NEIGHBORHOOD INTERCHANGEABILITY FOR NON-BINARY CSPS & APPLICATION TO DATABASES

by

Anagh Lal

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

May, 2005

NEIGHBORHOOD INTERCHANGEABILITY FOR NON-BINARY CSPS & APPLICATION TO DATABASES

Anagh Lal, M.S.

University of Nebraska, 2005

Advisor: Berthe Y. Choueiry

Neighborhood Interchangeability (NI) identifies the equivalent values in the domain of a variable in a Constraint Satisfaction Problem (CSP). We introduce for the first time an algorithm for computing NI sets in the presence of non-binary constraints. We integrate this mechanism with backtrack search, in a process we call dynamic bundling. We demonstrate that, as for the binary case [Beckwith *et al.*, 2001], dynamic bundling yields multiple robust solutions for less effort than necessary for computing a single solution.

We then identify the utility of this mechanism for database applications and introduce a new algorithm based on dynamic bundling for computing a join query, which we model as a CSP. We argue that the algorithm yields a compact solution space and saves memory, disk-space, and/or network bandwidth. Finally, we discuss the application of the join algorithm to materialize views.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Berthe Y. Choueiry, for her tireless support and guidance since the beginning of this research. Without her help this work would not have been possible. I would like to thank my committee members, Professors Matthew Dwyer, Steve Goddard, and Peter Revesz for their careful reading and insightful comments, which allowed me to significantly improve this document. In particular, Professor Peter Revesz inspired the idea discussed in Section 6.1.6 for an alternative approach for applying and exploiting the mechanisms exposed in this thesis.

I would like to acknowledge the invaluable guidance I have received for the statistical analysis of my empirical results from the Statistics Department's help-desk, especially from Mr. Bradford Danner. Mr. Mansour Abdoli, from the Department of Industrial and Management Systems Engineering at UNL, has helped us getting started with our statistical analysis. I am indebted to the Office of System Administration and the managers of the Research Computing Facilities of the Department of Computer Science and Engineering for their prompt and quality support. I had the pleasure of working with excellent colleagues at the Constraint Systems Laboratory (ConSystLab). I would like to acknowledge the help of Joel Gompert, Praveen Guddeti, Ryan Lim, and Yaling Zheng. I would like to thank Joel Gompert, Ryan Lim, and Cate Anderson for proof-reading various portions of my thesis (and in the case of Joel, the entire document).

I would also like to acknowledge the constant support of my friends. Finally, I am grateful to my parents and my brother who have provided me with constant motivation to maintain my commitment. This work is a fruit of their confidence in me throughout my life.

Supported by NSF CAREER Award #0133568 and the Maude Hammond Fling Faculty Research Fellowship. Experiments conducted using the Research Computing Facility of UNL.

Dedication

To my parents, who have always been a great source of support and inspiration.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Overview

The study of symmetry is receiving increased attention in Computer Science, in general, and in the area of Constraint Processing in particular. Most approaches try to exploit known symmetries in order to improve the performance of problem solving. Relatively less effort is devoted to uncovering symmetries inherent to a given problem instance, which is considered to be a computationally challenging task. Our work fits in the latter category.

In this thesis, we are concerned with the study of symmetry in decision problems, modeled as Constraint Satisfaction Problems (CSPs). In particular, we propose techniques for detecting symmetry relations and for exploiting these relations for reasoning and problem solving. Symmetry has been exploited to improve the performance of search at least as far back as 1874 [Glaisher, 1874]. Recently there has been a series of workshops on symmetry and CSPS [SymCon, 2003; 2004]. Our study focuses on the discovery and use of approximate symmetries during search that yield multiple, robust solutions. The symmetry relations we discuss are based on the notions of *local value interchangeability* [Freuder, 1991], which group equivalent values of a given variable in a bundle.

Most of the research in Constraint Satisfaction has focused on problems with binary constraints (i.e., binary CSPs). One can theoretically always reduce a non-binary CSP into a binary one [Rossi *et al.*, 1990], but it can be impractical in the case of large constraints of high arity. In this thesis, we address two issues in the study of symmetry:

1. The computation of a special form of symmetry known as neighborhood interchangeability in the presence of non-binary constraints.

2. The integration of the above mechanism with search as a process for solving the CSP and for finding sets of multiple robust solutions.

We validate our approach in two contexts:

1. Theoretical and empirical evaluations of the performance and effectiveness of our techniques on CSPs.

2. Design of a new sort-based join algorithm for databases

In this chapter, we summarize first the questions we answered, then our contributions. Finally, we give a short guide to this document.

1.1 Questions answered

In this thesis, we address the following questions:

1. How to detect neighborhood interchangeability in non-binary CSPs?
   Answer: We establish that the techniques for computing neighborhood interchangeability in binary CSPs cannot be trivially extended to the non-binary case, and propose a technique to compute neighborhood interchangeability in non-binary CSPs.

2. How to exploit neighborhood interchangeability during search for solving non-binary CSPs?
   Answer: The process of interleaving the computation of neighborhood interchangeability with search for solving a CSP was called *dynamic bundling* by Beckwith et al.

[2001]. We show how to implement dynamic bundling in the presence of non-binary constraints. We also show how to adapt the look-ahead strategy of forward checking, which is used for constraint propagation during search, to the context of non-binary CSPs.

3. Is dynamic bundling a viable strategy when looking for a single solution to the CSP?
   Answer: We empirically establish that dynamic bundling significantly improves the performance of search where it matters most, that is in the region of the phase transition where the cost of search peaks. At low tightness regions we show that dynamic bundling yields a large number of solutions at a cost comparable to that of finding a single solution using regular search (i.e., without bundling).

4. How does dynamic bundling behave with varying CSP parameters such as tightness, domain size, number of variables, number of constraints?
   Answer: We designed extensive experiments having datasets with varying CSP parameters to study exactly the above and we found interesting characteristics of the dynamic bundling algorithm.

5. Can we extend the techniques to detect more general forms of interchangeability?
   Answer: We show how the same mechanism for detecting neighborhood interchangeability in binary CSPs can be used to detect some *neighborhood substitutable* values too. We also describe the extension of this feature to non-binary CSPs.

6. Are these techniques useful beyond the area of Constraint Processing?
   Answer: We recognize the direct usefulness of our techniques in one other important area of Computer Science, namely, databases. We identify the gap between the two fields and re-design our algorithms to fit the requirements of this new context.

7. Where and how can dynamic bundling be used in databases?

Answer: We focus on perhaps the most expensive and fundamental operator in databases, namely, the join operator. We design a new join algorithm based on dynamic bundling that improves the speed of the join computation and presents results in a compacted manner. We also show that our new join algorithm is useful for materialized views.

1.2 Other contributions of the thesis

Below we summarize some results, by-products of our investigations:

- We introduced a metric for capturing the practical effort of checking a non-binary constraint.
- We improved the implementation of non-binary forward checking by using select and project operations on non-binary constraints and by storing the resulting partial constraints. This improvement allowed us to conduct fair comparisons between the performance of search with and without dynamic bundling.
- We presented a new approach for modeling a join query as a CSP.
- We showed that the join algorithm can be used as an algorithm for view materialization and can lead to savings in disk space, main memory, and network bandwidth.
- Finally, we identified research directions to pursue in the database area using techniques from CSPs.

1.3 Guide to thesis

This thesis is organized as follows. Chapter 2 reviews background information on CSPs and interchangeability, and discusses issues related to non-binary CSPs and our solutions to these issues. Chapter 3 describes the computation of neighborhood interchangeability

(NI) for domain partitioning and dynamic bundling (i.e., search using dynamically computed NI sets), and empirically validates our approach on randomly generated non-binary CSPs. Chapter 4 discusses how a weaker form of interchangeability called substitutability can be partially extracted from the same mechanism for computing neighborhood interchangeability, and the extension of this idea to non-binary CSPs. Chapter 5 extends our investigations to the context of databases. Finally, Chapter 6 states our conclusions and suggests directions for future research.

Appendix A provides the results of experiments over all datasets that are not included in the body of the dissertation. Appendix B discusses two alternative implementations to the algorithm for computing NI sets presented in Chapter 3. Appendices C and D document, a high level, the main components of our source code (e.g., directory and file structures, data structures, and main functions).

Chapter 2

Binary and Non-Binary CSPs

This chapter provides background information on Constraint Satisfaction Problems (CSPs). It gives the main definitions and notations used in this document. We first recall the definition of a CSP and list its parameters and characteristics. Then, we summarize how to solve CSPs with backtrack search and how to interleave constraint propagation with the search process in a look-ahead strategy known as forward checking. We provide an introduction to interchangeability (i.e., symmetry) restricted to our use of this concept. Finally, we introduce non-binary constraints and discuss our solution to extending forward checking to non-binary CSPs. This solution is fundamental for the implementation of dynamic bundling, which is the main topic of this thesis.

2.1 The Constraint Satisfaction Problem (CSP)

A Constraint Satisfaction Problem (CSP) is defined by $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where:

- $\mathcal{V}$= $\{V_i\}$ is a set of variables.
- $\mathcal{D}$= $\{D_{V_i}\}$ the set of their respective domains. In this thesis, we assume that the domains of the variables are finite. and,

- $\mathcal{C}$ a set of constraints that restrict the acceptable combination of values for variables.

The *scope* of a constraint is the set of variables to which the constraint applies, and its *arity* is the size of this set. We denote by NEIGHBORS(V) the set of variables that appears in the scope of any constraint that applies to a variable V. A constraint over the variables V_i, V_j, ..., V_k is specified as a set of tuples, which is a subset of the Cartesian product of the domains of the variables in its scope:

$$C_{V_i,V_j,\ldots,V_k} = \{(\langle V_i\ a_i\rangle, \langle V_j\ a_j\rangle, \ldots, \langle V_k\ a_k\rangle)^*\} \subseteq D_{V_i} \times D_{V_j} \times \ldots \times D_{V_k}$$

where $a_i \in D_{V_i}$ and $\langle V_i\ a_i\rangle$ denotes a *variable-value pair* (vvp). Solving a CSP requires assigning a value to each variable such that all constraints are simultaneously satisfied. The problem is **NP**-complete in general.

We call a *no-good* any combination of variable-value pairs that cannot be extended to a consistent solution.

2.1.1 Constraint network

A CSP is often represented by a *constraint graph*, or *constraint network*. In this graph, a node represents a variable and is labeled by the corresponding domain. An edge represents a constraint and links the nodes of the variables to which the constraint applies. Figure 2.1 shows the constraint network of a simple CSP instance with four variables and four constraints. For example, constraint C_{V_1,V_3}, defined on V_1 and V_3, states that the two variables cannot have the same value at the same time, C_{V_1,V_3} ={(d, a), (d, b)}. In the CSP shown in Figure 2.1 an example of a no-good is the set of vvps $\{\langle V_2 c\rangle\}, \langle V_3 a\rangle, \langle V_4 a\rangle\}$.

2.1.2 Parameters of a CSP

The parameters used to describe CSPs include the following:

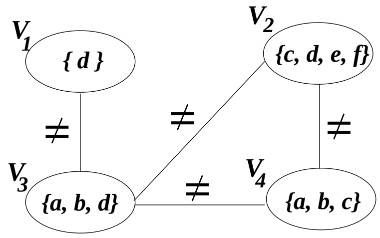

Figure 2.1: Representation of a binary CSP.

- *Number of variables* in the problem, denoted n (i.e., $|\mathcal{V}|$).
- *Maximum domain size*, denoted a.
- *Constraint ratio*, the proportion of constraints, denoted $p = \frac{\text{number of constraints}}{\text{all possible constraints}}$. Sometimes, we use the number of constraints C=$|\mathcal{C}|$.
- *Constraint tightness*, denoted t and measured for a given constraint as

$$t = \frac{\text{number of forbidden tuples}}{\text{total number of possible tuples}}$$

Notice that low values of p may cause the constraint graph to be disconnected (e.g., $C < n-1$). Further, for $p = 1$ (i.e., $C = n(n-1)/2$), the constraint graph is complete. In the example CSP shown in Figure 2.1, we have: $n = 4$, $a = 4$, $p = \frac{4}{6} = 0.67$, and $t =$ is $\frac{1}{3}$ for C_{V_1,V_3}, $\frac{1}{12}$ for C_{V_2,V_3}, $\frac{1}{12}$ for C_{V_2,V_4}, and $\frac{2}{9}$ for C_{V_3,V_4}. The arity of all constraints is obviously 2. Because no constraints have an arity larger than 2, this CSP is called a *binary* CSP.

2.1.3 Random CSPs

Empirical studies on CSPs are typically performed on randomly generated CSP instances with specified values for the above parameters: $\langle n, a, t, p \rangle$, where all variables have the same domain size, the constraints have the same tightness and are uniformly distributed in the graph. Several theoretical models for generating random CSPs have been proposed in the literature [Achlioptas *et al.*, 1997]. In this thesis, we use generators built according to the common and widely used Model-B.

2.1.4 Phase transition

Cheeseman et al. [Cheeseman *et al.*, 1991] presented empirical evidence, for some random combinatorial problems, of the existence of a phase transition phenomenon at a critical value (cross-over point) of an order parameter. They showed a significant increase in the cost of solving these problems around the critical value. Figure 2.2 illustrates this situation.

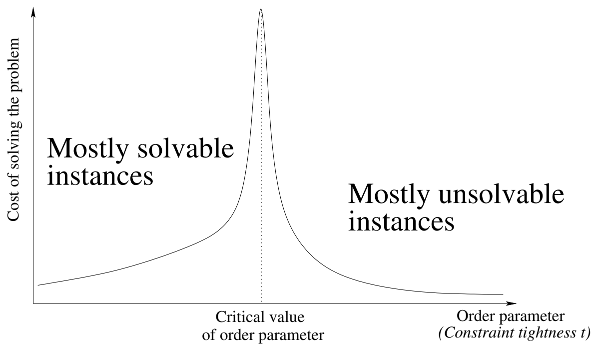

Figure 2.2: Cost of problem solving.

They also showed that the location of the phase transition and its steepness change with the size of the problem. Because problems at the cross-over point are acknowledged to be probabilistically the most difficult to solve, empirical studies to compare the performance of algorithms are typically conducted in this area. In the case of CSPs, constraint tightness (with fixed values for n, a, and p) and constraint ratio (with fixed values for n, a, and t) are often used as order parameters.

2.2 Solving binary CSPs with backtrack search

Because a CSP is in general **NP**-complete[1], it is usually solved with depth-first search using backtracking, an exponential-time procedure.

Depth-first search systematically instantiates (or assigns a value to) one variable at a time, checking to ensure that the instantiation made does not violate any constraints. Depth-first search for binary CSPs proceeds by iteratively choosing a current variable V_c and instantiating it, i.e. assigning to it a value taken from its domain and checking to ensure that the instantiation made does not violate any constraints. If a conflict is detected, the instantiation is revoked, its effects are undone, and an alternative instantiation to the current variable is attempted. When all alternatives fail, search backtracks to the previous assignment, and revokes the assignment done at this level. If the instantiation succeeds, V_c is added to the set of instantiated variables (which we call past variables and denote as $\mathcal{V}_p$) and search proceeds to the next variable determined by an ordering. The process repeats until all variables are instantiated. Uninstantiated variables are called future variables, and their set is denoted by $\mathcal{V}_f$. The process repeats until one or all solutions are found. This procedure creates a *search space* structured as a tree with n levels and a branching factor equal to a.

At any point during search, the path from the root of the tree to the current variable is a set of vvps $\{\langle V_i\ a_i\rangle\}$ for the variables $V_i \in \mathcal{V}_p$ and their instantiations a_i.

2.2.1 Variable ordering

The order in which the variables are considered for instantiation (i.e., variable ordering) and the order in which the values are assigned to a variable (i.e., value ordering) are known to significantly affect the performance of search. Common wisdom dictates to assign first the

[1]By reduction from 3SAT and because a solution is verifiable in polynomial time.

most constrained variable and to choose the most promising values. Many heuristics that implement these principles exist. Further, these heuristics can be applied prior to search, thus determining a static ordering, or during search, thus yielding a dynamic ordering. In practice, dynamic ordering is usually significantly more effective than static ordering. In this thesis, we only consider variable ordering and use the least domain heuristic (LD) for variable ordering, which we apply dynamically (DLD).

2.2.2 Look-ahead: combining search with constraint propagation

Forward checking (FC), a common improvement to backtrack search, is one way of conducting constraint propagation during search [Haralick and Elliott, 1980]. FC ensures that, each time a current variable is assigned a value, the domain of each neighboring future variable is revised to exclude from itself values inconsistent with the assignment of the current variable. This process is called pruning and is shown in Figure 2.3 for the simple example of Figure 2.1. Because of this pruning, FC detects failure early. Further, the

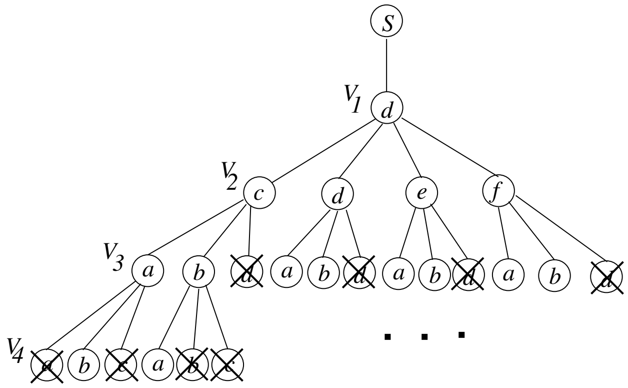

Figure 2.3: Forward checking during search.

domains of all future variables are always consistent, given the binary constraints, with the instantiation of every past variable, thus eliminating the need for back-checking (which is consistency checking against past variables). FC is a *partial* look-ahead technique. It revises the domains of future variables that are neighbors. More 'aggressive' look-ahead techniques exist, such as Directional Arc-Consistency (DAC) [Dechter, 2003] and Maintaining Arc-Consistency (MAC) [Sabin and Freuder, 1994]. However, the former requires a fixed variable ordering and the latter was shown to often be too costly in practice [Yang, 2003]. In this thesis, we use forward checking (FC) and order the variables dynamically during search according to the least domain heuristic. Search on non-binary CSPs proceeds as described above but FC requires particular attention as discussed in Section 2.5.

2.3 Interchangeability

In its broadest sense, interchangeability allows one to recover one solution to a CSP from another [Freuder, 1991]. When solutions to a CSP are given, one can always define a mapping between the solutions such that one solution can be obtained from another without performing search. This is called *functional* interchangeability. Permutation of values across variables is called *isomorphic* interchangeability. In this thesis, we focus our investigation on another restricted form of interchangeability: the interchangeability of values in the domain of a single variable. This type of interchangeability does not cover the permutation of values across variables, which is an isomorphic interchangeability. Below we recall some forms of interchangeability relevant to our work. Note that the discussion in this section primarily targets binary CSPs and may not be applicable to the non-binary case.

2.3.1 Definitions

Definition 1. Full interchangeability (FI) (Freuder [1991]): *Values a, $b \in D_V$ are FI iff every CSP solution involving a remains a solution when b is substituted for a, and vice versa.*

Checking all the solutions of the CSP in Figure 2.4 we find that the values d, e, and f are fully interchangeable for V_2. Computing full interchangeability may require finding

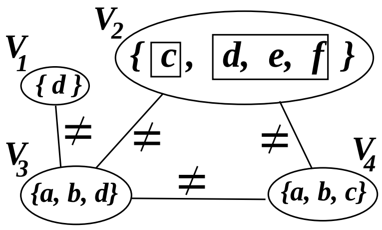

Figure 2.4: A binary CSP.

all solutions and hence is likely to be intractable. Freuder [1991] identified a form of *local* interchangeability, called *neighborhood interchangeability* (NI), that is a sufficient, but not a necessary condition for full interchangeability.

Definition 2. Neighborhood interchangeability (NI) (Freuder [1991]): *A value $a \in D_V$ is neighborhood interchangeable with a value $b \in D_V$ iff for every constraint C on V, a and b are consistent with exactly the same values:* $\{$x | (a, x) satisfies $C\} = \{$x | (b, x) satisfies $C\}$.

NI is a sufficient, but not a necessary condition for FI. Indeed, in the CSP of Figure 2.4, only values e and f are NI for V_2 whereas values e, f, and d are FI for V_2.

2.3.2 Computing NI sets

Algorithm 1 identifies the NI values for a variable V in $O(n \cdot a^2)$ by building a discrimination tree (DT) [Freuder, 1991].

Figure 2.5 shows the discrimination tree generated for V_2 of the CSP in Figure 2.4. In this tree, the nodes represent variable-value pairs in the neighborhood of V_2. Some nodes are annotated with values from D_{V_2}, these annotations form a partition of D_{V_2}. All the variable-value pairs that appear in a path from the root of the tree to an annotation are consistent with the values appearing in the annotation.

It is important, in this procedure, that variables and values be ordered in a canonical way (e.g., lexicographical). For the CSP of Figure 2.4, values e and f are NI for V_2. If we had

Input: V
current-node $\leftarrow$ *Root*, root of the discrimination tree
for *each value* $a \in D_V$ **do**
  for *each variable* $V_j \in$ NEIGHBORS*(V)* **do**
    for *each value* $x \in D_{V_j}$ *consistent with* a *for* V **do**
      if *current-node* has a child node n_t with '$\langle V_j\ x \rangle$' **then** *current-node* $\leftarrow n_t$ **else**
        Generate n_t a node with '$\langle V_j\ x \rangle$' and make it a child of *current-node*
        current-node $\leftarrow n_t$
      end
    end
  end
  Add a to the annotation of *current-node*
  current-node $\leftarrow$ *Root*
end
Output: *Root*

Algorithm 1: Algorithm to create a DT of a variable V.

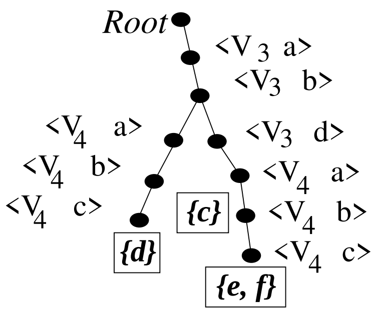

Figure 2.5: Partitioning the domain of V_2.

all the solutions of this CSP we would find that the values d, e, and f are interchangeable for V_2. Identifying such a situation may require finding all solutions to the CSP and hence is likely to be intractable.

2.3.3 Using NI in search

Benson and Freuder used NI to improve search [1992]. A weaker form of NI, called *neighborhood interchangeability according to one constraint* (NI_C), was also used in search by Haselböck [1993]. This search process yields solutions where some variables have a set of equivalent values, called a bundle. Both papers compute interchangeability sets *prior* to search, which corresponds to *static bundling*. Figure 2.6 shows a search tree for the example of Figure 2.4 without bundling (left) and with static bundling (center).

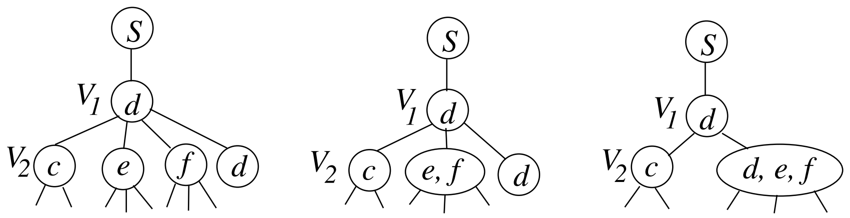

Figure 2.6: Search with no, static, and dynamic bundling.

Freuder [1991] noticed that computing interchangeability *during* problem solving results in a weak type of interchangeability, *dynamic interchangeability*. Beckwith et al. [2001] and Choueiry and Davis [2002] showed how to recompute interchangeability partitions *during* search such that the resulting process, *dynamic bundling* (DynBndl), is always beneficial: it yields larger bundles and reduces the search effort. Figure 2.6 (right) shows the tree generated by dynamic bundling. The computational savings can be traced to:

1. bundles of solutions,
2. factoring out no-goods, and
3. reusing information from the discrimination tree for FC.

Further, they showed that, in comparison to dynamic bundling, static bundling is prohibitively expensive, particularly ineffective, and should be avoided [Choueiry and Davis, 2002].

Finally, note that the Cross Product Representation (CPR) of Hubbe and Freuder [Hubbe and Freuder, 1989] yields the same resulting bundles as dynamic bundling, but it requires more space and does not bundle no-goods. It operates by doing forward checking for every value of the current variable, comparing the CSPs induced on the future variables, and then bundling the values of the current variable yielding the same induced CSPs. Hence, CPR necessarily visits more nodes than DynBndl, even though the difference is polynomially bounded.

2.4 Non-Binary CSPs

Although most research in constraint satisfaction focuses on binary CSPs, many real-life problems are more 'naturally' modeled as non-binary CSPs. The focus on binary CSPs has so far been tolerated because it is always possible in principle to reduce a finite non-binary CSP to a binary one [Rossi *et al.*, 1990; Bacchus and van Beek, 1998]. Research on non-binary constraints is still in its infancy, relatively speaking, and the traditional attitudes on this issue are now being challenged [Bessière *et al.*, 2002]: it appeared that sometimes it is more effective to operate on the non-binary encoding of the CSP than on its binary reduction.

2.4.1 Representation

As stated in Section 2.1.1, a CSP can be represented by a graph, or constraint network. Constraints are represented as simple edges in the binary-case. In the non-binary case, the constraints are represented as hyper-edges linking the nodes in the scope of the constraint. For sake of clarity, we represent a hyper-edge as another type of node connected to the variables in the scope of the constraint, as shown in Figure 2.7.

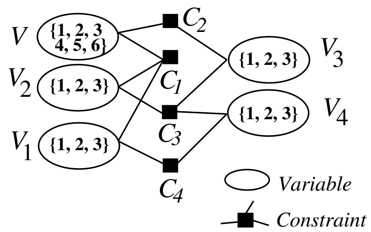

Figure 2.7: Example of a non-binary CSP.

2.4.2 Parameters

We use the parameters listed below to assess the worst-case complexity of an algorithm applied to a non-binary CSP and for generating random instances. They are a superset of the ones listed for the binary case (see Section 2.1.2).

- n number of variables,
- a maximum domain size,
- t constraint tightness defined as the ratio of the number of disallowed tuples over the number of all possible tuples.
- deg node degree,
- c_k number of constraints of arity k,
- $p_k = c_k/\binom{n}{k}$ constraint ratio of arity k, and

2.5 Solving non-binary CSPs with backtrack search

Search on non-binary CSPs proceeds as described in Section 2.2 but FC requires particular attention as discussed below.

2.5.1 Extending FC to non-binary CSPs

Uninstantiated variables are called future variables, and their set is denoted by $\mathcal{V}_f$. Instantiated variables are called past variables, and their set is denoted by $\mathcal{V}_p$. FC propagates the effect of instantiating a current variable V_c by removing values inconsistent with a from the domains of the future variables adjacent to V_c. If the instantiation does not wipe out the domain of any variable in $\mathcal{V}_f$, V_c is added to the set of past variables, $\mathcal{V}_p$. When we apply this process to non-binary constraints, two issues arise:

1. *Choosing the subset of constraints to take into account:* In binary CSPs after assigning V_c a value, the set of constraints to choose is straightforward as either a constraint (applicable to V_c) will have the second variable either in $\mathcal{V}_p$ or in $\mathcal{V}_f$ (but not in both). In the case of non-binary CSPs the set of constraints that apply to V_c may have constraints with some variables in $\mathcal{V}_p$ and other variables in $\mathcal{V}_f$. Figure 2.8 shows an instance of such a partially instantiated constraint.

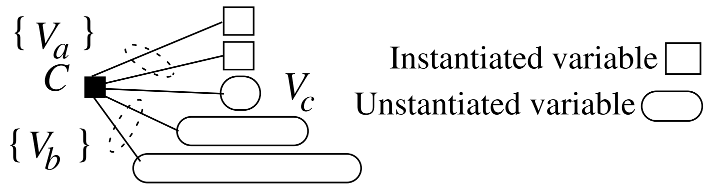

Figure 2.8: Partially instantiated non-binary constraint C.

Bessière et al. discuss the following options [2002]:

- The set of constraints with at least one past variable (including V_c) and at least one future variable.
- The set of constraints or constraint projections with at least one past variable (including V_c) and exactly one future variable.
- The set of constraint with at least one past variable (including V_c) and exactly one future variable.

2. *Updating the constraint definitions to reflect past instantiations and domain prunings:* The update of a non-binary constraint according to past instantiations amounts to intersecting the original definition of the constraint with the Cartesian product of the (updated) domains of V_c and future variables. This operation is time consuming in practice. We propose here an equivalent, more efficient implementation that uses a linear number of selection and projection operations.

2.5.2 Our approach

We adopt the strategy called `nFC2` [Bessière *et al.*, 2002], where the constraints considered are the ones that apply to the current variable and at least one future variable and any number of past variables (if any). We perform the update of a non-binary constraint according to past instantiations as follows. Let V_c be the current variable and C be a non-binary constraint on V_c (see Figure 2.8). Let

$$\text{Scope}(C) = \{\mathcal{V}_a\} \cup \{V_c\} \cup \{\mathcal{V}_b\},$$

where $\mathcal{V}_a \subseteq \mathcal{V}_p$ and $\mathcal{V}_b \subseteq \mathcal{V}_f$. The domains of variables in $\{V_c\} \cup \{V_b\}$ might have already been filtered by FC, and certain tuples in C might have become invalid. Thus, we need to select the tuples of C that have survived the filtering by FC according to instantiations of the past variables. The selected tuples must satisfy the conditions:

1. $\langle V_i \; a_i \rangle$ for $V_i \in \{\mathcal{V}_a\}$ and a_i the bundle instantiated to V_i; and
2. $a_j \in D_{V_j}$ for $V_j \in \{V_c\} \cup \{\mathcal{V}_b\}$, where D_{V_j} are filtered domains.

We denote this operation $\sigma^{FC}_{\mathcal{V}_p}(C)$. In order to compute the updated constraint, we project $\sigma^{FC}_{\mathcal{V}_p}(C)$ on $\{V_c\} \cup \{\mathcal{V}_b\}$,

$$C' = \pi_{\{V_c\} \cup \{\mathcal{V}_b\}}(\sigma^{FC}_{\mathcal{V}_p}(C)). \tag{2.1}$$

2.5.3 Implementing non-binary FC

The way non-binary FC (without bundling) is implemented affects, to a large extent, the number of constraint checks and CPU time spent to solve a CSP. The updated constraint of Equation (2.1) is valid for all values in D_{V_c}. It is wasteful to discard the result of this computation after instantiating V_c. If the instantiation is not consistent and the search backtracks to the variable, then C' is computed again. To avoid this expensive computation we store each C' associated with V_c. Note that by doing so we level the playing field when comparing FC with dynamic-bundling techniques we present in the Chapter 3. Thus, our empirical results reflect the gain due purely to bundling and exclude the gains from any additional data structure.

2.5.4 Measuring constraints checked

The count of the constraint-checking operations during search is an important aspect for evaluating and comparing of the performance of search algorithms [Kondrak and van Beek, 1995]. Because checking a binary constraint is easier than checking a non-binary one, Bacchus and van Beek [1998] proposed to count constraint checks by multiplying each operation by the arity of the constraint being checked. We propose below an alternative way for counting constraint checks that is a closer estimation of the real computational effort spent on this operation.

To this end, we count the number of comparisons of a vvp with a tuple of the constraint. The comparisons done during FC are primarily for two types of checks:

Case 1: checking whether the instantiation of a variable is equal to the value of the variable in a given tuple of a constraint. In this case we count only one comparison. This type of comparison is done to select constraint tuples consistent with past instantiations.

Case 2: checking whether the value for a variable in a constraint tuple is present in the current domain (of size a) of that variable. We may do more than one comparison in this case. In the worst case we will do a comparisons. This type of comparison is done to select, from the constraint definition, tuples that are consistent with the domains of the current and future variables. To illustrate the number of comparisons being made, let us consider an example. Let V_1 be a future variable whose domain is $\{1, 2, 4, 3, 5\}$ (the domain is stored in the order shown). Let C_{V_1,V_2,V_3} be a constraint on V_1. Let $t = \{\langle V_2\ 1\rangle\ \langle V_3\ 2\rangle\ \langle V_1\ 3\rangle\}$ be a tuple from the constraint C_{V_1,V_2,V_3}. It takes 4 comparisons to check whether t is valid given the domain of variable V_1. Had the value 3 been at a different position in the domain then the number of comparisons would have been different.

A constraint check over a k-ary constraint involves a maximum of k such checks, one for every variable of the constraint. This worst-case value for the constraint check occurs when the constraint check succeeds and when it fails due to the last variable in the scope of the constraint. In the case of an early failure, the number of comparisons of vvps will be less than in the case of success. Consequently, the constraint check will be less expensive than in the worst case. Thus, this approach to measure effort accurately reflects the cost of constraint checks in non-binary CSPs.

Summary

In this chapter, we reviewed the definition of a Constraint Satisfaction Problem, its characteristics, and how to solve it with backtrack search. We also reviewed those definitions of interchangeability relevant to our work. We discussed static and dynamic bundling, which result from interleaving search, statitically or dynamically, with the detection and use of NI-sets. Finally, we discussed non-binary CSPs, which are CSPs with one or more constraints of arity larger than 2, and proposed a strategy for performing forward checking in this context and a convention for measuring for measuring, in practice, the cost of constraint checks. It is important to remember that we denote by FC the process of solving a CSP with backtrack search with forward checking, and by DynBndl the same process enhanced with dynamic bundling.

Chapter 3

Bundling Non-Binary CSPs

In this chapter, we first discuss a technique for computing neighborhood interchangeability in non-binary CSPs. Then, we show how to integrate this technique with search and forward checking, which yields the dynamic bundling algorithm DynBndl. We study the properties of this algorithm, and empirically compare its performance with that of regular backtrack search with forward checking (FC) on randomly generated CSPs.

Appendix A describes the detailed experimental results and Appendix B discusses attempts to improve the implementation the algorithm for computing NI sets in non-binary CSPs. The content of this chapter has partially appeared in [Lal and Choueiry, 2003] and [Lal *et al.*, 2004].

3.1 Neighborhood interchangeability in non-binary CSPs

No technique is reported in the literature for computing the NI-sets of a CSP variable in the presence of non-binary constraints. Recall that these sets determine a partition of the domain of the variable, and their elements are values of the variable that are equivalent.

The idea is to identify the variable-value pairs in the neighborhood of a variable V consistent with each value in D_V. The values with 'the same neighborhood' form an equivalence class. The difficulty with non-binary constraints is that the constraints have different arities and the ‘neighborhoods’ of two values are difficult to compare.

A direct application of Algorithm 1 of Chapter 2 to the non-binary case may yield incorrect results. We now discuss how Algorithm 1 can be used for the non-binary case, and show, with an example, that it may yield errors.

3.1.1 Direct application of Algorithm 1

With binary constraints, it is guaranteed that every variable in NEIGHBORS(V) belongs to a different constraint, whereas this is not the case for non-binary constraints. The DT collects consistent values of V's neighbors for every value in D_V. For a given value a in D_V, we compute the set of values in D_{V_i}, where $V_i \in$ NEIGHBORS(V), that is consistent with $\langle V\ a\rangle$ as follows:

- For every constraint C that applies to V and V_i, select the tuples in C where $\langle V\ a\rangle$. Using the select operator of relational databases, the tuples selected are given by: $\sigma_{V=a}(C)$.
- For every C, compute the values in D_{V_i} that are consistent with $\langle V\ a\rangle$ using the projection operator as follows: $\pi_{V_i}(\sigma_{V=a}(C))$.
- Finally, intersect the consistent values resulting from all constraints such as C (i.e., constraint that apply to V and V_i).

Consider the non-binary CSP shown in Figure 3.1 and the constraint definitions shown in Figure 3.2.

From the definition of constraint C_1, we can see that $\langle V\ x\rangle$ and $\langle V\ y\rangle$ are consistent with unequal sets of tuples and are not interchangeable. We show that a direct application of Algorithm 1 will detect them as interchangeable. Indeed, this algorithm identifies

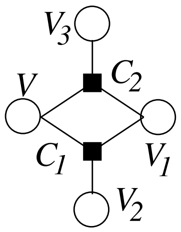

Figure 3.1: CSP.

C_1			C_2		
V	V_1	V_2	V	V_1	V_3
x	a	1	x	a	1
x	b	2	x	b	2
x	c	3	x	c	3
x	d	1	y	a	1
y	a	1	y	b	2
y	b	2	y	c	3
y	c	3			

Figure 3.2: C_1 and C_2.

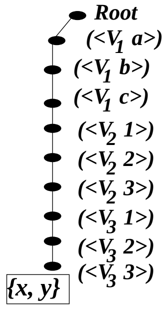

Figure 3.3: DT(V).

the two values x and y of variable V as interchangeable if they are consistent with the same variable-value pairs in the neighborhood of V. In Line 4 of Algorithm 1, this condition would require, for the non-binary case, checking consistency according to *all* the constraints on V simultaneously.

The values of V_1 consistent with $\langle V\ x\rangle$ are {a, b, c, d} given C_1 and {a, b, c} given C_2. Intersecting these two sets, we obtain {a, b, c} as the values of V_1 consistent with $\langle V\ x\rangle$ given C_1 and C_2. Similarly the values for V_2 and V_3 consistent with $\langle V\ x\rangle$ given the same constraints are {1, 2, 3} and {1, 2, 3}, respectively. For $\langle V\ y\rangle$, we get the same set of consistent values. The resulting DT is shown in Figure 3.3. Therefore, Algorithm 1 detects x, y as interchangeable for V when, in fact, they are not. The overlapping scopes of constraints make the direct application of Algorithm 1 for binary CSPs to the non-binary case unfit. Thus, the transition from binary to non-binary CSPs is non-trivial. We need a mechanism that tests the interchangeability in every constraint.

3.1.2 Our approach

Our technique is based on building a separate discrimination tree for *each* of the deg constraints that apply to a variable. We call such a tree a *non-binary discrimination tree* (nb-DT). Below, we introduce two processes:

Process 1 partitions the domain of the variable by building and combining the applicable nb-DTs; and

Process 2 determines the domains of the neighboring variables consistent with each set of the partition.

These two processes allow us to compute the NI-sets of a given variable in presence of any number of binary and non-binary constraints. Further, we use both processes in Section 3.2 for dynamic bundling (i.e., for computing the bundles of the current variable and for forward checking).

3.1.2.1 Process 1: Computing a domain partition

First, an nb-DT is created for each one of the deg constraints on V using Algorithm 2. This algorithm is similar to Algorithm 1 except that it operates only on one constraint and compares each value of V with a tuple of a constraint C.

Input: V, C
current-node $\leftarrow$ *Root*, root of the discrimination tree
$S \leftarrow \textsc{Scope}(C) \setminus \{\text{V}\}$
for *every value* $v \in D_V$ **do**
    for *every tuple* $t \in C \mid \sigma_{V=v}(t)$ *exists* **do**
        if *current-node* has a child node n_t equal to $\pi_S(t)$ **then** *current-node* $\leftarrow n_t$
        else
            Generate n_t a node with $\pi_S(t)$ and make it a child of *current-node*
            current-node $\leftarrow n_t$
        end
    end
    Add v to the annotation of *current-node*
    current-node $\leftarrow$ *Root*
end
Output: *Root*

Algorithm 2: Algorithm for building nb-DT(V, C).

Line 4 of Algorithm 2 replaces Line 3 and 4 of Algorithm 1. (σ and π are the selection and projection operators of relational algebra.)

The worst-case time complexity of Algorithm 2 is linear in the size of the constraint, which depends on the domain size of the variable, the tightness, and the arity of the constraint. Therefore, the cost of building deg such nb-DTs is $O(deg \cdot a^{k+1} \cdot (1-t))$.

Consider the non-binary CSP of Figure 3.4. The constraint definitions for this example are given in Figure 3.5.

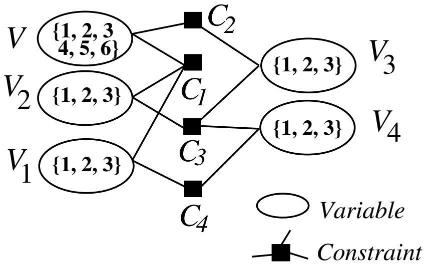

Figure 3.4: CSP.

C_1			C_2		C_3			C_4	
V	V_1	V_2	V	V_3	V_2	V_3	V_4	V_1	V_4
1	1	3	1	3	1	2	1	1	1
1	3	3	2	3	1	2	2	2	2
2	1	3	3	2	2	2	1	3	1
2	3	3	4	2	2	2	2		
3	1	1	4	2	3	1	1		
3	2	2	6	1					
4	1	1							
4	2	2							
5	3	2							
6	3	2							

Figure 3.5: Constraint definitions.

Figures 3.6 and 3.7 show the non-binary discrimination trees (nb-DT) for the constraints incident to V in the example of Figures 3.4 and 3.5. Every node of the nb-DT stores the

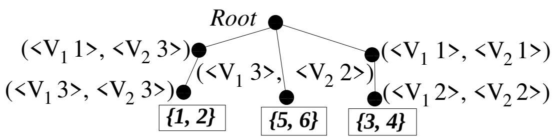

Figure 3.6: nb-DT(V, C_1).

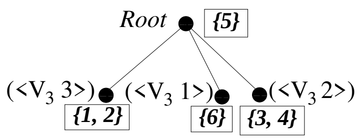

Figure 3.7: nb-DT(V, C_2).

tuple it represents, a list of nodes connected to it, and an annotation that is by default empty. A pointer called the *current-node* is maintained and points to the last node visited by the algorithm. Initially, the *current-node* points to *Root*. The algorithm builds the tree choosing one value v from D_V and processing each tuple of C corresponding to $\langle V\ v\rangle$ as follows:

When the projected tuple matches any of the children of *current-node*, *current-node* moves to the matching node. Otherwise, a new node is created and added to the *current-node*'s list of children, and *current-node* moves to the newly created node. After processing a $v \in D_V$, v is added to the annotation of *current-node* and *current-node* is repositioned at *Root*. Therefore, two nodes are connected if the tuples of each of these nodes lie on the path to a common annotation value.

Second, for each tree, we collect the annotations and the path where they appear. We traverse the tree from the root to each annotation A_i and construct P_i by collecting the nodes on the path. We form a list l_i = (P_i, A_i) of the particular path and the corresponding annotation, and a list $L_j = \{l_i\}$ of these lists for each nb-DT. In the example of Figures 3.6 and 3.7:

1. For the nb-DT of C_1, $L_1 = (l_1, l_2, l_3)$ with:
   - l_1 = ((($\langle V_1\ 1\rangle$, $\langle V_2\ 3\rangle$), ($\langle V_1\ 3\rangle$, $\langle V_2\ 3\rangle$)), $\{1, 2\}$),
   - l_2 = ((($\langle V_1\ 3\rangle$, $\langle V_2\ 2\rangle$)), $\{5, 6\}$),
   - l_3 = ((($\langle V_1\ 1\rangle$, $\langle V_2\ 1\rangle$), ($\langle V_1\ 2\rangle$, $\langle V_2\ 2\rangle$)), $\{3, 4\}$).
2. For the nb-DT of C_2, $L_2 = (l_4, l_5, l_6, l_7)$ with
   - l_4 = ((($\langle V_3\ 3\rangle$)), $\{1, 2\}$),
   - l_5 = ((($\langle V_3\ \texttt{nil}\rangle$)), $\{5\}$),
   - l_6 = ((($\langle V_3\ 2\rangle$)), $\{3, 4\}$),
   - l_7 = ((($\langle V_3\ 2\rangle$)), $\{6\}$).

We collect these lists in $L = (L_1, L_2, \ldots, L_{deg})$.

Third, we compute the partition of D_V by intersecting the annotation A_i from each tree using Algorithm 3 with L and V as input parameters. The worst-case time complexity

Input: L, V
dom-values $\leftarrow$ domain of V
partitioned-domain $\leftarrow$ `nil`
for *every value v remaining in* dom-values **do**
  select-path+annot $\leftarrow$ An l_i from every $L_j \in L$ for which $v \in$ ANNOTATION(l_i)
  annotation $\leftarrow$ Intersect annotations in the *select-path+annot*
  Add *annotation* to *partitioned-domain*
  dom-values $\leftarrow$ *dom-values* $\setminus$ *annotation*
end
Output: *partitioned-domain*

Algorithm 3: Algorithm to intersect annotations.

of this algorithm is $O(deg^2 \cdot a^4)$. For the example of Figure 3.4, the domain of V is partitioned as $\{\{1, 2\}, \{3, 4\}, \{5\}, \{6\}\}$. We denote by E_i an element of this partition, where E_i is a set of equivalent values of V given the constraints that apply to it.

3.1.2.2 Process 2: Computing neighboring values consistent with an E_1

This process computes the values in the neighborhood of V that are consistent with each equivalence class E_i using the nb-DTs built in Process 1. For a given E_i, we identify the paths $\{P_i\}$ in each nb-DT such that $E_i \subseteq A_i$. Then, for each $X \in$ NEIGHBORS(V), we project each path P_i on X. Intersecting the results of the projections yields the subset of D_X that is consistent with E_i. In Section 3.2, we use this information to update D_X by forward checking after assigning E_i to V. The worst-case time complexity of this process is $O(|\text{NEIGHBORS}(V)| \cdot a^{deg})$

3.1.2.3 Avenues for improving performance

In an effort to reduce the overhead for computing bundles, we have included in our implementation a mechanism for automatically 'switching off' some operations for partitioning the domain of a given variable V when it becomes clear that all partitions are necessarily singletons. This happens in two situations.

1. When any nb-DT of a V results in annotations exclusively made of singleton elements (see Algorithm 2). In this case we can safely switch off bundling for building the nb-DTs for the remaining constraints that apply to the variable V.

2. Another case is when the intersection of the annotations returns singletons (see Algorithm 3).

In practice, we implement this switching off mechanism as follows. We force Algorithm 2 not to check for matching children, but to always create new nodes because the information in the nb-DTs is useful for filtering the domains of the future variables.

3.2 Dynamic bundling

Dynamic Bundling (DynBndl) is the process of computing neighborhood interchangeability sets (domain bundles) during the search process (see Section 2.3.3). DynBndl operates by assigning a bundle to V_c and propagating the effect of this decision on the future variables. The bundles of V_c are obtained by applying Process 1 of Section 3.1.2.1 using the constraints on V_c determined by `nFC2`. Each constraint passed to Algorithm 2 is computed using Equation (2.1). The effects of this instantiation are then propagated using Process 2 of Section 3.1.2.2.

Figure 3.8 shows the partially explored search tree by FC for the example in Figures 3.4 and 3.5 with variable ordering $\{V, V_1, V_2, V_3, V_4\}$. Figure 3.9 shows the tree explored by DynBndl. The domain of V is partitioned as discussed in Section 3.1.2 and V is assigned the bundle $\{1, 2\}$. FC propogates this instantiation and the domains of V_1, V_2, and V_3 are set to $\{1,3\}$, $\{3\}$, and $\{3\}$, respectively. Next, the domain partitions of V_1 are computed. We find the two domain values, 1 and 3, to be interchangeable for V_1. V_1 is instantiated with $\{1, 3\}$. On propagating this instantiation, the domain of V_4 becomes $\{1\}$. Next, the search proceeds to instantiate V_2 with the only value in its domain $\{3\}$. This instantiation

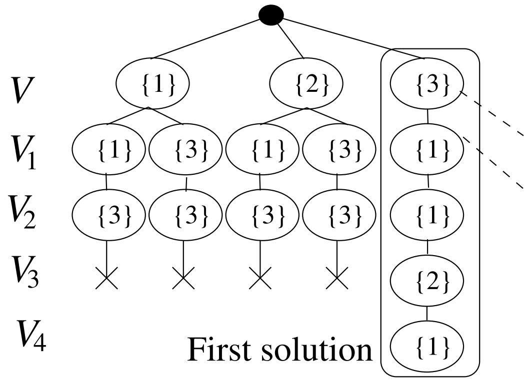

Figure 3.8: Search tree without bundling.

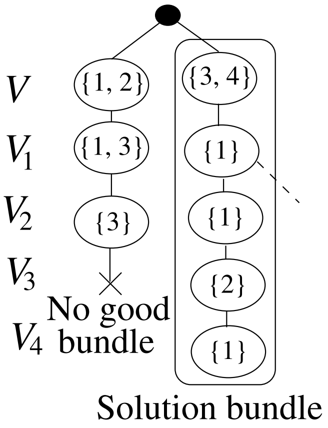

Figure 3.9: Search tree using DynBndl.

results in the annihalation of the domain of V_3, and search backtracks. There are no values remaining in the domains of V_2 and V_1. Hence, the search backtracks to V. Note here that V_1 was assigned a bundle of size 2. By bundling $\{1,\ 3\}$ for V_1 together, bundling saved visiting more nodes of V_2. On instantiating V with $\{3,\ 4\}$, search is able to assign values to the remaining variables of the CSP, yielding a solution as $\{\langle V\ \{3,4\}\rangle$, $\langle V_1\ \{1\}\rangle$, $\langle V_2\ \{1\}\rangle$, $\langle V_3\ \{2\}\rangle$, $\langle V_4\ \{1\}\rangle\}$. This example illustrates two situations that result in performance gain: bundling of no-goods and bundling of solutions.

- *Bundling no-goods*: When DynBndl assigns $\{1,2\}$ to V, $\{1,3\}$ to V_1 and $\{3\}$ to V_2, the domain of V_3 is anihilated after visiting 3 nodes, whereas FC visits 10. The gains due to bundling V multiply those due to bundling of V_1, which illustrates the gains of no-good bundling.

- *Bundling solutions*: When DynBndl next assigns $\{3,4\}$ to V, the path is successful and results in a solution bundle of size 2, while FC yields a single solution. DynBndl visits 8 nodes and yields 2 robust solutions whereas FC visits 15 nodes and finds a single solution.

Under the same variable and value ordering, DynBndl visits no more nodes than FC. Further, when looking for a first solution, any savings of nodes visited by DynBndl can only be explained by the bundling of no-goods.

3.3 Effect of look-ahead strategies on dynamic bundling

In this thesis, we use the look-ahead strategy known as forward checking. Forward checking is a partial look-ahead in the sense that it revises only the domains of those future variables connected to the current varirable. A more aggressive look-ahead strategy such as MAC (Maintaining Arc-Consistency [Sabin and Freuder, 1994]) performs arc-consistency on the entire subproblem induced by the current and future variables. Consequently, the use of a MAC-like algorithm necessarily performs a better filtering of the domains of the future variables. While this may increase the number of constraint checks, it would yield 'fatter' solution bundles (thus improving bundling), and reduce of number of nodes visited during search. However, even with MAC, our technique does not guarantee that the resulting bundling is maximal [Lesaint, 1994]. More generally, dynamic bundling, while it partitions the set of solutions (i.e., every solution appears in exactly one bundle) does not guarantee that the size of the solution bundle is maximal (i.e., the size of the bundle cannot be increased).

3.4 Criteria for evaluating the performance of search

The goal of DynBndl is to generate multiple robust solutions. We measure its effectiveness by computing the size of the first bundle (FBS) and the number of solution bundles (`SB`). To demonstrate that dynamic bundling is not an overkill and does not harm the performance of search, we compare the effectiveness of FC and DynBndl using the following measurements: Constraint Checks (`CC`), Nodes Visited (`NV`), and CPU time.

First Bundle Size (FBS): When DynBndl terminates successfully, it results in the assignment of a bundle (i.e., a set of one or more domain values) to each variable. A solution bundle, the resulting solution, is the set of solutions in the Cartesian product of the bundles assigned to the variables, and its size is the product of the sizes of the assigned bundles. The size of the first solution bundle is denoted as FBS. This measure is useful in comparing the performance of FC and DynBndl when finding only one solution. In the case of FC, each variable is assigned a single value, and FBS is thus one.

Solution Bundle (`SB`): When finding all solutions to a CSP, DynBndl partitions the set of solutions. We denote by `SB` the number of solution bundles found by search. (For the same problem, the lower this number, the more compact is the representation.) In the case of FC, this number is equal to the number of solutions to the CSP.

Constraint Checks (`CC`): We described the mechanism used to measure constraint checks in non-binary CSPs in Section 2.5.4 of Chapter 2.

Nodes Visited (`NV`): The count of node visited is incremented by one every time a a variable is instantiated during search.

CPU time: CPU time allows us to account for the overhead of bundling and estimate the savings due to bundling. We run our experiments on the computers of the Research Computing Facility (RCF) of the Department. The clock resolution of LISP on `prairiefire.unl.edu` and `sandhills.unl.edu` is 10 ms.

The goal is minimize the values of `CC`, `NV`, `SB` and CPU time. When looking for one solution, the goal is to maximize FBS.

3.5 Finding all solutions to a CSP

In this section we theoretically compare the performance of DynBndl and FC when finding all solutions using static variable ordering. We show that DynBndl is guaranteed never to perform worse than FC in terms of the following performance criteria: `NV`, `CC` and `SB`.

Theoretical guarantees of the relative performance of DynBndl and FC under *dynamic* variable ordering are more difficult to determine because the ordering of the variables in DynBndl and FC is no longer guaranteed to be the same. It is easy to see that forward checking may be more effective under one ordering or the other, and thus perform better in one strategy than the other.

3.5.1 Number of nodes visited

Theorem 3.1. *Under the same variable ordering, every node visited by DynBndl is also visited by FC when looking for all solutions to a CSP.*

Sketch of proof. Dynbndl partitions the domain of a variable into equivalence classes. By construction, when one value in an equivalence class is pruned by forward checking in DynBndl, the other values of the class are also removed. In FC, this additional pruning does not take place. Therefore, FC will never remove any more values from the domain of a variable than DynBndl. Therefore, if there is any bundling, DynBndl will visit fewer nodes than FC does. In the worst-case (i.e., no bundling is detected), DynBndl and FC visit exactly the same nodes. □

3.5.2 Number of constraint checks

Theorem 3.2. *Under the same variable ordering, DynBndl never checks more constraints than FC when looking for all solutions to a CSP.*

Sketch of proof. In DynBndl, nb-DT is used to partition the domain of the current variable. The nb-DT also determines the filtered domains of the future variables relative to each value of the current variable. The number of constraint checks done to build the nb-DT is equivalent to that necessary to perform forward checking on each value in the domain of the current variable. FC operates by performing forward checking on one value at a time. When looking for all solutions, all the values for a given variable are eventually visited. Because FC visits at least as many nodes as DynBndl, and at each node it requires exactly the same number of constraint checks, thus FC never performs fewer constraint checks than DynBndl. □

3.5.3 Solution bundles

Theorem 3.3. *When looking for all solutions to a CSP, DynBndl yields a partition of the set of solutions to a CSP.*

Sketch of proof. Depth-first search proceeds systematically through the search space visiting every combination of variable-value pairs and proceeding to the bottom of the tree. It never re-visits the same complete path, and guarantees unique solutions. Similarly DynBndl systematically visits the search space and never re-visits the same complete path. By not re-visiting complete paths DynBndl guarantees that every bundle is unique and that no solution appears in two distinct solution bundles. □

Corollary 3.4. *When looking for all solutions to a CSP, the number of solution bundles found by DynBndl is at most equal to the number of solutions found by FC.*

The above statement follows naturally from Theorem 3.3 and ensures that DynBndl never requires more space than FC.

Theorem 3.5. *The value of* `SB` *for FC is never less than that* `SB` *for DynBndl.*

Sketch of proof. When a CSP is solvable, the solution size of every solution using FC is one. For a solvable CSP, every solution bundle found by DynBndl has size one in the worst case. This can happen when there is no solution bundling (although there could be bundling of no-goods). Thus, the number of solution bundles (found by DynBndl) cannot exceed the number of solutions (found by FC). □

3.6 Finding the first solution to a CSP

The advantages of using dynamic bundling to find all solutions to a CSP were established in Section 3.5. In this section we discuss the the performance of DynBndl in finding one solution. We assume *a static variable ordering* and that FC and DynBndl use the same value ordering.

We first clarify the meaning of the statement: FC and DynBndl using the same value ordering. Let D_V={1, 3, 5, 7} and let, without loss of generality, the value ordering be the one of increasing value. Assume D_V is partitioned by NI into two bundles $\{1, 7\}$ and $\{3, 5\}$. In order to ensure the same ordering as in FC, the values in each bundle must be ordered according to the value ordering (here, ascending order). The first value in a bunlde is considered the representative value of the bundle and the bundles are visited according to position of their representative in the order. Therefore, in our example, we will have $\{1, 7\}$ then $\{3, 5\}$. This simple rule allows us to maintain the same value ordering in the sense that no permutation of domain partitioning will lead to selecting value 3 after value 5, unless 5 is in the equivalence class of 1, which is a value coming earlier in the ordering. Such a value ordering of bundles is implicit in our implementation.

3.6.1 Number of nodes visited

Theorem 3.6. *For the same variable ordering, every node visited by DynBndl is also visited by FC when finding the first solution.*

Sketch of proof. DynBndl partitions the domain of a variable into equivalence classes. By construction, if one value in an equivalence class is pruned by forward checking, the other values of the class are also pruned. Therefore, FC will never prune any more values from the domain of a variable than DynBndl. Hence, if there is any bundling during search, DynBndl will search a smaller tree than FC and visit fewer nodes. In the worst-case (i.e., when all bundles are singletons), DynBndl and FC search the same tree and visit the same nodes. Further, in a backtrack-free search, the number of nodes visited (`NV`) is the same for DynBndl and FC, irrespective of the amount of bundling. □

3.6.2 Number of constraint checks

When finding one solution, we cannot make theoretical guarantees on the relative numbers of constraint checks of DynBndl vs. FC. DynBndl partitions the domain of the current variable using the nb-DT. The nb-DT also provides, as a side effect, the domain of the future variables for each of the values in the domain of the current variable [Beckwith *et al.*, 2001; Choueiry and Davis, 2002]. This result is equivalent to performing constraint checks with future variables for every domain value. FC operates differently, it performs constraint checks for one domain value at a time successively until a domain value does not lead to domain annihilation of a future variable.

Let us consider two scenarios midway while searching for one solution to a CSP. The first scenario is when the value assigned to the current variable leads to a solution without any backtracking. The second is when the first value (e.g., assigned to the current variable fails to yield a solution, forcing search to consider a second value in the domain of the current variable.

In the first scenario, FC performs constraint checks for only a subset of the domain values when instantiating the future variables before reaching a solution. DynBndl, on the other hand, computes domain partitions of each of the future variables and effectively performs constraint checks for each value in the domain of the current variable. In this scenario, DynBndl performs more constraint checks than FC.

The second scenario illustrates a situation where DynBndl executes fewer constraint checks than FC. Let us assume that the two values l, $m \in D_{V_c}$ are interchangeable and consistent with a tleast one value in the domain of a future variable $V_{1f} \in \text{NEIGHBORS}(V_c)$. Let $D_{V_{1f}} = \{x, y, z\}$ such that y is consistent with l and m for V_c and x and z are not. FC executes 3 constraint checks comparing the instantiation of V_c to the values in the domain of V_{1f} and a_i constraint checks exploring the (inconsistent) sub-tree rooted at $\langle V_{1f}\ y \rangle$. See Figure 3.10. FC uncovers the inconsistency of the current path, and repeats the same

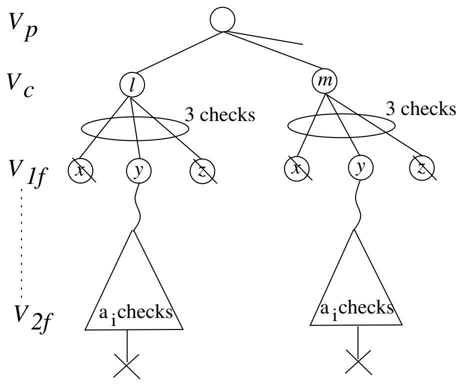

Figure 3.10: Search tree by FC.

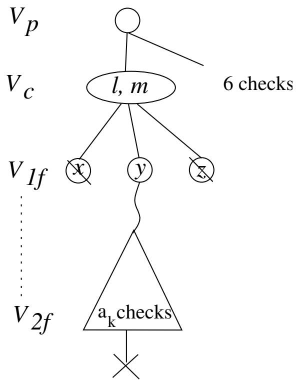

Figure 3.11: Search tree by DynBndl.

procedure when changing the instantiation of V_c from l to m, only to discover that the path is not consistent. This failure prompts search to explore other values for V_c, if there are any, or backtrack to a past variables otherwise. The total number of constraint checks executed by FC in this case is 6+2$\times a_i$ checks. DynBndl proceeds by first partitioning the domain of V_c. It performs 6 constraints checks to determine l and m are interchangeable for V_c. See

Figure 3.11. DynBndl proceeds as FC, but it explores the sub-tree rooted at $\langle V_{1f}\ y\rangle$ only once to discover that the current path is doomed to failure, thus executing 6+a_i constraint checks. Further, if any bundling opportunity arises in the in the sub-tree rooted at $\langle V_{1f}\ y\rangle$, the number of constraints checked in this sub-tree by DynBndl (i.e., a_k) is necessarily less than that under FC (i.e., a_i). Therefore, the savings in constraint checks are due to detecting symmetric domain values, which saves us revisiting the sub-tree rooted at $\langle V_{1f}\ y\rangle$.

From the above two scenarios, it becomes clear that *while DynBndl performs more checks at each node, by saving on the number of nodes visited, it compensates for these extra checks. As a result, at some point, DynBndl performs fewer constraint checks than FC does*. It is difficult to predict the point at which the savings over-compensate the additional effort it depends on the amount of bundling and the number of failed instantiations, which cannot be predicted.

3.6.3 Solution bundle

While FC always returns a unique solution, the first solution bundle returned by DynBndl has one or more solutions. Its size is measured by FBS. Under the same variable and value ordering, the solution returned by FC is always in the solution bundle returned by DynBndl.

3.7 Experiments

The goals of our experiments, in increasing order of importance, are as follows:

1. Assess the importance of solution bundling, which is the main goal of our approach.
2. Assess the overhead due to bundling in terms of effort necessary for computing the domain bundles.
3. Assess the impact of bundling versus non-bundling on the performance of search.

To this end, we compare the values of the following metrics for DynBndl and FC: the number of solutions FBS when finding a first solution, the CPU time, the number of nodes visited `NV`, and the number of constraint checks (`CC`).

When seeking all solutions, DynBndl is guaranteed to visit no more nodes and do no more constraint checks than FC (see Section 3.5). For finding the first solution, DynBndl does not visit more nodes than FC[1] for the same variable ordering (see Theorem 3.6). Under dynamic variable ordering, DynBndl and FC are not comparable either in terms of nodes visited or constraint checks. Their behaviors need to be evaluated experimentally and compared. Our experiments compare the performance of FC and DynBndl under dynamic variable ordering. We use the common heuristic known as the least domain heuristic (respectively, the least number of bundles in a domain).

3.8 Choice of test problems

In this section we discuss the choice of test problems: benchmarks used for symmetric CSPs (which are too specific and not appropriate for demonstrating bundling), real-world applications, and randomly generated problems.

NI aims at detecting equivalent values in the domain of a given CSP variable. It does not intend to uncover permutations of values across variables, which is isomorphic interchangeability and is the focus of most work on symmetry in CSPs. In most published work on symmetric CSPs, the symmetry relations are *declared*, not discovered, and as given as input to the search algorithm. One can expect NI, and its weaker version used in dynamic bundling, to be useful in real-world applications where domain redundancy exists or appears during search. This is not the case of the benchmark problems used for symmetric CSPs, which are not suitable for testing bundling for the following reasons:

[1]Note that the savings by DynBndl in the number of nodes visited when looking for a first solution can only be explained by the bundling of no-goods.

1. Most exhibit only symmetries that are permutations of values over variables.
2. Most have small domains (e.g., Boolean), which are not amenable to bundling.
3. Most are modeled using a unique global constraint of exponential size. Defining the constraint in extension amounts to solving the problem and is likely intractable.
4. Finally, for coloring problems, bundling can be done only in the case of list-coloring problems (typically used to model resource allocation problems). However, such bundling can be easily computed without nb-DTs as shown in [Choueiry and Noubir, 1998].

While looking for (strong or weak) NI sets is cost effective and should be always attempted, no technique can find multiple robust solutions in permutation problems where there are exactly as many variables as there are values.

The primary practical advantage of bundling is the production of robust solutions, where any value in a bundle for a given variable can replace any other value in the bundle, should the former become unavailable or undesirable. The practical usefulness of bundling was established in case-based reasoning [Neagu and Faltings, 2001], nurse scheduling [Weil and Heus, 1998], and databases [Lal and Choueiry, 2004] and Chapter 5. For example, in Chapter 5, we show how dynamic bundling reduces the size of a query result on a real-world database by 54% (yet storing the same information). While we still need to validate our approach on real-world applications, we focus, in this thesis, on introducing the techniques and their implementation and test our algorithms on randomly generated CSPs. Even though such problems lack the redundancy one expects to find in real-world applications (which makes them particularly amenable to bundling), our experiments show that dynamic bundling effectively yields multiple robust solutions.

3.9 Experiment design and set-up

Below, we report results demonstrating the benefits of dynamic bundling on randomly generated problems. We describe a non-binary CSP with the tuple $\langle k, n, a, p_2, c_3, c_4, t\rangle$, where:

- k is constraint arity,
- n is the number of variables,
- a is the domain size,
- p_2 is the constraint ratio of binary constraints, c_3 and c_4 are the number of ternary and quaternary constraint respectively, and
- t the constraint tightness.

We used the random generator provided by [Lal *et al.*, 2003] without enforcing solvability. We ran tests on a wide range of random problems with varying number of variables, domain size, constraint ratio and tightness values. Table 3.1 shows the data sets generated. For brevity, we refer to the constraint ratios using the classification Table 3.2. The tightness values for each of the 16 datasets are listed in Table 3.3. We generate 1,000 instances for every tightness value in all the 16 datasets of Table 3.1. The datasets are designed to study variation in performance over different values of tightness, domain size, constraint ratio, and number of variables. We have included datasets that we expect to be unfavorable to dynamic bundling. For example, the Dataset #4 in Table 3.1 has low domain size and a large number of constraints. In such a dataset, we expect few avenues for bundling, which would illustrate the worst-case behavior for dynamic bundling. We have also included datasets where we expect dynamic bundling to do well, typically datasets with large domain size and low number of constraints (e.g., Dataset #13). In order to ensure uniform server loads across experiments, we solve each problem instance first with DynBndl and, then immediately after that, by FC.

Table 3.1: Datasets of random problems.

Dataset	N	a	**Constraint ratio**		
			p_2	c_3	c_4
1	20	10	0.25	3	2
2	20	10	0.25	6	5
3	20	10	0.4	3	2
4	20	10	0.4	6	5
5	20	15	0.25	3	2
6	20	15	0.25	6	5
7	20	15	0.4	3	2
8	20	15	0.4	6	5
9	30	10	0.25	3	2
10	30	10	0.25	6	5
11	30	10	0.4	3	2
12	30	10	0.4	6	5
13	30	15	0.25	3	2
14	30	15	0.25	6	5
15	30	15	0.4	3	2
16	30	15	0.4	6	5

Table 3.2: Categories of constraint ratios.

p_2	c_3	c_4	Constraint ratio category
0.25	3	2	CR1
0.25	6	5	CR2
0.4	3	2	CR3
0.4	6	5	CR4

3.9.1 Justification of dataset size

A large number of samples ensures reliability of statistical analysis. Though a large sample size is not required in some analysis models, having a large sample size is generally helpful. This is because a larger number of samples better approximates the set of all possibilities called the 'population.' Parametric statistical tests like the t-test are sensitive to the number of samples. However non-parametric tests are comparatively less sensitive to sample size than parametric tests.

In order to determine the sample size, we generated a large number of samples (10,000) for Dataset #5 ($\langle n = 20, a = 15, \mathrm{CR1} \rangle$). We solved these instances and plotted the values of moving averages of the CPU time looking for a sample size where the value of the mean stabilizes. The moving average plot of the CPU time for three tightness values is shown in Figure 3.12. We can see that the value of the mean is 'unstable' from 10 to 100 sample

Table 3.3: Tightness values test for each dataset.

Dataset	Tightness values
1	{0.4000, 0.4500, 0.4750, 0.5000, 0.5250, 0.5500, 0.5550, 0.5750, 0.6000, 0.6500, 0.7000}
2	{0.2750, 0.3000, 0.3500, 0.4000, 0.4625, 0.4875, 0.5000, 0.5125, 0.5250, 0.5375, 0.5500, 0.6000}
3	{0.3250, 0.3625, 0.3750, 0.3875, 0.4000, 0.4125, 0.4250, 0.4750, 0.5500, 0.6500}
4	{0.3000, 0.3250, 0.3500, 0.3625, 0.3750, 0.3875, 0.4000, 0.4250, 0.5000}
5	{0.4500, 0.5000, 0.5500, 0.5750, 0.5875, 0.6000, 0.6125, 0.6250, 0.6500, 0.7000}
6	{0.4500, 0.5000, 0.5500, 0.5750, 0.5875, 0.6000, 0.6125, 0.6250, 0.6500, 0.6750, 0.7500}
7	{0.3500, 0.4000, 0.4250, 0.4375, 0.4500, 0.4625, 0.4750,0.5000,0.5500,0.6000}
8	{0.3500, 0.4000, 0.4500, 0.4750, 0.4875, 0.5000, 0.5125, 0.5250, 0.5500, 0.6000}
9	{0.2500, 0.3000, 0.3500, 0.3750, 0.3875, 0.4000, 0.4125, 0.4250, 0.4500, 0.5000, 0.6000}
10	{0.3000, 0.3500, 0.3750, 0.4000, 0.4125, 0.4500, 0.5000, 0.5500, 0.6000}
11	{0.2000, 0.2250, 0.2500, 0.2750, 0.2875, 0.3000, 0.3125, 0.3250, 0.3500, 0.4000}
12	{0.2000, 0.2500, 0.2625, 0.2750, 0.2875, 0.3000, 0.3125, 0.3500, 0.4000, 0.4500}
13	{0.3500, 0.4000, 0.4500, 0.4750, 0.4875, 0.5000, 0.5250, 0.5500, 0.6000}
14	{0.3500, 0.4000, 0.4500, 0.4750, 0.4850, 0.5000, 0.5250, 0.5750, 0.6500, 0.7000}
15	{0.2500, 0.3000, 0.3250, 0.3500, 0.3650, 0.4250, 0.5000, 0.5500}
16	{0.2500, 0.3000, 0.3250, 0.3350, 0.3500, 0.3650, 0.3750, 0.4000, 0.4500}

points. Therefore, such sample sizes are not representative. The mean starts stabilizing (having less variation on increasing the sample size) near 750 sample points. At 1000 sample points, it is relatively stable with significantly smaller variations than for smaller sample sizes. The mean value at 1000 sample points is close to that at larger samples. From the analysis above, we chose a sample size of 1000 instances for every tightness value in the 16 datasets.

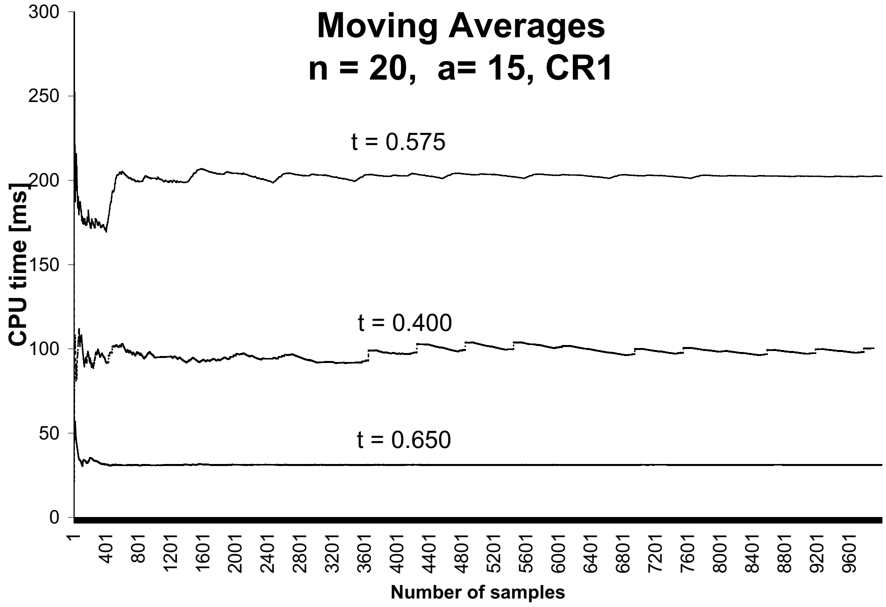

Figure 3.12: Moving average of CPU time for Dataset #1.

3.9.2 Statistical tests

The experiments described above generated a large amount of data. We describe here the steps taken to analyze the data statistically. We first discuss the characteristics of the data, followed by the transformation applied to the data to make it fit the data model of our analysis. We then discuss the statistical tests used to test for difference between the two algorithms being compared (i.e., DynBndl and FC).

The CPU time, NV, and CC of both algorithms have extremely high variances. In spite of the large sample size, the empirical distribution of the data did not approximate a normal distribution. Non-normality was primarily due to the presence of relatively large values in the data (also called outliers). Equal variances and normality of data are two important assumptions in most of the parametric statistical tests to compare the performance of two algorithms. In order to eliminate high variance and the effect of outliers, we applied a log transform to our data. The log-transformed data approximates a normal distribution and fits the data model of our tests.

We used ANOVA (ANalysis of Variance) to study the interaction of the two methods with varying tightness [Rees, 2001]. ANOVA results tell us whether there is a difference in the means of two sets of measurements with the same tightness value that can be attributed to the method used to solve the instances. It allows us to judge whether there is any performance improvement due to DynBndl. The null hypothesis for our analysis was that the difference in the means of CPU time (`NV` and `CC`) of DynBndl and FC is zero when finding the first solution. The ANOVA procedure returns an F-value as the test statistic and the F-value indicates whether we can reject the hypotheses. With an F-value larger than 9.4, we can confidently reject the null hypothesis and conclude that the means due to FC are different from those due to DynBndl.

For every tightness we estimated the difference in mean runtime and the confidence intervals of this difference using the t-distribution. We have a large sample size and hence we also apply the Bennerfoni correction while calculating confidence intervals of the means [Rees, 2001].

We report in our analysis the improvement percentage due to DynBndl over FC for the log-transformed data of CPU time, `NV`, and `CC`. We compute improvement as follows, where I is the improvement and X be one of the three metrics (CPU time, `NV` and `CC`).

$$I(X) = \frac{\mathrm{FC}(X) - \mathrm{DynBndl}(X)}{\mathrm{FC}(X)} = 1 - \frac{\mathrm{DynBndl}(X)}{\mathrm{FC}(X)}. \tag{3.1}$$

The statistical analysis yields the following results:

$$\mathrm{Mean}(ln(\mathrm{DynBndl}(X)) - ln(\mathrm{FC}(X))). \tag{3.2}$$

We can rewrite I as follows:

$$I(X) = 1 - e^{ln(\frac{\mathrm{DynBndl}(X)}{\mathrm{FC}(X)})} = 1 - e^{ln(\mathrm{DynBndl}(X)) - ln(\mathrm{FC}(X))}. \tag{3.3}$$

We report $I \times 100$ as the improvement percentage in our results.

3.10 Results and analysis

In this section, we present the results of our experiments and analyze the effect of varying CSP parameters. First we study the performance of DynBndl in comparison to FC across tightness values with the help of ANOVA results. From there on, we focus on tightness values around the phase-transition region (see Section2.1.4). We study the effect of increasing domain size and number of constraints. Finally, we analyze the results of the experiments to gain more insight into DynBndl.

3.10.1 Analysis with varying tightness

Quite expectedly, the largest FBS occurs at low tightness values, however, DynBndl finds non-singleton solution bundles also well into the area of the phase transition (see Fig. 3.13). Figure 3.13 compares the performance of DynBndl and FC in terms of CPU time, `NV`, `CC`, and FBS with varying tightness in Dataset #7. We choose this dataset to present our analysis because it has a relatively large domain size (which we expect to be favorable for DynBndl) and also relatively high constraint ratio (which we expect to be unfavorable for DynBndl). The results of all remaining datasets are presented in Appendix A. In our analysis, we distinguish the performance at the following three tightness regions in Figure 3.13: low tightness, around the cross-over point, and high tightness.

At small tightness values ($t \leq 0.425$). The benefit of DynBndl here is the large FBS. For example, FBS=33 at t=0.350. In Dataset #13 we get many robust solutions in the first bundle at t=0.35, with FBS=2254.7 (see Table A.8 and Figure A.3). The benefit of bundling no-goods is not yet visible as the first solution (bundle) is found without much backtracking. While the cost of computing the bundles is visible (the

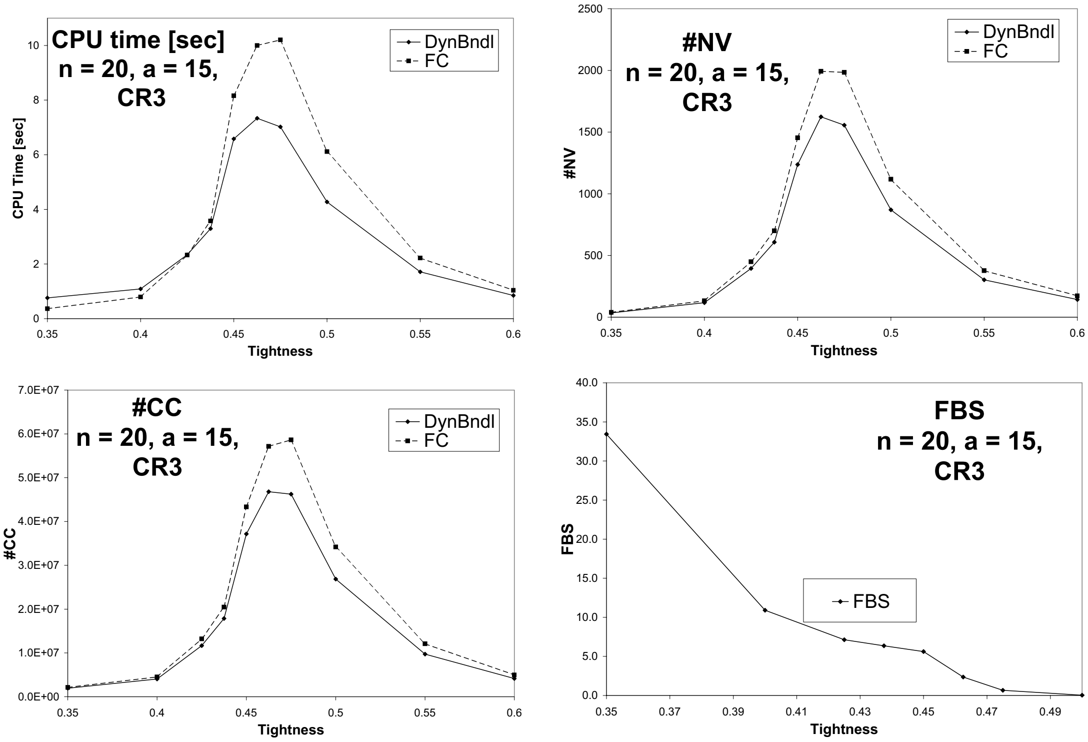

Figure 3.13: The CPU Time, NV, CC and FBS results for Dataset #7.

constraint definitions are large), the overhead can be ignored given the short total time for solving the problems. In Dataset #13, we get 2254.7 solutions from DynBndl at an additional cost of 275ms (see Table A.8 and Figure A.3), which in practice is not a significant additional cost. At t=0.425 (in Figure 3.13) ANOVA shows no significant difference between the CPU time of DynBndl and FC: the overhead of computing the bundles is compensated by the bundling of no-goods.

Around the cross-over point ($0.425 < t \leq 0.500$), DynBndl still returns multiple solutions. For example, FBS=5 at t=0.450 and FBS=2.3 at t=0.462. Furthermore, DynBndl improves the performance of search as bundling of no-goods becomes prevalent: we encounter the maximum amount of savings, in NV, CC, and CPU time. Here, the effort of computing bundles is insignificant compared to the savings due to bundling. ANOVA indicates significant improvement of DynBndl over FC across the entire region. From these results, we can conclude that, in the phase-transition region where solutions are the most costly to find, DynBndl still returns multiple robust solutions while significantly improving the performance of search.

For large tightness values $(0.500 < t)$. Most of the problems at high tightness are unsolvable and the advantage of multiple solutions is not seen here. Forward checking effectively detects that most of the CSPs are not solvable early on in the search process, thus reducing NV and the number of backtracks. The overhead due to bundling becomes apparent, although not alarmingly so. ANOVA indicates that DynBndl and FC are still comparable at t=0.600.

3.10.2 Effect of increasing domain size

Table 3.4 shows, in the phase-transition region, the average improvement of the CPU time and the value of FBS when increasing the domain size. We report the improvement $I(X)$ of a measurement X using Equation (3.3). In summary, increasing domain size or number of variables improves the benefit of DynBndl both in terms of FBS value and savings of CPU time. Increasing the domain size, for the same constraint ratio and tightness, increases

Table 3.4: Increasing a (n=30) around phase transition.

CR	I(CPU) %		FBS	
	a=10	a=15	a=10	a=15
CR1	33.35%	34.32%	5.55	11.93
CR2	28.58%	33.01%	5.01	5.52
CR3	29.82%	31.66%	3.55	4.95
CR4	28.45%	31.65%	1.23	1.43

the chances of bundling and the savings in terms nodes visited.

We know that the cost of computing the bundles increases with the domain size (i.e., $O(deg \cdot a^{k+1} \cdot (1-t))$, see Section 3.1.2.1). However, our experiments show that the additional savings due to the bundling of no-goods exceed the increase in the cost of bundling. Further, better bundling can only increase the value of FBS, which is the product of the size of each bundle in the solution found. This is explained as follows. The search tree with a=15 is larger than that for a=10, and therefore the number of nodes saved by DynBndl is much larger and the increase in savings in CPU time due to visiting these many fewer nodes overshadows the increase in cost of bundling due to increased domain size.

Therefore, with increasing domain size we observe more bundling and get more solutions due to DynBndl at a reduced cost in terms of CPU time. This is especially promising in the context of application to databases where large domain sizes are typical.

3.10.3 Analysis with varying constraint ratio

Table 3.5 shows to the left the effect of increasing the ratio of binary constraints p_2 from 0.25 to 0.40 while keeping constant the number of the non-binary constraints. To the right, it shows the effect of increasing the number of non-binary constraints from $c_3 = 3, c_4 = 2$ to $c_3 = 6, c_4 = 5$ while keeping constant the number of binary constraints. All values reported are for the region of the phase transition. In general, increasing the number of

Table 3.5: Varying constraint ratio around phase transition.

n, a	$c_3 = 3, c_4 = 2$				p_2=0.25			
	I(CPU) %		FBS		I(CPU) %		FBS	
	CR1	CR3	CR1	CR3	CR1	CR2	CR1	CR2
20, 10	27.77	25.95	2.11	0.63	27.77	27.95	2.11	0.55
20, 15	30.07	26.82	4.31	1.74	30.07	25.81	4.31	0.63
30, 10	33.34	29.82	5.55	3.55	33.34	28.57	5.55	5.01
30, 15	34.33	31.65	11.93	4.95	34.33	33.01	11.93	5.51

constraints, everything else remaining equal, reduces the benefit of DynBndl because of an increased probability of breaking bundles. This effect is clearly visible as both the values of FBS and the CPU-time improvement decrease. This is explained by the fact that with increased constraints the chances of symmetries breaking across constraints increase (at the intersection step), leading to thinner bundles for each variable. Thinner bundles decrease the savings by bundling of no-goods and result in smaller FBS values.

3.10.4 Global observations on DynBndl

We use the data in Table 3.6 to provide more insight into DynBndl.

Table 3.6: Effect of tightness and savings in NV on CPU time improvement.

Dataset #5, $n = 20, a = 15, \mathrm{CR1}$		
Tightness	NV**(FC)-**NV**(DynBndl)**	I**(CPU) %**
0.5500	43.63	-7.5
0.5750	123.40	14.3
0.5875	211.39	24.2
0.6000	292.76	29.8
0.6125	289.65	32.6
0.6250	204.53	31.2
0.6500	223.63	32.7

- Observe the savings for t=0.6 and t=0.6125. Even though the savings in NV are comparable (i.e., 292.765 versus 289.65), the improvement in CPU time increases with t (i.e., from 29.8% to 32.6%). This can be explained by the fact that, for the lower tightness, the constraint sizes are larger, which increases the cost of bundling. Therefore, even with the same of savings in NV and equal problem size, the CPU time improvement is larger for larger tightness values.

- At t = 0.55, while DynBndl visits 43 fewer nodes than FC, it takes more time to solve the problem[2]. This gap can be explained by the time spent on computing the bundles, which can reduced by an improved implementation. In summary, there exists a point

[2]Note however that DynBndl returns more solutions than FC.

where the cost of computing the bundles becomes comparable with the increasing savings due to the number of nodes visited.

Table 3.7 shows the improvement of DynBndl in terms of CPU time and the average value of FBS across all datasets in the region of the phase transition. The maximum

Table 3.7: Average improvement in CPU time across datasets.

Dataset	I**(CPU) %**	**FBS**
1	27.77	2.11
2	27.95	0.55
3	25.92	0.63
4	25.94	0.64
5	30.07	4.31
6	25.81	0.63
7	26.82	1.73
8	26.44	1.15
9	33.34	5.55
10	28.57	5.01
11	29.82	3.55
12	28.45	1.22
13	**34.32**	**11.93**
14	33.01	5.51
15	31.65	4.95
16	31.64	1.42

improvement of CPU time is seen for Dataset #13. This is expected because the maximum improvement occurs for the larger values of a (i.e., 15) and the minimum values of constraint ratio (see discussion in Sections 3.9, 3.10.2, and 3.10.3). Conversely, the least improvement is for Datasets #3 and #4, with n=10, a=15, and CR3 and CR4.

Finally, we observe that, across the phase-transition region, FBS is mostly larger than 1, especially for datasets with n=30 or a=15 (i.e., Datasets #5, 7, 8, 9, 10, 11, 12, 13, 14 and 15). It is not as large as for low values of tightness, to the left of phase transition (not shown on Table 3.7). The improvements of CPU time are more significant at the phase-transition area than for low tightness values because of the bundling of no-goods.

Summary

Extending the mechanism for computing neighborhood interchangeability to the context of non-binary constraints is a non-trivial task. We presented an algorithm to compute domain partitions of the domain of variable in non-binary CSPs. We described when it is advisable to switch off bundling and how to do it. We integrated the computation of NI sets in search with forward checking, and theoretically and empirically demonstrated the benefits of DynBndl, the resulting algorithm. In particular, we showed that DynBndl not only finds multiple solutions but also reduces search cost, especially around the phase transition where it matters most.

Chapter 4

Towards Detecting Substitutability

Previous chapters focused on detecting and exploiting values in the domain of a variable that are interchangeable. In this chapter, we consider substitutability, which is a more relaxed version of symmetry, and specifies a one-way interchangeability between two values of a given variable. We describe how to efficiently detect some of these substitutable values by modifying the DT and nb-DT, and how to integrate this approach with dynamic bundling. We then discuss the performance of dynamic bundling using substitutability. It is important to note that the techniques presented here detect only a subset of the possible substitutable values in the domain of a variable but this process is, nevertheless, beneficial.

4.1 Substitutability

Let A and B be two bundles in the domain of a variable V_i. We say that A is substitutable for B when, in *any* solution with $\langle V_i\ B\rangle$, replacing $\langle V_i\ B\rangle$ with $\langle V_i\ A\rangle$ results in a new solution to the CSP. We denote this relationship by $B \mapsto A$. Note that there could be solutions with $\langle V_i\ A\rangle$ that are not consistent solutions when $\langle V_i\ A\rangle$ is replaced with $\langle V_i\ B\rangle$. When $A \mapsto B$ is also true then A and B are interchangeable. We can see that interchangeability is a special case of substitutability. Let S_A be the set of solutions of a CSP that have $\langle V_i\ A\rangle$, and S_B be the set of solutions that have $\langle V_i\ B\rangle$. The above can be restated as follows, where S_i and S_j are two solution bundles to the CSP:

- $\forall\, S_i$=$\{\langle V_1\ X_1\rangle, \ldots, \langle V_i\ B\rangle, \ldots, \langle V_n\ X_n\rangle\} \in S_B \Rightarrow S_j$=$\{\langle V_1\ X_1\rangle, \ldots, \langle V_i\ A\rangle, \ldots, \langle V_n\ X_n\rangle\} \in S_A$.
- Unless A and B are interchangeable, $\exists\, S_i, S_j$ (S_i=$\{\langle V_1\ X_1\rangle, \ldots, \langle V_i\ A\rangle, \ldots, \langle V_n\ X_n\rangle\} \in S_A) \wedge$ (S_j=$\{\langle V_1\ X_1\rangle, \ldots, \langle V_i\ B\rangle, \ldots, \langle V_n\ X_n\rangle\} \notin S_B$).

4.1.1 Using DT to detect substitutability

For a variable V_i and two values $a, b \in D_{V_i}$, we can test whether a is substitutable for b by comparing the sets of values in the neighborhood of V_i that they are consistent with. If the set of consistent values of b is a subset of the consistent values for a, then a is substitutable for b. We propose to use the discrimination tree to detect *some* substitutable values. The bundles in annotations situated *along a path* in the discrimination tree are such that the bundle deeper in the tree is substitutable for the bundle at the shallower level. Consider the example shown in Figure 4.1. Bundle B is consistent with a subset of the values that

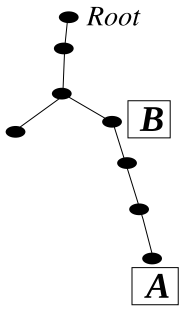

Figure 4.1: DT(V_i) showing substitutable values.

bundle A is consistent with. Hence, whenever $\langle V_i\ B\rangle$ is part of a solution, $\langle V_i\ A\rangle$ can replace $\langle V_i\ B\rangle$ in the same solution (i.e., $B \mapsto A$). The converse may not hold, because a neighbor of V_i may take a value that is in the path between the annotated nodes shown in the tree, and thus is consistent with $\langle V_i\ A\rangle$ but not with $\langle V_i\ B\rangle$.

We illustrate the ideas above using the binary CSP of Figure 4.2. Figure 4.3 shows DT(V_2). We can see that $\langle V_2\ c\rangle$ is consistent with a subset of values that are consistent with

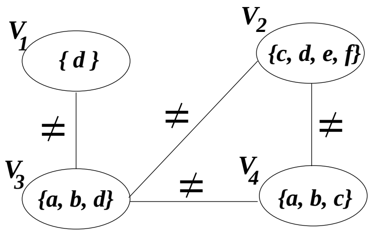

Figure 4.2: A binary CSP.

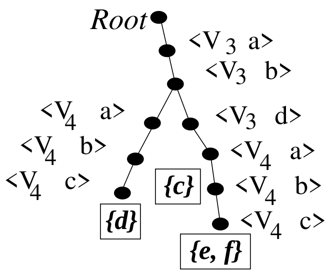

Figure 4.3: DT(V_2).

$\langle V_2\ \{e, f\}\rangle$. Therefore $\{c\} \mapsto \{e, f\}$. Table 4.1 shows how solutions with $\langle V_2\ \{e, f\}\rangle$ can be used to generate solutions with $\langle V_2\ \{c\}\rangle$.

Table 4.1: Solutions using substitutable values.

V_1	V_2	V_3, V_4	Solution
{d}	{e, f}	$\{\langle V_3\ a\rangle, \langle V_4\ b\rangle\}, \{\langle V_3\ b\rangle, \langle V_4\ a\rangle\}$	Yes
{d}	{c}	$\{\langle V_3\ a\rangle, \langle V_4\ b\rangle\}, \{\langle V_3\ b\rangle, \langle V_4\ a\rangle\}$	Yes
{d}	{e, f}	$\{\langle V_3\ a\rangle, \langle V_4\ c\rangle\}, \{\langle V_3\ b\rangle, \langle V_4\ c\rangle\}$	Yes
{d}	{c}	$\{\langle V_3\ a\rangle, \langle V_4\ c\rangle\}, \{\langle V_3\ b\rangle, \langle V_4\ c\rangle\}$	No

4.1.2 DT does not detect all substitutability relations

The DT is not guaranteed to detect all substitutability relations because a bundle B for V_i may be consistent with a subset of the tuples that a bundle A for V_i is consistent with and yet the two bundles may not lie on a same path in the DT. Consider the example DT for a variable V_i with a binary constraint with a variable V_1 shown in Figure 4.4. By the definition of substitutability, $B \mapsto A$ because B is consistent with a subset of the tuples that are consistent with A. However, A and B do not lie on the path from the root of the DT to a common leaf in the DT. Our approach fails to detect that $B \mapsto A$. The ordering of

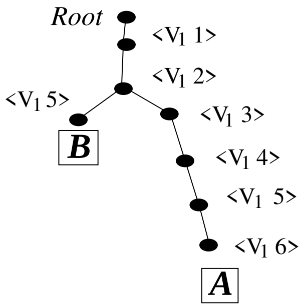

Figure 4.4: A DT(V_i) the illustrates the limits of our approach.

values affects the ability of our approach to detect all substitutability relations

4.1.3 Which bundle to use during search

Consider bundles A and B for a variable V_i such that $B \mapsto A$ (see Figure 4.1). We identify two ways in which to exploit this knowledge in order to reduce the search effort and infer the existence of solutions involving one bundle given the existence of solutions involving the other.

4.1.3.1 Using B as a representative

When $\langle V_i\ B\rangle$ is part of a solution, then we generate additional solutions by simply replacing B with A, and evidently with $A \cup B$. Replacing $\langle V_i\ B\rangle$ with $\langle V_i\ A \cup B\rangle$ leads to fatter solution bundles than those without the use of substitutability. Furthermore, no search needs to be performed on $\langle V_i\ A\rangle$.

However, when $\langle V_i\ B\rangle$ is part of a no-good, we cannot infer whether or not $\langle V_i\ A\rangle$ is also involved in the same no-good. Therefore, we cannot restrict search to B, but must consider $\langle V_i\ A\rangle$ during search.

4.1.3.2 Using A as a representative

When the instantiation $\langle V_i\ A\rangle$ leads to a solution, replacing $\langle V_i\ A\rangle$ with $\langle V_i\ B\rangle$ may or may not lead to a solution. Under these circumstances, we recommend to proceed as follows. Using the solutions involving $\langle V_i\ A\rangle$, we generate a new set of instantiations by replacing $\langle V_i\ A\rangle$ with $\langle V_i\ B\rangle$, and test whether or not the new instantiations are consistent (i.e., are solutions to the CSP). By doing so, we save the search effort, but not the consistency checking effort, on $\langle V_i\ B\rangle$.

When $\langle V_i\ A\rangle$ is involved in a no-good, $\langle V_i\ B\rangle$ is necessarily involved in the same no-good. Therefore, no search with $\langle V_i\ B\rangle$ is needed.

4.1.3.3 Discussion

Given the two above approaches, using A as a representative seems to be a more effective strategy for the following reasons:

- When using A as a representative, we never have to search for $\langle V_i\ B\rangle$, unlike the other approach where a non-solution for B necessitates to conduct search for $\langle V_i\ A\rangle$. Checking if an instantiation is consistent is more efficient than conducting search from scratch.

- The "using A as a representative" approach scales better than the alternative. So far, we have discussed only the case when one value (or a bundle) is substitutable for another one. Imagine a situation where a 'chain' of substitutability relations is identified such as $C \mapsto B \mapsto A$. In this situation, restricting search to A saves on the search effort for both $\langle V_i\ B\rangle$ and $\langle V_i\ C\rangle$. The alternative approach restricts search to C. Under this scenario, when $\langle V_i\ C\rangle$ is involved in a no-good, we still need to run search for $\langle V_i\ A\rangle$ and $\langle V_i\ B\rangle$. The above illustrates how longer substitutability chains amplify the difference between the two approaches.

In summary, in a substitutability relation of the form $B \mapsto A$, using A as a representative is a better approach than the alternative approach.

4.2 Algorithm for detecting substitutability

We present here a new algorithm, Algorithm 4, for binary CSPs that uses DT to detect some substitutable values in addition to NI sets. We denote as ext-DT the extended discrimination tree generated. Algorithm 4 uses A as a representative of two bundles A and B where $B \mapsto A$. Along with the annotation of a bundle (i.e., A of $B \mapsto A$), ext-DT additionally stores a list of values for which the annotation is substitutable, and is denoted by `subs-list` (i.e., B of $B \mapsto A$). ANNOTATION(n) retrieves the annotation of a node n. `subs-list`(n) refers to the list of values for which n is substitutable.

Input: V

current-node $\leftarrow$ Create the root of the discrimination tree
subs-list-work $\leftarrow \{\}$
for *each value* $a_v \in D_V$ **do**
  for *each variable* $V_j \in$ NEIGHBORS(V) **do**
    for *each value* $w \in D_{V_j}$ *consistent with* a_v *for* V **do**
      if *current-node* has a child node n with '$\langle V_j\ w\rangle$' **then**
        current-node $\leftarrow n$
        if $\exists$ *unmarked* ANNOTATION(n) **then**
          subs-list-work $\leftarrow$ *subs-list-work* $\cup$ ANNOTATION(n) $\cup$ `subs-list`(n)
          Mark ANNOTATION(n)
        end
      end
      else
        Generate n a node with '$\langle V_j\ w\rangle$' and make it a child of *current-node*
        current-node $\leftarrow n$
      end
    end
  end
  Add a_v to ANNOTATION(*current-node*)
  subs-list-work $\leftarrow$ *subs-list-work* $\setminus$ (ANNOTATION(*current-node*))
  if current-node *is not a leaf* **then**
    Traverse to a leaf from *current-node*
    Add *subs-list-work* to the `subs-list` in the annotation of the leaf node
    Mark ANNOTATION(*current-node*).
  end
  else
    Add *subs-list-work* to the `subs-list` in the annotation
    Un-mark ANNOTATION(*current-node*)
  end
  current-node $\leftarrow$ root of the discrimination tree
  subs-list-work $\leftarrow \{\}$
end
Output: Root of discrimination tree

Algorithm 4: ext-DT(V) detects interchangeable and some substitutable values for V.

When building the tree with a value $a_v \in D_V$, consider the case when the algorithm encounters a node with an *unmarked* annotation A_j (see Line 8). In this scenario, we are guaranteed that A_j is consistent with a subset of values that a_v is consistent with in NEIGHBORS(V) (i.e., $A_j \mapsto \{a_v\}$). Therefore, Lines 9-10 of Algorithm 4 add the annotation and the `subs-list` of the node with annotation A_j to `subs-list`(a_v).

Line 15 in Algorithm 4 checks whether the annotation where a_v appears is a leaf node in ext-DT. Let us consider the case when a_v appears in a non-leaf annotation, and let A_{leaf} be a leaf annotation reachable from the annotation of a_v. From the discussion in Section 4.1, we conclude that $\{a_v\} \mapsto A_{leaf}$. Therefore, the algorithm *mark*s ANNOTATION(a_v) and adds a_v to `subs-list`(A_{leaf}) in Lines 16–18. When the annotation of a_v is a leaf, then Line 20 of Algorithm 4 sets the annotation to *unmarked*. The output of the algorithm is the DT with the substitutability relations stored in all *unmarked* annotations of the DT. Observe that all *unmarked* annotations will be leaves of the DT.

The algorithm adds a few steps to the original DT building algorithm (Lines 8–10, 14–20). However, these additional steps operate on the DT directly and do not require any additional constraint or consistency checks. The additional operations are characterized as follows:

- Maintaining `subs-list` requires accessing a field in the annotation data structure and appending to a list. Both of these operations are inexpensive in terms of CPU processing (see Line 8–10).

- The other additional operation is of traversing to a leaf of the DT (see Line 16). The algorithm traverses the tree from the current node to any reachable leaf. Therefore, the process simply follows a chain of pointers. This traversal does not require any comparisons and is therefore inexpensive in terms of CPU processing.

In summary, the additional cost of detecting substitutable values is insignificant.

4.3 Extension to non-binary CSPs

Similarly to the binary case, we extend the mechanisms we built to detect NI sets in non-binary CSPs (i.e., the nb-DT and Algorithms 2 and 3 of Chapter 3) to detect some substitutable values along with the NI values. We call the resulting tree ext-nb-DT. As a reminder, in the non-binary case, to compute the NI sets for a given variable, we build an nb-DT for each of the constraints that apply to the variable, then we intersect the annotations to compute the NI sets and intersect the paths leading to these annotations to collect the values of the neighboring variables that are consistent with each NI set.

The change to Algorithm 2 for building nb-DTs into an algorithm for building the ext-nb-DT is similar to what we did above for the binary case. The resulting algorithm is Algorithm 5, which includes the changes to detect some substitutability relations.

4.3.1 Collecting path and annotation information

As we did in Section 3.1.2.1 for nb-DTs, we collect paths and annotations from the ext-nb-DTs to determines NI sets and the corresponding consistent values for the neighboring variables. The only difference in the extension to detect substitutability is that annotations additionally store the `subs-list` and are characterized as marked or unmarked. For clarity, we restate this step here. We traverse the tree from the root to each annotation A_i and construct P_i by collecting the nodes on the path. We form a list l_i= (P_i, A_i) of the particular path and the corresponding annotation, and a list $L_j = \{l_i\}$ of these lists for each ext-nb-DT. L is a list of all L_j's[1] and is an input to the intersection algorithm, Algorithm 6, which we discuss below.

[1]The number of L_j is deg, where deg is the degree of the variable or the number of constraints that apply to the variable.

```
Input: V, C
current-node ← Create the root of the discrimination tree
S ← SCOPE(C) \ {V}
for every value a_v ∈ D_V do
    for every tuple t = (⟨V_i a_i⟩, ⟨V_j a_j⟩, ..., ⟨V_k a_k⟩) ∈ C do
        if σ_{V=a_v}(t) then
            if current-node has a child node n with π_S(t) then
                current-node ← n
                if ∃ unmarked ANNOTATION(n) then
                    subs-list-work ← subs-list-work ∪ ANNOTATION(n) ∪ subs-list(n)
                    Mark ANNOTATION(n)
                end
            end
            else
                Generate n a node with π_S(t) and make it a child of current-node
                current-node ← n
            end
        end
    end
    Add a_v to the ANNOTATION(current-node)
    current-node ← root of the discrimination tree
    subs-list-work ← subs-list-work \ ( ANNOTATION(current-node) )
    if current-node is not a leaf then
        Traverse to a leaf from current-node
        Add subs-list-work to the subs-list of the leaf node
        Mark ANNOTATION(current-node).
    end
    else
        Add subs-list-work to the subs-list in the annotation
        Un-mark ANNOTATION(current-node)
    end
    current-node ← root of the discrimination tree
    subs-list-work ← {}
end
Output: Root of discrimination tree
```

Algorithm 5: ext-nb-DT(V, C), which also detects some substitutable values.

4.3.2 Intersecting the nb-DTs

When $B \mapsto A$ for V, the annotation of A must contain B in its `subs-list` for each of the deg ext-nb-DTs of V. To see if $B \mapsto A$ holds across the ext-nb-DTs of the variable V, we modify the process of intersecting annotations and detect some substitutable values.

The intersection process when detecting NI sets in non-binary CSPs (see Algorithm 3) works by processing each value, a_v, in the domain of variable V as follows:

1. It collects all annotations with the value a_v and intersects them.

2. The result of the intersection is a partition of the domain. The intersection process subtracts the partition from the values in the domain of the variable.

The process moves to the next value in the remaining domain, and repeats the above steps until all values in the domain are processed.

We now present an algorithm, Algorithm 6, to intersect results from individual ext-nb-DTs to include substitutable values. The algorithm has two steps. In the first step (Lines 3–10), the algorithm considers only those domain values that have an *unmarked* annotation (i.e., a leaf annotation) in at-least one ext-nb-DT (see Line 5) in order to not miss detecting the longest substitutability relationship. For example, let $C \mapsto B \mapsto A$ in all the ext-nb-DTs of a variable V and let B be the first bundle from D_V in the value ordering. If we ignore Line 5 in the algorithm, the algorithm will process B, even though all annotations of B are marked, and will detect that $C \mapsto B$. By doing so, it misses detecting the longer relationship $C \mapsto B \mapsto A$. Line 5 defers detecting such chains to the second step (Lines 11–17) of the algorithm. The second step of the algorithm (Lines 11–17) processes the skipped domain values to detect any remaining NI or substitutability relations.

The worst-case time complexity of the algorithm remains the same as Algorithm 3 (i.e., $O(deg^2 \cdot a^4)$) because the additional intersections done for `subs-list` are compensated

```
Input: L, V
1  dom-values ← domain of V
2  partitioned-domain ← nil
3  for every value a_v remaining in dom-values do
4      select-path+annot ← An l_i from every L_j ∈ L for which a_v ∈ ANNOTATION(l_i)
5      if Any Annotation in select-path+annot is unmarked then
6          annotation ← Intersect annotations in the select-path+annot
7          subs-list-intersect ← Intersect the subs-list's from each annotation
8          Make subs-list-intersect the subs-list of annotation
9          Add annotation to partitioned-domain
10         dom-values ← dom-values \ {annotation ∪ subs-list}
       end
   end
11 for every value a_v remaining in dom-values do
12     select-path+annot ← An l_i from every L_j ∈ L for which a_v ∈ ANNOTATION(l_i)
13     subs-list-intersect ← Intersect the subs-list's from each annotation
14     Make subs-list-intersect the subs-list of annotation
15     annotation ← Intersect annotations in the select-path+annot
16     Add annotation to partitioned-domain
17     dom-values ← dom-values \ {annotation ∪ subs-list}
   end
Output: partitioned-domain
```

Algorithm 6: Algorithm to intersect annotations and `subs-list`.

by the reduction in the *dom-vals* list (Lines 10 and 17). We denote an annotation and its `subs-list` as follows {annotation-bundle values}[`subs-list`].

Let us work through the algorithm with an example in which a variable V is constrained by two constraints C_1 and C_2. Consider the scenario shown in Figure 4.5. Let us assume the following value ordering {v, x, y, z} of the domain of V. Note that any of these values could represent a bundle. The value v has no unmarked annotations and, hence, is not processed in the first step. Next when working with value x, the algorithm performs:

$$\{x, y, z\}[\{v\}] \cap \{x, y\}[\{v\}] = \{x, y\}[\{v\}]$$

We delete the values $\{x, y, v\}$ from the values remaining in D_V. Figure 4.6 shows only

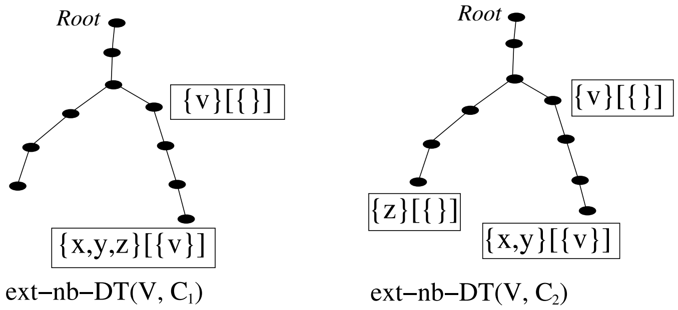

Figure 4.5: A scenario.

the annotations of unprocessed domain values after the first step. Now, when working with

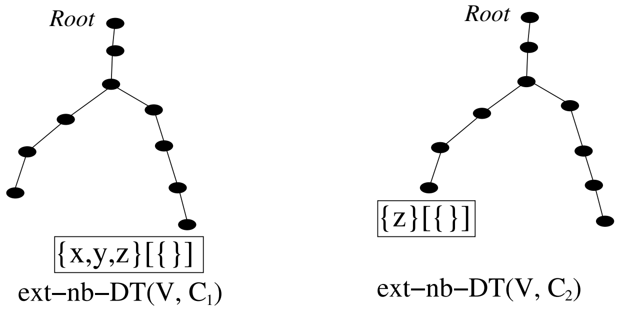

Figure 4.6: After processing.

value z, the algorithm performs:

$$\{x, y, z\}[\{v\}] \cap \{z\}[\{\}] = \{z\}[\{\}]$$

Thus, all domain values are processed and the substitutable values detectable by our approach are indeed detected.

4.4 Improving search performance using substitutability

Search using substitutability proceeds using the same two processes used for DynBndl described in Sections 3.1.2.1 and 3.1.2.2. These two processes use Algorithm 5 instead of Algorithm 2, and Algorithm 6 instead of Algorithm 3. When search finds a solution, it attempts to generate additional solutions by substituting values from the `subs-list` in the current assignments. Search checks whether this substitution breaks any constraints in the CSP. When no constraints are broken, the new assignment is output as a solution. No search is performed using values in the `subs-list`.

Exploiting substitutability during search improves performance by the same mechanisms as interchangeability, namely, bundling of no-goods and bundling of solutions. We compare the performance of dynamic bundling and that of substitutability with dynamic bundling. For the following results, we assume a static variable ordering and the same value ordering for the two approaches.

Theorem 4.1. *Given a static variable ordering, search with substitutability and dynamic bundling never visits more nodes than dynamic bundling.*

Sketch of proof. The worst-case scenario for substitutability is when there are no substitutable values. In this case, the process reduces to search with dynamic bundling. When substitutable values are detected, they reduce the number of nodes visited irrespective of whether or not the bundle to which search is restricted leads to a solution. Therefore, search detecting substitutability in addition to interchangeability never visits more nodes than dynamic bundling alone. □

Theorem 4.2. *Given a static variable ordering, search with substitutability and dynamic bundling never performs more constraint checks than dynamic bundling.*

Sketch of proof. As we note in the algorithm to build a ext-DT (see Section 4.2), the additional steps introduced act on discrimination tree structure. Therefore, given an ext-DT, there is no need to perform any additional consistency checking to detect substitutable values. Hence, search using dynamic bundling with substitutability will never check more constraints than search with dynamic bundling alone. □

Theorem 4.3. *Given a static variable ordering, search with substitutability and dynamic bundling finds a bundle with FBS at least as large as that found by dynamic bundling.*

Sketch of proof. The worst case for our approach occurs when no substitutable values are detected and the solutions are not fatter than those found by dynamic bundling. When substitutable values are present, solutions can be more representative than those found by dynamic bundling alone. □

The algorithms presented in this chapter generalize their predecessors by performing cheap data-structure management to detect some substitutability relations apparent in a discrimination tree. One additional computational step consists in verifying the consistency of solutions obtained by simple replacement of a bundle by another, which is an operation that is not guaranteed to succeed but is not costly to execute. Therefore, the use of substitutability will likely improve the CPU time performance for a majority of the CSPs in practice, but this hypothesis remains to be empirically tested and evaluated.

Summary

In this chapter, we discussed a special form of interchangeability, called substitutability. We showed how algorithms to detect NI sets can be extended to also detect some substitutable values. We presented algorithms for both binary and non-binary CSPs. We discussed integrating substitutability detection with dynamic bundling and theoretically compared its performance to that of dynamic bundling in terms of `NV`, `CC` and FBS. The experimental evaluation and validation of the above ideas still need to be conducted.

Chapter 5

Dynamic Bundling for Databases

This chapter discusses the extension of dynamic bundling to database algorithms. This extension is not a straightforward application of the mechanisms developed in Chapter 3 but requires lifting the concepts and designing new algorithms to apply these concepts in this new context. We first review some concepts of database relevant to the task at hand and the challenges of adapting CSP techniques to database engines. Then we introduce a new sorting-based bundling algorithm suitable for databases, and show its applicability to materialized views. We also identify areas of applicability in databases to be investigated in future research, and briefly discuss each one.

The content of this chapter has partially appeared in [Lal and Choueiry, 2004].

5.1 Introduction to database concepts

This section gives an overview of the concepts in database literature relevant to our task. We first give an introduction to join algorithms followed by an introduction to Materialized views.

5.1.1 Join algorithms

The join operator in relational algebra takes as argument two relations and a condition, known as the *join condition*, that constrains any two or more attributes, one from each of the two input relations. The notation of a join is $\mathtt{R}\bowtie_{x\theta y}\mathtt{S}$, where R and S are two relations, x and y are attributes from R and S respectively, and the join condition θ is a comparison operator (e.g., $=$, $\geq$, $\leq$, and $\neq$). Equality is the most commonly used join condition, and yields the *equi-join*, which requires that the values of two distinct attributes be the equal. A *natural join* is a special case of an equi-join that requires that the attributes themselves be the same. The join operation is among the most I/O-intensive operators in relational algebra because it may require multiple scans over the two input relations and also because the size of the result can be as large as the product of the sizes of these relations.

Join algorithms can be classified into three categories: hash-based, sort-based, and nested-loop algorithms. All these algorithms attempt to optimize the join by minimizing the number of times relations are scanned. Hash-based algorithms use hash-tables to partition relations according to the values of an attribute, and then join the partitions corresponding to the same values. The sort-based approach partitions relations by sorting them on the attributes involved in the join condition. Thanks to sorting, each tuple in a relation is compared with tuples of the other relation lying within a fixed range of values, which are significantly fewer than all possible tuples. Sorting reduces the number of scans of both relations and speeds up join processing. Nested-loop algorithms are used when relations fit in memory or when no adequate hashing function or useful sorting order is available. None of these techniques attempts to compact query results, although this can be beneficial given the large size of join results. The reduction of the number of I/O operations during query evaluation is a key factor in determining the efficiency of a database. Extensive research is devoted to the development of query-evaluation plans and evaluation algorithms that minimize the number of I/O operations.

Our new join algorithm, described in Section 5.5, adopts the principle of the Progressive Merge Join (PMJ) algorithm of Dittrich et al. [2003]. PMJ is a join algorithm that produces query results early, and hence can provide valuable information to the query-size estimator. These are the working conditions that we are targeting. PMJ is an instance of the sort-merge join algorithms, which have two phases: the sorting phase and the merging phase. We first describe sort-merge join algorithms in general, then discuss PMJ.

5.1.1.1 Sort-merge join algorithms

In the sorting phase of a sort-merge algorithm for computing the join of two relations, `R1` and `R2`, the memory of size M pages is first filled with pages of `R1`. These loaded pages are then sorted on the join-condition attributes, and stored back to disk as sub-lists, or *runs*, of the relations. When `R1` has N pages, $\frac{N}{M}$ runs are generated. This process is repeated for `R2`, which we assume to have the same size as `R1`. At the end of the sorting phase, we have produced sorted runs of `R1` and `R2`. Now, the merging phase can start.

We first consider that $M \geq 2 \times \frac{N}{M}$. Now $\frac{M^2}{2 \times N}$ pages from each of the $\frac{N}{M}$ runs of `R1` are loaded into memory, and the same is done for `R2`. The smallest unprocessed tuples from the pages of `R1` and `R2`, respectively, are tested for the join condition. The tuples that satisfy the join condition are joined, and the result written as output. A page is exhausted when all its tuples have been processed. When a page is exhausted, a page from the same run is loaded and brought in. When $M < 2 \times \frac{N}{M}$, multiple merge phases are necessary. Each intermediate merging phase produces longer, but fewer, sorted runs. This process of generating longer but fewer runs continues until the number of runs of the two relations is equal to the number of pages that can fit in memory.

Importantly, this sort-merge algorithm is a *blocking* algorithm in the sense that the first results come only after the sorting phase is completed.

5.1.1.2 Progressive Merge Join (PMJ)

PMJ delivers results early by joining relations already during the sorting phase [Dittrich *et al.*, 2003]. Indeed, during the sorting phase, pages from both relations are read into memory, sorted, and joined to produce early results. Because PMJ produces results early, it is a *non-blocking*, or a pipelined, version of the sort-merge join algorithm. The number of runs generated for each relation is more than that by a general sort-merge algorithm and is given by $\frac{M^2}{4 \times N}$. The merging phase is similar to that of a sort-merge algorithm, except that PMJ ensures that pages of `R1` and `R2` from the same run are not joined again as they have already produced their results in the sorting phase. The memory requirements of PMJ are more than those of a sort-merge algorithm because the number of runs generated during the sorting phase is double that of a sort-merge algorithm. The number of runs generated doubles because the memory is shared by both relations. Because of the increased number of runs, the chances of multiple merging phases taking place increase. The production of early results causes the results of PMJ to be unsorted. However, the unsorted results allow for more accurate estimation of the result size, which is an important feature.

5.1.2 Introduction to Views

In order to introduce views in databases, we use the particularly compact and effective introduction to views by Gupta and Mumick [1995], from which most of the following is taken verbatim.

What is a view? A view is a derived relation defined in terms of base (stored) relations. A view thus defines a function from a set of base tables to a derived table; this function is typically recomputed every time a view is referenced.

What is a materialized view? A view can be materialized by storing the tuples of the view in the database. Index structures can be built on the materialized view. Consequently, database accesses to the materialized view can be much faster than recomputing the view. A materialized view is thus like a cache–a copy of the data that can be accessed quickly.

Why use materialized views? Like a cache, a materialized view provides fast access to data; the speed difference may be critical in applications where the query rate is high and the views are complex so that it is not possible to recompute the view for every query. Materialized views are useful in applications such as data warehousing, replication servers, data visualization, and mobile systems. Integrity constraint checking and query optimization can also benefit from materialized views.

What is view maintenance? Just as a cache gets dirty when the data from which it is copied is updated, a materialized view gets dirty whenever the underlying base relations are modified. The process of updating a materialized view in response to changes to the underlying data is called view maintenance.

What is incremental view maintenance? In most cases it is wasteful to maintain a view by recomputing it from scratch. Often it is cheaper to compute the changes in the view to update its materialization. Algorithms that compute changes to a view in response to changes to the base relations are called incremental view maintenance algorithms.

5.2 CSP techniques for join computation

Although the computational problems studied in Constraint Processing and in databases are incredibly similar, few researchers address the overlap of these two areas. Exceptions include the work by Dechter and Pearl [1989; 1990], Bayardo [1996], and Miranker et al. [1997]. Table 5.1 summarizes our understanding of how the terminology used in databases maps into that used in Constraint Processing: In this section we present the motivation

Table 5.1: Terminology mapping.

DB terminology	CSP terminology
Table, relation	Constraint (which we call relational constraint)
Join condition	Constraint (which we call join-condition constraint)
Relation arity	Constraint arity
Attribute	CSP variable
Value of an attribute	Value of a CSP variable
Domain of an attribute	Domain of a CSP variable
Tuple in a table	Tuple in a constraint Tuple allowed by a constraint Tuple consistent with a constraint
Constraint relation (in Constraint Databases)	Constraint of linear (in)quality
A sequence of natural joins	All the solutions of the corresponding CSP

behind applying CSP concepts to join computation and the challenges involved.

5.2.1 Motivation

Dynamic bundling, discussed in Chapter 3, produces compact solutions to a CSP by detecting symmetries in the definition of a constraint. In this chapter, we extend this concept to the computation of a join query on a database. The idea is to detect symmetries in the relations on which the join operator is defined, and compute a compacted join result. Each compacted tuple in the resulting table yields a set of tuples by making the Cartesian product of the values of the tuple's attributes. The goal of dynamic bundling in CSPs is the production of multiple, robust solutions at an affordable cost[1]. The goal of extending this mechanism to the computation of a join query is to:

1. reduce the number of I/O operations, and

2. produce a result that can be used in data analysis and data mining.

[1]We showed, in Chapter 3, that the computational cost was in fact significantly reduced.

While the second goal is outside the scope of this thesis, we achieve the first one as follows:

- We design an algorithm for detecting symmetries in the relations on which the join query is defined (see Section 5.4). This algorithm does not use any external data structure such as the discrimination tree, but is entirely based on a sorting mechanism.
- Then, we design a join algorithm that exploits the symmetries detected by the above bundling algorithm (see Section 5.5).

We project two other important uses of our technique, namely:

Improving query-size estimation: Indeed, the fact that the size of the compacted tuples produced by our technique may be large is an indicator of high redundancy in the resulting join relation. This information can be used to boost the estimate of query-result size, which is particularly relevant to query planning.

Supporting data analysis and mining: The fact that these compacted results group similar solutions uncover semantic information that hold among the data items that is useful for data analysis. Let us illustrate this advantage with the following two-relation scenario. The relation

`Customer_List(Custid, Age, Gender)`

stores demographic information of all customers. The relation

`Customer_Choice(Custid, Favorite_Product)`

stores choices of some customers from an online survey. The query to find the favorite products by age:

```
SELECT Customer_List.Custid, Favorite_Product, Age
FROM   Customer_Choice, Customer_List
WHERE  Customer_Choice.Custid = Cust_List.Custid
```

Our techniques will produce results where customers with same product and age groups are bundled up together. This additional information can be used for data mining and in packages for data analysis.

5.2.2 Challenges

The challenge lies in adapting CSP techniques of to fit the Database paradigm. We list here the primary differences between the two fields, and discuss our solutions to porting CSP algorithms to databases.

5.2.2.1 Number of I/O operations and memory usage

All database operations are optimized towards minimizing the number of I/O operations performed as these are the most time-consuming operations. Typically, in Relational databases, the processed data cannot large cannot be loaded at the same into the main memory. All join algorithms are sensitive to this fact. The number of CPU operations performed is not considered as a significant factor. The CSP community, on the other hand, focuses on minimizing the number of CPU operations, often by introducing memory-intensive data-structures. Typically, in CSP algorithms, the main memory is not a bottleneck.

The algorithm for computing domain partitions relies on the nb-DT, which can, in the worst case, be as large as the relation (see Section 3.1). In the database context, we cannot afford such a large data structure. For this reason, our algorithm for computing the bundles of the input relations to the join query heavily uses sorting and requires data structures that are significantly lighter than the nb-DT. Further, our dynamic bundling algorithm assumes that the constraints are available in memory (see Section 3.2). In the context of databases, we are able to load only a sub-set of a relation in memory at a given time. Consequently, we modify the dynamic bundling algorithm to allow it to operate on the portions of the relations loaded in the main memory.

5.2.2.2 Fit in the iterator/lazy model of database engines

As we discuss in Section 5.3, computing the join query of two relations corresponds to finding all the solutions of the CSP that models the join query. In database, the interface of the join algorithm to the query processor is an iterator. The query processor treats the join algorithm as a black-box that has the two following interface methods:

- `nextTuple()`: returns the next tuple in the result of the join.
- `hasMoreTuples()`: returns a boolean value indicating whether there are any more tuples remaining in the join.

These two methods indicate that the algorithm must be able to remember its state after one tuple has been computed and be able to resume computation of the join when the `nextTuple()` or `hasMoreTuples()` is invoked. Our dynamic bundling algorithm does not directly fit in this interface. Our algorithms described below provide an iterator interface.

5.3 Modeling a join query as a CSP

In this section, we show how to model a join query as a CSP using the following join query as a running example:

```
SELECT R1.A, R1.B, R1.C
FROM   R1, R2
WHERE  R1.A = R2.A AND R1.B = R2.B AND R1.C = R2.C
```

We identify one straightforward representation of this query as a CSP where each attribute is mapped into a CSP variable. Because this particular example is an equi-join, we also identify an alternative CSP representation. We show in Figures 5.1 and 5.2 the constraint networks of these two representations, and specify the corresponding CSPs $\mathcal{P} = \{\mathcal{V}, \mathcal{D}, \mathcal{C}\}$ as follows:

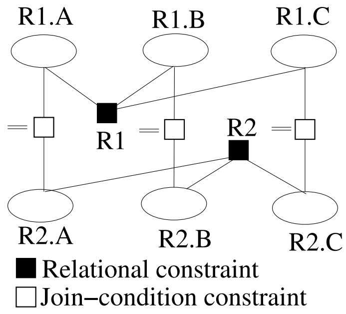

Figure 5.1: A join as a CSP.

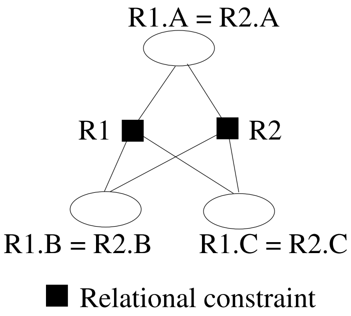

Figure 5.2: An equi-join as a CSP.

1. *The attributes as CSP variables.* In the first mapping, $\mathcal{V}$ is the set of attributes in the join query. There are 6 variables in our example, which are the attributes `R1.A`, `R2.A`, `R1.B`, `R2.B`, `R1.C`, and `R2.C` (see Figure 5.1). In the second mapping, which is an equi-join query, the attributes joined by an equality constraint can be represented by a unique CSP variable. The CSP representing the query consists of only the 3 variables: `R1.A = R2.A`, `R1.B =R2.B`, and `R1.C = R2.C` (see Figure 5.2). When the query lists the two equated attributes in its `SELECT` clause, the CSP variable is simply repeated in the output.

2. *The attribute values as variable domains.* $\mathcal{D}$ is the set of the domains of the variables. Under the first mapping, the domain of the variables is the set of values that the attribute takes in the relation. In the second mapping, the domain of the CSP variable representing the equated attributes is the union of the set of values that the equated attributes take in their respective relations.

3. *The relations and join conditions as CSP constraints.* $\mathcal{C}$ is the set of constraints of the CSP. These constraints originate from two sources, namely: the relations to be joined and the join conditions. The relations to be joined directly map to CSP constraints

> that are expressed extensionally. We call these constraints *relational constraints*. The join conditions map directly to CSP constraints expressed intensionally, which we call *join-condition constraints*. In the first mapping, there are 3 equality constraints due to the join conditions in the query. In the second mapping, where the equated attributes are represented by a unique variable, the join condition is implicit in the CSP representation and does not need to be expressed.

Our algorithm for bundling non-binary CSPs requires that constraints be enumerated (see Section 3.1). However, for computing interchangeability in the database scenario, we do not enumerate the join-condition constraints or store them explicitly. Instead, we proceed as follows. When joining two relations specified in extension, the resulting tuple is checked for consistency with the join-conditions specified in intension as this tuple is being built up. When the values in the partially built tuple are not consistent with a join-condition constraint, the tuple is discarded, as we explain in Section 5.5.1. This operation is possible because we are guaranteed that all the CSP variables appear in at least one constraint defined in extension, and thus all the join-condition constraints will necessarily be checked for consistency.

5.4 Sort-based bundling

This section describes the computation of interchangeable values (i.e., a bundle) of an attribute in a relation. Since our join algorithm is a sort-merge algorithm, the relations must first be sorted. Thus, we need to select the order of the attributes for sorting the relations. This order is necessarily static because we cannot afford to re-sort relations during processing. In terms of CSPs, this corresponds to a static ordering of the variables. We first describe our attribute ordering heuristic then the technique for computing interchangeability.

5.4.1 Heuristic for variable ordering

Let $\mathcal{V}$ be the set of variables in the CSP representing a query. We denote by $\mathcal{V}_q$ a first-in first-out queue of these ordered variables. We describe here how we build this queue. First, we initialize $\mathcal{V}_q$ to a queue with one arbitrarily chosen variable[2]. We denote by $\mathcal{V}_u$ the set of unordered variables (i.e., $\mathcal{V}_u = \mathcal{V} \setminus \mathcal{V}_q$). Let V_c be the last variable added to $\mathcal{V}_q$. The next variable V_n in the order is chosen from $\mathcal{V}_u$ as follows:

1. Consider the variables $\{V_i\} \subseteq \mathcal{V}_u$ such that V_i is linked to V_c with a join-condition constraint C_i. V_n is selected as the variable for which $|\mathcal{V}_u \cap scope(C_i)|$ is the smallest.
2. V_n is selected as any variable from the same relation as V_c.
3. V_n is selected arbitrarily from $\mathcal{V}_u$.

If no variables satisfy a rule in the sequence above, the next rule in sequence is applied to $\mathcal{V}_u$. Ties are broken by the the next rule in sequence. V_n is removed from $\mathcal{V}_u$ and added to $\mathcal{V}_q$. The process is repeated until $\mathcal{V}_u$ is empty. The goal of this ordering is to allow the checking of join-condition constraints as early as possible. For the example of Figure 5.1, one possible ordering is the sequence `R1.A`, `R2.A`, `R1.B`, `R2.B`, `R1.C`, and `R2.C`.

Note that the ordering of the variables affects the size of the generated bundles and that different ordering heuristics still need to be investigated.

5.4.2 The principle

Given the queue $\mathcal{V}_q$ of ordered variables, we build the bundles dynamically while joining the tuples loaded in memory. Variables in the queue are considered in sequence. The variable under consideration is called the current variable V_c, the set of previous ones is

[2]One can elaborate heuristics for choosing the first variable. One possibility is to exploit the meta-data maintained by the DBMS such as the number of unique values of an attribute. Other heuristics may choose first the attribute that participates in the largest number of constraints. The design and evaluation of such heuristics still needs to be investigated.

denoted by $\mathcal{V}_p$, and the set of remaining ones by $\mathcal{V}_f$. $\mathcal{V}_f$ is initialized to $\mathcal{V}_q$, keeping the same order of variables, and $\mathcal{V}_p$ is set to nil. First, we find a bundle for V_c as described in Section 5.4.4. Then, we determine the subset of values in the bundle that is consistent with at least one bundle from each of the variables in $\mathcal{V}_f$ with a join-condition constraint with V_c (see Algorithm 8). If such a subset is not empty, we assign it to V_c. In terms of CSPs, this corresponds to instantiating V_c. We move V_c to $\mathcal{V}_p$, and a new V_c is chosen as the first variable in $\mathcal{V}_f$. Otherwise, if the consistent subset for V_c is empty, we compute the next bundle of V_c from the remaining tuples and repeat the above operation. We continue this process until all the variables are instantiated. Then, we output these instantiations as the next nested tuple of the join.

Consider the scenario where a next bundle for V_c, an attribute of relation `R1`, needs to be computed during a sequence of instantiations (see Figure 5.3). The bundle depends on the

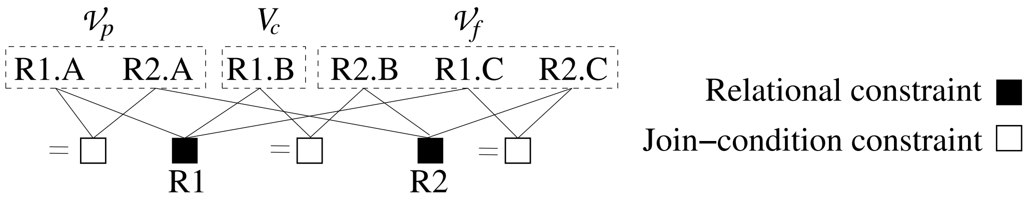

Figure 5.3: Instantiation sequence.

instantiation of variables from `R1` in $\mathcal{V}_p$ (i.e., previously instantiated variables). Although the computed bundle of V_c does not directly depend on the instantiations of past variables from `R2`, the bundle subset to be assigned to V_c must be consistent with those variables of $\mathcal{V}_p$ that share a join-condition constraint with V_c (a condition enforced by Algorithm 8 of Section 5.5). When such a variable is from `R2`, then the instantiation of V_c is affected by the instantiations of variables from `R2`.

Below, we describe the method for computing a bundle of V_c, an attribute of relation `R`, given that some of the variables of `R` are in $\mathcal{V}_p$. The bundles are computed on the tuples of `R` present in the memory, called `R'`. First, `R'` is sorted with the variable coming earliest in the static ordering (see Section 5.4.1) as the primary key, the one coming second as the secondary key, and so on. The sorting operation clusters tuples with the same values for variables as they appear in the static ordering.

5.4.3 Data structures

We first introduce the various data structures used for computing the bundles.

- *Current-Inst* is a record of size equal to the number of variables in the CSP. It is used to store the current instantiations of variables of `R` in $\mathcal{V}_p$. This data structure corresponds to a current path in a search tree. When a variable is assigned a bundle of size greater than one, only the smallest value in the bundle is stored in *Current-Inst*, as a representative of the bundle.

- *Processed-Values* is a similar record storing cumulatively all non-representative values of the assigned bundles. While computing bundles of V_c, tuples corresponding to values for V_c in *Processed-Values* are ignored.

- *Current-Constraint* is a selection of the relation `R'` (of which V_c is an attribute) such that:

  1. Past variables have the values stored in *Current-Inst*, and

  2. the value of V_c is greater than the previous instantiation of V_c.

  Initially, the *Current-Constraint* is set to `R'`.

The tuples with the same value for V_c in *Current-Constraint* form a partition p, and the value of V_c in this partition is denoted VALUE(p). Figure 5.4 shows these data-structures under various scenarios. A partition p is marked as *checked* when VALUE(p) is part of an instantiation bundle or when p is selected to be compared with other partitions. Otherwise,

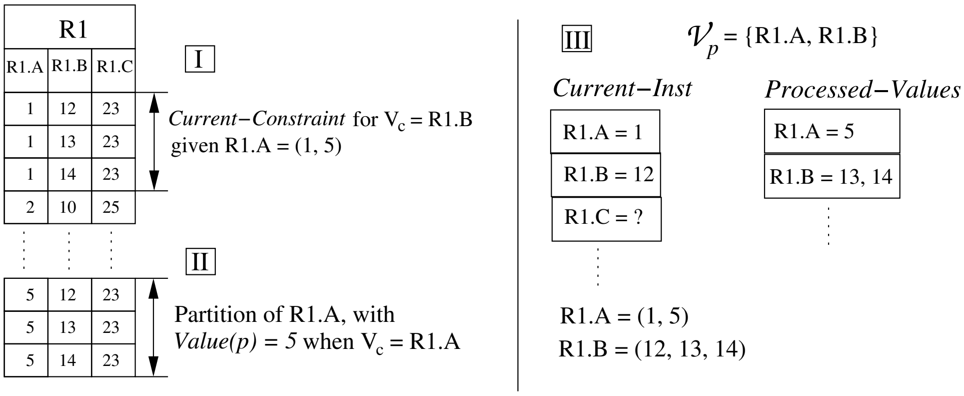

Figure 5.4: Data structures shown under 3 different scenarios.

the partition is considered *unchecked*. P_c refers to the unchecked partition with the lowest value of V_c in *Current-Constraint*. When no checked partition exists for V_c, P_c is set to a dummy value such as -1.

5.4.4 Algorithm for bundle computation

Algorithm 7 computes the next bundle of V_c given P_c. NEXT-PARTITION(p) returns the first *unchecked* partition in *Current-Constraint* following the partition p. For p= -1, NEXT-PARTITION(p) returns the first partition in *Current-Constraint*. P_c moves to the next unchecked partition at every call of Algorithm 7.

Algorithm 7 is called by Algorithm 8 of Section 5.5 for computing the bundle b_c of V_c and the bundles of the variables V_i connected to V_c with a join-condition constraint. Further, Algorithm 8 determines the subset *Inst* of the bundle b_c that is consistent with the variables V_i. This consistent set of values *Inst* is then used to instantiate V_c. This instantiation operation includes the update of the data structures *Current-Inst* and *Processed-Values*. In particular, the values in *Processed-Values* that are lesser than those associated with P_c are deleted.

We can compute all the bundles of V_c by repeatedly calling Algorithm 7, then assigning the returned bundle to V_c until Algorithm 7 returns `nil`. Thus, the algorithm described

Input: V_c, *Current-Constraint*

$bundle \leftarrow$ `nil`, the bundle to return

$P_c \leftarrow$ NEXT-PARTITION(P_c)

Mark P_c as checked

Push VALUE(P_c) into $bundle$

$P'_c \leftarrow$ NEXT-PARTITION(P_c)

while P'_c **do**

  $t \leftarrow$ tuples of P_c

  $p \leftarrow$ tuples of P'_c

  if $\pi_{\mathcal{V}_f}(t) \equiv \pi_{\mathcal{V}_f}(p)$ **then** push VALUE(P'_c) in $bundle$

  $P'_c \leftarrow$ NEXT-PARTITION(P'_c)

end

Output: $bundle$

Algorithm 7: Algorithm to generate the next bundle of V_c.

here implements a lazy approach for computing the bundles and avoids storing the entire partition of the domain of every variable.

In the method described above *Processed-Values* is the data structure that occupies the most space. Whereas all the other data structures have sizes proportional to the number of variables (and therefore cause insignificant memory overhead), the size of *Processed-Values* depends on the number of tuples and the amount of bundling performed. The worst-case scenario for *Processed-Values* occurs when all the values of a variable are in a single bundle. In this case, *Processed-Values* holds all the unique values of that variable. Suppose that there are N tuples in the relation, the relation has k attributes, and the number of unique values of the variable is $\frac{N}{l}$, where l is the average length of each partition of V_c. Then, the size of *Processed-Values* is $\frac{N}{l \times k}$ tuples. However, if this bundle goes on to form a result tuple, it will save more space than required for bundling. Even when this bundle fails to yield a result tuple, it still saves on many comparisons thereby speeding up computation.

5.5 Algorithm for join computation using dynamic bundling

This section explains how to use bundling while computing a join as a depth-first search, as sketched in Section 5.4.2. The join algorithm discussed in this section is based on the Progressive Merge Join. Our technique can be easily adapted to the simpler sort-merge join since PMJ is just an extension of sort-merge. We first describe the in-memory join algorithm, and then place it in the schema of the external join algorithm.

5.5.1 Join computation in memory

We present here the algorithm to join the sub-sets of two relations that are currently in memory. For the sake of readability, Algorithm 8 is restricted to binary join conditions (i.e., where the join conditions are between two attributes from different relations). It can be easily extended to join conditions with more than two attributes.

Algorithm 8 takes as input the level of V_c in the search tree (i.e., depth) and the current path represented by the data structure *Current-Solution*. *Current-Solution* is a record that stores the assigned bundles to the variables in $\mathcal{V}_p$. (Note that *Current-Solution* cannot be obtained from *Current-Inst* and *Processed-Values*, which were introduced in Section 5.4.3). *Variable*[] is the array of variables stored according to the static ordering sequence discussed in Section 5.4.1. When BACKTRACK is called the value for *Variable*[*depth*] in *Current-Inst* is reset, the *Processed-Values* for the variable is emptied, the value for the variable in *Current-Solution* is reset, and *Current-Constraint* is re-computed, thus undoing the effects of the previous instantiation. The function COMMON() computes the set of values in the input bundles that are consistent with each other given the applicable join-condition constraints. Because this algorithm combines sorting and constraint propagation with bundling, it produces solutions quickly, which compensates for the effort spent on bundling.

Input: *depth*, *Current-Solution*

while (depth $\leq\mid \mathcal{V} \mid$) *and* (depth $\geq$ *1*) **do**
    $V_c \leftarrow$ *Variable*[*depth*]
    $b_c \leftarrow$ next bundle for V_c using Algorithm 7
    if b_c *is empty* **then**
        BACKTRACK, decrement $depth$, and GOTO L1
    end
    $Inst \leftarrow b_c$
    repeat
        foreach $V_i \in \mathcal{V}_f$ *connected to* V_c *by a join-condition constraint* **do**
            Consider R_i the relational constraint that applies to V_i
            Select r_i from R_i according to *Current-Solution*
            repeat
                Find a bundle b_i applying Algorithm 7 on V_i and r_i
                if b_i *is empty* **then** break
                $I_i \leftarrow$ COMMON(b_i, b_c)
            until I_i *is not empty*;
            if *no* b_i **then** BACKTRACK, decrement $depth$ and Goto L1
        end
        Inst $\leftarrow$ COMMON($I_0, I_1, \ldots, I_n$)
    until *Inst is not empty*;
    Instantiate V_c with *Inst*
    Current-Solution[V_c] $\leftarrow$ *Inst*
    Increment *depth*
    L1:
end

Output: *Current-Solution*

Algorithm 8: Algorithm to compute the in-memory join using bundling.

5.5.2 Structure of the overall join algorithm

We have discussed join computation of tuples that are in memory and now describe the steps for computing the join of complete relations using our in-memory join algorithm, Algorithm 8. The join of the two input relations is computed using an approach similar to PMJ, in the two phases, sorting phase and merging phase, discussed below.

5.5.2.1 Sorting phase

The sorting phase is similar to PMJ, except that for joining the pages of relations in memory we use the bundling-based technique of Algorithm 8. The sorting phase produces the early results and also sorted sub-lists, or runs, of the relations. These runs are stored back on disk and used in the merging phase of the join. Since the memory is filled with pages from both relations, the number of runs generated for each relation is $\frac{2N}{M}$.

5.5.2.2 Merging phase

In the merging phase, as for PMJ, $\frac{M^2}{4 \times N}$ pages from every run created at the sorting phase are kept in memory. Let P_i^{rel} represent the pages in memory of relation rel and i^{th} run, where $rel \in \{0, 1\}$ and $i \in \{1, 2, \ldots, \frac{2N}{M}\}$. We define the array *solution* as follows:

$$solution[i][j] = P_i^0 \bowtie P_j^1 \,,\; i \neq j \tag{5.1}$$

We store only the first solution of a $P_i^0 \bowtie P_j^1$ in the array element $solution[i][j]$. The minimum solution from *solution*[][] is the next result of the join. The next solution from the pages that gave the minimum solution is then computed and used to fill the corresponding place in *solution*[][]. A page P_i^{rel} is removed from memory and replaced with another page from the same run only if it satisfies the following two conditions for every page P_j^{1-rel}. P_i^{rel} is being joined with:

1. No more join tuples result from $P_i^{rel} \bowtie P_j^{1-rel}$, and
2. the last tuple in P_i^{rel} is less than that of P_j^{rel}.

The tuples are compared using the same comparison criteria as the ones used for sorting. These conditions ensure the tuples are produced in sorted order (during the merging phase) and that the algorithm is complete.

5.6 Implementation and experiments

One of the goals of the XXL library [den Bercken *et al.*, 2001] is to provide an infrastructure for evaluating new algorithms in databases. For example, PMJ was evaluated experimentally using this library. In our experience, XXL provides a good infrastructure for building new database algorithms through its rich cursor algebra built on top of Java's iterator interface. We implemented our join algorithm by extending the BUFFEREDCURSOR class of the XXL library. The current implementation is a proof of concept and offers much room for improvement.

To show the feasibility of our technique, we tested our join algorithm on randomly generated relations and on data from a real-world resource allocation problem in development in our group [Lim *et al.*, 2004]. For the real-world application, we computed the sequence of the natural join of three relations, with 3, 4, and 3 attributes respectively. The corresponding CSP has 4 variables, with domain size 3, 3, 300, and 250 respectively. The resulting join of size 69 was compressed down to 32 nested tuples. For the random problems, we used relations of n = 10,000 tuples. We set the page size to 200 tuples and the available memory size to $M = \frac{2N}{5}$, where $N = 10000/200$. We executed the query of our running example over five such pairs of relations. The result of the query had an average of 8,500 tuples, which indicates that the the query is a selective one. The number of tuples produced by bundling was reduced to 5,760 bundled tuples, an average of 1.48 tuples per bundle. The number of pages saved was more than $\frac{N}{4}$ and slightly less than $\frac{N}{3}$. Even if the worst-case scenario for the join occurred for every in-memory join (which is a highly unlikely event), the additional cost due to bundling is given by $\frac{N}{l \times k}$, where $\frac{N}{l}$ is the number of unique values of an attribute and k is the number of attributes in one relation (which is 3 here). For the worst case when $l = 1$, the overhead in terms of pages is negligible. Again, the worst-case described here is of the current implementation, which offers much room for improvement.

5.7 View materialization using dynamic bundling

In this section, we discuss the benefits of using our dynamic-bundling-based join algorithm for view materialization. We also discuss issues related to view maintenance using dynamic bundling.

5.7.1 Benefits of materializing views with bundling

View materialization is a process that executes the query defining the view and stores the result as a new table in the database. Typically views are materialized when it is expensive (in terms of I/O operations) to recompute them every time they are queried. Views that are expensive to recompute invariably involve joins. We can materialize views defined by joins using a dynamic-bundling-based join algorithm. The dynamic-bundling-based join-algorithm produces compacted results that cause no overhead for 'de-compaction.' A compact representation of the query results leads to a materialized view that stores more tuples per page. Consequently, when the materialized view is queried the query processor reads fewer pages, thus reducing the amount of I/O performed and thereby speeding up query processing. These advantages make the dynamic-bundling-based join algorithm an ideal tool for materializing views.

5.7.2 View maintenance

Before discussing the issues related to maintaining bundled materialized views, we summarize known important concepts in view maintenance.

5.7.2.1 Background of view maintenance

View maintenance reflects the effect, on the materialized view, of an update to the base tables (i.e., the original tables). Views are typically maintained and updated incrementally, and the maintenance is transparent to the database user [Gupta and Mumick, 1995]. In incremental maintenance, we compute the change to the materialized view from the change in the base relations. Consequently, most of the view maintenance techniques treat the view definition as a mathematical formula and maintain the view as follows.

Computing the update: The view maintenance techniques apply a differentiation step to the formula defining the view in order to obtain a sequence of operations to update the materialized view [Ceri and Widom, 1991].

Refreshing the view: The operations performed on the view are a sequence of one or more insertions and/or deletions. The process of executing these operations is known as the *refresh* step. A view can be refreshed with the transaction that updates the base tables (i.e., immediate update), or the refresh can be delayed (i.e., deferred update).

5.7.2.2 View maintenance and bundling

Below, we study whether there exists a speed up or an additional cost in querying a bundled materialized view and in maintaining such view. From the above discussion on view maintenance, we note that incremental maintenance of views does not depend on the storage format of the tuples (i.e., bundled or otherwise). Further, the process of determining the tuples to insert or delete from the view does not depend on the storage format of the materialized view. We now address two issues:

1. Is it more difficult to search or query a bundled materialized view than a conventional one?

2. Is it more expensive to refresh a bundled view than to refresh a conventional view?

The remainder of the section addresses these questions.

5.7.2.3 Searching bundled materialized views

Searches or queries on materialized views are often optimized by generating indexes on one ore more attributes of the view. Tree-based indexing methods, such as B+ trees, are popular indexing strategies. Such strategies can be easily extended to create indexes for bundled materialized views. The basic structure of an index is a list of value-pointer pairs. Figure 5.5 shows one such simple tree-based index on an attribute of a view. The pointer

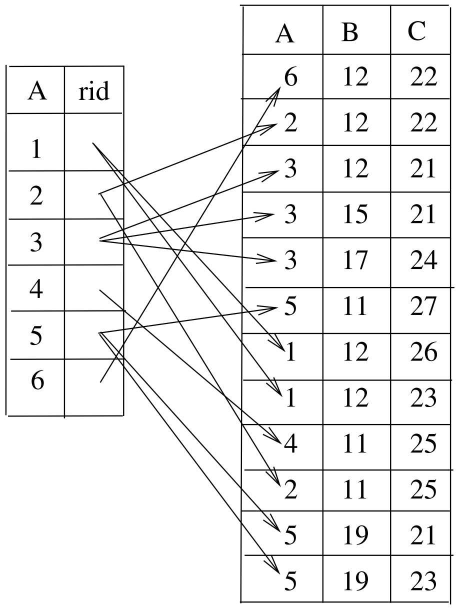

Figure 5.5: Illustration of an index.

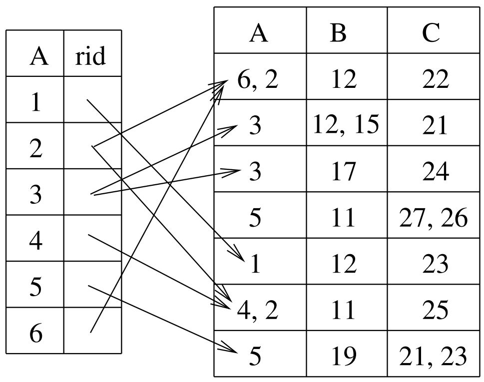

Figure 5.6: Illustration of an index on a bundled materialized view.

shown in the index is the row identifier (*rid*) of the tuple. An *rid* typically stores the page location of the tuple and the offset into the page. The presence of nested values in the view does not interfere in the generation of these value-pointer pairs. Figure 5.6 shows an index on a bundled materialized view. The differences between the two indexing schemes are as follows:

- In a bundled materialized-view, two different values may have the same *rid* whereas in a conventional view, a *rid* is associated with only one value.

- The number of rows, and hence the number of *rid*s and the size of the index, in a bundled materialized view may be less than the corresponding ones in a conventional view.

These differences do not hamper in any way searching the view using an index.

Another approach to speeding up searches on materialized views is by sorting materialized view on an attribute. Although this approach is not as effective as using indices, it is however used when view updates are so frequent that maintaining an index causes an expensive overhead. Bundled materialized views cannot be sorted using an ordering that will be useful to searching, which constitutes a limitation of bundled materialized views.

5.7.2.4 Refreshing bundled materialized views

Refreshing bundled materialized views is a straightforward operation. In order to reflect the changes made in the base tables, we can use either one of the two above-listed common approaches (i.e., immediate or deferred). As for a conventional view, a new tuple is appended to the end of the bundled view. Deleting a row requires searching for the tuple to delete. Searching can be optimized by the use of indices. If the tuple to be deleted is part of a bundled tuple, only the values representing the tuple to delete are removed, otherwise the entire tuple is deleted.

5.8 Related work

Below, we discuss works related to compacting databases and/or modeling database problems as a CSP.

The idea of data compression is not new and is used in compressed database systems [Roth and Horn, 1993]. In these systems, data is stored in a compressed format on disk. It is decompressed either while loading it into memory or while processing a query. The compression algorithms are applied at the attribute level and are typically dictionary-based techniques, which are less CPU-intensive than other classical compression techniques [Westmann *et al.*, 2000]. Although most of the work in compressed databases applied to relations with numerical attributes [Roth and Horn, 1993], investigations on string attributes also were carried out by Chen et al. [2001]. Another feature of compressed databases that differs from our approach is that the query results passed to the next operator are uncompressed and thus are likely to be large. Our work differs from the above in that we reduce some of the redundancy present between tuples of a given relation. Our techniques are independent of the data type of an attribute. Further, the results of our queries are compacted, thereby assisting the next operator and reducing the storage of materialized views on disk. When these compacted results are loaded into memory for query processing, the de-compaction operation is effectively cost-free. The only costs associated with our techniques are those for performing the compaction. Finally, the compaction is carried out while the query is being evaluated, and is not a distinct function performed in separation.

Mamoulis and Papadias present a spatial-join algorithm using mechanisms of search with forward checking [1998], which are fundamental mechanisms in Constraint Processing. They store the relations representing spatial data in R-tree structures, and use the structures to avoid unnecessary operations when computing a join. The constraints under consideration are binary. The key idea is to reduce the computational cost by propagating the effects of search, thereby detecting failure early. Our technique is not restricted to binary constraints, and is applicable to constraints of any arity. Further, it differs from the approach of Mamoulis and Papadias in that it reduces I/O operations and compacts join results in addition to reducing computational operations.

Bernstein and Chiu [1981], Wallace et al. [1995], Bayardo [1996], Miranker et al. [1997] exploit the standard consistency-checking techniques of Constraint Processing to reduce the number of the intermediate tuples of a sequence of joins. While Wallace et al. consider Datalog queries, Bayardo and Miranker et al. study relational and object-oriented databases. Our CSP model of join query differs from their work in that the constraints in our model include both relational and join-condition constraints, whereas Bayardo and Miranker et al. model the relational constraints as CSP variables and only the join-condition constraints as CSP constraints. Thus, our model is finer in that it allows a more flexible ordering of the variables of the CSP, which increases the performance of bundling.

Finally, Rich et al. [1993] propose to group the tuples with the same value of the join attribute (redundant value). Their approach does not bundle up the values of the join attribute or exploit redundancies that may be present in the grouped sub-relations.

Summary

In this chapter, we discussed the use of dynamic bundling in databases. We focused our investigations on the join operator, which we modeled as a CSP. We presented a space-efficient sort-based bundling algorithm and a dynamic-bundling-based join-algorithm, which uses the framework of the Progressive Merge Join algorithm. Our preliminary implementation established that the savings due to bundling are worthwhile even in the worst-case scenario. Finally, we discussed the use of our approach in materialized views.

Chapter 6

Future Work and Conclusions

In this chapter, we suggest some directions for future research and draw the conclusions of the thesis.

6.1 Future work

We first discuss how sorting the definitions of the constraints, which is a popular operation in databases but not in CSPs, can improve the time complexity of the bundling algorithm on CSPs. Then we envision some possibilities for using bundling in databases beyond query join computation, such as in Constraint Databases [Revesz, 2002], sampling methods, main-memory databases, and automatic categorization of query results. We also present an alternative approach for computing joins by using bundled relations.

6.1.1 Sorting constraints to improve bundling

In order to reduce the complexity of our bundling algorithms, we can sort the definitions of the constraints. By sorting a constraint, we introduce structure to the constraint that can be exploited for quickly comparing tuples when building a discrimination tree to compute interchangeability. We suspect that sorting would improve the time and space complexity, as well as the practical performance, of the algorithms we presented in this thesis. One possible impediment to this strategy would be its combination with dynamic variable ordering. The use of dynamic variable ordering during search significantly improves the performance of search in practice. Sorting constraint definitions when using dynamic variable ordering would require frequent re-sorting of the constraints, which could be prohibitively expensive.

Consequently, there is a trade-off that remains to be investigated between the benefits of sorting the constraint definitions to improve the performance of bundling and the cost of frequent resorting under dynamic ordering of variables in search.

6.1.2 Continuous CSPs and constraint databases

Our techniques are designed for finite domains. It would be interesting to apply the concepts and extend our techniques to infinite domain CSPs, usually called continuous CSPs. Such an extension is particularly useful in the context of constraint databases, such as in spatial databases, where the value of an attribute is a continuous interval [Revesz, 2002].

6.1.3 Sampling methods

We speculate here that the materialized views produced using dynamic bundling can help the sampling operator to sample more accurately. Exploratory analysis of data does not require exact answers to the queries. Results based on a sampling of data often prove satisfactory in exploratory analysis. To answer queries used for exploratory analysis, the query processor uses the sampling operator, which produces a subset of the relation it acts upon. In general, the query processor can improve the effectiveness and accuracy of the sampling operator by pushing it down in the query tree closer to the base relations [Gryz *et al.*, 2004]. IBM's DB2 implements sampling using two operators, RAND and TABLESAMPLE. The TABLESAMPLE operator is more effective than RAND, however it can only be applied to base tables or materialized views. Gryz et al. [2004] proposed two methods that enable pushing the sampling operator closer to the base tables and consequently allow the query planner to choose the TABLESAMPLE operator over RAND. The authors suggest materializing results of intermediate queries as one of the approaches.

We believe that the materialized views produced using dynamic bundling can improve the accuracy of TABLESAMPLE. Our claim is based on the fact that a bundled tuple is more representative of the content of the table and can be given more weight while sampling, thus making the sampled query-results a better approximation of the results of the whole relations. This improvement will be in addition to the space saved, thanks to dynamic bundling, by the compaction of the materialized view.

6.1.4 Main-memory databases

In his keynote address in SIGMOD 2004, Jim Gray [2004] mentioned main-memory databases as a promising research area given that large sized main-memory, up-to tera bytes, have become common. In the context of such large memory, the fact that random access is considerably slower than sequential access becomes critical to performance. In such a scenario, algorithms producing related results clustered together (similar to the bundled results of dynamic bundling) are valuable to the query processor. Further, with such large memory, the query processor will give priority to optimizing the number of comparisons over the amount of memory needed to process the queries. This priority matches that of most algorithms in Constraint Processing. Therefore, we identify main-memory databases as an area where algorithms from Constraint processing can improve query evaluation. Interestingly, no contributions on main-memory databases were presented at SIGMOD 2004, and, hence, this research direction may be a particularly promising and rewarding one.

6.1.5 Automatic categorization of query results

In this section, we discuss how modeling query execution as an interactive and incremental search algorithm can possibly minimize the amount of query processing when computing query results that are categorized using the algorithms of Chakrabarti et al. [2004].

The authors address the problem of information overload of users due to large query results by automatically categorizing the results. They develop a categorization technique that identifies attributes to use for creating categories, and the technique also suggests an ordering of the categorizing attributes. Let us consider the example of a user searching for a home to purchase. The categorization algorithm identifies `Neighborhood`, `Price`, and `#Bedrooms` as the attributes to use for effective categorization. It also gives an ordering of attributes as {`Neighborhood`, `Price`, `#Bedrooms`}. For every attribute, the categorization algorithm generates categories (i.e., ranges) for the results. For example, the algorithm may choose three categories of price ranges (e.g., 200K–225K, 225K–250K, and 250K–300K), and three categories for number of bedrooms (e.g., 1–2, 3–4, and 5–9).

The results are presented to the user as a hierarchical tree. The user is initially shown the root, which is used only as a structural element and gives access to the complete result of the query. The categories of the first attribute, here `Neighborhood`, are children of the root node. The categories of `Price` are the children nodes of a `Neighborhood` node and so on. The leaf nodes of the tree are the query results. The user can process the root and all non-leaf nodes in two ways:

1. *View* the results at that level.

2. *Expand* the node to see more categories (nodes).

In their paper, Chakrabarti et al. define and study the cost model for measuring the user's effort spent in browsing through the data. They focus on queries that do not involve any aggregations and are simple Select-Project-Join (SPJ) queries.

The current work does not address the issue of optimizing query evaluation knowing that results will be categorized. An interesting, but yet speculative, direction can be lazy evaluation of the query results responding to the expansion sequence of the user. Note that the user is expanding one attribute at a time, which is a variable in CSP terminology. The expansion of a tree node can be interpreted as instantiating a variable. We can model the query execution as solving a CSP using search and compute the query result incrementally as the user expands nodes of the tree. If a user does not expand a majority of the tree, then the approach described above can save a significant amount of computation. The issues to be addressed in this approach:

- When implementing such a strategy, the relational constraints may be challenging to implement. A mechanism to maintain the coupling between values of attributes from the same tuple will have to be enforced.
- The lazy evaluation method also has the risk of presenting values to the user that will need to be backtracked upon later as more variables are instantiated. One solution could be to show the path leads to an empty set. Using stronger look-ahead strategies than simple forward checking (e.g., maintaining arc-consistency [Sabin and Freuder, 1997]) in dynamic bundling may reduce the severity of this problem. However, user frustration with empty results may remain an issue.

6.1.6 Computing joins using bundled relations

The bundling-based join algorithm presented in Chapter 5 computes the join of normal (i.e., 'non-bundled') relations to produce a bundled result. One interesting alternative to investigate consists in storing the input relations in a bundled form, and using these bundled relations, instead of the normal relations, for query processing [Revesz, 2005]. Figure 6.1 shows the process explored in Chapter 5, and Figure 6.2 illustrates the alternative.

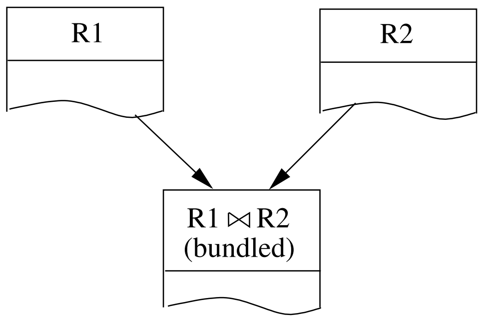

Figure 6.1: Bundling-based join algorithm of Chapter 5.

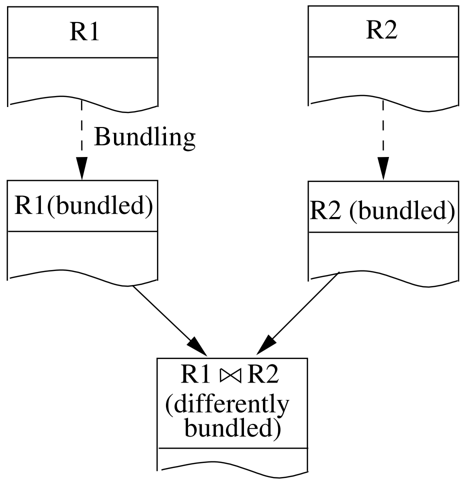

Figure 6.2: An alternative approach: joining bundled relations.

The advantage of this alternative approach is that bundled relations use less disk space than a normal relation. Further, when the query processor loads the bundled relations into the main-memory, the bundled relations occupy less main-memory than the original ones, thereby reducing the number of I/O operations. It is interesting to investigate whether or not the join algorithm presented in Chapter 5 can take the bundled relations as input. One of the challenges is the handling of bundles that are broken by join-condition constraints. Note that the detection of new symmetries may also arise as others are broken.

We re-state this idea using CSP terminology. Each attribute of a relation is a variable, and the relation itself is a constraint on the variable. The idea is to compute the NI sets for all variables given their respective relational constraints. Next, we re-create the relational constraints by using bundles instead of singletons in the constraint tuples. We use these bundled constraints when joining two relations while accounting for any join-condition constraints.

Once we have determined that the new approach is feasible, one still needs to compare both approaches, their cost, and advantages.

6.2 Conclusions

We presented an efficient method of computing Neighborhood Interchangeability (NI) in non-binary CSPs. We used the NI computation dynamically in search to solve non-binary CSPs and produce robust solutions. We showed that dynamic bundling is guaranteed never to perform worse than non-bundling when finding all solutions. We conducted extensive experiments to compare DynBndl and FC when finding one solution and to study the effect on performance of DynBndl with varying CSP parameters. We established empirically that in the phase transition region, DynBndl produces multiple solutions and also performs significantly better than FC in terms of CPU time, `NV` and `CC`. We also designed and implemented a better implementation strategy for non-binary forward checking. We showed how the DT for binary CSPs can be extended to detect some substitutable values and also presented a technique for non-binary CSPs.

We reviewed the database literature and analyzed the connections between the fields of Constraint Satisfaction and Databases. We presented a new approach to modeling a join query as a CSP. We developed a sort-based bundling algorithm that is more suited to database requirements. We then used the new bundling algorithm to develop a bundling-based join algorithm. We showed, with a proof of concept implementation, that such a join algorithm is feasible and advantageous. We showed that the join algorithm can be used as a view materialization algorithm and can lead to savings in disk space and main memory. Finally, we identified future directions for research.

Appendix A

Results of Experiments

This appendix presents the results of experiments over all the 16 datasets. The tables show the percentage improvement, the upper (UL) and lower (LL) confidence levels of the improvement at 95% confidence level for CPU time `NV` and `CC`. The tables also list the FBS due to dynamic bundling at every tightness value. The confidence intervals are computed using the t-distribution. The F-value is the test statistic of ANOVA. A higher F-value indicates more confidence in there being a difference in between in the two means. An F-value greater than 9.4 indicates a difference (poisitive or negative). The mean value indicates whether the difference was an improvement or otherwise. Table A.1 is a guide to the datasets and the tables showing their results.

The appendix also lists the graphs comparing CPU time, `CC`, `NV` and FBS performance of DynBndl and FC for a subset of the datasets (a = 15). Figures A.1, A.2, A.3, and A.4 show these results.

Table A.1: Index of results.

Dataset#	Table
1, 2	Table A.2
3, 4	Table A.3
5, 6	Table A.4
7, 8	Table A.5
9, 10	Table A.6
11, 12	Table A.7
13, 14	Table A.8
15, 16	Table A.9

Dataset #1, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.4000	-25.86	-42.59	-61.53	63.86	23.01	12.91	1.49	11.94	22.13	11.83	0.18	8.13	263.4
0.4500	6.21	-6.26	-20.37	1.87	25.91	16.19	5.2	18.21	24.48	14.49	3.19	12.57	80.53
0.4750	19.41	8.70	-3.43	4.20	28.83	19.50	8.94	26.44	27.14	17.51	6.60	18.98	48.46
0.5000	30.20	20.93	10.42	27.97	33.40	24.67	14.78	43.70	31.95	22.96	12.78	34.87	25.00
0.5250	37.65	29.37	19.98	61.32	36.1	27.71	18.23	56.74	34.84	26.23	16.48	47.43	7.89
0.5500	44.12	31.93	17.09	30.03	43.27	31.07	16.23	30.77	42.06	29.49	14.2	25.03	1.07
0.5750	34.70	26.03	16.20	46.10	34.27	25.65	15.89	47.65	33.64	24.87	14.94	41.89	0.30
0.6000	37.14	28.78	19.32	58.46	35.63	27.19	17.64	54.34	35.50	26.98	17.33	50.66	0.06
0.6500	34.22	25.47	15.57	43.86	31.18	22.16	11.95	34.61	31.29	22.21	11.93	32.32	0.00
0.7000	34.88	26.23	16.43	46.95	28.14	18.72	8.05	24.32	29.55	20.24	9.70	26.22	0.00

Dataset #2, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.2750	-12.21	-30.04	-50.70	25.05	37.41	25.00	10.14	22.00	31.42	20.24	7.23	17.68	43.05
0.3000	-86.44	-114.28	-146.27	236.82	23.88	9.15	-8.43	4.32	21.57	9.19	-5.14	3.42	173.05
0.3500	-67.6	-92.62	-121.38	175.23	24.84	10.29	-7.06	4.98	22.39	10.15	-4.03	4.21	111.35
0.4000	-42.29	-63.54	-87.96	98.65	25.8	11.44	-5.69	5.73	22.59	10.37	-3.77	4.41	30.03
0.4625	13.56	0.65	-14.18	0.02	29.35	15.68	0.01	9.34	26.04	14.37	0.86	8.85	9.71
0.4875	27.85	17.08	4.70	14.3	31.3	18.01	2.14	11.95	28.34	17.04	3.94	12.83	4.57
0.5000	35.24	25.57	14.46	35.57	33.5	20.63	5.28	15.48	30.86	19.95	7.32	18.21	2.44
0.5125	35.95	26.38	15.39	38.25	34.31	21.60	6.43	16.95	31.67	20.89	8.41	20.20	1.84
0.5250	38.51	26.41	11.92	23	37.08	20.94	0.67	10.36	34.13	20.42	3.85	11.51	0.68
0.5375	34.4	24.61	13.36	32.58	33.94	21.17	5.92	16.29	31.45	20.64	8.12	19.67	0.13
0.5500	43.23	36.14	28.17	114.84	40.95	31.43	20.37	52.31	38.39	30.27	21.07	66.92	0.11
0.6000	36.53	28.61	19.69	64.81	37.38	27.28	15.55	37.86	34.67	26.06	16.31	46.93	0.00

Table A.2: Results for Datasets #1 and 2.

Dataset #3, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.3250	25.6	13.77	0.07	7.97	41.84	30.31	16.5	33.51	37.42	27.21	15.34	34.88	15.59
0.3625	34.21	23.31	10.61	23.67	39.09	26.49	11.29	23.18	35.61	24.65	11.84	25.64	10.12
0.3750	32.36	19.36	3.86	11.83	34.22	18.4	-1.22	9.03	31.21	17.63	1.37	9.16	4.60
0.3875	41.16	22.85	-1.16	7.24	42.7	20.13	-11.34	5.62	39.04	19.54	-6.21	4.84	3.55
0.4000	38.94	27.2	13.21	25.75	36.53	21.27	2.33	11.72	33.9	20.85	5.23	13.3	2.16
0.4125	37.33	25.28	10.92	21.7	36.39	21.09	2.11	11.54	34.31	21.35	5.82	14.03	0.82
0.4250	42.95	28.83	11.21	18.66	41.25	22.95	0.1	9.5	38.31	22.61	2.92	10.1	0.19
0.4750	34.89	22.38	7.46	16.39	35.82	20.39	1.24	10.84	33.15	19.96	4.16	12.06	0.00
0.5500	32.77	19.85	4.45	12.51	34.14	18.3	-1.35	8.94	31.77	18.3	2.17	9.94	0.00
0.6500	38.27	26.4	12.26	24.01	31.21	14.66	-5.87	6.27	28.83	14.79	-2.03	6.23	0.00
Dataset #4, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.3000	-32.25	-48.52	-66.81	91.7	21.04	10.41	-1.64	7.99	19.7	9.61	-1.75	5.75	12.46
0.3250	0.48	-11.77	-25.53	7.26	22.52	12.09	0.26	10.22	21.23	11.33	0.19	8.15	7.10
0.3500	17.47	7.31	-4.1	3.38	23.57	13.28	1.61	12.06	22.61	12.89	1.94	10.73	5.01
0.3625	24.06	14.71	4.22	14.84	24.87	14.76	3.29	14.63	23.78	14.2	3.42	13.22	3.81
0.3750	33.97	25.84	16.71	52.37	26.29	16.37	5.11	17.82	25.36	15.98	5.42	17.09	1.93
0.3875	36.66	28.87	20.11	67.99	28.11	18.44	7.46	22.58	27.13	17.97	7.66	22.12	0.82
0.4000	37.49	29.79	21.15	73.32	28.29	18.63	7.68	23.06	27.39	18.27	8	22.94	0.47
0.4250	34.15	26.04	16.94	53.32	27.41	17.64	6.56	20.66	26.88	17.69	7.35	21.37	0.01
0.5000	29.81	21.17	11.47	33.16	27.71	17.98	6.94	21.46	27.3	18.17	7.88	22.65	0.00

Table A.3: Results for Datasets #3 and 4.

Dataset #5, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.4500	-89.63	-112.82	-138.84	338.55	21.68	10.25	-2.86	6.97	19.59	9.49	-1.88	5.6	531.28
0.5000	-37.49	-59.06	-84.01	80.1	26.45	12.63	-3.78	6.86	23.47	11.12	-3.21	4.91	425.49
0.5500	7.07	-7.51	-24.38	1.95	31.53	18.66	3.38	13.37	28.17	16.59	3.13	11.62	46.26
0.5750	25.94	14.32	0.88	8.89	34.79	22.54	7.99	19.39	31.8	20.8	8.02	19.2	27.49
0.5875	37.16	24.15	8.45	17.05	40.42	25.59	7.07	15.97	37.22	23.85	7.63	15.73	13.55
0.6000	39.33	29.81	18.8	46.6	39.28	27.87	14.32	30.43	36.79	26.6	14.76	33.76	5.86
0.6125	41.76	32.62	22.05	57.97	41.38	30.37	17.29	36.92	39.13	29.31	17.91	42.49	1.64
0.6250	40.49	31.15	20.35	51.81	39.99	28.72	15.33	32.55	37.71	27.66	15.99	37.01	0.47
0.6500	41.38	32.65	22.62	63.97	42.59	32.35	20.29	46.82	40.37	31.23	20.7	54.51	0.05
0.7000	39.17	31.2	22.18	72.8	43.52	34.67	24.44	69.62	40.89	32.92	23.89	78.83	0.00

Dataset #6, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.4500	-118.28	-145.36	-175.8	465.18	21.25	10.73	-1.2	8.46	20.2	9.87	-1.79	5.76	34.66
0.5000	-40.65	-58.1	-77.72	121.16	22.36	11.99	0.22	10.18	21.16	10.96	-0.57	7.18	15.90
0.5500	20.15	10.25	-0.89	6.75	24.66	14.59	3.18	14.48	23.7	13.82	2.67	11.8	6.77
0.5750	34.32	26.18	17.02	53.18	29.93	20.56	9.94	28.59	29.05	19.87	9.5	26.17	2.01
0.5875	35.53	27.54	18.55	59.9	30.17	20.84	10.26	29.41	29.36	20.22	9.89	27.2	0.95
0.6000	36.07	28.14	19.23	63.07	30.99	21.76	11.31	32.23	30.18	21.14	10.94	30.09	0.19
0.6125	33.36	25.1	15.8	48.21	29.68	20.29	9.63	27.79	29.08	19.9	9.53	26.25	0.03
0.6250	30.73	22.14	12.48	36.17	29.36	19.92	9.22	26.76	28.68	19.46	9.03	24.96	0.00
0.6500	24.99	15.68	5.22	16.8	27.67	18	7.04	21.76	27.07	17.63	6.98	20.07	0.00
0.6750	19.16	9.13	-2.14	5.29	26.97	17.21	6.14	19.89	26.5	16.99	6.25	18.49	0.00
0.7500	11.04	0	-12.4	0	23.58	13.36	1.78	12.32	23.76	13.9	2.76	11.94	0.00

Table A.4: Results for Datasets #5 and 6.

Dataset #7, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.3500	-83.12	-107.16	-134.34	275.46	20.81	10.24	-1.74	7.87	20.14	9.53	-2.5	5.08	33.44
0.4000	-20.91	-36.78	-54.73	50.95	22.06	11.65	-0.14	9.01	20.97	10.47	-1.43	6.2	10.91
0.4250	11.18	-0.47	-13.66	0.01	22.39	12.03	0.28	10.26	21.99	11.62	-0.12	7.74	7.13
0.4375	18.52	7.83	-4.27	3.45	23.45	13.23	1.65	12.13	23	12.77	1.18	9.48	6.36
0.4500	31.21	19.33	5.41	14.38	27.65	14.95	0.02	9.91	27	14.24	-0.75	7.18	5.62
0.4625	35.15	26.64	17.01	49.84	28.11	18.51	7.64	23.08	27.72	18.11	7.23	20.25	2.36
0.4750	39.19	31.2	22.18	72.65	30.82	21.59	11.12	31.74	30.38	21.13	10.64	28.57	0.66
0.5000	38.24	30.14	20.97	66.79	31.32	22.15	11.76	33.52	30.65	21.44	11	29.54	0.03
0.5500	31.75	22.8	12.67	34.76	29.24	19.79	9.09	26.47	29.12	19.7	9.03	24.42	0.00
0.6000	27.73	18.24	7.52	21.07	27.02	17.28	6.23	20.09	26.47	16.7	5.63	16.94	0.00

Dataset #8, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.3500	-111.68	-128.55	-146.78	916.85	16.97	9.92	2.27	14.96	16.05	9.35	2.12	12.91	8.57
0.4000	-12.39	-23.07	-34.76	41.3	18.93	10.71	1.67	12.89	18.15	10.37	1.84	11.46	4.77
0.4500	34.15	27.9	21.05	102.53	24.84	17.23	8.84	32.32	24.11	16.89	8.98	32.73	0.93
0.4750	35.4	29.26	22.54	114.88	26	18.5	10.24	37.48	25.37	18.28	10.5	38.96	0.04
0.4875	33.33	26.99	20.06	94.89	25.46	17.91	9.59	35.03	24.86	17.71	9.89	36.35	0.01
0.5000	30.98	24.43	17.25	75.18	25	17.4	9.04	33.01	24.29	17.09	9.2	33.6	0.00
0.5125	30.26	23.63	16.37	69.65	25	17.41	9.04	33.02	24.35	17.15	9.27	33.86	0.00
0.5250	27.61	20.73	13.2	51.73	24.63	16.99	8.58	31.41	23.94	16.71	8.79	31.97	0.00
0.5500	22.05	14.65	6.54	24.04	23.3	15.53	6.97	26.14	22.83	15.49	7.45	27.09	0.00
0.6000	14.2	6.04	-2.88	3.72	22.19	14.31	5.62	22.21	21.78	14.34	6.2	22.93	0.00

Table A.5: Results for Datasets #7 and 8.

Dataset #9, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.2500	-37.25	-59.57	-85.51	75.97	22.42	9.17	-6.34	4.94	22.4	9.26	-6.12	3.04	19574.08
0.3000	-8.86	-26.56	-47.14	19.3	25.87	13.21	-1.61	8.37	24.78	12.03	-2.88	5.29	11906.16
0.3500	19.26	6.13	-9.13	1.39	31.31	19.58	5.85	17.09	29.1	17.09	3.04	11.32	286.97
0.3750	36.32	25.97	13.93	31.46	33.78	22.47	9.23	22.58	32.22	20.74	7.31	17.4	93.63
0.3875	42.22	32.83	21.91	55.1	37.59	26.93	14.45	33.26	36.29	25.49	12.87	27.9	35.15
0.4000	44.13	35.05	24.49	64.79	37.49	26.82	14.32	32.97	36.35	25.57	12.95	28.09	21.37
0.4125	41.55	32.05	21	51.94	38.28	27.74	15.4	35.53	37.48	26.89	14.5	31.62	5.13
0.4250	46.42	37.71	27.58	77.95	40.52	30.36	18.47	43.6	39.92	29.74	17.84	40.16	0.96
0.4500	43.02	33.75	22.98	58.98	38.06	27.48	15.1	34.81	37.73	27.17	14.83	32.4	0.28
0.5000	38.23	28.18	16.51	38.13	38.18	27.62	15.26	35.19	37.67	27.11	14.76	32.22	0.00
0.6000	34.72	24.11	11.77	26.47	31.57	19.88	6.2	17.61	31.75	20.18	6.66	16.38	0.00

Dataset #10, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.3000	-33.05	-52.69	-75.23	74.63	24.02	11.64	-2.75	7.31	22.73	10.57	-3.49	4.62	296.19
0.3500	19.32	7.41	-6.25	2.47	29.86	18.44	5.16	16.42	28.33	17.05	4.01	12.94	59.57
0.3750	35.94	26.49	15.64	39.45	30.89	19.64	6.55	18.58	29.69	18.63	5.84	15.74	18.22
0.4000	42.85	34.42	24.74	74.13	34.87	24.27	11.93	28.8	34.05	23.68	11.68	27.04	4.8
0.4125	35.09	25.51	14.51	36.12	33.47	22.63	10.03	24.84	33.02	22.49	10.29	24.01	1.83
0.4250	41.65	33.03	23.15	66.97	35.66	25.18	12.99	31.18	35.04	24.83	13.01	30.15	0.21
0.4500	37.4	28.15	17.55	45.54	33.64	22.83	10.27	25.31	32.93	22.38	10.18	23.76	0.01
0.5000	31.84	21.78	10.23	25.13	33.35	22.5	9.88	24.54	32.82	22.26	10.03	23.47	0.00
0.5500	26.45	15.6	3.14	11.98	29.77	18.34	5.04	16.23	29.49	18.4	5.56	15.3	0.00
0.6000	30.55	20.3	8.54	21.45	28.49	16.85	3.31	13.81	28.61	17.39	4.4	13.5	0.00

Table A.6: Results for Datasets #9 and 10.

Dataset #11, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL	F-value	UL	Mean	LL		
0.2000	-13.29	-34.9	-60.62	23.22	26.76	12.19	-5.28	6.05	25.56	10.81	-6.86	3.16	388.07
0.2250	2.3	-15.29	-36.06	5.83	26.33	12.49	-3.95	6.74	25.23	11.24	-5.36	3.82	79.81
0.2500	20.61	6.32	-10.55	1.23	27.34	13.69	-2.52	7.78	26.3	12.51	-3.86	4.79	33.41
0.2750	38.25	27.12	14	28.83	30.48	17.42	1.91	11.76	30.04	16.95	1.41	9.26	10.4
0.2875	35.69	24.11	10.45	21.92	31.47	18.6	3.31	13.28	31.72	18.94	3.78	11.84	5.27
0.3000	44.77	34.82	23.08	52.74	36.05	24.04	9.77	22.14	35.76	23.74	9.48	19.72	1.44
0.3125	43.98	33.89	21.99	49.33	36.46	24.52	10.34	23.08	36.25	24.32	10.17	20.86	0.65
0.3250	39.98	29.17	16.42	34.26	35.72	23.64	9.3	21.38	35.49	23.43	9.1	19.13	0.01
0.3500	36.81	25.43	12	24.8	33.97	21.56	6.83	17.72	34.04	21.7	7.06	16.07	0.00
0.4000	32.41	20.24	5.87	14.72	32.89	20.28	5.31	15.69	33.09	20.57	5.71	14.24	0.00
Dataset #12, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.2000	-33	-55.06	-80.76	64.54	25.05	11.32	-4.91	6.03	23.91	10.42	-5.45	3.6	69
0.2500	17.56	3.89	-12.04	0.53	25.71	12.1	-3.99	6.65	24.75	11.41	-4.28	4.36	17.79
0.2625	27.48	15.46	1.44	9.46	26.58	13.14	-2.77	7.54	25.62	12.44	-3.08	5.23	11.43
0.2750	39.59	29.57	17.89	41.22	31.14	18.53	3.61	13.73	30.47	18.15	3.64	11.9	4.54
0.2875	29.42	17.72	4.08	12.76	29.9	17.06	1.88	11.78	29.53	17.04	2.35	10.36	1.15
0.3000	43.08	33.64	22.64	56.43	33.66	21.52	7.15	18.4	33.06	21.2	7.24	16.84	0.39
0.3125	42.47	32.93	21.81	53.52	34.38	22.37	8.15	19.91	33.87	22.15	8.36	18.61	0.05
0.3250	38.58	28.4	16.53	37.44	33.19	20.95	6.48	17.45	32.72	20.79	6.76	16.12	0.00
0.3500	34.48	23.62	10.95	24.35	31.36	18.79	3.92	14.1	31.14	18.94	4.58	13.08	0.00
0.4000	28.8	16.99	3.23	11.64	29.84	16.99	1.79	11.69	29.8	17.37	2.72	10.8	0.00
0.4500	27.07	14.98	0.88	8.84	30.08	17.28	2.14	12.06	29.6	17.13	2.45	10.48	0.00

Table A.7: Results for Datasets #11 and 12.

Dataset #13, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.3500	-76.6	-101.86	-130.74	217.96	21.88	10.31	-2.98	6.9	21.17	9.54	-3.81	4.19	2254.77
0.4000	-4.22	-19.13	-36.17	13.53	27.04	16.23	3.82	14.98	25.13	14.07	1.39	9.58	240.46
0.4500	30.97	21.09	9.81	24.79	28.84	18.3	6.2	18.91	27.58	16.89	4.62	14.26	47.62
0.4750	45.1	37.24	28.27	95.9	36.23	26.78	15.94	42.23	35.33	25.79	14.84	37.06	10.75
0.4875	45.79	38.04	29.17	101.21	37.4	28.13	17.48	47.16	36.92	27.61	16.92	43.48	1.21
0.5000	44.02	36.01	26.86	88.07	37.32	28.04	17.38	46.83	36.59	27.23	16.49	42.09	0.06
0.5250	41.51	33.14	23.58	71.61	35.95	26.46	15.57	41.1	35.52	26	15.08	37.78	0.02
0.5500	36.28	27.17	16.75	44.41	33.88	24.09	12.85	33.45	33.53	23.72	12.46	30.55	0.00
0.6000	27.81	17.49	5.69	16.32	31.54	21.4	9.76	26.01	31.39	21.26	9.64	23.8	0.00
Dataset #14, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.3500	-105.78	-135.24	-168.92	322.75	22.33	11.4	-1.06	8.69	21.04	10.28	-1.94	5.7	297.88
0.4000	-13.23	-29.44	-47.97	29.37	24.13	13.46	1.29	11.53	22.8	12.29	0.34	8.33	51.95
0.4500	35.86	26.68	16.18	42.48	27.94	17.8	6.24	19.52	26.94	16.99	5.69	16.8	17.28
0.4650	44.07	35.6	25.84	76.85	32.4	22.34	10.79	28.24	31.7	21.86	10.6	26.52	3.30
0.4750	44.77	35.86	25.51	69.57	33.76	23.26	11.09	27.56	33.1	22.83	10.99	26.02	1.18
0.4850	43.92	35.89	26.71	87.16	33.48	24.12	13.45	36.75	32.9	23.77	13.38	35.66	0.31
0.5000	40.71	32.23	22.52	66.74	32.51	23.02	12.19	33.21	31.95	22.69	12.16	32.06	0.01
0.5250	34.84	25.51	14.84	38.24	30.99	21.29	10.22	28.12	30.39	20.91	10.14	26.65	0.00
0.5750	34.72	11.95	-18.76	1.43	38.89	17.98	-10.08	5.58	38.04	17.57	-9.67	3.62	0.00
0.6500	3.78	-9.99	-25.74	4	25.97	15.56	3.68	15.04	25.78	15.67	4.19	14.08	0.00
0.7000	-5	-20.03	-37.21	14.7	23.95	13.26	1.06	11.23	24.29	13.98	2.27	10.99	0.00

Table A.8: Results for Datasets #13 and 14.

Dataset #15, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.2500	-73.36	-100.19	-131.17	183.73	21.8	10.34	-2.8	7.03	21.41	9.76	-3.61	4.36	79.40
0.3000	11.74	-1.92	-17.69	0.14	23.22	11.97	-0.93	8.86	22.87	11.44	-1.68	6.11	14.53
0.3250	31.4	20.79	8.53	20.71	24.11	12.99	0.24	10.18	23.87	12.59	-0.36	7.49	7.71
0.3350	36.46	26.63	15.27	36.56	26	15.15	2.72	13.4	25.72	14.71	2.07	10.47	6.09
0.3500	45.85	37.47	27.79	84.05	32.4	22.49	11.13	29.39	32.35	22.32	10.81	26.38	1.22
0.3650	45.1	36.6	26.79	79.22	32.59	22.71	11.39	30.02	32.39	22.37	10.86	26.5	0.16
0.3750	45.27	36.8	27.02	80.29	33.36	23.6	12.4	32.58	33.2	23.3	11.93	29.08	0.01
0.4250	37.81	28.19	17.08	41.82	31.36	21.3	9.77	26.23	31.15	20.95	9.24	22.86	0.00
0.5000	27.27	16.02	3.02	11.62	28.96	18.55	6.61	19.77	28.72	18.16	6.03	16.6	0.00
0.5500	61.28	55.28	48.36	246.9	26.94	16.23	3.95	15.23	26.91	16.08	3.64	12.7	0.00
Dataset #16, Improvement measurements													
t	Time				#NV				#CC				FBS
	t-distribution			F-value	t-distribution			F-value	t-distribution			F-value	
	UL	Mean	LL		UL	Mean	LL		UL	Mean	LL		
0.2500	-112.67	-139.44	-169.57	428.21	20.53	10.72	-0.3	9.5	19.45	9.91	-0.77	6.85	34.12
0.3000	16.87	5.51	-7.39	1.55	22.11	11.69	-0.13	9.73	21.48	11.39	0.01	7.91	8.75
0.3250	32.87	24.42	14.91	44.04	22.29	12.7	1.93	12.76	21.68	12.4	2.03	11.05	5.10
0.3350	40.55	28.75	14.6	27.65	26.58	12.3	-4.76	6.3	25.86	12.03	-4.38	4.43	3.03
0.3500	45.51	34.76	21.87	44.34	32.37	19.28	3.67	13.58	31.76	19.1	4.09	12.25	0.44
0.3650	42.63	35.4	27.27	107.28	29.15	20.41	10.59	32.39	28.67	20.22	10.77	32.14	0.01
0.3750	42.18	34.9	26.71	103.53	29.27	20.54	10.73	32.81	28.67	20.22	10.76	32.13	0.00
0.4000	38.83	31.13	22.46	78.11	28.7	19.9	10.02	30.72	28.13	19.62	10.1	30.05	0.00
0.4500	30.67	21.95	12.12	34.48	26.34	17.25	7.04	22.91	25.83	17.04	7.21	21.98	0.00

Table A.9: Results for Datasets #15 and 16.

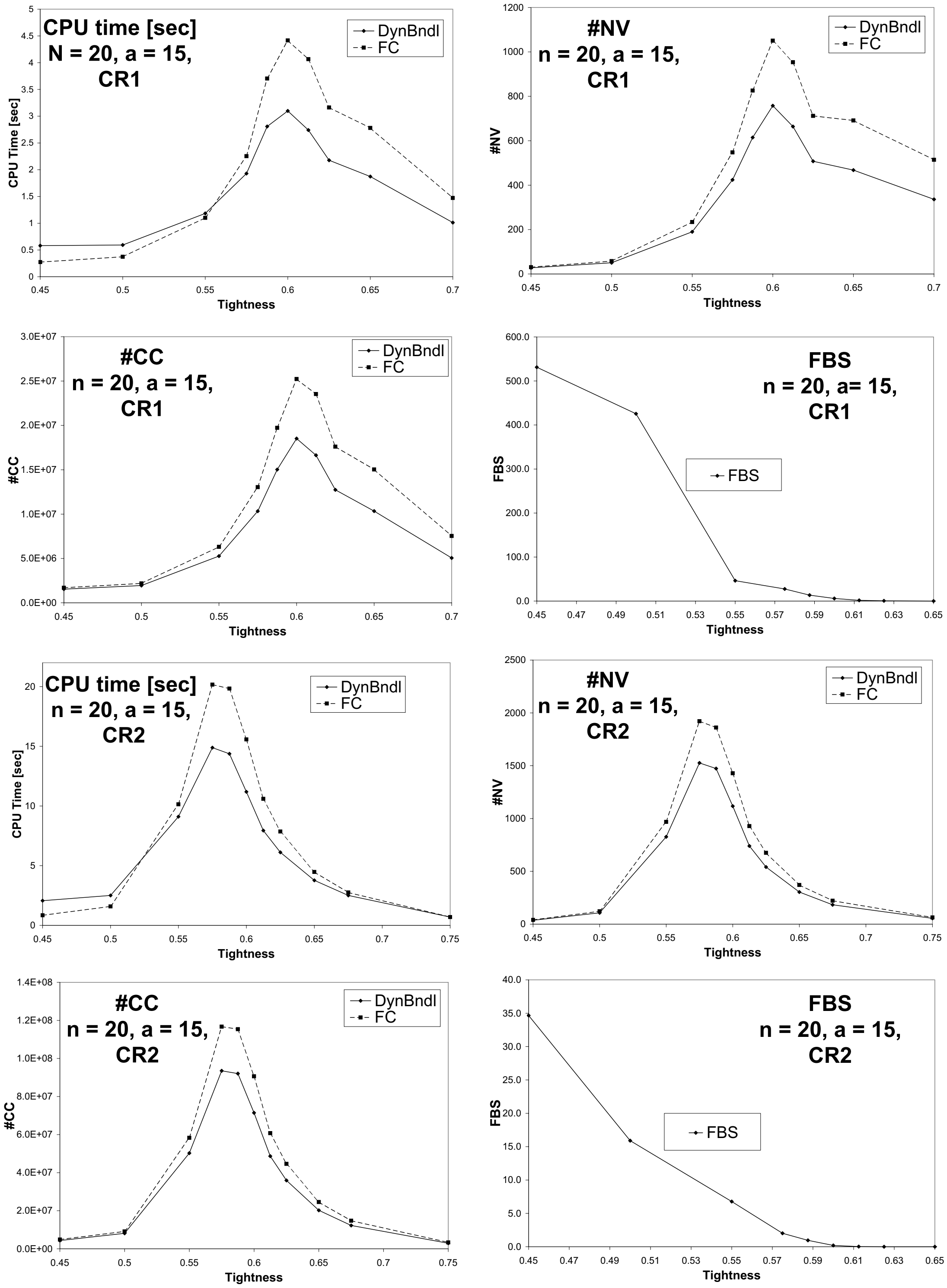

Figure A.1: CPU time, CC, NV, FBS results for n = 20, a = 15, CR1 and CR2.

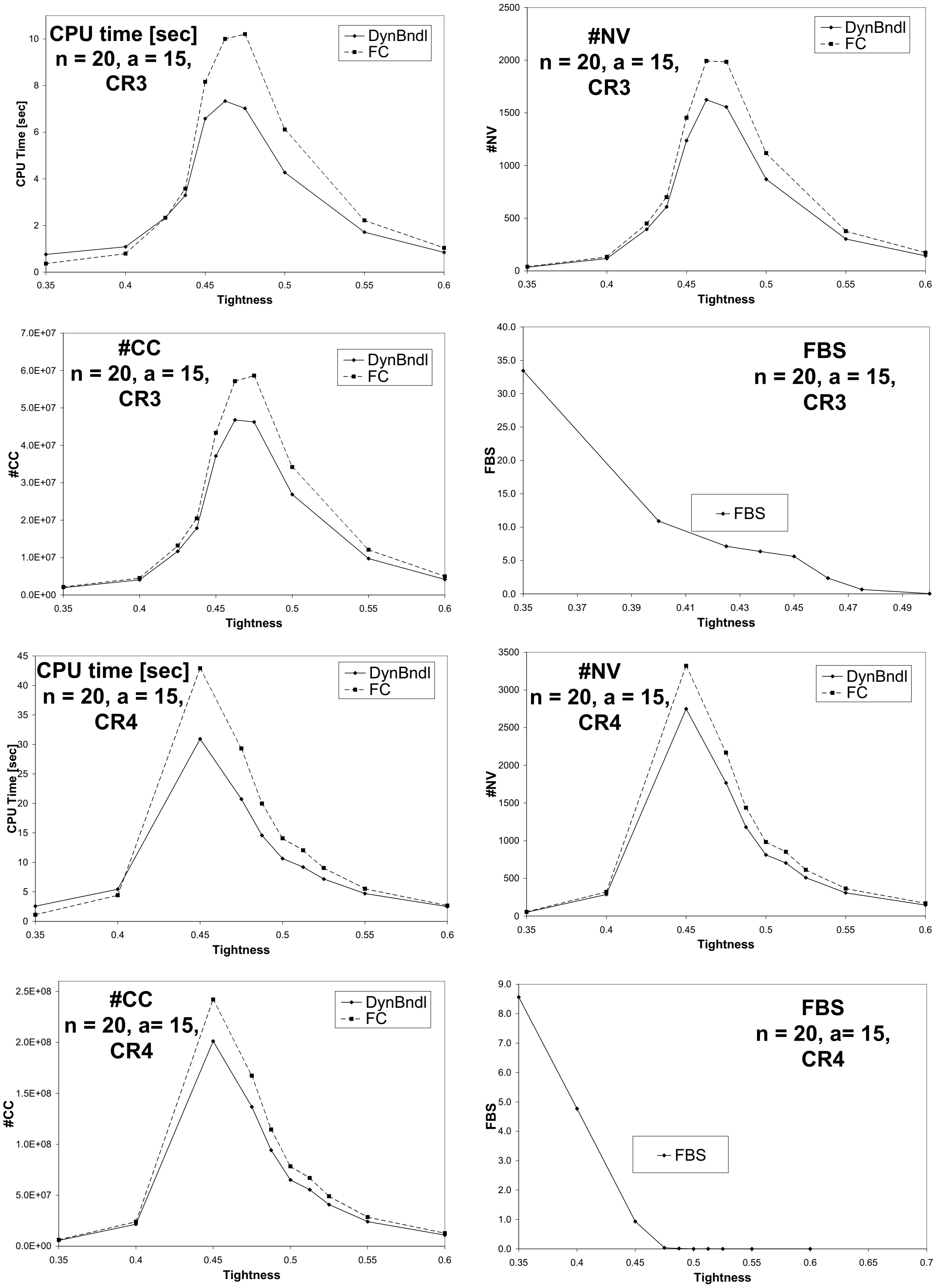

Figure A.2: CPU time, CC, NV, FBS results for n = 20, a = 15, CR3 and CR4.

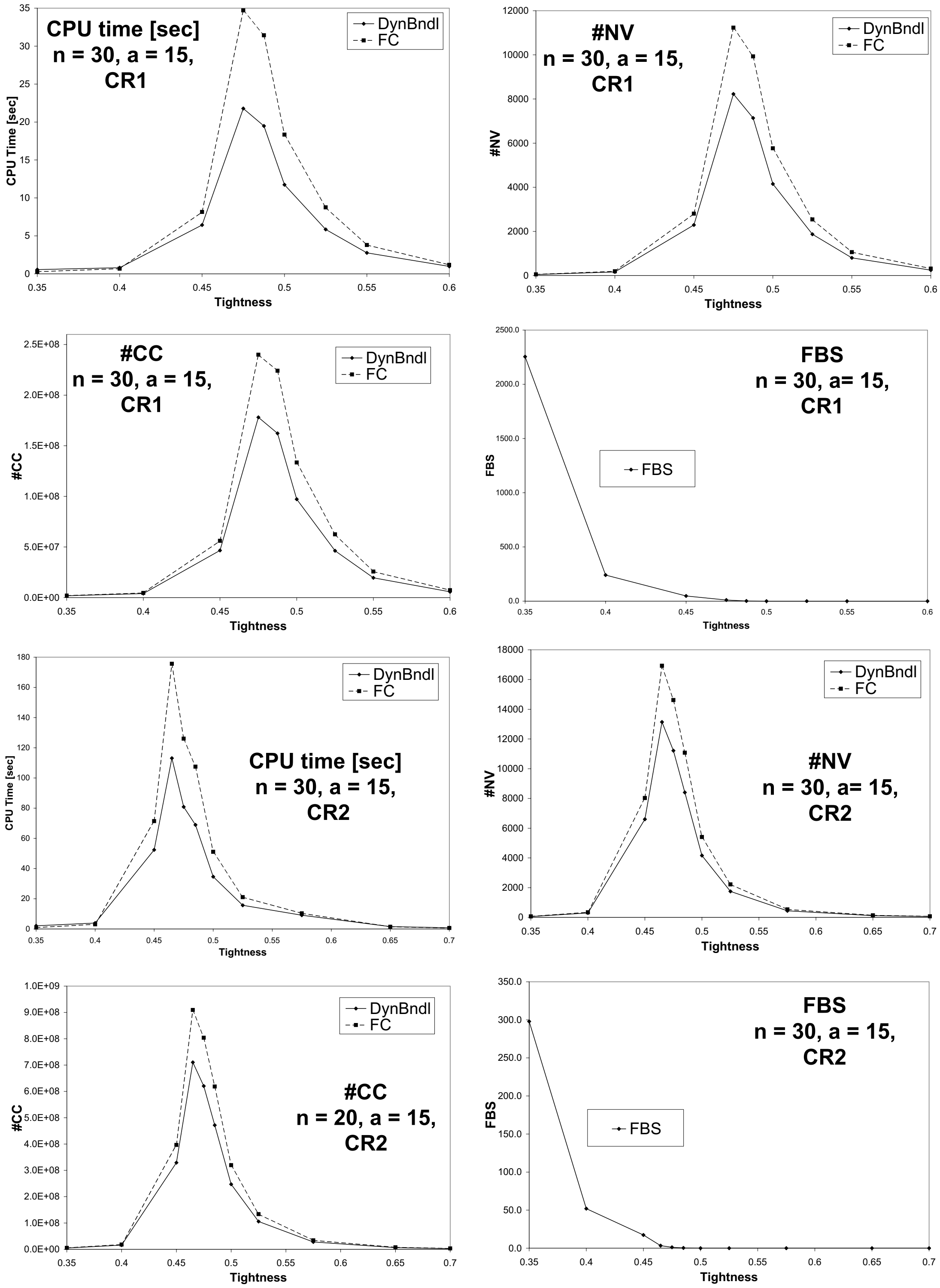

Figure A.3: CPU time, CC, NV, FBS results for n = 30, a = 15, CR1 and CR2.

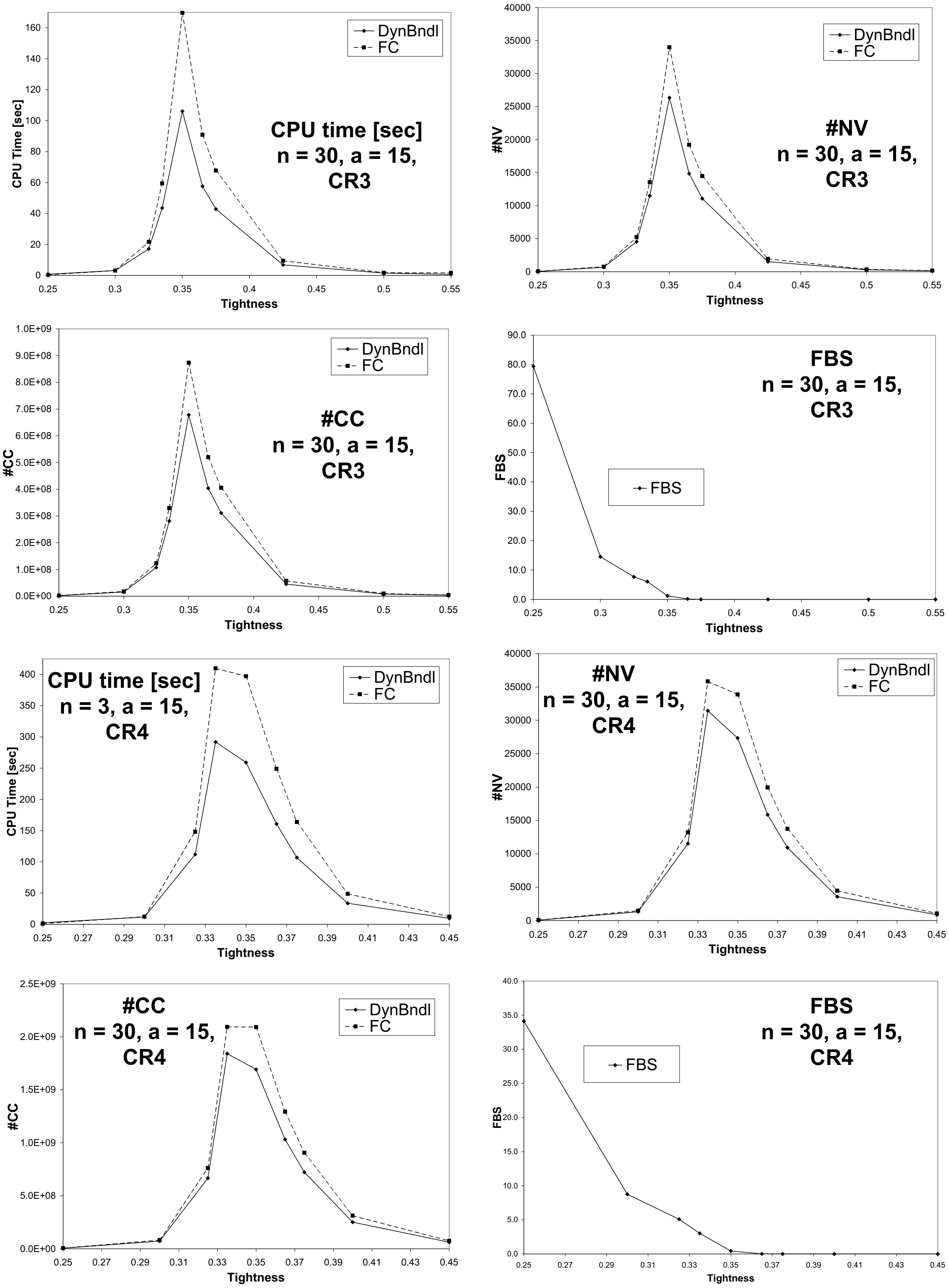

Figure A.4: CPU time, CC, NV, FBS results for n = 30, a = 15, CR3 and CR4.

Appendix B

Alternative Approaches to Computing NI Sets

Despite the theoretical guarantees and empirical evidence, there is always room for improvement in the implementation of any mechanism. In this appendix, we discuss two alternative approaches to implementing the mechanism for computing NI sets. We describe these approaches, discuss their performance, and justify our reasons for not adopting them.

B.1 Using the DT and nb-DT jointly to compute interchangeability

In this section, we discuss whether it is worth building a unique DT for all the binary constraints that apply to a variable, and one nb-DT for each of the non-binary constraints, in order to partition the domain of a variable V. The possible advantage of using a unique DT for all the binary constraints is avoiding the operation of intersecting the annotations for binary constraints.

Figure B.1 shows a CSP variable V and its neighborhood. Neighboring variables can be

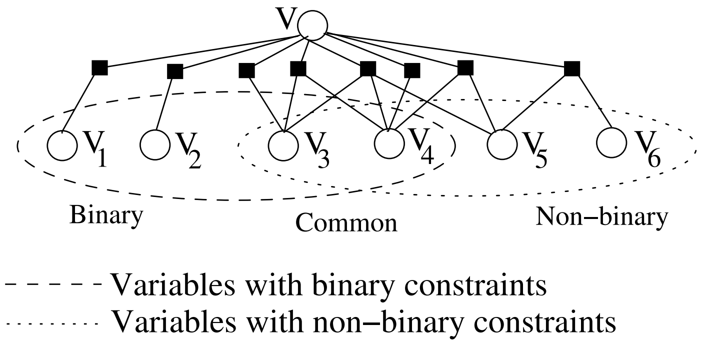

Figure B.1: The neighborhood of variable V.

in the scope of either binary constraints (i.e., V_1, V_2, V_3, and V_4) or non-binary constraints (i.e., V_3, V_4, V_5, and V_6) or both (i.e., V_3 and V_4). In this approach, we treat the variables common to both binary and non-binary constraints in a special way. The bundles of D_V in the presence of only non-binary constraints are computed using nb-DTs and those for binary are computed using one DT. We then intersect the bundles from the DT and nb-DTs to obtain the final partition of the domain.

The problem with this approach lies in computing the consistent values in the domains of the future variables with the partitions of V's domain. The updated domain of a variable that lies in the scope of both binary and non-binary constraints is derived from the nb-DTs and DT by collecting and then intersecting the consistent values from both these trees. To enable computing the updated domains of the common variables, we add additional steps for classifying variables and intersecting paths in the implementation. Our preliminary experiments show that the the gains from savings in intersection of annotations are lost in these additional steps and on an average this approach proves to be more time consuming than the implementation that builds one nb-DT per constraint defined on V.

B.2 Using a single nb-DT for all constraints to compute domain partitions

The rationale behind this approach is that by using a unique (combined) nb-DT for all the constraints defined on a variable, we may be able to save on the additional data structures and the management of separate nb-DTs. Consider the CSP of Figure B.2, where NEIGHBORS(V)=$\{V_1, V_2, V_3, V_4, V_5\}$, SCOPE($C_1$) $\cap$ SCOPE(C_2) =$\{V, V_1\}$, and SCOPE(C_2) $\cap$ SCOPE(C_3) =$\{V, V_3\}$. For every value in the domain of V, the combined nb-DT processes

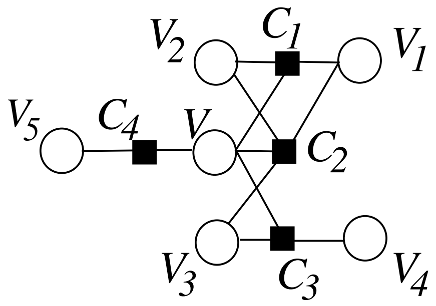

Figure B.2: Non-binary CSP.

consistent tuples with the domain value from each of the constraint as follows. Initially, the nb-DT is empty and for the first value (say a) we iteratively collect the consistent tuples from each constraint. Figure B.3 shows such a branch of the combined nb-DT for $a \in D_V$. When processing the subsequent domain values, the consistent constraint-tuples are added

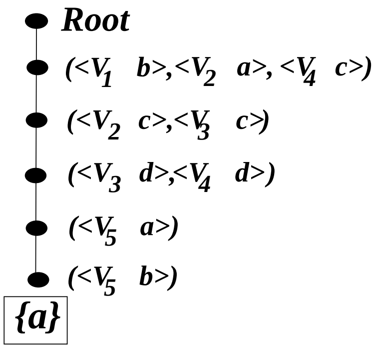

Figure B.3: Branch of a combined nb-DT.

to the tree as follows. We check if any children of the current node of the nb-DT are from the same constraint as the tuple, if there is one then we compare the two tuples. If the two tuples match, the tree building algorithm moves to that node. Otherwise, the algorithm creates a new node.

We now discuss the issues that indicate that a combined nb-DT for non-binary constraints is not a feasible option.

1. To accomodate tuples from constraints with different scopes and arities in a single nb-DT we additionally check for every node in the nb-DT whether it is comparable with a tuple from the current constraint. This additional processing can prove expensive given that the number of tuples in non-binary constraints is significantly greater than in binary constraints.

   Using the length of the tuples to detect un-comparable tuples quickly is a good heuristic. However it is ineffective when some constraints have the same arity. For our example, the heuristic will not work for tuples from C_2 and C_3. Therefore, we often check for the variables present in the tuples of a combined nb-DT.

2. Once search instantiates V, the nb-DT is used to determine the domain of the future neighborhood of V consistent with the instantiation of V (see Section 3.1.2.2). Determing the consistent domain of a neighbor using a combined nb-DT is cumbersome and expensive. Given a future variable $V_f \in$ NEIGHBOR(V), we first identify all constraints C such that $\{V, V_f\} \subset$ SCOPE(C). Next, we collect the consistent values of V_f due to each constraint C. To do so, for every node along a path of the nb-DT, we determine the constraint the node belongs to and extract the value for V_f. The intersection of these consistent values results in the domain of V_f consistent with the current instantiation of V. This operation involves a minimum of one additional comparison at every node in the path to an annotation (which can be long for

non-binary constraints) thereby, making the whole process prohibitively expensive.

3. The combined nb-DT does not indicate the constraint from which the tuple in each node comes and this can lead to errors. Consider the scenario when V_2 and V_4 have been instantiated. Figures B.4 and B.5 show partially the definitions[1] of C_2 and C_3 used by the combined nb-DT. In search, the nb-DT is created only for the future neighborhood of V. From the definition of NI and by looking at Figures B.4 and B.5,

V_p	V	V_3
Z	x	a
Z	x	b
Z	x	c
Z	y	a
Z	y	b
Z	y	c
Z	y	d

Figure B.4: Partial constraint C_2.

V_p	V	V_3
Z	x	d
Z	x	e
Z	y	e

Figure B.5: Partial constraint C_3.

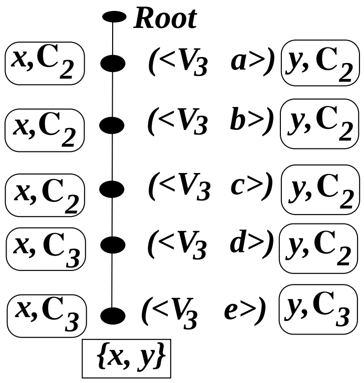

Figure B.6: A combined DT branch that incorrectly detects interchangeable values.

we know that x, y are not interchangeable values in D_V. However, in the combined nb-DT, the paths to annotations containing x and y respectively will have the same set of tuples as shown in Figure B.6. This figure also shows, for both values x and y, the constraint used at every step in building the path to their respective annotations. Observing the source of the tuples in Figure B.6, we can clearly identify the source of the error. The node $\langle V\ d \rangle$ in the path to the annotation of x originates from constraint C_3, and that for y is from C_2. Hence, the two should not be compared. However, in a combined nb-DT, there is no mechanism to detect the source of such tuples.

Thus, we need a mechanism to efficiently record and utilize the origin of each tuple. Even if we record the originating constraint of each tuple, the algorithm will still perform

[1]The definitions were updated using Equation (2.1).

an additional check at each node and the computation of future domains will remain cumbersome. In conclusion, this approach too is unlikely to improve performance in practice.

Appendix C

Documentation of the Implementation of DynBndl & FC

This chapter documents the code to implement DynBndl and FC. We first present the directory structure of the source code, and then give detailed documentation of important parts of the code. The code we present here has been initially developed by Davis and Choueiry and has been used in the following publications: [Beckwith *et al.*, 2001; Davis, 2002; Choueiry and Davis, 2002]. We have reused a substantial amount of the previously generated code and have made modifications and additions to implement the contributions of this document.

C.1 Directory structure

The directory structure of the source code is as shown below:

```
--src--+
        |-- make-nb.lisp
        |-- definitions----+
                           |-- csp-problem.lisp
                           |-- csp-var.lisp
                           |-- csp-value.lisp
                           |-- constraint.lisp
                           |-- package.lisp
        |-- search---------+
                           |-- dnpi-fc.lisp
                           |-- nb-fc.lisp
                           |-- driver.lisp
        |--interchange-----+
                           |-- nd-dt3.lisp
                           |-- nb-dt-definitions.lisp
        |-- utils
        |-- files-list
        |-- problems-------+
                           |-- nb-random.lisp
```

The file `make-nb.lisp` files is the root file to build the complete source tree. The list of files to compile and load are read from the files stored in the `files-list` directory. The `utils` directory has code for various utility functions used in the code. The file `problems/nb-random.lisp` defines the method `nb-random` for creating a CSP instance by reading a CSP definition file generated by our random generator. In the following sections we discuss the code to solve a CSP instance using DynBndl and FC.

C.2 Detailed documentation

This section first gives an introduction to the code defining the data-structures. It then goes on to describe how the basic search mechanism is implemented and then details the code for DynBndl and FC.

C.2.1 Basic data-structures

The CSP instance generated by the method `nb-random` is an object of class `csp-problem` defined in `definitions/csp-problem.lisp`. The `csp-problem` object uses many of the data structures and methods defined in the following files of the `definitions/` folder:

- `csp-var.lisp`,
- `csp-value.lisp`, and
- `constraint.lisp`.

The file `constraint.lisp` defines a hierarchy of constraint types. For our experiments using randomly generated CSPs we used the constraint class `relation` which is a subclass of `explicit-constraint`. `explicit-constraint` in turn is derived from `generic-constraint`.

C.2.2 Basic framework for search

We now present a high-level view of the code to show how a generic search is implemented in our code. The method `solve` defined in `search/driver.lisp` is where the process of solving a CSP begins. `solve` is characterized as follows:

Input: The arguments given to `solve` are:

- `problem`, a CSP instance of the type `csp-problem`. The CSP to solve.
- `ordering`, a string indicating the type of variable ordering to use for solving the CSP. The possible values for it are `dld` for Dynamic Least Domain, `sld` for Static Least Domain, and others. All our experiments used `dld`.
- `bundling`, an optional argument specifying the type of bundling to use, if at all. Some of the correct inputs are `none` for no bundling, `dnpi` for dynamic bundling. No bundling is the default value.
- `find-solutions`, an optional argument and it specifies the number of solutions to find. There are two valid input: `1` and `all` for finding one and all solutions, respectively. The default value is to find one solution.

Output: The output of the method is a reference to the CSP instance after search has finished processing it.

Processing: The method `solve` initializes the `problem` object and search by clearing all the solutions, resetting all metrics to their initial values and by performing other data-structure management. The method `solve` also implements the basic search mechanism using label and unlabel methods. Depending on the the input parameters such as `bundling` and `ordering` it invokes the appropriate label and unlabel methods. For example, if `bundling` is passed the value `dnpi` the label-method invoked is `dnpi-fc-label`. The label and unlabel methods for all types of search are defined in files placed under the `search` directory. It also binds the `next-var` function based on `ordering`.

A typical label method is characterized as follows:

Input: `problem` and `level`. The `level` or depth at the which the search process is currently in.

Output: The new `level` of search.

Processing: It takes a variable (returned by `next-var`), and iterates through it's domain, assigning a value and forward checking for that value, until one is found to be consistent i.e. no future domain is annihilated by that assignment. If no such value is found, it sets `consistent` to false, and returns the current `level`. It makes sure to undo any effects that this assignment would have had. If an acceptable value is found, then that assignment is carried out, the problem set to consistent, and the `level` incremented.

C.2.3 Implementation of DynBndl

We set the `bundling` parameter of `solve` to `dnpi` to perform search using DynBndl. As mentioned before, we set `ordering` to be `dld` in our experiments. `solve` binds the methods `dnpi-fc-label` and `dnpi-fc-unlabel` as the label and unlabel methods, respectively. Both these methods are defined in the file `search/dnpi-fc.lisp`. Let us look at `dnpi-fc-label` in more detail:

Input: `problem` and `level`. The `level` or depth at the which the search process is currently in.

Output: The new `level` of search.

Processing: In addition to the typical processing of a label method it partitions the domain of the current variable unless it has not already partitioned. The method invoked to partition the domain of the current variable is `partition-domain+fc` defined in `interchange/nb-dt3.lisp`.

We now discuss the method `partitioned-domain+fc` because this method is central to the implementation of DynBndl:

Input: The variable whose domain is to be partitioned.

Output: An internal data-structure called `path+annotation` representing the partitioned domain and the values in the input variable's neighborhood that each partition is consistent with,

Processing: `partition-domain+fc` is the method that implements Process 1 and Process 2 of DynBndl described in Section 3.1.2. It implements the algorithm for building an nb-DT using the method `nb-dt` (see Algorithm 2) defined in the same file. It also controls the switching on and off domain partitioning as described in section 3.1.2.3. And finally it implements the algorithm to intersect all the nb-DTs using the method `intersect-part` (see Algorithm 3) defined in the same file.

C.2.4 Implementation of FC

We set the `bundling` parameter of `solve` to `none` to perform search using DynBndl. As mentioned before, we set `ordering` to be `dld` in our experiments. `solve` binds the methods `nb-fc-label` and `nb-fc-unlabel` as the label and unlabel methods, respectively. Both these methods are defined in the file `search/nb-fc.lisp`. Let us look at `nb-fc-label` in more detail:

Input: `problem` and `level`. The `level` or depth at the which the search process is currently in.

Output: The new `level` of search.

Processing: This method implements our improved selection-projection based method for non-binary FC described in Section 2.5.3. The main helper methods used by it is `build-constraint-defs-from-past` which computes and stores partial definitions constraints for future use.

Appendix D

Documentation of the Implementation of the Join Algorithm

This chapter documents the code to implement the dynamic bundling based join algorithm. We first present a brief introduction to the XXL library, followed by a listing of the source code files. We then document the main data-structures and functionalities of the code.

D.1 XXL library

XXL is a Java library that contains a rich infrastructure for implementing advanced query processing functionality [den Bercken *et al.*, 2001]. XXL provides a demand-driven cursor algebra, a framework for indexing and a powerful package for supporting aggregation. The library is publicly available under GNU LGPL and comes with a full documentation. The documentation is available at

`http://dbs.mathematik.uni-marburg.de/Home/Research /Projects/XXL/Documentation`.

We used version 1.0 of the library in our implementation.

Cursors are one of the basic components of the XXL library. We use the concept of cursors extensively in our code. A cursor is an abstract mechanism to access objects within a stream. Cursors in XXL are independent from the specific type of the underlying objects. The interface of a cursor is given by

```
interface Cursor extends java.util.Iterator {
        Object peek();
        void update(Object o);
        void reset();
        void close();
}
```

A cursor extends the functionality of the iterator provided in the package java.util. The peek method reports the next object of the iteration without changing the state of the iteration. A call of reset sets the cursor to the beginning of the iteration. The method close stops the iteration and releases resources like file handles. The method update modifies the current object of the iteration. XXL offers an algebra for processing cursors, i. e. there are a set of operations that require cursors as input and return a cursor as output.

D.2 Source code documentation

In this section, we list the layout of the source tree and give detailed documentation of important segments of the code.

D.2.1 Source files

The source files are listed below:

```
--xxl---+--App--+
                |-- MyJoin.java
                |-- SortJoin.java
                |-- MergeJoin.java
                |-- MyJoin.java
                |-- Inter.java
                |-- PipeCursor.java
                |-- Bundle.java
                |-- Context.java
```

The file `MyJoin.java` defines the class representing the new bundling-based join algorithm. The classes `SortJoin` and `MergeJoin` represent the two steps of the join algorithm (see algorithm in Chapter). `PipeCursor` is a class defining an extended Cursor. `Inter` implements the algorithm to generate the next bundle of the current variable. The files `Bundle.java` and `Context.java` define the data-structures used in the code.

D.2.2 Data structures and bundle computation

The data-structures used in the implementation are as follows:

1. `PartitionValuePair` is a data-structure that represents a partition of a constraint. It stores the partition identifier and the value associated with the partition. It is defined in `Bundle.java`.

2. `Bundle` represents a bundle assigned to a variable during join computation. It stores the identifier of the bundle's variable, a linked list of `PartitionValuePair` objects and a boolean flag indicating whether the bundle has been checked. It is defined in `Bundle.java`.

3. Solution represents a solution bundle to the CSP or one tuple of the join query. It stores a list of Bundle objects and also maintains the number of solutions actually present in the solution bundle. It is defined in Bundle.java.

4. Context is a data-structure that maintains the state of the join. It maintains the *Processed-Values*, past instantiations and the last solution of the join. It is defined in Context.java.

The class Inter defines methods to compute the next bundle for the variable V_c from a constraint. Inter maintains the *Current-Constraint* data-structure and implements Algorithm 7 to compute the next bundle. It also provides method to perform backtracking when an instantiation fails.

D.2.3 Implementation of sorting phase

The sorting phase is implemented by the class SortJoin coded in the file SortJoin.java. An object of this class is used by MyJoin to initiate processing the join query. SortJoin implements the in-memory join algorithm described in Algorithm 8 in a private method of the class get_one_solution(int no_of_vars). The method next() computes the next query-result tuple (solution) when invoked. It returns null when no more solutions are possible from the sorting phase of the join. SortJoin uses Inter to compute interchangeable values. The data-structures defined in D.2.2 are also heavily used by this class. Further, SortJoin stores the sorted runs in temporary files, which are later used by the merging phase.

D.2.4 Implementation of merging phase

The merging phase is implemented by the class MergeJoin coded in the file MergeJoin.java. An object of this class is used by MyJoin after no further results are possible from

`SortJoin`. This class implements the merging phase described in Section 5.5.2.2. The method `next()` computes the next query-result tuple (solution) when invoked and when `next()` has no solutions to return it signals the end of join processing.

Bibliography

[Achlioptas *et al.*, 1997] Achlioptas, Dimitris; Kirousis, Lefteris M.; Kranakis, Evangelos; Krizanc, Danny; Molloy, Michael S.O.; and Stamatiou, Yanning C. 1997. Random Constraint Satisfaction: A More Accurate Picture. In *Principles and Practice of Constraint Programming, CP'97. Lecture Notes in Artificial Intelligence 1330*. Springer Verlag. 107–120.

[Bacchus and van Beek, 1998] Bacchus, Fahiem and Beek, Petervan 1998. On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems Using the Hidden Variable Method. In *Proc. of AAAI-98*, Madison, Wisconsin. 311–318.

[Bayardo, 1996] Bayardo, Roberto J. 1996. *Processing Multi-Join Queries*. Ph.D. Dissertation, University of Texas, Austin.

[Beckwith *et al.*, 2001] 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.*, volume 2256 of *LNAI*, Adelaide, Australia. Springer. 50–61.

[Benson and Freuder, 1992] Benson, Brent W. and Freuder, Eugene C. 1992. Interchangeability Preprocessing Can Improve Forward Checking Search. In *Proc. of the 10 th ECAI*, Vienna, Austria. 28–30.

[Bernstein and Chiu, 1981] Bernstein, Philip A. and Chiu, Dah-Ming W. 1981. Using semi-joins to solve relational queries. *J. ACM* 28(1):25–40.

[Bessière *et al.*, 2002] Bessière, Christian; Meseguer, Pedro; Freuder, Eugene C.; and Larrosa, Javier 2002. On Forward Checking for Non-binary Constraint Satisfaction. *Artificial Intelligence* 141 (1-2):205–224.

[Ceri and Widom, 1991] Ceri, Stefano and Widom, Jennifer 1991. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc. 577–589.

[Chakrabarti *et al.*, 2004] Chakrabarti, Kaushik; Chaudhuri, Surajit; and Hwang, Seungwon 2004. Automatic categorization of query results. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM Press. 755–766.

[Cheeseman *et al.*, 1991] 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.

[Chen *et al.*, 2001] Chen, Zhiyuan; Gehrke, Johannes; and Korn, Flip 2001. Query optimization in compressed database systems. In *2001 ACM International Conference on Management of Data (SIGMOD)*. 271–282.

[Choueiry and Davis, 2002] Choueiry, Berthe Y. and Davis, Amy M. 2002. Dynamic Bundling: Less Effort for More Solutions. In Koenig, Sven and Holte, Robert, editors 2002, *5th International Symposium on Abstraction, Reformulation and Approximation (SARA 2002)*, volume 2371 of *Lecture Notes in Artificial Intelligence*. Springer Verlag. 64–82.

[Choueiry and Noubir, 1998] 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. Revised version KSL-98-24, `ksl-web.stanford.edu/KSL_Abstracts/KSL-98-24.html`.

[Davis, 2002] Davis, Amy 2002. Dynamically Detecting and Exploiting Symmetry in Finite Constraint Satisfaction Problems. Master's thesis, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE.

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

[Dechter, 1990] Dechter, Rina 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence* 41:273–312.

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

[den Bercken *et al.*, 2001] Bercken, Jochen Vanden; Blohsfeld, Björn; Dittrich, Jens-Peter; Krämer, Jürgen; Schäfer, Tobias; Schneider, Martin; and Seeger, Bernhard 2001. Xxl - a library approach to supporting efficient implementations of advanced database queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc. 39–48.

[Dittrich *et al.*, 2003] Dittrich, Jens-Peter; Seeger, Bernhard; Taylor, David Scot; and Widmayer, Peter 2003. On producing join results early. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*. ACM Press. 134–142.

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

[Glaisher, 1874] Glaisher, J.W.L. 1874. On the Problem of the Eight Queens. *Philosophical Magazine, series 4* 48:457–467.

[Gray, 2004] Gray, Jim 2004. The next database revolution. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM Press. 1–4.

[Gryz *et al.*, 2004] Gryz, Jarek; Guo, Junjie; Liu, Linqi; and Zuzarte, Calisto 2004. Query sampling in db2 universal database. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM Press. 839–843.

[Gupta and Mumick, 1995] Gupta, Ashish and Mumick, Inderpal Singh 1995. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing* 18(2):3–18.

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

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

[Hubbe and Freuder, 1989] 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.

[Kondrak and van Beek, 1995] 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.

[Lal and Choueiry, 2003] Lal, Anagh and Choueiry, Berthe Y. 2003. Dynamic Detection and Exploitation of Value Symmetries for Non-Binary Finite CSPs. In *Third International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'03)*, Kinsale, County Cork, Ireland. 112–126.

[Lal and Choueiry, 2004] Lal, Anagh and Choueiry, Berthe Y. 2004. Constraint Processing Techniques for Improving Join Computation: A Proof of Concept. In *Proceedings of the 1^{st} International Symposium on Constraint Databases (CDB 04)*, volume 3074 of *LNCS*. Springer. 149–167.

[Lal *et al.*, 2003] Lal, Anagh; Choueiry, Berthe Y.; and Zou, Hui 2003. A Generator for Solvable Random Non-Binary Finite Constraint Satisfaction Problems. consystlab.unl.edu.

[Lal *et al.*, 2004] Lal, Anagh; Choueiry, Berthe Y.; and Freuder, Eugene C. 2004. Neighborhood Interchangeability and Dynamic Bundling for Non-Binary Finite CSPs. In *Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming (CSCLP 04)*, Lausanne, Switzerland. 114–130.

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

[Lim *et al.*, 2004] Lim, Ryan; Guddeti, Venkata Praveen; and Choueiry, Berthe Y. 2004. An Interactive System for Hiring and Managing Graduate Teaching Assistants. In *Conference on Prestigious Applications of Intelligent Systems (ECAI 04)*, Valencia, Spain. 730–734.

[Mamoulis and Papadias, 1998] Mamoulis, Nikos and Papadias, Dimitris 1998. Constraint-based algorithms for computing clique intersection joins. In *Proceedings of the sixth ACM international symposium on Advances in geographic information systems*. ACM Press. 118–123.

[Miranker *et al.*, 1997] Miranker, Daniel P.; Bayardo, Roberto J.; and Samoladas, Vasilis 1997. Query evaluation as constraint search; an overview of early results. In Gaede, Volker; Brodsky, Alexander; Günther, Oliver; Srivastava, Divesh; Vianu, Victor; and Wallace, Mark, editors 1997, *Second International Workshop on Constraint Database Systems (CDB '97)*, volume 1191 of *LNCS*. Springer. 53–63.

[Neagu and Faltings, 2001] Neagu, Nicoleta and Faltings, Boi 2001. Exploiting Interchangeabilities for Case Adaptation. In *International Conference on Case-Based Reasoning (ICCBR 01)*, volume 2080 of *LNCS*, Vancouver, British Columbia, Canada. Springer. 422–436.

[Rees, 2001] Rees, D.G 2001. *Essential Statistics*. Chapman and Hall.

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

[Revesz, 2005] Revesz, Peter 2005. Personal communication.

[Rich *et al.*, 1993] Rich, Christian; Rosenthal, Arnon; and Scholl, Marc H. 1993. Reducing duplicate work in relational join(s): A unified approach. In *International Conference on Information Systems and Management of Data*. 87–102.

[Rossi *et al.*, 1990] 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.

[Roth and Horn, 1993] Roth, Mark A. and Horn, Scott J. Van 1993. Database compression. *SIGMOD Record* 22(3):31–39.

[Sabin and Freuder, 1994] Sabin, Daniel and Freuder, Eugene C. 1994. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of the 11 th ECAI*, Amsterdam, The Netherlands. 125–129.

[Sabin and Freuder, 1997] Sabin, Daniel and Freuder, Eugene C. 1997. Understanding and Improving the MAC Algorithm. In *Principles and Practice of Constraint Programming, CP'97. Lecture Notes in Artificial Intelligence 1330*. Springer Verlag. 167–181.

[SymCon, 2003] SymCon, 2003. The 3rd International Workshop on Symmetry and Constraint Satisfaction Problems. http://scom.hud.ac.uk/scombms/SymCon03/.

[SymCon, 2004] SymCon, 2004. The 4th International Workshop on Symmetry and Constraint Satisfaction Problems. http://www.dis.uu.se/SymCon04/.

[Wallace *et al.*, 1995] Wallace, Mark; Bressan, Stéphane; and Provost, Thierry Le 1995. Magic checking: Constraint checking for database query optimization. In *CDB 1995*. 148–166.

[Weil and Heus, 1998] Weil, Georges and Heus, Kamel 1998. Eliminating Interchangeable Values in the Nurse Scheduling Problem Formulated as a Constraint Satisfaction Problem. In *Workshop on Constraint-based reasoning in conjunction with FLAIRS'95*, Indianlantic, FL. Available from www.sci.tamucc.edu/constraint95/kamel.ps.

[Westmann *et al.*, 2000] Westmann, Till; Kossmann, Donald; Helmer, Sven; and Moerkotte, Guido 2000. The implementation and performance of compressed databases. *SIGMOD Record* 29(3):55–67.

[Yang, 2003] Yang, Zheying (Jane) 2003. An Empirical Study of the Performance of Preprocessing and Lookahead Techniques for Solving Binary Constraint Satisfaction Problems. Master's thesis, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE.