these three versions of the Zebra problem Zebra-1, Zebra-11 and Zebra-210, respectively.

These puzzles are notorious for having dense, tight constraints while continuing to have at least one solution. Further, they are known to contain no *interchangeabilities*, the kind of symmetry that we address. Therefore, they are useful to show the worst-case performance of our code.

2.2.2 Randomly generated CSPs

In addition to the utilization of benchmarks, it is customary with CSP empirical research to include a number of randomly generated problems. In our case, we use three sorts of randomly generated problems. They are generated by the random generators of Bacchus [1995] and Zou Hui [Zou *et al.* 2001; Beckwith *et al.* 2001; Zou *et al.* 2002]. Typically, a generator of random binary CSPs takes as input the following parameters $\langle n, a, p, t \rangle$. The first two parameters, n and a relate to the variables—n gives the number of variables, and a the domain size of each variable. The second two parameters, p and t control the constraints—p gives the probability that a constraint exists between any two variables (which also determines the number of constraints in the problem $C = p\frac{n(n-1)}{2}$), and t gives the constraint tightness (defined as the ratio of the number of tuples disallowed by the constraint over all possible tuples between the two variables). It is commonly assumed that in random problems, all variables have the same size domain, and all the constraints have the same tightness. In the random generator of Zou Hui [2001; 2001], we introduce an additional parameter, `IDF`. `IDF`, which we will discuss later, is a measure of the interchangeability embedded in a problem. A shortcoming of current random problems is that problems with dense, tight constraints are likely to have no solutions[2]. Thus, the puzzles described above are also particularly useful as a supplement to random problems because their constraints are dense and tight, but the problem is contrived to have at least one solution.

2.2.3 Non-binary CSPs

Finally, we demonstrate the behavior of our algorithms on a few non-binary CSPs. The non-binary CSPs used in our tests include a real-world problem of programming a Xerox PARC reprographic machine [Kapadia and Fromherz 1998], an example of database as a CSP [Gyssens *et al.* 1994], non-binary formulations of the Zebra-1 and Vision problems, and randomly generated non-binary CSPs [Zou *et al.* 2002] which allow constraints on up to 4 variables simultaneously.

[2]Note that research has just begun looking at ways to generate random problems that are guaranteed to have at least one solution [Achlioptas *et al.* 2000; Xu and Li 2000; Kautz *et al.* 2001].

- The Xerox reprographic machine programming problem has:
  - 7 variables ($n = 7$), these are: $\{A$, T, $K1$, $K2$, $K3$, L, $C\}$.
  - with respective domains: [1, 2, . . . , 8], [100, 200, . . . , 500], [1000, 2000, 3000], [1000, 2000, 3000], [3000, 4000, 5000], [1, 2, . . . , 100], [1, 2, . . . , L].
  - The constraints are given by the following functions: $L = C - 1 + \lfloor \frac{K3+K1-1}{K1} \rfloor$, $C \leq L \leq 100$.

  The problem is unique among our problem set because it has disconnected variables. It has 194 solutions.

- The database example we use is given by Gyssens [1994], and is defined as:
  - $n = 9$, $\{x_0, \ldots, x_9\}$;
  - $domains$ ={0, 1, 2} (all domains are identical);
  - $C = 8$, where the constraints are

    $c_1 = \{(0,0,0),(0,1,0),(1,0,1),(1,1,1),(0,1,2)\}$, constraining $\{x_0, x_1, x_3\}$

    $c_2 = \{(0,0,0),(0,0,1),(1,1,0),(1,0,1),(0,1,2)\}$, constraining $\{x_1, x_2, x_3\}$

    $c_3 = \{(0,0),(1,1)\}$, constraining $\{x_1, x_4\}$

    $c_4 = \{(0,0),(1,1),(1,0),(2,0)\}$, constraining $\{x_3, x_6\}$

    $c_5 = \{(0,0,0),(0,0,1),(1,1,0),(1,1,1),(1,0,2)\}$, constraining $\{x_4, x_5, x_6\}$

    $c_6 = \{(0,1),(1,0)\}$, constraining $\{x_4, x_7\}$

    $c_7 = \{(0,1),(1,0),(1,1)\}$, constraining $\{x_5, x_8\}$

    $c_8 = \{(0,0),(1,1)\}$, constraining $\{x_6, x_9\}$

  This CSP is equivalent to computing a join of eight tables (the constraints represent tables), where each table has one column for each variable incident to the constraint—making $arity$ columns. The join results with a table of nine columns (one for each of the variables), and gives their acceptable combinations. These are the solutions to the CSP. In this case, there are five possible solutions. The SQL code to find the equivalent solutions is shown in Figure 2.5

below. An introduction to other work on the relationship between constraints and databases can be found in the textbook by Revesz [2002].

select $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9$
from $c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8$
where $c_1.x_1 = c_2.x_1$ **and** $c_1.x_1 = c_3.x_1$ **and**
$c_1.x_3 = c_2.x_3$ **and** $c_1.x_3 = c_4.x_3$ **and**
$c_3.x_4 = c_5.x_4$ **and** $c_3.x_4 = c_6.x_4$ **and**
$c_5.x_5 = c_7.x_5$ **and**
$c_4.x_6 = c_5.x_6$ **and** $c_4.x_6 = c_8.x_6$

Figure 2.5: *SQL code to solve the database example.*

Unlike their binary counterparts, these non-binary CSP problems are not particularly dense (though they are tight), nor are they particularly hard to solve. They are, however resistant to bundling. We use them as a base for demonstrating dynamic bundling in non-binary CSPs. We also use non-binary randomly generated problems [Zou *et al.* 2002] to show a wider range of behavior.

2.3 Solving CSPs

Because a CSP is in general **NP**-complete[3], it is usually solved by backtrack search, an exponential procedure. Backtrack search systematically assigns a value to (instantiates) one variable at a time, checking to make sure that the value assigned does not violate any constraints. If conflict is detected, the search procedure will backtrack to find a different assignment (one that is consistent) to the variables. When all variables are instantiated and no constraints are violated, we have a solution. This procedure creates a *search space*, which is a tree of n levels with a branching factor a, as partially shown in Figure 2.6.

At each point in the process of search, we have *past*, *current* and *future* variables. The current variable, V_c, is a variable for which a fitting value is being sought by the search procedure. Relative to the current variable, variables already assigned a value are past variables, V_p, and variables not yet assigned a value are future variables, V_f. In Figure 2.6 above, if a search procedure had assigned

[3]By reduction from 3SAT.

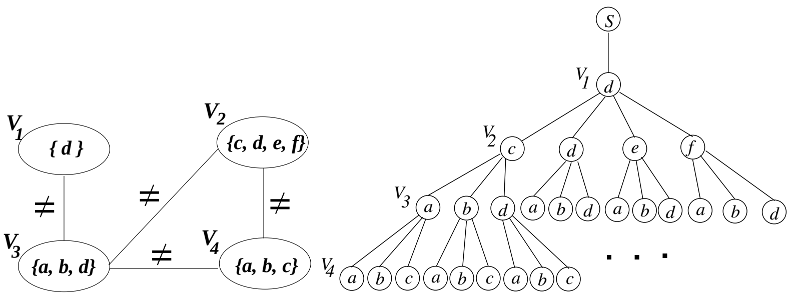


Figure 2.6: *The search space for the example CSP.*

d to V_1, and e to V_2, and was in the process of choosing a value for V_3, we would say that the current variable is V_3, past variables are V_1 and V_2, and the only future variable is V_4.

Forward checking (FC) [Haralick and Elliott 1980] is a common improvement to backtrack search. FC ensures that each time a current variable is assigned a value, the domain of each future variable connected to the current variable via a constraint is revised to exclude values inconsistent with the assignment of the current variable. This process is called pruning, and is shown in Figure 2.7. Because of this pruning, FC assigns only values that are consistent. Further, the domains of all future variables are always consistent with that of every past variable, given the binary constraints, thus eliminating the need for back-checking (i.e., consistency checking against past variables). FC is a *parital* lookahead technique. It looks ahead from the current variable, to part of the future (those variables that are connected to the current variable with a constraint), and removes inconsistent values.

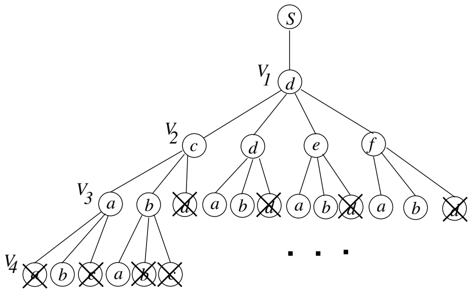


Figure 2.7: *Forward checking during search.*

2.4 Interchangeability

The idea behind interchangeability is to find and eliminate redundant values in a CSP. Interchangeable values behave similarily in either local or global environments and are thus redundant. Replacing interchangeable values with a set and treating them as one value reduces the search space of a problem while retaining all solutions, as shown in Figure 2.8. Below we explain the basic kinds of

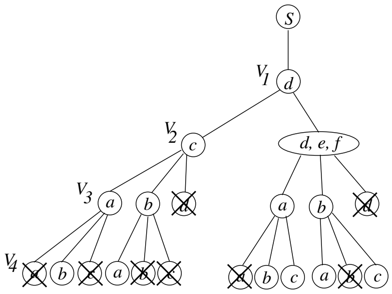
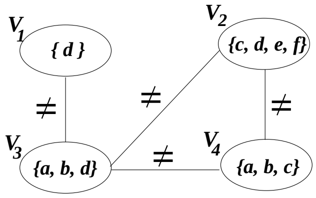


Figure 2.8: *Search space with interchangeability.*

interchangeability used here and describe efficient algorithms to find them.

2.4.1 Interchangeability definitions

A solution to a CSP finds values for the variables in a CSP that are consistent with respect to the constraints on those variables. Many CSPs have more than one solution. In such a situation, there exists a mapping between the solutions, such that, if the mapping is known, one solution can be obtained from another without performing search, (in the broadest sense, this is functional interchangeability [Freuder 1991]). The interchangeabilities we use are a simple form of such a mapping. In the following definitions, we will use the CSP from Figure 2.1 (recalled here) as a running example.



The solutions to this CSP are shown in Table 2.1. Notice that the solutions are grouped into four sets of solutions, each containing three or four solutions. In each of these sets, the values for V_1, V_3,

Solution #	V_1	V_2	V_3	V_4
1	d	c	a	b
2	d	d	a	b
3	d	e	a	b
4	d	f	a	b
5	d	d	a	c
6	d	e	a	c
7	d	f	a	c
8	d	c	b	a
9	d	d	b	a
10	d	e	b	a
11	d	f	b	a
12	d	d	b	c
13	d	e	b	c
14	d	f	b	c

Table 2.1: *All solutions to CSP example in Figure 2.1.*

and V_4 are the same, with only V_2 changing. This illustrates *full interchangeability* among some of the values of V_2.

Definition 2.4.1. *Full interchangeability*: A value a in the domain of variable V is interchangeable with a value b in the same domain iff every solution to the CSP that involves a remains a solution when b is substituted for a, and vice versa.

In other words, two values of the variable V are fully interchangeable when the only difference between two solutions of a CSP is the value of V itself. This is true of values d, e, and f for variable V_2 in Figure 2.1. For each solution that contains the variable-value pair (V_2, d) (solutions numbered 2, 5, 9, and 12 in Table 2.1), there is a solution that contains (V_2, e) with no other values changing (solutions 3, 6, 10, and 13 in Table 2.1). The same holds for (V_2, f), so that all three are interchangeable. One can readily see that this set, (V_2, (d, e, f)), provides groups of very similar solutions (differing on the value of only one variable), and that time and space would be saved by treating this set as one value rather than three. Note that (V_2, c) is not in this interchangeability set—it only participates in half of the solutions that (V_2, (d, e, f)) do.

Definition 2.4.2. *Equivalence classes and domain partitions*: A set of values that are interchangeable with each other is an *equivalence class*. The domain of each variable is seperated into *domain* *partitions* by interchangeability, such that each value in the domain resides in exactly one equivalence class.

With full interchangeability, the domain of V_2 above is partitioned into two equivalence classes, one containing (c), and the other (d, e, f).

The computation of full interchangeability requires, in general, finding all solutions. Therefore, it is likely to be intractable and impractical in use. However, Freuder [1991] also identifies a form of local interchangeability, called *neighborhood interchangeability*, which is a sufficient approximation of full interchangeability.

Definition 2.4.3. *Neighborhood interchangeability*: A value a in the domain of variable V is neighborhood interchangeable with a value b in the same domain iff for every constraint C incident to V, a and b are consistent with exactly the same values: $\{x \mid$ *(a,x) satisfies* $C\} = \{x \mid$ *(b,x) satisfies* $C\}$. *Neighborhood interchangeability is a sufficient, but not a necessary condition for full interchangeability.*

In neighborhood interchangeability, rather than consider what values are interchangeable with respect to the entire CSP, requiring that we find all solutions, we look only at one variable, and the variables connected to it via constraints—that is, its *neighborhood*. Figure 2.9 shows the same CSP example, with neighborhood of V_2 emphasized. Specifically, we notice that the values of V_2 are consistent with the neighboring values shown in Table 2.2 Notice that (V_2, e) and (V_2, f) are

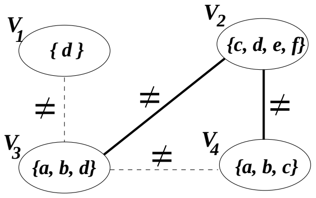


Figure 2.9: *Neighborhood interchangeability in the CSP.*

interchangeable in this context. (V_2, c) cannot join the group, because it is not consistent with (V_4, c), as the others are, and (V_2, d) because it is not consistent with (V_3, d), as the others are. Importantly, two values that are not fully interchangeable are guaranteed to not be neighborhood interchangeable either, and any two values found to be neighborhood interchangeable are guaranteed

V_2 value	consistent with V_3 values	consistent with V_4 values
c	a, b, d	a, b
d	a, b	a, b, c
e	a, b, d	a, b, c
f	a, b, d	a, b, c

Table 2.2: *Neighborhood interchangeable sets for* V_2.

to be fully interchangeable also. Thus, the computation of neighborhood interchangeability, which is $\mathcal{O}(n^2a^2)$ (we describe the algorithm in Section 2.4.3), finds *some* of the available fully interchangeable values. We can view the conceptual difference between the domain partitions produced by full interchangeability and neighborhood interchangeability as shown in Figure 2.10.

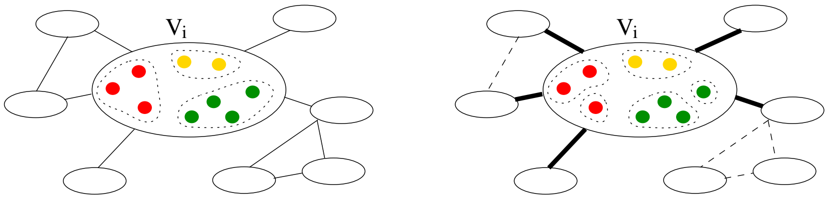


Figure 2.10: *Full interchangeability (left) and neighborhood interchangeability (right).*

Both full interchangeability and neighborhood interchangeability do not permit variables other than in the selected variable V in the CSP to change. Partial interchangeability is a *weaker* kind of interchangeability, based on the idea that when a value for V changes, values for other variables may also differ among themselves, but be fully interchangeable with respect to the rest of the CSP. We introduce a boundary of change, $\mathcal{S}$, within which we permit change. In Figure 2.11, the boudary

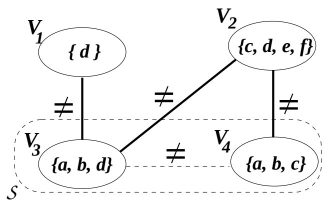


Figure 2.11: *Partial interchangeability in the CSP.*

of change is denoted by the dashed line around V_3 and V_4. It effectively says “For all values in V_3, I will ignore their effect on the value of V_4, but consider it with respect to the remainder of the CSP.”

Revisiting Table 2.1 allows us to note that values a and b in variable V_3 are partially interchangeable when $\mathcal{S}$ includes V_3 and V_4, and that values a and b in variable V_4 are partially interchangeable with respect to the same $\mathcal{S}$.

Partial interchangeability, like full interchangeability, is a global form of interchangeability. We can localize partial interchangeability in much the same way that we localized full interchangeability, by only considering those variables whose constraints cross the boundary of $\mathcal{S}$. (These are the variables in the neighborhood of $\mathcal{S}$.) This is called neighborhood partial interchangeability (NPI) [Choueiry and Noubir 1998]. In the example CSP of Figure 2.9, these are equivalent. However, we show in Figure 2.12 the conceptual difference. Dashed lines represent constraints ignored when finding interchangeability.

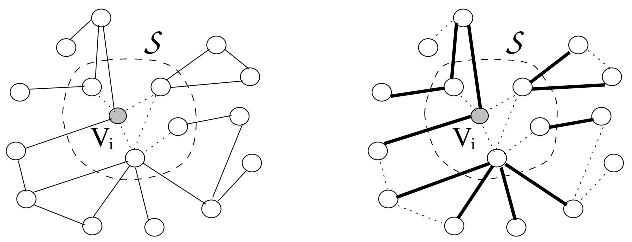


Figure 2.12: *Partial interchangeability (left) and neighborhood partial interchangeability, NPI (right).*

Before continuing, let us review the conceptual differences and characteristics of these four basic types of interchangeability. Figure 2.13 demonstrates these four types of interchangeability and their relationships to each other. Full and partial interchangeabilities are global and likely to be intractable. Neighborhood and NPI interchangeabilities are local, and have polynomial time algorithms to find them (described in Section 2.4.3). Notice that full interchangeability is equivalent to partial interchangeability with $\mathcal{S}$ surrounding only one variable. The same is true of neighborhood interchangeability and NPI.

2.4.2 Modifying NPI to obtain NI$_C$

A further weakening of localized interchangeability is to take the locality of neighborhood interchangeability and NPI to an extreme. Haselböck [1993] suggested that we find interchangeabilities across a *single* constraint. We call this interchangeability NI$_C$.

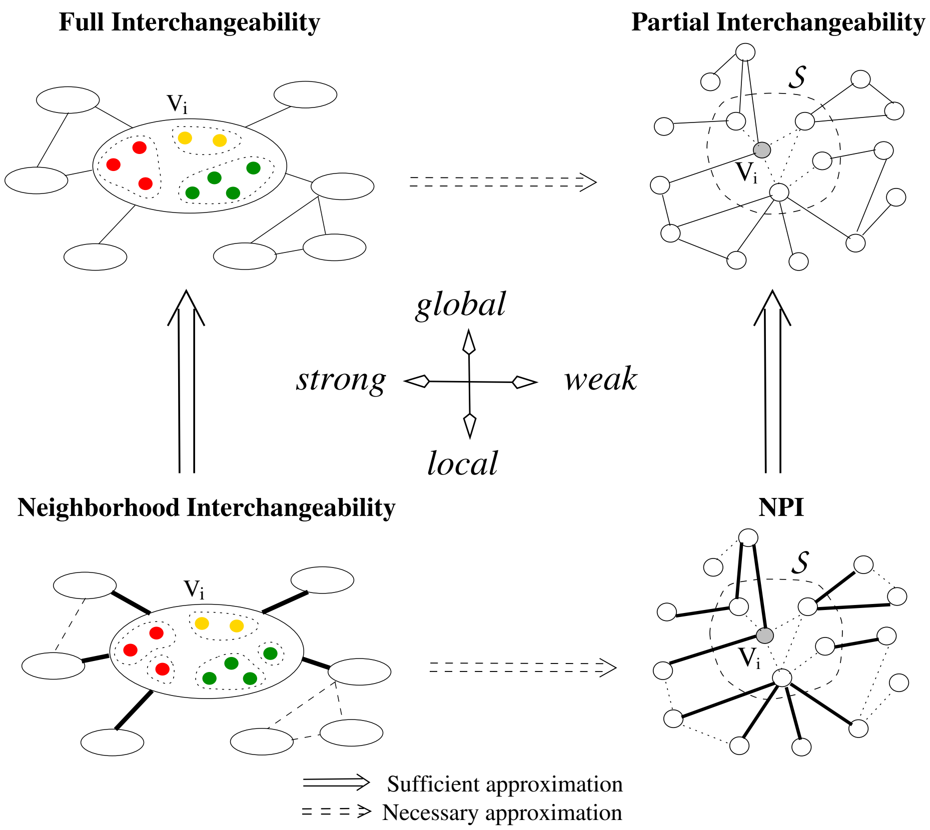


Figure 2.13: *Conceptual comparison between Full, Partial, Neighborhood and Neighborhood Partial interchangeabilities.*

Definition 2.4.4. *Neighborhood interchangeability according a constraint (NI$_C$)*: A value a in the domain of variable V_i is neighborhood interchangeable across a constraint (NI$_C$) with a value b in the same domain iff a and b are consistent with the same values in another variable V_j according to *one* constraint, C. NI$_C$ is a sufficient condition of NPI.

In order to visualize the relationship between NPI and NI$_C$, consider a variable V_i, and its neighborhood, as shown in Figure 2.14. The upper left-hand corner shows one of the many possible boundaries of change that could be used to compute NPI sets of V_i. Here, the constraints represented by dotted lines are ignored because they are inside the boundary of change, $\mathcal{S}$. The upper right-hand corner shows a potential NI$_C$ set that could be applied to V_i. NI$_C$, because it is across only one constraint, effectively ignores all other constraints. This is equivalent to defining the boundary of change in NPI to exclude effects of V on all the neighborhood of V *except* the one constraint the

NI_C considers, as shown in the lower left-hand corner of Figure 2.14. Thus, we see that NI_C is a special case of NPI. Finally, in the lower right-hand corner, we see that NPI can also be obtained from NI_C by calculating NI_C over several constraints and taking their intersection.

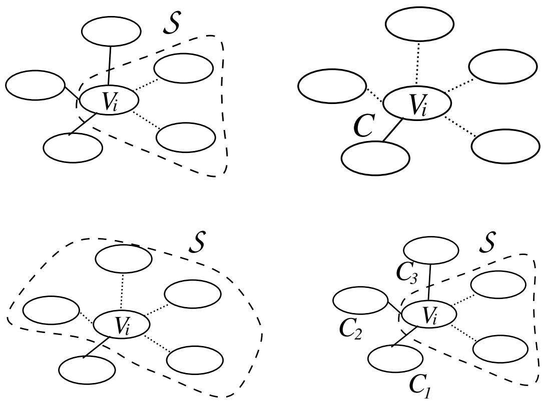


Figure 2.14: *The relationship between NPI and* NI_C.

Proposition 2.4.5. The NPI partition of the domain of a variable V for $\mathcal{S} = \{V, V_1, V_2, \ldots, V_k\}$ is equal to the intersection of the NI_C partitions of the domain of V according to V_1, $V_2, \ldots, V_k$.

The intersection of k partitions can be performed in $\mathcal{O}(ka^2)$.

2.4.3 Finding interchangeable sets

Freuder [1991] provided an algorithm, the discrimination tree, for computing the partitions of a domain into equivalence classes based on neighborhood interchangeability. This is performed one variable at a time. In Figure 2.15, we show this algorithm in action for the variable V_2, from the CSP example in Figure 2.1. The idea is based on a simple consistency-checking mechanism between a variable (here V_2) and its neighbors (here V_3 and V_4). For each value of V_2, it iterates through each variable-value pair in the neighborhood in a pre-defined order (a lexicographical order, for example). As it goes, it builds a tree, where the nodes of the tree are variable-value pairs that are consistent. To some of the nodes are attached *annotations*. In each annotation is one or more values of V_2; these annotations define sets of values that are neighborhood interchangeable. In Figure 2.15, the rectangles denote the annotations, and the partition of the domain of V_2. As we saw before, we see that values e and f in V_2 are neighborhood interchangeable. This calculation is polynomial in the size of the domain and the number of constraints incident to the variable; it is $\mathcal{O}(n^2a^2)$.

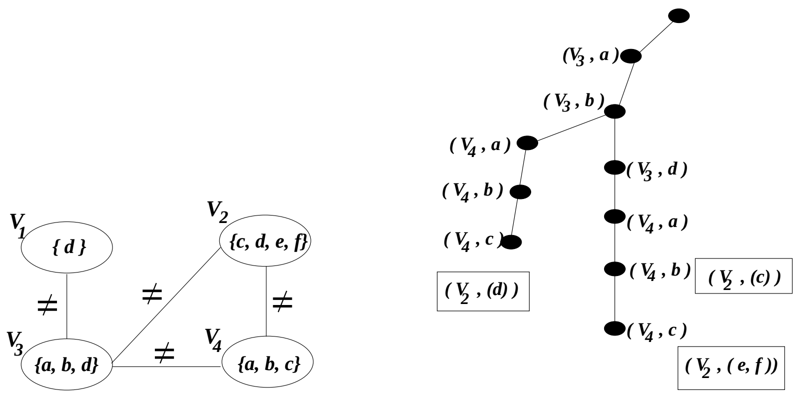


Figure 2.15: *The discrimination tree of* V_2.

Choueiry and Noubir [1998] showed how to extend the discrimination tree into the joint discrimination tree (JDT) to partition the domain of a variable V into sets of values that are NPI. It operates exactly like the discrimination tree of Freuder, but rather than considering all variables in the neighborhood of V, it iterates over variables in the neighborhood of $\mathcal{S}$, the boundary of change. In both the discrimination tree and the JDT, the path between a given annotation and the root of the tree gives the values for the variables adjacent to V that are consistent with the values of V in that particular annotation.

Figure 2.16 shows the JDT for the CSP example and boundary of change $\mathcal{S} = \{V_3, V_4\}$ in Figure 2.11.

When the JDT is restricted to computing the NPI sets of only one variable in $\mathcal{S}$[4], it has a time complexity of $O((n-s)a^2)$ and a space complexity of $O((n-s)a)$, where s is the size of $\mathcal{S}$. Thus, it is cheaper than neighborhood interchangeability (using the discrimination tree) although more expensive than NI_C, as we summarize in Table 2.3.

[4]The same JDT can be used to compute the NPI sets of all the variables in $\mathcal{S}$.

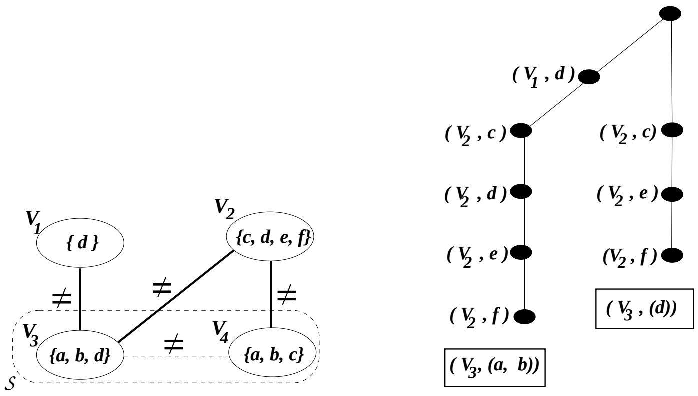


Figure 2.16: *Joint Discrimination Tree (JDT) for V_3 with $\mathcal{S} = \{V_3, V_4\}$.*

2.5 A new kind of interchangeability

Freuder [1991] noted that interchangeability sets can be recomputed after instantiations are made during backtrack search. This sort of interchangeability, called *dynamic* interchangeability can take advantage of the filtering of inconsistent values during the search process by interleaving backtracking with the interchangeability detection process. We propose here a new type of dynamic interchangeability, based on the dynamic computation of NPI.

Definition 2.5.1. *Dynamic NPI (DNPI)*: Given a variable ordering in a backtrack search integrating any kind of lookahead scheme, the partition of the domain of the current variable, V_c, obtained by the JDT of V_c with $\mathcal{S}$ $=\{V_p, V_c\}$, where V_p are the past variables in the search tree, defines a new type of NPI, which we call DNPI.

All strategies discussed here add the cost of computing the bundles to the cost of search. These costs are summarized in Table 2.3, and provide a worst-case bound on the overhead to search added by bundling. Because search is exponential, one may argue that they can be neglected. However, they still fail to show the *positive* effect of bundling on the cost of search. For this reason, we theoretically and empirically evaluate both search strategies based on (1) no bundling, (2) static bundling, and (3) dynamic bundling.

	Bundle generation			Overhead to search	
	Task	Time	Space	Time	Space
neighbor. int.	One domain partition	$O(na^2)$	$O(a)$	$O(n^2a^2)$	$O(na)$
NI_C	One domain partition	$O(a^2)$	$O(a)$	$O(n^2a^2)$	$O(n^2a)$
DNPI	One domain partition	$O((n-s)a^2)$	$O((n-s)a)$	$O(n^2a^2)$	$O(n(n-s)a)$

Table 2.3: *Cost of finding interchangeability.*

Summary

A Constraint Satisfaction Problem is composed of variables with domains and constraints. They are generally solved using backtrack search. We propose improvements to this backtrack search procedure. To test our algorithms thoroughly, we test them on puzzles and on randomly generated CSPs. Puzzles allow us to test our algorithms under adverse conditions and round out areas where random problems are likely to be unsolvable. Randomly generated problems give us a wide variety of problems, allowing us to easily see the overall behavior of the algorithms tested.

In an instance of a CSP may reside a kind of symmetry called interchangeability. We review the definitions of and illustrate full interchangeability, neighborhood interchangeability, partial interchangeability, and neighborhood partial interchangeability, and their relationships to each other. Then, we investigate two modifications of NPI. First, when taking NPI to the extreme, we can calculate neighborhood interchangeability across one constraint, NI_C. Second, we can calculate NPI interleaved with search to yield dynamic NPI, or DNPI. We show how to find neighborhood interchangeability and NPI using the discrimination tree [Freuder 1991] and the JDT [Choueiry and Noubir 1998]. In Table 2.3, we compare the cost of computing NI_C and DNPI with respect to the cost of computing neighborhood interchangeability. In the following chapter, we will see how to use these calculated interchangeabilities in the process of searching for solutions to a CSP.

Chapter 3

Search with interchangeability

Once a set of interchangeable values in a variable are discovered, they can be replaced by one representative of the set. This is useful both to find families of similar solutions and to reduce the size of the search space in backtrack search. We consider the effects of interchangeability when finding all solutions to a Constraint Satisfaction Problem using the three strategies in Table 3.1. In order to draw theoretical comparisons between the bundling strategies, we assume the same ordering for variables and values across strategies. We demonstrate theoretically and empirically that dynamic bundling is always worthwhile in this context.

Search		**Reference**
Non Bundling	FC	[Haralick and Elliott 1980], and Section 2.3
Static bundling	NIC-FC	[Haselböck 1993], and Section 3.1.2
Dynamic bundling	DNPI-FC	Section 3.1.3

Table 3.1: *Search and bundling strategies.*

3.1 Bundling search strategies

Recall that forward checking search (FC) systematically assigns a value to (instantiates) one variable at a time and for each assignment prunes the domains of variables connected to the current variable with a constraint. In these pruned variables, no values inconsistent with the current assignment remain. Below we describe a method of performing forward checking *while* finding interchangeability and two improvements to FC, both of which exploit interchangeability.

3.1.1 Using the JDT for forward checking

Recall that in the JDT (introduced in Section 2.4.3), a path in the tree from the root to any particular node with an annotation give the variable-value pairs that are consistent with the values in that annotation. These are *exactly* the new domains of the variables adjacent to V should forward checking revise their domains after assigning the values in A_i to V. Thus, these variable-value pairs along each path can be used to update the domains of the future variables and eliminate the need for an explicit forward checking procedure. *As a consequence, a (joint) discrimination tree provides not only the equivalence sets of values in the domain of a variable, but also the new domains of the neighboring variables for each assignment of the variable to one of its domain partitions—at no extra cost.* Our novel exploitation of this information guarantees that that our dynamic bundling strategy never requires more constraint checks than FC[1].

3.1.2 Search with static interchangeability (NIC-FC)

Haselböck [1993] proposes to compute all NI_C sets (for all variables according to every constraint) as a preprocessing step prior to search and then uses these interchangeabilities during search. Because the interchangeability is computed only once, it is *static*. The resulting search strategy, which we call NIC-FC, operates as follows. Before search begins, the domain of each variable V is partitioned according to NI_C, resulting in a domain partition for each constraint incident to V. Thus, if V has e constraints, then V has e domain partitions. Since each variable has at most $(n-1)$ incident constraints (thus $(n-1)$ NI_C partitions), this pre-processing requires $O(n^2a^2)$ time and $O(n^2a)$ space, reserved *throughout* the search process. During search, these partitions conceptually separate into two sets:

1. `NIC-with-past`, computed using constraints between V_c and a past variable V_p; and

2. `NIC-with-future`, computed using constraints between V_c and a future variable V_f.

Partitions in `NIC-with-past` are used when the domain of V_c is *revised* by FC (at the instantiation of a previous variable), and is shown in Figure 3.1.

[1]To this end, the implementation of the JDT has to stop expanding a path once it is clear that the domain of a neighboring variable is annihilated.

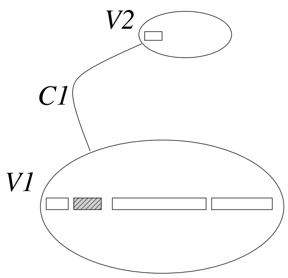


Figure 3.1: *Revise the domain of V_1 according to the assignment of V_2.*

The set of domain partitions in `NIC-with-future` are intersected to find the finest partition when V_c is instantiated. After the intersection, values that no longer are in the domain of V_c are removed. This is shown in Figure 3.2.

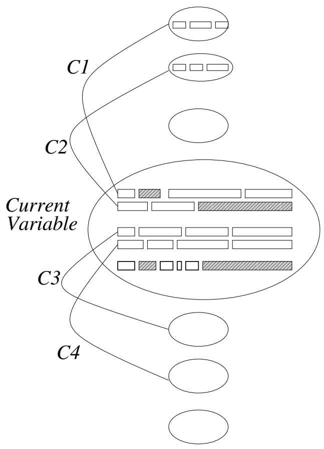
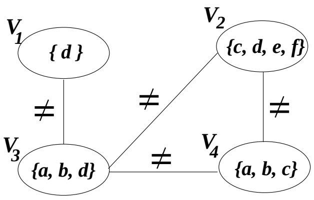


Figure 3.2: *Search with static interchangeability.*

Using the example CSP from Figure 2.1 (recalled in Figure 3.3), we demonstrate the performance of NIC-FC. NIC-FC first computes the NI_C sets for each of the constraints on all of the variables. These sets are shown in Figure 3.3 and can easily be verified by hand. Given these sets, let us suppose that NIC-FC uses the static variable ordering: V_1, V_2, V_3, V_4. In this situation, NIC-FC operates as follows:

1. Instantiate V_1. It only has one value in its domain and only one constraint. The bundle (d) is assigned to variable V_1. Forward checking removes the value d from the domain of V_3.

2. Instantiate V_2. There are two constraints on V_2, both of them with future variables. The



Variable	Constraint	Partition
V_1	C_{V_1,V_3}	(d)
V_2	C_{V_2,V_3}	$(c,e,f)\ (d)$
	C_{V_2,V_4}	$(c)\ (d,e,f)$
V_3	C_{V_3,V_1}	$(a,b)\ (d)$
	C_{V_3,V_2}	$(a,b)\ (d)$
	C_{V_3,V_4}	$(a)\ (b)\ (d)$
V_4	C_{V_4,V_2}	$(a,b)\ (c)$
	C_{V_4,V_3}	$(a)\ (b)\ (c)$

Figure 3.3: *NI_C sets for the example CSP*

constraint C_{V_2,V_3} yields the partition (c, e, f) (d), and the constraint C_{V_2,V_4} yields the partition (c) (d, e, f). Intersecting these two partitions gives the sets (c) (d) (e, f) from the domain of V_2. NIC-FC chooses one of these sets, and assigns them. In this exercise, we will assign (e, f) to variable V_2. Forward checking removes no values.

3. Instantiate V_3. Notice that the domain of V_3 has been modified by the constraint with V_1, and now is $\{a, b\}$. Only one of the constraints incident to V_3 concerns a future variable—V_4. The partitions of V_3 according to this constraint are (a) (b) (d) ((d) will be removed because it is not in the domain). We will assign (a) to V_3 and remove a from the domain of V_4 with forward checking.

4. Instantiate V_4. The domain, after modification because of the constraint to V_3, is $\{b, c\}$. Since there are no constraints with the future, the domain is not partitioned, and the bundle (b, c) is assigned to V_4.

5. At this point, we have found a solution. Though these algorithms perform best when finding all solutions, we leave the exercise with just one solution. The size of this solution bundle is four, meaning that is contains four solutions.

We note here two possible improvements to NIC-FC. When V_c is instantiated, its partitions in `NIC-with-future` are used. V_c is assigned the sets of values (i.e., bundles) obtained by intersecting *all* its NIC partitions in the set `NIC-with-future`. According to Proposition 2.4.5, these bundles are exactly the sets of the (static) NPI partition of V_c with $\mathcal{S} = \{V_p, V_c\}$. Thus, if k is the number of future variables V_f, the computation of the partitions in `NIC-with-future` ($O(ka^2)$) and their intersection ($O(ka^2)$) can be replaced with the computation of the NPI partition of the domain of V_c with $\mathcal{S} = \{V_p\}$ which is ($O(ka^2)$), saving the effort for computing the intersection. This offers a first opportunity to improve NIC-FC.

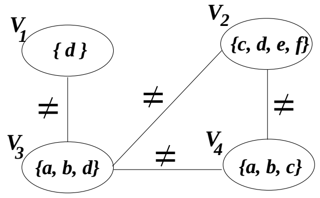
As noted above (in Section 3.1.1), the revised domains of the future variables V_f when V_c is assigned a set of values can be directly obtained from the JDT. Consequently, using the JDT of V_c with $\mathcal{S}$ would also save all the constraint checks otherwise spent by the revise step in NIC-FC. This constitutes a second opportunity to improve NIC-FC. We do not implement either improvement. Rather, we demonstrate the benefits of dynamic bundling.

3.1.3 Search with dynamic interchangeability (DNPI-FC)

As noted by [Freuder 1991], interchangeability sets can be re-computed after some assignments are made in the course of search. Because domains are pruned during forward checking, interchangeabilities that did not exist before search began may present themselves during search. This dynamic interchangeability must obviously be computed in steps interleaved with search.

We introduce here DNPI-FC, a search procedure with forward checking using the dynamic computation of NPI sets. It operates as follows. For a current variable V_c, the JDT of V_c with $\mathcal{S}$ $= \{V_p, V_c\}$ is computed. This yields a partition of the domain of V_c and, for each equivalence set, the new domains for all future variables, V_f. The domains of the future variables are updated according to the corresponding path in the JDT.

In order to demonstrate the performance of DNPI-FC, we use the same example CSP from Figure 2.1, and the ordering from the NIC-FC demonstration: V_1, V_2, V_3, V_4. DNPI-FC operates as follows.



1. Build the JDT for V_1. This rather small and simple JDT is shown in Figure 3.4. This gives

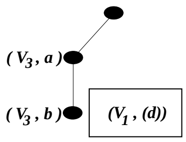


Figure 3.4: *The JDT for variable V_1.*

us (d) as the partition of the domain, and show us that the resulting domain for V_3 (the only connected variable), is $\{a, b\}$. We then assign (d) to V_1.

2. Build the JDT for V_2 with $\mathcal{S}$ including V_1 and V_2, as shown in Figure 3.5. The neighborhood

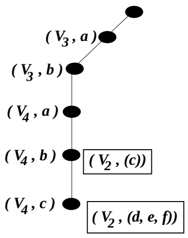


Figure 3.5: *The JDT for variable V_2.*

of $\mathcal{S}$ is V_3 and V_4. Recall that the domain of V_3 has been pruned to $\{a, b\}$ and that the domain of V_4 remains $\{a, b, c\}$. The JDT shows that the partition of the domain of V_2 is $(c), (d, e, f)$. Also from the JDT, we gather that if we assign (c) to V_2 the domain of V_3 remains the same, and the domain of V_4 becomes $\{a, b\}$, but if we assign (d, e, f) to V_2, the domains of both V_3 and V_4 remain the same. We will assign (d, e, f) to V_2.

3. Build the JDT for V_3. Here, $\mathcal{S}$ includes V_1, V_2 and V_3, leaving only V_4 in the neighborhood of $\mathcal{S}$, as shown in Figure 3.6. The JDT tells us that the partition of the domain is $(a)(b)$, and

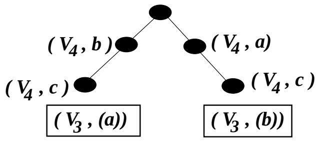


Figure 3.6: *The JDT for variable V_3.*

that in either case, the value assigned to V_3 should be removed from the domain of V_4. We instantiate V_3 with (a).

4. Finally, we come to V_4. We see that $\mathcal{S}$ includes V_1, V_2, V_3 and V_4, leaving no variables in the neighborhood of $\mathcal{S}$. The JDT is not employed here. We merely assign the set (b, c) to the V_4.

5. We have found a solution. Notice that the size of the solution bundle is six—slightly larger than the solution bundle found by NIC-FC. This is due to the re-calculation of interchangeability at each step, which found a larger equivalence class in variable V_2.

It has been argued [Meseguer 1989] that such dynamic computation of interchangeability would be too costly to be practical during search. We counter this assumption in this thesis—first by showing the scenarios where it is guaranteed to not cost more (in terms of constraint checks or nodes visited) than FC, and second by establishing empirically a variety of other situations in which dynamic bundling proves useful.

3.1.4 CPR-FC

In parallel to the work on interchangeability, Hubbe and Freuder [1989] introduced the Cross Product Representation (CPR) to represent in a compact manner the partial solutions of a CSP during search. They proposed two search procedures based on CPR, with and without forward checking, that find all solutions to a CSP and reduce significantly the number of constraint checks. We show in Section 3.3 that their algorithm, though not introduced as interchangeability, is somewhat equivalent to DNPI-FC.

During search, the Cross Product Representation (CPR) considers all possible values for V_c, revising the domains for future variables, V_f, by forward checking over each of these values. The filtered domains are then compared for equality. When equality holds for all future variables, the corresponding values for V_c are merged into a set, which constitutes the bundled assignment of V_c.

3.2 Search evaluation criteria

Above, we define three search strategies that will be compared rigorously in the remainder of this thesis. The first is forward checking search (FC) [Haralick and Elliott 1980], which does no bundling and serves as a baseline algorithm. The second is search with static bundling [Haselböck 1993], which we call NIC-FC, and the third is search with dynamic bundling, introduced here and called

DNPI-FC. The effectiveness of each search strategy will be assessed by the following measurements: Constraint Checks (`CC`), Nodes Visited (`NV`), number of Solution Bundles (`SB`), and CPU time. Each of these measurements is straight-forward and will be briefly discussed before continuing.

3.2.1 Constraint Checks (`CC`)

A constraint linking two variables is *checked* each time that a tuple of two variable-value pairs is tested to see if it is consistent with the constraint. For example, during forward checking, the chosen value for the current variable will be compared against values in the domains of future variables. Each comparison involves one value from the current variable and one value from the future variable. This is counted as one constraint check.

3.2.2 Nodes Visited (`NV`)

When, during search, a variable is instantiated, we say that a node in the search space is visited. In the search space (which can be viewed as a tree with n levels and a branching factor of a), each node is a variable-value pair. Instantiating a variable to a value (or set of values) visits that node in the search space.

3.2.3 Solution Bundles (`SB`)

A solution bundle is set of solutions found by instantiating every variable to one or more values such that no constraints are violated by the assignments. We count the size of the solution bundle by taking the product of the number of values in each assignment. If each variable is assigned exactly one value (as is the case with non-bundling FC), then the solution is size one. Notice that this value represents the total number of solutions that may be found by enumerating the possibilities stored in this bundle. The solution bundle is merely a compact way of representing multiple solutions.

3.2.4 CPU time

Finally, each set of binary CSP experiments reported below were conducted on `tonfano.unl.edu` (unless otherwise noted) under normal load. The clock resolution of LISP on `tonfano` is 10ms. Units of time are reported in ms, but often the measurements are hindered by the clock resolution.

The first two criteria, CC and NV, are orthogonal standard measures [Kondrak and van Beek 1995] for assessing the performance of search independent of the implementation details. We wish to minimize each of these four: CC, NV, SB, and CPU time. Note that minimizing SB maximizes the size of the bundles. In future chapters, when solving for one, rather than all solutions, we will use the size of the first bundle found as our measurement (First Bundle Size = FBS). In this case, we want to maximize FBS.

3.3 Theoretical comparisons of search strategies

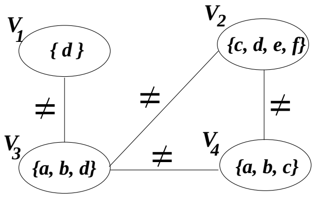
In Section 3.1, we discussed two bundling search strategies: NIC-FC and DNPI-FC. Recall that NIC-FC performs static bundling by computing all interchangeable sets before search. DNPI-FC performs dynamic bundling by repeatedly computing interchangeability sets during search. Common opinion holds that such repeated computations are far too expensive to be useful in practice. However, we provide theoretical guarantees that DNPI-FC costs less than FC in terms of the standard search criteria recalled in Section 3.2. In order to compare NIC-FC and DNPI-FC, we must first understand the difference in their bundling capabilities.

Theorem 3.3.1. Each value in the domain of a variable is a subset of at most one equivalence class in NI_C and DNPI.

This follows directly from the definition of interchangeability which creates a partition of the domain of a variable.

Theorem 3.3.2. Each equivalence class in NI_C is a subset of an equivalence class in DNPI but not vice versa.

NIC-FC may find more equivalence classes than DNPI. Recall from the search procedures described in Section 3.1 that NIC-FC partitions the domains of all variables before beginning search but that DNPI instead re-partitions the domain of the current variable at every point in search. Consider the variable V_2 of the CSP example from Figure 2.1 (recalled here).



Assume that we are performing search with the ordering V_1, V_2, V_3, V_4. We have already instantiated V_1 to (d) and are now considering V_2. NIC-FC partitions the domain of V_2 according to NI_C computed before search began. At this point, the domain of variable V_3 contained the value d. Now, however, d has been removed from the variable V_3. NIC-FC is unable to take advantage of this value being removed, and produces three equivalence classes for V_2: (c), (d), and (e, f). However, because DNPI is recomputed at every step in search, it produces only two equivalence classes for V_2: (c) and (d, e, f). A value pruned by some past assignment (in this case, the value d was pruned from the domain of V_3 when we assigned (d) to V_1) may be a value that prohibited two equivalence sets in the current variable from being equivalent (here (d) and (e, f) in V_2). NIC-FC is unable to detect such equivalence, but DNPI-FC finds and benefits from it.

DNPI-FC finds all interchangeabilities that NIC-FC finds. DNPI-FC will never separate two values that NIC-FC joins in an equivalence class. NI_C is computed based on the values in the CSP before beginning search. These values can only be deleted by the performance of search—new values will never be added to the domain of a variable. As we saw above, the deletion of values can only cause two equivalence classes to merge (the equivalence classes (d) and (e, f) were merged), if it affects them at all. Therefore, the process of forward checking during search only causes more interchangeability. Any interchangeability that NIC-FC utilizes was in the CSP at the beginning and will not disappear. Therefore DNPI-FC will find the same interchangeable value. It is important to remember that this *only* holds when static variable-value ordering is employed.

Based on Theorem 3.3.2, the additional comparisons shown in Figure 3.7 are easy to prove. We include the theoretical comparison of CPR-FC to DNPI-FC here for completeness, but we do not include CPR-FC in the future. These results hold for all static variable-value orderings, provided the orderings are the same for all strategies and search computes all solutions.

Theorem 3.3.3. Every node visited by DNPI-FC is visited by CPR-FC and by NIC-FC, and every

Number of Nodes Visited	Number of Constraints Checks	Number of Solution Bundles
FC $\xrightarrow{\geq}$ NIC-FC $\xrightarrow{\geq}$ DNPI-FC CPR-FC $\xrightarrow{\geq}$ DNPI-FC	NIC-FC FC $\xrightarrow{\geq}$ CPR-FC = DNPI-FC	FC $\xrightarrow{\geq}$ NIC-FC $\xrightarrow{\geq}$ CPR-FC = DNPI-FC

Figure 3.7: *Comparing bundling strategies.*

node visited by NIC-FC is visited by FC. Thus, the following orders hold:

$$\texttt{NV}(\text{FC}) \geq \texttt{NV}(\text{NIC-FC}) \geq \texttt{NV}(\text{DNPI-FC}) \text{ and } \texttt{NV}(\text{CPR-FC}) \geq \texttt{NV}(\text{DNPI-FC}).$$

Each search strategy, when finding one solution, visits one node for each value in the domain of the current variable (which may have been modified by forward checking from past variables). FC does no bundling and so demonstrates the worst case for NV. In the case of bundling, each equivalence class is treated as a single value, therefore if any bundling at all can be performed, NV will be smaller. We know from Theorem 3.3.2 that a variable in DNPI-FC never has more equivalence classes than in NIC-FC, so the number of nodes visited by DNPI-FC cannot be more than that of NIC-FC, and from Theorem 3.3.1 that NIC-FC never has more equivalence classes than the number of values in the domain.

Because CPR-FC performs dynamic bundling *after* generating all future subproblems for a current variable (the JDT does this before), it may visit more nodes than DNPI-FC and thus cannot be compared with NIC-FC. This difference in the number of nodes visited by CPR-FC and DNPI-FC is bounded by $\mathcal{O}(na)$[2].

Theorem 3.3.4. For the number of constraint checks (CC), the following orders hold:

$$\texttt{CC}(\text{FC}) \geq \texttt{CC}(\text{CPR-FC}) = \texttt{CC}(\text{DNPI-FC}) \quad .$$

However, CC(NIC-FC) is comparable to neither CC(FC) nor CC(DNPI-FC).

The domains of future variables in DNPI-FC are retrieved from the JDT, requiring the same number of constraint checks as CPR-FC requires to forward check over the domain of future variables. Further, because of the bundling, this number cannot exceed that required by FC. However, because interchangeability is computed before search in NIC-FC (which requires constraint checks), and is not used for forward checking, it is not comparable to either FC or DNPI-FC.

[2]The presence of a bound between the NV(CPR-FC) and NV(DNPI-FC) was suggested by a reviewer.

Theorem 3.3.5. For the number of solution bundles generated ($\texttt{SB}$), the following total order holds:

$$\texttt{SB}(\text{FC}) \geq \texttt{SB}(\text{NIC-FC}) \geq \texttt{SB}(\text{CPR-FC}) = \texttt{SB}(\text{DNPI-FC}).$$

A solution bundle is composed of a set of equivalence classes, one per variable, that do not conflict with any constraint in the CSP. Because FC produces no bundling of values, every solution bundle will be a single, distinct solution and the $\texttt{SB}$ will be merely the number of solutions. Because NIC-FC and DNPI-FC have the potential of bundling, the solutions may contain more than one value in the equivalence class assigned to any of the variables. This allows more than one actual solution to be combined into one solution bundle. Because the total number of solutions are the same, NIC-FC and DNPI-FC are guaranteed to generate no more solution bundles than FC. Further, DNPI-FC finds fewer equivalence classes than NIC-FC, which implies that the number of solution bundles will be similarly fewer. Regarding the solutions of a CSP, the following additional claims can be made:

Theorem 3.3.6. Every solution bundle generated by FC is part of at most one solution bundle generated by NIC-FC or DNPI-FC.

As a search proceeds systematically through the search space, every combination of variable-value pairs is examined at most once. Depth first search progresses to the bottom of the search tree, visiting nodes and backtracking. In this process, it never re-visits the exact same path in the tree. Similarly, as NIC-FC or DNPI-FC perform search, they systematically visit the search space, bundling where possible, but never re-visiting a path. We know that every solution generated by FC is unique because of this property. By the same property, we know that every bundle is unique and that no two solution bundles can contain the same solution (it would have been pruned from the search tree the first time it participated in a solution).

Lemma 3.3.1. Every solution bundle generated by NIC-FC is part of at most one solution bundle generated by DNPI-FC.

Each equivalence set in a variable during a DNPI-FC search procedure is either the same equivalence class assigned by NIC-FC search, or it is a combination of two (or more) equivalence classes from NIC-FC search. This is shown in Theorem 3.3.2. It follows directly from that fact that every solution that comes from a DNPI-FC search is also either a solution found by NIC-FC search or the combination of two (or more) solution bundles found by NIC-FC search.

Lemma 3.3.2. The solutions bundles found by NIC-FC and DNPI-FC provide a partitioning of the solution space.

This follows directly from Theorem 3.3.6.

3.3.1 Notable omissions

While the statements made above hold true, there are a number of statements that we cannot make:

- We cannot claim optimal bundling. However, our bundling strategy does produce a partition of the solution space into similar solutions. In any particular solution bundle produced by our strategies, there may be solutions that only differ on one variable, and thus could join the bundle. This idea was exploited in [Lesaint 1994] but allows any solution to reside in an arbitrary number of solution bundles, thus losing the partitioning of the solution space.

- We do not make any theoretical claims about dynamic variable ordering. While dynamic variable ordering performs in general better than static variable ordering (as we shall see in Chapter 4), we cannot guarantee anything due to the non-deterministic nature of dynamic variable ordering.

- Similarly, we make no claims about the performance of bundling when finding one solution. This problem will be addressed in Chapter 5.

3.4 Empirical demonstration of the proofs

As just shown, when using static variable-value ordering and solving for all solutions, these statements hold: DNPI-FC performs stronger bundling than NIC-FC, which allows it to visit fewer nodes and generate fewer, thus larger, solution bundles. All bundling search strategies visit fewer nodes and generate fewer bundles than FC, and DNPI-FC checks fewer constraints. Finally bundling search strategies provide a partitioning of the solution space of a CSP.

In order to verify and support these theoretical claims from Section 3.3, we implemented and ran elementary backtrack search with forward checking (FC), static bundling strategies with NI_C (NIC-FC), and dynamic bundling strategy DNPI (DNPI-FC). We used a static variable ordering according to the least domain (SLD) [Haralick and Elliott 1980] heuristic. In order to reduce the duration of our experiments to a reasonable value, we chose to make *all* problems arc-consistent with AC-3 [Mackworth *et al.* 1985] before search is begun. Since this is done uniformly in all experiments and for all strategies, it does not affect the quality of our conclusions. We conducted tests on the puzzles and randomly generated problems introduced in Section 2.2, and compared the strategies with respect to the four evaluation criteria given in Section 3.2.

3.4.1 Puzzles

We first discuss a case where interchangeability seems to have no visible profit and show that it also does not hurt the search strategy. It is well-known that the N-Queens problem may not benefit from 'simple' interchangeability such as neighborhood interchangeability [Freuder and Sabin 1997; Benhamou 2000]; thus, we expect it to not contain NI_C or NPI. We noticed this is also true for puzzles, such as Zebra[3]. We say that these puzzles 'resist bundling.' In both cases, the preprocessing step in NI_C [Haselböck 1993] adds to the number of constraint checks while drawing no benefits.

	Search	NV	CC	SB	**Time** [ms]
8-Queens	FC	2186	15508	92	290
	NIC-FC	2186	22196	92	1020
	DNPI-FC	2134	15508	92	540
Zebra-1	FC	209	972	1	30
	NIC-FC	209	4798	1	190
	DNPI-FC	175	972	1	40
Zebra-210	FC	285668	1803980	210	47050
	NIC-FC	285668	2018342	210	111690
	DNPI-FC	268812	1803980	210	51980

Table 3.2: *Results on puzzles.*

Table 3.2 reports the results of tests on the 8-Queens, Zebra-1, and Zebra-210 (each introduced in Section 2.2.1). The entries in the table support *each* of the theorems in Section 3.3. Further,

[3]To handle such cases, we should investigate other types of symmetries, such as isomorphic interchangeability.

1. NI_C degrades the number of constraint checks but not that of nodes visited and may sensibly degrade time performance by an order of magnitude.

2. Even when no *solution bundling* is possible, DNPI, which bundles dynamically, never does more constraint checks or visit more nodes than FC. Moreover,

3. DNPI-FC visits even fewer nodes, because it is bundling 'no-goods.'

4. Time performance of dynamic bundling is slightly worse, but of the same order of magnitude as FC.

In summary, DNPI is shown to be worthwhile even for known counter-examples where NI_C cannot possibly be effective.

3.4.2 Random problems

To generate the random problems, we used the random-CSP generator of [Bacchus and van Run 1995] with $\langle n, a, p, t\rangle$ as $\langle 10, 5, \{.1, .5., 9\}, \{0.04, 0.12, \dots, 0.92\}\rangle$. We generated 20 random instances for each value of density and tightness, and averaged the values of `NV`, `CC`, `SB`, and CPU time over the 20 instances. Numerical results for $t \leq 0.44$ are reported in Table 3.3. For $t > 0.44$, all CSPs were found un-solvable by the arc-consistency preprocessing step. We do not show them here.

t	p	**Nodes Visited** `NV` 0.1	0.5	0.9	**Constraint Checks** `CC` 0.1	0.5	0.9	**Solution Bundles** `SB` 0.1	0.5	0.9	**Time** `[ms]` 0.1	0.5	0.9
0.04	FC	7356600	4325913	2573138	2708141	3978502	3364460	5710371	3256031	1859533	114577	122483	93760
	NIC-FC	2530	31813	142848	4320	55799	289700	624	9177	46403	280	4044	21760
	DNPI-FC	2038	20098	77460	12043	161051	685384	481	5353	23178	440	5378	23239
0.12	FC	2638985	516245	106796	1310551	539847	214263	1890654	324581	53049	52198	17652	6389
	NIC-FC	30701	110709	61365	46913	219258	152561	9166	36268	18875	2842	13458	9533
	DNPI-FC	19835	61197	35587	63797	249995	163366	5607	17773	9301	2621	9929	6963
0.20	FC	828805	56534	4593	561741	90370	17609	520957	25361	1138	21904	2957	580
	NIC-FC	47531	29802	4143	95700	65784	20156	14278	8240	752	4892	3796	1062
	DNPI-FC	28049	16971	3193	77275	57899	16781	7895	3942	467	3316	2456	766
0.28	FC	230354	5926	372	156328	14278	3014	130375	1560	19	6388	459	108
	NIC-FC	23425	4558	369	42816	15668	6929	5848	788	16	2240	850	294
	DNPI-FC	15078	3210	333	36797	12284	2975	3514	472	11	1625	560	148
0.36	FC	73610	535	68	67515	2587	839	33493	50	0	2617	90	45
	NIC-FC	11637	488	72	22871	4998	4711	2879	30	0	1204	196	243
	DNPI-FC	7670	432	66	18888	2504	838	1698	23	0	802	111	64
0.44	FC	17784	136	12	16665	918	167	6030	4	0	691	46	36
	NIC-FC	3283	132	12	7121	3256	1804	614	2	0	380	137	106
	DNPI-FC	2160	119	12	5077	898	178	374	2	0	245	50	42

Table 3.3: *Results on random problems.*

Taking a close look at Table 3.3, we notice the following:

Nodes Visited `NV` (Theorem 3.3.3): We clearly see in Table 3.3 any bundling strategy always outperforms FC. We also see that `NV`(DNPI-FC) is always less than `NV`(NIC-FC).

Constraint Checks `CC` (Theorem 3.3.4): Table 3.3 shows that it is not always a good idea to compute interchangeability as a preprocessing step: NIC-FC may wastefully increase the number of constraint checks, see $\langle t = .20, p = .9\rangle$ and $\langle t = .28, .36, .44, p = .5, .9\rangle$. Moreover, we see that for $t \geq .20$ dynamic bundling is always superior to static bundling (NI_C) in spite of the repeated computation of the JDT.

Number of Solution Bundles `SB`**:** All bundling strategies obviously bundle better than a non-bundling strategy, as stated in Theorem 3.3.4. The number of solution bundles for FC, which does no bundling, is in fact the number of solutions to the CSP. We also see that dynamic bundling (DNPI) consistently produces fewer bundles (better bundling) than static bundling (NI_C).

CPU time: By looking at the right-most column in Table 3.3, we see that bundling is generally worthwhile, give or take experimental precision, except sometimes for NIC-FC (e.g., $\langle t = .12, p = 0.9\rangle$, $\langle t \leq .20, p = .5, .9\rangle$). Notice that for $t \geq .12$ dynamic bundling consistently outperforms static bundling (NIC-FC).

Summary

So far, we have introduced dynamic bundling with DNPI and its collaboration with search strategies. We have compared three search strategies (FC, NIC-FC and DNPI-FC) according to the standard evaluation criteria: Constraint Checks (`CC`), Nodes Visited (`NV`), number of Solution Bundles (`SB`) and CPU time. We prove theoretically that the statements made in Figure 3.7 (recalled here) hold.

Number of Nodes Visited	Number of Constraints Checks	Number of Solution Bundles
FC $\xrightarrow{\geq}$ NIC-FC $\xrightarrow{\geq}$ DNPI-FC CPR-FC $\xrightarrow{\geq}$ DNPI-FC	NIC-FC FC $\xrightarrow{\geq}$ CPR-FC = DNPI-FC	FC $\xrightarrow{\geq}$ NIC-FC $\xrightarrow{\geq}$ CPR-FC = DNPI-FC

Additionally, we demonstrate each of these proofs empirically. These comparisons show that our strategy does not cause any degradation *even when no bundling is possible*. However, this demonstration is made in a limited domain—finding all solutions with static variable ordering. We also establish and prove the equivalence of DNPI-FC and CPR [Hubbe and Freuder 1989]. This equivalence was independently stated by [Silaghi *et al.* 1999], but not proven. To continue our work, we enter a less deterministic realm, where theoretical results like those of Section 3.3 are not possible. Therefore, all remaining results will be drawn empirically.

Chapter 4

Bundling with dynamic variable ordering

In the context of finding all the solutions to a CSP, we proved that dynamic bundling is *always* worthwhile, provided the same ordering of variables and values is used for all strategies. More

Num. of Nodes Visited	Num. of Constraints Checks	Num. of Solution Bundles
FC $\xrightarrow{\succcurlyeq}$ NI_C $\xrightarrow{\succcurlyeq}$ DNPI	NI_C FC $\xrightarrow{\succcurlyeq}$ DNPI	FC $\xrightarrow{\succcurlyeq}$ NI_C $\xrightarrow{\succcurlyeq}$ DNPI

Figure 4.1: *Theoretical comparisons of bundling strategies.*

specifically, we established theoretically (recalled in Figure 4.1) and empirically that neither non-bundling search nor static bundling can perform better than dynamic bundling in terms of the quality of bundling (i.e., number of solution bundles generated) and in terms of the standard evaluation criteria for search (i.e., number of constraint checks and number of nodes visited). CPU time measurements were in line with the other criteria. In this chapter, we explore another opportunity to improve the performance of search—variable and value ordering.

4.1 Variable-value ordering heuristics

The order of variable expansion and the order of value assignment are known to fundamentally affect the performance of search and have been extensively studied [Tsang 1993b]. Two general principles guide these choices. Roughly speaking, they consist in first choosing the most *constrained* variable and the most *promising* value. An ordering heuristic can be applied as a preprocessing step to determine a *static ordering* that is maintained during the search process. It can also be computed during search, yielding a *dynamic ordering*. Finally, both variables and values can be dynamically ordered by a heuristic, which yields *dynamic variable-value ordering*[1]. An example of each of these three major ordering strategies is given in Figure 4.2.

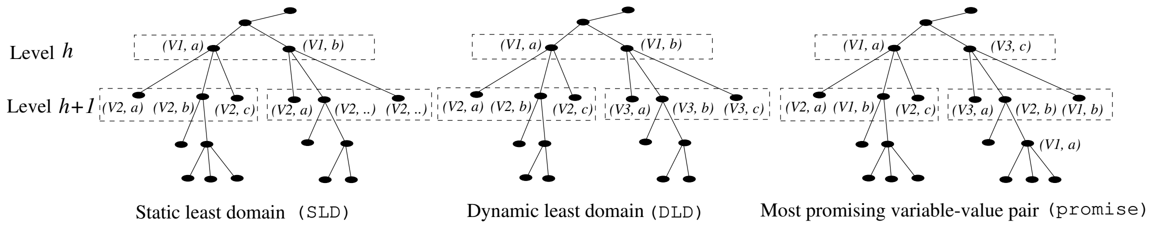


Figure 4.2: *Effects of ordering strategies on the search tree.*

4.1.1 Static least domain (`SLD`)

Variables are sorted prior to search according to increasing domain size; they are instantiated in this order during search. Nodes at any particular level in the search tree represent variable-value pairs (vvp) pertaining to the same variable, as shown in Figure 4.2 (left).

4.1.2 Dynamic least domain (`DLD`)

At each level of the search tree, the variable with the smallest *remaining* domain is chosen for instantiation. This heuristic yields a tree in which the order of instantiation of the variables may vary from branch to branch. Since the assignment of a value to the variable with the smallest domain can only decrease the size of its domain, the same variable is necessarily chosen again when another alternative is sought at the same level in the tree. When its domain is empty, backtracking occurs. Consequently, every value of the selected variable is tried before any other new variable can be considered. As a result, any two nodes in the tree that have the same parent represent variable-value pairs pertaining to the same variable, as shown in Figure 4.2 (center).

4.1.3 Most promising variable-value pair (`promise`)

A truly successful heuristic for dynamic variable-value pair is the one proposed by Geelen [1992] and executes as a three-step procedure:

[1] When computing all solutions, value ordering is futile since all possible assignments are eventually tested.

1. First it computes the promise of each value of every future variable. The promise of a vvp is defined as the product of the number of remaining values of the other future variables. For example, suppose A, B, and C are future variables with respective remaining domains: $D_A = \{1, 2, 3\}$, $D_B = \{2, 3, 4\}$, and $D_C = \{1, 3, 4\}$. Also, suppose that forward checking on the vvp $(A, 1)$ leaves $D_B = \{2, 3\}$, and $D_C = \{1\}$. The promise of vvp $(A, 1)$ is thus $2\times1 = 2$. The promise of each vvp pertaining to every future variable is computed this way.

2. The promise of every variable is then computed as the sum of the promises of its vvps. So, if the promise of $(A, 1)$ is 2, $(A, 2)$ is 1, and $(A, 3)$ is 6, the promise of A is 2+1+6 $= 9$.

3. After computing the promise of every value and the promise of every variable of all future variables, this heuristic expands the variable with the smallest promise (the most constrained variable) and assigns to it the value from its domain with the largest promise (the most promising value). In the above example, suppose that the promise of be B is 15 and that of C 12, then $(A, 3)$ would be the vvp chosen under the `promise` ordering since A has the smallest promise ($= 9$) among A, B, and C while $(A, 3)$ has the largest promise ($= 6$) among $(A, 1)$, $(A, 2)$ and $(A, 3)$.

Informally, the 'promise' of a value for a variable indicates the maximum number of possible remaining solutions if this value was chosen for the variable. The 'promise' of a variable indicates the total number of value sets that can be assigned to all future variables, and is thus an upper bound for the number of different solutions to the CSP at this point in search.

Importantly, as for `DLD`, the order of instantiation of the variables may, and usually does, vary from branch to branch. However, unlike `DLD`, two nodes with the same parent in the tree do not necessarily pertain to the same variable, as shown in Figure 4.2 (right). While this is a more complex ordering than previous ones, it is specifically designed to find *one* solution to a CSP quickly. When finding all solutions, a value ordering like `promise` is overkill, since all possible assignments are eventually tested.

This `promise` heuristic performs a search with nearly minimal backtracks, and has a strong potential for bundling the solution space of a problem. Because `promise` chooses the variable-value pair leaving maximum number of solutions, the domains of the future variables are left as large as possible. Additionally, each of the values in these future domains is consistent with all past assignments. As search progresses, the remaining values are likely to produce large bundles.

However, `promise` may be expensive in terms of CPU time due to the extensive forward checking it performs while choosing the next variable and value to instantiate.

4.2 Combining dynamic ordering heuristics with bundling

An important assumption for the validity of the results of Section 3.3 is that the same variable orderings are maintained across all search strategies. However, a given dynamic variable ordering would, in general, yield different orderings across these strategies since they have different capacities for pruning. When this happens, the results of Section 3.3 can no longer be guaranteed. Moreover, strong, theoretical claims about the relative performance of the search strategies cannot be made. Therefore we conduct empirical evaluations of three different ordering heuristics:

- static variable ordering (with static least domain, `SLD`),
- dynamic variable ordering (with dynamic least domain, `DLD`), and
- dynamic variable-value ordering (with `promise` [Geelen 1992]).

We combine each of these heuristics with standard backtrack search with forward checking and two bundling strategies, static bundling (NIC-FC) and dynamic bundling (DNPI-FC). We evaluate each of these combinations on a battery of puzzles and randomly generated problems.

In Figure 4.3, we review the five search strategies we use as our basis:

1. forward checking with static least domain (FC-`SLD`),
2. forward checking with dynamic least domain (FC-`DLD`),
3. forward checking with dynamic variable-value ordering according to `promise` (FC-`promise`),
4. forward checking with static (NIC-FC-`SLD`) and
5. dynamic (DNPI-FC-`SLD`) bundling.

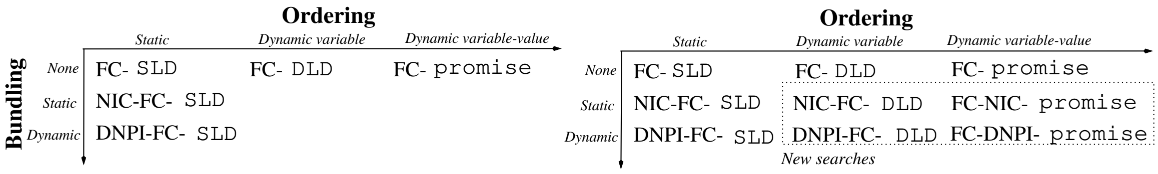


Figure 4.3: *Basic search algorithms (left) and their 'hybridization' with bundling (right) for finding all solutions.*

Each of these five base algorithms build on the pseudo-code for forward checking (FC) given by Prosser [1993] and implement exactly one combination of the following ordering and bundling heuristics.

Ordering: static variable-value ordering, dynamic variable/static value ordering, and dynamic variable-value ordering.

Bundling: non-bundling forward checking search, static bundling, and dynamic bundling.

We introduce four new search algorithms that combine one ordering and one bundling strategy of the strategies listed above. Each of these algorithms are then tested (including the five base algorithms) and their performance compared.

We assume familiarity with FC-`SLD` [Haralick and Elliott 1980], FC-`DLD` [Bacchus and van Run 1995] and FC-`promise` [Geelen 1992]. Below, we describe, as pseudo-code, the enhancements needed to generate the new dynamic ordering algorithms (i.e., NIC-FC-`DLD`, NIC-FC-`promise`, DNPI-FC-`DLD` and DNPI-FC-`promise`) starting from their respective static ordering procedures (i.e., NIC-FC-`SLD` and DNPI-FC-`SLD`).

To modify a strategy from a static ordering to a dynamic ordering, we introduce a new function, `NextVar`. `NextVar` takes as input the lists of future variables and that of past variables (needed to find the boundary of change in DNPI) and returns a choice for the next expansion. For static orderings, `NextVar` merely pops the first variable from the list of future variables sorted in increasing domain size. For dynamic orderings, we specialize `NextVar` in three ways: `NextVar-DLD`, `NextVar-NIC-promise`, and `NextVar-DNPI-promise` as shown in Table 4.1 below.

As specified above, `NextVar-DLD` returns the choice for the next variable according to the heuristic in place. In our case, this is the variable with the smallest domain. `NextVar-NIC-`

Search	NIC-FC-`DLD` and DNPI-FC-`DLD`	NIC-FC-`promise`	DNPI-FC-`promise`
NextVar	`NextVar-DLD`	`NextVar-NIC-promise`	`NextVar-DNPI-promise`
Pseudo-code	Figure 4.4	Figure 4.5	Figure 4.6
Output	The next variable	The next variable-value pair and information for forward checking	

Table 4.1: *Procedures for incorporating dynamic variable ordering with bundling*

`promise` and `NextVar-DNPI-promise` return the next variable-value pair (where a value is a bundle) and the filtered domains for each of the corresponding future variables.

NextVar-DLD (*Future-Vars*, *Past-Vars*):

Begin
best-var ← nil
least-domain ← 0
/* choose the variable with smallest domain */
For each variable V_i in *Future-Vars*
    if V_i domain has fewer elements than *least-domain*
        best-var ← V_i
        least-domain ← number of elements in domain of V_i
return *best-var*
End

Figure 4.4: *Finding the next variable to expand using* `DLD`.

Recall that FC-`promise` and DNPI-FC both perform forward checking implicitly. With `promise`, the remaining problem size for each possible value (or bundle) in each possible variable is calculated, and the most promising value in the least promising variable is chosen. Similarly for DNPI-FC, the JDT for a given variable provides all the future variables and their remaining domains. Therefore, when a variable-value pair is chosen, forward checking need not be executed.

Each search calls its own `NextVar` function, tailored for that particular search. It then uses the information returned to proceed with search. As we will see in the next section, the search strategies that use bundling indeed end up with a smaller search space, yielding a more effective search.

NextVar-NIC-promise (*Future-Vars*, *Past-Vars*):

Begin
best-var $\leftarrow$ nil
best-bundle $\leftarrow$ nil
min-var-promise $\leftarrow$ big-number
/* choose the variable with minimum promise*/
For each variable V_i in *Future-Vars*
  promise-var $\leftarrow$ 0
  past-constraints $\leftarrow$ all constraints between V_i
    and any variable in *Past-Vars*
  future-constraints $\leftarrow$ all constraints between V_i
    and any variable in *Future-Vars*
  Partition domain of V_i according to NIC on intersection of
    all *future-constraints*
  max-bundle-promise $\leftarrow$ 0
  local-best-bundle $\leftarrow$ nil
  /* choose the bundle with the maximum promise */
  For each bundle b in V_i.
    promise-bundle $\leftarrow$ 1
    For each variable V_j in path of JDT
      $left \leftarrow$ domain remaining for V_j
      promise-bundle $\leftarrow$ *promise-bundle* $\times left$
    if (*promise-bundle* $>$ *max-bundle-promise*)
      local-best-bundle $\leftarrow b$
    promise-var $\leftarrow$ *promise-var* + *promise-bundle*
  if (*promise-var* $<$ *min-promise-var*)
    best-var $\leftarrow V_i$
    best-bundle $\leftarrow$ *local-best-bundle*
return *best-var, best-bundle*, and *Future-Vars*
End

Figure 4.5: *Finding the next variable to expand using* `promise` *in NIC-FC.*

```
NextVar-DNPI-promise (Future-Vars, Past-Vars):

Begin
best-var ← nil
best-bundle ← nil
min-var-promise ← big-number
/* choose the variable with minimum promise*/
For each variable V_i in Future-Vars
    promise-var ← 0
    Boundary of change S ← V_i ∪ Past-Vars
    Partition domain of V_i according to NPI according to S
    /* Now, each bundle has an associated JDT */
    max-bundle-promise ← 0
    local-best-bundle ← nil
    /* choose the bundle with the maximum promise */
    For each bundle b in DNPI partition of V_i
        promise-bundle ← 1
        For each variable V_j in path of JDT
            left ← domain remaining for V_j
            promise-bundle ← promise-bundle ×left
        if (promise-bundle > max-bundle-promise)
          local-best-bundle ← b
        promise-var ← promise-var + promise-bundle
    if (promise-var < min-promise-var)
      best-var ← V_i
      best-bundle ← local-best-bundle
return best-var, best-bundle, and Future-Vars
End
```

Figure 4.6: *Finding the next variable to expand using* `promise` *in DNPI-FC.*

4.3 Empirical data and analysis

Table 4.2 reports the results of tests on the 8-Queens and three Zebra problems (Zebra-1, Zebra-11 and Zebra-210). Recall that these numbers (1, 11 and 210), represent the number of solutions to that version of the Zebra problem. For a more thorough explanation of the differences, see Section 2.2.

	Search	**Orderings**	NV	CC	Time [ms]	SB
8-Queens	FC	SLD	2186	15508	290	0
		DLD	1215	10243	200	0
		promise	869	391982	3150	0
	NIC-FC	SLD	2186	22196	1020	92
		DLD	1209	16708	570	92
		promise	923	190901	3300	92
	DNPI-FC	SLD	2134	15508	540	92
		DLD	1216	10356	290	92
		promise	824	177526	3520	92
Zebra-1	FC	SLD	209	972	30	0
		DLD	81	522	30	0
	NIC-FC	SLD	209	4798	190	1
		DLD	81	3612	70	1
		promise	59	56535	1130	1
	DNPI-FC	SLD	175	972	40	1
		DLD	79	522	30	1
		promise	92	60915	1450	1
Zebra-11	FC	SLD	922	4101	110	0
		DLD	377	2133	50	0
	NIC-FC	SLD	922	9527	390	11
		DLD	359	5216	180	11
		promise	302	206213	3950	11
	DNPI-FC	SLD	809	4101	200	11
		DLD	363	2129	100	11
		promise	333	163690	3700	11
Zebra-210	FC	SLD	285668	1803980	47050	0
		DLD	4754	22287	670	0
	NIC-FC	SLD	285668	2018342	111690	210
		DLD	4754	28937	1240	210
		promise	4969	3368816	67840	210
	DNPI-FC	SLD	268812	1803980	51980	210
		DLD	4682	22287	800	210
		promise	2725	1032091	23500	210

Table 4.2: *Performance of Dynamic-Variable ordered search strategies on puzzles.*

To generate the random problems, we used the random CSP generator of Bacchus and van Run [1995], we tested the above listed procedures on random CSPs, with $\langle\, n, a, p, t\,\rangle$ as $\langle$ 10, 5, {.1,

$.5, .9\}, \{.04, 0.12, \ldots, .92\} \rangle$. We generated 20 random instances for each density and tightness, for a total pool of 720 random problems. All results shown are the results of averaging the values of NV, CC, SB, and time over the 20 instances. Numerical results for $t \leq 0.44$ are reported in Table 4.3. For $t > 0.44$, all CSPs were found unsolvable by the arc-consistency preprocessing step prior to search. We do not show them here (all entries are 0).

t		p	Nodes Visited NV			Constraint Checks CC			Solution Bundles SB			CPU time [ms]		
			0.1	**0.5**	**0.9**	**0.1**	**0.5**	**0.9**	**0.1**	**0.5**	**0.9**	**0.1**	**0.5**	**0.9**
0.04	FC	SLD	7356600	4325913	2573138	2708141	3978502	3364460	5710371	3256031	1859533	114577	122483	93760
		DLD	7162999	4107646	2369881	2312579	3101095	2946881	5710371	3256031	1859533	106967	98542	86431
		promise	7462692	4419965	2687086	279786157	184895153	115331018	5710371	3256031	1859533	6045936	3880060	2478908
	NIC-FC	SLD	2530	31813	142848	4320	55799	289700	624	9177	46403	280	4044	21760
		DLD	1930	24811	78900	3755	46765	173030	474	7546	26665	232	3358	12628
		promise	3728	62426	155365	53885	992051	2497716	1310	22404	55570	2261	39204	98276
	DNPI-FC	SLD	2038	20098	77460	12043	161051	685384	481	5353	23178	440	5378	23239
		DLD	1831	24034	69037	11334	197127	642579	450	7336	23098	398	6666	21760
		promise	3376	54702	107864	143230	1980748	3759975	1116	16786	29931	4840	71125	135478
0.12	FC	SLD	2638985	516245	106796	1310551	539847	214263	1890654	324581	53049	52198	17652	6389
		DLD	2401250	434145	78143	997743	427077	137380	1890654	324581	53049	41438	13863	4166
		promise	2641860	521830	107606	103105050	25266114	6371414	1890654	324581	53049	2162975	513907	130028
	NIC-FC	SLD	30701	110709	61365	46913	219258	152561	9166	36268	18875	2842	13458	9533
		DLD	19591	61885	29400	37415	143665	87262	6039	21553	10436	2026	8068	4931
		promise	98362	226774	65725	1898884	4539850	1423742	40358	87655	23267	62678	152548	49223
	DNPI-FC	SLD	19835	61197	35587	63797	249995	163366	5607	17773	9301	2621	9929	6936
		DLD	17926	49903	23612	65550	224590	117149	5475	17131	7989	2626	8716	4760
		promise	68908	120095	26394	1869381	4152037	1139424	26797	38852	6511	66811	146900	38804
0.20	FC	SLD	828805	56534	4593	561741	90370	17609	520957	25361	1138	21904	2957	580
		DLD	673830	38434	2442	294998	52117	8726	520957	25361	1138	12208	1520	294
		promise	819046	53895	3702	36397138	3973663	579505	520957	25361	1138	728644	77246	11054
	NIC-FC	SLD	47531	29802	4143	95700	65784	20156	14278	8240	752	4892	3796	1062
		DLD	19341	13206	1708	40322	39568	11850	5903	4433	450	2004	1893	573
		promise	147343	38546	3204	3194075	1026862	168891	59656	13211	826	100052	32347	5016
	DNPI-FC	SLD	28049	16971	3193	77275	57899	16781	7895	3942	467	3316	2456	766
		DLD	18279	11167	1513	52768	41371	8483	5652	3599	374	2248	1762	370
		promise	97874	23072	1938	2788417	1059954	178154	37028	6190	346	94764	34698	5408
0.28	FC	SLD	230354	5926	372	156328	14278	3014	130375	1560	19	6388	459	108
		DLD	174763	2931	136	91718	6344	1497	130375	1560	19	3686	200	76
		promise	241875	4826	137	13629920	759178	92807	130375	1560	19	264552	13740	1742
	NIC-FC	SLD	23425	4558	369	42816	15668	6929	5848	788	16	2240	850	294
		DLD	9393	1551	129	24228	8398	5501	2820	409	11	1038	418	198
		promise	96066	4005	133	2299165	223841	46048	36775	1072	16	68666	6224	1289
	DNPI-FC	SLD	15078	3210	333	36797	12284	2975	3514	472	11	1652	560	148
		DLD	8343	1370	126	23529	5819	1509	2480	339	9	995	293	87
		promise	57032	2954	111	1782813	249061	50094	20093	597	9	57218	7694	1496
0.36	FC	SLD	73610	535	68	67515	2587	839	33493	50	0	2617	90	45
		DLD	45584	207	29	23608	1210	540	33493	50	0	1012	52	38
		promise	72097	308	16	5013315	138124	26606	33493	50	0	93314	2398	608
	NIC-FC	SLD	11637	488	72	22871	4998	4711	2879	30	0	1204	196	243
		DLD	3392	171	29	9144	3693	4409	940	22	0	390	132	206
		promise	26210	253	16	692436	56570	22430	9167	37	0	20253	1410	582
	DNPI-FC	SLD	7670	432	66	18888	2504	838	1698	23	0	802	111	64
		DLD	3223	170	29	8344	1206	543	880	21	0	359	66	46
		promise	10443	286	17	368990	68995	21878	3298	26	0	11369	1935	586
0.44	FC	SLD	17784	136	12	16665	918	167	6030	4	0	691	46	36
		DLD	8687	38	6	5839	400	121	6030	4	0	251	26	32
		promise	15882	51	3	1617320	40482	5329	6030	4	0	28539	682	142
	NIC-FC	SLD	3283	132	13	7121	3256	1804	614	2	0	380	137	106
		DLD	1013	36	7	3875	2750	1662	252	2	0	171	100	96
		promise	7132	47	2	265454	24077	4998	2254	3	0	7077	634	239
	DNPI-FC	SLD	2160	119	13	5077	898	178	374	2	0	245	50	42
		DLD	940	39	6	2795	417	115	230	2	0	131	36	45
		promise	1996	286	3	90665	27266	4751	529	3	0	2670	758	166

Table 4.3: *Results of search for all solutions of random problems.*

We show graphs demonstrating CPU time and SB in Figure 4.7. On the graphs, SLD and DLD ordering heuristics are shown, but promise ordering heuristics are omitted. An inspection of Table 4.3 shows that the CPU times reported from promise are up to two orders of magnitude larger than those of SLD and DLD ordering heuristics. In order to clearly show the comparison between SLD and DLD, we omit promise from the graph. From the entries and charts in Tables 4.2 and 4.3 and Figure 4.7, we summarize our observations as follows:

Observation 4.3.1. Promise is not a good ordering heuristic for finding all the solutions to a CSP.

Its performance is always poor, especially in terms of CPU time. This is true both in general and also when compared with any non-promise based SLD or DLD strategy. This holds for non-bundling, static bundling, and dynamic bundling. Even though it reduces the number of nodes visited (when $t > 0.28$), it uses an unusually large number of constraint checks, as shown in both tables.

Observation 4.3.2. Bundling is worthwhile.

This is made clear especially in Table 4.3, where we see that the FC search strategies are consistently beaten, on all criteria, by both NIC-FC and DNPI-FC strategies. Further:

Observation 4.3.3. Dynamic bundling (DNPI-FC) is always better than static bundling (NIC-FC) in terms of Nodes Visited (NV) and Solution Bundles (SB), and usually better than the NIC-FC search strategies in terms of Constraint Checks (CC) and CPU time when the problems are not too loose ($t \geq 0.2$). This holds for both static and dynamic bundling.

Observation 4.3.4. Dynamic ordering (DLD) is almost always better than static ordering (SLD).

Summary

Ordering strategies and bundling mechanisms are orthogonal processes for improving the performance of search. The former allows a better navigation in the search space and the latter shrinks its size. We demonstrate that both are successful in making search strategies run faster, and we propose a combination that we prove empirically to be worthwhile.

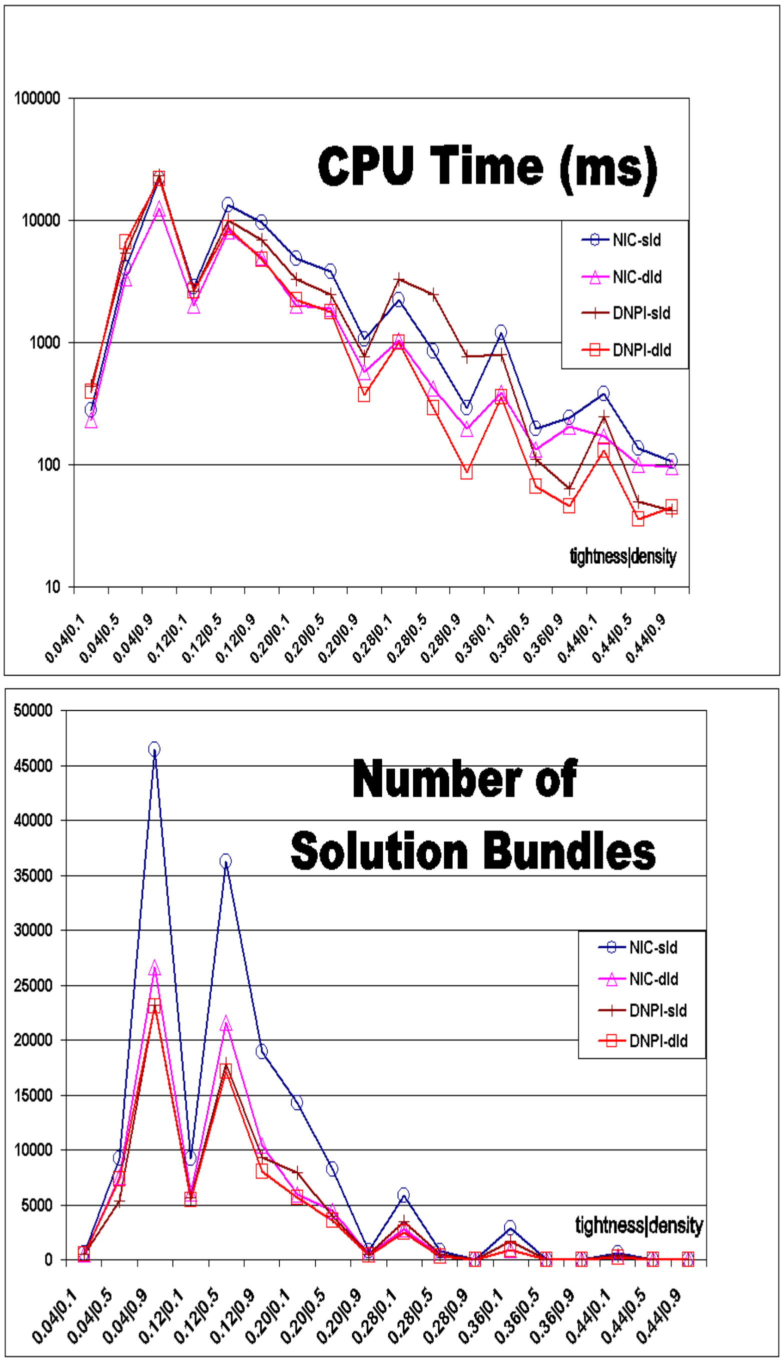


Figure 4.7: *Comparison of CPU time (top) and solution bundling (bottom) for four search strategies.*

Chapter 5

Finding one solution (bundle)

In the previous chapters, we have established that dynamic interchangeability improves the performance of forward checking search for all solutions to a CSP (Section 3.4.2). We guarantee this advantage theoretically with a static variable ordering (`SLD`) in Section 3.3. Further, we demonstrate that this advantage holds in the context of dynamic variable ordering (`DLD`) and dynamic variable-value ordering (`promise`) in Section 4.3. This counters the conventional wisdom that claims dynamic bundling is too costly to be worthwhile. In this chapter we address the task of finding a *first* solution, and the conventional wisdom that this also is too costly when bundling dynamically. We show that dynamic bundling *is* worthwhile, even when finding only one solution. Additionally, we propose two new variable-value ordering heuristics designed to work with bundling strategies for finding one solution and show their performance. We also test, for finding one solution, the nine search algorithms previously discussed in Section 4.2 and here.

Ordering

	Static	*Dynamic variable*	*Dynamic variable-value*
None	FC-`SLD`	FC-`DLD`	FC-`promise`
Static	NIC-FC-`SLD`	NIC-FC-`DLD`	FC-NIC-`promise`
Dynamic	DNPI-FC-`SLD`	DNPI-FC-`DLD`	FC-DNPI-`promise`

New searches

5.1 Ordering strategies for one solution

First we propose two new ordering heuristics specifically designed to work with bundling strategies. These are LeastDomain-MaxBundle (`LD-MB`) and Max-Bundle (`Max-Bundle`). Both can be classified as dynamic variable-value ordering heuristics, where a value is actually a bundle (i.e. an equivalence class). Every bundle is treated as a single value during search. We introduce two new dynamic variable-value ordering heuristics and test their behavior.

LeastDomain-MaxBundle: (`LD-MB`) Recall that static least domain ordering (`SLD`) sorts variables by increasing domain size before search and maintains this order during the search process. `DLD`, a dynamic variable ordering, recomputes this order after each instantiation. A straightforward improvement to `DLD` is to impose an order among the values considered, thus yielding a dynamic variable-value ordering. We propose a new heuristic `LD-MB` that, like `DLD`, chooses the variable V with the smallest domain, but then for V, it chooses the largest bundle induced on the domain of V by the adopted bundling strategy (e.g., NI_C or DNPI). In the case of non-bundling search (FC), `LD-MB` simply collapses to `DLD` since the size of any bundle is one, and in the case of finding all solutions, `LD-MB` also collapses to `DLD`, since each bundle in a particular domain must eventually be considered and expanded. `LD-MB` merely chooses the order in which those bundles are expanded, whereas `DLD` does not. Thus, when all solutions are sought, both strategies yield the same bundles, only in a different order.

Maximum bundle size: (`Max-Bundle`) While `LD-MB` chooses first the least domain variable and for this variable it chooses the the largest bundle for expansion, `Max-Bundle` finds the largest bundle *over all the future variables*, and chooses that bundle, and the variable it belongs to, for expansion. Therefore, it is tempting to think that it will take us to a very large solution very quickly. We will show that this intuition is false, and in fact this is a poor ordering heuristic.

5.2 Experiments

We performed tests on a battery of random problems generated using the problem generator of Bacchus and van Run [1995] with $\langle n, a, p, t\rangle$ as $\langle 10, 5, \{.1, .5, .9\}, \{.04, 0.12, \dots, .92\}\rangle$. We report as usual the of nodes visited `NV` , constraint checks `CC` , first bundle size `FBS` and CPU time. For each measurement point, the results were averaged over 20 instances. In addition to the 10 search strategies promised (five variable-value ordering heuristics $\times$ two bundling strategies), we also include the results of two non-bundling search strategies FC-`SLD` and FC-`DLD` to serve as baseline values of the comparison criteria. We ran each of the 12 search strategies on every instance to find the first bundled solution.

5.3 Discussion

Table 5.1 shows the numerical results. We first give general observations of our two new ordering heuristics, followed by a more specific commentary on each of the evaluation criteria shown.

Dynamic Bundling (DNPI): DNPI-FC is competitive, but not quite as cheap as non-bundling FC for finding one solution. The presence of an exponential number of solutions allows even non-bundling to reach a solution with almost no backtracking. However, in these cases, DNPI finds large bundles of very similar solutions .

Bundling with `LD-MB`: Like `promise`, `LD-MB` is a dynamic variable-value ordering heuristic. However, unlike `promise`, `LD-MB` does not guarantee that any vvp returned will yield a consistent assignment. As a result, backtracking is more common in `LD-MB` than it is in `promise`.

Bundling with `Max-Bundle`: `Max-Bundle` performs terribly. Because it chooses the large bundles first, it leaves only thin bundles in the remaining domains. Though intuition says that choosing a large bundling would lead us to a large solution quickly, these large bundles are generally no-good sets, i.e. they belong to no solutions. In fact, because `Max-Bundle` has a tendency to try to expand large no-good sets before expanding any ‘good’ sets, it requires a large amount of backtracking, and a lot of time is wasted on these sets. This is worsened

Finding one solution bundle													
		NV			CC			FBS			Time [ms]		
t	p	**0.1**	**0.5**	**0.9**	**0.1**	**0.5**	**0.9**	**0.1**	**0.5**	**0.9**	**0.1**	**0.5**	**0.9**
.04	FC-SLD	10	10	10	63	128	184	1	1	1	2	2	4
	FC-DLD	10	10	10	65	133	197	1	1	1	2	4	4
	NIC-FC-SLD	10	10	10	1326	2752	4178	57246	6740	164	43	90	128
	NIC-FC-DLD	10	10	10	1326	2754	4180	54498	7428	958	36	90	133
	NIC-FC-LD-MB	10	10	10	1326	2754	4181	506284	26814	2140	38	95	137
	NIC-FC-promise	10	10	10	2188	4520	6825	847680	101304	21979	90	237	320
	NIC-FC-Max-Bundle	10	10	10	1326	2754	4181	748608	65988	15559	42	101	156
	DNPI-FC-SLD	10	10	10	284	546	778	2179	164	25	8	14	20
	DNPI-FC-DLD	10	10	10	281	549	766	1757	179	25	8	14	20
	DNPI-FC-LD-MB	10	10	10	387	756	1055	491484	22558	1206	10	18	24
	DNPI-FC-promise	10	10	10	3255	5428	7414	847680	101304	21922	84	138	193
	DNPI-FC-Max-Bundle	10	10	10	3312	5700	7537	748608	65988	15560	78	134	177
.12	FC-SLD	10	10	11	60	111	159	1	1	1	1	2	6
	FC-DLD	10	10	10	64	124	170	1	1	1	2	4	8
	NIC-FC-SLD	10	10	10	1339	2775	4210	1771	8	3	37	96	204
	NIC-FC-DLD	10	10	10	1341	2782	4217	1880	25	5	41	93	136
	NIC-FC-LD-MB	10	10	10	1341	2782	4217	3916	80	6	43	98	264
	NIC-FC-promise	10	10	10	3038	5183	7738	7992	397	24	113	174	256
	NIC-FC-Max-Bundle	10	10	10	1341	2776	4210	3905	32	4	46	178	167
	DNPI-FC-SLD	10	10	11	253	435	581	236	22	5	6	10	16
	DNPI-FC-DLD	10	10	10	253	413	557	342	22	10	8	10	14
	DNPI-FC-LD-MB	10	10	10	332	560	728	2933	51	7	8	14	17
	DNPI-FC-promise	10	10	10	2844	4190	5920	8305	364	65	74	113	155
	DNPI-FC-Max-Bundle	10	10	12	3118	3983	5135	4147	75	16	73	94	120
0.2	FC-SLD	10	14	24	56	116	246	1	1	1	2	2	5
	FC-DLD	10	10	12	62	114	174	1	1	1	1	2	3
	NIC-FC-SLD	10	11	33	1347	2787	4359	152	4	1	40	89	138
	NIC-FC-DLD	10	10	12	1351	2793	4251	307	6	2	39	87	142
	NIC-FC-LD-MB	10	10	12	1351	2793	4251	355	7	2	40	103	131
	NIC-FC-promise	10	10	10	3044	5322	7792	2138	40	2	105	172	329
	NIC-FC-Max-Bundle	10	11	36	1348	2789	4378	258	4	1	43	98	178
	DNPI-FC-SLD	10	12	14	232	362	473	109	6	2	4	10	12
	DNPI-FC-DLD	10	10	11	223	331	439	191	5	4	10	8	12
	DNPI-FC-LD-MB	10	10	11	287	450	589	263	9	3	8	11	16
	DNPI-FC-promise	10	10	10	2251	3444	4717	2014	80	9	64	94	123
	DNPI-FC-Max-Bundle	10	15	16	2687	3573	4614	298	9	2	65	86	110
.28	FC-SLD	10	26	66	54	165	654	1	1	1	2	4	16
	FC-DLD	10	11	31	61	113	441	1	1	1	2	4	11
	NIC-FC-SLD	10	27	71	1346	2869	4738	52	2	1	38	98	152
	NIC-FC-DLD	10	13	29	1351	2805	4471	99	3	2	38	90	135
	NIC-FC-LD-MB	10	13	29	1351	2805	4464	121	3	2	40	94	155
	NIC-FC-promise	10	10	15	2904	5148	10419	1749	10	1	98	171	316
	NIC-FC-Max-Bundle	10	28	78	1346	2851	4811	54	2	1	43	111	345
	DNPI-FC-SLD	11	18	63	211	350	852	125	2	1	6	9	27
	DNPI-FC-DLD	10	10	29	204	281	570	130	4	1	2	6	16
	DNPI-FC-LD-MB	10	11	26	265	415	1124	159	4	1	6	11	28
	DNPI-FC-promise	10	10	14	1955	2852	5884	2220	24	3	55	77	150
	DNPI-FC-Max-Bundle	11	28	118	2338	3592	13736	93	3	1	52	85	331
.36	FC-SLD	10	26	66	54	165	654	1	1	1	2	4	16
	FC-DLD	10	11	31	61	113	441	1	1	1	2	4	11
	NIC-FC-SLD	12	42	3	1328	2834	231	25	1	0	34	90	6
	NIC-FC-DLD	10	15	2	1327	2705	223	95	3	0	32	82	5
	NIC-FC-LD-MB	10	15	1	1332	2707	216	106	3	0	35	86	12
	NIC-FC-promise	10	11	1	2635	5103	378	586	3	0	86	162	8
	NIC-FC-Max-Bundle	22	58	4	1352	2899	235	26	2	0	68	143	18
	DNPI-FC-SLD	11	30	1	193	381	19	50	2	0	5	10	0
	DNPI-FC-DLD	10	13	1	174	258	17	51	3	0	4	6	0
	DNPI-FC-LD-MB	10	16	2	231	484	90	49	2	0	4	14	2
	DNPI-FC-promise	10	11	1	1612	2695	158	886	5	0	44	68	4
	DNPI-FC-Max-Bundle	11	74	2	2148	6300	278	91	2	0	46	150	6

Table 5.1: *Finding one solution on random problems.*

by dynamic bundling, which allows even larger bundles (still no-good sets) to appear and be tried before finding a successful solution. Because of this, `Max-Bundle` is *not an effective ordering heuristic.*

5.3.1 Nodes visited (`NV`)

From examining the values of `NV` in Table 5.1, we confirm the following general trend: Fewer nodes are visited by the use of dynamic ordering instead of static ordering (except for `Max-Bundle`).

For $t < 0.2$, all algorithms find a first solution (almost) backtrack-free (i.e., visiting about 10 nodes only). This can be expected, since in loose problems, the number of solutions is exponential in the size of the problem [Bacchus and van Run 1995], and solutions are particularly easy to find.

For $t \geq 0.2$, the performance of `SLD` ordered search strategies starts to deteriorate slightly and that of performance of `Max-Bundle` ordered search strategies deteriorates more seriously. `Max-Bundle` is indeed a bad greedy heuristic: it tries to choose fat value bundles that will almost immediately wipe out domains of remaining future variables, and causes the increase in backtracking effort.

5.3.2 Constraint checks (`CC`)

If we examine `CC` for DNPI-FC search strategies versus NIC-FC strategies, we see that dynamic bundling always outperforms static bundling for `SLD`, `DLD`, and `LD-MB`, and almost always for `promise` (while $t > 0.04$). We have already shown in Sections 3.4.2 and 4.3 that dynamic bundling is significantly preferable to static bundling when looking for all solutions. Here, we show that, *even when looking for the first bundle*, dynamic bundling, in spite of the constraint checking effort necessary for re-bundling, remains a winner. An exception to this rule is the `Max-Bundle` heuristic, in which DNPI-FC is worse than NIC-FC. Note that despite the fact that `Max-Bundle` does not forward check on every value as `promise` does, it still requires as many constraint checks as `promise`, if not more.

Finally, we notice that `DLD` has almost the best performance according to this criterion, although `SLD` and `LD-MB` are quite near competitors.

5.3.3 Bundle size

`Promise` *always outperforms all other heuristics in terms of bundling power.* Table 5.1 shows that `promise` does benefit from DNPI over NI_C, especially when $t > 0.12$. In this sense, `promise` appears to be the absolute best heuristic for finding the fattest first bundle. This is to be contrasted with its poor behavior for finding all solutions noticed in Section 4.3[1].

Dynamic bundling continues to improve the bundle size over static bundling, although this does not hold for $t = 0.04$. The superiority of DNPI over NI_C is most significant in the context of looking for all solutions. Even here, the general trend remains in favor of dynamic bundling, especially with `promise`, and it proves that DNPI remains beneficial even in the context of looking for one solution.

Finally, the good performance of `LD-MB` is worth mentioning. While `LD-MB` does not perform the extensive forward checking used by `promise` on every value, it still makes a reasonable choice for the next vvp to expand and generates a large bundle. Indeed, the constraint checks count resulting from `LD-MB` is smaller than that of `promise` and similar to those for `SLD` and `DLD`, but the bundling yielded by `LD-MB` is significantly superior to that performed by either `SLD` or `DLD`.

5.3.4 CPU time

Let us first consider `Max-Bundle` to justify again why it is not a good heuristic. Although it does not forward check on every value like `promise` does, it is still almost as expensive as `promise` in terms of CPU time. Further, as justified in Section 5.2, it does not benefit from dynamic bundling. Consequently, from this point forth, we will exclude the `Max-Bundle` ordering heuristic from our evaluations.

Excluding `Max-Bundle`, we see that all DNPI-FC search strategies outperform NIC-FC search strategies, despite the re-computation of the domain partitions in DNPI. Dynamic bundling is thus largely justified. Note that `promise` remains more expensive than `SLD`, `DLD`, and `LD-MB` but rewards the effort with the size of the solution bundle it finds.

[1] In the context of finding all solutions, although the performance of `promise` was not satisfactory in terms of cost, we showed in Section 4.3 that it did yield a good quality bundling of the solution space.

5.3.5 Conclusions on ordering heuristics

In light of the experiments conducted in this chapter, we list the following observations to be used as recommendations:

Observation 5.3.1. The best strategy in terms of bundle size is DNPI-FC-`promise`. It is costly in terms of constraint checks and CPU time, but is well worth the effort for finding a first solution bundle: it gives the largest bundle.

Observation 5.3.2. When constraint checks are not cheap, DNPI-`LD-MB` is a good compromise according to all criteria (`NV`, `CC`, `FBS`, and CPU time). DNPI-`SLD` and DNPI-`DLD` are also good alternatives, however the bundles they yield are a bit slimmer than those found by DNPI-`LD-MB`.

Observation 5.3.3. When constraint checks are particularly expensive, DNPI-FC-`DLD` is a great choice since it almost always has the minimum number of constraint checks and CPU time, although the bundles it yields are slimmer than those of DNPI-FC-`promise` and DNPI-FC-`LD-MD`.

Observation 5.3.4. For ease of implementation, it is clear that `DLD` is best, second only to `SLD`.

Summary

We report the following: (1) Dynamic wins over static bundling, especially with `promise`. This advantage is even more visible when problems are not too loose. (2) Although `promise` performs very badly when searching for all solutions, as shown in Chapter 4, it consistently finds the largest first bundle, and nearly always yields a backtrack-free search. (3) `Max-Bundle` is not a good heuristic, contrary to our initial intuition. And (4) `LD-MB` is a competitive new heuristic with relatively few constraint checks, low CPU time, and good bundling.

Chapter 6

Controlling and changing the level of interchangeability

The random generator by Bacchus and van Run [1995] has been valuable to us. With it, we have shown that interchangeability can be found and exploited across a wide variety of constraint tightnesses and probabilities in a CSP. While their generator does not specifically incorporate interchangeability into the random CSP instances, neither does it exclude interchangeability. In order to more fully understand how the interchangeability of a problem effects the effort exerted in finding solutions, we introduce a random generator that controls the amount of interchangeability. We then use this generator to investigate how the performance of both bundling and non-bundling algorithms are affected by the presence (or absence) of interchangeability.

Our random generator that controls interchangeability was inspired by the random generator of Freuder and Sabin [1997]. They control interchangeability by designing constraints from two components, one of which controls interchangeability, and the other of which controls tightness. The resulting constraint is obtained by making the conjunction of the two components and is likely to be tighter than specified and also contain less interchangeability than specified. Our random generator, described below, corrects this problem while maintaining generality.

6.1 Interchangeability levels of a CSP

In order to understand how a problem, or a constraint, can have a level of interchangeability, recall that two values in a variable are interchangeable if they belong to the same equivalence class—that is, if they can be substituted for one another without affecting the assignments of the remaining variables. We define the number of distinct equivalence classes in the domain of a variable as its degree of *domain fragmentation*. A constraint between two variables will partition the values in the domains of both variables into equivalence classes, depending on which sets of values are consistent with each other. Therefore, the degree of domain fragmentation in any particular variable is determined by the constraints incident to that variable. We say then that each constraint *induces* domain fragmentation on its variables. Consequently, a constraint, and by extension a CSP, can have a *degree* of *induced domain fragmentation*, or `IDF`, as we call it here. This will be our measure of interchangeability—a high `IDF` means that the CSP has little or no static interchangeability.

We introduce a generator of random CSPs that allows us to control the level of interchangeability embedded in a problem, in addition to controlling the size of the CSP and the density and tightness of the constraints. Using this generator, we conduct experiments that test the previously listed search strategies across various levels of interchangeability. Recall that `LD-MB` for finding all solutions collapses to `DLD`. Because of their poor behavior in Sections 4.3 and 5.2, we exclude dynamic variable-value orderings (e.g., `promise` and `Max-Bundle`) for finding all solutions and `Max-Bundle`. See Table 6.1:

Problem		**Bundling**		**Orderings**
Finding all solutions	×	$\left\{ \begin{array}{l} \text{FC} \\ \text{NIC} \\ \text{DNPI} \end{array} \right\}$	×	$\left\{ \begin{array}{l} \texttt{SLD} \\ \texttt{DLD} \end{array} \right\}$
Finding first solution	×	$\left\{ \begin{array}{l} \text{FC} \\ \text{NIC} \\ \text{DNPI} \end{array} \right\}$	×	$\left\{ \begin{array}{l} \texttt{SLD} \\ \texttt{DLD} \\ \texttt{LD-MB} \\ \texttt{promise} \end{array} \right\}$

Table 6.1: *Search strategies tested.*

We show that:

1. Both static and dynamic bundling search strategies do indeed detect and benefit from interchangeability embedded in a problem instance.

2. Problems with embedded interchangeability are not easier or more difficult to solve for the naive FC algorithm.

3. Most bundling strategies are affected by the variance of interchangeability. However, DLD ordered search is less sensitive and performs surprisingly well in all situations.

6.2 A generator that controls interchangeability

Recall that a generator of random binary CSPs usually takes as input the following parameters $\langle n, a, p, t\rangle$. The first two parameters, n and a, relate to the variables—n gives the number of variables, and a the domain size of each variable. The second two parameters, p and t control the constraints—p gives the probability that a constraint exists between any two variables (which also determines the number of constraints in the problem $C = p\frac{n(n-1)}{2}$), and t gives the constraint tightness (defined as the ratio of the number of tuples disallowed by the constraint over all possible tuples between the two variables).

In order to investigate the effects of interchangeability on the performance of search for solving CSPs, we must guarantee from the outset that each CSP instance contains a specific, controlled amount of interchangeability. Since interchangeability within the problem instance is determined by the constraint, the main difficulty in generating such a CSP resides in the generation of the constraints. In addition to the above listed parameters, our generator of random instances takes as input the desired degree of induced domain fragmentation, IDF. IDF is a measure of the lack of interchangeability in a problem: a higher IDF means less interchangeability. We base our generator on the following assumptions:

1. All variables have the same domain size and, without loss of generality, the same values.
2. Any particular pair of variables has only one constraint.
3. All constraints have the same degree of induced domain fragmentation.
4. All constraints have the same tightness.
5. Any two variables are equally likely to be connected by a constraint.

6.2.1 Constraint representation and implementation

A constraint that applies to two variables is represented by a *binary matrix* whose rows and columns denote the domains of the variables to which it applies. The '1' entries in the matrix specify the tuples that are allowed and the '0' entries the tuples that are disallowed. Figure 6.1 shows a constraint c, with $a = 5$ and $t = 0.32$. This constraint applies to V_1 and V_2 with domains $\{1, 2, 3, 4, 5\}$. The matrix is implemented as a list of row vectors. Each row corresponds to a value in the domain of V_1. Each constraint partitions the domains of the variables to which it applies into equivalence classes.

$$
\begin{array}{cc}
 & V2 \\
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
V1 \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}
\end{array}
\begin{array}{l}
\leftarrow \textit{row1}\ [11001] \\
\leftarrow \textit{row2}\ [10011] \\
\leftarrow \textit{row3}\ [11001] \\
\leftarrow \textit{row4}\ [11001] \\
\leftarrow \textit{row5}\ [11111]
\end{array}
$$

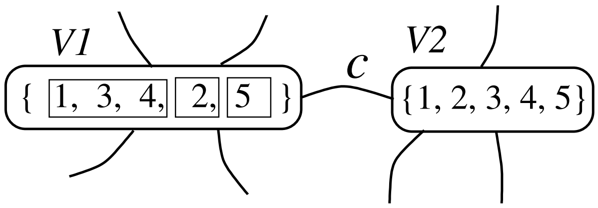


Figure 6.1: *Constraint representation as a binary matrix.* Left: Encoding as row vectors. Right: Domain partition by interchangeability.

The values in a given equivalence class of a variable are consistent with the same set of values in the domain of the other variable. Indeed, c fragments the domain of V_1 into three equivalence classes corresponding to rows $\{1, 3, 4\}$, $\{2\}$ and $\{5\}$ as shown in Figure 6.1.

We measure the degree of induced domain fragmentation (`IDF`) of a constraint as the number of equivalence classes it induces on the domain of the variable *whose values index the rows of the matrix*. Thus, as seen above, the degree of induced domain fragmentation of c for V_1 is $\texttt{IDF} = 3$. We do not control about the domain fragmentation of the remaining variable (V_2, here). Since we control the `IDF` for only one of the variables (the one represented in the rows), our constraints are not *a priori* symmetrical. This is in contrast to the random generator with interchangeability of Freuder and Sabin [1997]. In order to generate problems with *both* the requested level of interchangeability, and tightness (again unlike Freuder and Sabin's [1997]), our generator first generates a constraint with a specified tightness, imposes the degree of `IDF` requested, then confirms that the tightness has not changed.

6.2.2 Constraint generation

Constraint generation is done according to the following five-step process:

Step 1: *Matrix initialization.* Create an $a \times a$ matrix with every entry set to 1.

Step 2: *Tightness.* Set random elements of the matrix to 0 until specified tightness is achieved.

Step 3: *Interchangeability.* Modify the matrix to comply with the specified degree of induced domain fragmentation.

Step 4. *Tightness check.* Test the matrix. If tightness meets the specification, continue. Otherwise, discard this matrix and go to Step 1.

Step 5. *Row Permutation:* Randomly permute the rows of the generated matrix.

When C constraints have been successfully generated ($C = p\frac{n(n-1)}{2}$), each constraint is assigned to a distinct random pair of variables. Note that we do not impose any structure on the generated CSP other than controlling the `IDF` in the definition of the constraints. We also do not guarantee that the CSP returned is connected[1]. Obviously, when $C > \frac{(n-1)(n-2)}{2}$, connectedness is guaranteed. Below, we describe in further detail Steps 3 and 5 of the above process. Steps 1, 2, and 4 are straightforward.

6.2.3 Step 3: Achieving the degree of induced domain fragmentation (`IDF`).

After generating a matrix with a specific tightness, we compute the corresponding degree of induced domain fragmentation by counting the number of distinct row vectors. Each vector is assigned to belong to a particular induced equivalence class. In the matrix of Figure 6.1, $row1$, $row3$ and $row4$ would be assigned to the equivalence class 1, $row2$ assigned to equivalence class 2, and $row5$ assigned to equivalence class 3. When the value of `IDF` requested is different from that of the current matrix, we modify the matrix to increase or decrease its `IDF` by one until the specification is fulfilled.

To increase `IDF`, we select any row from any equivalence class that has more than one element and make it the only element of a new equivalence class. This is done by randomly swapping bits in

[1] Although connectedness is not guaranteed, a random check of CSPs with $C > n - 1$ found no disconnected CSPs.

the vector selected until obtaining a vector distinct from all other rows. Note this operation does not modify the tightness of the constraint. To decrease `IDF`, we select a row that is the only element of an equivalence class and set it equal to any other row. For example in Fig 6.1, setting $row2 \leftarrow row5$ decreases `IDF` from 3 to 2. This operation may affect tightness.

When this is complete, Step 4 verifies that the tightness of the constraint has not changed. If it has, we start over again, generating a new constraint. If the tightness is correct, we proceed to the following step, *row permutation*.

6.2.4 Step 5: Row permutation.

In order to increase the generality of our random constraints and avoid duplicating the domain fragmentation, the rows of each successfully generated constraint are permuted. The permutation process chooses and swaps random rows a random number of times. The input and output matrices of this process obviously have the same tightness and interchangeability—the process does not change these characteristics of the matrix.

6.2.5 Constraint generation in action

An example of this five-step process is shown in Figure 6.2, where we generate a constraint for $a = 5$, `IDF` $= 3$ and $t = 0.32$. Note that Step 3 and Step 4, which control the interchangeability

$$\begin{array}{l} \textit{row1}[11111] \to \\ \textit{row2}[11111] \to \\ \textit{row3}[11111] \to \\ \textit{row4}[11111] \to \\ \textit{row5}[11111] \to \end{array} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Step 1

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Step 2

Set *row4* ← *row2*

Step 3

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Step 4

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Step 5

Figure 6.2: *Constraint generation in action.*

and tightness of a matrix, may fail to terminate successfully. This happens when:

1. No solution exists for the combination of the input parameters. For example, when $a = 5, t = 0.04$, there exists only solutions with `IDF` $= 2$, due to the presence of only one 0 in the matrix.

2. Although a solution may exist, the process of modifying interchangeability in the matrix continuously changes tightness.

To avoid entering an infinite loop in either of these situations, we use a counter at the beginning of the process of constraint generation. After 50 attempts to generate a constraint, it times out, and the generation of the current CSP is interrupted. Our implementation of the generator exhibits a failure rate below 5%, and guarantees constraints with both the specified tightness and degree of induced domain fragmentation.

6.3 Tests and results

We generated two pools of test problems using our random generator, each with a full range of values for induced domain fragmentation, constraint tightness, and constraint probability. The first pool has the following input parameters: $\langle$n, a, p, t, `IDF`$\rangle = \langle$ 10, 5, {.1, .2, ..., 1.0)}, {.04, .12, ..., .92}, {2, 3, 4, 5} $\rangle$. In the second pool, $\langle$n, a, p, t, `IDF`$\rangle$ is $\langle$ 10, 7, {.1, .2, ..., 1.0)} ,{.04, .12, ..., .92}, {2, 3, 4, 5, 6, 7}$\rangle$. The only difference between the two pools is a, or the domain size. In the first pool, each variable has five values ($a = 5$). In the second, each variable has seven values ($a = 7$), making the instances more difficult to solve. An instance with `IDF` $= a$ has *no* embedded interchangeability and provides the most adverse condition for bundling algorithms. We tested the strategies listed in Table 6.1 on each of these two pools, and took the averages (of 20 instances) of the number of nodes visited (`NV`), constraint checks (`CC`), size of the first bundled solution (`FBS`), number of solution bundles (`SB`) (when finding all solutions), and the CPU time.

Constraint tightness has a large effect on the solvability of a random problem. Problems with loose constraints are likely to have many solutions, whereas the values of all measured parameters (`CC`, `NV`, CPU time, and bundle size) quickly die to zero as tightness grows because almost all problems become unsolvable (especially for $t \geq 0.5$). In order to demonstrate and analyze our data, we show in Figures 6.3 and 6.4 the charts for $t = 0.28$, where all the problems had some solutions, with a domain size of $a = 7$. These problems proved to be the more difficult set, and the comparative performance of the search strategies is more easily seen here. The patterns observed on this data set are similar across all values for tightnesses for both problem pools.

6.3.1 Finding the first solution bundle

In our experiments for finding the first solution bundle, we report the charts for `CC`, `FBS` and CPU Time. We omit the chart for `NV` because nearly all search strategies found a first solution bundle without backtracking.

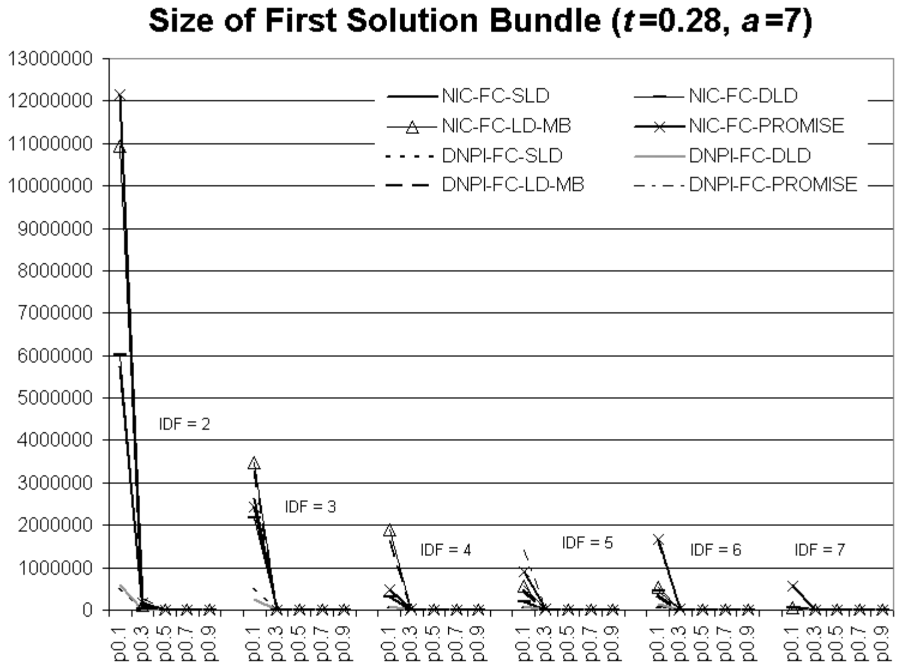
A brief comparison of the search strategies on the evaluation criteria shown reveals that results stated in past chapters are upheld.

The size of the first bundle (`FBS`) We can see that the size of the first bundle found by search strategies is large when $p = 0.1$ and quickly decreases. An examination of the same chart in logarithmic scale (Figure 6.3 bottom), shows that this rapid decrease in bundle sizes found still maintains a separation between search strategies. In general, we see that NIC-FC-`promise` performs very strong bundling, followed closely by DNPI-FC-`promise`.

Constraint checks (`CC`) NIC-FC-`promise` consistently performs the most constraint checks, and FC-`DLD` the least. On average, the search strategies that use DNPI with dynamic variable-value ordering (DNPI-FC-`LD-MB` and DNPI-FC-`promise`), and all search strategies that employ NI_C (NIC-FC-`SLD`, NIC-FC-`DLD`, NIC-FC-`LD-MB` and NIC-FC-`promise`) perform noticeably more constraint checks than the two non-bundling search strategies and than DNPI-FC-`SLD` and DNPI-FC-`DLD`.

CPU Time CPU Time, like constraint checks, separates the search strategies into two groups, with one performing noticeably better than the other. Here, we see the same strategies as before in each group. This leads us to state that the most effective methods when searching for one solution are non-bundling and DNPI bundling without dynamic value ordering.

Further, we can observe trends along the changing levels of interchangeability. Specifically, we see that the size of the first solution bundle decreases as `IDF` increases. Interestingly, even in the absence of interchangeability (large `IDF`) and when density is high (large p), some bundling is still performed (bundle size > 1). We also see that as `IDF` rises, some search strategies are sensitive to the level of interchangeability (NIC-FC-`promise`, for example), and others are not (DNPI-FC-`DLD`, for example). The same can be observed in CPU Time. We see that DNPI-FC search strategies



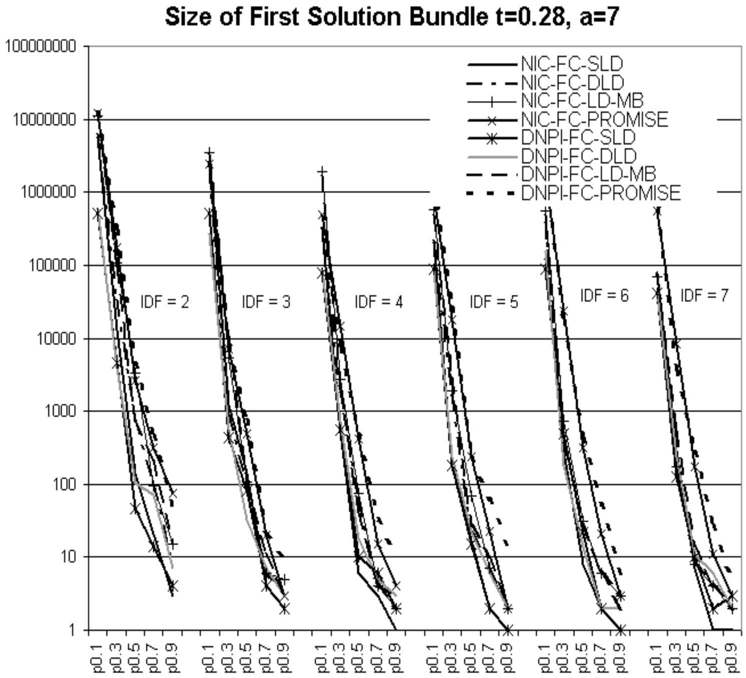
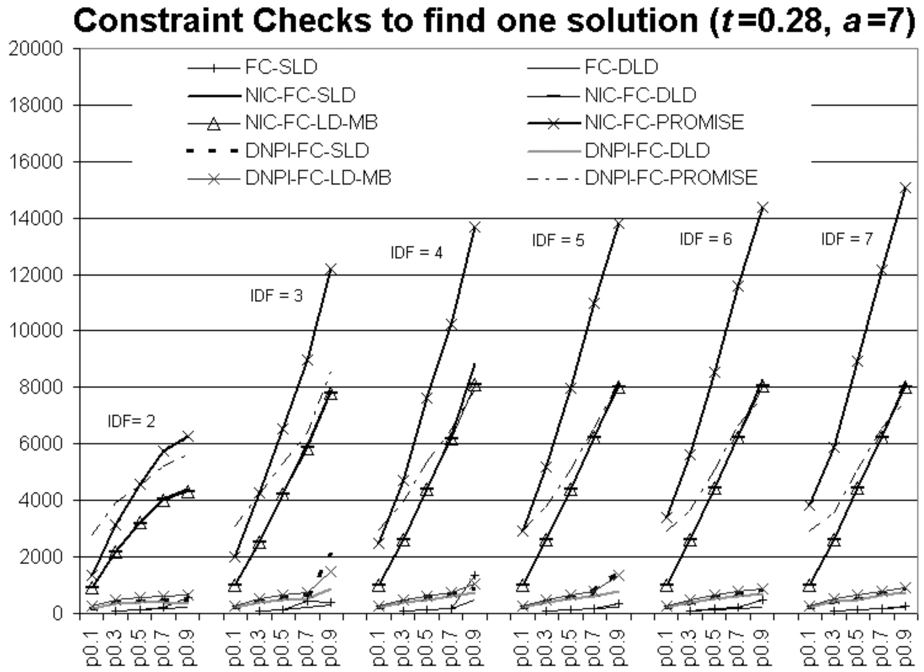


Figure 6.3: *Size of First Solution Bundle (*`FBS`*).*



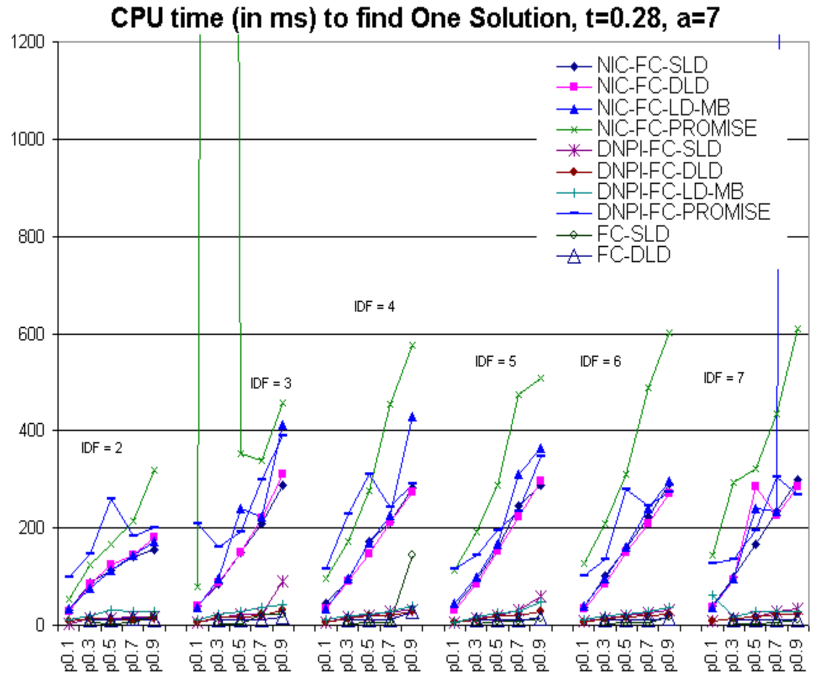


Figure 6.4: *Comparing performance of search for finding one solution.*

without dynamic variable-value ordering and non-bundling search strategies seem resistant to the changing levels of interchangeability.

Because FC is shown slightly below DNPI at the bottom of the chart, it is tempting to think that FC outperforms all the bundling algorithms. However, recall that FC does no bundling at all, so it is finding one solution, while DNPI finds up to one million solutions *that differ only slightly from each other*. Therefore DNPI is finding not only a multitude of solutions, but these solutions are of high quality (due to their nearness)—for a insignificant increase in cost.

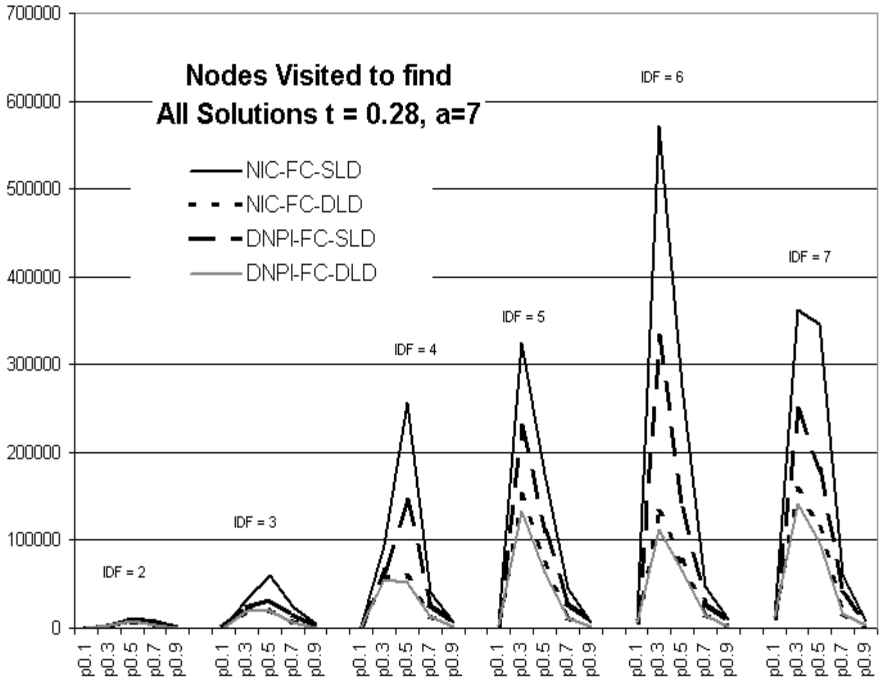
6.3.2 Finding all solutions

The effects of interchangeability in a problem instance are much more striking when finding all solutions, as shown in Figures 6.5 and 6.6.

It is easy to see in all four charts of Figures 6.5 and 6.6 that both static (NI_C) and dynamic (DNPI) bundling search strategies naturally perform better where there is interchangeability (low values of `IDF`) than when there is not (`IDF` approaches a). However, this behavior is less drastic for `DLD`-based search strategies, which are less sensitive to the increase of induced domain fragmentation than `SLD`-based search strategies. Indeed the curves for `DLD` (both NIC and DNPI) rise significantly slower than its `SLD` counterparts as the value of `IDF` increases. Additionally, we see here more clearly than in Chapter 4, that search with `DLD` outperforms search with `SLD` in all cases and for all evaluation criteria.

From this data, one is tempted to think that the problems with the most interchangeability (e.g., $\mathtt{IDF} = 2$) are easier to solve in general than those with higher values of `IDF`. However, notice that the data for non-bundling search strategies is omitted from these charts. For all of the data points shown, FC-`SLD` and FC-`DLD` search strategies could not solve any of the problem instances in less than two hours CPU time[2]. In our tables, notice that the largest CPU time reported is still well under two minutes (100,000 ms = 1.67 minutes). Not only did the performance of FC-`SLD` and FC-`DLD` not vary with interchangeability, they performed so much worse than their bundling counterparts that they caused distortion of the graphs.

[2]One particular instance ran for well over two weeks. With 20 problem instances for each data point, and over two hours CPU time consumed by each problem instance, each data point took a minimum of 40 hours to compute. This made completing the tests prohibitive.



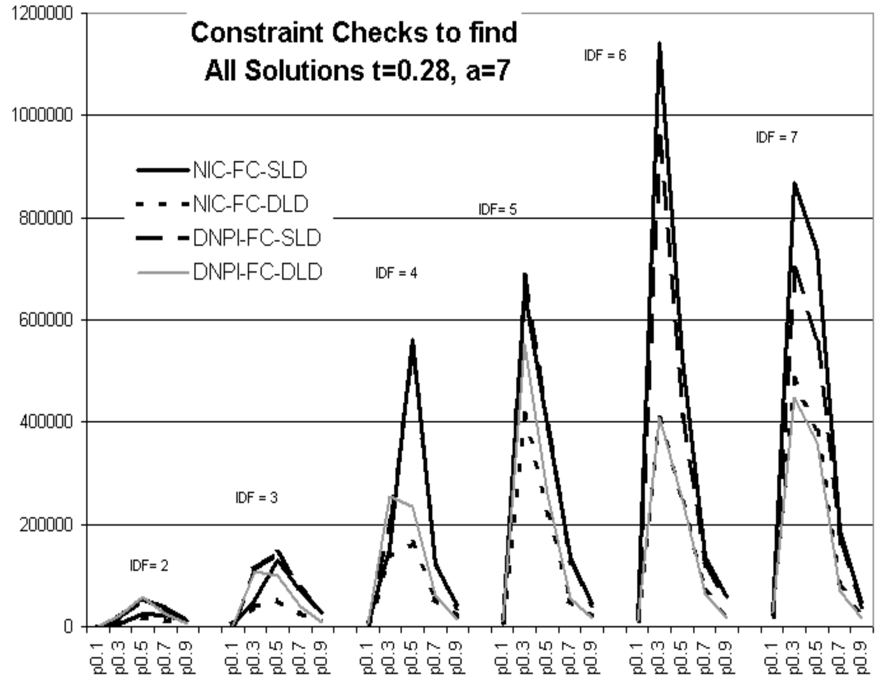


Figure 6.5: *Nodes visited (top) and Constraint Checks (bottom) during search for all solutions.*

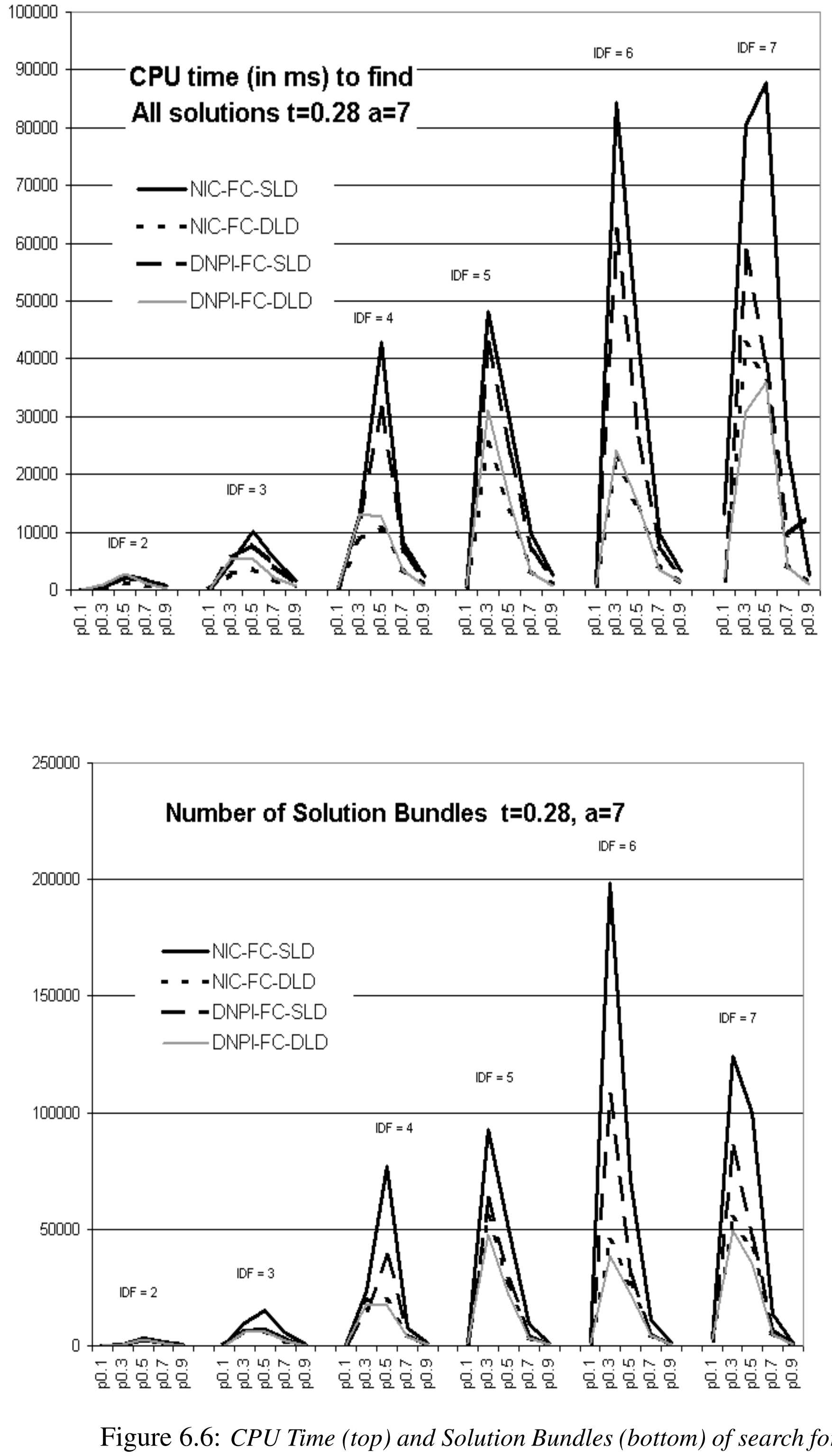


Figure 6.6: *CPU Time (top) and Solution Bundles (bottom) of search for all solutions.*

Even when interchangeability was specifically not included in a problem ($\texttt{IDF} = a$), all bundling strategies, especially dynamic bundling, were able to bundle the solution space. This is due to the fact that as search progresses, some values are eliminated from domains, and thus more interchangeability may become present. This establishes again the superiority of dynamic bundling even in the absence of explicit interchangeability: its runtime is by far faster than FC, and its bundling capabilities are clear.

Summary

We demonstrate that:

1. While the performance of bundling generally decreases with decreasing interchangeability, this effect is muted when finding a first solution.

2. Dynamic ordering strategies are significantly more resistant to this degradation than static ordering and maintain nearly constant effort across the varying levels of interchangeability.

3. Dynamic bundling strategies perform overall significantly better than static bundling strategies when finding one solution, and, in this case, are less sensitive to the level of interchangeability.

4. The combination of dynamic ordering heuristics with dynamic bundling is advantageous. We conclude that this combination, in addition to yielding the best results, is also less sensitive to the level of interchangeability, and thus, is indeed superior to other search strategies.

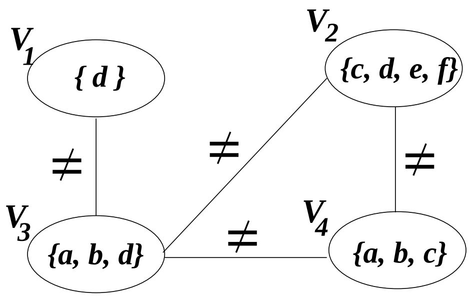
Chapter 7

Full lookahead and dynamic bundling

We have already established that:

- DNPI-FC is useful, both for finding all solutions, grouped into robust sets (Section 3.4) and finding one set of solutions (Section 5.2).
- DNPI-FC is enhanced by dynamic variable ordering, particularly `DLD` (Section 4.3)
- We can see the effect of interchangeability on bundling strategies and that dynamic bundling remains superior to non-bundling (and static bundling) in the midst of situations resistant to bundling (Section 6.3).

Now, we consider another enhancement to dynamic bundling—namely full lookahead. In all of our previous work, we have employed forward checking search (FC), which is a partial lookahead technique. In 1994, Sabin and Freuder [1994] proposed to use a more aggressive lookahead strategy that insures arc consistency among future variables throughout search. It is called Maintaining Arc Consistency (MAC). We illustrate the two (FC and MAC) for the example CSP of Figure 2.1, recalled here:



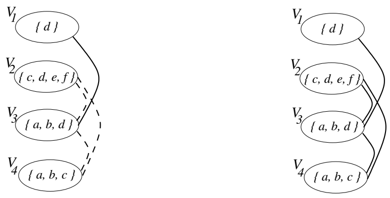


Figure 7.1: *Consistency checking of FC (left) and of MAC (right). FC ignores the constraints with dashed lines.*

The only difference between our version of DNPI-FC and DNPI-MAC is that when a current variable, for example V_1 in the example CSP, is assigned a value the consequences of the assignment are propagated through the entire remaining (future) CSP. FC merely propagates effects of the assigned variable to future variables that are connected to V_1 via constraints. This comparison is shown in Figure 7.1.

We easily see that DNPI-MAC performs a stronger pruning. Sabin and Freuder [1997] claim that this stronger consistency is the most competitive lookahead strategy. Such claims have generated a strong movement *away* from FC and toward MAC in hopes that a new ‘champion’ algorithm, that performs well on all types of problems, had been found.

However, the movement was premature. Gent and Prosser [2000] perform empirical tests, and show that in problems that are dense (high constraint probability), or constraints are loose (tightness is low), FC has an advantage over MAC. They also demonstrate that a dynamic variable ordering such as `DLD` that we employ, causes MAC to lose the advantage over FC. This work was continued by Xu Lin [Xu and Choueiry 2001], by comparing the performance MAC and FC as shown in Table 7.1.

		Tightness	
		Low	High
Probability	Low	FC	MAC
	High	FC	FC

Table 7.1: *MAC only wins with sparse, tight problems.*

7.1 Advantages of full lookahead (MAC)

In the past, MAC has proven advantageous to the search process because of the stronger filtering that it performs. This filtering has the potential to be particularly advantageous when coupled with dynamic bundling. We call this new search strategy DNPI-MAC.

A similar idea has independently been reported by Silaghi [1999], who coupled MAC with the Cross Product Representation (CPR) (recall that this is nearly equivalent to DNPI-FC). A major shortcoming of this study is that they compare two different implementations of MAC, and show that they are relatively equivalent. They test *only* CPR and *only* MAC. They do not address whether MAC actually is beneficial, nor do they compare it to static interchangeability. We seek to quantify the benefit (or hindrance) that MAC adds to search with dynamic bundling.

In this section, we compare dynamic bundling with MAC and with FC under various orderings heuristics. In Chapter 8, we will further compare them (i.e., DNPI-MAC and DNPI-FC) to non-bundling and static bundling strategies with the goal of studying their behavior at the phase transition. We anticipate that the integration of DNPI and MAC will fulfill the expectations discussed below.

Expectation 7.1.1. We expect DNPI-MAC to visit fewer nodes than DNPI-FC. Given that MAC performs a stronger pruning than FC, we can expect that DNPI-MAC will also visit fewer nodes. Any value that is pruned using a MAC search, and not pruned using FC, will be an additional node that FC examines. Further, it is guaranteed to fail and will result in extra useless work for the FC search strategy.

We suspect that Expectation 7.1.1 could stand as a theorem. It is supported by strong empirical evidence in Section 7.3.1 and Figure 7.3 (summarized in Observation 7.2.2), which relate average values over a pool of 6040 random problems. However, a careful examination of the individual results uncovered a single exception that we have not yet been able to justify. This is explained in detail in Section 7.4.

Expectation 7.1.2. We expect DNPI-MAC to generate larger bundles than DNPI-FC. Intuitively, we believe that for any bundle found by DNPI-FC, DNPI-MAC will find the same, or even larger bundle. Just as we prove that DNPI-FC consistently performs better bundling than NIC-FC when using SLD variable ordering with FC in Theorem 3.3.5, we would like to extend this sort of comparison to include DNPI-MAC. Namely, the following expression should hold:

$$\texttt{SB}(\text{FC}) \geq \texttt{SB}(\text{NIC-FC}) \geq \texttt{SB}(\text{DNPI-FC}) \geq \texttt{SB}(\text{DNPI-MAC}) \tag{7.1}$$

where SB is the number of solution bundles found. When finding only the first solution, Equation 7.1 makes a statement about the First Bundle Size (FBS). Recall that when SB is small, bundles are large. Therefore we anticipate the following:

$$\texttt{FBS}(\text{FC}) \leq \texttt{FBS}(\text{NIC-FC}) \leq \texttt{FBS}(\text{DNPI-FC}) \leq \texttt{FBS}(\text{DNPI-MAC}) \tag{7.2}$$

Expectation 7.1.3. Due to these two expectations, we infer that DNPI-MAC should be computationally cheaper than DNPI-FC and perform better bundling.

We now turn to the empirical tests to evaluate these expectations.

7.2 Tests

Although Expression 7.1 should be tested by solving for all solutions, a test run to find all solutions for only one instance in the test pool (which has 6040 problems) took 373 hours of CPU time (275 hours for DNPI-FC and 98 hours for DNPI-MAC). Thus, solving for all solutions is prohibitive on this problem pool. We instead report the results for finding one solution bundle[1].

In order to compare the behavior of DNPI-MAC and DNPI-FC in a dynamic bundling environment, we conducted the tests shown in Table 7.2:

Using these tests, we demonstrate empirically the effects of dynamic bundling on random problems with a constraint probability of $p = 0.5$ and $p = 1.0$ for tightnesses ranging from $t = 0.15$ to $t = 0.85$. This allows us to compare the performance of DNPI-FC and DNPI-MAC (to see if and when DNPI-MAC is useful).

[1] The empirical evaluations in this Chapter were conducted on PCs in the Computer Science laboratory.

Dynamic bundling: MAC or FC?		
Compared strategies	Orderings	Criteria
DNPI-MAC versus DNPI-FC	`SLD`, `DLD`, `LD-MB`	Nodes visited, Figure 7.3 Constraint checks, Figure 7.4 CPU time, Figure 7.5 First bundle size, Figure 7.6

Table 7.2: *Search strategies tested for finding a first solution.*

We used the random generator for binary CSPs described in Section 6.2 [Beckwith *et al.* 2001]. Recall that this generator allows us to control the level of interchangeability embedded in an instance of a CSP by controlling the number of equivalence classes of values induced by every constraint on the domain of one of the variables in its scope. We call this number the degree of induced domain fragmentation `IDF`. Figure 7.2 shows a constraint C with an `IDF`=3.

$$\begin{array}{cc} & V2 \\ & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ V1 \; \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

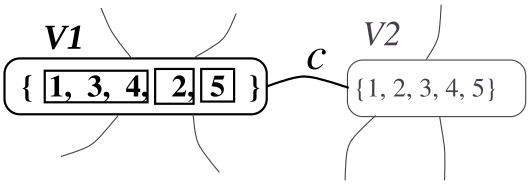


Figure 7.2: *Left:* Constraint representation as a binary matrix. *Right:* Domain of V_1 partitioned by interchangeability.

For each measurement point, we generated 20 instances with $\langle n, a, p, t,$`IDF`$\rangle$ as ⟨20, 10, {0.5, 1.0}, {0.15, 0.20, . . . , 0.85}, {2, 3, . . . , 10 } ⟩ and computed both the average and the median values of nodes visited (`NV`), constraint checks (`CC`), CPU time and size of first bundle found (`FBS`) for each data point. We found that the average and median curves almost always have the same shapes. More importantly, our experiments yield the following general observations that we summarize here before justifying them in detail:

Observation 7.2.1. The curves for `CC` and CPU time are often similar in shape and differ from that for `NV`, suggesting that constraint checks dominate the computational cost in our implementation.

Observation 7.2.2. DNPI-MAC always visits less nodes (`NV`) than DNPI-FC, in confirmation of

Expectation 7.1.1.[2]

Observation 7.2.3. DNPI-MAC in general requires more constraints checks (`CC`) than DNPI-FC. This effect always holds under dynamic orderings (`DLD` and `LD-MB`) where DNPI-MAC performs particularly poorly.

When constraint checks, and not the nodes visited, dominate the cost of computation as it does in our implementation, DNPI-FC will perform better than DNPI-MAC. This shows that, against Expectation 7.1.3, the advantage in fewer nodes visited does not translate into saved time, which yields the following observation:

Observation 7.2.4. Either because CPU time (which, in our implementation, seems to reflect more the effort spent on checking constraints than that spent on visiting nodes), or because the advantages of DNPI-MAC in terms of `NV` does not balance out the loss for constraint checks, DNPI-MAC is more costly than DNPI-FC. This tendency is aggravated under dynamic orderings where the performance of MAC further deteriorates.

Observation 7.2.5. The solution bundle found by DNPI-MAC is in general not significantly larger than that found by FC and does not justify the additional computational cost.

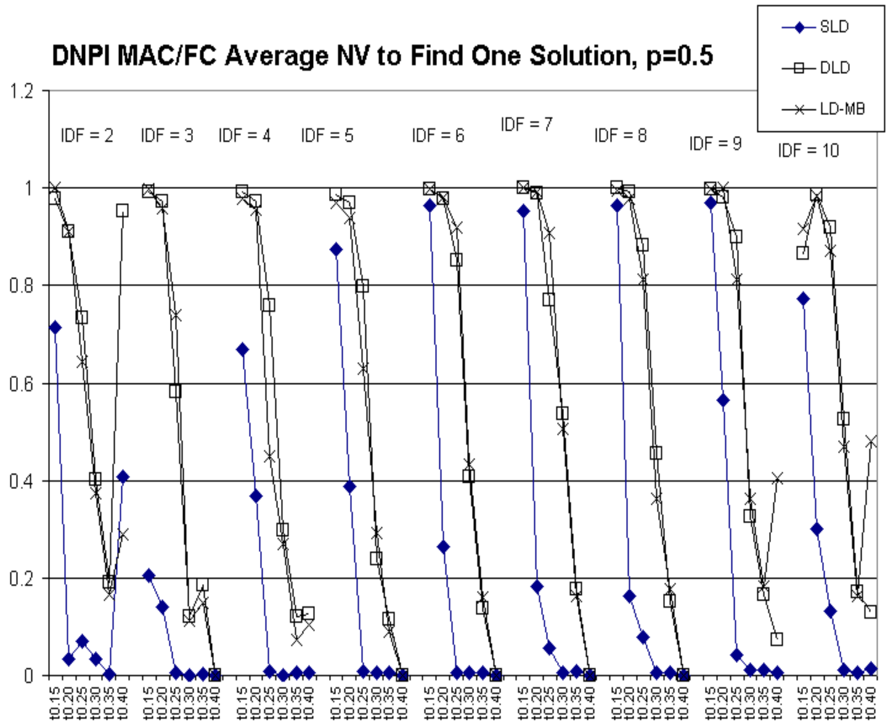
[2]This observation holds for the *average* values reported in our graphs of Figure 7.3, however we detected a single anomaly (in 6040 random problems) that we discuss in detail in Section 7.4.

7.3 Empirical data

In the plots of Figures 7.3, 7.4, 7.5, and 7.6 on the following pages, we report the *ratio* of the values of the evaluation criteria for DNPI-MAC versus FC under dynamic bundling and for the three ordering heuristics `SLD` (⧫) , `DLD` (□), `LD−MB` (×). Note that for values above 1, the value of DNPI-MAC is higher than the value of FC, and for values below 1, the value of DNPI-MAC is lower than the value of FC. Following the charts is a discussion of the data, one page at a time.

7.3.1 Nodes visited (`NV`)

In Figure 7.3, the value of the ratio is consistently below 1 across ordering strategies, `IDF` values, and p values. This indicates that DNPI-MAC *always* visits less nodes than DNPI-FC and supports Observation 7.2.2. Note that this effect becomes more pronounced as the constraint probability increases (shown by the values in the bottom graph being lower than those in the top graph) and as tightness increases (shown by the downward slope of the lines).



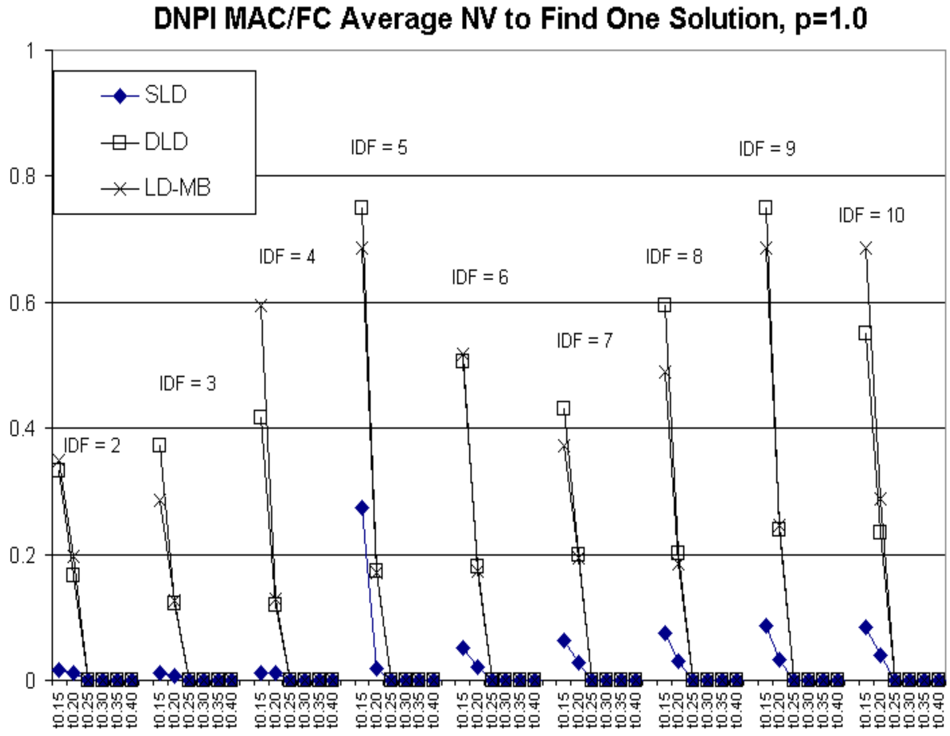


Figure 7.3: *DNPI-MAC versus DNPI-FC:* Nodes visited with constraint probability p = 0.5 (top) and p = 1.0 (bottom).

7.3.2 Constraint checks (CC)

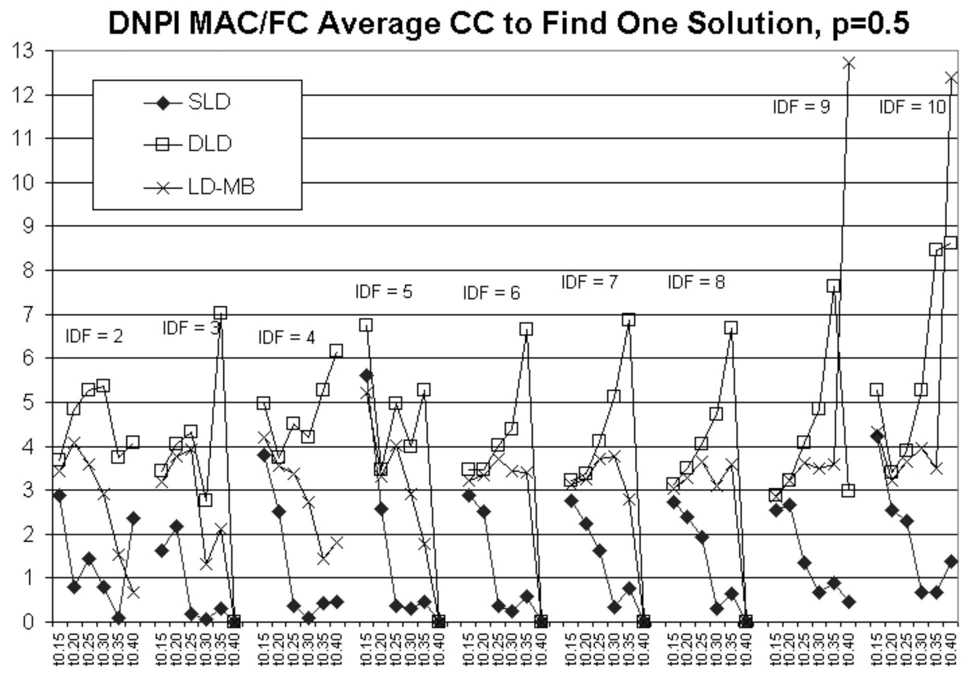
Secondly, in Figure 7.4, we notice that the non-zero values (except for a few that we discuss below) are above 1, which means that DNPI-FC is superior to DNPI-MAC in support of Observation 7.2.3.

There are a few cases that contradict this observation and where DNPI-MAC outperforms DNPI-FC (ratio < 1). Note however that this happens mostly under the static ordering SLD (⧫), once with LD−MB ($\times$) (when IDF = 2 and t = 0.40), and never under DLD (□). In particular, we see that DNPI-MAC performs fewer constraint checks than DNPI-FC when:

1. p=0.5, and t is high, across all values of IDF. We see here a support of the work performed by Xu and Choueiry [2001]. When tightness is high and probability is low, the stronger pruning of DNPI-MAC is advantageous. We also see that DNPI-MAC performs fewer constraint checks than DNPI-FC when

2. p=1.0 and IDF is small. This provides an interesting result, because DNPI-MAC is sensitive to IDF where DNPI-FC is insensitive (specifically, when constraint probability is high). As IDF increases, we see that DNPI-MAC loses the edge it had when more interchangeability was present.

Note, once again, that when using dynamic variable ordering such as DLD (□) and LD−MB ($\times$), DNPI-FC is a clear winner over DNPI-MAC. When we use dynamic variable ordering, DNPI-MAC checks from 3 to 13 times as many constraints as DNPI-FC. This supports Observation 7.2.3 and is a clear indication of an expense in MAC that is not vindicated by dynamic bundling.



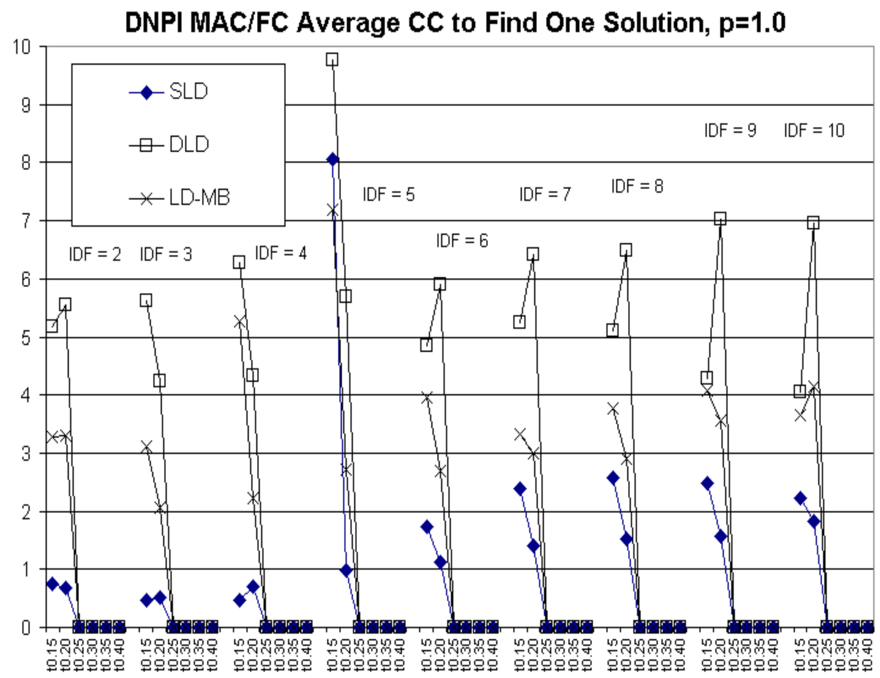


Figure 7.4: *DNPI-MAC versus DNPI-FC:* Constraint checks with constraint probability p=0.5 (top) and p=1.0 (bottom).

7.3.3 CPU time

The advantage of DNPI-FC over DNPI-MAC that was demonstrated by the constraint checks is reinforced by the CPU time data of Figure 7.5. This shows that the use of DNPI-MAC (except for a few cases, mostly in `SLD` ordering) is detrimental to the performance of search despite the savings in terms of nodes visited, in support of Observation 7.2.4.

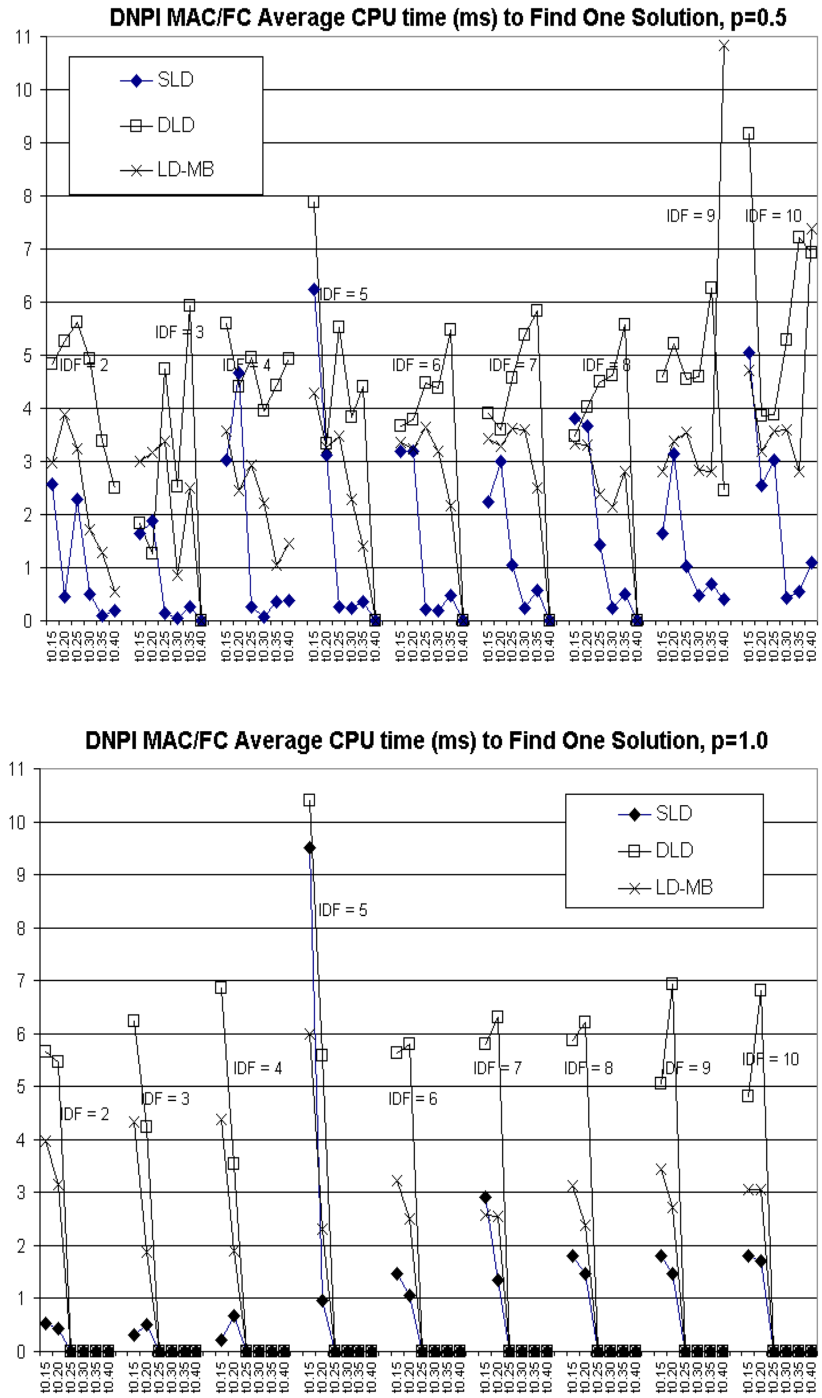


Figure 7.5: *DNPI-MAC versus DNPI-FC:* CPU time with constraint probability p=0.5 (top) and p=1.0 (bottom).

7.3.4 Size of first bundle (`FBS`)

Finally, we look at Figure 7.6 to check whether the additional propagation effort of DNPI-MAC benefits the size of the bundle found and justifies Observation 7.2.5. We see that, and in accordance to Expectation 7.1.2, DNPI-MAC does generate slightly larger bundles than DNPI-FC. For p=0.5, the bundle comparisons huddle mostly just above 1. This means that the bundle sizes are comparable, with DNPI-MAC generally producing bundles that are just a little bit larger. We note two extreme behaviors:

1. When `IDF`=2, t=0.40 with `SLD` ordering (⧫) ordering, the bundle produced by DNPI-MAC is fifteen times larger than that of DNPI-FC. Additionally, this much larger bundle took less time to find. The advantage of using DNPI-MAC is demonstrated and justified at this point. Note however that the extent of this divergence between DNPI-MAC and DNPI-FC does not hold when `IDF`>2.

   Indeed, for p=1.0, the bundles found by DNPI-MAC are never more than three times the size of that found by DNPI-FC, and are frequently smaller (especially when dynamic variable ordering is used).

2. When `IDF`=4, t=0.15 with `SLD` ordering (⧫), the bundle produced by DNPI-MAC is smaller than that of DNPI-FC. This is the only major opposition to our Expectation 7.1.2. (There are several small exceptions, where DNPI-FC produces a bundle that is 1 or 2 solutions larger than DNPI-MAC's bundle. This behavior can be accounted for in non-deterministic bundle ordering.) However, for one problem in this set (the tenth of the twenty problems), DNPI-MAC found a bundle of size 84, with DNPI-FC finding a bundle of size 168, which is in violation of Expectation 7.1.2. Further examination of this particular problem yields even more questions, and is covered in Section 7.4.

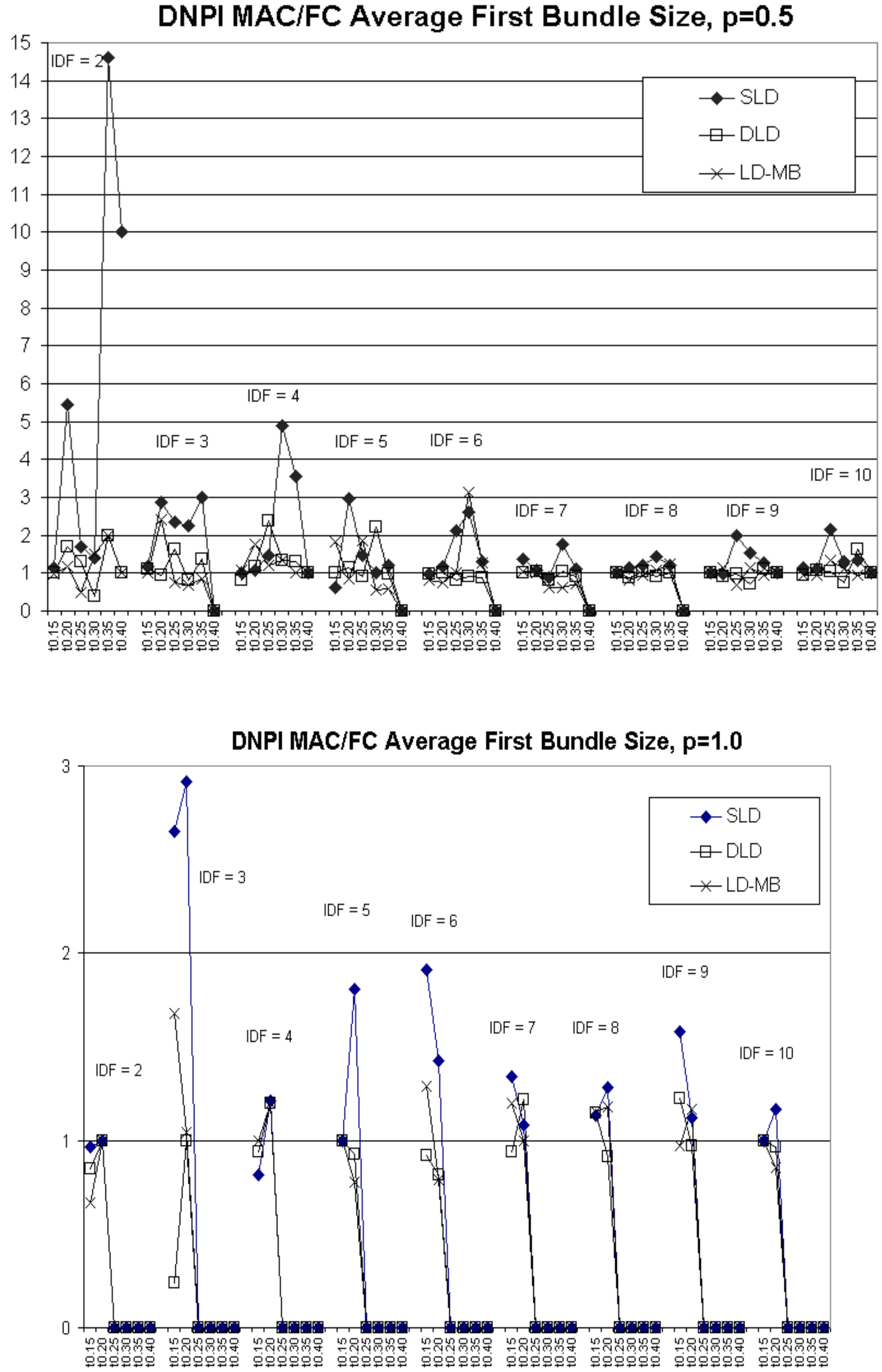


Figure 7.6: *DNPI-MAC versus DNPI-FC:* First bundle size with constraint probability p = 0.5 (top) and p=1.0 (bottom).

7.3.5 Conclusions of the experiments

We conclude that the cost of DNPI-MAC is neither systematically nor predictably worthwhile—even when considering only `SLD` ordering. This tendency becomes even stronger when considering dynamic orderings `DLD` and `LD-MB`.

7.4 An anomaly

As stated above, we expected that DNPI-MAC would *always* perform better bundling when using `SLD` variable ordering. In particular, we expected Expression 7.2 to hold:

$$\texttt{FBS}(\text{FC}) \leq \texttt{FBS}(\text{NIC-FC}) \leq \texttt{FBS}(\text{DNPI-FC}) \leq \texttt{FBS}(\text{DNPI-MAC})$$

In most cases, this does hold, however we note one exception: when `IDF`=4 and t=0.15, see Figure 7.6. Upon closer expectation, this difference is due to *only* one problem in the set of 20 random problems. In taking a closer look at that particular CSP, we can see in Table 7.4, that DNPI-FC and DNPI-MAC produce two very differently sized bundles, with the DNPI-FC bundle significantly larger. In order to find the cause of this unexpected behavior, we begin by closely

Finding one solution	**DNPI-MAC**	**DNPI-FC**
`CC`	9487	1735
`NV`	20	25
`FBS`	**84**	**168**
CPU time [ms]	630	100

Figure 7.7: *First solution statistics for DNPI-MAC vs. DNPI-FC on the tenth instance of random problem of the pool with n=20, a=10, p=0.5, t=0.15,* `IDF`*=5.*

examining the first bundle found by each strategy, and the differences that they took in arriving there. These solutions are shown side by side in table 7.4 to allow easy comparison. Note that the same variable ordering is used, namely `SLD`.

We can see that early in the search, DNPI-FC and DNPI-MAC choose different values for the variable 2: DNPI-MAC chooses bundle (5) and DNPI-FC chooses bundle (10). This difference in assignment propagated, and followed by three other different choices. Because of these different choices a different bundle was found by the two strategies.

Variable	**DNPI-MAC assignment**	**DNPI-FC assignment**
6	(4)	(4)
8	(9)	(9)
14	(6)	(6)
17	(9)	(9)
19	(7)	(7)
3	(8)	(8)
12	(8)	(8)
2	**(5)**	**(10)**
4	(2)	(2)
9	(7)	(7)
1	(7)	(7)
5	(6)	(6)
7	(9)	(9)
10	(5)	(7)
11	(10)	(10)
13	**(10 9 8 7 $\underline{5}$ 2 1)**	**(10 9 8 7 $\underline{6}$ 2 1)**
15	(7 5)	(7 5)
16	**(10 8)**	**(9 6 5)**
18	(1)	(1)
20	**(6 5 1)**	**(8 6 5 1)**

Figure 7.8: *First solution found by DNPI-MAC (left) and DNPI-FC (right).*

Theoretically, it would seem that DNPI-MAC and DNPI-FC could be forced to produce bundles in the same order. Indeed, this would be true if either they followed the same procedure for finding the domain partitions or the bundles within each domain partition were ordered. We must note the the latter is not reasonable. To order partitions would require not only finding a consistent ordering scheme (does bundle (2) come before or after bundle (1, 3) and such), and would require computation to sort these partitions. We choose to avoid enforcing such a behavior.

It is tempting to say that in spite of choosing a larger first bundle, DNPI-MAC would perform better bundling than DNPI-FC when finding all solutions. In this particular problem, however, this is *not* the case. DNPI-FC performs better bundling even here shown by the smaller number of bundles found in Table 7.4.

From this table, we notice that the two search strategies find the same total number of solutions to the CSP, and hence seem to be functioning correctly. Notice also that DNPI-FC found fewer bundles than did DNPI-MAC. Since it is finding fewer bundles, bundles generated must be larger in

Finding all solutions	**DNPI-MAC**	**DNPI-FC**
Number of solutions	1124402637	1124402637
Number of bundles	**22286318**	**21641683**
Maximum bundle size	12800	12800
Average bundle size	50	52
`CC`	389589279	769568323
`NV`	75406160	70420528
CPU time [ms]	35278700	98860770

Table 7.3: *All solution statistics for DNPI-MAC vs. DNPI-FC on the tenth instance of random problem of the pool with* $n = 20$*,* $a = 10$*,* $p = 0.5$*,* $t = 0.15$*,* `IDF`*=5.*

average, indeed, the average solution bundle size is larger. This means that they are *not* merely finding the same bundles in a different order, but that DNPI-FC is actually performing better bundling than DNPI-MAC. Clearly this is an exception to Expectation 7.1.2.

Further, notice that DNPI-MAC only performs about 1/3 the constraint checks of DNPI-FC. In time, like in constraint checks, DNPI-MAC is about 1/3 of DNPI-FC. In this case, it's a very substantial savings. DNPI-FC took 274.61 hours (recall that the units shown are milliseconds, with a clock resolution of 10 ms) and DNPI-MAC only 98 hours, so DNPI-MAC saved over 176 hours. Note also that this run-time is merely CPU time, and wall-clock time was even longer. This demonstrates why it is sometimes prohibitive to run experiments for finding all solutions.

Finally, notice that this problem is also a counterexample to Expectation 7.1.1 and Observation 7.2.2 stating that DNPI-MAC visits fewer nodes than DNPI-FC. This is the only problem of 6040 problems that produces such results, but warrants investigation beyond that performed here. The sheer size of the instance and our inability to replicate this problem on a smaller problem size prevent us from investigating the details of this anomaly.

Summary

In spite of the presence of an anomaly, we establish that, *in general*, the following holds:

1. DNPI-MAC visits less nodes `NV` than DNPI-FC.

2. DNPI-MAC in general requires more constraints checks `CC` than DNPI-FC.

3. The CPU time taken by DNPI-MAC is in general higher than the CPU time taken by DNPI-FC. This is especially true in dynamic variable ordering (i.e., `DLD` and `LD-MB`).

4. The larger bundles found by DNPI-MAC do not balance out the cost in terms of constraints checks and CPU time.

In conclusion, unless we are using `SLD`, DNPI-MAC is not worth the effort and DNPI-FC should be used instead.

Chapter 8

Bundling and phase transition

In 1991, Cheeseman *et al.* [1991] presented empirical evidence of the existence of a phase transition phenomenon in **NP**-complete problems. In particular, they showed that the location of the phase transition and its steepness increase with the size of the problem[1], thus yielding a new characterization of this important class of problems. The idea is based on the fact that, for these problems,

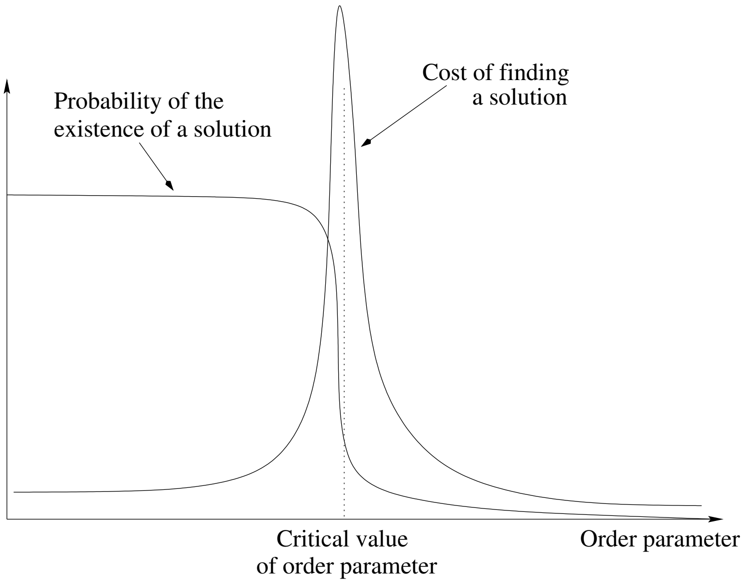


Figure 8.1: *Phase transition phenomenon.*

one can choose one or more order parameters to describe the problem at a macroscopic level, and detect a phase transition around a critical value of this order parameter (Figure 8.1). For instance,

[1]Problems in **P** do not in general have a phase transition, but when they do, the transition is fixed and is not affected by an increase of problem size.

in the case of CSPs, constraint tightness has been found to be one such parameter. On one side of this critical value, problems have a large number of solutions, thus the probability of the existence of a solution is close to 1. And, on the other side of the critical value, problems are unlikely to have solutions at all and the probability of the existence of a solution drops quickly to near 0. Further, both sets of problems are easy to solve (for one finding solution). On the one hand, a solution is easy to find. And on the other, proving that no solution exists is equally easy. Therefore, the cost of search in these areas is low as shown in Figure 8.1. The problems that are challenging to solve appear at the phase transition, around the critical value of the order parameter. In this region, problems are notoriously difficult. This conjecture [Cheeseman *et al.* 1991] has dramatically influenced the research and spurred active investigations in the field as witnessed in [Hogg *et al.* 1996]. In the CSP, the placement of a phase transition depends on constraint tightness (and is somewhat affected by constraint probability) [Prosser 1996].

In previous chapters, we have noticed that randomly generated problems with $t > 0.45$ were found insolvable, and they could be found insolvable after running a mere arc-consistency procedure. This is evidence of the phase transition. In this chapter we consider the effect of bundling (statically and dynamically) on the phase transition. Because these are known to be the most difficult problems to solve in general, determining the performance of our algorithms in this region is important.

So far, we have not found a scenario where dynamic bundling is any hindrance to the performance of search, both for all solutions and for one solution. It is further aided by dynamic variable ordering, but not by MAC as discussed in Observations 7.2.2 and 7.2.3. In these experiments with the phase transition, we are preparing the most adverse situation that we know of for our algorithms. However, given the past behavior, we anticipate the following:

Expectation 8.0.1. Dynamic bundling will significantly reduce the steepness of the phase transition, that is, the problems in the phase transition will be easier to solve with dynamic bundling.

8.1 Experiments

To determine the behavior of bundling algorithms on the phase transition, we conducted the experiments shown in Table 8.1. For these tests, we used the same problem pool described in Section 7.2. Recall that these problems were generated by the random CSP generator described in Section 6.2 [Beckwith *et al.* 2001], and have $\langle n,\, a,\, p,\, t,\, \texttt{IDF}\rangle$ as $\langle 20,\, 10,\, \{0.15,\, 0.20,\, \ldots,\, 0.85\},\, \{0.5,\, 1.0\},\, \{2,\, 3,\, \ldots,\, 10\}\rangle$. Given the large size of these problems (i.e., $n = 20$, $a = 10$) it is impractical to run experiments for finding all solutions. We measured, as usual, the nodes visited (NV), constraint checks (CC), CPU time, and first bundle size (FBS). As before, the average and median curves almost always have the same shapes (except for one case discussed in Section 8.3). Our experiments yield the observations reported below. In particular, Obervations 8.1.2 and 8.1.11 disprove incorrect assumptions held in the community and establish that dynamic bundling is consistently worthwhile and MAC is not always so.

Bundling: Effects on phase transition	
Compared strategies	Orderings
DNPI-FC DNPI-MAC NIC-FC FC	SLD, Section 8.2.2 DLD, Section 8.2.3 LD−MB, Section 8.2.4

Table 8.1: *Search strategies tested for finding a first solution.*

Observation 8.1.1. The magnitude and steepness of the phase transition increases proportionally with p, in accordance with past research [Hogg *et al.* 1996].

Observation 8.1.2. Although dynamic bundling does not completely eliminate the phase transition, it dramatically reduces it.

Observation 8.1.3. DLD orderings are generally less expensive than SLD orderings for all search strategies and yield larger bundles.

Observation 8.1.4. DLD orderings are also generally less expensive than LD−MB for dynamic bundling but similar for static bundling. However, LD−MB orderings produce larger bundles.

Observation 8.1.5. `LD−MB` orderings are generally less expensive than `SLD` orderings for all search strategies and yield larger bundles.

Observation 8.1.6. The bundle sizes found by all bundling strategies are comparable, thus their respective advantages are better compared using other criteria.

Observation 8.1.7. DNPI-MAC is effective in reducing the nodes visited `NV` at the phase transition.

Observation 8.1.8. DNPI-MAC does not significantly reduce the overall cost at the phase transition.

Observation 8.1.9. In static orderings, the reduction of the phase transition due to the use of MAC seems to be more significant than that due to the use of dynamic bundling.[2]

Observation 8.1.10. Static bundling (NI_C) is expensive in general and we identify no argument to justify using it in practice. Further, under dynamic orderings, its high cost extends beyond the critical area of the phase transition to the point of almost concealing the spike.[3]

Observation 8.1.11. As we saw in Section 7.3, dynamic bundling has a better advantage when paired with FC and *not* with MAC in dynamic orderings.[4]

Observation 8.1.12. In dynamic orderings, DNPI-FC is a clear 'champion' among all strategies with regard to cost (i.e., constraint checks and CPU time).

8.2 Data and analysis

These charts are arranged into three sets, according to Table 8.1. The first set is for `SLD` static variable ordering, the second for `DLD` variable ordering (both reviewed in Section 4.1), and the third for `LD−MB` ordering, which is the new ordering we introduced in Section 5.1. Each graph shows four search strategies, namely non-bundling FC ($\triangle$), static-bundling NIC-FC ($\times$), dynamic bundling

[2]We stress that this effect is reversed in dynamic orderings.

[3]The high cost of NI_C in the zone of 'easy' of problems is linked to the overhead of computing interchangeablity as a pre-processing step prior to search while many solutions exist.

[4]Recall that these dynamic orderings otherwise offer the best overall performance as argued in Observation 4.3.4.

with forward checking DNPI-FC (♦), and dynamic bundling with Maintaining Arc Consistency DNPI-MAC (□). Each page contains two graphs, the top one for p =0.5, and the lower one for p =1.0.

In Section 8.2.1 we state some global observations over the experiments as a group and across ordering heuristics. Then, in Sections 8.2.2 through 8.2.4 we examine the data page by page for each of the three ordering heuristics in this order: `SLD`, `DLD`, then `LD-MB`. In each graph, we pay particular attention to the relative behavior of the search strategies at the phase transition, demonstrated here by the presence of a 'spike' in nodes visited, constraint checks and CPU time.

8.2.1 Global observations

The comparision of the top and bottom charts in all 12 figures of this section, from Figure 8.3 to Figure 8.16, confirms past research [Hogg *et al.* 1996] on how the slope and importance of the phase transition augment with the constraint probability, in accordance with Observation 8.1.1. A careful examination of all 24 graphs at the phase transition peak confirms that dynamic bundling is not only practical at the critical boundary of the phase transition but actually succeeds in dramatically reducing its magnitude, in accordance Observation 8.1.2.

Finally, the comparision of graphs for CPU time and bundling power, interpreted as first bundle size across ordering strategies, shows that `DLD` is consistently an excellent ordering unless one is specifically seeking larger bundles at the expense of a slight increase in cost in which case `LD-MB` is justified. This conclusion is supported by Observations 8.1.3, 8.1.4, and 8.1.5.

8.2.2 Static variable ordering (SLD)

Recall from Observation 7.2.4 and Section 7.3 that DNPI-MAC performs best using an SLD ordering heuristic. When using dynamic orderings (DLD and LD−MB), it is particularly non-competitive.

8.2.2.1 Nodes visited (NV) with SLD

A look at Figure 8.3, next page, shows that strategies based on dynamic bundling (□ and ♦) expand in general fewer nodes than strategies based on non-bundling or static bundling in accordance with Observation 8.1.2. When we examine this figure more carefully, we easily notice that the phase transition is indeed present for DNPI-FC (to some extent) and for NIC-FC and FC (to a large extent) but seemingly absent in DNPI-MAC, in support of Observation 8.1.7. Recall that MAC almost always visits fewer nodes than DNPI-FC, which is guaranteed to visit fewer nodes than the others *when finding all solutions* (Lemma 3.3.1 in Section 3.3). We see here that MAC expands by far the fewest nodes in the phase transition. It seems to almost not have a phase transition at all from the graphs in Figure 8.3. A closer inspection of the data shown in Figure 8.2 shows that a phase transition is present, even in DNPI-MAC, but is three to four orders of magnitude smaller than the competing methods. DNPI seems to benefit very much from the pairing with MAC in SLD.

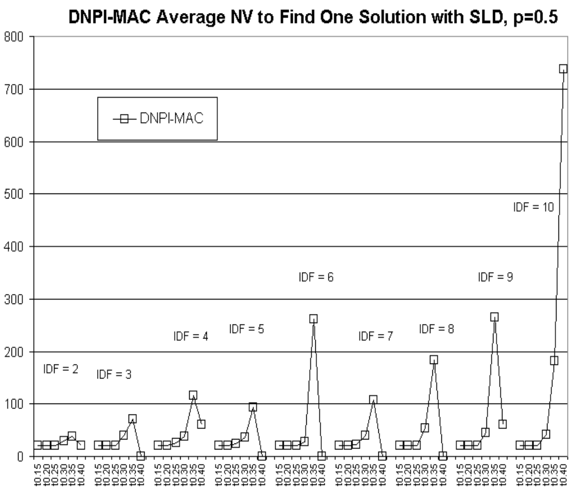
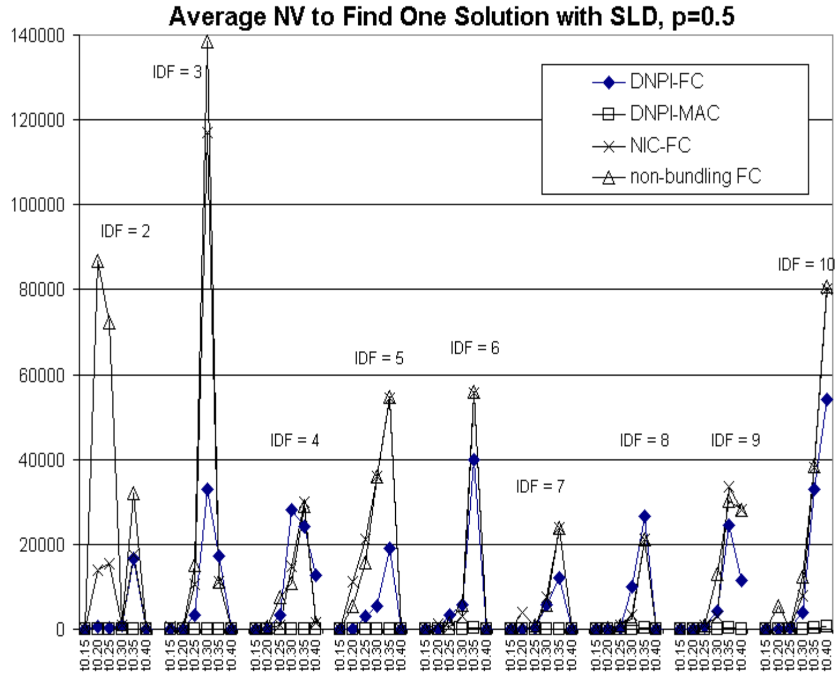


Figure 8.2: *DNPI-MAC nodes visited with* SLD *ordering and constraint probability p =0.5* The phase transition is present.



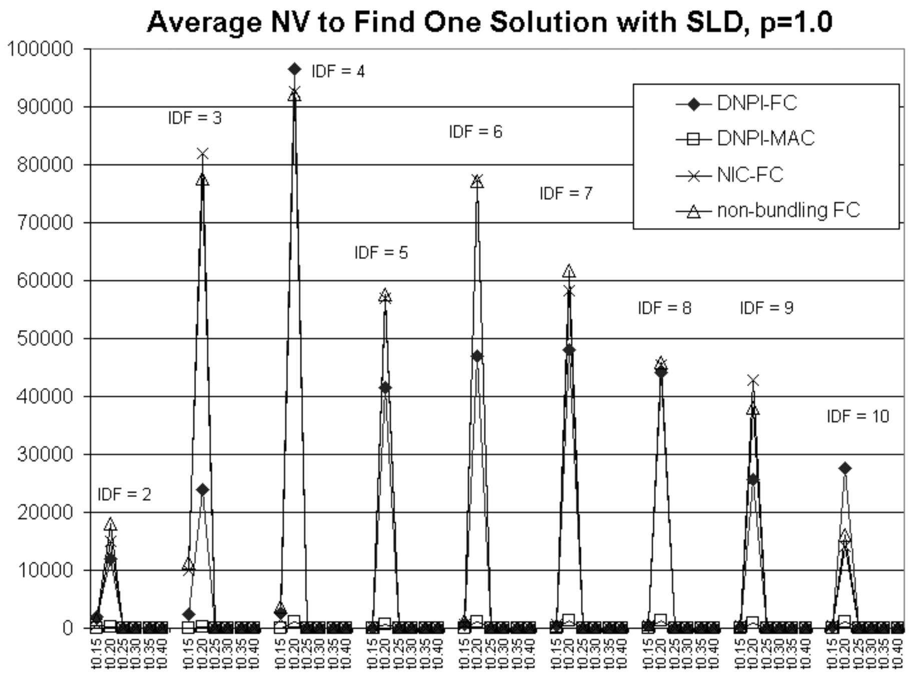
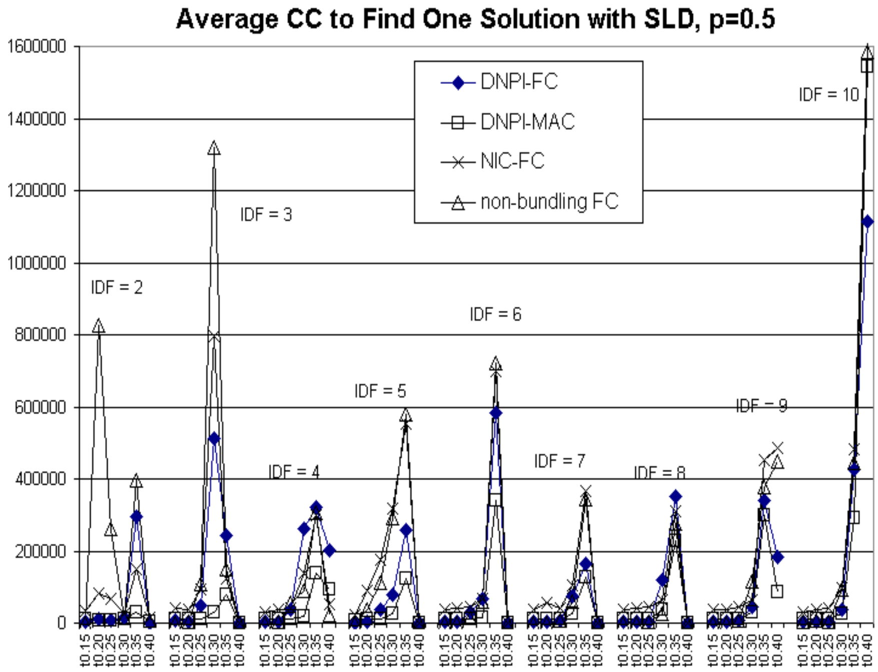


Figure 8.3: *Nodes visited with* `SLD` *ordering and constraint probability p =0.5 (top) and p =1.0 (bottom).*

8.2.2.2 Constraint checks (`CC`) with `SLD`

From the top chart ($p = 0.5$) of Figure 8.4, we notice that, in general, either FC ($\triangle$) or NIC-FC ($\times$) performs the most constraint checks. In particular, the performance of static bundling ($\times$) is quite disappointing, see `IDF` $= 7$, 9 for $p = 0.5$. This supports Observation 8.1.10. Moreover, DNPI-MAC performs the fewest constraint checks at every phase transition except when `IDF` $= 10$. Thus, dynamic bundling remains the winning strategy as stated in Observation 8.1.2. Recall that the 'traditional' fear of dynamic bundling is the excessive number of constraint checks it requires to compute the interchangeability sets. We show here that in the most critical region, it is the non-bundling and static bundling strategies that are actually requiring the most constraint checks. We are now confident to recommend the use of dynamic bundling in search not only to output several interchangeable solutions (which is useful in practice) but moreover to reduce the severity of the phase transition. This result becomes even more significant under dynamic orderings (Sections 8.2.3 and 8.2.4).

In the bottom chart ($p = 1.0$), this tendency is less pronounced and all strategies seem to perform fairly similarly: none of them consistently performing fewer or more constraint checks than any other. Nevertheless, the behavior of dynamic bundling never deteriorates the performance of search enough to make it impractical.



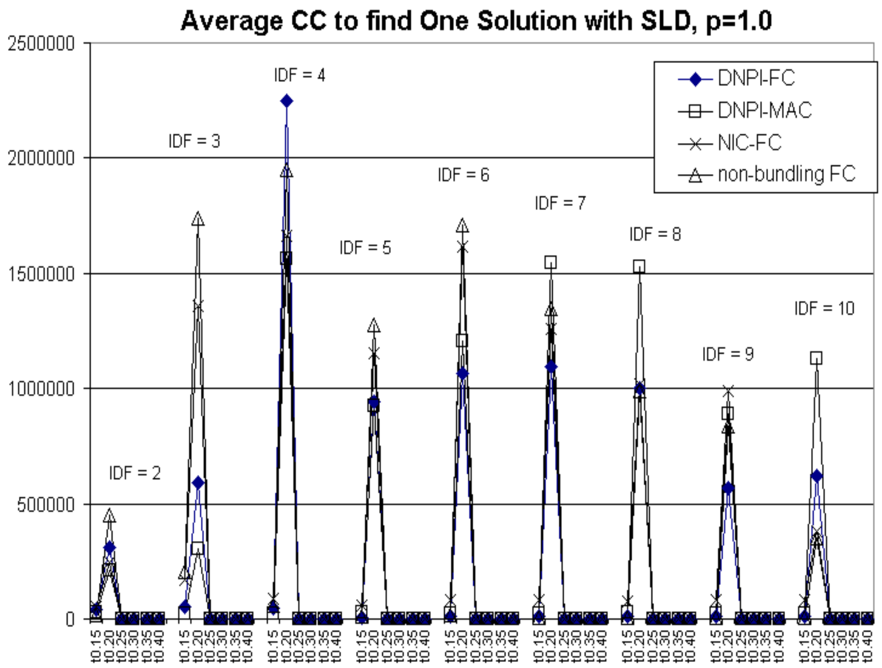
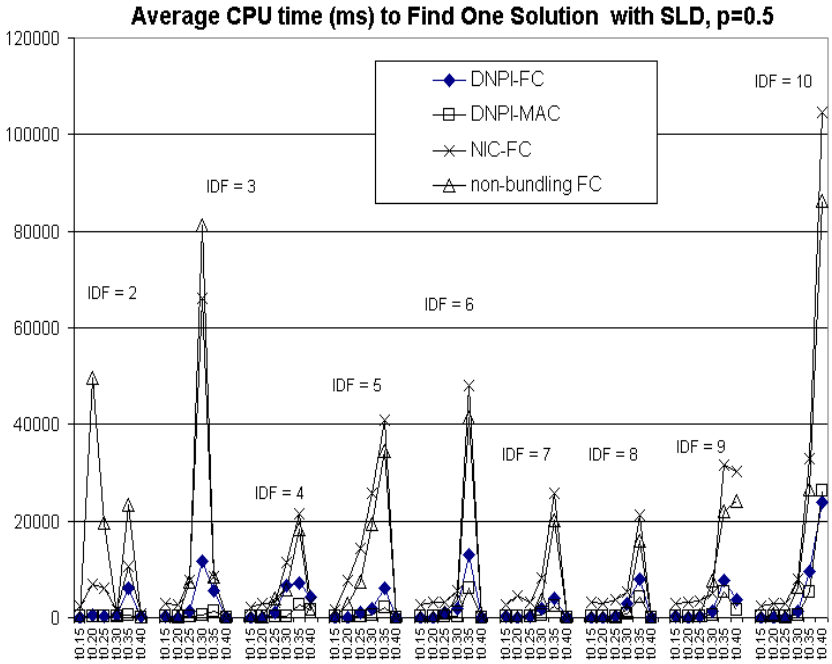


Figure 8.4: *Constraints checked with* SLD *ordering and constraint probability* p *=0.5 (top) and* p *=1.0 (bottom).*

8.2.2.3 CPU time with `SLD`

The graphs for CPU time in Figure 8.5 bear quite a resemblance to that of constraint checks in Figure 8.4 (recall Observations 7.2.1). However, the distance between the two dynamic bundling strategies and the two others is more clearly visible, in favor of dynamic bundling. Indeed, both dynamic bundling strategies DNPI-FC (⧫) and DNPI-MAC (□) are well below the other strategies in both graphs. This is likely thanks to the significant reduction in the number of nodes visited by these strategies, see Figure 7.3 and again supports the use of dynamic bundling to reduce the steepness of the phase transition. Both Observations 8.1.2 and 8.1.9 are supported here.

In the upper chart ($p = 0.5$), DNPI-MAC (□) usually consumes the least CPU time. In the lower chart ($p = 1.0$), DNPI-MAC consumes more time than DNPI-FC for high `IDF` values. This is consistent with the behavior that we observed for constraint checks.



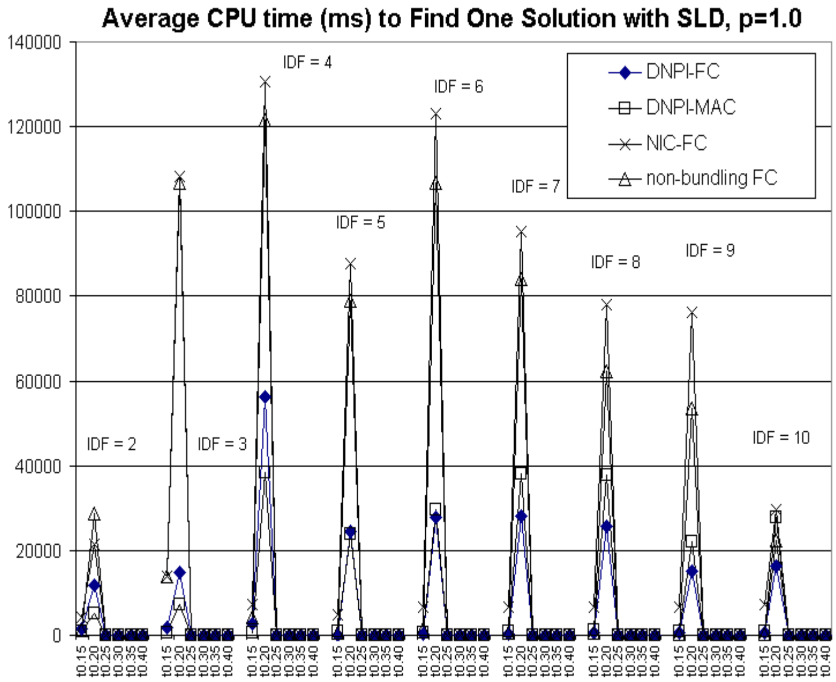


Figure 8.5: *CPU time with* SLD *ordering and constraint probability* p *=0.5 (top) and* p *=1.0 (bottom).*

8.2.2.4 First bundle size (FBS) with SLD

We do not report the First Bundle Size FBS for non-bundling FC, since the solution size is either 1 (when a solution exists) or 0 (when the problem is unsolvable). In general, we find that the sizes of the first bundle found are comparable across strategies (Observation 8.1.6) with a few exceptions addressed below. The graphs are shown in Figure 8.7 across, with a blow up of $p = 5$ in Figure 8.6 below.

For $p = 0.5$ (top graph with blow-up below), we see that NIC-FC surprisingly performs the best bundling for low values of IDF (i.e., 2 and 3). When IDF increases to and beyond 4, dynamic bundling regains its advantage: DNPI-FC and DNPI-MAC compete for the larger bundle in much of the top chart. However, for $p = 1.0$ (bottom graph), DNPI-MAC nearly always performs the best bundling. Notice that these bundles are quite small; less than 5 solutions are contained in each.

One exception worth mentioning here from the bottom chart is when IDF=10 and $t = 0.15$: the data point here is off the chart indicating an exceedingly large first bundle. This is effect is traced to a single instance of the 20 values averaged here and is discussed in more detail in Section 8.3.

8.2.2.5 Summarizing conclusions relative to SLD

Overall, we see that for a static variable ordering (SLD), dynamic bundling, especially when coupled with MAC in DNPI-MAC, drastically reduced the phase transition for a CSP, making the most difficult instances easier to solve.

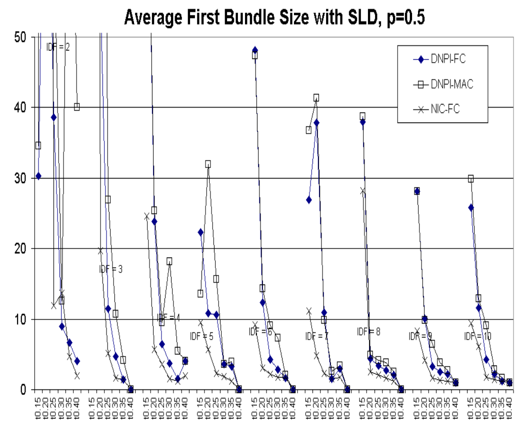
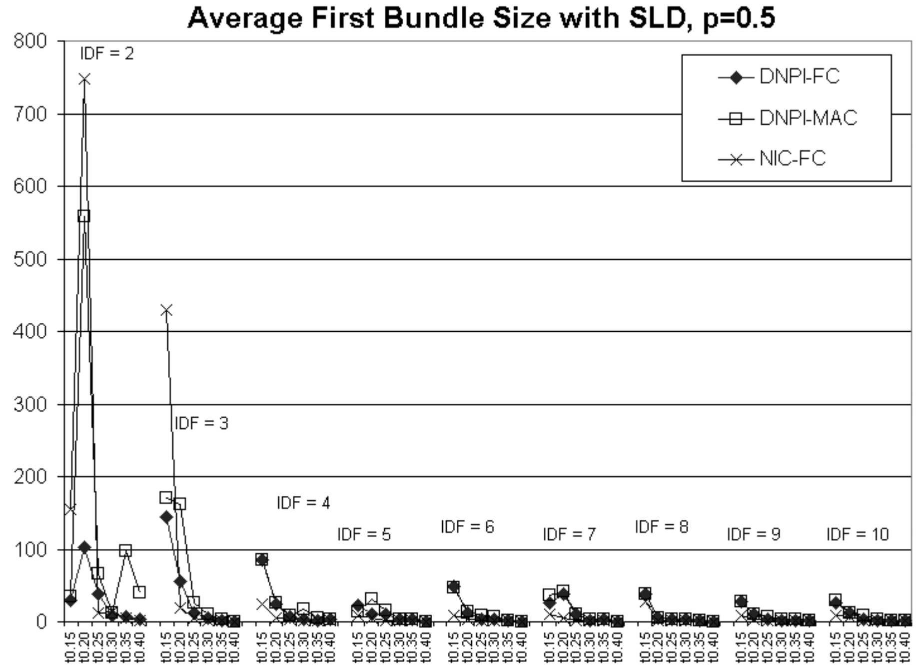


Figure 8.6: *First Bundle Size with* SLD *ordering and constraint probabiliyt* p *=0.5 blow-up*



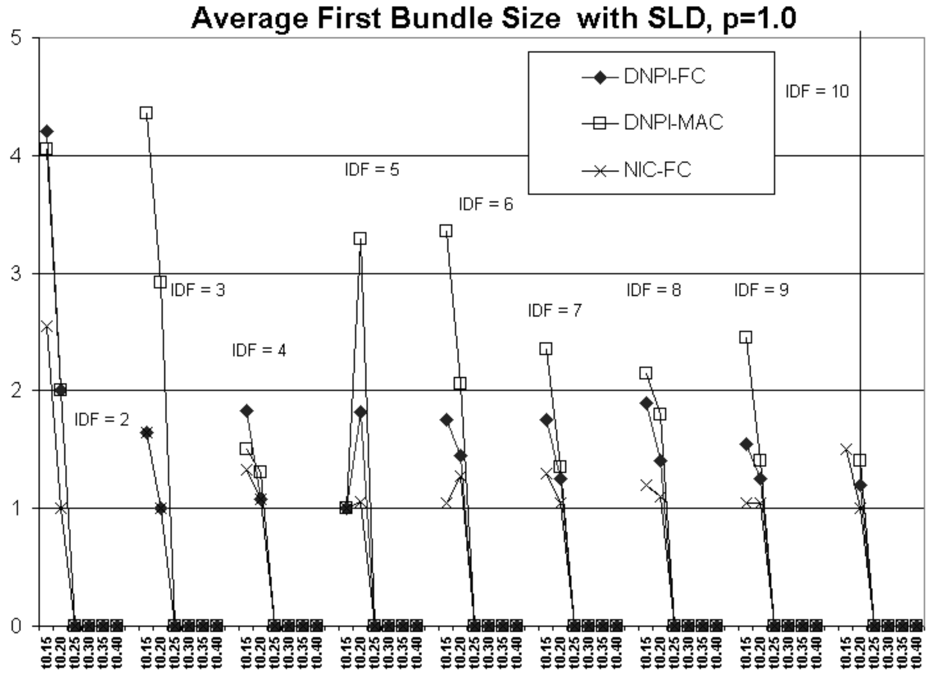


Figure 8.7: *First Bundle Size with* SLD *ordering and constraint probability p =0.5 (top) and p =1.0 (bottom).*

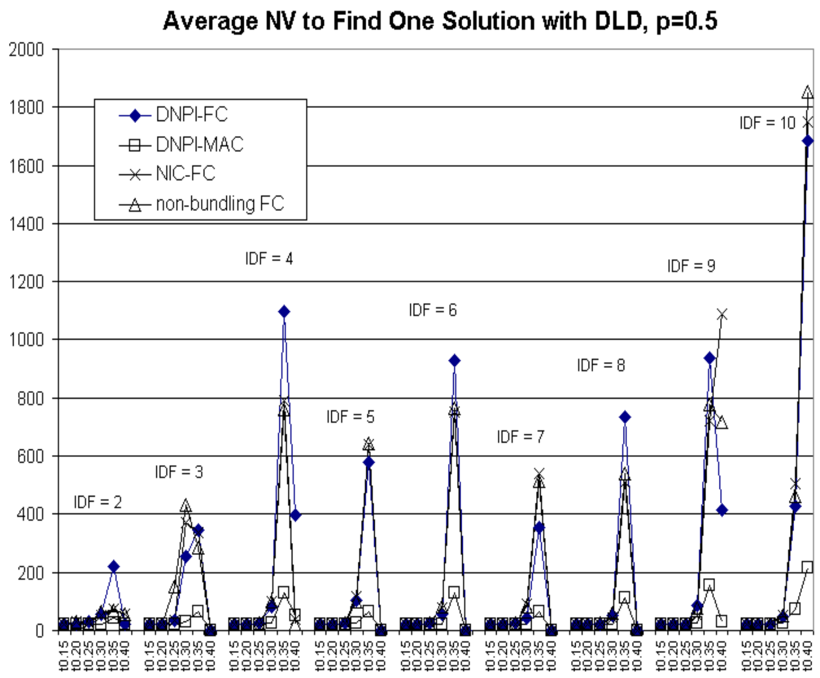
8.2.3 Dynamic variable ordering (DLD)

In general, search strategies with dynamic variable orderings (i.e., DLD and LD−MB) almost always perform better than statically ordered search strategies (see Observation 4.3.4 in Section 4.3 and Observation 5.3.3 in Section 5.3.5). In this section we examine DLD and in Section 8.2.4 we will examine LD−MB. The results are presented in Figures 8.8 through 8.11 showing in turn the charts for the number of nodes visited (NV), the number of constraint checks (CC), the CPU time, and the first bundle size (FBS).

When we compare of the graphs in Figures 8.8 through 8.11 (DLD) with those for Figures 8.3 through 8.7) (SLD), we notice that in general, DLD indeed performs better than SLD. For example, the highest CPU time in DLD ordering is under 8000 ms (Figure 8.10), while it almost reaches 12000 ms in SLD (Figure 8.5). Moreover, we see that dynamic variable ordering heuristics have a stronger effect on some strategies than others. Specifically, it seems to hurt DNPI-MAC while helping the other strategies. These arguments justify Observation 8.1.11.

8.2.3.1 Nodes visited (NV) with DLD

Similar to what we saw in Figure 8.3 for SLD (Section 8.2.2), DNPI-MAC (□) with DLD clearly visits fewer nodes than any other search strategy in Figure 8.8 (Observation 8.1.7). Further, we see that the other three strategies DNPI-FC (⧫), NIC-FC ($\times$), and FC ($\triangle$) compete for visiting the most nodes. DNPI-FC performs the worst most frequently as shown in ($p = 0.5$ at IDF $= 2$, 4, 5, 8, and 9) and ($p = 1.0$ at IDF $= 2$, 4, 8, and 10). Fortunately, poor performance in nodes visited does not affect the other performance of DNPI-FC, which remains a champion as we state in Observation 8.1.12.



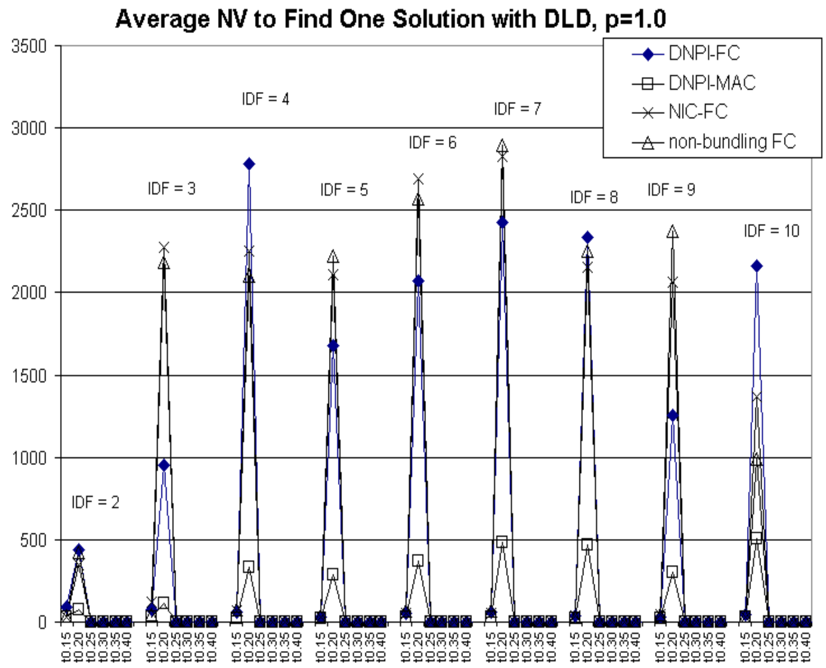


Figure 8.8: *Nodes visited with* DLD *ordering and constraint probability p =0.5 (top) and p =1.0 (bottom).*

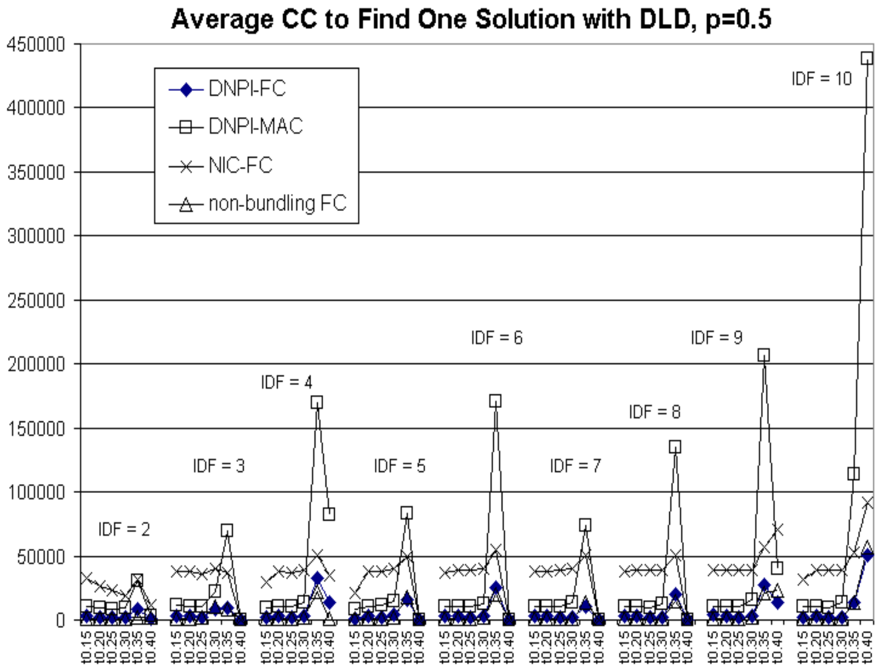
8.2.3.2 Constraint checks (`CC`) with `DLD`

The data here discredits NIC-FC and DNPI-MAC and demonstrates that DNPI-FC is a champion under dynamic ordering (Observation 8.1.12).

DNPI-MAC performs quite poorly with dynamic ordering, beginning with `DLD` in Figure 8.9 and carrying over `LD−MB` in Figure 8.13. We see that in all cases, DNPI-MAC checks the most constraint checks at the phase transition in support of Observation 8.1.11. In almost all cases, DNPI-FC performs the least constraint checks in support of Observation 8.1.2. *Therefore, we can safely conclude that the large amount of checks performed by DNPI-MAC is due to the addition of MAC, rather than to dynamic bundling*.

Further, we see that DNPI-FC (⧫) is quite effective. It clearly and significantly reduces the phase transition (Obervation 8.1.2) and, in general, outperforms the other methods (Obervation 8.1.12). Interestingly, the strongest competitor to DNPI-FC is FC (△) itself. Note, however, that FC provides only one solution while DNPI-FC provides a set of several robust solutions.

Neither NIC-FC (in support of Observation 8.1.10) nor DNPI-MAC (in support of Observation 8.1.11) prove worthwhile here: they *increase* the phase transition rather than decrease it. This justifies our argument in favor of dynamic bundling and confirms our doubts about the appropriateness of MAC in dynamic orderings.



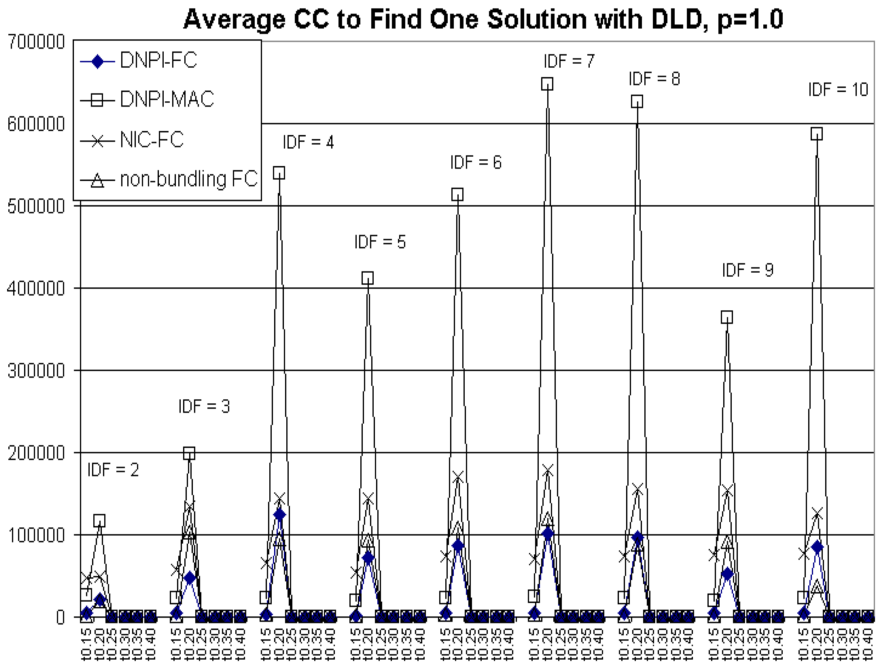


Figure 8.9: *Constraints checked with* DLD *ordering and constraint probability* p *=0.5 (top) and* p *=1.0 (bottom).*

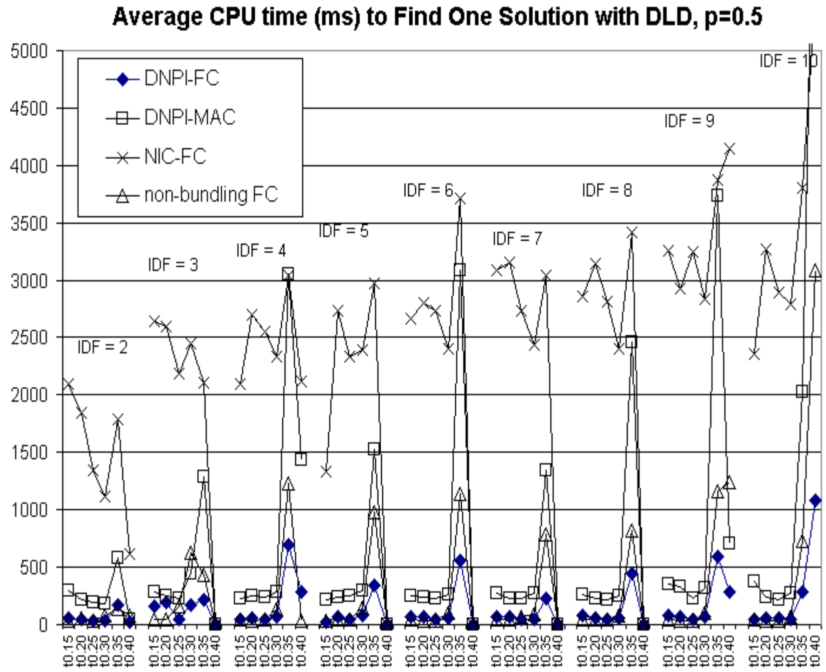
8.2.3.3 CPU time with `DLD`

The charts of Figure 8.10 amplify the effects just discussed in Section 8.2.3.2 (`CC` with `DLD`).

The disadvantage of static bundling becomes apparent in the top graph (p =0.5), in support of Observation 8.1.10. Though the phase transition is less steep (i.e., the spike is almost concealed), the overall cost of performing NIC-FC ($\times$) search is unnecessarily high. This is due to the overhead of finding all NI_C interchangeabilities before beginning search and *constitutes a serious justification against static bundling strategies*.

We can also see that DNPI-MAC ($\square$) continues its trend of performing poorly (Observation 8.1.11). DNPI-MAC is consistently above DNPI-FC (⧫) and FC ($\triangle$) even when not at the phase transition. When p =1.0, it takes more CPU time than even NIC-FC.

Once again, DNPI-FC performs the best overall (Observation 8.1.12). In the bottom graph (p =1.0), we see that DNPI-FC (⧫) reduces the phase transition and performs best at every phase transition (in support of Observations 8.1.2 and 8.1.12).



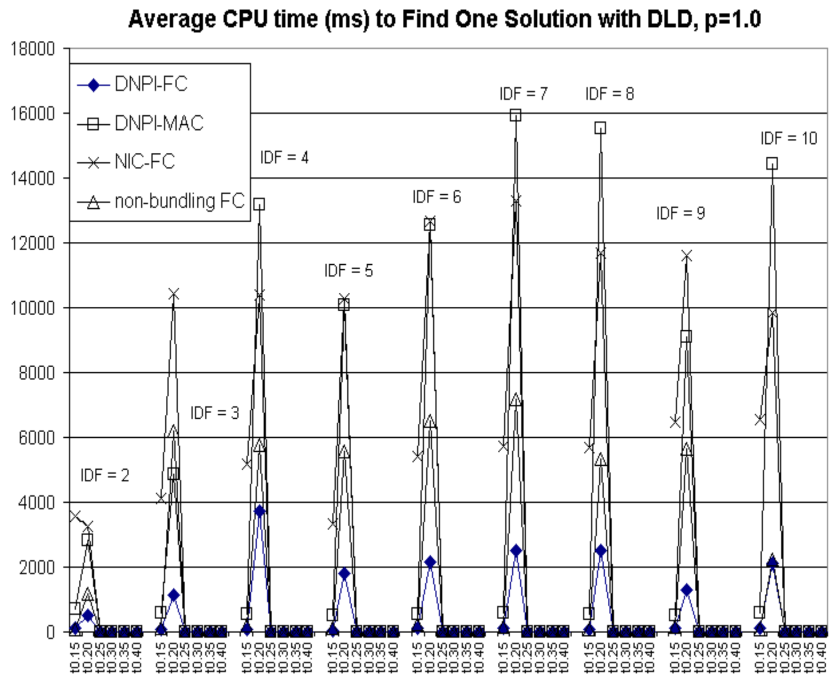


Figure 8.10: *CPU time with* `DLD` *ordering and constraint probability* p *=0.5 (top) and* p *=1.0 (bottom).*

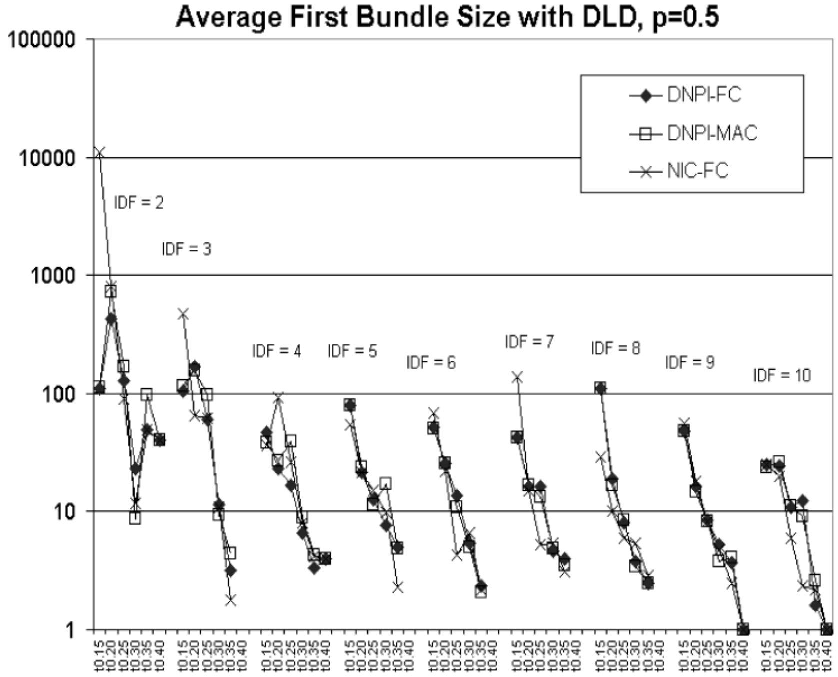
8.2.3.4 First Bundle Size (`FBS`) with `DLD`

Finally, in Figure 8.11, we see that though DNPI-MAC (□) puts forth much effort, it does not even produce the best bundles. In reality, NIC-FC ($\times$) performed unexpectedly good bundling, especially where $p = 0.5$. DNPI-FC (⧫) also bundles very well; it is even with DNPI-MAC in much of the top chart and slightly better for most of the bottom chart, in support of Observation 8.1.6.

8.2.3.5 Summarizing conclusions relative to `DLD`

Using a dynamic variable ordering, we see that DNPI-FC continues to perform better than non-bundling FC. The phase transition is effectively reduced by dynamic bundling.

Further, we also see that the addition of MAC to DNPI is disastrous with a `DLD` ordering: it *increases* the amplitude of the spike of the phase transition in support of Observation 8.1.11. Similarly, NIC-FC behaves worse than FC overall under this ordering (though it finds large bundles).



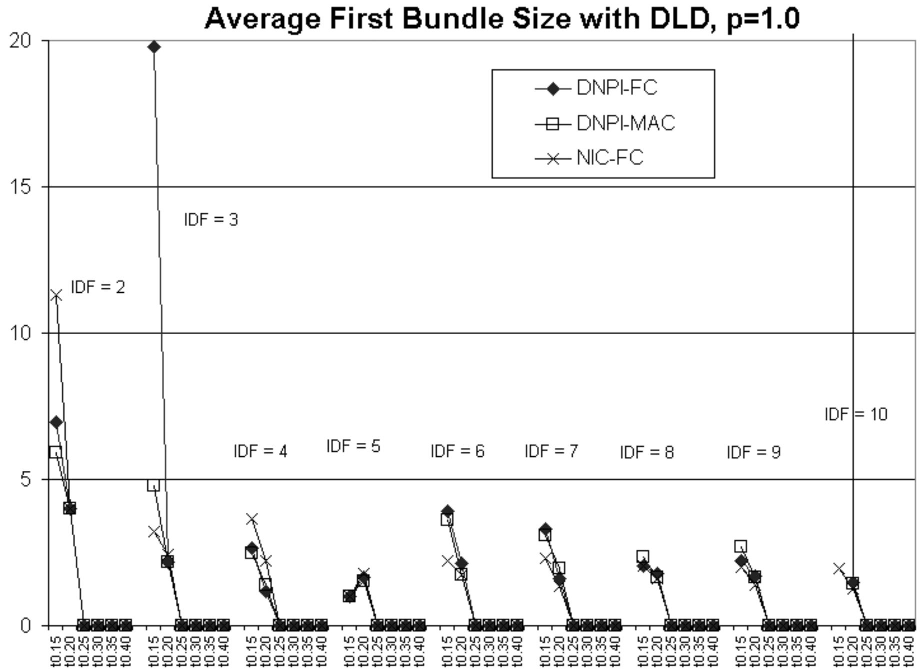


Figure 8.11: *First Bundle Size with* DLD *ordering and constraint probability* p *=0.5 (top) and* p *=1.0 (bottom).*

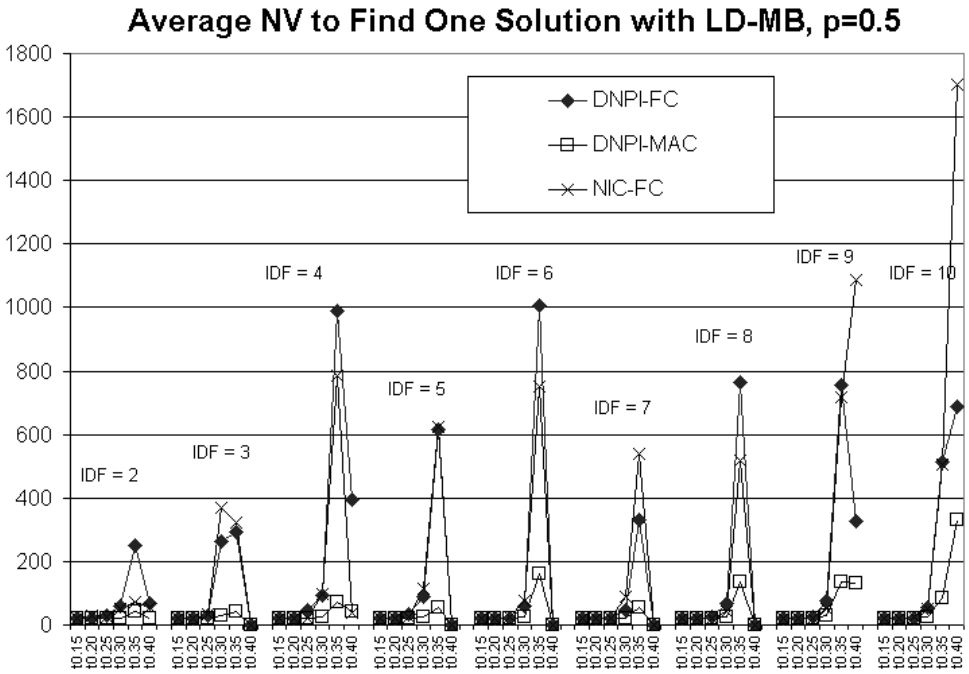
8.2.4 Dynamic variable-value ordering (`LD-MB`)

The remaining results are shown in Figures 8.12 through 8.16. Notice the absence of FC (non-bundling) search on these graphs. Since `LD-MB` (Section 5.1) is a strategy specific to *bundling*, a non-bundling search strategy such as FC makes no sense in this context. Therefore the comparisons drawn here are among the different bundling strategies.

Recall that `LD-MB` is merely `DLD` with a bundle ordering enforced since it assigns to the variable chosen the largest bundle in the partition of its domain. Because of its similarity to `DLD`, it often generally performs as well as `DLD` but produces a larger first bundle. Revisiting this strategy, we see that its effect on the phase transition (when combined with bundling) is also very similar to `DLD`.

8.2.4.1 Nodes visited (`NV`) with `LD-MB`

Like in the other orderings, we see in Figure 8.12 that DNPI-MAC (□) visits fewer nodes than any other strategy for both values of p, in support of Observation 8.1.7. As with `DLD` in Section 8.2.3, we see that DNPI-FC (⧫) often visits the most nodes. This may serve as a notice that, when looking for only one solution, if it is expensive to expand nodes but cheap to check constraints (here it is the opposite), then DNPI-MAC may be an appropriate choice, as highlighted in Observation 7.2.4.



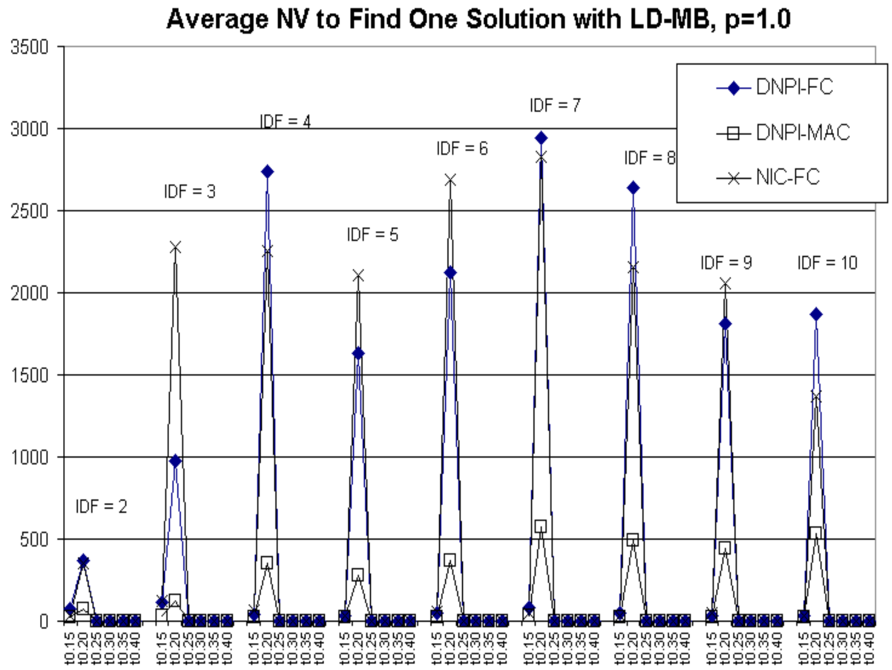


Figure 8.12: *Nodes visited with* LD−MB *ordering and constraint probability* p *=0.5 (top) and* p *=1.0 (bottom).*

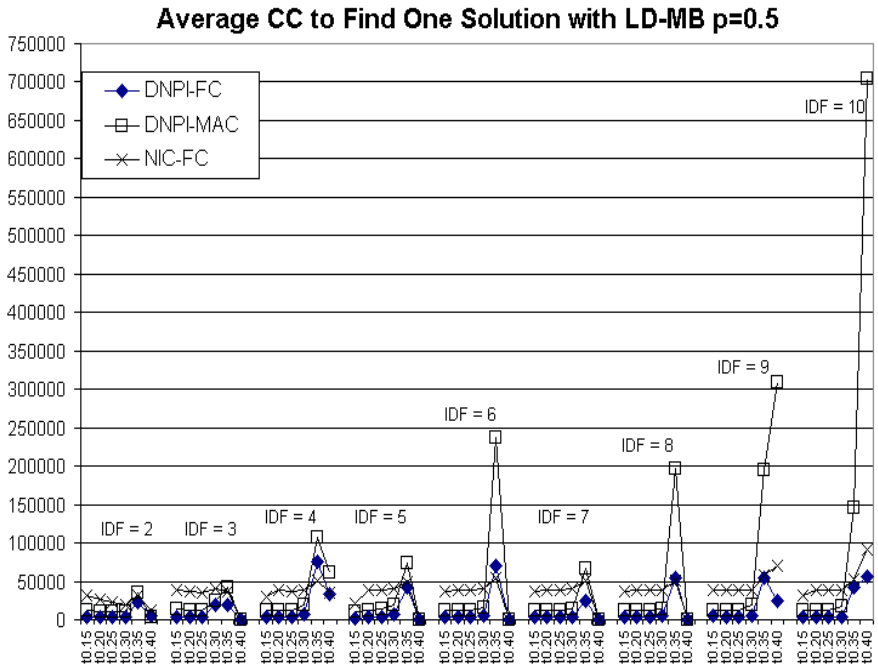
8.2.4.2 Constraint checks (CC) with LD−MB

Regarding the constraints checked by the three bundling strategies, we see in Figure 8.13 that DNPI-MAC (□) checks significantly more constraints than either DNPI-FC (⧫) or NIC-FC ($\times$) at the phase transition, in support of Observation 8.1.11.

However, notice that at most points, especially for a low p (top graph), NIC-FC ($\times$) performs many more constraint checks than the others, almost concealing the phase transition (Observation 8.1.10). This again is due to the overhead of static interchangeability computation. This disadvantage is less visible with a high p, where NIC-FC performs more constraint checks for very loose problems ($t = 0.15$), but reduces the phase transition more effectively than either DNPI-FC or DNPI-MAC.

The comparison of Figures 8.13 and 8.9 shows the following. NIC-FC ($\times$) performs about the same number of constraint checks for LD−MB ordering as for DLD ordering (i.e., between 100,000 and 200,000 when p =1.0), but DNPI-FC (⧫) and DNPI-MAC (□) both perform more constraint checks in LD−MB than in DLD, in support of Observation 8.1.4. This is a disadvantage of the combination of LD−MB with dynamic bundling. LD−MB requires more backtracking because the largest bundle in a variable is often a bundle of no-goods and will trigger backtracking. This will require a re-computation of dynamic interchangeability, and becomes more costly in general. Even when looking for only one solution, it seems that DLD is best suited to dynamic bundling.

Further, the comparison of Figures 8.13 and 8.4 confirms an obvious expectation of LD−MB ordering being less expensive and yielding better bundles than static variable ordering SLD, in support of Observation 8.1.5.



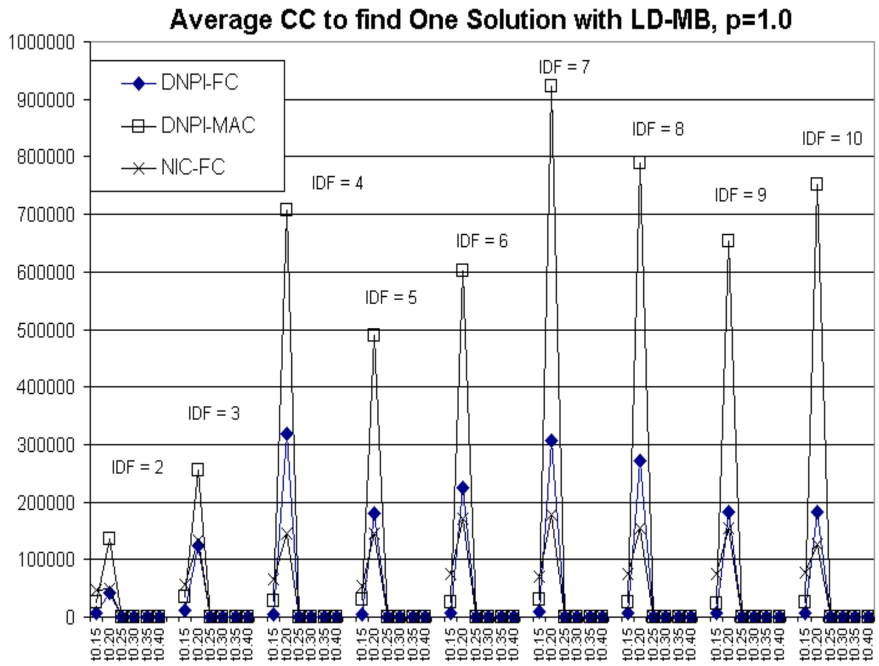


Figure 8.13: *Constraints checked with* `LD-MB` *ordering and constraint probability* p *=0.5 (top) and* p *=1.0 (bottom).*

8.2.4.3 CPU time with `LD-MB`

The trends we noted in the nodes visited and constraints checked for `LD-MB` orderings consistently extend to the CPU time consumed, shown in Figure 8.15 across, with a blow-up for p =0.5 below. We see that both NIC-FC ($\times$) and DNPI-MAC ($\square$) have generally poor performance, requiring more time that DNPI-FC (⧫). This confirms Observations 8.1.12, 8.1.10, and 8.1.11.

Also, the comparison of CPU time with the first two ordering heuristics (i.e., `SLD` in Figure 8.5 and `DLD` in Figure 8.10) confirms Observations 8.1.5 and 8.1.4, respectively.

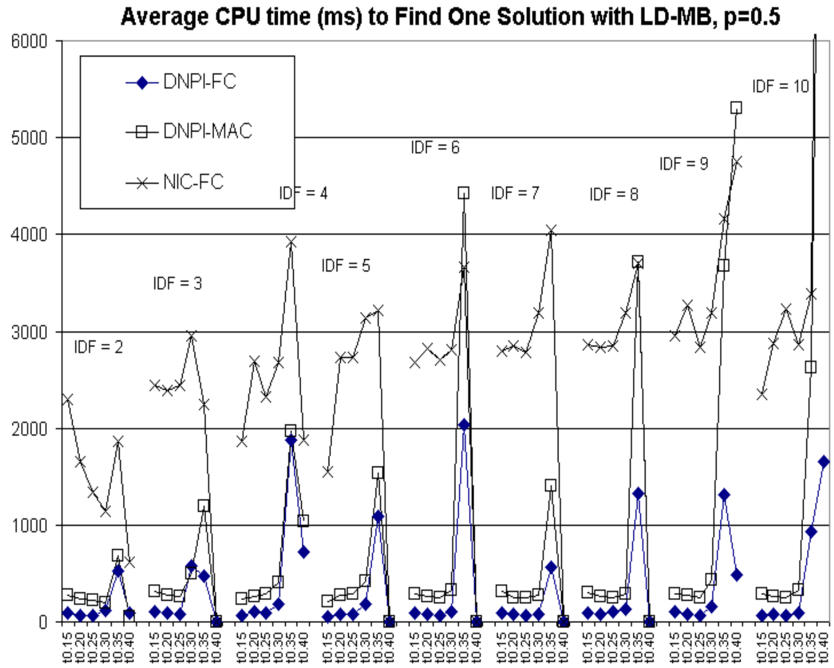


Figure 8.14: *CPU time with* `LD-MB` *ordering and constraint probability* p *=0.5 blow up.*

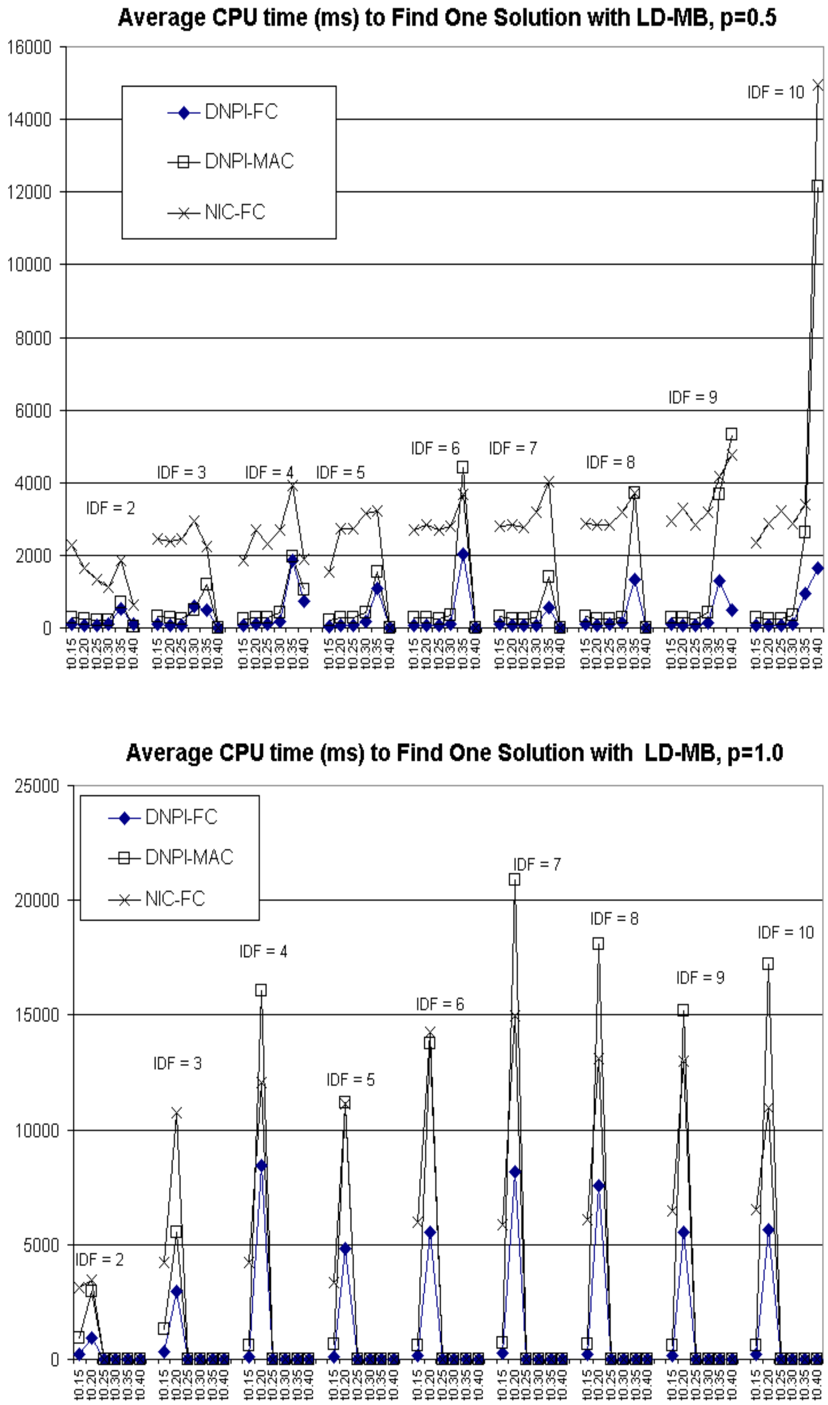


Figure 8.15: *CPU time with* LD-MB *ordering and constraint probability p =0.5 (top) and p =1.0 (bottom).*

8.2.4.4 First Bundle Size (FBS) with LD-MB

Finally, in Figure 8.16 we see that no particular bundling strategy can be declared a *clear* winner based on the size of the first bundle. In general, dynamic bundling is a little stronger than static bundling, with two exceptions (p =0.5, IDF=2 and p =1.0, IDF=4). As far as lookahead strategies are concerned, DNPI-MAC and DNPI-FC are comparable and quite competitive with respect to their bundling capabilities, thus justifying again the power of the dynamic bundling and the superfluity of MAC. This supports Observation 8.1.6.

The comparison across ordering heuristics show that LD-MB yields better bundles than both SLD (Observation 8.1.5) and DLD (Observation 8.1.4)

8.2.4.5 Summarizing conclusions relative to LD-MB

When considering the four criteria shown above, it becomes obvious that dynamic bundling, *without* MAC, that is DNPI-FC, effectively reduces the phase transition and produces large bundles across all variable ordering heuristics. Further, we note that, in the phase transition, DLD seems to be the most effective ordering.

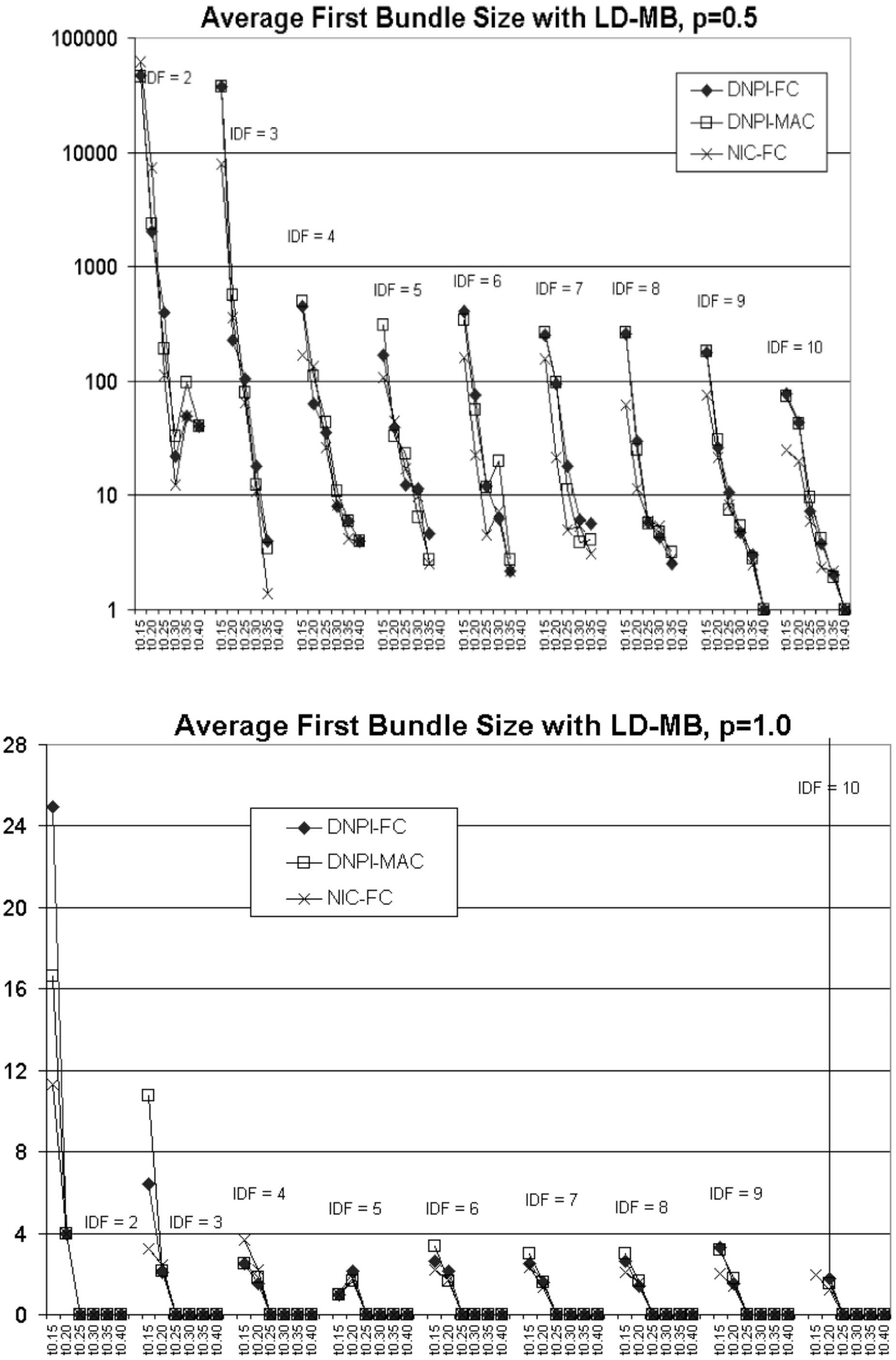


Figure 8.16: *First Bundle Size with* LD−MB *ordering and constraint probability* p *=0.5 (top) and* p *=1.0 (bottom).*

8.3 An anomaly

At the bottom of each of the Figures 8.7, 8.11, and 8.16, which report the average size of the first bundle, we notice a strange spike when IDF is 10. The size of the first bundle is much, much larger than everything else on the graphs. This is unexpected because when IDF=10, there should be no interchangeability. We see that most averages are well below 100, for all orderings, yet this data point is 377915.9 for SLD, 800002.1 for DLD, and 425155.3 for LD-MB. These data points go orders of magnitude beyond the scale in the Figures 8.7, 8.11, and 8.16 (bottom).

Recall that each data point is an average of 20 points. Upon looking at the raw data, we see that the bundle sizes found by DNPI-FC-SLD for this data point are (1 2 2 2 1 7558272 1 1 2 1 2 1 2 1 1 1 2 1 1 1). One problem, the sixth, has an unusually large first bundle size. This particular problem was a randomly generated problem, with $n = 20$, $a = 10$, 190 constraints, $p = 1.0$, $t = 0.15$ and IDF=10. An inspection of the problem reveals that 173 of the 190 constraints are identical. These identical constraints maintain both the correct IDF and tightness. However, because so many are alike, there is a huge amount of structure in this problem, allowing our algorithms to find very large bundles. We do not know what caused these constraints to be identical. We noticed no similar occurences anywhere in our extensive experiments. It is for such rare occurrences that median measurements prove informative, and we choose to report it rather than ignore it or generate a new problem instance to replace this one.

Summary

Dynamic bundling has continued to prove worthwhile by reducing the phase transition for random CSPs. This is especially true for DNPI-FC. DNPI-MAC is a good search strategy to reduce the phase transition if static ordering must be used, but if dynamic ordering is permitted, DNPI-FC is much more effective. The phase transition of these CSPs is best reduced by DNPI-FC with DLD, variable ordering, which has never before been implemented. These strategies, and their relative rank (1st, 2nd, and 3rd), are summarized in Figure 8.17.

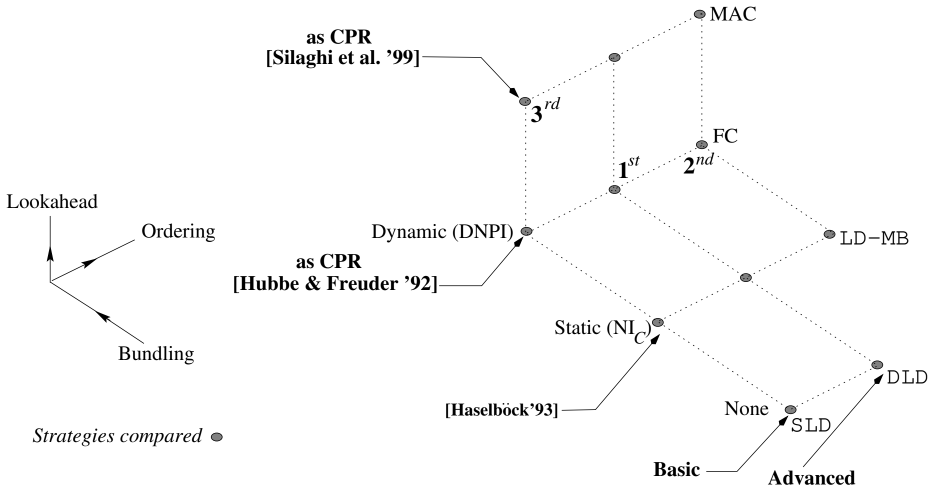


Figure 8.17: *A summary of the strategies tested on binary CSPs and the best ranking ones. All strategies not otherwise marked were proposed by us. Additionally, all methods were implemented by us.*

Chapter 9

Non-Binary CSPs, a proof of concept

Although most of the research in constraint satisfaction is performed on *binary* CSPs, many real-life problems are more easily modeled as *non-binary* CSPs. Because it is always possible to reduce a non-binary CSP into a binary one[1] [Pierce 1933; Freuder 1978; Dechter and Pearl 1989; Rossi *et al.* 1990], the focus on binary constraints has so far been tolerated by the research community. Research on non-binary constraints is still at its infancy and the traditional attitudes on this issue are now being challenged. The techniques developed in this thesis, therefore, face their final test in the realm of non-binary CSPs.

9.1 Example of a non-binary CSP

A non-binary CSP, as shown in Figure 9.1, has a slightly more complicated representation than a binary CSP. In a binary CSP, a constraint is represented as a edge in the constraint graph and links the two variables in its scope. In the non-binary case, a constraint's scope may have more than

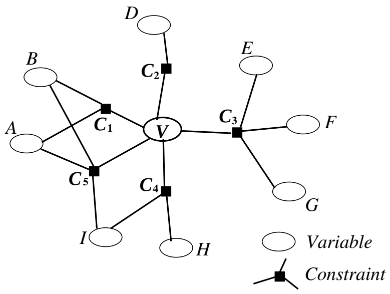


Figure 9.1: *Network of a CSP with non-binary constraints.*

[1]This reduction is not in general polynomial.

C_1		
A	*B*	*V*
1	1	1
1	3	1
2	1	1
2	3	1
4	1	1
5	5	1
6	2	1
1	1	2
1	3	2
2	1	2
2	3	2
4	1	2
5	5	2
6	2	2
1	1	3
1	2	3
2	1	3
3	1	3
4	1	3
4	3	3
5	5	3
6	2	3
1	1	4
1	2	4
2	1	4
3	1	4
1	3	5
2	1	5
2	3	5
3	3	5
4	1	5
4	3	5
5	5	5
6	2	5
1	3	6
2	1	6
2	3	6
3	3	6
4	1	6
4	3	6
5	5	6
6	2	6

C_2	
D	*V*
1	1
2	1
1	2
2	2
1	3
2	3
3	3
6	3
4	4
5	4
6	4
4	5
5	5
6	5
4	6
5	6
6	6

C_3			
E	*F*	*G*	*V*
1	2	1	1
1	2	2	1
1	2	3	1
1	3	1	1
1	2	1	2
1	2	3	2
1	3	1	2
1	3	3	2
2	2	1	2
2	2	3	2
3	2	1	2
1	2	1	3
1	2	2	3
1	2	3	3
1	3	1	3
1	3	3	3
2	2	1	3
2	2	3	3
3	2	1	3
1	2	1	4
1	2	2	4
1	2	3	4
1	3	1	4
1	2	1	5
1	2	3	5
1	3	1	5
1	3	3	5
2	2	1	5
2	2	3	5
3	2	1	5
1	2	1	6
1	2	3	6
1	3	1	6
1	3	3	6
2	2	1	6
2	2	3	6
3	2	1	6

C_4		
H	*I*	*V*
1	1	1
1	2	1
2	1	1
3	1	1
1	1	2
1	2	2
2	1	2
3	1	2
4	1	2
4	3	2
5	5	2
6	2	2
1	1	3
1	3	3
2	1	3
2	3	3
4	1	3
5	5	3
6	2	3
1	1	4
1	2	4
2	1	4
3	1	4
1	3	5
2	1	5
2	3	5
3	3	5
4	1	5
4	3	5
5	5	5
6	2	5
1	3	6
2	1	6
2	3	6
3	3	6
4	1	6
4	3	6
5	5	6
6	2	6

C_5			
A	*B*	*I*	*V*
1	2	1	1
1	2	2	1
1	2	3	1
1	3	1	1
1	2	1	3
1	2	3	3
1	3	1	3
1	3	3	3
2	2	1	3
1	2	3	3
3	2	1	3
1	2	1	4
1	2	2	4
1	2	3	4
1	3	1	4
1	3	3	4
2	2	1	4
2	2	3	4
3	2	1	4
1	2	1	2
1	2	2	2
1	2	3	2
1	3	1	2
1	2	1	5
1	2	3	5
1	3	1	5
1	3	3	5
2	2	1	5
2	2	3	5
3	2	1	5
1	2	1	6
1	2	3	6
1	3	1	6
1	3	3	6
2	2	1	6
2	2	3	6
3	2	1	6

Table 9.1: *Constraints of the example of Figure 9.1 shown as tables.*

two variables and it must be permitted to connect to an arbitrary number of variables. Non-binary constraints are commonly represented as a new kind of node in the constraint network. Figure 9.1 shows a portion of a non-binary CSP, the neighborhood of one variable, V. We will use this CSP as an example in this chapter. Thus, we carefully specify it. Each variable has a domain of $\{1, 2, 3, 4, 5, 6\}$ and the constraints are shown in Table 9.1.

9.2 Constraint probability in non-binary CSPs

Notice that the constraints are permitted to overlap. Here, the scope of C_1 is a strict subset of that of C_5. A non-binary CSP can have an arbitrary number of such overlapping constraints, so the concept of constraint probability becomes more difficult to define. We will use two types of parameters to specify the constraint probability. We use p to indicate the overall constraint probability, that is the total number of constraints over the total number of possible constraints. We use p_x to indicate the constraint probability of all constraints of arity x. The theoretical maximum arity of any constraint is n, that is, a constraint cannot constrain more variables than the CSP has. We know that a problem can have at most $\frac{n\times(n-1)}{2}$ binary constraints (i.e., a complete graph). We define here p_2, p_3, and p_4, as probabilities of binary, ternary, and 4-ary constraints, respectively. We define C as the number of all constraints in the CSP (regardless of their arities), and c_2, c_3 and c_4 as the number of binary, ternary and 4-ary constraints, respectively. These could obviously be extended to p_n and c_n. The relations between the constraint probability p, the total number of constraints in the CSP C, and the various p_x and c_x are then as follows:

$$
\begin{aligned}
C &= c_2 + c_3 + c_4 && (9.1)\\
C &= \binom{n}{2} \cdot p_2 \cdot p + \binom{n}{3} \cdot p_3 \cdot p + \binom{n}{4} \cdot p_4 \cdot p && (9.2)\\
&= \frac{n!}{(n-2)! \times 2!} \cdot p_2 \cdot p + \frac{n!}{(n-3)! \times 3!} \cdot p_3 \cdot p + \frac{n!}{(n-4)! \times 4!} \cdot p_4 \cdot p\\
&= \frac{n(n-1)}{2} \cdot p_2 \cdot p + \frac{n(n-1)(n-2)}{6} \cdot p_3 \cdot p + \frac{n(n-1)(n-2)(n-3)}{24} \cdot p_4 \cdot p\\
c_2 &= p_2 \cdot C && (9.3)\\
c_3 &= p_3 \cdot C && (9.4)\\
c_4 &= p_4 \cdot C && (9.5)
\end{aligned}
$$

For the example shown in Figure 9.1, we clearly see that $C = 5$, $c_2 = 1$, $c_3 = 2$, and $c_4 = 2$. Further,

$$p_2 = \frac{c_2}{\frac{n(n-1)}{2}} = \frac{1}{36} = .0278 \tag{9.6}$$

$$p_3 = \frac{c_3}{\frac{n(n-1)(n-2)}{6}} = \frac{2}{84} = .0238 \tag{9.7}$$

$$p_4 = \frac{c_4}{\frac{n(n-1)(n-2)(n-3)}{24}} = \frac{2}{126} = .0159 \tag{9.8}$$

9.3 Solving a non-binary CSP

Recall that FC for a binary CSP works by assigning a current variable V_c, then pruning the domains of future variables V_f connected to that current variable. FC thus makes each variable individually consistent with the assigned value. When we extend these concepts to non-binary constraints, we must decide how to deal with partially instantiated constraints, with more than one future variable. We choose here to perform consistency checking on every constraint that involves both the current variable, and at least one future variable. This is equivalent to the strategy for forward checking with non-binary constraints nFC2 [Bessière *et al.* 1999].

In order to check a non-binary constraint, we must check every possible combination of the values for future variables. Consider the constraint C_5 from the example in Figure 9.1, which involves variables A, B, I, and V. Suppose that A has been instantiated to 2, leaving the domains of B, I and V as $\{2\}$, $\{1, 3\}$ and $\{3, 4, 5, 6\}$, respectively, as shown in Figure 9.2. When the search

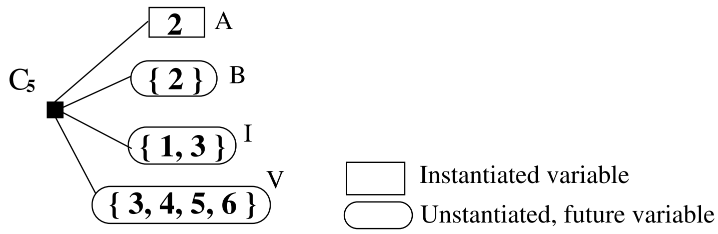


Figure 9.2: *Variable instantiation order during search.*

procedure instantiates B, each possible assignment of B (which is only the value 2 here), must be checked against *both* I and V. Effectively, we check each of the tuples shown in Table 9.2.

A	B	I	V
2	2	1	3
2	2	1	4
2	2	1	5
2	2	1	6
2	2	3	4
2	2	3	5
2	2	3	6

Table 9.2: *Tuples to check in the non-binary constraint C_5.*

When checking the non-binary constraint C_5, we first select, from the constraint definition, the tuples in which $A = 2$, then we project the result of this selection over the future variables (i.e., B, I and V). In terms of relational algebra, the set S_a of tuples acceptable by C_5 can be written as follows:

$$S_a = \Pi_{B,V,I}(\sigma_{A=2}(C_5)) \tag{9.9}$$

We then intersect this set with the cross product of the domains of the future variables, which is the set of all tuples that need to be checked and is equal at this point to $D_B \times D_I \times D_V$ = $\{2\} \times \{1, 3\} \times \{3, 4, 5, 6\}$. The intersection is finally projected over each future variable to give the new domain of the variable. In this case, we count one constraint check for each tuple in the cross product. This is consistent with our way to count constraint checks in binary forward checking. However, a constraint check in this context is significantly more expensive than in the binary context. This cost will be apparent in the empirical analysis given.

9.4 Bundling non-binary FC

The computation of domain partitions for a non-binary CSP variable has so far been considered a challenge, and no procedure is reported in literature for this purpose. We report here for the first time how this can be obtained simply by a straightforward extension to the binary case.

In binary CSPs, dynamic bundling is performed by partitioning the domain of a current variable using a data structure that we call the joint discrimination tree, or JDT, defined in Section 2.4.3. We introduce a new structure for partitioning the domain of a variable in a non-binary CSP. This structure, the non-binary discrimination tree (NB-DT), grows from both the *discrimination tree* [Freuder 1991], and the JDT [Choueiry and Noubir 1998], but bears more resemblance to the former. In it, we create a distinct tree for each of the constraints, building nodes according to combinations of variables.

In Figures 9.3, 9.4, 9.5, 9.6, and 9.7 below, we show the non-binary discrimination tree (NB-DT) for each of the constraints incident to V in the example of Section 9.1.

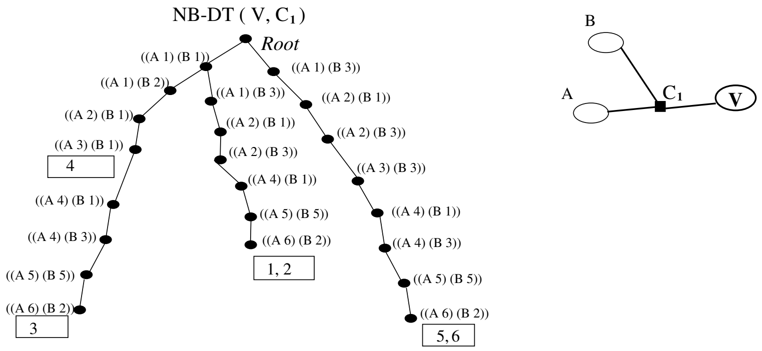


Figure 9.3: *NB-DT for constraint* C_1.

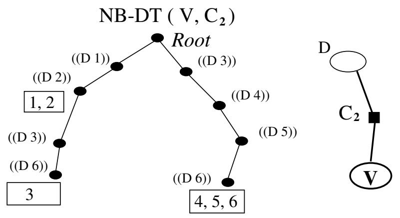


Figure 9.4: *NB-DT for constraint* C_2.

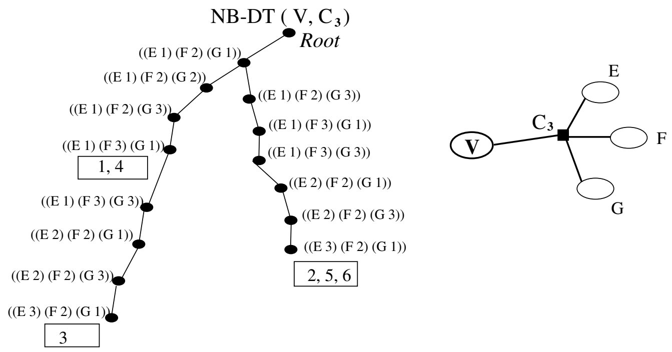


Figure 9.5: *NB-DT for constraint* C_3.

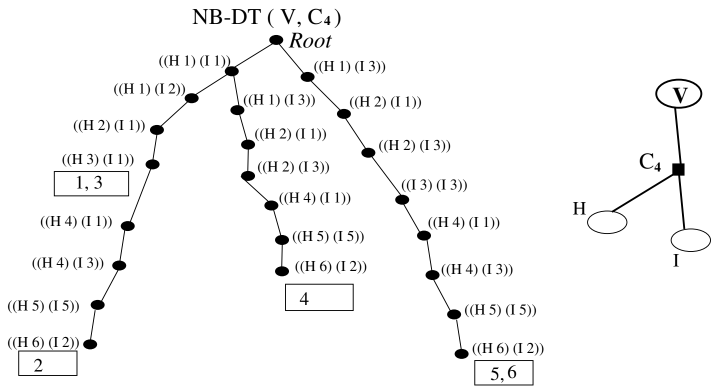


Figure 9.6: *NB-DT for constraint* C_4.

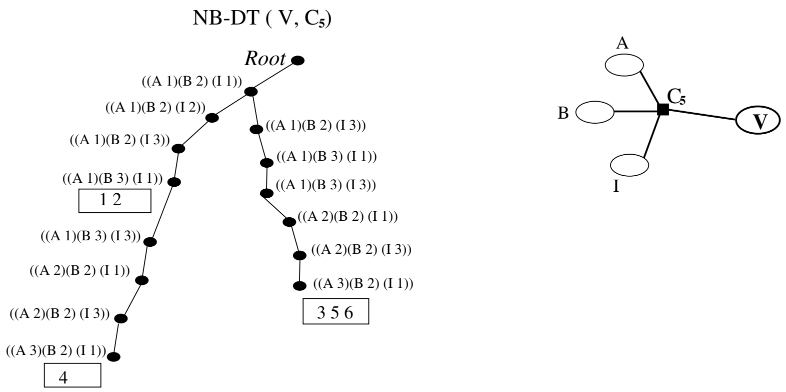


Figure 9.7: *NB-DT for constraint* C_5.

After finding each of these trees, we combine the trees to find the partitioning of the domain for V. This combination requires three sub-tasks:

1. Collect, from each tree, not only the annotations but also the tuples leading, along the path from the root, to each annotation. These tuples, as before, allow us to perform forward checking implicitly.

2. Intersect the domain partitions obtained from each tree.

3. Associate each set in the resulting partition with its corresponding forward checking information. This is done by collecting the path for this partition from *each* of the NB-DT trees, and taking the only the variable-value pairs that are consistent with *all* the constraints (assuming that if a particular variable is not in the NB-DT, any combination of tuples is allowed, because the variables are not constrained).

The execution of this procedure for performing dynamic bundling with an `SLD` variable ordering on the example of Section 9.1 yields the solutions shown in Table 9.3. Note that, as in Section 3.3, we provide no guarantee that this bundling is as compact at it could be.

9.5 Dynamic bundling vs. non-bundling in non-binary CSPs

It is natural to expect that the savings obtained in binary CSPs through dynamic bundling would continue in non-binary CSPs[2]. To test this hypothesis, we compare the performance, for the non-binary case, FC and DNPI-FC, both for static `SLD` and dynamic `DLD` variable orderings. We demonstrate their performance on CSP problems and examples from Section 2.2, and a small battery of randomly generated non-binary CSPs ([Zou *et al.* 2002]). The tests performed are shown in Table 9.4. The same evaluation criteria as used for binary CSPs are reported, namely nodes visited `NV`, constraints checked `CC`, CPU time, and number of solution bundles `SB` when finding all solutions and first bundle size `FBS` when finding a first solution.

[2]It goes beyond the scope of this thesis to carry out an extensive verification of this expectation. Instead, we choose to simply provide here a proof of concept and we leave more detailed investigations as future research work.

A	B	D	E	F	G	H	I	V
(1)	(3)	(4 5 6)	(1)	(3)	(1)	(2 4)	(1)	(5 6)
(1)	(3)	(1 2)	(1)	(3)	(1)	(4)	(1)	(2)
(1)	(3)	(1 2)	(1)	(3)	(1)	(1 2 3)	(1)	(1 2)
(1)	(3)	(4 5 6)	(1)	(3)	(1)	(1 2 3 4)	(3)	(5 6)
(1)	(3)	(4 5 6)	(1)	(3)	(3)	(1 2 3 4)	(3)	(5 6)
(1)	(3)	(4 5 6)	(1)	(3)	(3)	(2 4)	(1)	(5 6)
(1)	(3)	(1 2)	(1)	(3)	(3)	(1 2 3 4)	(1)	(2)
(1)	(3)	(1 2)	(1)	(2)	(2)	(1 2 3)	(1)	(1)
(1)	(3)	(4 5 6)	(1)	(2)	(1 3)	(1 2 3 4)	(3)	(5 6)
(1)	(3)	(4 5 6)	(1)	(2)	(1 3)	(2 4)	(1)	(5 6)
(1)	(3)	(1 2)	(1)	(2)	(1 3)	(4)	(1)	(2)
(1)	(3)	(1 2)	(1)	(2)	(1 3)	(1 2 3)	(1)	(1 2)
(1)	(3)	(4 5 6)	(2)	(2)	(1 3)	(1 2 3 4)	(3)	(5 6)
(1)	(3)	(4 5 6)	(2)	(2)	(1 3)	(2 4)	(1)	(5 6)
(1)	(3)	(1 2)	(2)	(2)	(1 3)	(1 2 3 4)	(1)	(2)
(1)	(3)	(4 5 6)	(3)	(2)	(1)	(1 2 3 4)	(3)	(5 6)
(1)	(3)	(4 5 6)	(3)	(2)	(1)	(2 4)	(1)	(5 6)
(1)	(3)	(1 2)	(3)	(2)	(1)	(1 2 3 4)	(1)	(2)
(1)	(2)	(4 5 6)	(1)	(3)	(1)	(1)	(2)	(4)
(1)	(2)	(4 5 6)	(1)	(3)	(1)	(1 2 3)	(1)	(4)
(1)	(2)	(4 5 6)	(1)	(2)	(1 2 3)	(1)	(2)	(4)
(1)	(2)	(4 5 6)	(1)	(2)	(1 2 3)	(1 2 3)	(1)	(4)

Table 9.3: *Solutions to the non-binary CSP example using DNPI-FC-*`SLD`*; 380 solutions in 22 bundles.*

Non-binary CSPs: To bundle or not to bundle?		
Compared strategies	Orderings	Problem pools
FC versus DNPI-FC	`SLD`, `DLD`	Puzzles, Examples and Random problems

Table 9.4: *Tests for non-binary search strategies.*

Toy problems	**Real-life problems**	**Other**
Vision	Xerox reprographic machine	DB example
Zebra		Example in Figure 9.1

Table 9.5: *Non-binary CSP tests.*

9.5.1 Non-binary puzzles and examples

Table 9.5 recalls the non-binary CSPs listed in Section 2.2. We include also the example CSP of Section 9.1. Table 9.6 compares non-bundling and dynamic bundling with `SLD` and `DLD` orderings on these problems. Note that in these problems, almost no bundling is possible. Therefore, we expect our non-binary DNPI-FC to not perform well on these problems. This is apparent in Table 9.6.

		Finding one solution				Finding all solutions			
	Search	`NV`	`CC`	`FBS`	Time	`NV`	`CC`	`SB`	Time
Vision	FC-`SLD`	9	25	1	10	23	48	4	10
	DNPI-FC-`SLD`	9	30	1	10	23	48	4	10
	FC-`DLD`	9	26	1	0	21	46	4	0
	DNPI-FC-`DLD`	9	30	1	0	21	46	4	0
Zebra	FC-`SLD`	101	5927	1	380	167	11191	1	650
	DNPI-FC-`SLD`	101	6790	1	360	167	11191	1	640
	FC-`DLD`	64	2074	1	90	76	2286	1	90
	DNPI-FC-`DLD`	64	2091	1	90	76	2286	1	100
Xerox	FC-`SLD`	7	19012	1	4850	199	37636	97	20430
	DNPI-FC-`SLD`	7	37636	2	20160	199	37636	97	20430
	FC-`DLD`	7	19012	1	4580	394	56454	194	36200
	DNPI-FC-`DLD`	7	37636	2	19520	199	37636	97	19790
DB	FC-`SLD`	10	28	1	0	44	99	5	0
	DNPI-FC-`SLD`	10	46	1	0	41	99	4	10
	FC-`DLD`	10	30	1	0	35	94	5	10
	DNPI-FC-`DLD`	10	44	1	0	34	94	4	0
Figure 9.1	FC-`SLD`	11	87	1	0	829	1400	380	180
	DNPI-FC-`SLD`	10	174	27	20	121	852	22	50
	FC-`DLD`	9	117	1	10	570	755	380	50
	DNPI-FC-`DLD`	9	183	27	0	91	668	20	30

Table 9.6: *Dynamic bundling vs. non-bundling search in non-binary CSP benchmarks.*

When finding only one solution, DNPI-FC visits no more nodes than FC. But, it generally checks more constraints and requires more CPU time. However, this effect is reversed when finding all solutions. For the Xerox PARC problem, we begin to observe the savings that result from dynamic bundling. Some bundling is performed, and the search is completed in about 2/3 the time of FC without bundling. The real savings is shown in the Example problem from Figure 9.1, where more bundling is possible. As we see in Table 9.6, there is a lot of interchangeability in this problem, and DNPI-FC creams non-bundling FC—finding even *one* solution bundle (of 27 near solutions) faster than FC finds one solution when using `DLD` variable ordering and demonstrating well the savings of bundling.

To see if dynamic bundling provides an advantage *in general* for non-binary CSPs, we need to look at a wider range of problems, such as randomly generated ones.

9.5.2 Non-binary random CSP results

We used the random generator of non-binary CSPs by Zou Hui [Zou *et al.* 2002] with 10 variables ($n = 10$), a fixed domain size ($a = 5$), constraint probability $p = \{.2, .5\}$ (CSPs with a higher probability were impossible to generate using this experimental generator). Constraint tightness t was chosen from 0.05 to 0.95 by steps of 0.05. Each problem had an even distribution of constraints with arity 2, 3, and 4. We generated 20 random instances for each value of density and tightness, and averaged the values of `NV`, `CC`, `SB`, and CPU time over the 20 instances. Numerical results for $t \leq 0.45$ are reported in Tables 9.7 and 9.8. For $t > 0.45$, all CSPs were not solvable. For $t = 0.05$, the non-bundling search strategy could not complete an instance of any of the CSPs in less than two hours CPU time, which forced us to interrupt it. Therefore, these (when $t = 0.05$) are also not reported here.

From Table 9.7, we see that DNPI-FC for non-binary problems requires a more careful implementation in order to win the competition with non-bundling FC for finding one solution. DNPI-FC may expand fewer nodes when finding one solution, but it never checks fewer constraints[3]. However, even in these preliminary stages, DNPI-FC takes less time for finding one solution when $t > 0.15$ and $p = 0.5$. Therefore, with a more efficient implementation, it is easy to conjecture that a dynamic bundling algorithm will be very useful.

As with binary CSPs, bundling really shines when searching to find all solutions. Once again, we find that dynamic bundling performs *much* better than non-bundling forward checking. In particular, we notice that the following hold:

1. DNPI-FC never visits more nodes than FC.

2. DNPI-FC never checks more constraints than FC.

[3]Using our simplistic model for counting constraint checks.

		Finding one solution							
		NV		CC		FBS		Time [ms]	
t	p	**0.2**	**0.5**	**0.2**	**0.5**	**0.2**	**0.5**	**0.2**	**0.5**
0.05	FC-SLD	10	10.1	1597.65	3585	1	1	199	327
	DNPI-FC-SLD	10	10.05	7751.1	17048.35	32.65	3.6	317	922.5
	FC-DLD	10	10.05	1664.05	3831.15	1	1	151	566.5
	DNPI-FC-DLD	10	10.05	7764.05	16928.9	33.6	7.3	258	799
0.10	FC-SLD	10	17.35	1511.35	3432.15	1	1	104.5	484
	DNPI-FC-SLD	10	17.1	7158.8	14384.05	6.4	1.55	385.5	694
	FC-DLD	10	10.6	1626.35	3419.35	1	1	139.5	473.5
	DNPI-FC-DLD	10	10.6	7100.1	13471.6	7.35	2.65	228.5	571.5
0.15	FC-SLD	10.8	57.5	1456.8	6001.05	1	1	194	420.5
	DNPI-FC-SLD	10.65	57.4	6737	14495.15	5	1.15	241.5	721
	FC-DLD	10	20.3	1578.05	4321.6	1	1	125	602
	DNPI-FC-DLD	10	20.3	6551.05	12258.3	3.6	1.5	202	459
0.20	FC-SLD	12	366.2	1375.65	42523.85	1	1	192	2228.5
	DNPI-FC-SLD	12	366.2	6255.35	48933.7	2.5	1.05	243	1914.5
	FC-DLD	10.15	114.9	1540.8	21916.3	1	1	103.5	1194.5
	DNPI-FC-DLD	10.15	114.9	6039.75	27837.65	2.65	1.1	196.5	1248.5
0.25	FC-SLD	18.1	488.5	1463.75	65137	1	1	182.5	3285
	DNPI-FC-SLD	17.5	488.5	6067.05	68829	1.35	1	209.5	2035
	FC-DLD	11.45	162	1615.8	44756.5	1	1	97.5	2145
	DNPI-FC-DLD	11.45	162	5851	47151	2	1	176.5	3785
0.30	FC-SLD	24.7	0	1796.05	0	1	0	193.5	0
	DNPI-FC-SLD	24.25	0	6242	0	1.9	0	215.5	0
	FC-DLD	12.7	0	1623.1	0	1	0	94.5	0
	DNPI-FC-DLD	12.7	0	5727.5	0	2.25	0	166	0
0.35	FC-SLD	37.9	0	2297.55	0	1	0	193.5	0
	DNPI-FC-SLD	37.85	0	6220.05	0	1.35	0	183	0
	FC-DLD	17.15	0	1783.4	0	1	0	77.5	0
	DNPI-FC-DLD	17.15	0	5342.1	0	1.55	0	152.5	0
0.40	FC-SLD	112.6	0	5089	0	1	0	273	0
	DNPI-FC-SLD	112.3	0	8482.95	0	1.1	0	263.5	0
	FC-DLD	38.55	0	3952.2	0	1	0	149.5	0
	DNPI-FC-DLD	38.45	0	6682.85	0	1.4	0	186.5	0
0.45	FC-SLD	427.47	0	25686.27	0	1	0	736	0
	DNPI-FC-SLD	426.07	0	28074.67	0	1.27	0	741.33	0
	FC-DLD	94.53	0	10757.2	0	1	0	416.67	0
	DNPI-FC-DLD	94.53	0	12687.67	0	1.267	0	302	0

Table 9.7: *Dynamic bundling vs. non-bundling search for one solution in randomly generated non-binary CSPs.*

Table 9.8: *Dynamic bundling vs. non-bundling search for finding all solutions in randomly generated non-binary CSPs.*

		Finding all solutions							
		NV		CC		NB		Time [ms]	
t	p	**0.2**	**0.5**	**0.2**	**0.5**	**0.2**	**0.5**	**0.2**	**0.5**
0.10	FC-SLD	995887.2	33200.25	5077855	974843	564380	7059.95	498830.5	89281.5
	DNPI-FC-SLD	452348.56	28956.75	4755347	970640.5	150637.16	4386.65	156106	47964.5
	FC-DLD	793922.4	17983.1	3945387.2	624972.3	564380	7059.95	419552	47108.5
	DNPI-FC-DLD	283743.7	14318.25	3687239.5	624531.06	109702.7	3606.95	101375	27174.5
0.15	FC-SLD	310503.8	4964.65	2008886.4	321944.8	144771.16	239.25	140559.5	23018
	DNPI-FC-SLD	173081.4	4853.95	1892508.5	321749.3	50449.85	195.7	65992.5	13958.5
	FC-DLD	224656.7	2105.4	1486267.5	191384.95	144771.16	239.25	88713	13496
	DNPI-FC-DLD	106710.35	2036	1450562	191370.66	38459.6	173.55	42036	8239
0.20	FC-SLD	109452.6	1589.7	951677.2	175150.55	38387.5	7.15	71019	11396.5
	DNPI-FC-SLD	75417.25	1587.8	936370.8	175146.5	16682.6	6.65	36768	7214.5
	FC-DLD	69579.35	560.75	667369.7	97105.65	38387.5	7.15	35709.5	6571
	DNPI-FC-DLD	41236.1	559.85	658818.75	97105.65	12996.85	6.25	26249	4296.5
0.25	FC-SLD	37189.95	548.5	434369.2	81778.5	8188.95	0.15	28973.5	4671.5
	DNPI-FC-SLD	30605.65	548.5	428115	81778.5	4836.45	0.15	16411	3229
	FC-DLD	21416	196.7	384522.25	50132.1	8188.95	0.15	19300	3243
	DNPI-FC-DLD	16750.5	196.7	382388.9	50132.1	3965.6	0.15	12329	2295
0.30	FC-SLD	9387.05	202	196539.25	36702.55	974.35	0	10325.5	2682
	DNPI-FC-SLD	8587.55	202	194579.75	36702.55	659.8	0	7075.5	1670
	FC-DLD	3791.2	69.55	108777.9	24503.55	974.35	0	5070.5	1841
	DNPI-FC-DLD	3322.8	69.55	108688.5	24503.55	547.5	0	3800.5	1312
0.35	FC-SLD	3822.95	131.75	124914.65	32039.25	183.25	0	5498	1852.5
	DNPI-FC-SLD	3703.15	131.75	124761.45	32039.25	138.6	0	4071	1418
	FC-DLD	1363.85	48.85	66151.3	20996.35	183.25	0	2863	1470
	DNPI-FC-DLD	1283.85	48.85	66123.05	20996.35	112.25	0	2217	1136
0.40	FC-SLD	1773.75	62.85	74466.15	20020.8	27.4	0	3066	1498.5
	DNPI-FC-SLD	1738.3	62.85	74407.1	20020.8	21.5	0	2519.5	996.5
	FC-DLD	520.55	29.9	40500.3	14063.85	27.4	0	1647.5	1024
	DNPI-FC-DLD	511	29.9	40496.4	14063.85	19.1	0	1319	528
0.45	FC-SLD	975.65	38.75	53272.9	15199.45	2.3	0	1944	992
	DNPI-FC-SLD	968.15	38.75	53247.25	15199.45	2.05	0	1616	505
	FC-DLD	215.8	20.8	23127.85	10966.8	2.3	0	935.5	836
	DNPI-FC-DLD	215.1	20.8	23126.95	10966.8	1.75	0	697	378.5

3. DNPI-FC is always able to perform bundling, but this bundling is not spectacular. Notice that in general `SB`(DNPI-FC) is only about half that of FC. This means that every bundle contains about 2 solutions.

4. DNPI-FC always takes much less time to find all solutions than FC. The CPU time taken by DNPI-FC is up to an order of magnitude less than that of FC, showing the true savings of bundling *even in the face of a non-optimal implementation*.

Additionally, we make the following general observations on the data:

Observation 9.5.1. *Similarity of DNPI-FC and FC:* When there are no solutions to be found ($p = 0.5, t > .25$), DNPI-FC and FC visit exactly the same number of nodes, and check the same number of constraints.

This is surprising because it means that DNPI-FC was not able to bundle no-good sets as it had been able to before.

Observation 9.5.2. *Phase transitions in non-binary CSPs:* We are able to observe a phase transition in these non-binary CSPs, as we were in the binary CSPs. When finding one solution, we notice that the highest CPU times correspond with the largest number of constraint checks at $t =$ 0.45 when $p =$ 0.2, and at $t =$ 0.25 when $p =$ 0.5. Such a phase transition for non-binary CSPs has not before been observed or studied.

When finding all solutions, a phase transition is not applicable because of the high cost of finding a large number of 'easy to find' solutions.

Observation 9.5.3. *Dynamic vs. Static Variable ordering:* We see that, as with binary CSPs, the performance of search with a `DLD` ordering heuristic is in general better than that of a search with an `SLD` ordering heuristic. Exceptions occur only at $t =$ 0.10 and 0.20 when $p =$ 0.2 for finding one solution. Again, we find that the most effective technique is DNPI-FC with `DLD` variable ordering.

Summary

In the much-closer-to-real-world realm of non-binary CSPs, dynamic bundling proves itself a worthwhile endeavor. We demonstrate that in non-binary CSPs, dynamic bundling is capable of bundling the solution space of CSPs and of solving them much faster than a non-bundling search procedure. This result is a particularly interesting one, heavy with promise for new applications such as design and relational databases.

Chapter 10

Conclusions

We have studied the advantages of *bundling* values of variables when solving a constraint satisfaction problem. We have demonstrated that *dynamic* bundling interleaved with forward checking search is a more effective method for solving Constraint Satisfaction Problems than non-bundling search strategies, static bundling search strategies, or full lookahead search strategies. We thus disprove all doubts against the utility and practicality of dynamic bundling. This holds true in general both when finding one solution or all solutions of binary CSPs, using static variable ordering, dynamic variable ordering, or dynamic variable-value ordering. Further, we have demonstrated that the concept of dynamic bundling is useful in the context of non-binary CSPs. We have shown these statements to be true of CSPs exhibiting all ranges of constraint probability and constraint tightness—including problems that are contrived to be difficult (puzzles). We have also shown the effects of the presence and absence of interchangeability on these search strategies. Importantly, we have established how they can reduce the peak of the phase transition in CSPs, thus confirming our intuitions and the promises of symmetry and uncovering a new venue for overcoming complexity in problem solving.

10.1 Summarizing our contributions

We review the concept of interchangeability, categorized by [Freuder 1991], and introduce a new kind of interchangeability, *dynamic neighborhood partial interchangeability*, or DNPI. We show how to integrate both the static interchangeability of [Haselböck 1993] (NI_C) and DNPI into search with forward checking.

We compared three search strategies: FC (non-bundling forward checking), NIC-FC (static-bundling forard checking) and DNPI-FC (dynamic-bundling forward checking) both theoretically and empirically. Theoretically, we established that the relations recalled in Figure 10.1 hold when solving for *all solutions* with a *static* variable ordering (such as `SLD`).

Num. of Nodes Visited	Num. of Constraints Checks	Num. of Solution Bundles
$\mathrm{FC} \xrightarrow{\succcurlyeq} \mathrm{NI}_C \xrightarrow{\succcurlyeq} \mathrm{DNPI}$	NI_C $\mathrm{FC} \xrightarrow{\succcurlyeq} \mathrm{DNPI}$	$\mathrm{FC} \xrightarrow{\succcurlyeq} \mathrm{NI}_C \xrightarrow{\succcurlyeq} \mathrm{DNPI}$

Figure 10.1: *Theoretical comparisons of bundling strategies.*

We demonstrated that dynamic bundling can be further improved by a dynamic variable ordering such as `DLD` or a dynamic variable-value ordering such as `LD-MB`. The variable-value orderings `promise` and `Max-Bundle`, a new ordering heuristic, were also tested. `Promise` was found to be effective for finding one solution to a CSP with a minimal number of nodes expanded but consumed too much CPU time. `Max-Bundle` was an altogether poor heuristic, which seemed counter intuitive but is now understood. All combinations of bundling with dynamic variable or dynamic variable-value orderings were designed and implemented only by us.

We showed that in addition to finding all solutions, dynamic bundling remains a good strategy for finding one solution, particularly if the CSP lies at the phase transition, see Section 8.2.

We demonstrated that the interchangeability exploited by both static and dynamic bundling can be controlled and changed without changing the overall difficulty of the CSP problem. Therefore problems with various amounts of structure can be generated randomly, see Section 6.2. The interchangeability present in the structured random problems is consistently found and exploited by our algorithms. Even when a problem had been constructed to contain no static interchangeability, our algorithms were able to find and exploit *some* interchangeability dynamically.

We compared forward checking (DNPI-FC) to a full lookahead strategy (DNPI-MAC) with dynamic bundling. This combination was found fruitful *only* when using static variable ordering, see Section 7.2. In particular, DNPI-MAC was helpful for reducing the effort at the phase transition, or when constraint probability, p, was low and constraint tightness, t, high. In all other situations, DNPI-FC was a more effective search strategy.

Finally, we demonstrated that the concepts introduced here easily, naturally, and advantageously extend to non-binary CSPs, see Chapter 9.

10.2 Future work

This study has uncovered a large number of new exciting avenues for further research. Below we sketch a few of these:

1. *Relational databases:* The problem of computing the natural join of a number of relations in relational databases can easily be modeled as a CSP, with each table as a constraint, each attribute a variable, and the entries in a column the domain of the variable. Such an application is sure to benefit from interchangeability both in its computation and in its compact representation of the solution space, and thus the disk storage.

2. *AI Planning:* Planning problems are likely to benefit from the presence of interchangeability. In particular GraphPlan [Blum and Furst 1997] can be formulated as a CSP [Kambhampati 1999], in which interchangeability detection and exploitation methods would be directly applicable.

3. *Human-Computer Interaction:* In this thesis, we focused on proving the utility of bundling techniques for improving the performance of search. However, this is only half the story. The most important aspect of using symmetries resides in their potential to support human users during problem solving as identified by Choueiry [1994] and investigated by Melissargos [2000]. We believe that bundles provide the building blocks for a new paradigm for high-level interactions with users.

4. *General symmetries:* The Constraint Systems Laboratory is working on detecting functional interchangeabilities which were also introduced by [Freuder 1991].

5. *Continuous CSPs:* We firmly believe that the use of interchangeability should be extended to CSPs with continuous domains, especially CSPs with monotonic constraints, functional constraints, and what we call pseudo-functional constraints, which are constraints that can be represented by block-diagonal binary matrices.

6. *Backbone and SAT:* We would like to investigate whether interchangeability is connected to the idea of a *backbone* in SAT [Parkes 1997].

7. *Solution-preserving random generators:* A random CSP generator that is guaranteed to preserve one solution, even in dense, tight problems would be quite useful. Such a generator has been proposed by [Xu and Li 2000]. We would like to implement this generator, giving us the ability to test our search strategies on an even wider range of CSPs.

8. *Phase transition in non-binary CSPs:* We can see evidence for the existence of a phase transition in non-binary CSPs in the results of Chapter 9. The presence and characteristics of the phase transition in relation to varying the arity of constraints should be explored.

9. *Dynamic variable-value ordering heuristics for non-binary CSPs:* We confirmed that an intelligent variable-value ordering can make a significant difference in solving a binary CSP. The development and study of dynamic variable-value ordering heuristics for non-binary CSPs are likely to reveal a similar improvement. In Chapter 9, we implement `SLD` and `DLD`. Additional ordering heuristics for non-binary CSPs could be developed using the same principles for variable-value ordering employed in binary CSPs.

10. *Implementation of non-binary constraint checks:* Our current implementation of a non-binary constraint check is quite inefficient. We are investigating better ways to perform these checks. We expect this direction to have important implications on practical applications and open new horizons.

11. *Benchmark problems:* Though random CSPs are required to demonstrate the utility of a new algorithm, they are not considered representative of real-world problems. Developing such a library and making it publicly available on the World Wide Web would constitute a significant service to the research community.

10.3 Final note

In this study we thoroughly validated through both theoretical and empirical means the claims that the detection of symmetry relations improves the performance of problem solving. We refuted the myths that dynamic bundling is too expensive to be worthwhile and that full lookahead strategies such as MAC are always beneficial. Furthermore, we established in our pursuit that bundling techniques are actually a powerful, cost-effective tool for dramatically reducing the peak of the phase transition, which has been established as the most critical phenomenon challenging the efficient processing of highly combinatorial problems in practice. Thus, our study ends with a new confidence of ways to overcome the complexity barrier that has hindered the advances of AI in the last three decades.

References

Abramson, Bruce and Yung, Moti 1989. Divide and conquer under global constraints: A solution to the n-queens problem. *J. Parallel and Distributed Computing* 6:649–662. 12 C.A. Herrmann and C. Lengauer.

Achlioptas, Dimitris; Gomes, Carla P.; Kautz, Henry A.; and Selman, Bart 2000. Generating satisfiable problem instances. In *AAAI/IAAI*. 256–261.

Bacchus, Fahiem and Run, Paulvan 1995. Dynamic Variable Ordering in CSPs. In *Principles and Practice of Constraint Programming, (CP'95). Lecture Notes in Artificial Intelligence #976, Springer Verlag.* 258–275.

Backofen, Rolf 2001. The protein structure prediction problem: A constraint optimization approach using a new lower bound. *Constraints* 6(2/3):223–255.

Beckwith, Amy M. and Choueiry, Berthe Y. 2001. On the Dynamic Detection of Interchangeability in Finite Constraint Satisfaction Problems. In Walsh, Toby, editor 2001, *Proceedings of* 7^{th} *International Conference on Principle and Practice of Constraint Programming (CP'01)*, volume 2239 of *Lecture Notes in Computer Science*, Paphos, Cyprus. Springer Verlag. 760.

Beckwith, Amy M.; Choueiry, Berthe Y.; and Zou, Hui 2001. How the Level of Interchangeability Embedded in a Finite Constraint Satisfaction Problem Affects the Performance of Search. In *AI 2001: Advances in Artificial Intelligence, 14th Australian Joint Conference on Artificial Intelligence. Lecture Notes in Artificial Intelligence LNAI 2256*, Adelaide, Australia. Springer Verlag. 50–61.

Benhamou, Beliad 2000. Symmetry and Dominance in Constraint Satisfaction Problems (Draft). Research Report TR-353, LIM, Centre de Mathématiques et Informatique de Marseille.

Bessière, Christian; Meseguer, Pedro; Freuder, Eugene; and Larrosa, Javier 1999. On forward checking for non-binary constraint satisfaction.

Blum, Avrim L. and Furst, Merrick L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Brown, Cynthia A.; Finkelstein, Larry; and Purdom, Jr., Paul W. 1988. Backtrack Searching in the Presence of Symmetry. In Mora, T., editor 1988, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*. Springer-Verlag. 99–110.

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

Choueiry, Berthe Y. and Noubir, Guevara 1998. On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems. In *Proc. of AAAI-98*, Madison, Wisconsin. 326–333.

Choueiry, Berthe Y.; Faltings, Boi; and Weigel, Rainer 1995. Abstraction by Interchangeability in Resource Allocation. In *Proc. of the 14* th *IJCAI*, Montréal, Québec, Canada. 1694–1701.

Choueiry, Berthe Y. 1994. *Abstraction Methods for Resource Allocation*. Ph.D. Dissertation, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. Thesis No 1292.

Dechter, Rina and Pearl, Judea 1989. Tree Clustering for Constraint Networks. *Artificial Intelligence* 38:353–366.

Ellman, Thomas 1993. Abstraction via Approximate Symmetry. In *Proc. of the 13* th *IJCAI*, Chambéry, France. 916–921.

Fillmore, Jay P. and Williamson, S. Gill 1974. On backtracking: A combinatorial description of the algorithm. *SIAM J. Comput.* 3(1):42–55.

Fox, Mark 1987. *Constraint Directed Search: A Case Study of Job-Shop Scheduling*. Morgan and Kaufmann, Los Altos, CA.

Freuder, Eugene C. and Sabin, Daniel 1997. Interchangeability Supports Abstraction and Reformulation for Multi-Dimensional Constraint Satisfaction. In *Proc. of AAAI-97*, Providence, Rhode Island. 191–196.

Freuder, Eugene C. 1978. Synthesizing Constraint Expressions. *Communications of the ACM* 21 (11):958–966.

Freuder, Eugene C. 1991. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proc. of AAAI-91*, Anaheim, CA. 227–233.

Geelen, Pieter Andreas 1992. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *Proc. of the 10 th ECAI*, Vienna, Austria. 31–35.

Gent, Ian P. and Prosser, Patrick 2000. Inside MAC and FC. Technical Report APES-20-2000, APES Research Group. Available from `http://www.dcs.st-and.ac.uk/ apes/apesreports.html`.

Ginsberg, Matthew L.; Parkes, Andrew J.; and Roy, Amitabha 1998. Supermodels and Robustness. In *Proc. of AAAI-98*, Madison, Wisconsin. 334–339.

Glaisher, James Whitbread Lee 1874. On the Problem of the Eight Queens. *Philosophical Magazine, series 4* 48:457–467.

Gyssens, Marc; Jeavons, Peter; and Cohen, David A. 1994. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence* 66(1):57–89.

Haralick, Robert M. and Elliott, Gordon L. 1980. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* 14:263–313.

Haselböck, Alois 1993. Exploiting Interchangeabilities in Constraint Satisfaction Problems. In *Proc. of the 13 th IJCAI*, Chambéry, France. 282–287.

Hogg, Tad; Hubermann, Bernardo A.; and Williams, Colin P., editors 1996. *Special Volume on Frontiers in Problem Solving: Phase Transitions and Complexity*, volume 81. Elsevier Science.

Hubbe, Paul D. and Freuder, Eugene C. 1989. An Efficient Cross Product Representation of the Constraint Satisfaction Problem Search Space. In *Proc. of AAAI-92*, San Jose, CA. 421–427.

Kambhampati, Subbarao 1999. Planning graph as (dynamic) csp: Exploiting ebl, ddb and other csp techniques in graphplan.

Kanellakis, Paris C.; Kuper, Gabriel M.; and Revesz, Peter Z. 1990. Constraint query languages. In *Symposium on Principles of Database Systems*. 299–313.

Kapadia, Ravi and Fromherz, Markus 1998. Data for Configuration of Xerox Reprographic Machines. Personal communication.

Kautz, Henry A.; Ruan, Yongshao; Achlioptas, Dimitris; Gomes, Carla P.; Selman, Bart; and Stickel, Mark 2001. Balance and filtering in structured satisfiable problems. In *Proc. of the 17 th IJCAI*, Seatte, WA. 351–358.

Kondrak, Grzegorz and Beek, Petervan 1995. A Theoretical Evaluation of Selected Backtracking Algorithms. In *Proc. of the 14 th IJCAI*, Montréal, Québec, Canada. 541–547.

Kumar, Vipin 1992. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine* 13 (1):32–44.

Lesaint, David 1994. Maximal Sets of Solutions for Constraint Satisfaction Problems. In *Proc. of the 11 th ECAI*, Amsterdam, The Netherlands. 110–114.

Mackworth, Alan K.; Mudler, Jan A.; and Havens, William S. 1985. Hierarchical Arc Consistency: exploiting structured domains in constraint satisfaction problems. *Computational Intelligence* 1:118–126.

Mackworth, Alan K. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8:99–118.

Melissargos, Georgios 2000. *Interactive Visualization for Resource Allocation Tasks*. Ph.D. Dissertation, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. Thesis No 2166.

Meseguer, Pedro 1989. Constraint Satisfaction Problems: An Overview. *AI Communications, the European Journal on Artificial Intelligence* 2 (1):3–17.

Parkes, Andrew J. 1997. Clustering at the phase transition. In *AAAI97*. 340–345.

Petrie, Charles; Jeon, Heecheol; and Cutkosky, Mark R. 1996. Combining Constraint Propagation and Backtracking for Distributed Engineering. Research Report CDR TR 1996081, Center for Design Research, Stanford University.

Pierce, C. S. 1933. In Hartshorne, C. and Weiss, P., editors 1933, *Collected Papers, Vol. III*. Harvard University Press, 1933. Cited in: F. Rossi, C. Petrie, and V. Dhar, 1990.

Prosser, Patrick 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9 (3):268–299.

Prosser, Patrick 1996. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence* 81(1-2):81–109.

Puget, Jean-Francois 1993. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In *ISMIS'93*. 350–361.

Revesz, Peter 2002. *Introduction to Constraint Databases*. Springer-Verlag.

Rossi, Francesca; Petrie, Charles; and Dhar, Vasant 1990. On the Equivalence of Constraint Satisfaction Problems. In *Proc. of the 9 th ECAI*, Stockholm, Sweden. 550–556.

Sabin, Daniel and Freuder, Eugene C. 1994. Contradicting Conventional Wisdom in Constraint Satisfaction. In Borning, Alan, editor 1994, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA*, volume 874. 10–20.

Sabin, Daniel and Freuder, Eugene C. 1997. Understanding and improving the MAC algorithm. In *Principles and Practice of Constraint Programming*. 167–181.

Silaghi, Marius Calin; Sam-Haroud, Djamila; and Faltings, Boi 1999. Ways of Maintaining Arc Consistency in Search using the Cartesian Representation. In *Proc. of ERCIM'99*, LNAI, Paphos, Cyprus. Springer Verlag.

Sosic, Rok and Gu, Jun 1990. A polynomial time algorithm for the N-queens problem. *SIGARTN: SIGART Newsletter (ACM Special Interest Group on Artificial Intelligence)* 1.

Tsang, Edward 1993a. *Foundations of Constraint Satisfaction*. Academic Press, London, UK.

Tsang, Edward 1993b. *Foundations of Constraint Satisfaction, Chapter 6*. Academic Press, London, UK.

Xu, Lin and Choueiry, Berthe Y. 2001. A Comparative Study of Arc-Consistency Algorithms. Working Note of the Constraint Systems Laboratory (UNL). September, 2001.

Xu, Ke and Li, Wei 2000. Exact phase transitions in random constraint satisfaction problems. *JAIR* 12:93–103.

Zou, Hui; Beckwith, Amy M.; and Choueiry, Berthe Y. 2001. A Generator of Random Instances of Binary Finite Constraint Satisfaction Problems with Controllable Levels of Interchangeability. Technical Report CONSYSTLAB-01-01. Available from `http://consystlab.unl.edu/CSL-01-01.doc`, University of Nebraska-Lincoln.

Zou, Hui; Choueiry, Berthe Y.; and Davis, Amy M. 2002. A Generator of Random Non-Binary Finite Constraint Satisfaction Problems. Technical Report CONSYSTLAB-02-02. Available from http://consystlab.unl.edu/CSL-02-02.ps, University of Nebraska-Lincoln.

Index