University of Alberta

Library Release Form

Name of Author: Xinguang Chen

Title of Thesis: A theoretical comparison of selected CSP solving and modeling techniques

Degree: Doctor of Philosophy

Year this Degree Granted: 2000

. . . . . .

Xinguang Chen

307-7180, Lindsay Road
Richmond, B.C.
Canada, V7C 3M6

Date: . . . . . .

University of Alberta

A theoretical comparison of selected CSP solving and modeling techniques

by

Xinguang Chen

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2000

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A theoretical comparison of selected CSP solving and modeling techniques** submitted by Xinguang Chen in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

. . . . . .

Dr. Peter van Beek

. . . . . .

Dr. Randy Goebel

. . . . . .

Dr. Joe Culberson

. . . . . .

Dr. Fraser Forbes

. . . . . .

Dr. Toby Walsh

Date: . . . . . .

Abstract

A constraint programming approach to problem solving usually goes through two phases: modeling the problem as a CSP and then solving the CSP. It has been recently recognized that both choosing the right solving algorithm and the right problem model are crucial for efficient problem solving. In the past, much of the research activities in the constraint community has been concentrated on developing various improving techniques to the naive backtracking algorithm (BT). These techniques can be classified as *look-ahead* schemes and *look back* schemes. Unfortunately, it has been observed by different researchers that the enhancement of look-ahead techniques is sometimes counterproductive to the effects of look-back techniques. In this thesis, we show theoretically that the effect of the backjumping will be diminished as a backtracking algorithm is equipped with an appropriate variable ordering heuristic or a certain level of local consistency enforcement. We propose a new algorithm, named *GAC-CBJ*. In contrast to Bessière and Régin's conclusion (1996) that CBJ is useless to an algorithm maintaining arc consistency (MAC or GAC), our experiments in several problem domains show that the use of CBJ can provide significant improvements on the hard instances.

There also exists a variety of techniques to improve the quality of a CSP formulation. The dual graph transformation and hidden variable transformation are two important modeling techniques that translate a general CSP to an equivalent binary CSP. However, little has been known about how these transformations will influence the effectiveness of the CSP solving techniques. Some preliminary results include: Stergiou and Walsh (1999) study the the effectiveness of consistency techniques under the above transformations, and Bacchus and van Beek (1998) study how the two transformations will affect the performance of the forward checking algorithm (FC).

In this thesis, we present a comprehensive theoretical comparison of these two transformations for BT, FC and MAC (GAC). Among other results, we show that FC applied on the hidden problem is only bounded worse than FC applied on the dual problem, and GAC applied on the original problem visits exactly the same nodes as MAC applied on the hidden problem.

Acknowledgements

I would like to thank a great many people for making this dissertation possible. I owe many thanks to my supervisor, Dr. Peter van Beek, who has been a great source of advice and inspiration throughout the period of my study.

Thanks to my committee members, Dr. Randy Goebel, Dr. Joe Culberson, Dr. Toby Walsh and Dr. Fraser Forbes for their careful reading my thesis and many valuable comments and guidances.

Other members of the AI research group at the University of Alberta have also given me support. Thanks to my friends, Huang Guohu, Zheng Tong, and Wang Jiankang for the time we have been together.

Thanks to my parents, Chen Zhaohui and Liu Ming, for years of support and love. Last and most, thanks to my beloved wife, Xu Ke, for all her love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constraint programming (CP) is the study of computational systems based on constraints. A constraint is simply a logical relation among several unknowns or variables, each taking a value in a given domain. Constraint programming has recently emerged as a research area that combines ideas from a number of fields, including artificial intelligence, programming languages, symbolic computing, complexity theory, operations research and computational logic [47]. The most appealing characteristic is that constraint programming techniques can be more declarative and maintainable than standard imperative languages, without sacrificing efficiency. It is remarkable that, in the last ten years, constraint programming has moved from purely academic research into commercial products, *e.g.*, CHIP, ECLiPSe and Ilog Solver. Constraint programming has been successfully applied in numerous domains. Recent applications include computer graphics [12], natural language processing [97], database systems [20], scheduling and planning problems [73], and electrical circuit design [105].

In this introductory chapter, we will give a brief review of the basic elements of the constraint programming approach in problem solving, and various ways to improve problem solving efficiency with respect to the problem modeling and problem solving techniques. Then, we will explain the motivations and contributions in this work, and give an overview of the structure of the dissertation.

1.1 Constraint Programming Approach

The success of constraint programming is largely ascribed to its general applicability. Under the same problem solving framework, problems from a wide range of domains can be solved efficiently while the performance is competitive to those of specially designed software packages[1, 122]. The basic theoretical foundation for constraint

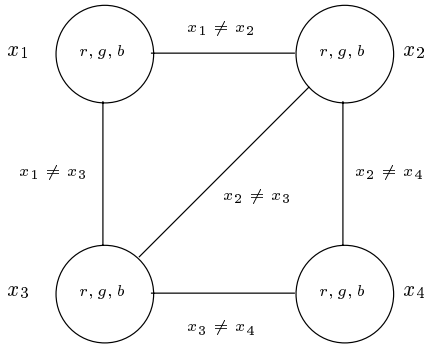


Figure 1.1: The CSP representation of a graph coloring problem

programming is a generic problem solving framework called *constraint satisfaction problems*, or CSPs. A *constraint satisfaction problem* consists of a set of variables, each associated with a domain of values, and a set of constraints. Each of the constraints is expressed as a relation, defined on some subset of the variables, denoting the consistent value assignments that satisfy the constraint. A solution to a CSP is an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied [34, 75, 77, 85].

Example 1.1 *3-SAT problems can be formulated as CSPs. Consider a 3-SAT problem with* 6 *propositions,* $x_1, \ldots, x_6$*, and* 4 *clauses,* $x_1 \vee x_3 \vee x_6$*,* $\neg x_1 \vee \neg x_3 \vee x_4$*,* $x_4 \vee \neg x_5 \vee x_6$ *and* $x_2 \vee x_4 \vee \neg x_5$*. In one CSP representation of the 3-SAT problem, there is a variable for each proposition,* $x_1, \ldots, x_6$*, each variable has the domain of values* $\{0, 1\}$*, and there is a constraint for each clause specifying the value combinations that will make the clause be true. For example, there is a constraint* $C(x_1, x_3, x_6)$ *for the first clause. The constraint can be represented implicitly, saying "*$x_1 = 1$ *or* $x_3 = 1$ *or* $x_6 = 1$*," or it can be represented explicitly by listing all valid value combinations,* $\{(0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$.

Example 1.2 *Given a graph* $G = (V, E)$ *and* k *colors, the graph coloring problem asks whether the vertices of the graph can be labelled by these colors in a way such that each pair of adjacent vertices are labelled with different colors. The graph coloring problem can be formulated as a CSP in which each vertex is given a variable, and each variable has the same domain of* k *values, denoting the* k *colors. There is a constraint for each pair of adjacent vertices such that two variables must have different values. Figure 1.1 shows the CSP representation of a graph coloring problem on a graph with* 4 *vertices and* 3 *colors.*

Since 3-SAT and graph coloring are well known NP-complete problems, in general, to find a solution for a CSP instance is NP-complete [51, 75]. Fortunately, like most combinatorial problems, some CSP instances are not so "hard" to solve in practice. For large real world applications, a carefully built CSP formulation can speed up the problem solving dramatically.

Constraint programming has become a vast field. Research interests in CSPs include problem modeling techniques, consistency inference, systematic search algorithms, heuristics, stochastic search methods, structure-driven algorithms, tractable problems, generating hard instances, over-constrained problems, and applications.

A constraint programming approach to problem solving generally goes through two phases: modeling the problem as a CSP and then solving the CSP. The modeling translates the problem description from a natural language to the language of CSPs, *i.e.*, defining variables, domains and constraints of the CSP. Having formulated the problem as a CSP, there are plenty of constraint techniques to solve the CSP. In the past, most of the research activities were concentrated on developing various constraint solving algorithms or techniques and relatively less attention was given to modeling techniques. However, recently the importance of problem modeling has been recognized and it is known that **both** choosing the right model and choosing the right constraint satisfaction algorithm are crucial for efficient problem solving.

1.1.1 Solving Constraint Satisfaction Problems

Constraint satisfaction problems are usually solved by search methods, among which the *backtracking* algorithm (BT) and its improvements are the most widely used. BT incrementally attempts to extend a partial solution that instantiates consistent values for some of the variables, towards a complete solution, by repeatedly assigning a value for an uninstantiated variable from its domain. If that value is not consistent with the values in the current partial solution, BT is able to identify that the subspace given by the Cartesian product of the domains of the uninstantiated variables does not contain a solution and thus can be pruned. The heart of the constraint satisfaction approach is to use constraints to prune the search space; a number of techniques have been developed to improve the naive backtracking algorithm by more intelligently exploiting the constraint. Improvements to the backtracking algorithm have focused on the two phases of the algorithm, looking backward [32, 48, 52, 57, 92] and looking ahead [16, 60, 85, 87, 96]. An appropriate integration of these techniques can dramatically improve the performance of the backtracking algorithm. Sometimes all

these algorithms are named as *backtracking* algorithms to emphasize their backtracking nature, while the naive backtracking algorithm is then identified as *chronological backtracking* algorithm.

The family of backtracking algorithms are capable of finding all solutions of a CSP or reporting it is insoluble if no solutions exist, and they will halt in at most exponential number of steps. In fact, they belong to an important class of approaches, namely, *systematic search methods*. The other systematic methods include *solution synthesis* algorithms, which aim to find all solutions of a CSP instance [42, 100, 109]. In contrast to the systematic methods, *stochastic search methods* do not always find a solution if the solution exists and will not always terminate if there is no solution. In the last few years, local search strategies have been reintroduced into the satisfiability and constraint satisfaction literature [58, 89]. These algorithms incrementally alter inconsistent value assignments to all the variables. They use a "repair" or "hill climbing" metaphor to move towards more and more complete solutions. To avoid getting stuck at "local optima", they are equipped with various heuristics for randomizing the search. Their stochastic nature generally voids the guarantee of completeness provided by the systematic search methods. In some problem domains, stochastic methods are very successful in solving large and hard problems that are too hard for backtracking algorithms [102]. However, throughout this dissertation, we will limit our attention to the family of backtracking algorithms.

Consistency inferencing [42, 75, 78, 84, 119] is a well known operation on CSPs which acts as a way of problem reduction that makes the constraints tighter. In the last two decades, consistency techniques have been extensively studied in the constraint programming community. Many consistencies have been proposed including *node consistency*, *arc consistency*, *path consistency*, and more generalized *k-consistency* [42, 43, 75, 77]. Generally speaking, backtracking will benefit from representations that are as explicit as possible; that is, from representations having a high level of consistency. Some levels of consistency can be so powerful that no search is required after a CSP is made consistent, such as *strong n-consistency* [42, 43] and *adaptive consistency* [37].

In a backtracking algorithm, both the variable to be instantiated and the value assigned to that variable can be determined *on the fly*. They are called *dynamic variable ordering* (DVO) and *dynamic value ordering* respectively. A "good" heuristic can improve the search by several orders of magnitude. Examples of dynamic variable ordering heuristics include the minimum width ordering [43] which exploits information of the graph of a CSP, and the *fail first* heuristic [60] which chooses the next

variable with the smallest remaining size, often used in a look-ahead backtracking algorithm. An example of a dynamic value ordering heuristic is the minimal-conflicts heuristic [81].

The identification of tractable classes of CSPs that can be solved in polynomial time is important from both the theoretical and the practical point of view and has been extensively studied over the last two decades. Such work involves identifying either the topological properties of CSPs, or the properties of constraints in the CSPs, or both. One of the basic topological properties that supports tractability is a tree structure. This has been observed from different perspectives, in constraint theory, complexity theory and database theory. The induced width [43, 44] of constraint satisfaction problem is an important property and the complexity of solving a CSP is known to be bounded by an exponential in its induced width. Thus a topological structure that has a bounded induced width is tractable, *e.g.*, the *k-tree* structure [45]. Tractable classes that are characterized by constraint properties are thoroughly studied in [26, 62, 63, 64]. Generally speaking, a CSP is tractable if all the constraints in the CSP are restricted to a family of constraints which are closed under some algebraic operations.

Structure driven techniques emerged from an attempt to exploit the tractability properties. Various graph based techniques whose complexity are tied to graph parameters were identified. These techniques include *adaptive consistency* [37], *tree clustering* [38], and *graph based learning* [30], all of which are exponentially bounded by the induced width of the constraint graph, and the *cycle-cutset scheme* [32], which is exponentially bounded by the size of the constraint graph's *cycle-cutset* [74]. Furthermore, a CSP with a tree structure can be solved in a backtrack free manner after it is made arc consistent. A path consistent row convex CSP can also be solved in a backtrack free manner [112, 113].

1.1.2 Problem Modeling

Problem statements are usually stated in natural languages. A very important part of solving real-life problems using the constraint programming approach is modeling the problem in terms of CSPs, *i.e.*, variables, domains and constraints.

Generating a Formulation

Let us consider the well known "SEND + MORE = MONEY" puzzle used in [114]. The problem can be stated as "to give each letter $\{S, E, N, D, M, O, R, Y\}$ a different

digit from $\{0, \ldots, 9\}$ so that the equation SEND + MORE = MONEY is satisfied".

Example 1.3 *The easiest way to model this problem is to give one variable for each of the letters and set one constraint corresponding to the equation and an* alldifferent *constraint specifying that each of the letters must take a different value from* $\{0, \ldots, 9\}$. *In such a CSP formulation, there are* 8 *variables,* $\{s, e, n, d, m, o, r, y\}$ *and each variable has the same domain of values* $\{0, \ldots, 9\}$. *Two non-binary constraints are*

$$10^3(s+m) + 10^2(e+o) + 10(n+r) + d + e = 10^4 m + 10^3 o + 10^2 n + 10e + y,$$
$$\text{alldifferent}(s, e, n, d, m, o, r, y).$$

For BT, this model is not very efficient because with BT, all of the variables need to be instantiated before the "large" equation constraint and the *alldifferent* constraint can be tested. Thus little of the search space can be pruned to speed up the solving. If an algorithm performing more consistency checks at each step of the backtrack search is applied to solve the CSP, *e.g.*, the maintaining arc consistency algorithm (see Section 2.5.4), the search space can be more effectively pruned. However, generally it is very expensive to enforce arc consistency on the two global constraints unless there exists some specially designed methods to speed up the constraint propagation.

Example 1.4 *A more efficient model uses the carry bits to decompose the "large" equation constraint into a collection of "small" constraints. With a little thought, we can see that* M *must have the value* 1 *and* S *can only take values from* $\{1, \ldots, 9\}$. *Besides the variables in the first model, the new model includes three "carrier" variables,* c_1, c_2, c_3. *The domains of variables* e, n, d, o, r *and* y *are* $\{0, \ldots, 9\}$, *the domain of* s *is* $\{1, \ldots, 9\}$, *the domain of* m *contains a single value* $\{1\}$, *and the domains of all the carrier variables* c_1, c_2, c_3 *are* $\{0, 1\}$. *With the help of the carrier variables, the equation constraint can be decomposed into several smaller constraints,*

$$\begin{aligned} e + d &= y + 10c_1, \\ c_1 + n + r &= e + 10c_2, \\ c_2 + e + o &= n + 10c_3, \\ c_3 + s + m &= 10m + o. \end{aligned}$$

The alldifferent *constraint is unchanged,* i.e., $\text{alldifferent}(s, e, n, d, m, o, r, y)$.

The advantage of this model is that these smaller constraints can be tested earlier in the backtrack search, and thus many inconsistent valuations can be pruned. Also, when the maintaining arc consistency algorithm is used, it is relatively cheaper to achieve arc consistency over these smaller constraints than over the original global constraint.

In the above formulations, the *alldifferent* constraint can also be replaced by a set of "small" constraints, *e.g.*, $s \neq e, \ldots, r \neq y$, and we obtain a third and a fourth model for the problem. However, if the maintaining arc consistency algorithm is used to solve the CSP, we have to decide whether it is worth decomposing the *alldifferent* constraint as enforcing arc consistency on alldifferent constraints can be processed very quickly by using some specially designed methods, or *propagators* [94]. A propagator enables the constraints to be checked earlier and more efficiently and sometimes it enforces a stronger consistency than enforcing arc consistency on the decomposition of the constraints [94]. However, it is usually more difficult and time-consuming to design a propagator for a class of constraints than to find a decomposition scheme for them, and the use of propagators may hardly be combined with other CSPs techniques, *e.g.*, the backjumping method. In contrast, a decomposition solution does not demand such special treatment and thus it can be used with all possible CSP solving methods.

As we can see, there are often many different approaches to formulating a given problem and these formulations result in very different problem solving performances. For another example, Nadel [86] presents 9 essentially different formulations for the n-Queens problem, which asks "to place n queens on an $n \times n$ chess board such that none of the queens attacks the other". In order to facilitate the expression of constraint satisfaction problems, several languages or tools have been developed. Examples of these range from the earliest Alice [72], to modern languages like ILOG Solver [61], CHIP [2, 40], ECLiPSe [41], Oz [107], and Prolog III [24].

However, it is one thing to say "all one has to do is to express the problem with constraints". It is another to express the constraints in a manner which permits efficient solution. In the past, the quality of a problem formulation has largely depended on a problem solver's experience or on *trial and error*. Alternatively, some very general guidelines have been suggested to aid the problem solver to find a high quality formulation. Examples of such guidelines, or rules of thumb, include the use of redundant constraints, making the constraints as tight as possible and keeping the arity of of the constraints as low as possible. More practically, there are some methodologies to improve a given CSP formulation with respect to a certain class of problem solving techniques. These methodologies include adding or removing redundant constraints,

adding or removing redundant variables, exploiting symmetries, and transforming a CSP formulation into a different but equivalent representation.

Improving the Formulations

There can be many ways of representing a problem with constraints. A constraint in a CSP is called *redundant* if its removal does not change the solutions of the CSP. Adding redundant constraints in a CSP formulation makes the implicit knowledge of the problem explicit and by using such knowledge, or in constraint programming words, by using these constraints, during the backtrack search, a large portion of the search space that does not contain a solution can be pruned at an early stage of the search. In particular, adding redundant constraints is crucial to solving real world problems [56, 90, 111, 114]. In the second formulation of the puzzle problem (see Example 1.4), we make use of the fact that variable m must have the value 1 explicit which immediately prunes the search space that assigns the other values to m. Adding redundant constraints is somehow similar to achieving a certain level of consistency in the CSP formulation, but in a less systematic manner. Should redundant constraints always be added into the formulation? Sometimes removing redundant constraints can also improve the problem solving. Dechter and Dechter [29] argue that there are cases for removing redundant constraints such that the CSP instance becomes acyclic and thus can be solved in polynomial time.

A *redundant variable* or *hidden variable* does not participate in the solution of a problem. Adding redundant variables may help to represent a constraint which otherwise must be expressed as a global constraint. For example, the carrier variables in Example 1.4 are redundant variables used to decompose the global equation constraint. Removing redundant variables is also meaningful because the size of the search space is decreased as the CSP has fewer variables. The variables in a CSP formulation can also be manipulated in the form of *variable joining* or *variable grouping* [32]. That is, several variables can be condensed to one variable and thus the arity of constraints over those variables may be decreased.

Symmetries widely exist in non-random problems. For example, a flip of a solution of the n-Queens problem is also a valid solution. In a graph coloring problem, a permutation of colors in one labelling scheme will also fulfill the problem requirements. Exploiting such symmetries by adding constraints to exclude the symmetries can greatly reduce the search space [111]. Symmetries can also be used in the form of *meta values* in variable domains. A *meta value* of a variable is an abstraction of a subset of domain values which behave similarly in the constraints involving the variable.

The use of meta values can reduce the search space by several orders of magnitude. Current methods include identifying meta values by interchange-ability properties of the domain values [13, 46]. For example, Weigel and Faltings [121] present a method to take advantage of interchange-abilities to represent sets of equivalent values by meta values and thus obtain more compact representations.

A CSP formulation can be transformed into an equivalent representation in which a different set of variables and constraints are defined. The original formulation is equivalent to its transformation under a proper definition of equivalence [95], *e.g.*, a solution of the original formulation can be extracted from a solution of its transformation in a polynomial number of steps. Sometimes, it is useful to completely change the denotation of the variables in the original formulation and thus redefine the constraints. For example, two general transformation techniques exist to translate a general CSP into an equivalent binary CSP, namely the *dual graph* method [38, 95] and *hidden variable* method [95]. The other transformation approaches include: Jégou [65] considers transforming CSP formulations based on an analysis of what he describes as the "micro-structure" of the CSP. Weigel *et al* [120] describe a method to convert a general CSP into a binary boolean form which can be used to find different formulations of the original CSP.

The ability to generate a range of different CSP formulations can result in very different solving performance. One significant and open question, "which is the best formulation?" needs to be addressed. A precise answer to the above question seems unlikely. First, it cannot be answered without tying the formulation with a specific problem solving technique. For example, a backtracking algorithm will generally benefit from adding more redundant constraints. However, the redundant constraints should be carefully selected as they may not always contribute to the pruning of the search space but just result in the backtracking algorithm performing more constraint checks at each step of the search. Second, modeling still remains an "art" in most problem domains. The quality of the formulations has largely depended on the problem solver's experience and knowledge in the problem domain. Selman *et al* [101] list 10 challenges in AI research, in which the challenge for problem modeling is stated as, "characterize the computational properties of different encodings of a real-world problem domain, and/or give general principles that hold over a range of domains."

1.2 Motivations and Contributions

Since constraint satisfaction problems are intractable in the general case, it is natural to use all possible techniques to improve the efficiency. Given a problem statement, we may ask what is the best CSP formulation for the problem, and given a CSP formulation, we may want the best solving algorithm. Perhaps, we are more interested in the best combination of algorithm and formulation. Nevertheless, these demand a better understanding of the modeling techniques, the solving techniques, and the interactions between them.

In the past, most of the research activities in the constraint programming community have concentrated on the development of efficient problem solving techniques. Among these solving techniques, the backtracking algorithm is a very important CSP solving method. So far, a number of improvements to the naive backtracking algorithm have been proposed. These techniques can be conveniently classified as *look-ahead schemes* and *look-back schemes* [34]. In general look-ahead schemes involve enforcing a certain level of consistency, using a dynamic variable ordering heuristic and using a dynamic value ordering heuristic. The backtracking algorithms using look-ahead schemes include the well known forward checking algorithm (FC) and the maintaining arc consistency algorithm (MAC). Whenever the algorithm encounters a dead-end and prepares for the backtracking step, look-back schemes are invoked to perform the functions that decide how far to backtrack by analyzing the reasons for the dead-end (backjumping), and record the reasons for the dead-end in the form of new constraints so that the same conflicts will not arise again later in the search (learning). The hybrids of the above two schemes, including the forward checking with conflicts-directed backjumping algorithm (FC-CBJ) and maintaining arc consistency algorithm with conflicts-directed backjumping algorithm (MAC-CBJ), have also been developed in the literature. Unfortunately, sometimes the look-ahead schemes are counterproductive to the look-back schemes. For example, Prosser [91] observes the fact that, in some cases, backjumping may become less efficient after enforcing consistency. Bacchus and van Run [8] observe that adding CBJ to an algorithm that already uses a dynamic variable ordering based on the minimal domain heuristic is unlikely to yield much improvement. Previous experiments on random harder problems and benchmark problems show that FC-CBJ significantly improves FC in most cases, but MAC-CBJ hardly improves MAC in these experiments and may actually show a degradation in performance [16]. As a result, MAC is evaluated to be the best algorithm that is capable of solving large and hard CSP instances, and CBJ seems to

be a "costly gadget" for MAC as it rarely provides enough benefits compared to its overhead in the backtrack search.

In this dissertation, we argue that the above observations have serious limitations. First, they are established purely on experiments on random binary problems and some toy problems, and those problems are not hard to solve using today's computational power. Second, there exists little theoretical justification for the above conclusions about CBJ. We theoretically investigate the relation between some look-ahead techniques, including dynamic variable ordering and consistency enforcement, and the backjumping technique. Our theoretical results partially explain why a backtracking algorithm doing more in the look-ahead phase cannot benefit more from the backjumping schemes. We propose a new algorithm, called GAC-CBJ, a hybrid of generalized maintaining arc consistency algorithm (GAC) and conflict directed backjumping (CBJ) that can be applied to general CSPs. Although Bessière and Régin [16] conclude from their experiments on random binary CSPs that the enhancement of CBJ to MAC (the binary version of GAC) will not pay off in the general cases, our experiments in some real world domains show that GAC-CBJ improves GAC by several orders of magnitude on hard and large instances and does not degrade performance too much on relatively easier instances.

Different formulations for a given problem can be roughly evaluated by identifying one or more characteristics or parameters of the formulations. These parameters include the size of the search space, the density of solutions, and the "constrainedness" of search used in "phase transition" analysis [55]. Sometimes these parameters can provide a good "predictor" of how difficult it would be to solve the formulation. For example, random instances are usually hard to solve when the "constrainedness" is close to 1. Nadel [86] presents an evaluation of different formulations of the n-Queens problem by a theoretical estimate of the expected cost of search for a particular algorithm and search ordering. Borrett extends Nadel's work to more general cases in [19]. However, their approach depends on a statistical model to compute the expected cost of a given backtracking algorithm, and the assumptions made in the analysis do not hold for real world problems. Also, their evaluations cannot be used to provide a general principle saying under which circumstances one formulation is better than the other.

We are more interested in a purely theoretical evaluation of modeling techniques depending only on the basic elements of a CSP formulation, *i.e.*, variables, domains and constraints. To evaluate a modeling technique, we mean to fix the solving method and apply the modeling technique to the original CSP formulation to see how such a

reformulation affects the performance of the solving method or whether the reformulated model can be improved with respect to the solving method. Such an evaluation can provide a general guideline for problem modeling as to whether or not a modeling technique should be applied. Also, it may help us to understand the interaction between a modeling technique and a solving method and thus to design a new algorithm or improve an existing algorithm that may work better in a specific problem formulation. Furthermore, such a comparison is useful to the problem solving approaches based on commercial constraint programming packages, *e.g.*, ILog Solver, in which the solving methods are fixed to several well-known fast algorithms, such as the forward checking algorithm and the maintaining arc consistency algorithm.

In this dissertation, we theoretically study two modeling techniques, the dual and hidden transformations, which translate a general CSP into an equivalent binary representation. We choose the above two modeling techniques for the following reasons. First, in the past, much research has concentrated in binary CSPs because such transformations exist. However, to date, few results have been given which evaluate the performance of backtracking algorithms on the original formulation and its translated representations. Our results can be used to justify whether we should apply the transformation and solve the binary CSP or whether we should just select a backtracking algorithm to solve the original non-binary problem directly. Second, there exists strict definitions of the above two transformations, which enable a theoretical analysis on the efficiency of the solving techniques. Third, although a "pure" form of the dual or hidden transformation is rarely used in modeling a problem, the dual and hidden transformations contribute to a wide range of modeling techniques, often in the form of partial conversions. Thus, the results are meaningful to guide modeling in practice.

We compare an original problem formulation, its dual transformation, and its hidden transformation to see whether one formulation is "stronger" than or "equivalent" to another with respect to the effectiveness of achieving arc consistency. We find that arc consistency on the original formulation and the hidden translation are "equivalent" under the above meaning, but arc consistency on the dual translation is stronger than the arc consistency on the original formulation. Moreover, if the original problem has a special structure, arc consistency on the dual is "equivalent" to arc consistency on the original and the hidden representations. Then we compare more extensive consistency properties that can be applied to the dual and hidden formulations, and establish a hierarchy of the above "strongness" relation with respect to the combinations of consistency and formulation. Some of these relations have been identified

by Stergiou and Walsh in [106]. For example, they compare arc consistency on the original problem and the one on its dual and hidden transformations. However, they only give some illustrative proofs for their results. We present here stricter proofs for the above relations based on the formal definitions of the dual and hidden transformations. Bacchus and van Beek [7] compare the performance of the forward checking algorithm under the above three formulations. For example, they give examples to show that FC on the original may be exponentially better or worse than FC on the dual problem or hidden problem. In this dissertation, we extend their comparison to include two more backtracking algorithms, the chronological backtracking algorithm and the maintaining arc consistency algorithm, and we also present some new results about the forward checking algorithm, *e.g.*, the relation between its performance on the dual transformation and the hidden transformation. As a result, we either give a theoretical bound saying how much one formulation is better than the other under a given algorithm, or give examples to show that such a bound does not exist. These comparisons contribute exactly to the question which Selman *et al* [101] regard as a challenge in future AI research.

1.3 Overview of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 gives the definition of constraint satisfaction problems and a review of local consistency and backtracking algorithms. Then we define the search tree explored by backtracking algorithms. Also in Chapter 2, we present our specification of the arc consistency achievement algorithm and several backtracking algorithms, which will be used in later chapters. It is worth noting that all these algorithms can be applied to non-binary CSPs.

Chapter 3 studies the relationship between look-back and look-ahead techniques for backtracking algorithms. First, we show by example that CBJ may be exponentially better than an algorithm that maintains strong k-consistency in the backtrack search and we show that backjumping becomes useless if an appropriate variable ordering strategy is used in the chronological backtracking. Second, we use the concept of *backjump level* in the execution of a backjumping algorithm and show that an algorithm maintaining strong k-consistency always visits no more nodes than a backjumping algorithm that is allowed to backjump no more than k levels. Third, we present a new algorithm, *GAC-CBJ*, which is an extension of Prosser's MAC-CBJ [93] to general CSPs. In our experiments, GAC-CBJ shows significant improvements over GAC on some real world problems.

The formal definitions of the dual and hidden transformations are given in Chapter 4. We compare the "strongness" of arc consistency on three formulations of any problem, an original formulation, and its dual and hidden transformations. We show that arc consistency on the original formulation and its hidden transformation are "equivalent", but arc consistency on the dual transformation is "stronger" than the one on the original formulation. Then we compare several local consistency properties that can be applied to the dual and hidden formulations, and establish a hierarchy for the various combinations of consistency and formulation with respect to the "strongness" relation.

In Chapter 5, we compare the performance of selected backtracking algorithms on the above three formulations. Given a backtracking algorithm, we identify one of two mutually exclusive relations between two formulations, either "one may be exponentially worse than another" or "it is at most (polynomially) bounded worse than another." Three algorithms are used in the comparisons, including the chronological backtracking algorithm, the forward checking algorithm, and the maintaining arc consistency algorithm. In this chapter, we present a hierarchy for the various combinations of algorithm and formulation with respect to the above relations.

Chapter 6 concludes the dissertation and discusses about possible future work.

Chapter 2

Background

In this chapter, we introduce much of the background necessary to understand the rest of the dissertation. We give formal definitions of constraint satisfaction problems and solutions of a CSP. Then we briefly review various local consistency techniques and backtracking algorithms and define the search tree explored by backtracking algorithms. At last, we present our specification of the arc consistency achievement algorithm and several backtracking algorithms that will be used in later chapters.

2.1 Definition

Definition 2.1 (CSP) *An instance of a constraint satisfaction problem, P, is a tuple $(\mathcal{V}, \mathcal{D}, \mathcal{C})$, where* [1]

- $\mathcal{V} = \{x_1, \ldots, x_n\}$ *is a finite set of n variables,*
- $\mathcal{D} = \{dom(x_1), \ldots, dom(x_n)\}$ *is a set of domains. Each variable $x \in \mathcal{V}$ is associated with a finite domain of possible values, $dom(x)$. The maximum domain size $max_{x \in \mathcal{V}} |dom(x)|$ is denoted by d,*
- $\mathcal{C} = \{C_1, \ldots, C_m\}$ *is a finite set of m constraints or relations. Each constraint $C \in \mathcal{C}$ is a pair $(vars(C), rel(C))$, where*
  - $vars(C) = \{x_{i_1}, \ldots, x_{i_{r_i}}\}$ *is an ordered subset of the variables, called the constraint scope or scheme, the size of $vars(C)$ is known as the arity of the constraint. If the arity of the constraint is equal to 2, it is called a binary constraint. A* non-binary *constraint is a constraint with arity greater*

[1] Throughout the dissertation, we use n, d, m, and r to denote the number of variables, the maximum domain size, the number of constraints, and the maximum arity of the constraints in the CSP, respectively.

than 2. The maximum arity of the constraints in $\mathcal{C}$*,* $max_{C \in \mathcal{C}} |vars(C)|$*, is denoted by* r*,*

- $rel(C)$ *is a subset of the Cartesian product* $dom(x_{i_1}) \times \cdots \times dom(x_{i_{r_i}})$ *that specifies allowed combinations of values for the variables in* $vars(C)$*. An element of the Cartesian product* $dom(x_{i_1}) \times \cdots \times dom(x_{i_{r_i}})$ *is called a tuple on* $vars(C)$*. Thus,* $rel(C)$ *is often regarded as a set of tuples over* $vars(C)$*.*

Generally, we do not consider a variable that is not involved in any constraint. In the following, we assume that for any variable $x \in \mathcal{V}$, there is at least one constraint $C \in \mathcal{C}$ such that $x \in vars(C)$. By definition, a tuple over a set of variables $X = \{x_1, \ldots, x_k\}$ is an ordered list of values $(a_1, \ldots, a_k)$ such that $a_i \in dom(x_i)$, $i = 1, \ldots, k$. A tuple over X can also be regarded as a set of variable-value pairs $\{x_1 \leftarrow a_1, \ldots, x_k \leftarrow a_k\}$. Furthermore, a tuple over X can be viewed as a function $t : X \rightarrow \cup_{x \in X} dom(x)$ such that for each variable $x \in X$, $t[x] \in dom(x)$. For a subset of variables $X' \subseteq X$, we use $t[X']$ to denote a tuple over X' by restricting t over X'. We also use $vars(t)$ to denote the set of variables for tuple t.

An *assignment* to a set of variables X is a tuple over X. We say an assignment t to X is *consistent* with a constraint C if either $vars(C) \not\subseteq X$ or $t[vars(C)] \in rel(C)$. A *partial solution* to a CSP is an assignment to a subset of variables. We say a partial solution is *consistent* if it is consistent with each of the constraints. A *solution* to a CSP is a consistent partial solution over all the variables. If no solution exists, the CSP is said to be insoluble. The set of solutions to a CSP P is denoted by $sols(P)$. Given two CSP instances P_1 and P_2, we say $P_1 = P_2$ if they have exactly the same set of variables, the same set of domains and the same set of constraints between the variables; *i.e.*, they are syntactically the same. We say P_1 is *equivalent to* P_2 *iff* they have exactly the same set of solutions, *i.e.*, $sols(P_1) = sols(P_2)$ [2]. It is easy to verify that, if $P_1 = P_2$, then P_1 is equivalent to P_2. A CSP is *empty* if either one of its variables has an empty domain or one of its constraints has an empty set of tuples. Obviously, an empty CSP is insoluble, *i.e.*, it has an empty set of solutions.

Definition 2.2 (projection) *Given a constraint* C *and a subset of variables* $S \subseteq$

[2]Two equivalent CSPs do not always have the same domains and the same constraints, but they do have the same variables. Equivalence is usually used in consistency techniques, as will be discussed in the next section. However, there exists other types of equivalence between two CSPs which do not have the same variables. For example, the dual transformation and hidden-variable transformation of a CSP are equivalent to the original representation. In that case, a more flexible definition should be used, as in [95].

vars(C), *the projection* $\pi_S C$ *is a constraint, where* $vars(\pi_S C) = S$ *and* $rel(\pi_S C) = \{t[S] \mid t \in rel(C)\}$.

Definition 2.3 (selection) *Given a constraint* C *and an assignment* t *to a subset of variables* $X \subseteq vars(C)$*, the selection* $\sigma_t C$ *is a constraint, where* $vars(\sigma_t C) = vars(C)$ *and* $rel(\sigma_t C) = \{s \mid s[X] = t \text{ and } s \in rel(C)\}$.

2.2 Consistency Techniques

Consistency achievement is a process of removing values from domains, removing tuples from constraints, and adding new constraints into the set of constraints, without removing any solutions from a CSP. A *local inconsistency* or *inconsistency* in a CSP formulation is a partial solution over $k-1$ variables that cannot be consistently extended to a k^{th} variable and so cannot be part of any solution of the CSP. Sometimes, an inconsistency is also called a *no-good* [32, 48]. The basic idea of consistency techniques is that if we can deduce an inconsistency in the CSP formulation, then it can be removed by means of removing a value from the domain of a variable if the inconsistency involves only one variable, removing a tuple from a constraint if the variables in the inconsistency are already constrained by a constraint, or adding a new constraint if there is no such a constraint that constrains those variables in the inconsistency. By removing inconsistencies, we reduce a CSP to an equivalent but tighter problem.

The objective of consistency achievement is not to solve the problem, but to get a formulation having as few inconsistencies as possible. A *minimal network* [3] is a formulation that does not have any inconsistency; *i.e.*, for each value in the domain of a variable, there is a solution having that value assigned to the variable, and for each tuple in a constraint, there is a solution in which the tuple appears. To compute the minimal network of a CSP formulation is an NP-complete task [75, 84].

Although consistency achievement alone rarely generates solutions, it can help to solve CSPs in various ways. It can be used in *preprocessing*, which means reducing the problem before any other techniques are applied to find solutions. It can also be used during a backtrack search, by pruning off search space after each step to extend the partial solution.

Consistency techniques were first introduced for binary CSPs. Mackworth [75, 77]

[3] It is called a *network* because in early research, binary CSPs are usually identified with a *constraint graph*, or *constraint network*.

defines three properties of binary CSPs that characterize local consistencies: *node*, *arc*, and *path* consistency. A binary CSP is *node consistent* if for each unary constraint C constraining a single variable x, and for each value $a \in dom(x)$, $\{x \leftarrow a\}$ satisfies C. The CSP is arc consistent if for each constraint C constraining a pair of variables x and y, and for each value $a \in dom(x)$, there is a value $b \in dom(y)$ such that $\{x \leftarrow a, y \leftarrow b\}$ satisfies C. The CSP is path consistent if for any triple of variables x, y and z, and any value $a \in dom(x)$ and $b \in dom(y)$ such that $\{x \leftarrow a, y \leftarrow b\}$ is consistent, there is a value $c \in dom(z)$ such that $\{x \leftarrow a, y \leftarrow b, z \leftarrow c\}$ is also consistent.

The concept of Mackworth's arc consistency has been generalized to non-binary CSPs [76].

Definition 2.4 (arc consistency) *Let $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a CSP. Given a constraint C and a variable $x \in vars(C)$, a value $a \in dom(x)$ is* supported *in C if there is a tuple $t \in rel(C)$, such that $t[x] = a$. t is then called a* support *for $\{x \leftarrow a\}$ in C. C is* arc consistent *iff for each of the variables $x \in vars(C)$, and each of the values $a \in dom(x)$, $\{x \leftarrow a\}$ is supported in C. P is* arc consistent *iff each of its constraints is arc consistent.*

Freuder [42, 43] generalizes Mackworth's consistencies to a family of k-consistencies.

Definition 2.5 (k-consistency) *A CSP is k-consistent if and only if given any consistent partial solution over $k-1$ distinct variables, there exists an instantiation of any k^{th} variable such that the partial solution plus that instantiation is consistent. A CSP is strongly k-consistent* iff *it is j-consistent for all $1 \leq j \leq k$.*

For binary CSPs, node, arc and path consistency correspond to one-, two- and three-consistency, respectively. Moreover, the definition of k-consistency does not require the CSP to be binary. Note that arc consistency is not the same as two-consistency for general CSPs. A strongly n-consistent CSP is called *globally consistent*. Globally consistent CSPs have the property that any consistent partial solution can be successively extended to a full solution of the CSP without backtracking [35]. A globally consistent CSP formulation is always a minimal network, but in general the converse is not true.

Mackworth [75] presents several algorithms to achieve node, arc, or path consistency on a binary CSP. Arc consistency is widely used in solving CSPs because it

only changes the domains of variables [4]. It can be easily implemented and cheaply achieved but has more pruning power than node consistency. The algorithms to achieve arc consistency on binary CSPs have been extensively studied. Mackworth presents three successively improved arc consistency achievement algorithms, named *AC-1*, *AC-2* and *AC-3*. The worst case complexity of the best among them, AC-3, is $O(md^3)$. Mohr and Henderson propose *AC-4* [82] which has an optimal worst case complexity of $O(md^2)$. However, AC-4 lags behind AC-3 on average time complexity, while AC-4 is too near to the worst case time complexity, and AC-3 runs faster than AC-4 in practice despite its non-optimal worst case complexity [78, 118]. *AC-5* [115] is a generic framework which summarizes all previous algorithms and with which special routines can be designed for particular constraint classes. *AC-6* and *AC-7* [14, 15] keep the optimal worst case complexity $O(md^2)$ and improve the average time complexity significantly. Furthermore, Schiex *et al* proposes a "lazy" version of AC-7, called *lazy arc consistency* [98]. Lazy arc consistency does not enforce full arc consistency on a CSP instance but guarantees if a CSP passes a lazy arc consistency test, it will also be able to pass an arc consistency test. Thus it is cheaper than AC-7 when used in a backtrack search but has the ability to prune branches. AC-3 can be easily extended to non-binary CSPs. Mackworth proposes the algorithm CN [76], which is a kind of generalization of AC-3 and has the worst case complexity of $O(mr^2d^{r+1})$. Mohr and Massini propose GAC4 [83], based on AC-4 for binary CSPs, which has the worst case complexity of $O(md^r)$. An AC-7 version of an arc consistency achievement algorithm for non-binary CSPs is given in [17].

Cooper [25, 108] proposes an algorithm optimal in the worst case for achieving strong k-consistency. The principle is that if a partial solution is found to be inconsistent and therefore rejected, all tuples in which the partial solution is a projection will be rejected. Enforcing arc consistency on a CSP will only remove the inconsistent values from their domains and will not change the constraints. However, enforcing strong $k-$consistency for $k \geq 3$ will remove inconsistent tuples from the constraints and possibly add more constraints to the CSP. Thus achieving strong k-consistency may dramatically change the formulation of a CSP, as the number of new constraints could be exponential in k. Consequently, it is more expensive to maintain strong $k-$consistency for $k \geq 3$ in a backtrack search. In fact, it is not evident yet that it is worthwhile to maintain a stronger consistency than arc consistency. Thus maintaining strong k-consistency in a backtrack search is currently only of theoretical interest.

[4] Sometimes, a constraint may be represented implicitly by a predicate or function call, thus it is very difficult to update the constraint.

One issue for algorithms to achieve the same consistency on a CSP instance is, whether they all compute the same results, *i.e.*, whether the CSP instance after applying the consistency achievement algorithms has the same domain for each of the variables, and the same set of constraints. For arc consistency, as we will show in a later chapter, there is a unique arc consistent subdomain, called *arc consistency closure* and each arc consistency achievement algorithm should compute the arc consistency closure. For k-consistency achievement algorithms, in general, the results are not unique. One reason is that new constraints can be added into the CSP so that *universal constraints*, *i.e.*, those that permit any tuples, can be arbitrarily added or removed from the CSP formulation, and sometimes different non-universal constraints can also be added. However, it is reasonable to assume that after achieving strong k-consistency, the domains of the variables are the same for different strong k-consistency achievement algorithms.

Also, the concepts a CSP is X-consistent and a CSP can be made X-consistent should be distinguished. We say a CSP is X-consistent if it conforms to the conditions defined in X-consistency. Otherwise, there are some domain values and tuples in the formulation that violate the consistency, and by removing them using a consistency achievement algorithm we can establish X-consistency in the new CSP formulation. We say a CSP can *be made X-consistent* if the resulting CSP is not empty.

Arc Consistency Achievement Algorithm (AC-3)

We achieve arc consistency by removing from the domains those values that are not supported in some constraint. When a value is removed from its domain, some tuples using the value in a constraint restricting that variable become *invalid*. The invalid tuples are removed from the constraints implicitly. The changes in one domain are propagated to other variables for which a new support needs to be sought in the tightened constraints.

We present a variant of AC-3 for general CSPs in Figure 2.1. The original version of AC3 propagates the deletions of domain values *via* constraints, while we use a variable propagation strategy. In experiments, the above two strategies are competitive to each other. S in AC-3 is a *queue* or *stack* to keep those deletions which have not yet been propagated. Function *exists* in AC3 tries to find a valid support for $\{v \leftarrow solution[v]\}$ in constraint C. A generic implementation of *exists* is to enumerate and verify every possible tuple in which v receives value $solution[v]$ according to the current domains of the variables. Certainly, this brute-force approach may be very expensive if the arity of the constraint is large. It is possible to use specially

```
function exists( in C : constraint; in v : variable ) : boolean;
   %% return true if solution[v] has a valid support in constraint C.

function revise( in C : constraint; in v : variable ) : boolean;
1  changed ← false;
2  for each a ∈ dom(v) do
3     solution[x] ← a;
4     if not exists( C, v ) then
5        changed ← true;
6        dom(v) ← dom(v) − {a};
7  if changed then push( v, S );
8  if |dom(v)| = 0 then return false else return true;

function AC3() : boolean;
1  S ← V;
2  while S ≠ ∅ do
3     y ← top( S ); pop( S );
4     for each C ∈ C and y ∈ vars(C) do
5        for each v ∈ vars(C) and v ≠ y do
6           if not revise( C, v ) then return false;
7   return true;
```

Figure 2.1: AC-3.

designed routines for some classes of constraints to speed up the above process as long as completeness is guaranteed.

2.3 Search Tree and Backtracking Algorithms

The simplest algorithm to solve a CSP is the *generate and test* (GT) algorithm. The GT algorithm starts from a null assignment or an empty partial solution, and recursively extends the partial solution to a full solution by first choosing an uninstantiated variable and then assigning it a value from the domain. When a full assignment is obtained, the full assignment is checked whether it is a solution of the CSP that satisfies all the constraints. The GT algorithm terminates when each possible assignment over all the variables has been examined or a certain number of solutions have been found.

A *generate and test search tree* or *search tree* for short can be constructed from the execution of the GT algorithm. Each partial solution generated in the execution of the GT algorithm is identified by a node. There is an edge from node u to node v if v is an immediate extension of u. The empty solution at the start is the *root* of the search tree. At each node in the search tree, each variable occurring in the current partial solution is said to be *instantiated* to some value from its domain. The variable being chosen to be instantiated is called the *current variable*. Accordingly, the variables having been instantiated are called *past variables* and the variables having not been instantiated yet are called *future variables*.

In the search tree, if there is an edge from node u to node v, u is called the *parent* of v and v is a *child* of u. If there is a path from u to w, u is called an *ancestor* of w and w is a *descendant* of u. A node with no children is called a *leaf node*. The leaf nodes in the search tree are the full assignments which can not be extended. The total number of leaf nodes in the search tree is $\prod_{x \in \mathcal{V}} |dom(x)|$ if all solutions are to be found. A node u and all its descendants form a *subtree* of the search tree. u is called the *root* of the subtree. The *level* of node u in a search tree is the length of the path from the root to u. Hence, the nodes in the search tree can be classified as *first level nodes*, *second level nodes*, and so on. The levels closer to the root are called *lower levels* and the levels farther from the root are called *higher levels*.

The construction of a search tree is determined by several factors, including the variable ordering and value ordering used to generate a full assignment, and the solutions requirements. The influence of variable ordering strategy and value ordering strategy will be discussed in the next section. The solutions requirements means

whether a single solution, a certain number of solutions, or all solutions are to be found.

The execution of a backtracking algorithm or *a backtrack search* can be seen as a search tree[5] traversal to extend a current partial solution to a full solution of the CSP. At each node in the search tree explored by a backtracking algorithm, an uninstantiated variable is selected and assigned a value from its domain to extend the current partial solution. Constraints are used to check whether such an extension may lead to a possible solution of the CSP and to prune subtrees containing no solutions based on the current partial solution. For example, the chronological backtracking algorithm checks whether the current partial solution is consistent with all the constraints and rejects those inconsistent ones. A *dead-end* is the situation where all values of the current variables are rejected by a backtracking algorithm when it tries to extend a partial solution. In such a case, some instantiated variables become *uninstantiated*, *i.e.*, they are removed from the current partial solution. This process is called *backtracking*. If only the most recently instantiated variable becomes uninstantiated then it is called *chronological backtracking*; otherwise, it is called *backjumping*. A backtracking algorithm terminates when all possible assignments have been tested or a certain number of solutions have been found. We say that a backtracking algorithm visits a node in the search tree if at some stage of the algorithm's execution the current partial solution identifies the node. The nodes visited by a backtracking algorithm form a subset of all the nodes belonging to the search tree. We call this subset, together with the connecting edges, the *backtrack search tree* generated by a backtracking algorithm.

Much of the work in constraint satisfaction during the last several decades has been devoted to improving the performance of the naive backtracking algorithm. Because the problem is known to be NP-complete [51], polynomial variants of backtracking algorithms are unlikely. Nevertheless, the average performance of the naive backtracking algorithm can be improved tremendously by equipping it with various enhancements.

The techniques to improve the naive backtracking algorithm can be conveniently classified as *look-ahead schemes* and *look-back schemes*, in accordance with back-

[5]The search tree discussed here is the search tree generated by the GT algorithm. If a static variable instantiation order and a static value instantiation order are used, we know such a search tree exists because the entire execution of the GT algorithm is known according to the orderings. However, if a dynamic variable ordering or a dynamic value ordering strategy is used in the backtrack search, we may not be able to declaratively describe the execution of the GT algorithm under the dynamic ordering. We will address this issue later in the next section.

tracking's two main phases of going forward to extend current partial solution and going back in case of a dead-end [36]. Look-ahead schemes can be invoked whenever the algorithm is preparing to assign a value to the current variable. The essence of these schemes is to reduce the search space through the use of a dynamic variable ordering, a dynamic value ordering, and a certain amount of constraint propagation or consistency enforcement. Enforcing a local consistency in the backtrack search has two benefits: The dead-ends are found out earlier such that backtracking occurs immediately and much futile search effort can be avoided, and inconsistent values are temporarily removed from the domain of the future variables and we need not consider these values until they are restored in backtracking. However, enforcing a local consistency in the backtrack search brings extra costs which may outweigh its benefits. Because the complexity of enforcing strong k-consistency is exponential in k, in practice, only restricted levels of consistencies are enforced. Among the backtracking algorithms with look-ahead enhancements, *forward checking algorithm* (FC) [60] and *maintaining arc consistency algorithm* (MAC) [96] are widely used to solve relatively hard and large CSPs [18]. For most applications, there is no evidence yet that enforcing a higher level consistency in the backtrack search will be better than FC and MAC.

Look-back schemes are invoked when the algorithm is preparing the backtracking step after encountering a dead-end. The reasons for the dead-end are analyzed. Knowing that the same dead-end will be encountered again if the instantiations which caused the dead-end have not been changed, the algorithm goes back directly to the source of the failure, instead of the immediate preceding variable in the ordering, *e.g.*, the *conflict-directed backjumping algorithm* (CBJ). Look-back schemes also include various learning algorithms which record the reasons for the dead-end in the form of new constraints so that the same conflicts will not arise again later in the search [30, 99].

A backtracking algorithm can be a hybrid of both a look-ahead scheme and a look-back scheme. A successful hybrid is the *forward checking with conflict-directed backjumping algorithm* (FC-CBJ) [92]. Over a long period, FC-CBJ was evaluated empirically [60, 87, 92] as the fastest algorithm until MAC was rediscovered to be the best in solving harder CSPs [96].

2.4 Variable Ordering and Value Ordering

It is known that a dynamic variable ordering can be a great improvement over a static variable ordering. The improvements from value ordering heuristics may not be as significant as those from variable ordering heuristics, especially in the case that all solutions are searched[6]. Throughout this dissertation, we assume that a static value ordering is used in the backtrack search.

We view a backtrack search as a search tree traversal in which the search tree is as generated by the GT algorithm. If a static variable instantiation order and a static value instantiation order are used, we know the search tree in advance because the entire execution of the GT algorithm is known. However, if a dynamic variable ordering or a dynamic value ordering is used in the backtrack search, we do not know the search tree before the execution of the backtracking algorithm. Nevertheless, such a search tree does exist and we can figure it out after the completion of the backtrack search. Because we know the order of instantiations made by the backtracking algorithm, the GT algorithm can follow this ordering. The ordering information could be missed at some node due to the pruning of an insoluble subtree. In such a case, GT algorithm will follow a pre-defined variable ordering, for example, $x_1, \ldots, x_n$, and choose the first uninstantiated variable in the ordering to be the current variable. Thus the search tree can be constructed after the execution of the backtracking algorithm.

Example 2.1 *Consider an integer linear program with* 5 *variables,* $x_1, \ldots, x_5$. *The domain for each variable is restricted to contain only* 3 *values,* $\{0, 1, 2\}$. *The linear constraints are*

$$\begin{aligned} x_1 + x_2 &\leq x_3 \\ x_1 + x_3 &> x_5 + 1 \\ x_2 - x_4 &\geq x_5 \end{aligned}$$

Figure 2.2 shows a fragment of the BT backtrack search tree to solve the above CSP instance. A (hypothetical) dynamic variable ordering is used in the backtrack search. For example, x_4 *is instantiated before* x_5 *when* x_3 *is instantiated with* 0 *and* 1, *and* x_4 *is instantiated after* x_5 *when* x_3 *is instantiated with* 2. *An inconsistent node is represented by a shadowed node and a solution node is marked with a* $\surd$ *sign. In the above figure, since* $\{x_1 \leftarrow 0, x_2 \leftarrow 1, x_3 \leftarrow 0\}$ *is an inconsistent node, BT does not*

[6]However, value ordering may affect backjumping algorithms dramatically [50].

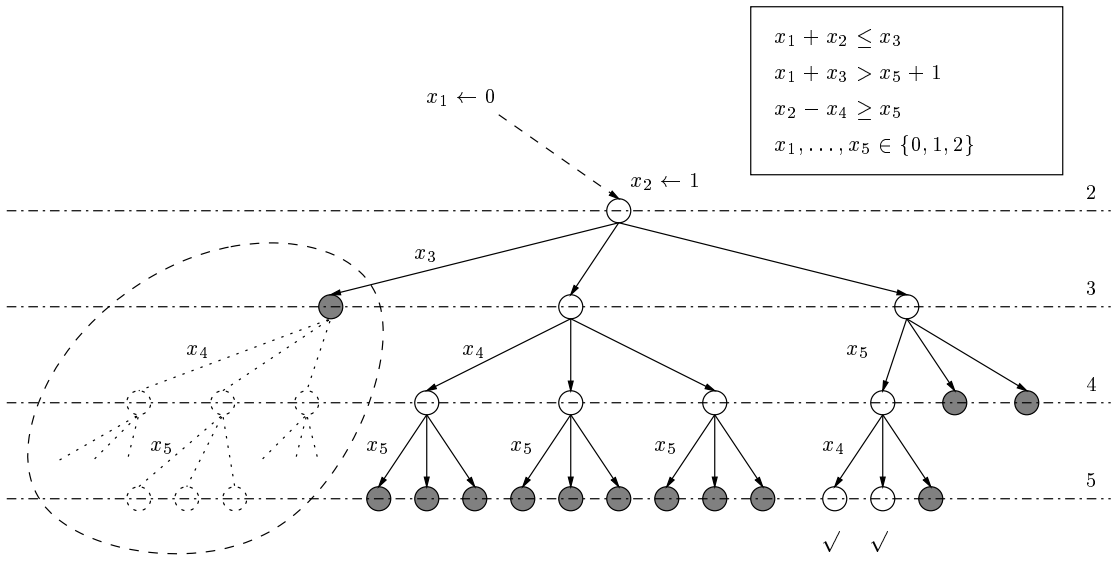


Figure 2.2: A fragment of the BT backtrack search tree for the CSP in Example 2.1

extend this branch. The dashed part attached to this node denotes a subtree in the generate and test search tree which is not part of the backtrack tree generated by BT.

2.5 Backtracking Algorithms

In this section, we will introduce several backtracking algorithms used in our study: BT, CBJ, FC, GAC, and FC-CBJ. We identify an algorithm by presenting a specification that is close to an implementation. There could be many possible ways to implement an algorithm. It is important, however, that all implementations of the same algorithm generate the same backtrack tree under the same variable ordering and the same ordering of constraint checks [7]. These algorithms are implemented to stop after finding the first solution. In order to find all solutions, a simple change to the termination condition is sufficient for the algorithms doing chronological backtracking, *e.g.*, BT, FC, and GAC, but in the cases of CBJ and its hybrids further modifications are necessary. For more explanations, please see [68]. It is also worth noting that all these algorithms can be applied to non-binary CSPs and the C++ implementations of the algorithms can be obtained from "ftp://ftp.cs.ualberta.ca/pub/xinguang/csp.zip."

[7] The order of constraint checking can affect the computation of the conflicts sets and thus the calculation of the backjumping point from a dead-end state.

2.5.1 Chronological Backtracking (BT)

Chronlogical backtracking (BT) is the starting point for all the more sophisticated backtracking algorithms. The pseudo code of BT is shown in Figure 2.3. BT uses the following data structures: *current* identifies the current level in the search tree starting from level 1; $solution[x]$ stores the current instantiation to variable x; $instantiated[x]$ marks whether x is currently instantiated and initially it is set to be false; $order[i]$ identifies the i^{th} variable instantiated in the current partial solution; and $count\_uninst[C]$ counts the number of uninstantiated variables in the scheme of constraint C and it is intialized to be $|vars(C)|$, *i.e.*, all the variables in the scheme are initially uninstantiated. BT starts from level 1 and terminates when it reaches level $n+1$ at which a solution is found. At level i, BT first chooses the next uninstantiated variable to be the current variable and records it in $order[i]$. Then BT tentatively instantiates the current variable with a value in its domain and checks whether the current partial solution is consistent with all the constraints. $count\_uninst[C]$ is used to control whether the constraint C is *checkable* at this stage, where C is checkable only if all the variables in its scheme have been instantiated; *i.e.*, the condition $count\_uninst[C] = 0$ is satisfied. If the current partial solution passes the consistency check, the instantiation of the current variable is admitted and BT goes on to the next level. Otherwise, the next value in the domain is tried. When all values in the domain of the current variable fail to extend the partial solution, a dead-end state is encountered and BT backtracks one level to the most recently instantiated variable, revokes the value assigned to that variable and continues at that stage. BT reports the problem is insoluable if a dead-end state is encountered at level 1.

2.5.2 Conflicts-directed Backjumping (CBJ)

An *inconsistency* or a *no-good* is a partial solution that does not appear in any solution. If the current partial solution is found to contain a no-good, it cannot be extended to a full solution and some variables must be chosen to be removed from the current partial solution to invalidate the no-good. A dead-end state indicates that the current partial solution t failed to extend to the current variable, and thus t is found to be a no-good. The backtracking step in BT removes the most recent variable from t and invalidates the no-good. However, if t has a no-good subtuple not including that variable, BT will inevitably fail to extend the current branch. A *minimal no-good* does not have no-good subtuples. To compute the minimal no-good from a dead-end state is computationally prohibitive. However, there are many

```
procedure update_constraint_counts( in x : variable);
    %% As x is chosen to be instantiated next, update (decrease) the number
    %% of uninstantiated variables for each of the constraints involving x.
    for each C ∈ 𝒞 and x ∈ vars(C) do count_uninst[C] ← count_uninst[C] − 1;

procedure restore_constraint_counts( in x : variable);
    %% As a backtracking occurs at the current level, restore (increase) the number
    %% of uninstantiated variables for each of the constraints involving x.
    for each C ∈ 𝒞 and x ∈ vars(C) do count_uninst[C] ← count_uninst[C] + 1;

function consistent( in current : integer) : boolean;
    %% check whether it is possible to extend the current partial solution to
    %% a full solution.
1   x ← order[current];
2   for each C ∈ 𝒞 and x ∈ vars(C) do
3       if count_uninst[C] = 0 then
4           if not check_constraint( C, solution ) then return false;
5   return true;

function BT( in current : integer ) : boolean;
1   if current > n then return true;
2   x ← get_next_var( current ); order[current] = x;
3   update_constraint_counts( x );
4   for each a ∈ dom(x) do
5       solution[x] ← a;
6       instantiated[x] ← true;
7       if consistent( current ) then
8           if BT( current + 1 ) then return true;
9       instantiated[x] ← false;
10  restore_constraint_counts( x );
11  return false;
```

Figure 2.3: BT.

```
function consistent( in current : integer) : boolean;
    %% check whether it is possible to extend the current partial solution to
    %% a full solution.
1   x ← order[current];
2   for each C ∈ 𝒞 and x ∈ vars(C) do
3       if count_uninst[C] = 0 then
4           if not check_constraint( C, solution ) then
5               cs[x] ← cs[x] ∪ (vars(C) − {x});
6               return false;
7   return true;

function  CBJ( in current : integer ) : integer;
1   if current > n then return true;
2   x ← get_next_var( current ); order[current] = x;
3   update_constraint_counts( x );
4   for each a ∈ dom(x) do
5       solution[x] ← a;
6       instantiated[x] ← true;
7       if consistent( current ) then
8           j ← CBJ( current );
9           if j ≠ current then
10              instantiated[x] ← false;
11              restore_constraint_counts( x );
12              return j;
13      instantiated[x] ← false;
14  j ← max{ i | 1 ≤ i ≤ current and order[i] ∈ cs[x] };
15  cs[order[j]] ← (cs[order[j]] ∪ cs[x]) − {order[j]};
16  for i ← j + 1 to current do cs[order[i]] ← ∅;
17  restore_constraint_counts( x );
18  return j;
```

Figure 2.4: CBJ.

ways to cleverly use the no-goods information discovered in the consistency checks. *Backjumping* (BJ) [53, 54] computes a set of past variables, called *conflicts*, which contributed to some failures in the consistency check for the current variable. Every time the current partial solution fails to satisfy a constraint, the variables except the current variable in the scheme of the constraint are added to the conflict set. If all values of the current variable failed in the consistency check, BJ jumps back to the highest variable in the conflicts set. BJ backjumps only from the special case of dead-end states. All other backtracks are chronological. To perform a "multiple backjumpings" from any dead-end states, conflicts-directed backjumping (CBJ) [91] maintains for each past variable (and the current variable) its own *conflicts set*. We use $cs[x]$ to denote the set of past variables in the current conflicts set of variable x. In a dead-end state, CBJ backjumps to the highest variable, called the *culprit variable*, in the conflicts set of the current variable. At the same time, the conflicts set of the current variable is merged into the conflicts set of the culprit variable. The pseudo code of CBJ is shown in Figure 2.4.

2.5.3 Forward Checking (FC)

BT and CBJ perform consistency checks backward; that is, a constraint is checked only if all the variables in its scheme have been instantiated. In constrast, the forward checking algorithm (FC) [60] performs consistency checks forward; *i.e.*, a constraint is chosen to be checked even if some of its variables have not been instantiated. Generally, a constraint is *forward checkable* at the current state if all but one of its variables was instantiated [8]. The uninstantiated variable is called the *forward checked variable* in the constraint. In the consistency checks of FC, for each forward checkable constraint, the domain of the forward checked variable is filtered in the following way: for each value in the domain, if the instantiation of the forward checked variable with that value along with the instantiations in the current partial solution do not satisfy the constraint, the value is temporarily removed from or marked *inactive* in its domain at the current level. The consistency check fails if the domain of a forward checked variable is found to be empty, which is called a *domain wipe out* (dwo). If the current partial solution fails a consistency check or later in extending to a full solution, the effect of forward checking is undone; *i.e.*, all the values removed from the domains of future variables in the forward checking at the current level are restored in their

[8] Following Van Hentenryck [114], we say that a *k-ary* constraint, $k \geq 2$, is *forward checkable* if $k-1$ of its variables have been instantiated and the remaining variable is uninstantiated.

```
procedure  restore( in current : integer);
1   x ← order[current];
2   for each y ∈ 𝒱 and instantiated[y] do
3      if checking[x][y] > 0 then
4         checking[x][y] ← 0;
5         for each a ∈ dom(y) and domains[y][a] = current do
6            domains[y][a] ← 0;
7            domain_count[y] ← domain_count[y] + 1;

function check_forward( in C : constraint; in current : integer) : boolean;
1   x ← order[current];
2   y ← the uninstantiated variable in vars(C);
3   changed ← false;
4   for each a ∈ dom(y) and domains[y][a] = 0 do
5      solution[y] ← a;
6      if not check_constraint( C, solution ) then
7         changed ← true;
8         domains[y][a] ← current;
9         domain_count[y] ← domain_count[y] − 1;
10  if changed then checking[x][y] = current;
11  if domain_count[y] = 0 then return false;
12  else return true;

function consistent( in current : integer) : boolean;
1   x ← order[current];
2   for each C ∈ 𝒞 and x ∈ vars(C) do
3      if count_uninst[C] = 1 then
4         if not check_forward( C, current ) then return false;
5   return true;
```

Figure 2.5: FC.

```
function FC( in current : integer ) : boolean;
1  if current > n then return true;
2  x ← get_next_var( current ); order[current] = x;
3  update_constraint_counts( x );
4  for each a ∈ dom(x) and domains[x][a] = 0 do
5     solution[x] ← a;
6     instantiated[x] ← true;
7     if consistent( current ) then
8        if FC( current + 1 ) then return true;
9     restore( current );
10    instantiated[x] ← false;
11 restore_constraint_counts( x );
12 return false;
```

Figure 2.5: FC.

domains. FC backtracks chronologically in the case of dead-ends.

The pseudo code of FC is shown in Figure 2.5. FC uses three additional data structures beside those in BT, *domains*, *domain_count* and *checking*. $domains[x][a]$ denotes whether value a is active in the domain of variable x, where $domain[x][a] = 0$ indicates that a is still an active value in its domain, and $domain[x][a] = i > 0$ indicates that a has been removed from the domain at level i of the backtrack search. $domain\_count[x]$ records the number of active values in the domain of variable x. For example, if value a is marked inactive in the domain of x at level i, $domain[x][a]$ is set to be i to indicate that a is inactive now and $domain\_count[x]$ is deducted by 1. Thus, a dwo can be found if the condition $domain\_count[x] = 0$ is satisfied for some future variable x. $checking[x][y]$ is set to be i if at level i the instantiation of the current variable x makes some constraint become forward checkable and the domain of the forward checked variable y is pruned in forward checking. When the instantiation of x is revoked, $checking[x][y]$ is restored to be 0, and for each value a in the domain of y, if condition $domain[y][a] = i$ is true, $domains[y][a]$ is restored to be 0 and $domain\_count[y]$ is increased by 1.

2.5.4 Generalized Maintaining Arc Consistency (GAC)

GAC performs at each node in the search tree, one full cycle of arc consistency. An arc consistency achievement algorithm is applied to the problem instantiated with the current assignments and the tentative value of the current variable being considered.

```
procedure restore( in current : integer);
1  x ← order[current];
2  for each y ∈ V and instantiated[y] do
3     if checking[x][y] > 0 then
4        checking[x][y] ← 0;
5        for each a ∈ dom(y) and domains[y][a] = current do
6           domains[y][a] ← 0;
7           domain_count[y] ← domain_count[y] + 1;

function check_forward( in C : constraint; in current : integer) : boolean;
1  x ← order[current];
2  y ← the uninstantiated variable in vars(C);
3  changed ← false;
4  for each a ∈ dom(y) and domains[y][a] = 0 do
5     solution[y] ← a;
6     if not check_constraint( C, solution ) then
7        changed ← true;
8        domains[y][a] ← current;
9        domain_count[y] ← domain_count[y] − 1;
10 if changed then
11    checking[x][y] = 1; push( y, S );
12 if domain_count[y] = 0 then return false;
13 else return true;

function exists( in C : constraint; in v : variable ) : boolean;
   %% return ture if solution[v] has a valid support in constraint C.

function revise( in C: constraint; in v : variable;
                 in current : integer) : boolean;
1  x ← order[current];
2  changed ← false;
3  for each a ∈ dom(v) and domains[v][a] = 0 do
4     solution[v] ← a;
5     if not exists( C, v ) then
6        changed ← true;
7        domains[v][a] ← current;
8        domain_count[v] ← domain_count[v] − 1;
9  if changed then
10    checking[x][v] = 1; push( v, S );
11 if domain_count[v] = 0 then return false;
12 else return true;
```

Figure 2.6: GAC.

```
function consistent( in current : integer) : boolean;
1   S ← ∅;
2   x ← order[current];
3   for each C ∈ 𝒞 and x ∈ vars(C) do
4      if count_uninst[C] = 1 then
5         if not check_forward( C, current ) then return false;
6   push( x, S );
7   while S ≠ ∅ do
8      y ← top( S ); pop( S );
9      for each C ∈ 𝒞 and y ∈ vars(C) do
10        if count_uninst[C] ≥ 2 then
11           for each v ∈ vars(C) and not instantiated[v] and v ≠ y do
12              if not  revise( C, v, current ) then return false;
13  return true;

function  GAC( in current : integer ) : boolean;
1   if current > n then return true;
2   x ← get_next_var( current ); order[current] = x;
3   update_constraint_counts( x );
4   for each a ∈ dom(x) and domains[x][a] = 0 do
5      solution[x] ← a;
6      instantiated[x] ← true;
7      if consistent( current ) then
8         if GAC( current + 1 ) then return true;
9      restore( current );
10     instantiated[x] ← false;
11  restore_constraint_counts( x );
12  return false;
```

Figure 2.6: GAC.

If, as a result, one of the future domains becomes empty, the tentative value will lead to a dead-end and it should be excluded. If none of the future domains becomes empty, the tentative value will be selected. The pseudo code of generalized maintaining arc consistency (GAC) is shown in Figure 2.6. As achieving arc consistency is a costly process, the general wisdom is to perform cheap consistency checks first; that is, a forward checking phase is performed before the full propagation. A constraint is said to be *arc consistency checkable* at the current state if at least two of its variables have not been instantiated [9]. Again, $count\_inst[C]$ is used to determine whether a constraint should participate in the arc consistency check at the current level.

2.5.5 Forward Checking with Conflict-directed Backjumping (FC-CBJ)

Both look-backward and look-ahead strategies make a tradeoff, doing extra work at one phase of the backtracking search in order to reduce the amount of work required later. This extra work, however, frequently leads to a significantly reduced search space. It is reasonable to conjecture that a combination of improvement techniques will be useful on some problems. A hybrid of FC and CBJ, known as FC-CBJ [92], outperforms its parents by several orders of magnitude on many applications.

Unlike CBJ, FC-CBJ performs consistency checks on future variables. It is convenient to divide the conflicts information into two pieces. The no-goods found in the forward checking phase are recorded in *checking*, in which $checking[x][y]$ denotes the instantiation of variable x has caused some constraints to be forward checkable and the domain of the forward checked variable y is pruned. However, *checking* is used differently here than in FC and GAC. FC and GAC only need to restore those inactive values in future domains which are pruned as a result of the instantiation of the current variable. For the current variable x and the pruned variable y , $checking[x][y]$ set to be the current level i is enough for such a purpose. With forward checking enhanced with backjumping, at level i, once a future variable y is forward checked in constraint C, for each of the instantiated variables $x \in vars(C)$, if $checking[x][y] = 0$, $checking[x][y]$ is set to be i to record a complete no-good. Because x may have multiple chances to participate in a forward checkable constraint to forward check against y[10], FC-CBJ only keeps the lowest level checking occurrence;

[9] A more sophisticated control of arc consistency checkability can be determined, for example, by the numbers of active values in the domains of uninstantiated variables.

[10] For instance, the variable x_1 can check against x_4 twice by two constraints, $C(x_1, x_2, x_4)$ and $C(x_1, x_3, x_4)$, and two forward checkings occur at different levels of the search tree.

```
procedure restore(in current: integer);
1   x ← order[current];
2   for i ← current + 1 to n do
3       y ← order[i];
4       if checking[x][y] = current then
5           for each a ∈ dom(y) and domain[y][a] = current do domain[y][a] ← 0;
6           for j ← 1 to current do
7               v ← order[j];
8               if checking[v][y] = current then checking[v][y] = 0;

procedure record_checking( in C: constraint; in y: variable; in current: integer);
    %% bookmark the fact that the domain of y was pruned due to the propagation
    %% on the constraint C.
1   for each v ∈ vars(C) and v ≠ y do
2       if checking[v][y] = 0 then checking[v][y] ← current;

function check_forward( in C : constraint; in current : integer;
                        out fail : variable) : boolean;
1   x ← order[current];
2   y ← the uninstantiated variable in vars(C);
3   changed ← false;
4   for each a ∈ dom(y) and domains[y][a] = 0 do
5       solution[y] ← a;
6       if not check_constraint( C, solution ) then
7           changed ← true;
8           domains[y][a] ← current;
9           domain_count[y] ← domain_count[y] − 1;
10  if changed then record_checking( C, y, current );
11  fail ← y;
12  if domain_count[y] = 0 then return false;
13  else return true;
```

Figure 2.7: FC-CBJ.

```
function consistent( in current : integer; out fail : variable) : boolean;
1   x ← order[current];
2   for each C ∈ 𝒞 and x ∈ vars(C) do
3      if count_uninst[C] = 1 then
4         if not check_forward( C, current, fail ) then return false;
5   return true;

function  FC_CBJ( in current : integer ) : boolean;
1   if current > n then return true;
2   x ← get_next_var( current ); order[current] = x;
3   update_constraint_counts( x );
4   for each a ∈ dom(x) and domains[x][a] = 0 do
5      solution[x] ← a;
6      instantiated[x] ← true;
7      if consistent( current, fail ) then
8         j ← FC_CBJ( current + 1 );
9         if j ≠ current then
10           restore( current );
11           instantiated[x] ← false;
12           restore_constraint_counts( x );
13           return j;
14        else
15           cs[x] ← cs[x] ∪ { y | instantiated[y] and checking[y][fail] ≠ 0};
16     restore( current );
17     instantiated[x] ← false;
18  cs[x] ← cs[x] ∪ { y | instantiated[y] and checking[y][x] ≠ 0};
19  j ← max{ i | 1 ≤ i ≤ current and order[i] ∈ cs[x] };
20  cs[order[j]] ← (cs[order[j]] ∪ cs[x]) − {order[j]};
21  for i ← j + 1 to current do cs[order[i]] ← ∅;
22  restore_constraint_counts( x );
23  return j;
```

Figure 2.7: FC-CBJ.

that is, $checking[x][y] = i$ indicates that x first checked against y by some constraint at level i. Thus until a backtracking occurs at level i, we know that x is still in the conflicts set of y. The no-goods found in the backward phase, *i.e.*, those from a conflicts set of a high level variable in a backjumping, are still recorded in the data structure cs. When a dead-end is encountered, these two pieces of information are merged together to form a complete conflicts set for the current variable. The pseudo code of FC-CBJ is shown in Figure 2.7.

Chapter 3

Look-ahead and Backjumping

The techniques for improving the naive backtracking algorithm can be conveniently classified as *look-ahead schemes* and *look-back schemes* [34]. Look-ahead schemes are invoked whenever the algorithm is preparing to assign a value to the current variable or to choose the next variable to be instantiated. In general, look-ahead schemes involve enforcing a certain level of consistency (on the subproblem consisting of all the future variables), using a dynamic variable ordering and using a dynamic value ordering heuristic. Look-back schemes are invoked whenever the algorithm encounters a dead-end and prepares for the backtracking step. Look-back schemes perform the functions that decide how far to backtrack by analyzing the reasons for the dead-end, and record the reasons for the dead-end in the form of new constraints so that the same conflicts will not arise again later in the search. Unfortunately, sometimes the look-ahead schemes are counterproductive to the look-back schemes, as it is well believed [60] that : "Look ahead to the future in order not to worry about the past." That is, the more we do in the forward phase, the less we can save in the backward phase. For example, Bacchus and van Run [8] observe that adding CBJ to an algorithm that already uses a dynamic variable ordering based on the minimal domain heuristic is unlikely to yield much improvement. They explain that the use of the minimal domain heuristic will tend to cluster conflicted variables together, and hence CBJ is unlikely to generate large backjumps and its savings are likely to be minimal. Also, in [16], Bessière and Régin state that: "CBJ was cheap to incorporate in BT, it was not prohibitive in FC, but it palpably slows down the search in MAC." Thus they conjectured that "when MAC and a good variable ordering heuristic are used, CBJ becomes useless."

However, all the above observations are based on experimental results, and they have never been justified theoretically. There is a preliminary result in Kondrak

and van Beek's work [69] saying that: "Given a binary CSP and a static variable ordering, FC always visits fewer nodes than BJ does." Furthermore, the previous experimental results for CBJ have been limited to random and toy problems, which are usually formulated as binary CSPs. For example, Bacchus and van Run [8] use the *zebra* problem, n-Queens problem, and random binary problems to evaluate the effect of the minimal domain heuristic on several backtracking algorithms. They observe that CBJ provides hardly any savings if the minimal domain heuristic is used in the backtracking search. Bessière and Régin's conclusion about MAC-CBJ is solely based on experiments on random binary CSPs. On the other hand, it has been observed that look-back techniques, including backjumping and learning mechanisms, can dramatically improve problem solving on hard 3-SAT problems and real-world planning problems [10, 11].

This chapter presents three results that deepen our understanding of the relationship between look-back and look-ahead schemes. First, we show by example that CBJ may be exponentially better than an algorithm that maintains strong k-consistency in the backtrack search and we show that backjumping becomes useless if an appropriate variable ordering strategy is used in the chronological backtracking algorithm. Second, we introduce the concept of *backjump level* in the execution of a backjumping algorithm and some background results for maintaining strong k-consistency. Then we show that an algorithm maintaining strong k-consistency always visits no more nodes than a backjumping algorithm that is allowed to backjump no more than k levels. Third, we introduce a new backjumping algorithm, named GAC-CBJ, which is an extension of Prosser's MAC-CBJ [93] to general CSPs. We show by experimental results that for some real world problems, GAC-CBJ can provide a huge amount of improvement over GAC.

3.1 CBJ and Variable Ordering

Experimental comparisons have shown that CBJ is, on average, not competitive with look-ahead algorithms, such as FC and MAC [8, 16, 92]. For example, the experimental results in [8] show that CBJ usually runs twice as slow as FC (in terms of the number of the constraint checks performed) when solving the *zebra* problem, n-Queens problem, and random binary problems. However, as the next example shows, CBJ has the potential to defeat many look-ahead algorithms.

Example 3.1 *Given a fixed integer k, we can construct a binary CSP with $n+k+2$*

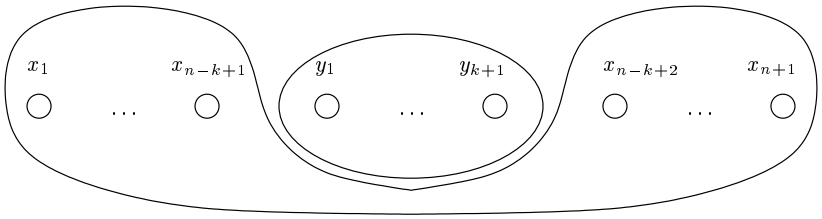


Figure 3.1: A CSP mixed with two pigeon-hole problems.

variables, $x_1, \ldots, x_{n-k+1}, y_1, \ldots, y_{k+1}, x_{n-k+2}, \ldots, x_{n+1}$, *where* $dom(x_i) = \{1, \ldots, n\}$ *for* $1 \leq i \leq n+1$ *and* $dom(y_j) = \{1, \ldots, k\}$ *for* $1 \leq j \leq k+1$. *The constraints are:* $x_i \neq x_j$, *for* $i \neq j$, *and* $y_i \neq y_j$, *for* $i \neq j$. *The problem consists of two separate pigeon-hole subproblems, one over variables* $x_1, \ldots, x_{n+1}$ *and the other over variables* $y_1, \ldots, y_{k+1}$, *and is insoluble. As we know, the pigeon-hole problem is highly locally consistent [110]. The first subproblem is strongly n-consistent and the second is strongly k-consistent. Under the above static variable ordering, a backtracking algorithm maintaining strong k-consistency would not encounter a dead-end until* x_{n-k+1} *is instantiated. Then it would find that the subproblem of* $x_{n-k+2}, \ldots, x_{n+1}$ *is not strongly k-consistent. Thus the algorithm will backtrack before it reaches the second pigeon-hole subproblem. It will explore* $\frac{n!}{k!}$ *nodes at level n-k+1 of the search tree and thus take an exponential number of steps to find the problem is insoluble. CBJ does not encounter a dead-end at the level of* x_{n-k+1} *and it continues to the second pigeon-hole problem. Eventually it will find the second-pigeon hole problem is insoluble and backjump to the root of the search tree. The total number of nodes explored is bounded by a constant,* $O((k+1)^k)$, *for a fixed k. Therefore, CBJ can be exponentially better than an algorithm maintaining strong k-consistency.*

Independently, Bacchus and Grove present a similar example in [6] to show that given a fixed k, CBJ may be exponentially better than an algorithm called $MlkC$, which essentially maintains k-consistency in the backtrack search for binary CSPs.

Theorem 3.1 *For any fixed integer k, there is a CSP instance and a static variable ordering such that CBJ visits exponentially fewer nodes than an algorithm that maintains strong k-consistency in the backtrack search.*

Proof: It is true from the CSP in Example 3.1. ∎

One may argue that in the above example, if FC or MAC explores the smaller pigeon-hole problem first, it could perform much better under the new variable order-

ing. It is true that if an appropriate variable ordering strategy is used, an algorithm doing chronological backtracking usually backtracks to the most relevant variable to the current dead-end such that backjumping becomes less useful. For example, it was said in [8]: "CBJ is unlikely to generate large backjumps and its savings are likely to be minimal if we use a good variable ordering heuristic."

We are going to justify the above statement more precisely. That is, given a CSP instance and a variable ordering strategy for CBJ, there is a variable ordering strategy for the chronological backtracking algorithm (BT) such that BT visits no more nodes than CBJ does. We first consider the case of insoluble CSPs. When CBJ is applied to solve an insoluble CSP, it always backjumps from a dead-end state; *i.e.*, it will not terminate or backjump from a situation in which a solution of the CSP is found.

Lemma 3.2 *Given an insoluble CSP instance and a (possibly dynamic) variable ordering strategy for CBJ, there is a (possibly dynamic) variable ordering strategy for BT, such that BT visits no more nodes than CBJ to solve the CSP.*

Proof: In the backtrack tree generated by CBJ under the variable ordering strategy, let the last backjump that terminates the execution of CBJ be from variable x_j to the root of the backtrack tree. We choose x_j to be the first variable for BT. For each value a in the domain of x_j, the next variable chosen to be instantiated after assigning a to x_j is the variable that backjumps to x_j and causes the assignment $x_j \leftarrow a$ to be revoked. For example, in Figure 3.2, the first variable chosen for BT is x_j. After assigning value a to x_j, variable x_{j_a} is instantiated next, and so on. The entire variable ordering for BT can be worked out in a recursive manner. The only situation where BT could possibly be unable to follow the above ordering is if, at some stage, CBJ finds out that the current node is inconsistent so that there is no such backjump from a higher level variable to the current variable, but because BT instantiates fewer variables along the path from the root to the dead-end, it might not be able to detect the inconsistency and so it has to extend the current node. We can prove that such a situation does not exist in the ordering constructed for BT. That is, the variables skipped in the variable ordering constructed for BT are irrelevant to the dead-end states encountered by CBJ. Suppose at a stage we have ordered the variables to be instantiated for BT as $x_{j_1}, \ldots, x_{j_k}$, and for value $a \in dom(x_{j_k})$ we choose the next variable $x_{j_{k+1}}$ as the variable which backjumps to the current variable x_{j_k} in the CBJ backtrack tree. We will prove by induction that the conflicts set of $x_{j_{k+1}}$ used in the backjumping is subsumed by $\{x_{j_1}, \ldots, x_{j_k}\}$. $k = 1$ is the case of the last backjump that terminates the execution of CBJ. The hypothesis is true because the

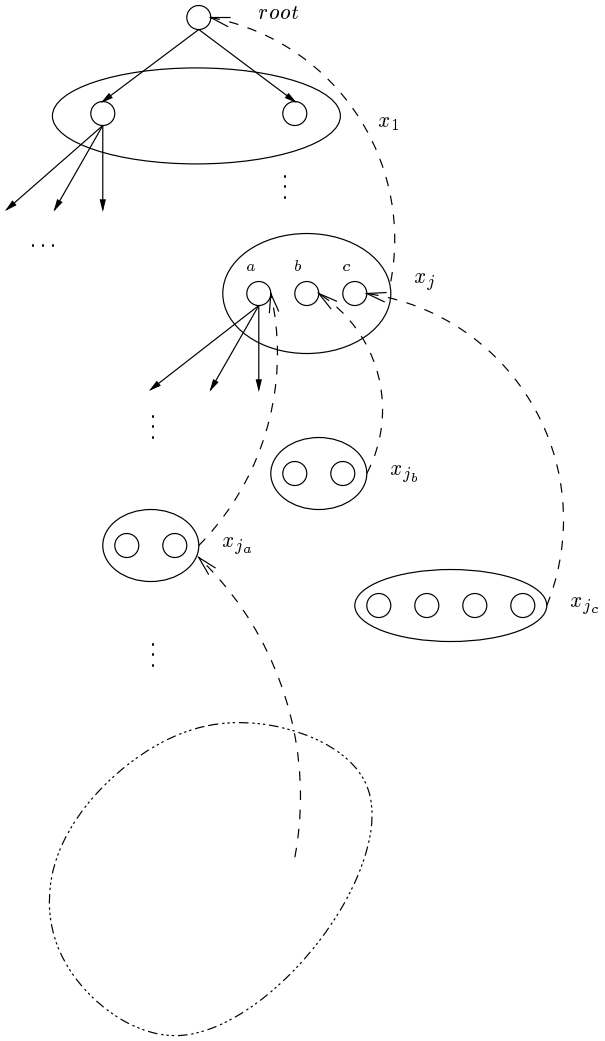


Figure 3.2: A backtrack tree generated by CBJ to solve an insoluble CSP.

conflicts set of x_{j_1} is an empty set. Suppose it is true for the case of k. Because $x_{j_{k+1}}$ backjumps to x_{j_k}, the conflicts set of $x_{j_{k+1}}$ is merged in the conflicts set of x_{j_k}. From the inductive assumption, the conflicts set of x_{j_k} is subsumed by $\{x_{j_1}, \ldots, x_{j_{k-1}}\}$, and thus the conflicts set of $x_{j_{k+1}}$ is subsumed by $\{x_{j_1}, \ldots, x_{j_k}\}$. Therefore, the hypothesis holds for the case of $k+1$. If CBJ finds out that instantiation $x_{j_k} \leftarrow a$ is inconsistent with the assignments of some past variables which are added to the conflicts set of x_{j_k}, BT is also able to find out the inconsistency because the conflicts set of x_{j_k} is subsumed by $\{x_{j_1}, \ldots, x_{j_{k-1}}\}$. Thus it is a feasible variable ordering for BT. Under such a variable ordering, BT visits no more nodes than CBJ does. ∎

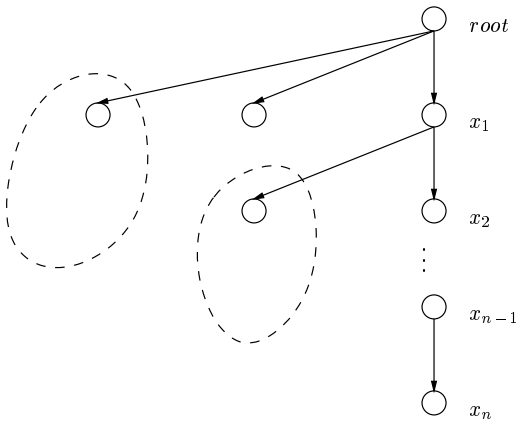


Figure 3.3: A backtrack tree generated by CBJ to find one solution.

For soluble CSPs, we further distinguish the problem between finding one solution and finding all solutions. When just one solution is required, CBJ will stop once it has found the first solution, $\{x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n\}$, as shown in Figure 3.3.

Lemma 3.3 *Given a CSP instance and a variable ordering for CBJ to find the first solution, there is a variable ordering strategy for BT such that BT will visit no more nodes than CBJ to find the first solution.*

Proof: A variable ordering for BT can be constructed in the following way: The first variable chosen for BT is x_1 as it is the first variable in the path from the root to the solution in the CBJ backtrack tree. Because we assume a static value ordering in the backtrack search, all values in the domain of x_1 that precede value a_1 must be rejected by CBJ and BT before value a_1 is used to instantiate x_1. Furthermore, because $\{x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n\}$ is the first solution encountered by CBJ under the above variable ordering and value ordering, the instantiation of x_1 with a value preceding a_1 leads to an insoluble subproblem and eventually CBJ will backjump from a higher level variable to x_1 to revoke that assignment. Note that x_1 cannot be skipped by a backjump from a higher level variable because x_1 is on the first level of the search tree and there is a solution for the CSP. We can arrange the instantiation order for BT in the insoluble subproblem, after assigning x_1 with each of the values that precede a_1 in its domain. Whenever x_k is instantiated with value a_k, x_{k+1} is chosen to be the next variable, as it follows x_k in the path from the root to the solution in the CBJ backtrack tree. Again, all values in the domain of x_{k+1} that precede a_{k+1} in the value ordering must be rejected by CBJ and BT before a_{k+1} is assigned to x_{k+1}. The instantiation of x_{k+1} with each of these values will lead to an

insoluble subproblem and eventually CBJ will backjump from a higher level variable to x_{k+1}. Similarly, x_{k+1} cannot be skipped by a backjump from a higher level variable because otherwise at least one of the assignments to $x_1, \ldots, x_k$ must be changed so that $\{x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n\}$ is not the first solution encountered by CBJ. We can arrange the instantiation order for BT in these insoluble subproblems. Finally, x_n is instantiated with a_n and BT finds the solution. Under the above ordering, BT will visit no more nodes than CBJ does. ∎

When CBJ is used to find all solutions, special steps must be taken to handle the conflicts sets. The problem here is that the conflict sets of CBJ are meant to indicate which instantiations are responsible for some previously discovered inconsistency. However, after a solution is found, conflict sets cannot always be interpreted in this way. It is the search for other solutions, rather than an inconsistency, that causes the algorithm to backtrack. We need to differentiate between two causes of CBJ backtracks: (1) detecting an inconsistency, and (2) searching for other solutions. In the latter case, the backtrack must be always chronological; that is, to the immediately preceding variable. A simple solution is to remember the number of solutions found so far when a variable is chosen to be instantiated, and later when a dead-end state is encountered at this level, we compare the recorded number with the current number of solutions. A difference indicates that some solutions have been found in this interval of search, and forces the algorithm to backtrack chronologically. Otherwise the algorithm performs a normal backjumping by analyzing the conflicts set of the current variable.

Lemma 3.4 *Given a CSP instance and a variable ordering for CBJ to find all solutions, there is a variable ordering strategy for BT such that BT will visit no more nodes than CBJ to find all solutions.*

Proof: Let the first solution found by CBJ be $\{x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n\}$ in the order of $x_1, \ldots, x_n$. We first construct the variable ordering for BT as it is applied to find the first solution. However, because BT follows a strict chronological backtracking, it will inevitably visit all the nodes $\{x_1 \leftarrow a_1, \ldots, x_{j-1} \leftarrow a_{j-1}, x_j \leftarrow a'_j\}$, where $1 \leq j \leq n$ and a_j precedes a'_j in the domain of x_j. If CBJ skips any of these nodes, for example, from a higher level variable x_h to x_{j-1}, while the instantiations of $x_1, \ldots, x_j$ have not been changed, BT will possibly visit more nodes than CBJ does. We will show this cannot happen by induction on the distance between the current level j and the highest level n. After CBJ has found the solution at level n, it will

try other values for x_n and eventually backtrack to x_{n-1}. So the nodes at level n cannot be skipped. Suppose it is true for the case of level $j+1$ and now we consider the case of level j. Because $x_j \leftarrow a_j$ was not skipped in the backjumping, if a_j is the last value in its domain, CBJ will backtrack to x_{j-1} because the number of solutions has been changed. So it is true for the case of j. Otherwise CBJ will change the instantiation of x_j to the next value in its domain. Let the current partial solution be $t = \{x_1 \leftarrow a_1, \ldots, x_{j-1} \leftarrow a_{j-1}, x_j \leftarrow a'_j\}$. If the subtree rooted by t contains solutions, from the inductive hypothesis, CBJ will not skip this node because it is on level j. If the subtree rooted by t contains no solution, there exists a backjump from a higher level variable x_h to escape this subtree. Could it jump beyond x_j such that t is skipped? In that case, the conflicts set of x_h is subsumed in $\{x_1, \ldots, x_{j-1}\}$. From the definition of conflicts set, we know that the current instantiations of the variables in the conflicts set cannot lead to a solution. However the current instantiations of $\{x_1, \ldots, x_{j-1}\}$ do lead to a solution, $\{x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n\}$. That is a contradiction. So the conflicts set of x_h must contain x_j and thus the node t at level j cannot be skipped. After all the values in the domain of x_j have been tried, CBJ will chronologically backtrack to x_{j-1} because the number of solutions has changed. Thus $x_{j-1} \leftarrow a_{j-1}$ will not be skipped. The hypothesis is true for the case of any level j. Then we construct the variable ordering for BT in the following way: If the current partial solution $t = \{x_1 \leftarrow a_1, \ldots, x_{j-1} \leftarrow a_{j-1}, x_j \leftarrow a'_j\}$ cannot be extended to a solution, we construct a variable ordering for the insoluble subproblem. If t can be extended to a solution, we construct a variable ordering for BT as the case of finding the first solution in this subproblem, and recursively apply the above steps until a backjump to level x_j changes the instantiation $x_j \leftarrow a'_j$. Under the above variable ordering, BT will visit no more nodes than CBJ does. ∎

Theorem 3.5 *Given a CSP instance and a variable ordering for CBJ, there is a variable ordering strategy for BT such that BT will visit no more nodes than CBJ to solve the CSP.*

Proof: It is straightforward from Lemma 3.2, Lemma 3.3, and Lemma 3.4. ∎

Example 3.2 *Figure 3.4 shows the BT backtrack tree based on the variable ordering constructed from the execution of CBJ to solve the CSP in Example 2.1 under a (hypothetical) dynamic variable ordering. The first solution found by CBJ is* $\{x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 2, x_5 \leftarrow 0, x_4 \leftarrow 0\}$. *Thus BT first instantiates* x_1 *and* x_2 *to* 0. *The*

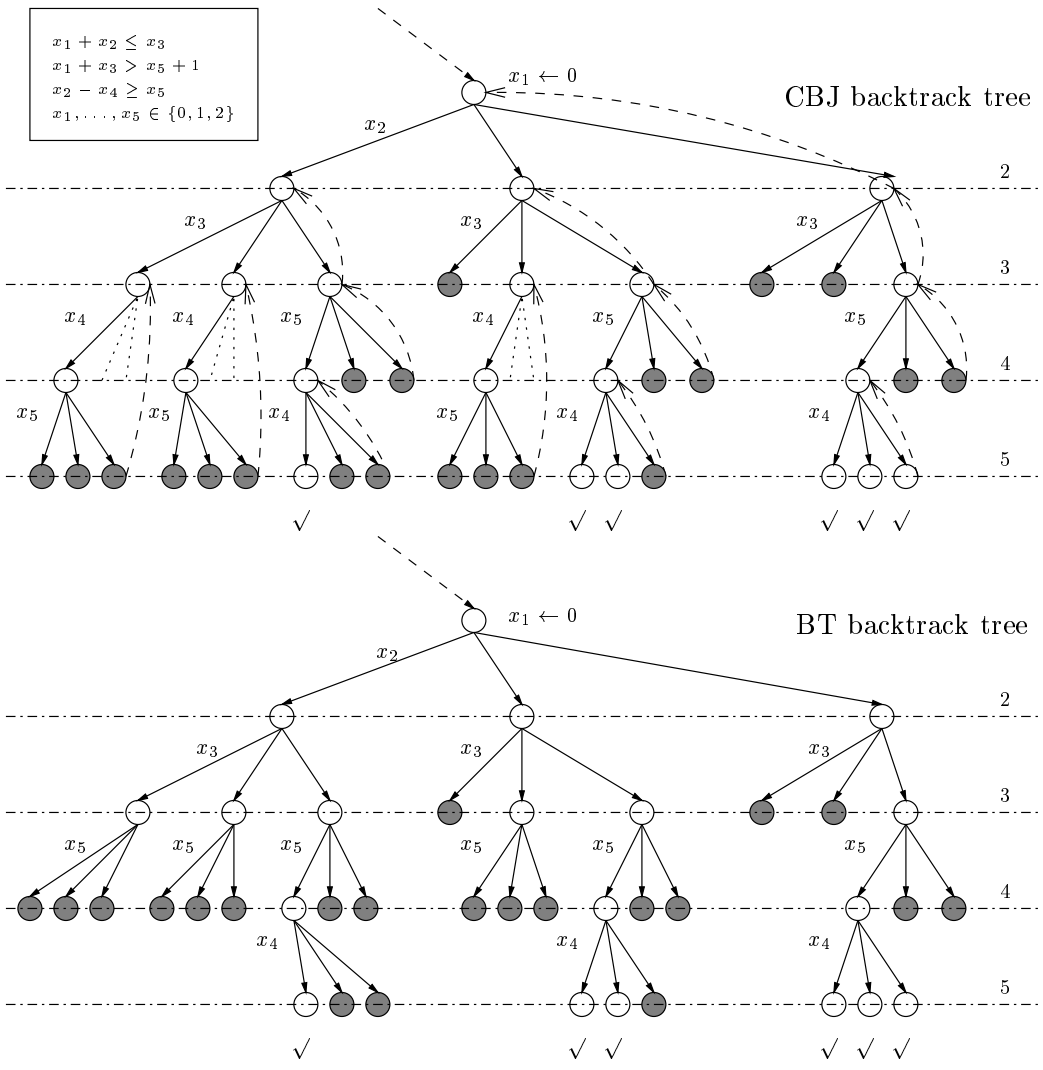


Figure 3.4: An example of the variable ordering constructed for BT from the CBJ backtrack tree.

node $\{x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 0\}$ *and* $\{x_1 \leftarrow 0, x_2 \leftarrow 0, x_2 \leftarrow 1\}$ *in the CBJ backtrack tree lead to two insoluble subproblems. The variable ordering for BT at each of these nodes is constructed as in the case of insoluble CSPs. For example, in the CBJ backtrack tree, the last backjump to revoke the node* $\{x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 0\}$ *is from* x_5 *to* x_3*, so the next variable instantiated in BT at this node is* x_5*. Under such an ordering, BT avoids instantiating* x_4 *and visits fewer nodes than CBJ. Then BT instantiates* x_3 *to* 2*,* x_5 *to* 0*, and* x_4 *to* 0*, and finds the first solution. As we can see in the above figure, after CBJ finds the first solution, denoted by* $\{x'_1 \leftarrow a_1, \ldots, x'_n \leftarrow a_n\}$*, none of the nodes* $\{x'_1 \leftarrow a_1, \ldots, x'_{j-1} \leftarrow a_{j-1}, x'_j \leftarrow a'_j\}$*, where* $j \leq n$ *and* a_j *precedes* a'_j *in the domain of* x'_j*, is skipped by CBJ. Thus, both BT and CBJ will visit these nodes and the variable ordering for BT at each of these nodes is constructed in the same way as in the case for the insoluble subproblems or the case of finding the first solution in the subproblem.*

Therefore, the effect of backjumping may be degraded by the use of an appropriate variable ordering. Of course, the above "perfect" variable ordering strategy for BT will, in general, not be known until the completion of CBJ. So we have used too much magic to make BT perform better than CBJ. Also, in practice, it is not our primary goal to devise a variable ordering that enables a chronological backtracking algorithm to simulate the execution of the CBJ, but to find a variable ordering that can greatly improve the backtrack search. There are many efficient heuristics for solving CSPs. For example, the fail first heuristic selects the next variable to be instantiated with the minimal remaining domain size. As a result, variables that have conflicts with past instantiations are likely to be instantiated sooner, and thus the conflict variables tend to be clustered together in the backtrack search. Hence, CBJ is unlikely to generate large backjumps [8]. However, the fail first heuristic is not always consistent with the above "perfect" variable ordering. Hypothetically, because we do not have the "perfect" variable ordering *a priori*, or we do not want to use it in the backtrack search even if we could find one, CBJ still has the chance to improve the search and sometimes it can be dramatically better than an algorithm doing chronological backtracking.

3.2 Backjump Level and $\mathbf{BJ}_k$

From Theorem 3.1, we know that under a static variable ordering, CBJ can perform much better than a look-ahead algorithm that maintains strong k-consistency

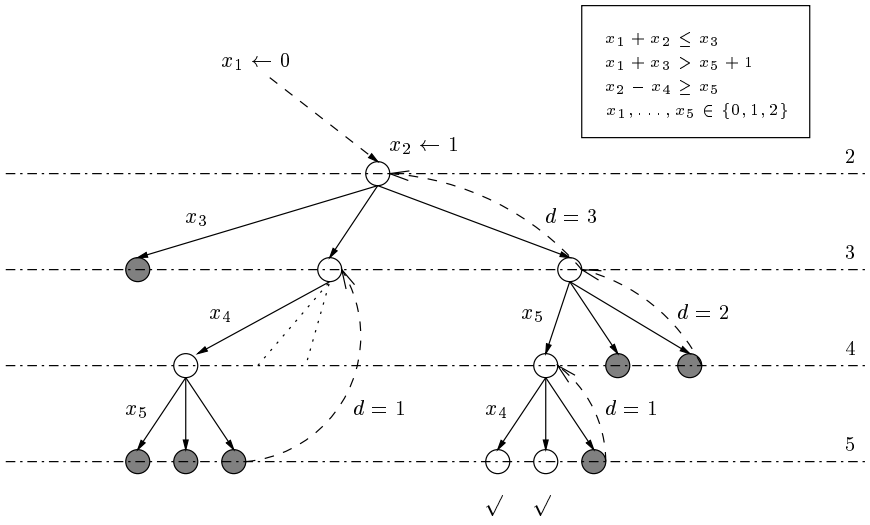


Figure 3.5: An illustration of backjump levels in a CBJ backtrack tree to solve the CSP in Example 2.1.

in the backtrack search. The use of consistency in a backtrack search will reduce the "chances" for backjumping. To analyze the influence of the level of consistency on the backjumping, we need the notion of *backjump level*. Informally, the level of a backjump is the distance, measured in backjumps, from the backjump destination to the "farthest" dead-end [68].

Definition 3.1 (backjump level) *The definition of backjump level is recursive:*
1. A backjump from variable x_i to variable x_h is of level 1 *if it is performed directly from a dead-end state in which all values of x_i fail in the consistency check.*
2. A backjump from variable x_i to variable x_h is of level $d \geq 2$, if all backjumps performed to variable x_i are of level less than d, and at least one of them is of level $d - 1$.

Figure 3.5 shows the backjump levels in the CBJ backtrack tree to solve the CSP in Example 2.1. There is a one-level backjump from x_5 to x_3 because all values in the domain of x_5 fail in consistency checks. Then CBJ finds two solutions for the problem and thus it chronologically backtracks from x_4 to x_5, and later to x_3. The backjumps are of level one and two respectively. At last there is a three-level backjump from x_3 to x_2.

By classifying the backjumps performed by a backjumping algorithm into different levels, we can now weaken CBJ into a series of backjumping algorithms which perform limited levels of backjumps. BJ_k is a backjumping algorithm which is allowed to

perform at most k-level backjumps and it chronologically backtracks when a j-level backjump for $j > k$ is encountered [1]. BJ_n is equivalent to CBJ, which performs unlimited backjumps, and BJ_1 is equivalent to Gaschnig's BJ [53], which only does the first level backjumps.

One may immediately conclude that BJ_{k+1} is always better than BJ_k because it does one more level of backjumps. However, to be more precise, we need to justify that a situation where BJ_k may skip a node visited by BJ_{k+1} does not exist. Similar to the proof of Theorem 9 in Kondrak's work [68], we can show that:

Theorem 3.6 *BJ_k visits all the nodes that BJ_{k+1} visits.*

3.3 Maintaining Strong k-Consistency (MC_k)

Many people in the CSP community have talked about the possibility of applying a higher level of consistency in a backtrack search. However, a backtracking algorithm maintaining strong k-consistency (MC_k) has never been fully addressed in the literature. In order to study the relation between BJ_k and MC_k, we need some background on strong k-consistency and MC_k.

3.3.1 Achieving Strong k-Consistency

Strong k-consistency achievement is a "rough" concept because two algorithms both achieving strong k-consistency may not always compute the same resulting CSP [2]. One reason is that some redundant constraints or universal constraints can be arbitrarily added into and removed from the CSP, without affecting its consistency. For example, given a CSP with three constraints $x_1 \neq x_2$, $x_1 \neq x_3$, and $x_2 \neq x_3$, the constraint, $alldifferent(x_1, x_2, x_3)$ is redundant and it can be added into or removed from the CSP without affecting its consistency.

After a CSP instance is made strongly k-consistent, for any partial solution over less than k variables, $t = \{x_{i_1} \leftarrow a_{i_1}, \ldots, x_{i_j} \leftarrow a_{i_j}\}$ where $j < k$, t is either inconsistent in the resulting CSP, or it can be consistently extended to any $(j+1)^{th}$ variable. The execution of a strong k-consistency achievement algorithm can be viewed as a proving process. That is, an algorithm enforcing strong k-consistency on a CSP instance should detect and remove all those inconsistencies

[1] BJ_k is only of theoretical interest since in practice one would use CBJ rather than artificially prevent backjumping; *i.e.*, one has to actually add code to prevent backjumping.

[2] Also an algorithm enforcing strong $(k+1)$-consistency can be used to achieve strong k-consistency.

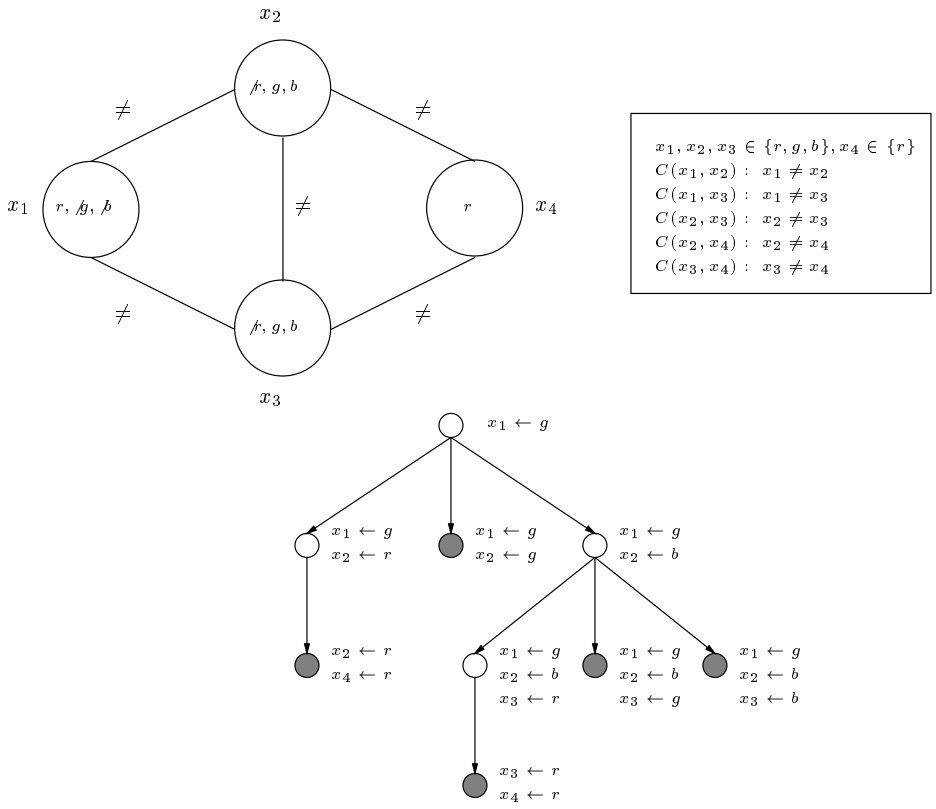


Figure 3.6: A three-proof-tree for $\{x_1 \leftarrow g\}$ in the graphing coloring problem. All leaf nodes are inconsistent in the CSP.

$t = \{x_{i_1} \leftarrow a_{i_1}, \ldots, x_{i_j} \leftarrow a_{i_j}\}$ where $j < k$ and t is consistent but cannot be consistently extended to some $(j+1)^{th}$ variable $x_{i_{j+1}}$. To remove an inconsistency, we make it inconsistent in the resulting CSP by means of removing values from the domains, removing the inconsistent tuples from the existing constraints, and adding new constraints to the CSP. Usually, we are more interested in the domains of the variables after achieving strong k-consistency. We assume that for each variable x, if there is a unary constraint C over x in the resulting CSP, then for each value a in the domain of x, a is removed from the domain if $\{x \leftarrow a\}$ does not satisfy C.

We use the concept of k-proof-tree to characterize the strong k-consistency achievement algorithms.

Definition 3.2 (k-proof-tree) *A k-proof-tree for a partial solution t over no more than k variables in a CSP is a tree in which each node is associated with a partial solution over at most k variables in the CSP, where (1) the root of the k-proof-tree is associated with t, and (2) each leaf node of the k-proof-tree is inconsistent in the*

CSP, and (3) each intermediate node s of the k-proof-tree is consistent in the CSP, and the children of s at the next level are nodes $s' \cup \{x \leftarrow a_1\}, \ldots, s' \cup \{x \leftarrow a_l\}$ such that $s' \subseteq s$, $x \notin vars(s)$, and $dom(x) = \{a_1, \ldots, a_l\}$.

Figure 3.6 shows a three-proof-tree (more than one is possible) for $\{x_1 \leftarrow g\}$ in the given graphing coloring problem. For example, in the above figure, the root of the three-proof-tree is $\{x_1 \leftarrow g\}$ and all the leaf nodes are inconsistent in the original CSP. Because node $\{x_1 \leftarrow g\}$ is consistent in the original CSP, it is not a leaf node. In this three-proof-tree, its children at the next level are $\{x_1 \leftarrow g, x_2 \leftarrow r\}, \{x_1 \leftarrow g, x_2 \leftarrow g\}$ and $\{x_1 \leftarrow g, x_2 \leftarrow b\}$.

After a CSP is made strong k-consistent, if a partial solution t over no more than k variables is inconsistent in the resulting CSP, we can construct a k-proof-tree for t from the execution of the strong k-consistency achievement algorithm. If t is inconsistent in the original CSP, the k-proof-tree contains a single node t. Otherwise, there must exist a point in the execution of the algorithm at which t or a subtuple t' of t failed to be extended to one additional variable x. That is, at this point, all the partial solutions $t' \cup \{x \leftarrow a_1\}, \ldots, t' \cup \{x \leftarrow a_l\}$, where $dom(x) = \{a_1, \ldots, a_l\}$, are inconsistent in the resulting CSP. Then we can construct the k-proof-tree recursively for each of those inconsistencies. On the other hand, given a k-proof-tree for an inconsistency in a CSP, any algorithm achieving strong k-consistency is able to deduce and remove the inconsistency. After applying a strong k-consistency achievement algorithm on the CSP, if all the children of a node in the k-proof-tree are inconsistent in the resulting CSP, that node is also inconsistent in the resulting CSP because one of its subtuples cannot be consistently extended to one additional variable. Because all the leaf nodes in the k-proof-tree are inconsistent in the original CSP and thus in a bottom-up manner, the inconsistency on the root of the tree can be deduced and removed from the resulting CSP.

3.3.2 Induced CSP and Maintaining Strong k-Consistency

A generic scheme to maintain a level of local consistency in a backtrack search is to perform at each node in the search tree, one full cycle of consistency achievement. A consistency achievement algorithm is applied to the problem instantiated with the current partial solution. This problem is called an *induced CSP* of the original CSP. If, as a result, the induced CSP becomes empty after applying the consistency algorithm, the instantiation of the current variable will lead to a dead-end and it should be excluded. If the resulting CSP is not empty, the instantiation of the current variable

is accepted and the search continues to the next level.

The simplest form of an induced CSP, as used in GAC, is to restrict the domains of the instantiated variables to have only one value and leave the set of constraints unchanged. This idea can be traced back to Gaschnig's implementation of MAC, referred to as DEEB [54], *i.e.*, Domain Element Elimination with Backtracking. In DEEB, when a variable x is instantiated with a value a, the domain of the current variable is set momentarily to a single value, *i.e.*, $dom(x) \leftarrow \{a\}$, and the uninstantiated variables are then made arc consistent. An identical approach was taken by Burke when designing the constraint maintenance system for the Distributed Asynchronous Scheduler. A scheduling decision is viewed as the addition of a unary constraint [23].

Definition 3.3 (induced CSP) *Given a partial solution t of a CSP P, the CSP induced by t, denoted by $P|_t$, is exactly the same as the original CSP except that the domain of each variable $x \in vars(t)$ contains only one value $t[x]$, which has been assigned to x by t.*

For example, GAC at each node of the search tree performs generalized arc consistency achievement on the CSP induced by the current partial solution [3]. GAC continues to extend the current partial solution if none of the future domains becomes empty after achieving arc consistency on the induced CSP.

However, MC_k cannot be simply defined as applying strong k-consistency achievement on the induced CSP at each node in the backtrack search. Such an implementation is problematic. For example, if a CSP contains only $(k+1)$-ary constraints, its induced CSPs are always strong k-consistent because no constraint can be checked for a tuple with no more than k variables. Intuitively, the arity of a constraint should be lowered if some of its variables have been instantiated. That is, the subproblem used to achieve strong k-consistency should include the selections and projections of the constraints with respect to the current partial solution. For example, if there is a constraint $C(x_1, x_2, x_3)$ and x_1 has been instantiated in the current partial solution t, the constraint $\pi_{\{x_2,x_3\}}\sigma_{\{x_1 \leftarrow t[x_1]\}}C(x_1, x_2, x_3)$ should be included in the induced CSP. In order to establish a relation between BJ_k and MC_k, we need a more restricted definition of the induced CSP, called *s-induced CSP*, where "s" denotes selections of the constraints.

[3] In our implementation of GAC, the arc consistency achievement algorithm is applied to a more restricted problem than the induced CSP, in which the domains of future variables are also pruned according to the result of the arc consistency achievement at an earlier stage. Nevertheless, these domain prunings would be redone in the case that the arc consistency algorithm was applied to the induced CSP.

Definition 3.4 (s-induced CSP) *Given a partial solution t of a CSP P, the CSP s-induced by t, denoted by $P|_t^s$, is constructed as the following: $P|_t^s$ has all the variables in P except those having been instantiated by t. The domains of the variables are the same as those in P. For each of the constraints C in P where $vars(C) \not\subseteq vars(t)$, a new constraint C' is added to $P|_t^s$, where $vars(C') = vars(C) - vars(t)$ and $rel(C') = \{s[vars(C) - vars(t)] \mid s \in rel(C)$ and $s[vars(C) \cap vars(t)] = t[vars(C) \cap vars(t)]\}$.*

Example 3.3 *Consider the graph coloring problem in Figure 3.6. The original CSP has 4 variables, $x_1, \ldots, x_4$, where $x_1, x_2, x_3 \in \{r, g, b\}$ and $x_4 \in \{r\}$. There are 5 binary constraints, $x_1 \neq x_2$, $x_1 \neq x_3$, $x_2 \neq x_3$, $x_2 \neq x_4$ and $x_3 \neq x_4$. Given a partial solution, $t = \{x_1 \leftarrow g\}$, the induced subproblem $P|_t$ has 4 variables, $x_1, \ldots, x_4$, where the domains of $x_1, \ldots, x_4$ are $\{g\}$, $\{r, g, b\}$, $\{r, g, b\}$ and $\{r\}$, respectively. The constraints in $P|_t$ are the same as those in the original CSP. The CSP s-induced by t, $P|_t^s$, has 3 variables, x_2, x_3 and x_4. The constraints in $P|_t^s$ are, $C'(x_2) = \{(r), (b)\}$, $C'(x_3) = \{(r), (b)\}$, $x_2 \neq x_3$, $x_2 \neq x_4$ and $x_3 \neq x_4$.*

We may notice the difference between the induced CSP and the s-induced CSP. The induced CSP has all the variables and the constraints in the original CSP, but restricts the domains of the instantiated variable (in the partial solution) to have only one value. The s-induced CSP has only the uninstantiated variables (with respect to the partial solution) in the original CSP. The constraints in the s-induced CSP are the selections (and projections) of the constraints in the original CSP.

The maintaining strong k-consistency algorithm (MC_k) at each node in the backtrack search tree applies a strong k-consistency achievement algorithm to the CSP s-induced by the current partial solution. Under such an architecture, FC can be viewed as maintaining one-consistency, and for binary CSPs, MAC can be viewed as maintaining strong two-consistency [4].

The following lemmas (Lemma 3.7 to Lemma 3.12) reveals some basic properties about the induced (s-induced) CSPs and the strong k-consistency enforcement on the induced (s-induced) CSPs, which will be used in the proofs of Theorem 3.14 and Theorem 3.18.

Lemma 3.7 *Given a CSP P and two partial solutions t and t' of P, if $t \subset t'$, then $P|_{t'} = (P|_t)|_{t'-t}$ and $P|_{t'}^s = (P|_t^s)|_{t'-t}^s$.*

Proof: It is easy to verify that $P|_{t'} = (P|_t)|_{t'-t}$. Note that $P|_{t'}^s$ and $(P|_t^s)|_{t'-t}^s$ have the same set of variables and the same set of domains. For each constraint C in P,

[4] However, for general CSPs, arc consistency is not equivalent to strong two-consistency.

because $\pi_{vars(C)-vars(t')}\sigma_{t'}C = \pi_{vars(C)-vars(t')}\sigma_{t'-t}(\pi_{vars(C)-vars(t)}\sigma_t C)$, the constraint C makes the same selection and projection in $P|^s_{t'}$ and $(P|^s_t)|^s_{t'-t}$. Therefore, $P|^s_{t'}$ and $(P|^s_t)|^s_{t'-t}$ have the same set of constraints. That is, $P|^s_{t'} = (P|^s_t)|^s_{t'-t}$. ∎

Lemma 3.8 *Given a CSP P, if P is not empty after achieving strong k-consistency, it is not empty either after achieving strong j-consistency, for $j \leq k$.*

Proof: Suppose P is empty after enforcing strong j-consistency. Thus there is a j-proof-tree for the empty inconsistency in P. Because a j-proof-tree is also a k-proof-tree for $j \leq k$, P is empty after achieving strong k-consistency. That is a contradiction. ∎

Intuitively, the s-induced CSP is more restrictive than the induced CSP. Given a CSP P and a consistent partial solution t, if $P|_t$ is empty after enforcing strong k-consistency, there is a k-proof-tree for the empty inconsistency in $P|_t$. We can convert the k-proof-tree of $P|_t$ into a k-proof-tree for the empty inconsistency in $P|^s_t$. The transformation is done in two steps:
(1) Each node t' in the original k-proof-tree is replaced by $t'[vars(t') - vars(t)]$. Note that $t'[vars(t') - vars(t)]$ is a valid partial solution in $P|^s_t$ (because $P|^s_t$ does not have the variables in $vars(t)$). Furthermore, if t' is not a leaf node in the original k-proof-tree, *i.e.*, t' is consistent in $P|_t$, it is easy to verify that $t'[vars(t') - vars(t)]$ is consistent in $P|^s_t$, *i.e.*, $t'[vars(t') - vars(t)]$ is a valid intermediate node in the k-proof-tree (from the definition of k-proof-tree, an intermediate node in a k-proof-tree must be consistent in the CSP). If t' is a leaf node in the original k-proof-tree, then $vars(t') - vars(t) \neq \emptyset$. Otherwise, we have $t' \subseteq t$. Because t is consistent in P and thus t is consistent in $P|_t$ (note that $P|_t$ has exactly the same set of constraints as P and t is a valid partial solution in $P|_t$), thus t' is consistent in $P|_t$. That is a contradiction. Because t' is inconsistent in $P|_t$, there is a constraint C in $P|_t$ such that t' does not satisfy C, and thus $t'[vars(t') - vars(t)]$ does not satisfy the selection of C in $P|^s_t$. Therefore $t'[vars(t') - vars(t)]$ is inconsistent in $P|^s_t$, *i.e.*, $t'[vars(t') - vars(t)]$ is a valid leaf node in the k-proof-tree.
(2) For each node t' in the original k-proof-tree, if a subtuple $t'' \subseteq t'$ is used to be extended to a variable x instantiated in t, because the domain of x has only one value $t[x]$ in $P|_t$, t' has only one descendant $t'' \cup \{x \leftarrow t[x]\}$ at the next level. After t' is replaced by $t'[vars(t') - vars(t)]$ and $t'' \cup \{x \leftarrow t[x]\}$ is replaced by $t''[vars(t'') - vars(t)]$, we notice that $t''[vars(t'') - vars(t)]$ is subsumed in $t'[vars(t') - vars(t)]$. We further drop the node $t''[vars(t'') - vars(t)]$ and make all its descendants to be

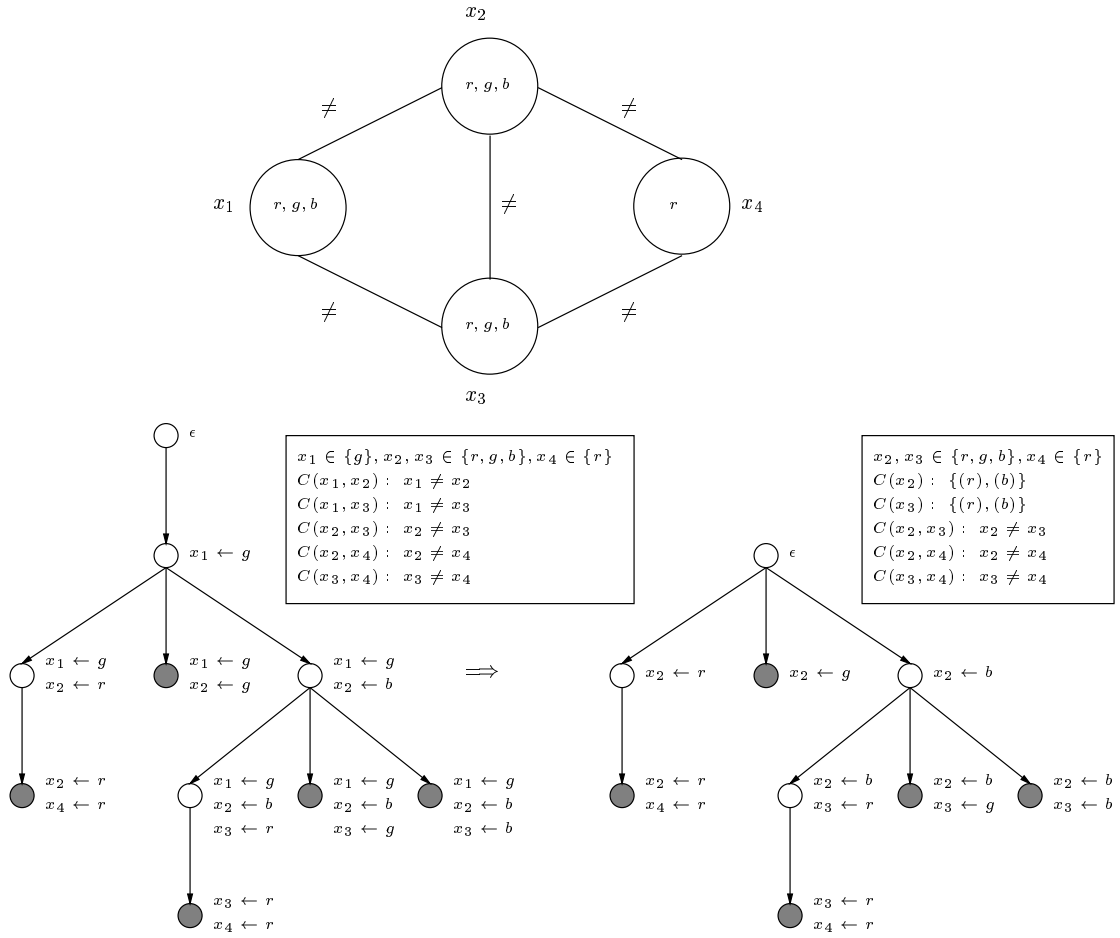


Figure 3.7: In the above graph coloring example, given a partial solution $t = \{x_1 \leftarrow g\}$, there is a three-proof-tree for the empty inconsistency in the induced problem. Furthermore, this three-proof-tree can be converted into a three-proof-tree for the empty inconsistency in the s-induced problem.

the descendants of $t'[vars(t') - vars(t)]$. This operation is necessary because x is not a variable in $P|_t^s$ and from the definition of k-proof-tree, a node in a k-proof-tree can only be extended to a variable of the CSP.

Example 3.4 *Consider the graph coloring example again. Given a partial solution $t = \{x_1 \leftarrow g\}$, we can construct a three-proof-tree for the empty inconsistency in the induced subproblem. Furthermore, the three-proof-tree can be converted into a three-proof-tree for the empty inconsistency in the s-induced subproblem, as shown in Figure 3.7. For example, in the above figure, the node $\{x_1 \leftarrow g, x_2 \leftarrow g\}$ in the original three-proof-tree is replaced by the node $\{x_2 \leftarrow g\}$, while $\{x_1 \leftarrow g, x_2 \leftarrow g\}$ is inconsistent in the induced subproblem and $\{x_2 \leftarrow g\}$ is inconsistent in the s-induced subproblem. As we can see, the root of the original three-proof-tree is extended to variable x_1, which is instantiated in t. The root has only one descendant $\{x_1 \leftarrow g\}$ at the level of x_1. In the three-proof-tree for the s-induced subproblem, the above two nodes are merged into one node ϵ.*

After the above operations, we have made a k-proof-tree for the empty inconsistency in $P|_t^s$. Hence, we have the following result.

Lemma 3.9 *Given a CSP P and a consistent partial solution t, if $P|_t^s$ is not empty after achieving strong k-consistency, $P|_t$ is not empty either after achieving strong k-consistency.*

Lemma 3.10 *Given a CSP P and a partial solution t of P, if $P|_t$ is not empty after achieving strong k-consistency, P is not empty either after achieving strong k-consistency. Furthermore, for each variable $x \in vars(t)$, value $t[x]$ will not be removed from the domain of x when achieving strong k-consistency on P.*

Proof: Suppose P is empty after achieving strong k-consistency. Thus there is a k-proof-tree for the empty inconsistency in P. By removing all the nodes (and their descendents) in the k-proof-tree that are invalid in $P|_t$, *i.e.*, the tuple has instantiated a variable with a value not in its domain in $P|_t$, we can construct a k-proof-tree for the empty inconsistency in $P|_t$. Therefore, $P|_t$ is empty after achieving strong k-consistency. That leads to a contradiction. For each variable $x \in vars(t)$, suppose value $t[x]$ is removed from the the domain of x when achieving strong k-consistency on the original CSP, there is a k-proof-tree for $\{x \leftarrow t[x]\}$ in P. Similarly, we can construct a k-proof-tree for $\{x \leftarrow t[x]\}$ in $P|_t$ by removing all the invalid nodes of the original k-proof-tree. Therefore, the only value $t[x]$ in the domain of x in the $P|_t$

must be removed when achieving strong k-consistency and thus $P|_t$ is empty after achieving strong k-consistency. That is a contradiction. ∎

Corollary 3.11 *Given a CSP P and a consistent partial solution t of P, if $P|^s_t$ is not empty after achieving strong k-consistency, P is not empty after achieving strong k-consistency. Furthermore, for each variable $x \in vars(t)$, value $t[x]$ will not be removed from the domain of x when achieving strong k-consistency on P.*

Proof: It is straightforward from Lemma 3.9 and Lemma 3.10. ∎

Lemma 3.12 *Given a CSP P, if P is not empty after achieving strong k-consistency and a value $a \in dom(x)$ is not removed from the domain of variable x in the resulting CSP, the s-induced CSP $P|^s_{\{x \leftarrow a\}}$ is not empty after achieving strong (k-1)-consistency.*

Proof: Suppose $P|^s_{\{x \leftarrow a\}}$ is empty after achieving strong (k-1)-consistency. Thus there is a (k-1)-proof-tree for the empty inconsistency in $P|^s_{\{x \leftarrow a\}}$. We now convert the ($k$-1)-proof-tree to a k-proof-tree for $\{x \leftarrow a\}$ in P. For each node t in the original (k-1)-proof-tree, t is replaced by $t \cup \{x \leftarrow a\}$. Thus the root of the tree becomes $\{x \leftarrow a\}$. Furthermore, if t is not a leaf node in the original (k-1)-proof-tree; *i.e.*, t is consistent in $P|^s_{\{x \leftarrow a\}}$, it is easy to verify that $t \cup \{x \leftarrow a\}$ is consistent in P. If t is a leaf node in the original (k-1)-proof-tree; *i.e.*, t is inconsistent in $P|^s_{\{x \leftarrow a\}}$, there is a constraint C' in $P|^s_{\{x \leftarrow a\}}$ such that t does not satisfy C'. Let C' be the selection and projection of the constraint C in P. Thus, t does not satisfy the constraint C in P. Therefore, $t \cup \{x \leftarrow a\}$ is inconsistent in P. Hence, we have constructed a k-proof-tree for $\{x \leftarrow a\}$ in P and thus a will be removed from the domain of x when achieving strong k-consistency on P. That is a contradiction. ∎

MC$_k$ will extend the current node if the s-induced CSP by the current partial solution is not empty after achieving strong k-consistency. The node is thus called a *k-consistent node*.

Definition 3.5 (k-consistent node) *A node t in the search tree is a* k-consistent node *if the CSP s-induced by t is not empty after enforcing strong k-consistency.*

Lemma 3.13 *If node t is k-consistent, its ancestors are also k-consistent.*

Proof: Let t' be one of t's ancestors. Because $t' \subset t$, from Lemma 3.7, $P|^s_t = (P|^s_{t'})|^s_{t-t'}$. Thus $P|^s_t$ is an s-induced subproblem of $P|^s_{t'}$. From Corollary 3.11, if $P|^s_t$ is not empty after achieving strong k-consistency, $P|^s_{t'}$ is not empty either after achieving strong k-consistency. Thus t' is k-consistent. ∎

The following theorem applies to the case of finding all solutions.

Theorem 3.14 *MC_k visits a node only if its parent is k-consistent. MC_k visits a node if it is k-consistent.*

Proof: The first part is true because MC_k will not branch on this node if its parent was found not strong k-consistent. We prove the second part by induction on the depth of the search tree. The hypothesis is trivial for $j = 1$. Suppose it is true for j and we have a k-consistent node t at level $j+1$. Let the current variable be x. From Lemma 3.13, t's parent t' at level j is k-consistent. Thus MC_k will visit t'. From Lemma 3.7, $P|^s_t = (P|^s_{t'})|^s_{\{x \leftarrow t[x]\}}$. Because $(P|^s_{t'})|^s_{\{x \leftarrow t[x]\}}$ is not empty after achieving strong k-consistency, from Corollary 3.11, value $t[x]$ will not be removed from the domain of x when achieving strong k-consistency in $P|^s_{t'}$. As a consequence, MC_k will visit t. ∎

A sufficient and necessary condition for MC_k to visit a node t is: t's parent is k-consistent and the value assigned to current variable by t has not been removed from its domain when enforcing strong k-consistency on t's parent.

Theorem 3.15 *Given a CSP instance and a variable ordering strategy, MC_k visits all the nodes that MC_{k+1} visits.*

Proof: It is true from Theorem 3.14 and Lemma 3.10. ∎

3.4 Backjumping Interleaved with Consistency Enforcement

In this section, we first study the relation between MC_k and BJ_k. Kondrak and van Beek [69] have shown that for binary CSPs, BJ (BJ_1) visits all the nodes that FC (MC_1) visits. We can extend their result to the case of general CSPs.

Lemma 3.16 *If CBJ performs a one-level backjump from a higher level variable x_i to a low level variable x_h, the node t_h at the level of x_h is not one-consistent.*

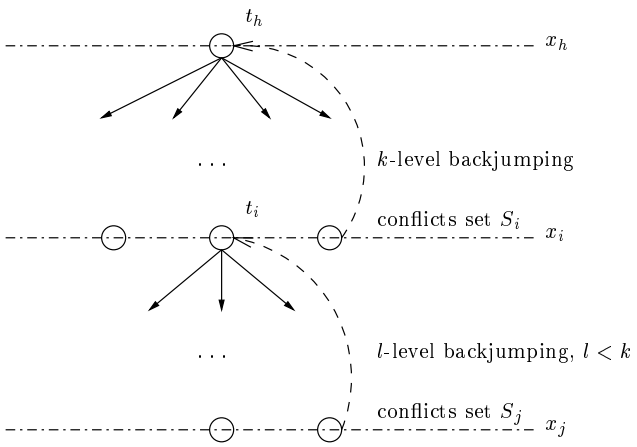


Figure 3.8: A scenario in the CBJ backtrack search tree used in the proof of Lemma 3.17.

Proof: Let S_i be the conflicts set of x_i used in the backjumping in which x_h is the highest level variable. We will show that x_i will experience a domain wipe out when enforcing one-consistency on the s-induced CSP $P|^s_{t_h[S_i]}$. Each node t_i at the level of x_i is a leaf node; *i.e.*, t_i is inconsistent in P. Suppose t_i does not satisfy constraint C where $x_i \in vars(C)$ and $vars(C) \subseteq S_i \cup \{x_i\}$. The selection of C in $P|^s_{t_h[S_i]}$, which constrains only one variable $\{x_i\}$, should prohibit value $t_i[x_i]$ of x_i. Thus x_i will experience a domain wipe out when enforcing one-consistency on $P|^s_{t_h[S_i]}$. Note that $P|^s_{t_h}$ is an s-induced subproblem of $P|^s_{t_h[S_i]}$. From Corollary 3.11, $P|^s_{t_h}$ is empty after enforcing one-consistency. Thus t_h at the level of x_h is not one-consistent. ∎

Lemma 3.17 *If CBJ performs a k-level backjump from a higher level variable x_i to a low level variable x_h, the current node t_h at the level of x_h is not k-consistent.*

Proof: Let S_i be the current conflicts set of x_i in which x_h is the highest level variable. We will show that if there is a k-level backjump from x_i to x_h, then $P|^s_{t_h[S_i]}$ is empty after enforcing strong k-consistency and thus t_h is not k-consistent. We perform an induction on k in the above statement. $k = 1$ is true from Lemma 3.16. Suppose the hypothesis is true for the case of $k-1$ but it is not true for the case of k. That is, there is a k-level backjump from x_i to x_h, but the s-induced CSP $P|^s_{t_h[S_i]}$ is not empty after enforcing strong k-consistency. So there is at least one value a left in the domain of x_i after enforcing strong k-consistency on $P|^s_{t_h[S_i]}$. We know that the node t_i at the level of x_i instantiating x_i with a is either incompatible with t_h; *i.e.*, it is a leaf node, or l-level backjumped from some higher level variable x_j, for some

$1 \leq l < k$. t_i cannot be a leaf node otherwise a will be removed from the domain of x_i when enforcing strong k-consistency. Let S_j be the conflict set of x_j. From the hypothesis, the s-induced CSP $P|^s_{t_i[S_j]}$ is empty after achieving strong l-consistency. Because the s-induced CSP $P|^s_{t_h[S_i]}$ is not empty after achieving strong k-consistency and value a is not removed from the resulting CSP, from Lemma 3.12, the s-induced CSP $P|^s_{t_h[S_i]\cup\{x\leftarrow a\}}$ is not empty after achieving strong (k-1)-consistency. Because $t_i[S_j] \subseteq t_h[S_i] \cup \{x \leftarrow a\}$, the s-induced CSP $P|^s_{t_i[S_j]}$ is not empty after achieving strong (k-1)-consistency. That leads to a contradiction. Thus $P|^s_{t_h[S_i]}$ is empty after achieving strong k-consistency and t_h at the level of x_h is not k-consistent. ∎

Theorem 3.18 *Given a CSP instance and a variable ordering strategy, BJ_k visits all the nodes that MC_k visits.*

Proof: We prove it by performing induction on the level of the search tree. If MC_k visits a node at level j in the search tree, BJ_k will visit the same node. $j = 1$ is trivial. Suppose that it is true for the case of j and we have a node t visited by MC_k at level $j + 1$. We know both MC_k and BJ_k will visit t's parent at level j. The only chance that t may be skipped by BJ_k is that BJ_k backjumps from some higher level variable x_i at level i to a low level variable x_h at level h, such that $h < j + 1 < i$. Thus the node at level h is not k-consistent. Since the node at level h is an ancestor of t and we know t's parent is k-consistent from Lemma 3.13, the node at level h is k-consistent. That is the contradiction. Therefore, BJ_k will visit t at level $k+1$. ∎

We have proved that MC_k always visits no more nodes than BJ_k. When $k = n$, because MC_n, *i.e.*, maintaining strong n-consistency in the search, can solve the problem without backtracking [39], it always visits fewer nodes than CBJ. Thus the more we check forward, the less we jump backward. Certainly, in most cases, MC_k will visit dramatically fewer nodes than CBJ does, but there are instances such that it is exponentially worse than BJ_{k+1}. For instance, in Example 3.1, BJ_{k+1} can be exponentially better than MC_k.

Presumably, MC_k may be combined with backjumping, namely MC_k-CBJ, provided the conflicts sets are computed correctly after achieving strong k-consistency on the s-induced CSPs.

Theorem 3.19 *Given a CSP instance and a variable ordering strategy, MC_k visits all the nodes that MC_k-CBJ visits.*

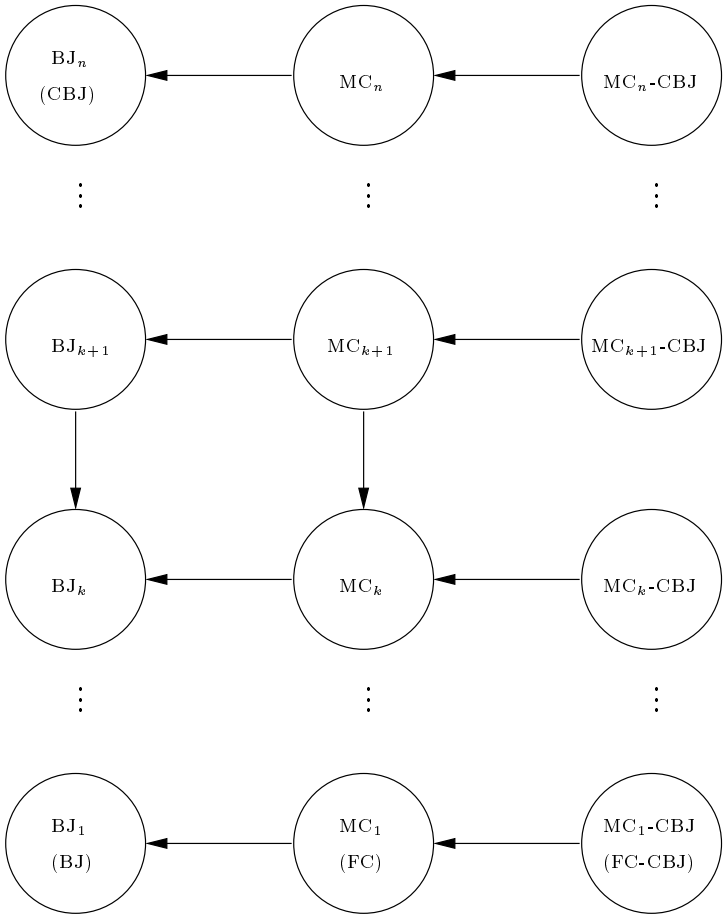


Figure 3.9: A hierarchy for BJ$_k$, MC$_k$, and their hybrids in terms of the size of the backtrack search tree.

Proof: Because MC$_k$-CBJ behaves exactly the same as MC$_k$ in the forward phase of a backtrack search, it is easy to verify that MC$_k$-CBJ visits a node t only if t's parent is k-consistent and the value assigned to the current variable by t was not removed from its domain when achieving strong k-consistency on t's parent. Therefore, MC$_k$-CBJ always visits no more nodes than MC$_k$ does. ∎

Consider Example 3.1 again. At each level of the backtrack tree, the instantiation of each of the past variables removes one distinct value from the domain of the current variable (because of the binary difference constraints), thus the conflicts set of the current variable should include all the past variables. Therefore, there are no chances for MC$_k$-CBJ to backjump in the specially constructed CSP so that MC$_k$-CBJ and MC$_k$ visit exactly the same nodes. Consequently, BJ$_{k+1}$ can be exponentially better than MC$_k$-CBJ. Furthermore, because MC$_{k-1}$-CBJ can reach the second pigeon-hole

problem without encountering a dead-end, it can finally retreat from the second pigeon-hole problem to the root of the backtrack search tree by backjumps. Thus, MC_{k-1}-CBJ may be exponentially better than MC_k-CBJ.

In Figure 3.9, we present a hierarchy in term of the size of the backtrack search tree, for BJ_k, MC_k and MC_k-CBJ. If there is a path from algorithm $\mathcal{A}$ to algorithm $\mathcal{B}$ in the figure, we know that $\mathcal{A}$ always visits no more nodes than $\mathcal{B}$ does. Otherwise, there are instances to show $\mathcal{A}$ may be exponentially better than $\mathcal{B}$, and *vice versa*.

Although the benefits from backjumping are offset by the efforts of look-ahead, FC-CBJ is a good trade-off between FC and CBJ and it may improve FC and CBJ by orders of magnitude. Could the combination of CBJ with an algorithm maintaining a stronger consistency still provide improvement? In the following, we will discuss the combination of CBJ with MAC or GAC, which enforces a stronger consistency than FC does.

3.5 Generalized Maintaining Arc Consistency with Conflict-directed Backjumping (GAC-CBJ)

Maintaining arc consistency with conflict-directed backjumping for binary CSPs, called MAC-CBJ, was proposed by Prosser [93]. Prosser's implementation of MAC-CBJ is based on the AC3 algorithm for binary CSPs [75]. We present a generalized version of MAC-CBJ, called GAC-CBJ.

3.5.1 Implementation

The pseudo code of GAC-CBJ is shown in Figure 3.10. GAC-CBJ can be viewed as an integration of GAC in Figure 2.6 and FC-CBJ in Figure 2.7 with careful handling of conflicts information in constraint propagation. Whenever value a of a future variable y fails to find a valid support in a constraint and thus is removed from its domain, GAC-CBJ needs to compute a no-good accountable for such a removal. To ensure completeness, Prosser suggests that the conflicts sets be propagated along with constraint propagation. The failure of $y \leftarrow a$ to find a valid support in a constraint C may be due to: (1) an instantiation of a past variable does not support $y \leftarrow a$ in C and thus the past variable should be added to the current conflicts set of variable y, (2) a value of an uninstantiated variable x, which could be used to form a support for a in C, is removed from its domain. Thus the current conflicts set of x, which is accountable for current removings in the domain of x, should be merged in the

```
procedure restore(in current: integer);
1  x ← order[current];
2  for i ← current + 1 to n do
3     y ← order[i];
4     if checking[x][y] = current then
5        for each a ∈ dom(y) and domain[y][a] = current do domain[y][a] ← 0;
6        for j ← 1 to current do
7           v ← order[j]; if checking[v][y] = current then checking[v][y] = 0;

procedure record_checking( in C: constraint; in y: variable; in current: integer);
1  for each v ∈ vars(C) and v ≠ y do
2     if instantiated[v] then
3        if checking[v][y] = 0 then checking[v][y] ← current;
4     else
5        for i ← 1 to current do
6           x ← order[i];
7           if checking[x][v] ≠ 0 and checking[x][y] = 0 then checking[x][y] ← current;

function check_forward(  in C: constraint; in current: integer;
                         out fail: variable) : boolean;
1  y ← the uninstantiated variable in the scheme of constraint C; changed ← false;
2  for each a ∈ dom(y) and domains[y][a] = 0 do
3     solution[y] ← a;
4     if not check_constraint( C, solution ) then
5        changed ← true; domains[y][a] ← current;
6        domain_count[y] ← domain_count[y] − 1;
7  if changed then
8     record_checking( C, y, current ); push( y, S ); fail ← y;
9  if domain_count[y] = 0 then return false else return true;

function revise(  in C: constraint; in v: variable;
                  in current : integer; out fail : variable) : boolean;
1  changed ← false;
2  for each a ∈ dom(v) and domains[v][a] = 0 do
3     solution[v] ← a;
4     if not  exists( C, v ) then
5        changed ← true; domains[v][a] ← current;
6        domain_count[v] ← domain_count[v] − 1;
7  if changed then
8     record_checking( C, v, current ); push( v, S ); fail ← v;
9  if domain_count[v] = 0 then return false else return true;
```

Figure 3.10: GAC-CBJ.

```
function consistent( in current : integer; out fail : variable) : boolean;
1  S ← ∅;
2  x ← order[current];
3  for each C ∈ 𝒞 and x ∈ vars(C) do
4     if count_uninst[C] = 1 then
5        if check_forward( C, current, fail ) then return false;
6  push( x, S );
7  while S ≠ ∅ do
8     y ← top( S ); pop( S );
9     for each C ∈ 𝒞 and y ∈ vars(C) do
10       if count_uninst[C] ≥ 2 then
11          for each v ∈ vars(C) and not instantiated[v] and v ≠ y do
12             if not  revise( C, v, current, fail ) then return false;
13 return true;

function  GAC_CBJ( in current : integer ): boolean;
1  if current > n then return true;
2  x ← get_next_var( current ); order[current] = x;
3  update_constraint_counts( x );
4  for each a ∈ dom(x) and domains[x][a] = 0 do
5     solution[x] ← a;
6     instantiated[x] ← true;
7     if consistent( current, fail ) then
8        j ← GAC_CBJ( current + 1 );
9        if j ≠ current then
10          restore( current );
11          instantiated[x] ← false;
12          restore_constraint_counts( x );
13          return j;
14       else
15          cs[x] ← cs[x] ∪ { y | instantiated[y] and checking[y][fail] ≠ 0};
16    restore( current );
17    instantiated[x] ← false;
18 cs[x] ← cs[x] ∪ { y | instantiated[y] and checking[y][x] ≠ 0};
19 j ← max{ i  | 1 ≤ i ≤ current and order[i] ∈ cs[x] };
20 cs[order[j]] ← (cs[order[j]] ∪ cs[x]) − {order[j]};
21 for i ← j + 1 to current do cs[order[i]] ← ∅;
22 restore_constraint_counts( x );
23 return j;
```

Figure 3.10: GAC-CBJ.

current conflicts set of y. Procedure record_checking in GAC-CBJ records all the above conflicts information in the *checking* data structure for y. Of course, the conflicts set built in such a way is far from minimal. However, to compute a smaller conflicts set in constraint propagation is not straightforward.

Bessière and Régin show in experiments on random binary CSPs that MAC-CBJ does not provide noticeable improvement over MAC [16]. Moreover, the run time performance of MAC-CBJ is usually worse than the performance of MAC. Thus they conclude that CBJ becomes useless to MAC except for some sparse CSPs. What may affect the performance of the backjumping? We know that a "good" dynamic variable ordering may degrade the improvement of the backjumps. Besides the influence of the heuristics, in random CSPs, the conflicts sets used to backjump in MAC-CBJ are more likely to be saturated with all the past variables because the conflicts sets are propagated along with constraint propagation, so that MAC-CBJ most of the time performs a chronological backtracking. Grant and Smith [59] have studied the phase transition behavior of MAC and MAC-CBJ on several classes of random binary problems. They observe in experiments that the behavior of MAC and MAC-CBJ are very similar at the median and higher percentile levels apart from the maximum, for all random problem classes. This suggests that CBJ's biggest effect be on the most difficult problems, and that its performance be otherwise similar to chronological backtracking when a dynamic variable ordering is used. They observe that MAC-CBJ does significantly reduce the difficulty of the exceptionally hard problems (*ehps*) [103] that MAC finds in the populations of the random problems in the experiments. A sparse random CSP is more likely to be an instance of *ehps* and it has been observed that CBJ is a useful technique to decrease the abnormal behaviors in *ehps* [10].

3.5.2 Empirical Evaluations

In the following, we evaluate GAC-CBJ over several domains of problems, besides the random CSPs. Bessière and Règin's conclusion for MAC-CBJ is based on empirical evaluations on random binary CSPs. However, the random problems used in their experiments are out-dated. For example, they compare the performance of MAC-CBJ and MAC on the problems used by Frost and Dechter in 1994 [49], while these problems can be solved in less than 0.01 seconds in a 400 MHz Pentinum II computer. Thus, such a comparison is less meaningful given today's computational power. Note that MAC was once evaluated to be worse than FC-CBJ on some instances which were hard in the past and can be easily solved today. So, could it be possible that

GAC-CBJ can produce noticeable improvements over GAC on harder problems? In this section, we compare the performance of GAC and GAC-CBJ on two sets of "real" hard non-binary random problems. Our experimental results show that GAC-CBJ is usually 10% slower than GAC on the dense problems, but GAC-CBJ can provide noticeable improvements over GAC on the sparse problems.

Furthermore, real world problems are very different to random binary CSPs. Real world problems are often more naturally represented as non-binary CSPs. Their CSP formulations tend to be structured; *i.e.*, some variables are more likely to be constrained with each other, and the constraints rendered in the CSP are not randomly generated either. These problems are hard to solve because of the extremely huge search space to be explored. For example, in a CSP formulation for a logistics problem, there are hundreds of variables, and each variable may take tens of values. On the other hand, there are few constraints compared to the number of variables. There could be many solutions to the logistics problem because of the symmetries and parallelism among the planning actions. Thus the CSP is not in the "phase transition region" of the random CSPs. The question is: how can we quickly find one solution? So it is critical for any backtrack algorithm to efficiently prune the search space and avoid the *thrashing* searches. Intuitively, there are opportunities for an algorithm that performs more checks and uses a backjumping technique to improve the search. In fact, the improvement of look-back techniques to planning problems has been observed by Bayardo and Schrag [11]. They first model a planning problem as a SAT problem, then solve the SAT problem by the well-known Davis-Putnam [27] algorithm, the SAT version of maintaining arc consistency, along with advanced general heuristics for SAT problems. Most importantly, they use backjumping and learning mechanisms in the backtrack search. These enhancements have been shown to significantly improve the problem solving. So it is still too early to say that CBJ is useless to GAC. One reason that Bessière and Régin did not test MAC-CBJ on real world problems may be due to an implementation of MAC-CBJ for general CSPs was not available. The generalization of MAC-CBJ to GAC-CBJ enables us to test several real world problems, the planning problems and the crossword puzzle problems. Our experimental results lead us to differ with their conclusion for GAC-CBJ. Although GAC-CBJ does not improve GAC on relatively easy instances, the overhead is almost negligible. However, we do observe that GAC-CBJ can provide several orders of magnitude improvement over GAC on some harder instances.

Table 3.1: Time(seconds) to solve 100 instances of $(100, 3, 3, 300, 0.73)$ problems.

	dom+deg		*dom/deg*	
	GAC	GAC-CBJ	GAC	GAC-CBJ
1	25.53	32.04	16.87	21.28
2	6.32	7.87	0.12	0.18
3	0.02	0.02	0.12	0.14
4	10.80	13.63	10.36	12.75
5	6.68	8.45	5.42	6.98
6	18.74	23.67	8.25	10.56
7	1.71	2.31	1.43	1.84
8	0.22	0.31	0.60	0.80
9	0.45	0.62	1.77	2.20
10	15.82	19.78	4.93	6.32
.				
.				
Average	13.55	17.32	7.00	8.90

Random CSPs

Both Bessière and Régin's evaluation of MAC-CBJ, and Grant and Smith's study of the phase transition behavior of MAC-CBJ use binary random CSPs, because MAC-CBJ is only applicable to binary CSPs. As we may expect, GAC-CBJ will not provide much improvement over GAC on non-binary random CSPs. As the constraints become non-binary, the "saturation" problem of the conflicts sets is even worse because the conflicts sets of more uninstantiated variables are propagated along with the constraint propagation.

Table 3.1 and Table 3.2 show the run time performance of GAC and GAC-CBJ on two sets of randomly generated non-binary CSPs. Each set contains 100 random instances. The tables show the run time performance of the algorithms on the first 10 instances from each set and the average on all instances. A set of random problems is defined by a 5-tuple (n, d, r, m, q), where n is the number of the variables, d is the uniform domain size, r is the uniform arity of the constraints, m is the number of randomly generated constraints, and q is the uniform tightness of the constraints. The constraint tightness q is chosen to make about half of the instances in the population to be insoluble, *i.e.*, q is in the phase transition region. Two dynamic orderings are used to solve the problems. One is the popular *dom+deg* heuristic which chooses the next variable with the minimal domain size and breaks ties by choosing the variable with the maximum static degree, *i.e.*, the number of the constraints that constrain

Table 3.2: Time(seconds) to solve 100 instances of $(300, 5, 3, 300, 0.25)$ problems.

	dom+deg		*dom/deg*	
	GAC	GAC-CBJ	GAC	GAC-CBJ
1	87.99	102.10	3.25	4.88
2	437.10	559.40	101.30	144.10
3	546.90	573.70	9.37	13.24
4	392.00	524.50	12.94	19.22
5	**1277.00**	**600.60**	1.91	2.10
6	0.33	0.27	0.11	0.15
7	**11980.50**	**2522.00**	22.89	34.70
8	**27015.32**	**2713.00**	0.73	1.01
9	787.40	577.40	31.64	44.12
10	11.32	11.98	0.76	0.97
.				
.				
Average	2617.59	1112.87	29.67	43.18

that variable. The other is the *dom/deg* heuristic proposed in [16] which chooses the next variable with the minimal value of the domain size divided by its degree. The experiments were run on 400 MHZ Pentinum II's with 256 Megabytes of memory. The problems in $(100, 3, 3, 300, 0.73)$ are relatively dense, in which GAC-CBJ is not expected to perform better than GAC. As we can see from the above tables, GAC-CBJ in general is about 30% slower than GAC in solving the problems, under both dynamic variable orderings. In contrast, the problems in $(300, 5, 3, 300, 0.25)$ are in general very sparse. As Bessière and Régin suggest in [16], GAC-CBJ is more likely to provide improvements over GAC on these problems. For example, under *dom+deg* heuristic, GAC-CBJ ran an order of magnitude faster than GAC to solve instances 7 and 8. Under *dom/deg*, both algorithms can solve the problems very quickly, and GAC-CBJ is generally about 40% slower than GAC.

Although GAC-CBJ could provide remarkable improvements on some sparse CSPs, in our experiments, it is usually 30% to 40% slower than GAC, thus it is questionable whether in practice we should pay the 40% overhead of CBJ enhancement to GAC and hope that it will sometimes produce significant savings. Nevertheless, the first implementation of GAC-CBJ can be improved. In the original version of GAC-CBJ, procedure *record_checking* takes $O(rn)$ steps to record the conflicts information generated in the constraint propagation, and procedure *restore* takes $O(n^2 + nd)$ steps to restore the domains and conflicts information in a backtracking step. As the

```
procedure restore(in current: integer);
1   x ← order[current];
2   for i ← current + 1 to n do
3       y ← order[i];
4       if checking[x][y] = current then
5           for each a ∈ dom(y) and domain[y][a] = current do domain[y][a] ← 0;
6           top ← checking_top[y];
7           while top > 0 and checking[checking_set[y][top]][y] = current do
8               checking[checking_set[y][checking_top[y]]][y] ← 0;
9               checking_top[y] ← checking_top[y] − 1;
10              top ← checking_top[y];

procedure record_checking( in C: constraint; in y: variable; in current: integer);
1   for each v ∈ vars(C) and v ≠ y do
2       if instantiated[v] then
3           if checking[v][y] = 0 then
4               checking[v][y] ← current;
5               checking_top[y] ← checking_top[y] + 1;
6               top ← checking_top[y]; checking_set[y][top] ← v;
7       else
8           for i ← 1 to checking_top[v] do
9               if checking[checking_set[v][i]][y] = 0 then
10                  checking[checking_set[v][i]][y] ← current;
11                  checking_top[y] ← checking_top[y] + 1;
12                  top ← checking_top[y]; checking[y][top] ← checking_set[v][i];
```

Figure 3.11: Improved GAC-CBJ implementation.

above two procedures are called very frequently in the backtrack search, the original implementation will cause noticeable overhead in the overall performance. For example, when we profile the execution of GAC-CBJ to solve the first instance of the $(300, 5, 3, 300, 0.25)$ problems under *dom+deg* heuristic, the profiling result shows that the overhead on *record_checking* and *restore* procedures accounts for about 40% of the total run time. However, it is not necessary that it be so expensive to maintain the conflicts and domains information if the sizes of the conflicts sets found in the backtrack search are much smaller than the number of the variables n. As shown in Figure 3.11, the overhead in the above two procedures can be reduced by using two auxiliary data-structures, *checking_set* and *checking_top*. Remember that the data-structure *checking* implements a table representation of the conflicts information founded in the forward phase of the backtrack search; *i.e.*, $checking[x][y] = 1$ indicates that the instantiation of the past variable x has caused some of the values in the domain of the future variable y to be pruned, while *checking_set* and *checking_top* implement a list representation of the above conflicts information, in which $checking\_top[y]$ denotes the number of the past variables x such that $checking[x][y] = 1$ and $checking\_set[y][1], \ldots,$ and $checking\_set[y][checking\_top[y]]$ record each of these variables. Again, we profile the execution of the new implementation of GAC-CBJ on the same instance mentioned above. The profiling result shows that the overhead on *record_checking* and *restore* has been reduced to 13.6% of the overall run time.

We use the improved implementation of GAC-CBJ to solve the same set of problems in the above. The results are shown in Table 3.3 and Table 3.4. On the dense $(100, 3, 3, 300, 0.73)$ problems, GAC-CBJ is still slower than GAC, but the difference in their performance is reduced to less than 10%. As we can see in Table 3.3, the performance of the improved GAC-CBJ is generally better than the one of the original GAC-CBJ. On the sparse $(300, 5, 3, 300, 0.25)$ problems, the improved GAC-CBJ ran faster than GAC on the first 10 instances under *dom+deg* heuristic, and under *dom/deg* heuristic, the difference between GAC and GAC-CBJ is reduced to less than 10%.

Planning Problems

The constraint programming approach to planning problems is a relatively new but promising field of study. It is not surprising that a constraint planner can do much better in the planning domains, as the SAT methods have been successfully applied to solve some real world planning problems [66, 67]. In the constraint programming methodology we formulate a planning problem as a CSP in terms of variables, do-

Table 3.3: Time(seconds) to solve 100 instances of $(100, 3, 3, 300, 0.73)$ problems.

	dom+deg			*dom/deg*		
	GAC	GAC-CBJ original	GAC-CBJ improved	GAC	GAC-CBJ original	GAC-CBJ improved
1	25.53	32.04	27.62	16.87	21.28	18.58
2	6.32	7.87	6.77	0.12	0.18	0.15
3	0.02	0.02	0.02	0.12	0.14	0.13
4	10.80	13.63	11.88	10.36	12.75	11.18
5	6.68	8.45	7.28	5.42	6.98	6.02
6	18.74	23.67	20.20	8.25	10.56	9.14
7	1.71	2.31	1.96	1.43	1.84	1.59
8	0.22	0.31	0.25	0.60	0.80	0.67
9	0.45	0.62	0.52	1.77	2.20	1.90
10	15.82	19.78	17.07	4.93	6.32	5.51
.						
.						
Average	13.55	17.32	14.92	7.00	8.90	7.73

Table 3.4: Time(seconds) to solve 100 instances of $(300, 5, 3, 300, 0.25)$ problems.

	dom+deg			*dom/deg*		
	GAC	GAC-CBJ original	GAC-CBJ improved	GAC	GAC-CBJ original	GAC-CBJ improved
1	87.99	102.10	70.83	3.25	4.88	3.61
2	437.10	559.40	385.00	101.30	144.10	101.80
3	546.90	573.70	414.10	9.37	13.24	10.12
4	392.00	524.50	357.80	12.94	19.22	13.95
5	**1277.00**	**600.60**	**452.60**	1.91	2.10	1.75
6	0.33	0.27	0.22	0.11	0.15	0.12
7	**11980.50**	**2522.00**	**1799.00**	22.89	34.70	25.22
8	**27015.32**	**2713.00**	**1900.00**	0.73	1.01	0.79
9	787.40	577.40	409.40	31.64	44.12	33.66
10	11.32	11.98	9.24	0.76	0.97	0.80
.						
.						
Average	2617.59	1112.87	823.52	29.67	43.18	32.23

mains, and constraints. The choice of the variables and domains defines the search space and the choice of the constraints defines how the search space can be reduced in a backtrack search. A state space formulation is to model each state by a collection of variables. For example, in the logistics problem, we define the following variables for each state S_t: $package_{i,t}, truck_{j,t}$, and $plane_{k,t}$, where i, j, k range over the number of packages, trucks and planes, respectively and t ranges over the number of steps in the plan. The domain of a package variable includes all possible locations for the package, trucks and planes that may be used to deliver the package. Assigning a package variable a location means the package is at that location in that state and assigning a package variable a truck means the package is in that truck in that state. The basic constraints enforce the assignments of variables to represent a consistent state or a valid transition between states. The essence in the constraint planner is to use domain knowledge, in terms of redundant constraints, to improve the backtrack search. Most of the constraints are non-binary and represented intensionally as functions which return true or false, given a set of assignments to the variables in the scheme of the constraint. The compact representation of constraints is one advantage of the constraint planner to the SAT planner, which needs to convert each tuple in a constraint into clauses and thus demands a large amount of space to store those clauses.

Given a CSP formulation of the planning problem, we then need to determine which algorithm should be used to solve the CSP. Table 3.5 shows the comparison between GAC and GAC-CBJ in solving 35 instances of logistics problems. Each instance was tried to be solved within 20 hours of CPU time. Two heuristics are tested in the experiments, *dom+deg* and *dom/deg*. On about one third of the instances, GAC-CBJ has shown improvement over GAC. The improvement is even significant on the hard instances. For example, on instance 18, 20 and 27, GAC-CBJ ran several orders of magnitude faster than GAC, and on instance 15, GAC ran out the 20 hours limit but GAC-CBJ can find a solution within 3 minutes. GAC-CBJ and GAC perform similarly on easier instances and sometimes GAC-CBJ is about 10% slower than GAC.

The improvement of GAC-CBJ may be partly ascribed to the variable ordering used in the backtrack search. A wrong decision at an early stage in the backtrack search will lead GAC to exhaustively explore an insoluble subproblem. GAC-CBJ has the ability to identify the source of inconsistencies and escape the insoluble subproblem more quickly. However, in our experiments, both heuristics gave similar results. One reason is that adding redundant constraints has dramatically changed

Table 3.5: Time (seconds) to solve logistics planning problems. The absence of an entry indicates that the problem was not solved correctly within the given resource limits.

	dom+deg		*dom/deg*	
	GAC	GAC-CBJ	GAC	GAC-CBJ
1	0.03	0.03	0.03	0.03
2	0.03	0.05	0.03	0.06
3	**10.91**	**0.86**	**9.63**	**0.81**
4	0.16	0.17	0.14	0.18
5	1.51	1.54	1.54	1.57
6	**36.49**	**16.86**	**35.77**	**16.76**
7	0.08	0.08	0.08	0.09
8	0.15	0.15	0.14	0.16
9	0.30	0.33	0.32	0.33
10	.	.	.	.
11	0.04	0.05	0.05	0.05
12	0.11	0.13	0.11	0.11
13	0.54	0.57	0.54	0.56
14	0.63	0.64	0.64	0.68
15	**.**	**182.51**	**.**	**8540.58**
16	**12.49**	**0.42**	**12.32**	**0.41**
17	**264.46**	**0.32**	**261.33**	**0.32**
18	**15382.82**	**1165.54**	**15157.71**	**1184.67**
19	1.29	1.37	1.33	1.31
20	**6268.16**	**27.66**	**6125.87**	**28.55**
21	0.66	0.70	0.68	0.74
22	.	.	.	.
23	.	.	.	.
24	0.08	0.09	0.08	0.09
25	34.03	13.03	11.58	12.10
26	.	.	.	.
27	**12239.26**	**47.06**	**12105.62**	**47.76**
28	.	.	.	.
29	.	.	.	.
30	.	.	.	.

Table 3.6: Time (seconds) to solve blocks planning problems. The absence of an entry indicates that the problem was not solved correctly within the given resource limits.

	dom+deg		*dom/deg*	
	GAC	GAC-CBJ	GAC	GAC-CBJ
1	0.11	0.11	0.12	0.12
2	1.28	1.46	1.17	1.35
3	105.44	86.96	126.98	127.41
4	11712.28	11116.15	21534.81	21199.86

the structure of the CSP formulation. For example, we can increase the degree of a variable by adding more redundant constraints on that variable. Thus the heuristics depending on the degree of a variable may not work properly under our formulation of the problem. Unless there is a more precise and domain dependent heuristic, the general heuristics cannot help GAC to step away from a wrong decision at an early stage. GAC-CBJ is relatively robust to the heuristics as it can rescue a bad decision using backjumpings. This may be an advantage in problems where we do not have much domain knowledge *a priori*.

We ran the same experiments on the blocks world planning problems. The results are shown in Table 3.6. It worth noting that the run time performance of the original implementation of GAC-CBJ on these problems was about twice that of GAC, while the improved GAC-CBJ ran faster than GAC on the hard instances. In this domain, GAC-CBJ does not produce huge savings. A deep analysis shows that there are some large jumps during the execution of GAC-CBJ, but the skipped variables were usually instantiated with the last value in their domains. Thus the savings are not large. One explanation for the difference of GAC-CBJ's behavior between the logistics problems and the blocks world problems is that the variables in the formulations of the blocks world problems have smaller domains, usually 4 values, than those in the logistics problems which may take 30 values. In fact, the logistics problems are very different from the blocks world problems. In a logistics problem, the whole planning task can be easily decomposed into several relatively independent sub-tasks. For example, the packages within different cities are competing for different trucks. These relatively independent sub-tasks provide many chances for CBJ to backjump and produce savings. In the blocks world problem, all the blocks are competing for one robotic arm. Thus it is not obvious that the whole planning task can be decomposed into several smaller sub-tasks.

Table 3.7: Time (seconds) to solve gripper planning problems. The absence of an entry indicates that the problem was not solved correctly within the given resource limits.

	dom+deg		*dom/deg*	
	GAC	GAC-CBJ	GAC	GAC-CBJ
1	0.01	0.01	0.00	0.00
2	0.03	0.03	0.03	0.03
3	0.08	0.07	0.06	0.08
4	0.15	0.16	0.13	0.14
5	0.27	0.29	0.23	0.27
6	0.43	0.48	0.39	0.42
7	0.68	0.76	0.59	0.66
8	1.02	1.13	0.87	0.96
9	1.45	1.62	1.21	1.36
10	2.02	2.25	1.66	1.90
11	2.74	3.06	2.22	2.53
12	3.62	4.04	2.90	3.33
13	4.71	5.26	3.72	4.28
14	6.00	6.70	4.73	5.44
15	7.55	8.43	5.87	6.75
16	9.37	10.47	7.22	8.36
17	11.51	12.87	8.79	10.17
18	13.97	15.65	10.63	12.33
19	16.81	18.86	12.67	14.73
20	20.09	22.52	15.04	17.62

Table 3.8: Time (seconds) to solve grid planning problems. The absence of an entry indicates that the problem was not solved correctly within the given resource limits.

	dom+deg		*dom/deg*	
	GAC	GAC-CBJ	GAC	GAC-CBJ
1	0.66	0.68	1.58	0.86
2	**762.47**	**33.33**	**3965.10**	**321.17**
3	.	.	.	.
4	**.**	**1753.13**	.	.
5	.	.	.	.

Table 3.9: Time (seconds) to solve 5x5 crossword puzzle problems. The absence of an entry indicates that the problem was not solved correctly within the given resource limits.

	dom+deg				*dom/deg*			
	UK		Linux		UK		Linux	
	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ
1	1.40	1.45	1.24	1.21	1.37	1.45	1.21	1.21
2	1.05	1.10	0.28	0.29	1.03	1.05	0.29	0.29
3	0.93	0.91	0.30	0.30	0.88	0.91	0.29	0.29
4	0.85	0.84	0.19	0.19	0.81	0.82	0.20	0.20
5	0.74	0.71	0.17	0.18	0.73	0.73	0.17	0.17
6	0.94	0.95	0.38	0.39	0.95	0.95	0.37	0.39
7	0.94	0.93	0.35	0.36	0.93	0.96	0.34	0.34
8	0.88	0.92	0.31	0.32	0.89	0.89	0.32	0.32
9	0.77	0.78	0.21	0.20	0.80	0.78	0.20	0.19
10	0.70	0.75	0.17	0.17	0.71	0.73	0.18	0.16

We tested the other two planning problems, the gripper problems and the grid problems. The gripper problems are easy to solve due to the use of domain knowledge in the formulation. Generally GAC-CBJ is about 10% slower than GAC, as shown in Table 3.7. The grid problems are much harder. As we can see in Table 3.8, two out of five instances cannot be solved by both algorithms in 20 hours. GAC-CBJ shows improvement on the grid problems. For example, it can solve problem 4 in about half an hour, but GAC failed to find a solution in 20 hours.

Crossword Puzzle Problem

The crossword puzzle problem is different from the planning problems we tested above. An instance of a crossword puzzle problem is shown in Example 4.3. As we know, there are at least 3 practical ways to formulate the problem, to give each letter a variable and set a non-binary constraint for each word in the puzzle to enforce the letters forming a legal word, or to transform the above formulation to the dual representation or hidden representation. In its original formulation, the domain of a variable consists of 26 letters, from a to z. The arity of a constraint reflects the length of the word that the constraint represents. For example, a word of 10 letters will result in a 10-ary constraint over those letter variables. The tuples in the non-binary constraint represent the words that are of the same length as the arity of the constraint in a pre-defined dictionary. The number of tuples ranges from 5000 to 30000 according to the

Table 3.10: Time (seconds) to solve 15x15 crossword puzzle problems. The absence of an entry indicates that the problem was not solved correctly within the given resource limits.

	dom+deg				*dom/deg*			
	UK		Linux		UK		Linux	
	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ
1	18.79	19.13	112.70	114.80	25.71	25.73	12.06	11.72
2	30.30	30.54	345.00	339.20	35.30	35.58	129.10	126.58
3	26.30	27.90	9.50	9.40	29.13	29.59	13.72	13.29
4	16.95	17.26	3261.20	2082.80	18.42	18.74	.	.
5	19.11	19.76	7.50	7.60	31.34	31.42	8.20	8.10
6	41.58	42.27	10021.50	8910.70	74.93	75.10	9963.14	9593.49
7	123.17	124.15	14319.40	13170.80	33.22	33.68	**29051.51**	**2422.66**
8	18.55	18.77	9.10	9.30	19.00	19.58	7.60	7.79
9	22.82	23.28	8.30	8.60	21.79	22.27	8.56	8.67
10	39.93	41.05	.	.	36.62	37.64	12110.76	12398.72

Table 3.11: Time (seconds) to solve 19x19 crossword puzzle problems. The absence of an entry indicates that the problem was not solved correctly within the given resource limits.

	dom+deg				*dom/deg*			
	UK		Linux		UK		Linux	
	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ
1	59.06	60.72	24.54	24.92	59.90	61.96	21.81	22.41
2	46.72	47.82	.	.	59.35	60.29	.	.
3	70.87	72.58	.	.	76.64	77.24	.	.
4	30.89	31.82	.	.	32.58	33.22	.	.
5	26.87	27.76	.	.	27.48	28.08	.	**547.92**
6	46.23	47.41	15.35	15.60	41.03	41.78	15.08	15.30
7	40.43	41.04	14.08	14.55	44.03	44.58	51.46	22.53
8	44.54	45.44	.	**35.40**	50.37	51.02	.	**42.00**
9	29.54	30.14	9.04	9.41	36.07	36.28	.	**37.46**
10	30.90	31.58	9.02	9.30	42.36	42.09	7.95	8.15

Table 3.12: Time (seconds) to solve 21x21 crossword puzzle problems. The absence of an entry indicates that the problem was not solved correctly within the given resource limits.

	dom+deg				*dom/deg*			
	UK		Linux		UK		Linux	
	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ
1	86.36	88.67	.	.	**12963.30**	**238.00**	.	.
2	109.29	108.69	.	.	94.87	97.40	.	.
3	92.33	87.75	69.20	70.81	85.10	86.01	98.97	99.70
4	202.54	206.07	.	.	**8779.56**	**2122.34**	.	.
5	127.76	128.41	.	.	109.89	110.93	.	.
6	86.77	87.65	**.**	**77.36**	97.08	98.00	.	.
7	93.26	95.89	98.05	89.23	94.81	96.89	43.54	41.37
8	76.33	78.84	30.65	31.13	92.88	95.21	37.73	37.43
9	114.79	119.22	77.14	55.62	101.07	100.26	52.97	53.08
10	**2093.13**	**950.52**	.	.	**23367.88**	**586.87**	.	.

Table 3.13: Time (seconds) to solve 23x23 crossword puzzle problems. The absence of an entry indicates that the problem was not solved correctly within the given resource limits.

	dom+deg				*dom/deg*			
	UK		Linux		UK		Linux	
	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ	GAC	GAC-CBJ
1	**.**	**158.05**	0.14	0.16	59.27	60.13	0.14	0.16
2	97.65	101.87	40.25	41.47	100.34	100.71	59.54	59.80
3	142.65	149.98	.	.	167.95	168.75	.	.
4	181.62	190.01	.	.	324.40	325.81	.	.
5	128.57	133.92	70.73	74.07	220.34	224.20	69.9	72.31
6	**.**	**24642.63**	.	.	609.70	607.88	.	.
7	112.85	117.80	.	.	183.11	175.09	.	.
8	496.64	511.62	.	.	**.**	**293.30**	.	.
9	289.43	291.44	.	.	534.76	541.41	.	.
10	.	.	.	.	2855.66	2921.58	.	.

dictionaries we used in the experiments. These constraints are very tight compared to all possible combinations of 26 letters. In the dual representation, each word in the puzzle is represented by a dual variable and thus the domain of a dual variable may have 5000 to 30000 values. A dual constraint enforces the instantiations of a pair of dual variables agreeing on their intersecting letter. In the hidden representation, each of the letters and each of the words in the puzzle are given a variable. A hidden constraint enforces an assignment of a letter variable to be compatible with an assignment of a word variable.

The original formulation, however, cannot be used to test GAC and GAC-CBJ, because of the large arity of the non-binary constraints. For example, a generic approach to seek a valid support for a value in the constraint propagation is to enumerate all possible value combinations and return the first support in the list. To revise a non-binary constraint over 10 letter variables, the number of value combinations is 26^{10}, and because the non-binary constraint is very tight, it is rare to encounter a valid support in the list. The number of potential enumerations is too large to be accepted.

Therefore, all experiments were run on the dual representation of the puzzle problem. Both GAC and GAC-CBJ use some specialized routines to take advantage of the dual constraints and thus speed up the constraint propagation. We use two dictionaries to solve the problem: the UK dictionary, which collects about $220,000$ words and in which the largest domain for a word variable contains about $30,000$ values, and the Linux dictionary, which collects $45,000$ words and in which the largest domain for a word variable has about $5,000$ values. Although use of a larger dictionary increases the size of search space, the number of solutions also increases. Generally the use of a larger dictionary makes the problem easier to solve. We tested 5 sets of puzzle instances, ranging from the easiest 5x5 puzzles, to the hardest 23x23 puzzles. Two heuristics, *dom+deg* and *dom/deg*, were used in the experiments.

The experimental results are shown in Table 3.9 to Table 3.13, each presenting the results for a set of instances. There are no noticeable difference between the performance of GAC and GAC-CBJ on smaller and easier puzzle problems, such as 5x5 ones. For 15x15 puzzles, GAC-CBJ runs an order of magnitude faster than GAC on instance 8 under *dom/deg* heuristic with the Linux dictionary. For other instances, the two algorithms perform similarly. The noticeable difference shows up on 19x19 puzzles. Those problems become hard for the Linux dictionary, as there are several absences in Table 3.11 for the Linux dictionary under both heuristics. With *dom+deg* heuristic, GAC-CBJ found a solution in less than one minute for instance 8 while

GAC failed to solve the problem in 20 hours CPU time. With *dom/deg* heuristic, the difference is even more dramatic, GAC-CBJ found a solution very quickly for instance 5, 8 and 9, while GAC ran out of the time limit on these instances. 19x19 puzzles formulated with the UK dictionary are not hard as both algorithms can solve them very quickly under both variable ordering heuristics. The 21x21 and 23x23 puzzles are too hard for the Linux dictionary. Both algorithms time out in solving these instances. The difference in behaviors between GAC and GAC-CBJ shows up for the UK dictionary. We observed orders of magnitude improvement by GAC-CBJ on instance 1, 4, and 10 under *dom/deg* heuristic for those 21x21 puzzles. On 23x23 puzzles, GAC was more likely to time out but GAC-CBJ still could solve all the instances under *dom/deg* heuristic within 20 hours time limit. The cases that GAC-CBJ improves GAC are not rare, especially for hard problems. For example, GAC-CBJ can solve three out of ten 19x19 puzzles within 10 minutes with the Linux dictionary under *dom/deg* heuristic, which cannot be solved by GAC within the time limit.

3.6 Summary

We have given some theoretical evidence to show that look ahead techniques are counterproductive to backjumping. In general, there is a close relation between an algorithm maintaining strong k-consistency and a backjumping algorithm doing a limited level of backjumps. Then we presented our implementation of GAC-CBJ, a generalized version of MAC-CBJ on general CSPs. Experimental results show that GAC-CBJ can significantly improve the backtrack search on large, hard real world problems, and the overhead of CBJ on those easy instances is negligible.

Chapter 4

Dual and Hidden Transformations with Consistencies

The dual and hidden transformations are two general methods to convert a non-binary CSP into an equivalent binary CSP. The dual graph representation was introduced to the CSP community by Dechter and Pearl in the tree clustering method [38], and the basic idea comes from research in relational databases [79]. The hidden variable transformation has an even longer history. Peirce formally proved, in the field of philosophic logic, that binary and non-binary relations have the same expressive power [88], and Peirce's method for representing a non-binary relation with a collection of binary relations forms the foundation of the hidden variable method. Recently, Rossi *et al* showed that that a non-binary CSP is equivalent to its dual and hidden transformations under various definitions of equivalence in [95]. In the past, because such conversions exist between a non-binary CSP and its equivalent binary CSP representation, most solving techniques for CSPs have been restricted to binary CSPs. Moreover, the dual and hidden transformation methods are widely used in modeling practice and problem solving. For example, Freuder had used an incremental version of the dual method in a solution synthesis method [42]. Dechter shows how to represent any non-binary relation with binary relations using hidden variables that have bounded domain sizes [33].

4.1 Definitions

In the dual transformation, the constraints of the original problem become variables in the new representation. We refer to these variables, which represent the original constraints, as *dual variables*, and the variables in the the original CSP as *ordinary*

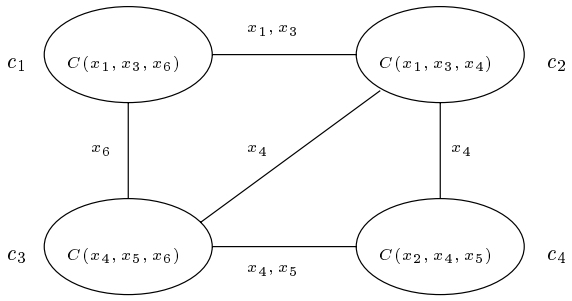


Figure 4.1: The dual transformation of the CSP in Example 1.1

variables. The domain of each dual variable is exactly the set of tuples that satisfy the original constraint and there is a binary constraint between two dual variables *iff* two original constraints share some variables. We refer to the binary constraints as *dual constraints*. A dual constraint prohibits pairs of tuples which do not agree on the shared variables.

Definition 4.1 (dual transformation) *Given a CSP instance* $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$*, its dual transformation* $dual(P) = (\mathcal{V}^{dual(P)}, \mathcal{D}^{dual(P)}, \mathcal{C}^{dual(P)})$ *is defined as:*

- $\mathcal{V}^{dual(P)} = \{c_1, \ldots, c_m\}$ *and they are called* dual variables. *Each dual variable* c_i *corresponds to a unique constraint* $C_i \in \mathcal{C}$*. In the following discussion, we may use* $vars(c_i)$ *and* $rel(c_i)$ *to denote their correspondences* $vars(C_i)$ *and* $rel(C_i)$ *respectively if there are no ambiguities,*

- $\mathcal{D}^{dual(P)} = \{dom(c_1), \ldots, dom(c_m)\}$ *is the set of domains for the dual variables. For each dual variable* c_i*,* $dom(c_i) = rel(C_i)$*,* i.e.*, each value for* c_i *is a tuple over* $vars(C_i)$*,*

- $\mathcal{C}^{dual(P)}$ *is a set of binary constraints over* $\mathcal{V}^{dual(P)}$ *and they are called* dual constraints. *There is a dual constraint between dual variables* c_i *and* c_j *if* $vars(c_i) \cap vars(c_j) \neq \emptyset$ *such that a tuple* $a \in dom(c_i)$ *is compatible with a tuple* $b \in dom(c_j)$ iff $a[vars(c_i) \cap vars(c_j)] = b[vars(c_i) \cap vars(c_j)]$, i.e., *they have the same values over the common variables,* $vars(c_i) \cap vars(c_j)$*.*

Example 4.1 *In the dual graph transformation of the CSP in Example 1.1, there are* 4 *dual variables,* $c_1, \ldots, c_4$*, one for each* 3*-ary constraint in the original problem as shown in Figure 4.1. For example, the dual variable* c_1 *corresponds to the non-binary*

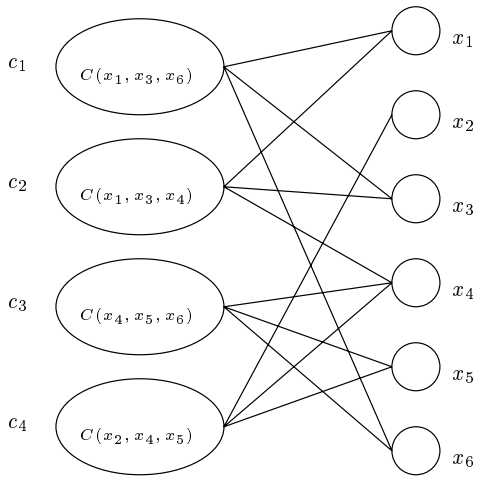


Figure 4.2: The hidden transformation of the CSP in Example 1.1

constraint $C(x_1, x_3, x_6)$ *and the domain of* c_1 *contains the tuples* $(0,0,1), \ldots, (1,1,1)$. *The dual constraints enforce that the ordinary variables appearing in more than one dual variable have the same value. For example, in the dual constraint between* c_1 *and* c_2, $\{c_1 \leftarrow (0,0,1)\}$ *is compatible with* $\{c_2 \leftarrow (0,0,0)\}$, *but* $\{c_1 \leftarrow (0,0,1)\}$ *is not compatible with* $\{c_2 \leftarrow (0,1,0)\}$.

In the hidden-variable transformation, the set of variables includes all the ordinary variables in the original problem with their original domains, plus a new set of *hidden variables*. For each constraint in the original problem, we add a hidden variable. The domain of the hidden variable is the same as the domain of the dual variable, consisting of the set of tuples that satisfy the original constraint. There is a binary constraint between a hidden variable and an ordinary variable, if in the original problem the constraint represented by the hidden variable constrains that ordinary variable. The binary constraint is called a *hidden constraint*. A hidden constraint enforces that a value of the ordinary variable must be the same as the value assigned to that ordinary variable in a tuple of the hidden variable.

Definition 4.2 (hidden-variable transformation) *Given a CSP instance* $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, *its hidden variable transformation, or hidden transformation in short,* $hidden(P) = (\mathcal{V}^{hidden(P)}, \mathcal{D}^{hidden(P)}, \mathcal{C}^{hidden(P)})$ *is defined as:*

- $\mathcal{V}^{hidden(P)} = \{x_1, \ldots, x_n\} \cup \{c_1, \ldots, c_m\}$, *where* $x_1, \ldots, x_n$ *are called* ordinary variables *and* $c_1, \ldots, c_m$ *are called* hidden variables. *Similar to dual variables, each hidden variable* c_i *corresponds to a unique constraint* $C_i \in \mathcal{C}$,

- $\mathcal{D}^{hidden(P)} = \{dom(x_1), \ldots, dom(x_n)\} \cup \{dom(c_1), \ldots, dom(c_m)\}$. *For each hidden variable* c_i, $dom(c_i) = rel(C_i)$,

- $\mathcal{C}^{hidden(P)}$ *is a set of binary constraints over* $\mathcal{V}^{hidden(P)}$ *and they are called* hidden constraints. *For each hidden variable* c, *there is a hidden constraint between* c *and each of the ordinary variables* $x \in vars(c)$ *such that a tuple* $t \in dom(c)$ *is compatible with a value* $a \in dom(x)$ if $t[x] = a$. *Thus each of the tuples* $t \in dom(c)$ *corresponds to a unique value* $t[x] \in dom(x)$.

The hidden transformation has some special properties. For example, the constraint graph of the hidden transformation is a bipartite graph, *i.e.*, the ordinary variables are only constrained with the hidden variables, and *vice versa*, and the hidden constraints are one-way functional constraints, in which a tuple in the domain of the hidden variable is compatible with at most one value in the domain of the ordinary variable.

Example 4.2 *In the hidden variable transformation of the CSP in Example 1.1, there are* 10 *variables, including* 6 *ordinary variables from the original problem, and* 4 *hidden variables, one for each of the original constraints, as shown in Figure 4.2. For example, the constraint* $C(x_1, x_3, x_6)$ *corresponds to hidden variable* c_1, *whose domain is the set of tuples* $\{(0,0,1), \ldots, (1,1,1)\}$. *The hidden constraints enforce a value of the ordinary variable to agree with a tuple of the hidden variable. For example, in the hidden constraint between* c_1 *and* x_1, $\{c_1 \leftarrow (0,0,1)\}$ *is compatible with* $\{x_1 \leftarrow 0\}$, *but* $\{c_1 \leftarrow (0,0,1)\}$ *is not compatible with* $\{x_1 \leftarrow 1\}$.

A dual or hidden representation does not always arise as a transformation of the original CSP formulation. Sometimes, it is very natural to model the problem as a dual or hidden representation (of an "original" formulation).

Example 4.3 *Crossword puzzle generation can be formulated as a CSP. Figure 4.3 shows a crossword puzzle with 5 by 5 grid. One such formulation consists of 19 variables* $\{x_1, \ldots, x_{19}\}$, *and each variable takes values from an alphabet set* $\{a, \ldots, z\}$. *There are 3 unknown words with length 3 and 5 unknown words with length 5 in the puzzle, resulting in 8 non-binary constraints in the CSP. The tuples of the constraints are the words with 3 or 5 letters in a pre-defined dictionary. In its dual representation, each of the unknown words is represented by a dual variable which takes values from the dictionary. A pair of dual variables are constrained such that they have the same letter in the crossing grid. Thus, there are 8 dual variables and 15 dual constraints*

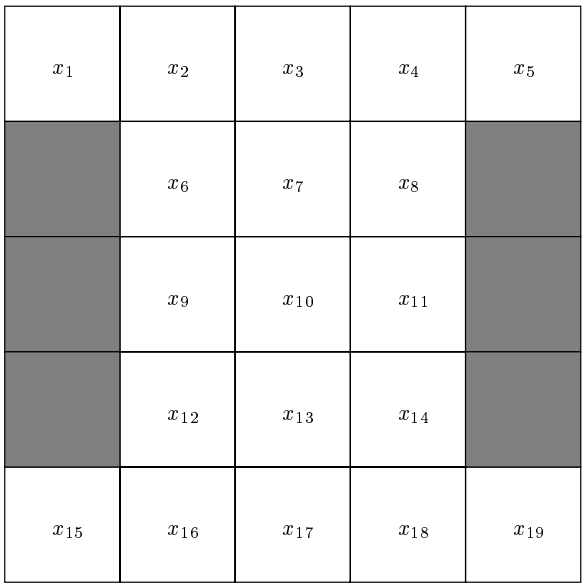


Figure 4.3: A crossword puzzle.

in the dual representation. Its hidden representation has 19 ordinary variables, and 8 hidden variables corresponding to the 8 non-binary constraints in the original problem. Each of the hidden variables is constrained with 3 or 5 ordinary variables in its scheme. Thus, the hidden representation has 34 hidden constraints.

In this chapter, given the original formulation of a problem, and its dual and hidden transformations, we are going to compare various local consistency properties on the above three formulations. Similar to Debruyne and Bessière's approach to comparing some selected local consistency properties on binary CSPs in [28], we identify a "strongness" relation between two pairs of consistency property and formulation.

Definition 4.3 *Given two local consistency properties $\mathcal{LC}_1$ and $\mathcal{LC}_2$, and two CSP formulations for a problem $\mathcal{A}$ and $\mathcal{B}$, We say $\mathcal{LC}_1$ on formulation $\mathcal{A}$ is stronger than $\mathcal{LC}_2$ on formulation $\mathcal{B}$* iff *given any problem, if $\mathcal{LC}_1$ can be achieved on $\mathcal{A}$ without an empty resulting problem, then $\mathcal{LC}_2$ can also be achieved on $\mathcal{B}$ without an empty resulting problem, and $\mathcal{LC}_1$ on $\mathcal{A}$ is strictly stronger than $\mathcal{LC}_2$ on $\mathcal{B}$* iff *furthermore there is a CSP instance on which $\mathcal{LC}_2$ can be achieved on $\mathcal{B}$ without an empty resulting problem but $\mathcal{A}$ will experience a domain being wiped out when enforcing $\mathcal{LC}_1$ on it. We say $\mathcal{LC}_1$ on $\mathcal{A}$ is* equivalent *to $\mathcal{LC}_2$ on $\mathcal{B}$ if $\mathcal{LC}_1$ on $\mathcal{A}$ is stronger than $\mathcal{LC}_2$ on $\mathcal{B}$, and* vice versa.

In the following, we call a CSP instance the *original problem* with respect to its dual transformation and hidden transformation. Because we usually deal with more than one CSP formulation in a context, to our convenience, given a CSP formulation P, we use the notations $\mathcal{V}^P$, $\mathcal{D}^P$ and $\mathcal{C}^P$ to denote the set of variables, the set of domains and the set of constraints in P respectively. Also, we use $dom^P(x)$ to denote the domain of variable x in P. The notations, $\mathcal{V}$, $\mathcal{D}$, $\mathcal{C}$, and $dom(x)$ are still used if there are no ambiguities in the context.

4.2 Related Work

The dual and hidden transformations are two fundamental methods in modeling practice. They completely change the original formulation such that all the variables and constraints have to be rewritten. A serious drawback of these transformations is that if the non-binary constraints are represented implicitly, the above transformations could take exponential time and demand exponential space with respect to the size of the original problem. Sometimes it may require that the original problem be solved. For example, if an n-ary constraint over $x_1, \ldots, x_n$ is represented by a function or predicate $f(x_1, \ldots, x_n)$, which returns true if the assignments to the variables satisfy the constraint, in the dual (hidden) transformation, each legal tuple in a constraint becomes a value in the domain of the corresponding dual (hidden) variable. Thus, it may take exponential steps to find all these tuples and use exponential space to store these tuples in the domain of the dual (hidden) variable. In the CSP formulation of the "Send+More=Money" puzzle in Example 1.3, if we try to represent two global constraints, the "equation" constraint and "alldifferent" constraint by two dual (hidden) variables, we can then solve the problem by looking up the domains of two dual (hidden) variables and picking up common tuples in the two domains. In practice, often, a partial conversion is used to improve the formulation. That is, a subset of the constraints become dual variables, or a subset of the variables in a constraint are aggregated into a hidden variable and thus the arity of the constraint is lowered. Furthermore, with the help of identifying meta values in dual or hidden variables, the domains of the dual or hidden variables may be condensed to an acceptable degree.

In temporal reasoning, there are two approaches to representing temporal information. In the point-based representation [116, 117], each event A is identified by a pair of end points, A^- and A^+, denoting the starting time and finishing time of the event. The temporal relations among events are represented by a set of inequalities, equalities and disequalities between any of two end points. For example, the relation that A

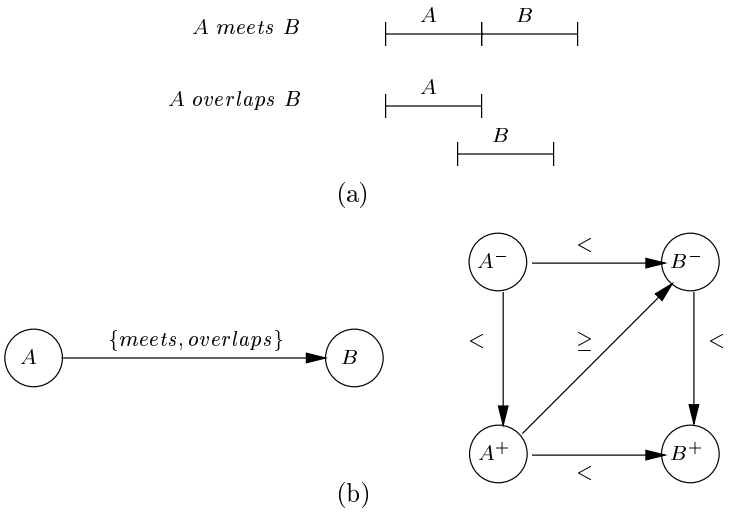


Figure 4.4: An example of translation between interval-based and point-based representation for temporal information.

must finish before B starts can be represented by $\{A^- < A^+, A^+ < B^-, B^- < B^+\}$. In the interval-based representation [3, 4], each event A is identified by an interval. A temporal relation between two events is represented by a conjunction of *basic* relations. Allen [3] has identified a set of thirteen basic relations between two intervals and the relation between two events can be any subset of the set of basic relations. For example, as shown in Figure 4.4(a), one basic relation between interval A and B says A *meets* B, which means that B starts at the same time as A finishes. The basic relation A *overlaps* B denotes the scenario in which A starts before B starts, A finishes after B starts, and A finishes before B finishes. For a fuller description of the basic relations, see Allen's paper [3].

Both the point-based representation and the interval-based representation can be formalized as CSPs. The domain of an end point is a set of time points allowed for the event under consideration. The domain of an interval contains all possible ordered pair of time points. The definitions of the constraints are straightforward from the temporal relations between two end points or two intervals respectively. The interval-based CSP can be regarded as a partial dual conversion of the point-based CSP [1]. For example, as shown in Figure 4.4(b) the interval relation $\{meet, overlaps\}$ between event A and B can be represented as,

$$(A^- < A^+) \wedge (B^- < B^+) \wedge (A^- < B^+) \wedge (A^+ \geq B^-) \wedge (A^+ < B^-)$$

[1] In general, such a conversion may not exist due to the restricted types of the relations adopted in the two representation schemes [70, 117].

In such a conversion, some of the constraints, $A^- < A^+$ and $B^- < B^+$, become dual variables and the other constraints are transformed into constraints over the dual variables. Also, when actually finding a consistent scenario for these networks, we solve the dual representation, which can often be processed very quickly [71].

The concept of a dual is also reflected in problem solving techniques. In a constraint scheduling problem, an activity A is represented by a pair of variables, $start(A)$ and $end(A)$, denoting the starting and release time of A respectively. The domains of the variables contain all possible time points for the activity. There are two main classes of constraints, *precedence constraints* to specify sequencing requirements among activities; for example, activity A must be scheduled to start at least 5 steps after activity B is released, $start(A) \geq end(B) + 5$, and *resource constraints* to specify that several tasks may compete for a resource with a certain capacity. Disjunctive constraints are widely used to ensure that the time intervals over which two activities require the same resource do not overlap in time [73, 104]. For example, the constraint $(end(A) < start(B)) \vee (start(A) > end(B))$ ensures that the time intervals over two activities do not overlap in time when both A and B demand a unique resource. The time bounds of activities, *i.e.*, the domains of variables, can be improved by a series of propagation techniques, known as edge-finding rules [5, 9]. To prove optimality of schedules, a *branch and bound* search is used. It is a general wisdom to not *branch on variables*, as a traditional backtracking algorithm does, but *branch on constraints*, especially the disjunctive constraints. For example, given a disjunctive constraint $(end(A) < start(B)) \vee (start(A) > end(B))$, we may choose $end(A) < start(B)$ or $start(A) > end(B)$ as a choice point. The effect of branching on disjunctive constraints essentially establishes an order of activities which compete for a resource. New bounds can then be deduced after branching on the constraints and used to prove optimality. For example, in the job shop scheduling problem, the operation of branching on constraints is known as *edge directing* [21, 22], where the job-shop problem is represented in a disjunctive graph, in which all precedence constraints are represented by directed edges and all resource constraints are represented by undirected edges. Thus establishing an order is essential to giving a preference to all the undirected edges. Branching on constraints coincides with the idea of the dual, that is, treating constraints as variables. In the domain of scheduling problems, such techniques may bring significant improvements.

The dual and hidden variable techniques also help in the representation of large complex constraints. To model a problem as a CSP and solve the CSP using a backtracking algorithm, a central problem is how to represent a constraint in an economic

and efficient manner. A general guideline is to limit the number of the variables in each of the constraints. However, real world constraints are usually non-binary and the redundant constraints are more likely non-binary. The question turns out to be how to decompose a large constraint into several small constraints. An intuitive approach is to project the constraint onto a set of pairs of variables and the projections are binary constraints. Unfortunately, such projections may lose information stored in the original constraint [31]. Another approach is to add projections for each of the subsets of the variables in the original constraint. This approach is dangerous as a possible exponential number of constraints would be added.

A general approach is to add extra hidden variables. The hidden variable transformation uses a star-decomposition scheme by adding one hidden variable. A drawback of the star-decomposition is that the domain size of the hidden variable cannot be bounded to an acceptable degree. In fact, there could be many possible ways to decompose a constraint by adding more than one hidden variable. Example 1.4 showed how to decompose the large "equation" constraint into 3 smaller constraints by adding some "carrier" (hidden) variables. Dechter [33] proposes a tree-structured decomposition scheme by adding multiple hidden variables with bounded domain sizes. Unfortunately, the number of the hidden variables is not bounded and there are cases that an exponential number of hidden variables are required.

4.3 Arc Consistency

Arc consistency (see Definition 2.4) is an important concept in constraint programming. Because achieving arc consistency on a CSP only changes the domains of the variables, it is moderately cheaper than achieving strong k-consistency, for $k > 2$, which may dramatically change the CSP formulation, as the number of new constraints could be exponential in k. In this section, we are going to compare arc consistency on the original CSP with arc consistency on its dual and hidden transformations. The comparison is based on the justification that if one formulation is not empty after achieving arc consistency, the other formulation is not empty either after achieving arc consistency. (Thus arc consistency on the first formulation is at least as strong as arc consistency on the second formulation.) The relations between arc consistency on the original problem and the one on its dual and hidden transformations have been studied by Stergiou and Walsh [106]. However, they only give some illustrative proofs for their results. We present here stricter proofs for the above relations based on the formal definitions of the dual and hidden transformations.

4.3.1 Arc Consistency Closure

As we discussed in Chapter 3, two strong k-consistency achievement algorithms may not always compute the same results when achieving strong k-consistency on a CSP. For arc consistency, as we are going to show in the following, the resulting arc consistent CSP is unique. We achieve arc consistency on a CSP by repeatly removing from the domains those values that are not supported in a constraint. When a value is removed from its domain, some tuples using the value in a constraint involving that variable become *invalid*. The invalid tuples are removed from the constraint implicitly. The changes in one domain are propagated to other variables for which a new support needs to be sought in the tightened constraints.

Definition 4.4 (subdomain) *A subdomain $\mathcal{D}'$ of a CSP P is a set of domains, $\{dom^{\mathcal{D}'}(x_1),\ldots,dom^{\mathcal{D}'}(x_n)\}$, where $dom^{\mathcal{D}'}(x_i) \subseteq dom^{P}(x_i)$, for each of the variables $x_i \in \mathcal{V}$. In the following, we use the notation $dom^{\mathcal{D}'}(x)$ to denote the domain of variable x in a subdomain $\mathcal{D}'$. We say a subdomain is empty if it contains one empty domain for a variable. We say a subdomain $\mathcal{D}'$ is* arc consistent iff *for each of the constraints $C \in \mathcal{C}$, each of the variables $x \in vars(C)$ and each of the values $a \in dom^{\mathcal{D}'}(x)$, $\{x \leftarrow a\}$ has at least one support t in C, where $t[x] = a$, and for each of the variables $y \in vars(C)$, $t[y] \in dom^{\mathcal{D}'}(y)$. Given two subdomains of P, $\mathcal{D}_1$ and $\mathcal{D}_2$, we use $\mathcal{D}_1 \subseteq \mathcal{D}_2$ to denote the fact that for each of the variables $x \in \mathcal{V}$, $dom^{\mathcal{D}_1}(x) \subseteq dom^{\mathcal{D}_2}(x)$.*

Note that a subdomain is a set of domains, each associated with a variable in the CSP. Under the above $\subseteq$ relation, the *maximum subdomain* is the set of the original domains in the CSP, and the *minimal subdomain* is the set of empty domains; *i.e.*, each variable has an empty domain. It is easy to verify that the minimal subdomain is arc consistent.

Theorem 4.1 *Given two arc consistent subdomains $\mathcal{D}_1$ and $\mathcal{D}_2$ of a CSP P, the union of $\mathcal{D}_1$ and $\mathcal{D}_2$, $\mathcal{D}'$, where $dom^{\mathcal{D}'}(x) = dom^{\mathcal{D}_1}(x) \cup dom^{\mathcal{D}_2}(x)$ for each variable x in P, is an arc consistent subdomain of P.*

Proof: Because for each of the values $a \in dom^{\mathcal{D}'}(x)$, either $a \in dom^{\mathcal{D}_1}(x)$ or $a \in dom^{\mathcal{D}_2}(x)$, it is easy to verify that for each constraint C in P, each of the variables $x \in vars(C)$, and for each value $a \in dom^{\mathcal{D}'}(x)$, $\{x \leftarrow a\}$ has at least one support in C. ∎

Since the union of two arc consistent subdomains is still arc consistent and there are a finite number of subdomains in a CSP P, the union of all the arc consistent subdomains is arc consistent. Furthermore, the set of arc consistent subdomains of P is not empty because it always includes the minimal subdomain in which each variable has an empty domain. Thus, there is a unique maximum arc consistent subdomain of P and each arc consistency achievement algorithm should compute the maximum arc consistent subdomain. We denote the resulting CSP by $ac(P)$, called the *arc consistency closure* of P. Thus the maximum arc consistent subdomain of the original CSP, *i.e.*, the set of domains in $ac(P)$, is $\mathcal{D}^{ac(P)}$.

Corollary 4.2 *Given a CSP P and an arc consistent subdomain $\mathcal{D}'$ of P, $\mathcal{D}' \subseteq \mathcal{D}^{ac(P)}$.*

From the above corollary, an arc consistency subdomain of a CSP P is also an arc consistent subdomain of $ac(P)$.

4.3.2 Arc Consistency on the Hidden Transformation

Consider a CSP P with 4 variables, $x_1, \ldots, x_4$. The domain for each variable has 3 values $0, 1$, and 2. There are 3 linear constraints between these variables, $x_1 + x_2 < x_3$, $x_1 + x_3 < x_4$ and $x_2 + x_3 < x_4$. Figure 4.5 shows the relations between the original CSP P, its arc consistency closure $ac(P)$, its hidden transformation $hidden(P)$, and the arc consistency closure of its hidden transformation $ac(hidden(P))$ which is also the hidden transformation of its arc consistency closure $hidden(ac(P))$. As we can see from the above figure, an ordinary variable has the same domain in $ac(P)$ and $ac(hidden(P))$. The domain of a hidden variable in $ac(hidden(P))$ is the same as the set of tuples in the corresponding constraint, which have not been (implicitly) removed from the constraint when achieving arc consistency on the original problem. We are going to show in the following that the above relations are generally true.

Theorem 4.3 *Given a CSP instance P, (1) P is arc consistent if and only if $hidden(P)$ is arc consistent; (2) $ac(P)$ is not empty if and only if $ac(hidden(P))$ is not empty; and (3) $hidden(ac(P)) = ac(hidden(P))$.*

Proof: (1) Suppose the original problem P is not arc consistent. There is a value a in the domain of an ordinary variable x and a constraint C such that $x \leftarrow a$ does not have a support in C. Thus, in $hidden(P)$, $x \leftarrow a$ does not have a support in the

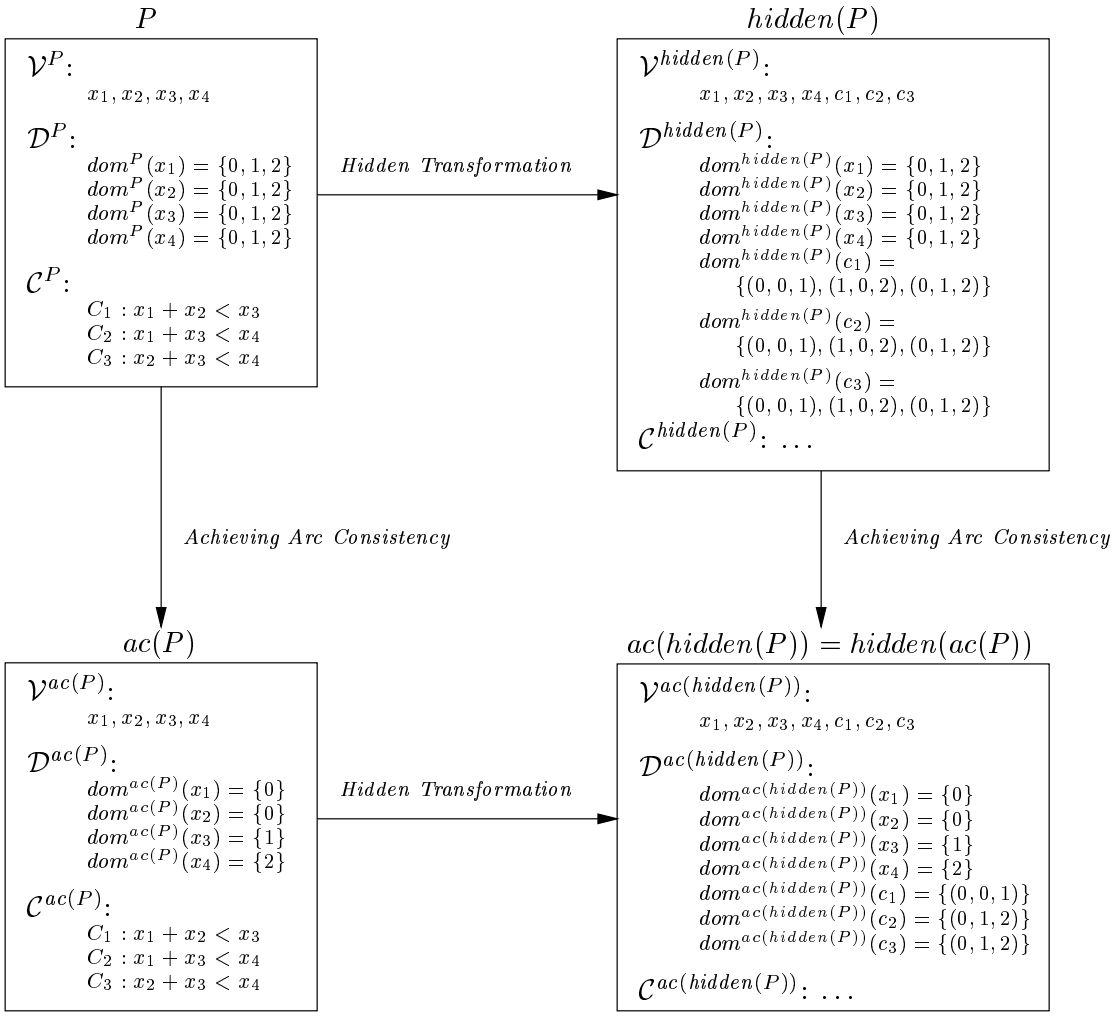


Figure 4.5: An example to show the relations between an original CSP, its hidden transformation, its arc consistency closure, the arc consistency closure of its hidden transformation, and the hidden transformation of its arc consistency closure.

hidden constraint between the ordinary variable x and the corresponding hidden variable c. Thus, $hidden(P)$ is not arc consistent. On the contrast, suppose $hidden(P)$ is not arc consistent. From the definition of the hidden transformation, a tuple in the domain of a hidden variable always has a support in a hidden constraint between the hidden variable and an ordinary variable. (Note that a tuple in a constraint is an element of the Cartesian product of the domains of the variables in the constraint.) Thus, in $hidden(P)$, there is a value a of an ordinary variable x and a hidden variable c such that $\{x \leftarrow a\}$ does not have a support in the hidden constraint between x and c. Thus $\{x \leftarrow a\}$ does not have a support in the corresponding original constraint C in the original problem. Therefore, P is not arc consistent. That is, P is arc consistent if and only if $hidden(P)$ is arc consistent.

(2) Suppose $ac(P)$ is not empty. Thus its hidden transformation $hidden(ac(P))$ is arc consistent and not empty. Because the set of the domains of $hidden(ac(P))$ is an arc consistent subdomain of $hidden(P)$, from Corollary 4.2, for each ordinary variable x, $dom^{hidden(ac(P))}(x) \subseteq dom^{ac(hidden(P))}(x)$, and for each hidden variable c, $dom^{hidden(ac(P))}(c) \subseteq dom^{ac(hidden(P))}(c)$. Therefore, $ac(hidden(P))$ is not empty. On the other hand, suppose $ac(hidden(P))$ is not empty, then the set of the domains of all the ordinary variables in $ac(hidden(P))$ is an arc consistent subdomain of P. That is, for each ordinary variable x, $dom^{ac(hidden(P))}(x) \subseteq dom^{ac(P)}(x)$. Thus $ac(P)$ is not empty.

(3) The hidden transformation of $ac(P)$, $hidden(ac(P))$, has the same variables as those in $ac(hidden(P))$, including ordinary variables and hidden variables. For an ordinary variable x in $hidden(ac(P))$, we know the facts that $dom^{hidden(ac(P))}(x) = dom^{ac(P)}(x)$, $dom^{hidden(ac(P))}(x) \subseteq dom^{ac(hidden(P))}(x)$ and $dom^{ac(hidden(P))}(x) \subseteq dom^{ac(P)}(x)$, thus $dom^{hidden(ac(P))}(x) = dom^{ac(hidden(P))}(x)$. Now we consider the hidden variables. For each hidden variable c in $ac(hidden(P))$ and for each tuple t in the domain of c, because $t[x] \in dom^{ac(hidden(P))}(x)$ and thus $t[x] \in dom^{ac(P)}(x)$ for each of the ordinary variables $x \in vars(c)$, t will not be (implicitly) removed from the constraint C when achieving arc consistency on the original problem. Thus t is a tuple in the domain of the hidden variable c in $hidden(ac(P))$. We have $dom^{ac(hidden(P))}(c) \subseteq dom^{hidden(ac(P))}(c)$. On the other hand, for each constraint C in $ac(P)$ and for each tuple t in $rel(C)$, because t is not (implicitly) removed from $rel(C)$ when achieving arc consistency on the original problem, that is, $t[x] \in dom^{ac(P)}(x)$ and thus $t[x] \in dom^{ac(hidden(P))}(x)$ for each of the ordinary variables $x \in vars(C)$, the tuple t is not removed from the domain of the hidden variable c when achieving arc consistency on $hidden(P)$ (otherwise, add t to the domain of c and the hidden problem

is still arc consistent). We have $dom^{hidden(ac(P))}(c) \subseteq dom^{ac(hidden(P))}(c)$. Therefore, for each of the hidden variables c, $dom^{hidden(ac(P))}(c) = dom^{ac(hidden(P))}(c)$. The hidden constraints in a hidden transformation are set automatically having specified the ordinary variables, the hidden variables, and their domains. Thus, $hidden(ac(P))$ and $ac(hidden(P))$ have exactly the same set of variables, the same set of domains, and the same set of hidden constraints; that is, they are syntactically the same. Therefore, $hidden(ac(P)) = ac(hidden(P))$. ∎

Theorem 4.3 was obtained independently by Stergiou and Walsh in [106].

Corollary 4.4 *Given a CSP P, $dom^{ac(P)}(x) = dom^{ac(hidden(P))}(x)$, for each of the ordinary variables x in P.*

Although achieving arc consistency on the original problem is "equivalent" to achieving arc consistency on its hidden transformation, their performance could be quite different. The worst case complexity of achieving arc consistency on the original problem by AC3 is $O(mr^2d^{r+1})$ [76]. In its hidden transformation, there are n ordinary variables and m hidden variables. The maximum domain size of the ordinary variables is d and the maximum domain size of the hidden variables is bounded by d^r. Thus it takes $O(d^{r+1})$ steps to revise a hidden constraint. During the execution of AC3, each time a value is removed from the domain of a variable x, all the constraints involving x need to be revised. Let $deg(x)$ denote the degree of x. The total number of the steps in the execution of AC3 on the hidden problem is bounded by $O(\sum_{x \in \mathcal{V}^{hidden(P)}} deg(x) d^{2r+1})$. Note that the term $\sum_{x \in \mathcal{V}^{hidden(P)}} deg(x)$ can be replaced by mr, which is a bound on the number of the constraints in $hidden(P)$. Thus the worst case complexity of achieving arc consistency on the hidden problem by AC3 is $O(mrd^{2r+1})$. Achieving arc consistency on the hidden problem is more expensive than on the original problem unless the constraints are very tight. In that case, let M denote the maximum domain size of the hidden variables, and the worst case complexity of AC3 on the hidden problem is rewritten as $O(mrdM^2)$. As discussed in Chapter 3, in the original formulation of the crossword puzzle problems, the constraints are very tight. For instance, there are about 30000 tuples among 10^{26} possible value combinations in a non-binary constraint which represents an unknown word of 10 letters. It is less expensive to achieve arc consistency on the hidden representation than on the original non-binary problem. In contrast, if the constraints in the original problem are very loose, AC3 on the original problem can perform much better than the worst case complexity because it is easy to find a support in a constraint. Furthermore,

AC3 on the original problem can use some specialized methods or propagators to speed up constraint propagation for some classes of constraints, *e.g.*, an alldifferent constraint can be revised in $O(dr^{1.5})$ worst case time [94]. Constraint propagation in the hidden problem can also be improved by exploiting the special properties (the bipartite graph topology and the one-way functionality of the hidden constraints) of the hidden transformation. For example, when a tuple t in the domain of a hidden variable needs to find a support in a hidden constraint, we need not go through the whole domain of the constrained ordinary variable x, but just check whether $t[x]$ is in the domain of x at a constant cost.

4.3.3 Arc Consistency on the Dual Transformation

We have identified the equivalence relation that the original problem is arc consistent if and only if its hidden transformation is arc consistent. However, such an equivalence does not hold in the case of the dual transformation.

Example 4.4 *Consider a CSP P with 4 Boolean variables, $x_1, \ldots, x_4$, and three constraints*

$$\begin{aligned} C(x_1, x_2, x_3) &= \{(0,0,0), (1,1,1)\}, \\ C(x_2, x_3, x_4) &= \{(0,0,0), (1,1,1)\}, \\ C(x_1, x_3, x_4) &= \{(0,0,1), (1,1,0)\}. \end{aligned}$$

The original problem P is arc consistent. In its dual transformation, let the dual variables c_1, c_2, and c_3 correspond to the above constraints, respectively. Because neither of the tuples $(0,0,0)$ and $(1,1,1)$ in the domain c_2 has a support in the dual constraint between c_2 and c_3, the domain of c_2 is wiped out after achieving arc consistency on the dual transformation. Thus $dual(P)$ is not arc consistent and $ac(dual(P))$ is empty.

Example 4.5 *Consider a CSP P with three Boolean variables, x_1, x_2 and x_3, and three constraints*

$$\begin{aligned} C(x_1, x_2) &= \{(1,1)\}, \\ C(x_2, x_3) &= \{(1,1)\}, \\ C(x_1, x_3) &= \{(1,1)\}. \end{aligned}$$

The dual transformation $dual(P)$ is arc consistent. However, the original problem is not arc consistent, because value 0 for each of the variables will be removed from the domain when achieving arc consistency.

In Example 4.5, although the original problem is not initially arc consistent, it can be made arc consistent without a domain being wiped out. We will show that if the dual problem is not empty after achieving arc consistency, then the original problem is not empty either after achieving arc consistency.

Lemma 4.5 *Given a subdomain $\mathcal{D}'$ of the dual problem $dual(P)$, $\mathcal{D}'$ is arc consistent if and only if for each pair of dual variables c_i and c_j in $dual(P)$, where $vars(c_i) \cap vars(c_j) \neq \emptyset$, and for each subset of the ordinary variables $S \subseteq vars(c_i) \cap vars(c_j)$, $\pi_S dom^{\mathcal{D}'}(c_i) = \pi_S dom^{\mathcal{D}'}(c_j)$.*

Proof: The *if part:* In the dual problem, there is a dual constraint between a pair of dual variables c_i and c_j if $vars(c_i) \cap vars(c_j) \neq \emptyset$. Because $\pi_{vars(c_i) \cap vars(c_j)} dom^{\mathcal{D}'}(c_i) = \pi_{vars(c_i) \cap vars(c_j)} dom^{\mathcal{D}'}(c_j)$, for each of the tuples $t_i \in dom^{\mathcal{D}'}(c_i)$, there exists a tuple t_j in $dom^{\mathcal{D}'}(c_j)$ such that t_i and t_j agree on the part $vars(c_i) \cap vars(c_j)$. So $\{c_i \leftarrow t_i\}$ has a valid support in the dual constraint. Thus $\mathcal{D}'$ is arc consistent. The *only if part:* For each of the tuples $t \in \pi_S dom^{\mathcal{D}'}(c_i)$, there is a tuple $t_i \in dom^{\mathcal{D}'}(c_i)$ such that $t_i[S] = t$. Because $\mathcal{D}'$ is arc consistent, there is a tuple $t_j \in dom^{\mathcal{D}'}(c_j)$ such that $t_i[vars(c_i) \cap vars(c_j)] = t_j[vars(c_i) \cap vars(c_j)]$. Thus $t = t_i[S] = t_j[S]$. Because $t_j[S] \in \pi_S dom^{\mathcal{D}'}(c_j)$, we have $\pi_S dom^{\mathcal{D}'}(c_i) \subseteq \pi_S dom^{\mathcal{D}'}(c_j)$. Similarly, we can show that $\pi_S dom^{\mathcal{D}'}(c_j) \subseteq \pi_S dom^{\mathcal{D}'}(c_i)$. Therefore, $\pi_S dom^{\mathcal{D}'}(c_i) = \pi_S dom^{\mathcal{D}'}(c_j)$. ∎

Theorem 4.6 *$dual(P)$ is arc consistent if and only if for each pair of dual variables c_i and c_j, where $vars(c_i) \cap vars(c_j) \neq \emptyset$, and for each subset of the ordinary variables $S \subseteq vars(c_i) \cap vars(c_j)$, $\pi_S dom^{dual(P)}(c_i) = \pi_S dom^{dual(P)}(c_j)$.*

Proof: The theorem is true by considering the set of the original domains in $dual(P)$ in Lemma 4.5 ∎.

From the arc consistency closure of $dual(P))$, *i.e.*, $ac(dual(P))$, we can construct a subdomain for the original problem P, denoted by $\mathcal{D}^{dualac(P)}$, in which for each ordinary variable x in P, we choose a dual variable c such that $x \in vars(c)$[2], and set $dom^{\mathcal{D}^{dualac(P)}}(x)$ to be $\pi_{\{x\}} dom^{ac(dual(P))}(c)$. From Lemma 4.5, for any two dual variables c_i and c_j such that $x \in c_i$ and $x \in c_j$, $\pi_{\{x\}} dom^{ac(dual(P))}(c_i) = \pi_{\{x\}} dom^{ac(dual(P))}(c_j)$

[2]This is always possible because we have assumed that each variable should be constrained by at least one constraint in a CSP.

Table 4.1: Comparison of the worst case complexity of AC3 on the original problem, the dual problem, and the hidden problem.

AC3	worst case complexity
original	$O(mr^2d^{r+1})$.
hidden	$O(mrd^{2r+1})$ or $O(mrdM^2)$.
dual	$O(m^2d^{3r})$ (in the general case). $O(m^2d^{2r})$ or $O(m^2M^2)$ (if a propagator is used).

because $ac(dual(P))$ is arc consistent [3]. So the domain of x in $\mathcal{D}^{dualac(P)}$ is irrelevant to whichever dual variable we choose to make the projection.

Note that $\mathcal{D}^{dualac(P)}$ is a set of domains for the ordinary variables in P, and it is constructed from the set of domains for the dual variables in $ac(dual(P))$. For example, the dual problem of the CSP in Example 4.5 is arc consistent and let dual variables c_1, c_2 and c_3 correspond to three original constraints, thus $\mathcal{D}^{ac(dual(P))}$ is $\{dom(c_1) = \{(1,1)\}, dom(c_2) = \{(1,1)\}, dom(c_3) = \{(1,1)\}\}$, and $\mathcal{D}^{dualac(P)}$ is $\{dom(x_1) = \{1\}, dom(x_2) = \{1\}, dom(x_3) = \{1\}\}$.

Theorem 4.7 *If $ac(dual(P))$ is not empty, $ac(P)$ is not empty either.*

Proof: Because the domain of each dual variable in $ac(dual(P))$ is not empty, its projection over an ordinary variable cannot be empty either. So there is no empty domain in $\mathcal{D}^{dualac(P)}$. In the original problem, for each ordinary variable x, for each of the values $a \in dom^{\mathcal{D}^{dualac(P)}}(x)$, and for each constraint C, where $x \in vars(C)$, suppose a is the projection of the tuple t of the corresponding dual variable c, then for each of the variables $y \in vars(C)$, $t[y] \in dom^{\mathcal{D}^{dualac(P)}}(y)$. Thus t is a valid support for $\{x \leftarrow a\}$ in constraint C. Therefore, $\mathcal{D}^{dualac(P)}$ is a non-empty arc consistent subdomain of P and thus $ac(P)$ is not empty. ∎

From Theorem 4.3 and Theorem 4.7, we have the following comparison between arc consistency on the dual problem and the one on the hidden problem.

Corollary 4.8 *If $ac(dual(P))$ is not empty, $ac(hidden(P))$ is not empty either.*

[3] In a strict sense, Lemma 4.5 only applies to the dual transformation. However, $ac(dual(P))$ can be viewed as the dual transformation of a CSP, which has the same variables and domains as the original problem P, and in which the constraints in the original problem are tightened according to the domains of the corresponding dual variables in $ac(dual(P))$.

In the dual transformation, there are m dual variables. The maximum domain size is bounded by d^r. The number of the dual constraints is bounded by m^2, *i.e.*, each pair of dual variables constrain each other. Thus the worst case complexity of achieving arc consistency on the dual problem by AC3 is $O(m^2 d^{3r})$. In the worst case, among arc consistency on the original problem, its hidden transformation and its dual transformation, the one on the dual is the most powerful and yet the most expensive. Constraint propagation in the dual problem can be speeded up by exploiting the special structure of the dual constraints. For example, given a dual constraint between two dual variables, c_i and c_j, each time we want to find supports for the values in the domain of c_i, we allocate a table of size $d^{|vars(c_i) \cap vars(c_j)|}$, and for each of the tuples $t_j \in dom^{dual(P)}(c_j)$, we record the projection $t_j[vars(c_i) \cap vars(c_j)]$ in the table. Then we go through the domain of c_i and for each tuple $t_i \in dom^{dual(P)}(c_i)$, we check whether the projection $t_i[vars(c_i) \cap vars(c_j)]$ has been recorded in the table. If not, t_i is removed from the domain of c_i. By two passes of the domains, we can revise the dual constraint. Therefore, the worst case complexity of AC3 on the dual problem is lowered to $O(m^2 d^{2r})$, which is competitive to the one on the hidden problem. Achieving arc consistency on the dual problem is worth doing only in the case that the original constraints are very tight. Let M denote the maximum domain size of the dual variables, the worst case complexity of AC3 on the dual problem can be rewritten as $O(m^2 M^2)$. For example, in Chapter 3, we have used the dual representation of the crossword puzzle problems to compare GAC and GAC-CBJ, in which arc consistency is enforced (on the induced problem) at each node in the backtrack search tree. Table 4.1 summarizes the results about the worst case complexity of AC3 on the original problem, the dual problem, and the hidden problem.

We have shown that in general, arc consistency on the dual problem is stronger than arc consistency on the original problem and the hidden problem. However, for some CSPs with a special structure, achieving arc consistency on the dual is equivalent to achieving arc consistency on the original problem.

Theorem 4.9 *Given a CSP P, if for any two constraints C_i and C_j of P, $vars(C_i) \cap vars(C_j)$ contains at most one ordinary variable, (1) $ac(dual(P))$ is not empty if and only if $ac(P)$ is not empty; (2) $\mathcal{D}^{dualac(P)} = \mathcal{D}^{ac(P)}$; and (3) $dual(ac(P)) = ac(dual(P))$.*

Proof: (1) From Theorem 4.7, if $ac(dual(P))$ is not empty, $ac(P)$ is not empty. On the converse, suppose $ac(P)$ is not empty, we have the dual transformation of $ac(P)$ as $dual(ac(P))$. Note that $\mathcal{D}^{dual(ac(P))}$, the set of the domains in $dual(ac(P))$, is a

subdomain of $dual(P)$ and $\mathcal{D}^{dual(ac(P))}$ is not empty. Given any two dual variables c_i and c_j in $dual(P)$, where $vars(c_i) \cap vars(c_j) \neq \emptyset$, let x be the only ordinary variable in $vars(c_i) \cap vars(c_j)$. We will prove that $\pi_{\{x\}} dom^{dual(ac(P))}(c_i) = \pi_{\{x\}} dom^{dual(ac(P))}(c_j)$ $= dom^{ac(P)}(x)$. From the definition of the dual transformation, because each tuple in the domain of a dual variable must be a valid tuple in the corresponding original constraint, we have $\pi_{\{x\}} dom^{dual(ac(P))}(c_i) \subseteq dom^{ac(P)}(x)$. Because each value $a \in dom^{ac(P)}(x)$ has at least one valid support in the constraint C_i in $ac(P)$, we have $dom^{ac(P)}(x) \subseteq \pi_{\{x\}} dom^{dual(ac(P))}(c_i)$. Therefore, $\pi_{\{x\}} dom^{dual(ac(P))}(c_i) = dom^{ac(P)}(x)$. For the same reason, $\pi_{\{x\}} dom^{dual(ac(P))}(c_j) = dom^{ac(P)}(x)$. Thus, $\pi_{\{x\}} dom^{dual(ac(P))}(c_i)$ $= \pi_{\{x\}} dom^{dual(ac(P))}(c_j)$. From Lemma 4.5, $\mathcal{D}^{dual(ac(P))}$ is an arc consistent subdomain of $dual(P)$. Therefore, $\mathcal{D}^{dual(ac(P))} \subseteq \mathcal{D}^{ac(dual(P))}$ and $ac(dual(P))$ is not empty.
(2) From the construction of $\mathcal{D}^{dualac(P)}$, $dom^{\mathcal{D}^{dualac(P)}}(x)$, the domain of an ordinary variable x in $\mathcal{D}^{dualac(P)}$, is set to be $\pi_{\{x\}} dom^{ac(dual(P))}(c)$ for some dual variable c. Because $dom^{ac(P)}(x) = \pi_{\{x\}} dom^{dual(ac(P))}(c)$ and $\mathcal{D}^{dual(ac(P))}$ is a subdomain of $ac(dual(P))$, that means, $dom^{ac(P)}(x) \subseteq \pi_{\{x\}} dom^{ac(dual(P))}(c)$, and thus, $dom^{ac(P)}(x) \subseteq dom^{\mathcal{D}^{dualac(P)}}(x)$. Therefore $\mathcal{D}^{ac(P)} = \mathcal{D}^{dualac(P)}$.
(3) Because $\mathcal{D}^{dualac(P)}$ is an arc consistent subdomain of $ac(P)$, for any dual variable c in $ac(dual(P))$, and for each of the tuples $t \in dom^{ac(dual(P))}(c)$, $t[x] \in dom^{ac(P)}(x)$ for each of the ordinary variables $x \in vars(c)$. Thus t is not removed from the corresponding original constraint C in $ac(P)$. Therefore, $dom^{\mathcal{D}^{ac(dual(P))}}(c) \subseteq dom^{\mathcal{D}^{dual(ac(D))}}(c)$. We have $\mathcal{D}^{ac(dual(P))} = \mathcal{D}^{dual(ac(D))}$. Because $ac(dual(P))$ and $dual(ac(P))$ have the same set of variables, the same set of domains and the same set of dual constraints, $ac(dual(P)) = dual(ac(P))$. ∎

In a crossword puzzle problem, there is no overlap between two horizontal unknown words or two vertical unknown words, and a horizontal word overlaps with a vertical word on at most one letter. Thus, in its original formulation, every two non-binary constraints overlap on at most one ordinary variable. From the above theorem, we know that achieving arc consistency on the dual representation is equivalent to achieving arc consistency on the original representation. Given a CSP, if two original constraints overlap on more than one variable, arc consistency on the dual problem may be strictly stronger than arc consistency on the original problem. An example is the problem in Example 4.4.

For a binary CSP, we assume that there is at most one binary constraint between two variables. Thus, if the original problem is a binary CSP, each pair of the original constraints overlap on at most one ordinary variable. From Theorem 4.9, arc

consistency on its dual is equivalent to arc consistency on the original.

Corollary 4.10 *Given a binary CSP P, (1) $ac(P)$ is not empty if and only if $ac(dual(P))$ is not empty; (2) $\mathcal{D}^{ac(P)} = \mathcal{D}^{dualac(P)}$; and (3) $dual(ac(P)) = ac(dual(P))$.*

Because the dual representation is a binary CSP, Corollary 4.10 prevents the attempt to take the dual transformation of a dual problem in order to achieve a higher level consistency by enforcing arc consistency on the "double-dual" transformation.

4.4 Consistencies Hierarchy

Because both the dual problem and the hidden problem are binary CSPs. The kinds of consistency that only apply to binary CSPs can be used and compared on the dual problem and the hidden problem. Debruyne and Bessière have studied and compared some selected local consistencies on binary CSPs in [28]. Following their definitions, a binary CSP is (i,j)-consistent *iff* it is not empty and any consistent partial solution over i variables can be extended to a consistent partial solution involving j additional variables. A problem is arc consistent (AC) if it is (1,1)-consistent. A problem is path consistent (PC) if it is (2,1)-consistent. A problem is strongly path consistent (ACPC) if it is (i,1)-consistent for each $1 \leq i \leq 2$. A problem is path inverse consistent (PIC) if it is (1,2)-consistent. A problem is neighborhood inverse consistent (NIC) *iff* any instantiation of a single variable x can be extended to a consistent partial solution over all the variables that are constrained with x, called the *neighborhood* of x. A problem is restricted path consistent (RPC) *iff* it is arc consistent and if an instantiation of a variable is consistent with just a single value of an adjoining variable, then for any other variable there exists a value compatible with these instantiations. A problem is singleton arc consistent (SAC) *iff* it is not empty, and the CSP induced by any instantiation of a single variable is not empty after achieving arc consistency.

Debruyne and Bessière compare these consistencies in a way similar to our approach in the above sections. They call a consistency property $\mathcal{LC}_1$ is stronger than $\mathcal{LC}_2$ ($\mathcal{LC}_1 \geq \mathcal{LC}_2$) *iff* in any problem in which $\mathcal{LC}_1$ holds, then $\mathcal{LC}_2$ holds, and $\mathcal{LC}_1$ is strictly stronger than $\mathcal{LC}_2$ ($\mathcal{LC}_1 > \mathcal{LC}_2$) if $\mathcal{LC}_1$ is stronger than $\mathcal{LC}_2$ and there is at least one instance such that $\mathcal{LC}_2$ holds but $\mathcal{LC}_1$ does not. They have shown that, ACPC > SAC > PIC > RPC > AC, and NIC > PIC. Note that our definition of the "strongness" is slightly different than theirs. We mean $\mathcal{LC}_1$ on formulation $\mathcal{A}$ is stronger than $\mathcal{LC}_2$ on formulation $\mathcal{B}$ if for any problem, if $\mathcal{LC}_1$ can be achieved on

$\mathcal{A}$ without an empty resulting problem, $\mathcal{LC}_2$ can also be achieved on $\mathcal{B}$ without an empty resulting problem [4]. Nevertheless, we can justify that the above hierarchy still holds under our definition. Suppose in the above hierarchy, a local consistency property $\mathcal{LC}_1$ is stronger than a local consistency property $\mathcal{LC}_2$. Given any CSP instance P, if $\mathcal{LC}_1$ can be achieved without an empty resulting problem, there is a non-empty subproblem of P in which $\mathcal{LC}_1$ holds and thus $\mathcal{LC}_2$ also holds. Therefore $\mathcal{LC}_2$ can be achieved on P without an empty resulting problem. As we can see, arc consistency lies at the bottom in the above hierarchy. Furthermore, it has been observed by Stergiou and Walsh [106] that, due to the special topology of the hidden transformation, certain consistency techniques fail to achieve any additional pruning than AC.

Theorem 4.11 *[106] Given a CSP instance P, $hidden(P)$ is not empty after enforcing NIC if and only if it is not empty after enforcing AC.*

Proof: Since NIC is stronger than AC, we only need to consider the if part in the above. Suppose $ac(hidden(P))$ is not empty. For a hidden variable c, its neighborhood is $vars(c)$ in $ac(hidden(P))$. Thus an instantiation of c with a tuple t from its domain in $ac(hidden(P))$ can be extended to a consistent partial solution including its neighborhood, where for each of the ordinary variables $x \in vars(c)$, x is instantiated with $t[x]$ ($t[x]$ must be in the domain of x because it is the only support for t in the hidden constraint between x and c). For an ordinary variable x, x only constrains with hidden variables. An instantiation of x with a value a from its domain in $ac(hidden(P))$ can be extended to a consistent partial solution including all its neighborhood, where for each of the hidden variables c in its neighborhood, c is instantiated with a tuple t such that $t[x] = a$ (also, such a tuple must exist because $\{x \leftarrow a\}$ has at least one support in the hidden constraint between x and c). Therefore, the hidden problem is not empty after enforcing NIC. ∎

Because on the hidden problem NIC collapses down onto AC, those consistencies that are weaker than NIC but stronger than AC, *e.g.*, PIC and RPC, are also equivalent to AC. However, for the dual problem, NIC is still strictly stronger than AC.

Example 4.6 *Consider a binary CSP with* 3 *Boolean variables. The constraints are*

$$C(x_1, x_2) \quad = \quad \{(0,0),(1,1)\},$$

[4] Because in the comparison of arc consistency on the original problem and arc consistency on the dual problem, Debruyne and Bessière's definition of "strongness" cannot be applied. (See Example 4.4 and Example 4.5.)

$$C(x_2, x_3) = \{(0,0),(1,1)\},$$
$$C(x_1, x_3) = \{(0,1),(1,0)\}.$$

The original problem is AC but not NIC. Also, its dual transformation is AC but not NIC. The dual transformation is neither PIC nor RPC.

Although on the hidden problem NIC and PIC do not provide any more pruning than AC, Stergiou and Walsh have shown by example that on the hidden problem, ACPC is strictly stronger than SAC, which itself is strictly stronger than AC [106].

Theorem 4.12 *SAC on the dual problem is stronger than SAC on the hidden problem.*

Proof: If the dual problem is not empty after enforcing SAC, let $sac(dual(P))$ denote the resulting CSP. We will show that the hidden problem is not empty either after enforcing SAC. From $sac(dual(P))$, we can construct a subdomain $\mathcal{D}^{dualsac(P)}$ for the hidden problem, in which each hidden variable has the same domain as the corresponding dual variable in $sac(dual(P))$, and the domain of an ordinary variable x is set to be $\pi_{\{x\}} dom^{sac(dual(P))}(c)$ for some dual variable c such that $x \in vars(c)$. Because $sac(dual(P))$ is arc consistent, from Lemma 4.5, the domain of x is irrelevant to whichever dual variable we choose to make the projection. For each hidden variable c and for each of the tuples $t \in dom^{\mathcal{D}^{dualsac(P)}}(c)$, $hidden(P)|_{\{c \leftarrow t\}}$ is arc consistent if and only if $P|_t$ is arc consistent. $P|_t$ is arc consistent because $dual(P)|_{\{c \leftarrow t\}}$ is arc consistent. For each ordinary variable x in $hidden(P)$ and for each of the values $a \in dom^{\mathcal{D}^{dualsac(P)}}(x)$, there is a hidden variable c and a tuple t of c such that $x \in vars(c)$ and $t[x] = a$. Thus $hidden(P)|_{\{x \leftarrow a\}}$ is arc consistent because $hidden(P)|_{\{c \leftarrow t\}}$ is arc consistent. ∎

In the hidden problem, for each pair of hidden variables c_i and c_j, where $vars(c_i) \cap vars(c_j) \neq \emptyset$, enforcing strong path consistency will add a constraint between c_i and c_j, which restricts a tuple from c_i to agree with a tuple from c_j on the shared ordinary variables. The constraint is essentially the same as the dual constraint between c_i and c_j in the dual transformation. Thus, enforcing strong path consistency on the hidden problem actually results in a subproblem, which is identical to the dual problem. Therefore, strong path consistency on the hidden problem is at least as strong as the one on the dual.

Theorem 4.13 *Achieving strong path consistency on the hidden problem is equivalent to achieving strong path consistency on the dual problem.*

Proof: If the dual problem is not empty after enforcing strong path consistency, we will show that the hidden problem is not empty either after enforcing strong path consistency. Let $pc(dual(P))$ denote the resulting binary CSP after enforcing strong path consistency on the dual problem, and for each pair of dual variables c_i and c_j, if there is one constraint between c_i and c_j in $pc(dual(P))$, let $DC(c_i, c_j)$ denote that constraint, otherwise, $DC(c_i, c_j)$ denotes the universal constraint between c_i and c_j. Suppose $pc(dual(P))$ is not empty, we can make the hidden problem strongly path consistent without an empty resulting CSP. In the hidden problem, we do the following operations.

- For each hidden variable c, its domain is restricted to be the domain of the dual variable c in $pc(dual(P))$. For each ordinary variable x, we choose a dual variable c in $pc(dual(P))$ such that $x \in vars(c)$, and restrict the domain of x to be $\pi_{\{x\}} dom^{pc(dual(P))}(c)$. Because $pc(dual(P))$ is arc consistent, from Lemma 4.5, the domain of x is irrelevant to whichever dual variable we choose to make the projection.

- Given two hidden variables c_i and c_j, we add a new constraint $HC(c_i, c_j)$ between c_i and c_j, which is the same as the constraint $DC(c_i, c_j)$ in $pc(dual(P))$.

- Given an ordinary variable x and a hidden variable c. If x is included in $vars(c)$, there is an original hidden constraint $HC(x, c)$ between x and c and we tighten this constraint according to the new domains. Otherwise x is not included in $vars(c)$. We choose another dual variable c' such that $x \in vars(c) \cup vars(c')$ (this is possible because x involves at least one constraint). Then we add a new constraint $HC(x, c)$ between x and c in the hidden problem, which specifies a value a of x to be compatible with a tuple t of c if there is a tuple t' in the domain of c', such that $\{c \leftarrow t, c' \leftarrow t'\}$ satisfies the constraint $DC(c, c')$ and $t'[x] = a$. The new constraint is irrelevant to whichever dual variable we choose. Suppose there is a dual variable c'' such that $x \in vars(c) \cup vars(c'')$. If $\{c \leftarrow t, c' \leftarrow t'\}$ satisfies $D(c, c')$, because $pc(dual(P))$ is strongly path consistent, there is a tuple t'' of c'', such that $\{c' \leftarrow t', c'' \leftarrow t''\}$ satisfies $DC(c', c'')$ and $\{c \leftarrow t, c'' \leftarrow t''\}$ satisfies $DC(c, c'')$. Note that $x \in vars(c') \cap vars(c'')$, so $t'[x] = t''[x] = a$. Thus a is also compatible with t in the case that c'' is considered.

- Given two ordinary variables x and y, we add a new constraint $HC(x, y)$ between x and y in the hidden problem. If there is a dual variable c such that $x \in vars(c)$ and $y \in vars(c)$, $HC(x, y)$ is set to be $\pi_{\{x,y\}} dom^{pc(dual(P))}(c)$. The

new constraint is irrelevant to whichever dual variable we choose to make the projection, because from Lemma 4.5, the domains of two dual variables will make the same projection over $\{x, y\}$. If there is no such a dual variable c that x and y are contained in the scheme of c simultaneously, we choose two dual variables, c_i and c_j, such that $x \in vars(c_i)$ and $y \in vars(c_j)$. $HC(x, y)$ specifies that value a of x is compatible with value b of y if there is a tuple t_i of c_i and a tuple t_j of c_j, such that $\{c_i \leftarrow t_i, c_j \leftarrow t_j\}$ satisfies $D(c_i, c_j)$, while $t_i[x] = a$ and $t_j[y] = b$. Also, the new constraint is irrelevant to whichever dual variables we choose. Suppose there are two dual variables c'_i and c'_j such that $x \in vars(c'_i)$ and $y \in vars(c'_j)$. Because $pc(dual(P))$ is strongly path consistent, there is a tuple t'_i of c'_i, such that $\{c_i \leftarrow t_i, c'_i \leftarrow t'_i\}$ satisfies the constraint $DC(c_i, c'_i)$ and $\{c'_i \leftarrow t'_i, c_j \leftarrow t_j\}$ satisfies the constraint $DC(c'_i, c_j)$. Note that $x \in vars(c_i) \cap vars(c'_i)$, so $t'_i[x] = t_i[x] = a$. Furthermore, there is a tuple t'_j of c'_j such that $\{c'_i \leftarrow t'_i, c'_j \leftarrow t'_j\}$ satisfies the constraint $DC(c'_i, c'_j)$ and $\{c_j \leftarrow t_j, c'_j \leftarrow t'_j\}$ satisfies the constraint $DC(c_j, c'_j)$. Because $y \in vars(c_j) \cap vars(c'_j)$, we have $t'_j[y] = t_j[y] = b$. Thus a is also compatible with b in the case c'_i and c'_j are considered.

Let $pc(hidden(P))$ denote the resulting CSP. $pc(hidden(P))$ is not empty because none of the above projections is empty. Then we need to verify that $pc(hidden(P))$ is strongly path consistent. To verify that it is arc consistent, we need to consider three possibilities: a constraint between two hidden variables, a constraint between two ordinary variables and a constraint between an ordinary variable and a hidden variable. The case of two hidden variables can be released because the domains of the hidden variables and the constraints between the hidden variables are the same as those in $pc(dual(P))$. To save space, we only discuss the case of two ordinary variables here. Given two ordinary variables x and y. If the constraint $HC(x, y)$ was constructed from a single dual variable c, *i.e.*, $x \in vars(c)$ and $y \in vars(c)$, note that $dom^{pc(hidden(P))}(x) = \pi_{\{x\}} dom^{pc(dual(P))}(c)$ and $dom^{pc(hidden(P))}(y) = \pi_{\{y\}} dom^{pc(dual(P))}(c)$, then for each value a of x, there is a tuple t in $dom^{pc(dual(P))}(c)$ such that $t[x] = a$. Thus the value $t[y]$ in the domain of y is compatible with a in $HC(x, y)$. If $HC(x, y)$ was constructed from two dual variables c_i and c_j. For each value a of x, there is a tuple t_i of c_i such that $t_i[x] = a$. Because $pc(dual(P))$ is arc consistent, there is a tuple t_j of c_j such that $\{c_i \leftarrow t_i, c_j \leftarrow t_j\}$ is consistent in $DC(c_i, c_j)$. Thus a is compatible with value $t_j[y]$ in the domain of y. In either case, each value in the domain of x can find a support in the domain of y. Thus, $HC(x, y)$

is arc consistent. Therefore, $pc(hidden(P))$ is arc consistent.

To verify $pc(hidden(P))$ is path consistent, we need to consider six possibilities, given any ordinary variables x, y and z, and hidden variables c_i, c_j and c_k, a consistent partial solution over x and y to be extended to a consistent partial solution over x, y and z, denoted by (1) $(x, y) \rightarrow z$; (2) $(x, y) \rightarrow c_i$; (3) $(x, c_i) \rightarrow y$; (4) $(x, c_i) \rightarrow c_j$; (5) $(c_i, c_j) \rightarrow x$; and (6) $(c_i, c_j) \rightarrow c_k$. The case of $(c_i, c_i) \rightarrow c_k$ can be released because $pc(dual(P))$ is path consistent. To save space, we only discuss the case of $(x, c_i) \rightarrow c_j$ here. Suppose $\{x \leftarrow a, c_i \leftarrow t_i\}$ satisfies the constraint $HC(x, c_i)$. If $x \in vars(c_i)$ and $x \in vars(c_j)$, because $pc(dual(P))$ is arc consistent, there is a tuple t_j of c_j such that $\{c_i \leftarrow t_i, c_j \leftarrow t_j\}$ satisfies the constraint $HC(c_i, c_j)$. Also, we have $t_i[x] = a$ and $t_j[x] = a$. That is, $\{x \leftarrow a, c_j \leftarrow t_j\}$ satisfies the constraint $HC(x, c_j)$. If $x \in vars(c_i)$ but $x \notin vars(c_j)$, then $t_i[x] = a$. The constraint $HC(x, c_j)$ could be constructed from $DC(c_j, c_i)$. Because there is a tuple t_j of c_j to be compatible with t_i of c_j in the constraint $HC(t_i, t_j)$, $\{x \leftarrow a, c_j \leftarrow t_j\}$ satisfies the constraint $HC(x, t_j)$. If $x \in vars(c_j)$ but $x \notin vars(c_i)$, the constraint $HC(x, c_i)$ could be constructed from $DC(c_i, c_j)$. Because $\{x \leftarrow a, c_i \leftarrow t_i\}$ satisfies $HC(x, c_i)$, there is a tuple t_j of c_j such that $\{c_i \leftarrow t_i, c_j \leftarrow t_j\}$ satisfies $HC(c_i.c_j)$ and $t_j[x] = a$. Thus $\{x \leftarrow a, c_j \leftarrow t_j\}$ satisfies the constraint $HC(x, c_j)$. If $x \notin vars(c_i)$ and $x \notin vars(c_j)$, there is a dual variable c such that $HC(x, c_i)$ was constructed from $DC(c_i, c)$, and $HC(x, c_j)$ was constructed from $DC(c_j, c)$. Thus, there is a tuple t of c, such that $\{c \leftarrow t, c_i \leftarrow t_i\}$ satisfies the constraint $HC(c, c_i)$ and $t[x] = a$. Because $pc(dual(P))$ is path consistent, there is a tuple t_j of c_j, such that $\{c \leftarrow t, c_j \leftarrow t_j\}$ satisfies $HC(c, c_j)$ and $\{c_i \leftarrow t_i, c_j \leftarrow t_j\}$ satisfies $HC(c_i, c_j)$. We have $\{x \leftarrow a, c_j \leftarrow t_j\}$ satisfies the constraint $HC(x, c_j)$. Therefore, the case of $(x, c_i) \rightarrow c_j$ is verified after we have considered all the possibilities. ∎

We summarize the above results in a hierarchy graph, as shown in Figure 4.6. In the above figure, an appendix to a consistency denotes on which problem the consistency is applied. For example, NIC-dual denotes the case of NIC on the dual transformation. If there is a path between consistent properties $\mathcal{A}$ and $\mathcal{B}$ in the figure, it means $\mathcal{A}$ is stronger than $\mathcal{B}$. If there is also a path from $\mathcal{B}$ to $\mathcal{A}$, then $\mathcal{A}$ is equivalent to $\mathcal{B}$.

4.5 Summary

In this chapter, we formally defined the dual transformation and the hidden variable transformation of an original CSP. We studied the arc consistency property on the

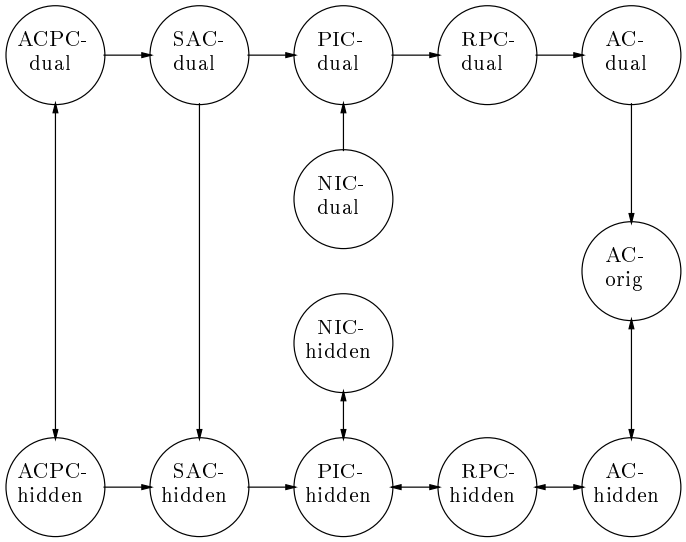


Figure 4.6: A hierarchy about the relations between consistencies on the original problem, its dual transformation and its hidden transformation.

original problem, and its dual and hidden transformations. We show that arc consistency on the dual problem is stronger than arc consistency on the original problem, which itself is equivalent to arc consistency on the hidden problem. We identified a special structure of non-binary CSPs, in which arc consistency on the dual is equivalent to one on the original problem. We enriched the consistencies hierarchy by considering some other local consistencies that only apply to binary CSPs. For example, we show that path consistency on the hidden problem is equivalent to one on the dual, and singleton arc consistency on the dual is stronger than one on the hidden.

Chapter 5

Dual and Hidden Transformations with Backtracking Algorithms

In Chapter 4, we formally defined the dual and hidden transformations and gave some theoretical comparisons of the consistency properties on the original problem and its dual and hidden transformations. One may be more interested in how these transformations affect the problem solving. That is, given three formulations of a problem, the original formulation, its dual transformation and hidden transformation, how will they affect the backtracking algorithms when solving the problem. Bacchus and van Beek have presented some preliminary results based on the forward checking algorithm(FC)[7]. For example, they give examples to show that FC on the original may be exponentially better or worse than FC on the dual problem or hidden problem. Also, they present a new algorithm, known as *FC+*, as an enhancement to FC on the hidden problem, which is shown to be the best among those "algorithm+formulation" couples. We will present in this chapter more theoretical comparisons about the performance of the above three formulations in selected backtracking algorithms, including the chronological backtracking algorithm, the forward checking algorithm, and the maintaining arc consistency algorithm.

5.1 A Few Issues about the Comparisons

Given a CSP formulation of a problem, we can always transform the original formulation into the dual problem or the hidden problem. Note that our purpose is to theoretically evaluate the modeling techniques (the dual and hidden transformation) by means of comparing the performance of the selected backtracking algorithms on the above formulations, which is different from the approach of evaluating (all possible)

CSP formulations for a specific problem as Nadel [86] did for the n-Queens problem. A general methodology in algorithms evaluation is to compare their performance empirically or theoretically on the same CSP instance, *i.e.*, to fix the problem formulation and evaluate the algorithm. Intuitively, the formulations evaluation should work in the reverse direction. That is, to fix the algorithm and compare the performance of the algorithm on each of the formulations. In practice, this approach is reasonable because there are only a few best algorithms that are widely studied in the literature and used in commercial systems, such as Ilog Solver [61], *e.g.*, the forward checking algorithm and the maintaining arc consistency algorithm. Hence, in this chapter, we will focus on three main-stream backtracking algorithms, the chronological backtracking algorithm, the forward checking algorithm and the maintaining arc consistency algorithm. However, given a backtracking algorithm and a CSP formulation, we are still unable to precisely describe the execution of the algorithm without knowing the instantiation order of the variables. As a matter of fact, different variable orderings may result in tremendous differences in the performance of the algorithm.

Example 5.1 *Consider a CSP over the set of Boolean variables,* $\{x_1, \ldots, x_n\}$. *The constraints are*

$$\begin{aligned} C(x_1, x_2, x_n) &= \{(0,0,1),(1,1,0)\}, \\ C(x_1, x_n) &= \{(0,0),(1,1)\}, \\ &\ldots \\ C(x_{n-1}, x_n) &= \{(0,0),(1,1)\}. \end{aligned}$$

Under the static variable ordering $x_1, \ldots, x_n$, *FC applied to the original problem is able to detect that every node at the level of* x_2 *is a dead-end, because the domain of* x_n *will be wiped out due to the instantiations to* x_1 *and* x_2. *In the hidden problem, if the variables are instantiated in the order,* $x_1, \ldots, x_n, c(x_1, x_2, x_n), c(x_1, x_n), \ldots, c(x_{n-1}, x_n)$. *FC applied on the hidden problem is unable to detect a dead-end until the variable* x_n *has been instantiated. Thus, under the above variable orderings, FC applied on the hidden is exponentially worse than FC on the original problem. However, if the variables in the hidden problem are instantiated in the order,* $x_1, x_2, c(x_1, x_2, x_n), c(x_1, x_n), \ldots$, *FC applied on the hidden is able to detect that every node at the level of* $c(x_1, x_n)$ *is inconsistent with* x_n. *Therefore, under the new variable ordering, the performance of FC on the hidden problem is comparable to the one on the original.*

The issue about variable ordering may not be a serious problem when we are comparing two algorithms on the same problem formulation, because the same static

variable ordering or the same dynamic variable heuristic (if applicable) can be used for both algorithms [69]. However, because all three formulations that we are going to evaluate have different sets of variables, inherently, we have to use a different variable ordering for each formulation. The question is, how can we select the variable orderings to ensure a fair and meaningful comparison? One simple solution is to assign each formulation a default static variable ordering. For example, the default variable ordering for the original problem is to instantiate the variables in a static order in which they are presented in the problem, usually in the lexicographic order as, $x_1, \ldots, x_n$. The default variable ordering for the dual problem is to instantiate the dual variables in the order in which their corresponding constraints are presented in the original problem, *e.g.*, $c_1, \ldots, c_m$. Such an arrangement is meaningful, because in the dual transformation, the first dual variable is usually given to the first original constraint and so on. If both the original problem and the dual problem are solved in a static variable ordering, it is very natural to instantiate the variables in the order in which they are presented in the formulation. Similarly, we can set the default variable ordering for the hidden problem as $x_1, \ldots, x_n, c_1, \ldots, c_m$. By assigning each problem formulation a default variable ordering, we can show the worst case differences between the performance of the formulations. For instance, in the above example, under the default variable orderings, FC applied on the hidden is exponentially worse than FC applied on the original problem.

Generally, given a backtracking algorithm, it is possible to find one instance to show that one formulation may be exponentially better than the other, and meanwhile there is an instance in which the converse holds. Unlike the results of the algorithms evaluation (for example, it is known that CBJ is never worse than BT in terms of nodes visited and constraint checks performed) there is usually no constant relation between two formulations under the same algorithm, saying that one is always better than the other. Although the above bounds are valid in the worst case or in the instances we constructed, they are not very interesting because too little information is provided to guide us to determine whether or not the dual or hidden transformation should be applied on the original problem. The problem here is that the restriction of the algorithm to the default variable orderings gives us too much freedom in contriving a CSP instance that maximally shows off the drawbacks of one formulation but relatively hides those from the other formulation.

In this chapter, given two formulations $\mathcal{A}$ and $\mathcal{B}$ of a problem, we are going to identify one of two relations between $\mathcal{A}$ and $\mathcal{B}$ with respect to a fixed backtracking algorithm; namely, $\mathcal{A}$ may be exponentially worse than $\mathcal{B}$, or $\mathcal{A}$ is only bounded worse

than $\mathcal{B}$.

Definition 5.1 *Given a backtracking algorithm and two formulations $\mathcal{A}$ and $\mathcal{B}$ of a problem, we say $\mathcal{A}$ may be exponentially worse than $\mathcal{B}$ if there exists a CSP instance and a variable ordering for $\mathcal{B}$ such that the performance of the algorithm applied on $\mathcal{A}$ is exponentially worse than its performance on $\mathcal{B}$ no matter what variable ordering is used in $\mathcal{A}$, and we say $\mathcal{A}$ is only bounded worse than $\mathcal{B}$ if for any CSP instance and any variable ordering for $\mathcal{B}$, there is a variable ordering for $\mathcal{A}$ such that the performance of the algorithm applied on $\mathcal{A}$ is bounded by a polynomial from its performance on $\mathcal{B}$.*

Note the above two relations are mutually exclusive. To be fair and meaningful, we have to restrict the variable orderings that can be constructed and used for $\mathcal{A}$ when proving that $\mathcal{A}$ is only bounded worse than $\mathcal{B}$. As we have learned from Theorem 3.5 about BT and CBJ, BT can always perform as well as CBJ if an appropriate variable ordering is used for BT to simulate the execution of CBJ, whereas CBJ is known to be never worse than BT. Note that the variable ordering constructed for BT entirely depends on the execution of the CBJ to solve the problem, *i.e.*, it cannot be known before the completion of the CBJ. Generally, a comparison conducted in such a way between two algorithms or two formulations is not fair and it does not reflect their actual performance in solving the problem. In the following, to prove the bounded worse relation between $\mathcal{A}$ and $\mathcal{B}$, we will use static variable orderings for both of them. That is, given any CSP instance and a static variable ordering for $\mathcal{B}$, we can always construct a static variable ordering for $\mathcal{A}$ from the the one of $\mathcal{B}$, such that the performance of $\mathcal{A}$ is bounded by a polynomial factor from the performance of $\mathcal{B}$. Under certain circumstances, we can relax the static variable orderings to dynamic ones. For example, since the hidden problem has all the information (domains, degrees, *etc.*) in the original problem, given a dynamic variable ordering heuristic for the original problem, *e.g.*, FF+Deg, it is possible to construct a dynamic variable ordering for the hidden problem without knowing the execution of the algorithm on the original problem.

When proving the above relations between formulations $\mathcal{A}$ and $\mathcal{B}$, we actually use the number of the nodes visited by the backtracking algorithm as the measure of its performance. Because the backtracking algorithm only performs a polynomial number of constraint checks at each node in the search tree, these relations are also valid if the number of the constraint checks performed by the algorithm is considered.

The last issue is which of the parameters related to the size of a CSP formulation should be used to establish an exponential or a polynomial bound. Those parameters include the number of the variables, n, the maximum domain size, d, the maximum arity of the constraints, r, the number of constraints, m, and the maximum number of the tuples in the constraints, M[1]. We assume that the parameters d, r, and m are all bounded by a polynomial with respect to the number of variables n and M is bounded by $O(d^r)$ unless otherwise stated. Therefore, the exponential or polynomial bound can be expressed in terms of n, or a combination of n, d, r and m.

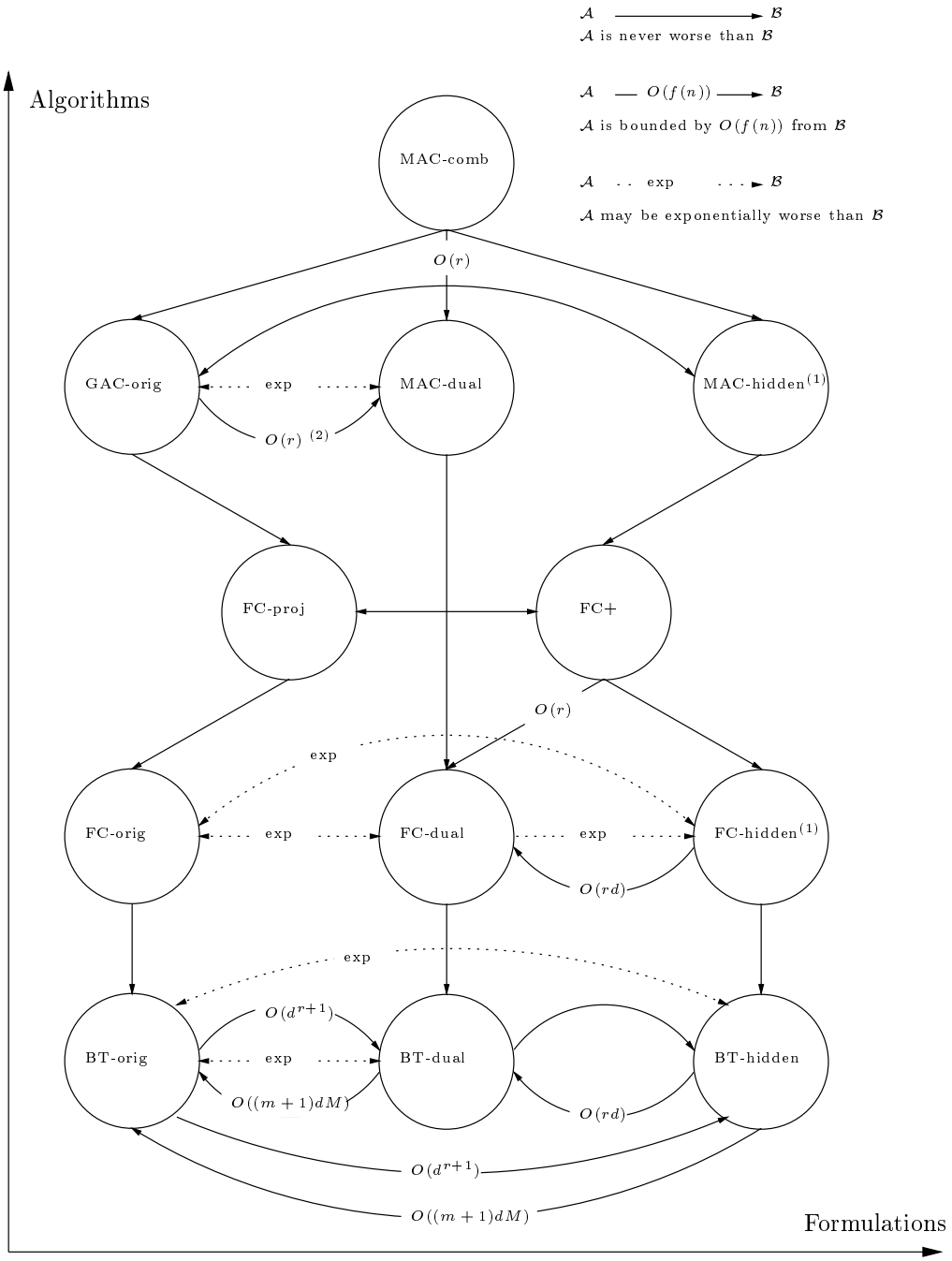
Figure 5.1 summarizes the results that we are going to present in this chapter. In the above figure, we refer to a backtracking algorithm X applied on a problem formulation Y as X-Y. For example, GAC-orig denotes the case in which GAC is applied on the original formulation of a problem. We will show that GAC-orig always visits that same nodes as MAC-hidden if MAC-hidden instantiates all the ordinary variables first in the backtrack search, and GAC-orig is only bounded worse than MAC-dual if in the original formulation every pair of constraints share at most one variable. The above relations related to the chronological backtracking algorithm are identified in Section 5.2, the relations about the forward checking algorithm are identified in Section 5.3, and those about the maintaining arc consistency algorithm are given in Section 5.4.

Given a CSP instance and the variable orderings for the original problem, the dual problem, and the hidden problem, because a backtracking algorithm explores distinct search trees over the above three formulations, in the following, we refer to the search tree explored by the backtracking algorithm in the original problem as the *original search tree*, the one explored in the dual problem as the *dual search tree*, and the one explored in the hidden problem as the *hidden search tree*. We prove the bounded worse relation for a given backtracking algorithm and two formulations $\mathcal{A}$ and $\mathcal{B}$ by establishing a correspondence between (some) nodes in the search tree explored in $\mathcal{A}$ and (some) nodes in the search tree explored in $\mathcal{B}$. In the following, we will use notations *orig-dual*, *hidden-orig*, *dual-hidden*, *hidden-dual*, *hidden*, *allhidden*, *dual* and *alldual* to denote such correspondences, which will be defined in the context.

5.2 Chronological Backtracking Algorithm (BT)

The properties of the nodes in the BT backtrack tree have been characterized by Kondrak and van Beek [69].

[1] Also, M is the maximum domain size of the dual (hidden) variables in the dual (hidden) problem.



(1) In the case that all the ordinary variables in the hidden problem are instantiated first.

(2) In the case that each pair of the original constraints share at most one variable.

Figure 5.1: A two dimensional diagram showing the relations between the combinations of algorithms and formulations.

Theorem 5.1 *[69] BT visits a node if it is consistent. BT visits a node only if its parent is consistent* [2].

In the following discussion, we denote BT applied on the original problem as *BT-orig*, BT applied on the dual problem as *BT-dual*, and BT applied on the hidden problem as *BT-hidden*.

Example 5.2 *Consider a CSP over the set of Boolean variables,* $\{x_1, \ldots, x_n\}$. *The only constraint in the problem is an n-ary constraint over all the variables,* $C(x_1, \ldots, x_n) = \{(0, \ldots, 0), (1, \ldots, 1)\}$. *The dual problem has only one variable and its domain has only two values. So BT-dual visits two nodes, each identifying a solution of the problem. In the hidden problem, if the only hidden variable is instantiated first, each of the ordinary variables has only one value in the domain to be compatible with the instantiation of the hidden variable. Thus BT-hidden visits* $2n + 2$ *nodes. However, by any variable ordering, BT-orig is unable to check the constraint until all the variables have been instantiated, then it will recursively test every possible instantiations to* $x_1, \ldots, x_n$. *Thus, BT-dual and BT-hidden are exponentially better than BT-orig in the above example.*

Theorem 5.2 *There is a CSP instance in which BT-dual and BT-hidden are always exponentially better than BT-orig no matter what variable ordering is used in the original problem.*

Proof: It is true from the CSP in Example 5.2. ∎

Example 5.3 *Consider a non-binary CSP with n Boolean variables,* $x_1, \ldots, x_n$ *and n constraints given by* $(x_1), (\neg x_1 \vee x_2), (\neg x_1 \vee \neg x_2 \vee x_3), \ldots$, *and* $(\neg x_1 \vee \neg x_2 \cdots x_n)$. *BT-orig would visit* $2n$ *nodes, whereas, because the maximum domain size of the dual (hidden) variables is* $2^n - 1$, *by any variable ordering strategy, BT-dual or BT-hidden has to visit at least* $O(2^n)$ *nodes.*

Theorem 5.3 *There is a CSP instance in which BT-orig is always exponentially better than BT-dual and BT-hidden no matter what variable orderings are used in them.*

Proof: It is true from the CSP in Example 5.3. ∎

[2] Their original result only applies to binary CSPs. However, it is also valid on non-binary CSPs from their proof.

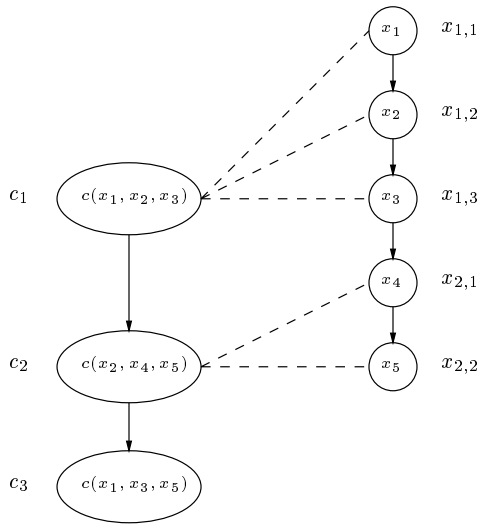


Figure 5.2: The correspondence between the ordering of the dual variables and the ordering of the ordinary variables.

5.2.1 BT-orig and BT-dual

In the worst case, BT-orig is exponentially worse than BT-dual and BT-hidden. As we can see from Example 5.2, the exponent comes from the arity of the non-binary constraint. We show next that: If the maximum arity of the constraints in the original problem is bounded by a constant r, BT-orig is only bounded worse than BT-dual. That is, given a variable ordering in the dual problem, we can construct a variable ordering in the original problem, such that the number of nodes visited by BT-orig in the original search tree is bounded by a polynomial from the number of nodes visited by BT-dual in the dual search tree.

Given an ordering of the dual variables, $c_1, \ldots, c_m$, because an instantiation of a dual variable is equivalent to the instantiations of several ordinary variables, we can arrange the ordinary variables in the original problem in the order that they are instantiated. For example, as shown in Figure 5.2, given the static ordering of the dual variables, $c(x_1, x_2, x_3)$, $c(x_2, x_4, x_5)$, and $c(x_1, x_3, x_5)$, the ordinary variables can be ordered as x_1, x_2, x_3, x_4, and x_5. For convenience, we denote the ordinary variables, which are "instantiated" as the dual variable c_i is instantiated, as $x_{i,1}, \ldots, x_{i,r_i}$, *i.e.*, $x_{i,j} \in vars(c_i)$ and $x_{i,j} \notin \bigcup_{k=1}^{i-1} vars(c_k)$. Therefore, under the above orderings, each ordinary variable $x_{i,j}$ in the original problem corresponds to a unique dual variable c_i in the dual problem. However, not all the dual variables have some correspondence in the ordinary variables, such as the dual variable c_3 in the above figure, which does

not make any new instantiations to the ordinary variables. Therefore, in the original problem, x_{i,r_i} may not always be followed by $x_{i+1,1}$ (because $x_{i+1,1}$ may not exist).

Although BT-orig and BT-dual explore different search trees, as we show next, there exists a correspondence between (some) nodes at the level of x_{i,r_i} in the original search tree and (some) nodes at the level of c_i in the dual search tree. We then show that the total number of the nodes visited by BT-orig (in the original search tree) is bounded by a polynomial factor from the total number of the nodes visited by BT-dual (in the dual search tree).

Observation 5.1 *Under the above orderings, if a node t at the level of x_{i,r_i} in the original search tree is consistent with constraints $C_1, \ldots, C_i$,* i.e., *$t[vars(C_j)] \in rel(C_j)$, $j = 1, \cdots, i$, t corresponds to a unique node at the level of c_i in the dual search tree, given by orig-dual$(t) = \{c_1 \leftarrow t[vars(c_1)], \ldots, c_i \leftarrow t[vars(c_i)]\}$.*

The condition that t is consistent with the constraints $C_1, \ldots, C_i$ cannot be relaxed. Otherwise, suppose t does not satisfy a constraint C_j, for $1 \leq j \leq i$, *i.e.*, $t[vars(c_j)] \notin rel(C_j)$, then $t[vars(c_j)]$ is not a valid tuple in the domain of the dual variable c_j and thus *orig-dual*(t) is not a legal node in the dual search tree.

Lemma 5.4 *If a node t at the level of x_{i,r_i} in the original search tree is consistent, its corresponding node at the level of c_i in the dual search tree, orig-dual(t), is consistent in the dual problem.*

Proof: Suppose *orig-dual*(t) does not satisfy a dual constraint between two dual variables c_j and c_k, where $1 \leq j, k \leq i$. Thus *orig-dual*$(t)[c_j]$ does not agree with *orig-dual*$(t)[c_k]$ on the part $vars(c_j) \cap vars(c_k)$. Because *orig-dual*$(t)[c_j]$ is set to be $t[vars(c_j)]$ and *orig-dual*$(t)[c_k]$ is set to be $t[vars(c_k)]$, they must agree on the shared ordinary variables. That is a contradiction. ∎

We have established the correspondence between (some of) the nodes at the level of x_{i,r_i} in the original search tree and (some of) the nodes at the level of c_i in the dual search tree. We now show that there is a correspondence between the nodes visited by BT-orig in the original search tree and (some of) the nodes visited by BT-dual in the dual search tree.

Theorem 5.5 *Given any CSP instance, there is a variable ordering such that BT-orig visits at most $O(d^{r+1})$ times as many nodes as BT-dual does.*

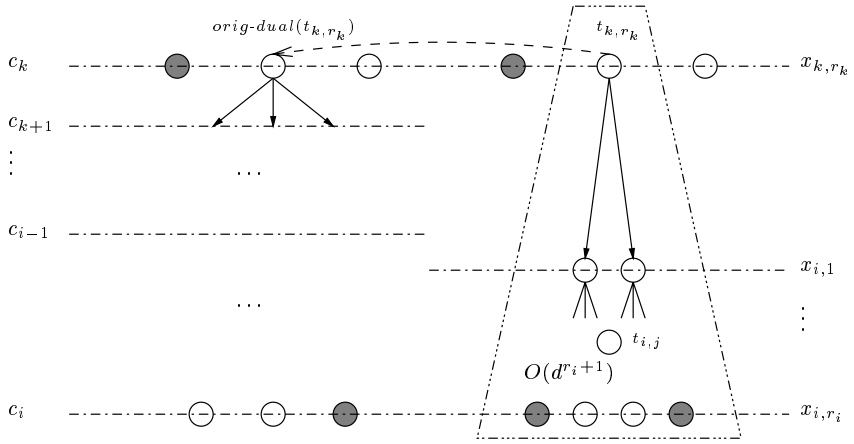


Figure 5.3: The correspondence between the nodes visited by BT-orig in the original search tree and the nodes visited by BT-dual in the dual search tree.

Proof: From Lemma 5.4 and Theorem 5.1, if a node t at the level of x_{i,r_i} in the original search tree is consistent, its correspondence *orig-dual*(t) at the level of c_i in the dual search tree is consistent and thus BT-dual will visit *orig-dual*(t). If BT-orig visits a node $t_{i,j}$ at the level of $x_{i,j}$ in the original search tree, let x_{k,r_k} be the variable immediately followed by $x_{i,1}$, from Theorem 5.1, $t_{i,j}$'s ancestor t_{k,r_k} at the level of x_{k,r_k} is consistent. The total number of the descendants of t_{k,r_k} from the level of $x_{i,1}$ to the level of x_{i,r_i} is bounded by $O(d^{r+1})$, as shown in Figure 5.3. Thus the total number of the nodes visited by BT-orig is at most $O(d^{r+1})$ times as many as the total number of the consistent nodes in the dual search tree, which is bounded by the total number of the nodes visited by BT-dual. ∎

The above bound is tight as we can verify it in Example 5.2. Thus, BT-orig may be exponentially worse than BT-dual, but if the maximum arity of the non-binary constraints in the original problem is bounded by a constant, BT-orig is only bounded worse than BT-dual.

Example 5.4 *We apply BT-dual and BT-orig to solve the CSP in Example 2.1, as shown in Figure 5.4. Because the node* $\{x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 2\}$ *is consistent in the original problem, it has a correspondence at the level of* c_1 *in the dual search tree,* $\{c(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$. *From Lemma 5.4, its correspondence is consistent in the dual problem. Therefore, BT-orig will visit* $\{(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$, *and BT-dual will visit* $\{c(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$. *Also, as we can see in the above figure, the consistent node* $\{(x_1, x_2, x_3, x_4, x_5) \leftarrow (0, 0, 2, 0, 0)\}$ *at the level of* $x_{2,2}$ *in the original search tree*

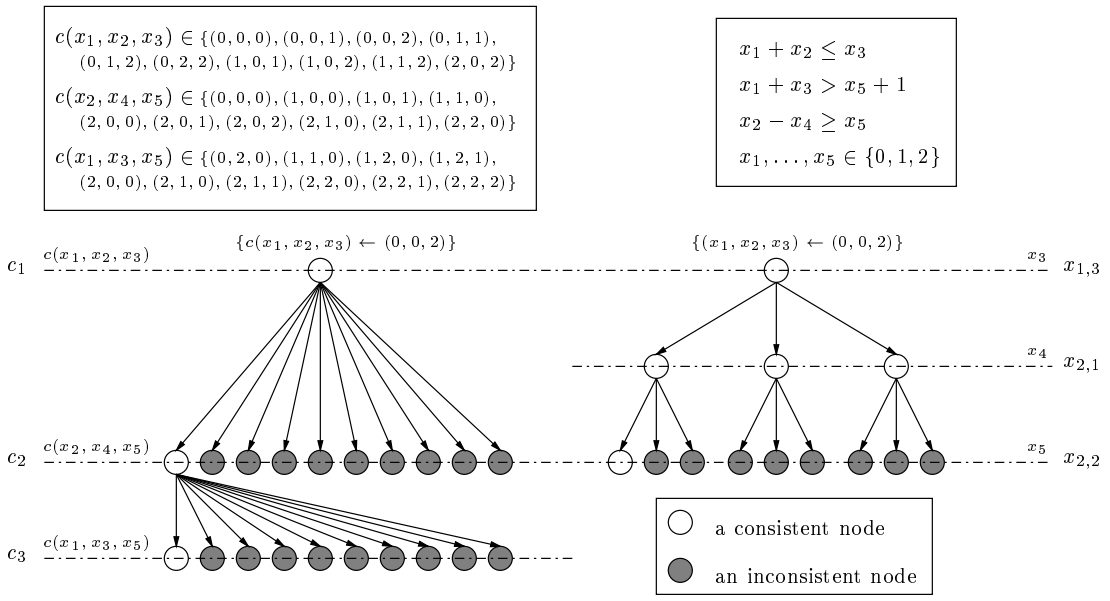


Figure 5.4: The comparison of BT-dual and BT-orig in solving the CSP in Example 2.1.

corresponds to the consistent node $\{(c(x_1, x_2, x_3), c(x_2, x_4, x_5) \leftarrow ((0,0,2),(0,0,0))\}$ *at the level of* c_2 *in the dual search tree. The node* $\{(x_1, x_2, x_3) \leftarrow (0,0,2)\}$ *has 12 (bounded by* 3^4*) descendents from the level of* $x_{2,1}$ *to the level of* $x_{2,2}$ *in the original search tree, and the total number of the nodes visited by BT-orig is at most* 3^4 *times as many as the total number of the nodes visited by BT-dual.*

5.2.2 BT-hidden and BT-orig

Note in Example 5.3, BT-hidden has to visit an exponential number of nodes because the maximum domain size of the hidden variables is exponential in n. However, if the maximum domain size of the hidden variables is bounded by a constant M, we can arrange a variable ordering for BT-hidden such that it is only bounded worse than BT-orig.

Given a variable ordering for BT-orig in the original problem, $x_1, \ldots, x_n$, we can construct an ordering for BT-hidden on the hidden problem. The ordinary variables in the hidden problem are instantiated in the same order as they are in the original problem. Furthermore, at each node in the original search tree, if the instantiation of the current variable x_i makes some constraints $C_{i,1}, \ldots, C_{i,r_i}$ checkable, in the variable ordering for the hidden problem, we instantiate the hidden variables $c_{i,1}, \ldots, c_{i,r_i}$ corresponding to those newly checkable constraints (breaking ties arbitrarily). Thus the

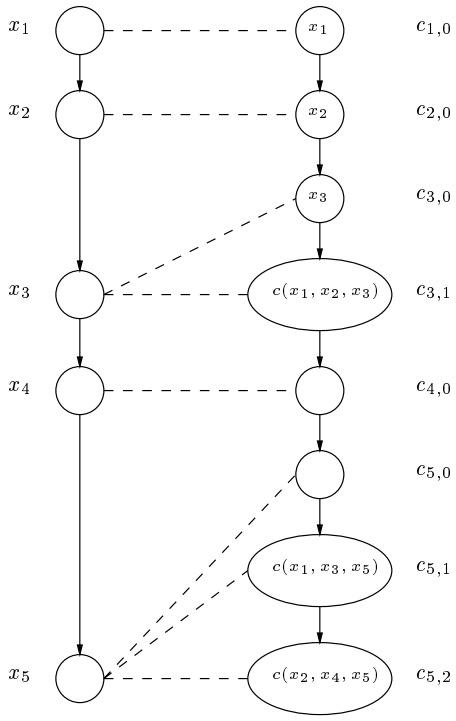


Figure 5.5: The correspondence between the variables in the original problem and the variables in the hidden problem.

variable ordering for the hidden problem is $c_{1,0}, \ldots, c_{1,r_1}, \ldots, c_{n,0}, \ldots, c_{n,n_r}$, where $c_{i,0}$ denotes the ordinary variable x_i and $c_{i,j}$ for $j \geq 1$ denotes a hidden variable such that $x_i \in vars(c_{i,j})$ and $vars(c_{i,j}) \subseteq \{x_1, \ldots, x_i\}$, *i.e.*, the constraints $C_{i,j}$ becomes checkable at the level of x_i in the original search tree. Note that the number r_i is bounded by the total number of the constraints in the original problem, m. For example, suppose in the original problem the ordinary variables are ordered as $x_1, \ldots, x_5$, and there are three constraints $C(x_1, x_2, x_3)$, $C(x_2, x_4, x_5)$ and $C(x_1, x_3, x_5)$. The variable ordering constructed for the hidden problem is $x_1, x_2, x_3, c(x_1, x_2, x_3), x_4, x_5, c(x_2, x_4, x_5)$, $c(x_1, x_3, x_5)$, as shown in Figure 5.5. Under the above orderings, a node t at the level of c_{i,r_i} in the hidden search tree, corresponds to a node *hidden-orig*(t) at the level of x_i in the original search tree, where *hidden-orig*$(t)[x_j] = t[x_j]$ for $1 \leq j \leq i$.

Lemma 5.6 *If a node t at the level of c_{i,r_i} in the hidden search tree is consistent, its correspondence hidden-orig(t) is a consistent node at the level of x_i in the original search tree.*

Proof: Suppose *hidden-orig*(t) does not satisfy the constraint C, *i.e.*, *hidden-orig*$(t)[vars(C)] \notin rel(C)$. That means, in the hidden problem, the instantiation of the hidden variable c, $t[c]$ is incompatible with the instantiations to the ordinary variables in the scheme of c. Thus t is not consistent. That is a contradiction. ∎

Therefore, the total number of the consistent nodes at the level of c_{i,r_i} in the hidden search tree is bounded by the total number of the consistent nodes at the level of x_i in the original search tree.

Theorem 5.7 *Given any CSP instance, there is a variable ordering such that BT-hidden visits at most $O((m+1)dM)$ times as many nodes as BT-orig does.*

Proof: If BT-hidden visits a node $t_{i,j}$ at the level of $c_{i,j}$ in the hidden search tree, we know that $t_{i,j}$'s ancestor $t_{i-1,r_{i-1}}$ at the level of $c_{i-1,r_{i-1}}$ is consistent. Now we estimate the total number of the descendants of $t_{i-1,r_{i-1}}$ at the levels of $x_i, c_{i,1}, \ldots, c_{i,r_i}$. $t_{i-1,r_{i-1}}$ has at most d children at the level of x_i for each value in the domain of x_i. Once the ordinary variable x_i is instantiated, there is at most one tuple left in the domain of each of the hidden variables $c_{i,1}, \ldots, c_{i,r_i}$ to be compatible with previous instantiations, as shown in Figure 5.6. The total number of the descendants of $t_{i-1,r_{i-1}}$ at the levels of $x_i, c_{i,1}, \ldots, c_{i,r_i}$ is bounded by $O((r_i+1)dM)$. Note that r_i is bounded by the number of the constraints that involve x_i. Thus the total number of the nodes visited

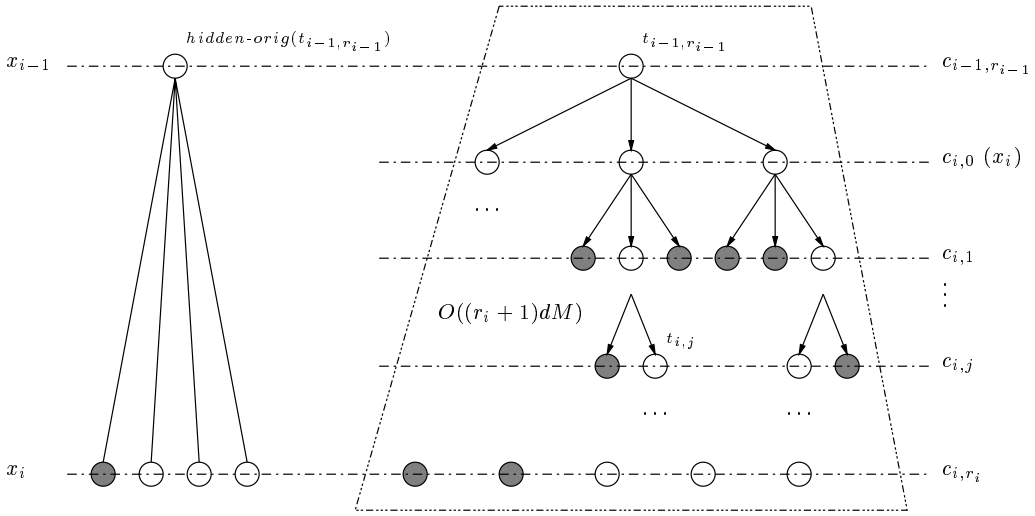


Figure 5.6: The total number of the descendants at the levels of $x_i, c_{i,1}, \ldots, c_{i,r_i}$ of a consistent node at the level of $c_{i-1,r_{i-1}}$ in the hidden search tree is bounded by $O((r_i+1)dM)$.

by BT-hidden is bounded by a factor $O((m+1)dM)$ from the total number of the consistent nodes in the original search tree, which is bounded by the total number of the nodes visited by BT-orig. ∎

Example 5.5 *We apply BT-orig and BT-hidden to solve the CSP in Example 2.1, as shown in Figure 5.7. For example, in the hidden search tree, the node* $t = \{(x_1, x_2, x_3, c(x_1, x_2, x_3), x_4) \leftarrow (0, 0, 2, (0, 0, 2), 0)\}$ *at the level of* $c_{4,0}$ *is consistent, thus it has a correspondence* $hidden\text{-}orig(t) = \{(x_1, x_2, x_3, x_4) \leftarrow (0, 0, 2, 0)\}$ *at the level of* x_4 *in the original search tree. From Lemma 5.6, its correspondence is consistent in the original problem. Therefore, BT-hidden will visit* t*, and BT-orig will visit* $hidden\text{-}orig(t)$*. As we can see,* t *has* 3 *children at the level of* $c_{5,0}$*, and each of these nodes has at most one consistent descendant at the level of* $c_{5,1}$ *and the level of* $c_{5,2}$*. Thus the total number of the descendants of* t *at the levels* $c_{5,0}$*,* $c_{5,1}$ *and* $c_{5,2}$ *are bounded by* $((m+1)dM)$*.*

5.2.3 BT-dual and BT-hidden

Given a variable ordering for the hidden problem, we can construct an ordering of the dual variables for BT-dual, in which the dual variables are ordered exactly the same as they are in the ordering of the hidden problem. For example, if variables in the

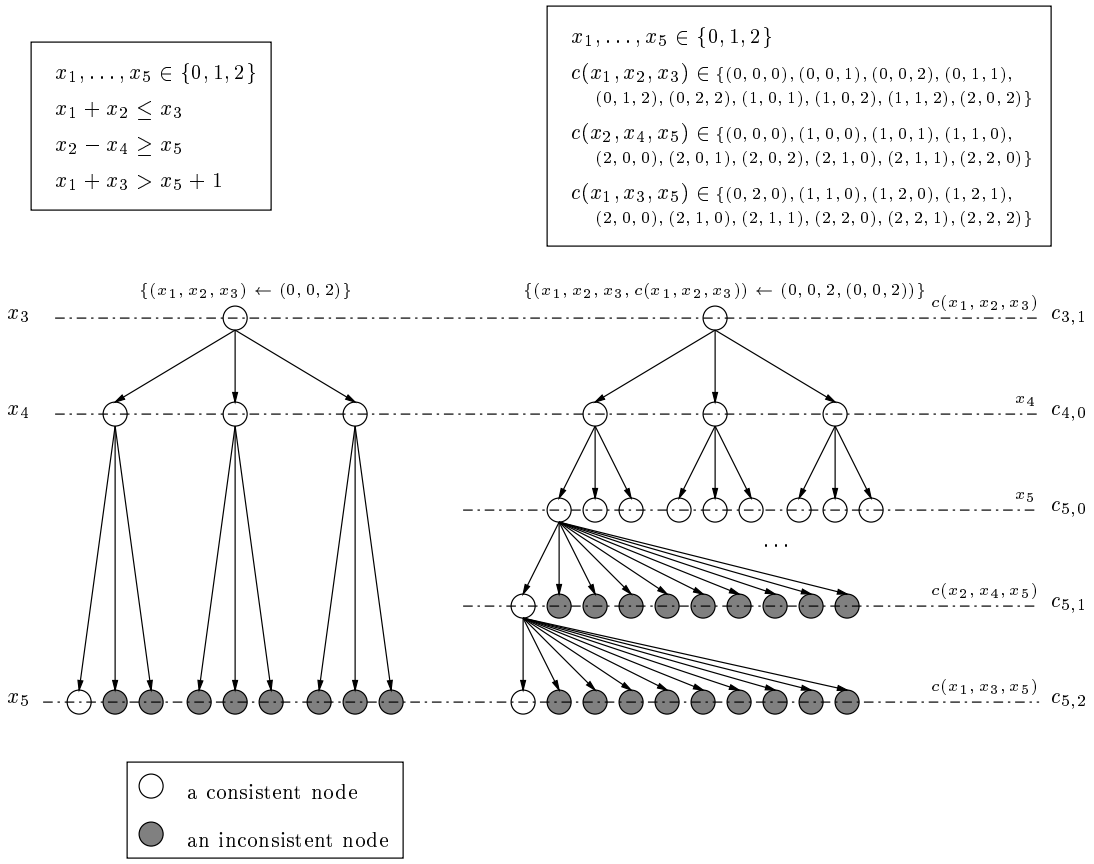


Figure 5.7: The comparison of BT-orig and BT-hidden in solving the CSP in Example 2.1.

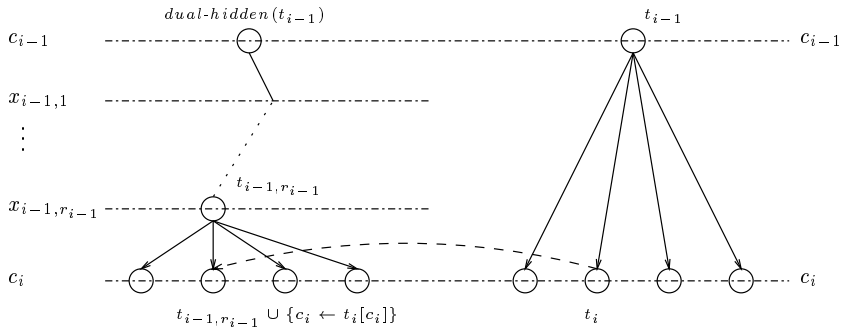


Figure 5.8: A node t_i visited by BT-dual at the level of c_i in the dual search tree corresponds to a unique node visited by BT-hidden at the level of c_i in the hidden search tree.

hidden problem are ordered as $x_1, x_2, c(x_1, x_2, x_3), x_3, x_4, c(x_1, x_3, x_5), x_5, c(x_2, x_4, x_5)$, the ordering for the dual problem is $c(x_1, x_2, x_3), c(x_1, x_3, x_5), c(x_2, x_4, x_5)$.

Observation 5.2 *If a node t at the level of c_i in the dual search tree is consistent, t corresponds to a unique node dual-hidden(t) at the level of c_i in the hidden search tree, where for each hidden variable c, dual-hidden$(t)[c]$ is set to be $t[c]$, and for each ordinary variable x, if there is an instantiated dual variable c in t such that $x \in vars(c)$, dual-hidden$(t)[x]$ is set to be the projection, $(t[c])[x]$. Otherwise, dual-hidden$(t)[x]$ is set to be the first value in the domain of x. Because t is consistent, dual-hidden$(t)[x]$ is irrelevant to whichever dual variable we choose to make the projection.*

Lemma 5.8 *If a node t at the level of c_i in the dual search tree is consistent, its correspondence dual-hidden(t) is a consistent node at the level of c_i in the hidden search tree.*

Proof: Suppose *dual-hidden*(t) does not satisfy the hidden constraint between the hidden variable c and the ordinary variable x, where $c \in vars(t)$ and $x \in vars(c)$. Because *dual-hidden*$(t)[x]$ is set to be $(t[c])[x]$ and *dual-hidden*$(t)[c]$ is set to be $t[c]$, the instantiation of c is compatible with the instantiation of x in *dual-hidden*(t). That is a contradiction. ∎

Because BT-dual visits a node only if its parent is consistent, we can immediately conclude that BT-dual visits at most $O(M)$ times as many nodes as BT-hidden does. Moreover, we can prove BT-dual will never visit more nodes than BT-hidden does.

Theorem 5.9 *Given a CSP instance and a variable ordering for BT-hidden, there is a variable ordering such that BT-dual never visits more nodes than BT-hidden does.*

Proof: Suppose BT-dual visits a node t_i at the level of c_i in the dual search tree, we will prove that t_i corresponds to a unique node visited by BT-hidden at the level of c_i in the hidden search tree. From Theorem 5.1, t_i's parent t_{i-1} at the level of c_{i-1} in the dual search tree is consistent. Thus, *dual-hidden*(t_{i-1}) is a consistent node at the level of c_{i-1} in the hidden search tree. Suppose in the variable ordering for the hidden problem, the ordinary variables $x_{i-1,1}, \ldots, x_{i-1,r_{i-1}}$ are instantiated after c_{i-1} but before c_i. We will show that *dual-hidden*(t_{i-1}) has a consistent descendant $t_{i-1,r_{i-1}}$ at the level of $x_{i-1,r_{i-1}}$ in the hidden search tree. For each of the ordinary variables $x_{i-1,j}$, if $x_{i-1,j} \in \bigcup_{k=1}^{i-1} vars(c_k)$, *i.e.*, $x_{i-1,j}$ has been "instantiated" by the instantiation to a hidden variable c, then $t_{i-1,r_{i-1}}[x_{i-1,j}]$ is set to be $(t[c])[x_{i-1,j}]$. Because the node t_{i-1} is consistent in the dual problem, $t_{i-1,r_{i-1}}[x_{i-1,j}]$ is irrelevant to whichever hidden variable we choose to make the projection. Otherwise, if $x_{i-1,j}$ has not been "instantiated" from the instantiations of the hidden variables, $t_{i-1,r_{i-1}}[x_{i-1,j}]$ is set to be the first value in the domain of $x_{i-1,j}$. $t_{i-1,r_{i-1}}$ is consistent because for each of the ordinary variables $x_{i-1,j}$, if $x_{i-1,j}$ is constrained with an instantiated hidden variable c, $t_{i-1,r_{i-1}}$ should satisfy the constraint. Therefore, BT-hidden will visit $t_{i-1,r_{i-1}}$ and extend it to the level of c_i. For each tuple t in the domain of c_i, the node $t_{i-1,r_{i-1}} \cup \{c_i \leftarrow t\}$ is visited by BT-hidden. Let the node t_i corresponds to the node, $t_{i-1,r_{i-1}} \cup \{c_i \leftarrow t_i[c_i]\}$, visited by BT-hidden at the level of c_i in the hidden search tree, as shown in Figure 5.8. Therefore, the total number of the nodes visited by BT-dual is bounded by the total number of the nodes visited by BT-hidden. ∎

Example 5.6 *We apply BT-hidden and BT-dual to solve the CSP in Example 2.1, as shown in Figure 5.9. For example, BT-dual visits a node* $t_2 = \{(c(x_1, x_2, x_3), c(x_1, x_3, x_5)) \leftarrow (0, 0, 2), (1, 1, 0)\}$ *at the level of* c_2 *in the dual search tree. Thus* t_2*'s parent* $t_1 = \{c(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$ *at the level of* c_1 *is consistent in the dual problem. From Lemma 5.8,* t_1*'s correspondence hidden-dual*$(t_1) = \{(x_1, x_2, c(x_1, x_2, x_3)) \leftarrow (0, 0, (0, 0, 2))\}$ *at the level of* c_1 *in the hidden search tree is consistent. Furthermore, hidden-dual*(t_1) *has a consistent descendant* $t_{1,2} = \{(x_1, x_2, c(x_1, x_2, x_3), x_3, x_4) \leftarrow (0, 0, (0, 0, 2), 2, 0)$ *at the level of* $x_{1,2}$ *in the hidden search tree, and* t_2 *corresponds to one of* $t_{1,2}$*'s children* $\{(x_1, x_2, c(x_1, x_2, x_3), x_3, x_4, c(x_1, x_3, x_5)) \leftarrow (0, 0, (0, 0, 2), 2, 0, (1, 1, 0))$ *visited by BT-hidden at the level* c_2 *in the hidden search tree. Note that* x_4 *is not in the scheme of the hidden variables* c_1 *and* c_2*, thus the instantiation of* x_4 *in* $t_{1,2}$ *is set to be the first value in its domain. As we can see, BT-dual visits fewer nodes than BT-hidden does in the above example.*

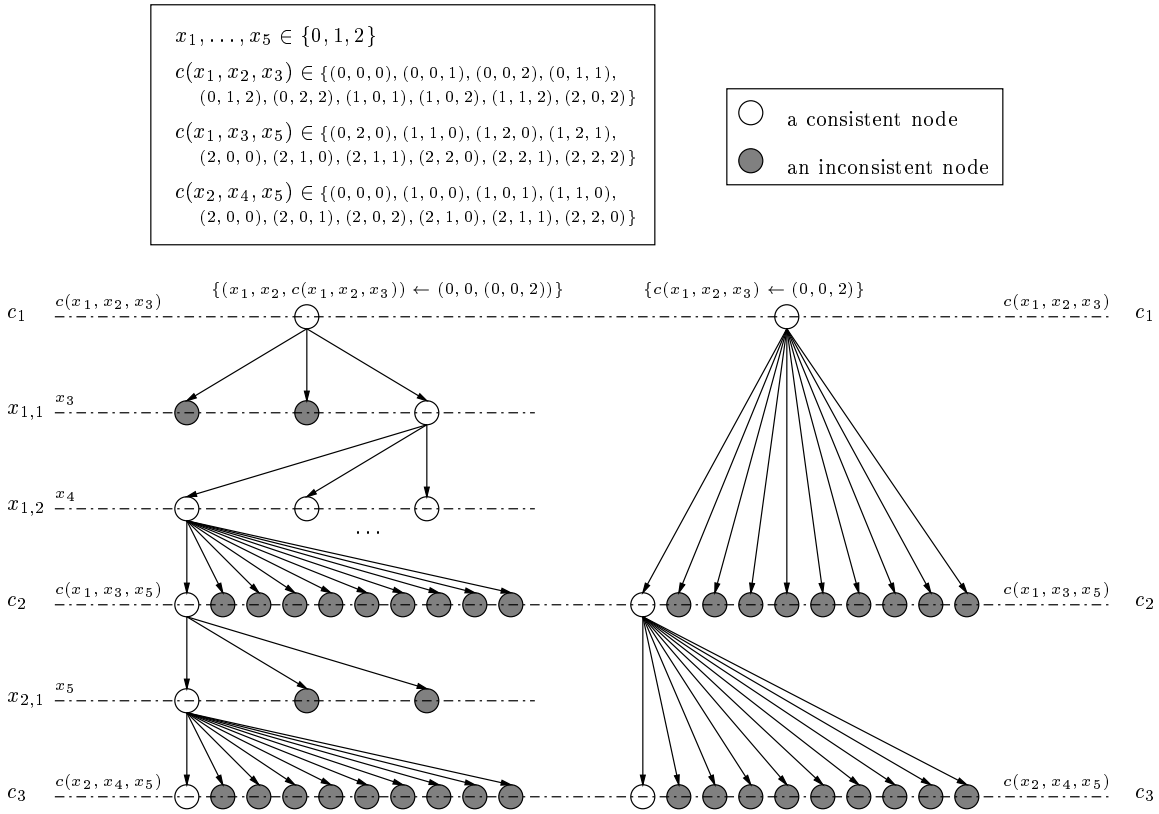


Figure 5.9: The comparison of BT-hidden and BT-dual in solving the CSP in Example 2.1.

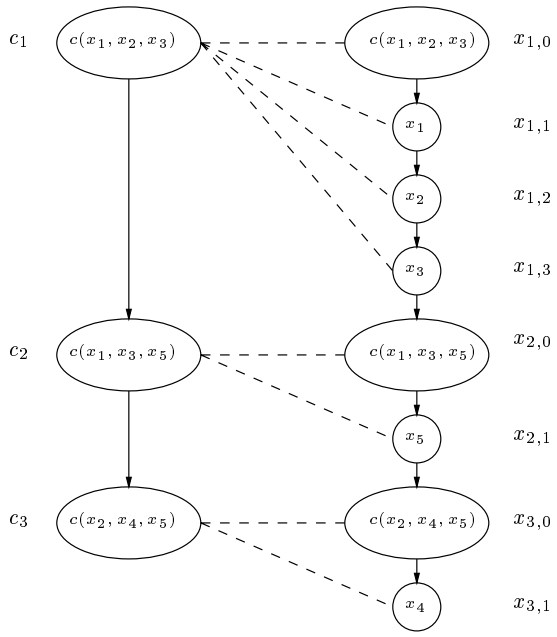


Figure 5.10: The correspondence between the variables in the hidden problem and the variables in the dual problem.

From Theorem 5.7 and Theorem 5.9, given a CSP instance and a variable ordering for BT-orig, we can construct a variable ordering for BT-hidden such that BT-hidden visits at most $O(mdM)$ as many nodes as BT-orig does. Then we can construct a variable ordering for BT-dual such that BT-dual will never visit more nodes than BT-hidden does. Therefore, BT-dual will visit at most $O((m+1)dM)$ as many nodes as BT-orig does.

Corollary 5.10 *Given a CSP instance and a variable ordering for BT-orig, there is a variable ordering such that BT-dual visits at most $O((m+1)dM)$ times as many nodes as BT-orig does.*

We may conclude from the above results that BT-hidden visits at most $O((m+1)d^{r+2}M)$ times as many nodes as BT-dual does. Moreover, we can construct a variable ordering for BT-hidden such that the above bound is even tight. Given a variable ordering in the dual problem, $c_1, \ldots, c_m$, we choose the hidden variable c_1 to be the first in the ordering for the hidden problem. Then we instantiate each of the ordinary variables $x_{1,1}, \ldots, x_{1,r_1}$ in $vars(c_1)$ (if the ordinary variable has not been instantiated yet and breaking ties arbitrarily). After c_1 and its ordinary variables have been instantiated, we go on to c_2, and so on. Thus, the variable ordering for

the hidden problem is $x_{1,0}, x_{1,1}, \ldots, x_{1,r_1}, \ldots, x_{m,0}, x_{m,1}, \ldots, x_{m,r_m}$, where $x_{i,0}$ denote the hidden variable c_i and $x_{i,j}$ is an ordinary variable such that $x_{i,j} \in vars(c_i)$ and $x_{i,j} \notin \bigcup_{k=1}^{i-1} vars(c_k)$. For example, as shown in Figure 5.10, if the dual variables are ordered as $c(x_1, x_2, x_3), c(x_1, x_3, x_5)$ and $c(x_2, x_4, x_5)$, then the variable ordering in the hidden problem is $c(x_1, x_2, x_3), x_1, x_2, x_3, c(x_1, x_3, x_5), x_5, c(x_2, x_4, x_5), x_4$. Under the above orderings, for each of the nodes t at the level of $x_{i,0}$ in the hidden search tree, t corresponds to a unique node *hidden-dual*(t) at the level of c_i in the dual search tree, where for each of the instantiated dual variables c, *hidden-dual*$(t)[c]$ is set to be $t[c]$.

Lemma 5.11 *If the node t at the level of $x_{i,0}$ in the hidden search tree is consistent, its correspondence hidden-dual(t) is a consistent node at the level of c_i in the dual search tree.*

Proof: Suppose *hidden-dual*(t) does not satisfy the dual constraint between c_j and c_k,for $1 \leq j, k \leq i$. That is, $(t[c_j])[vars(c_j) \cap vars(c_k)]$ does not agree with $(t[c_k])[vars(c_j) \cap vars(c_k)]$. Thus, there is an ordinary variable $x \in vars(c_j) \cap vars(c_k)$ such that $(t[c_j])[x] \neq (t[c_k])[x]$. Because in the hidden problem x was constrained with hidden variables, c_j and c_k, x must have been instantiated in t, either before the instantiation of c_j or the instantiation of c_k, or both. Note that t is consistent in the hidden search tree. So that $t[c_j]$ and $t[c_k]$ must have the same value over x. That is a contradiction. ∎

Thus the total number of the consistent nodes at the level of $x_{i,0}$ in the hidden search tree is bounded by the total number of the consistent nodes at the level of c_i in the dual search tree.

Theorem 5.12 *Given a CSP instance and a variable ordering for BT-dual, there is a variable ordering such that BT-hidden visits at most $O(rd)$ times as many nodes as BT-dual does.*

Proof: For each of the consistent nodes $t_{i,0}$ at the level of $x_{i,0}$ in the hidden search tree, $t_{i,0}$ has exactly one consistent descendant at each of the levels, $x_{i,1}, \ldots, x_{i,r_i}$, because the ordinary variables $x_{i,1}, \ldots, x_{i,r_i}$ only constrain with $x_{i,0}$ (c_i) in the context, and each of them has only one value in the domain to be compatible with the instantiation of $x_{i,0}$. Thus, the total number of the consistent nodes in the hidden search tree is bounded by a factor $O(r)$ of the total number of the consistent nodes in the dual search tree. Therefore, the total number of the nodes visited by BT-hidden is at most $O(rd)$ as many as the nodes visited by BT-dual in the dual search tree. ∎

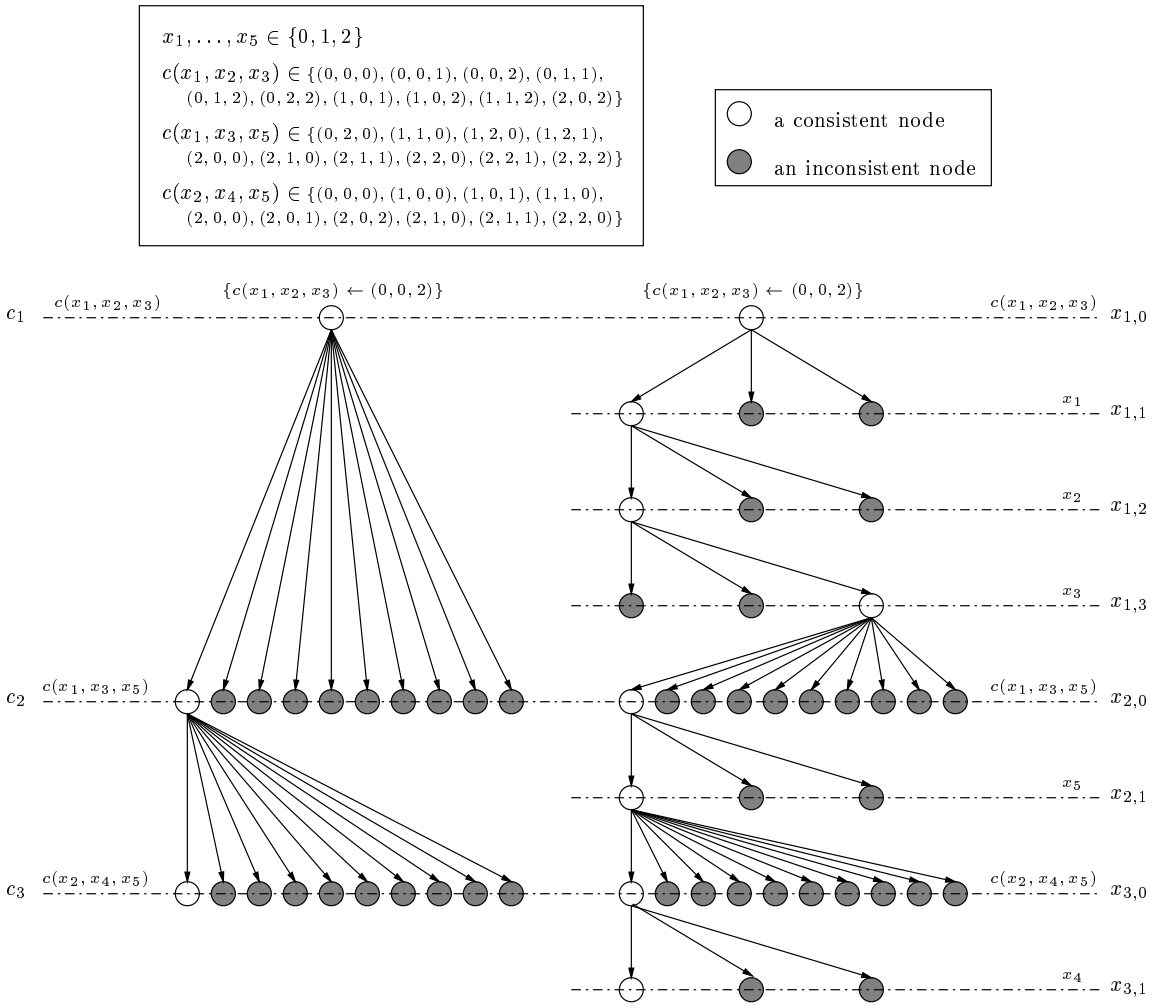


Figure 5.11: The comparison of BT-dual and BT-hidden in solving the CSP in Example 2.1 such that BT-hidden visits at most $O(rd)$ times as many nodes as BT-dual visits.

Example 5.7 *Again, we use BT-dual and BT-hidden to solve the CSP in Example 2.1, but under different variable orderings, as shown in Figure 5.11. For example, in the hidden search tree, the node* $\{c(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$ *at the level of* $x_{1,0}$ *is consistent, thus it corresponds to a unique node* $\{c(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$ *at the level of* c_1 *in the dual search tree. From Lemma 5.11, its correspondence is consistent in the dual problem. Furthermore, in the hidden search tree,* $\{c(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$ *has at most one consistent descendant at each of the levels of* $x_{1,1}$, $x_{1,2}$ *and* $x_{1,3}$. *Therefore, the total number of the nodes visited by BT-hidden is bounded by* $O(rd)$ *from the total number of the nodes visited by BT-dual.*

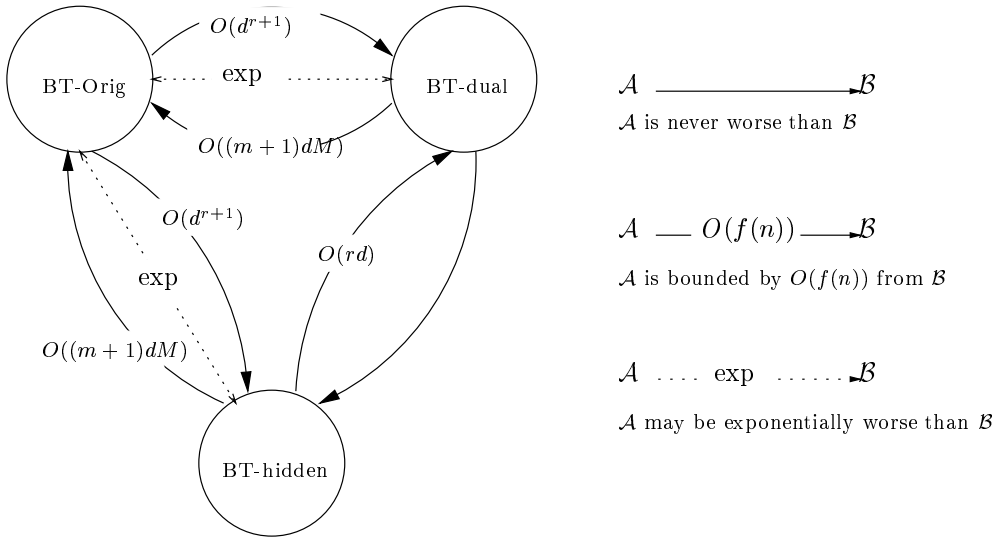


Figure 5.12: The relations between BT-orig, BT-dual and BT-hidden.

We summarize the above results in Figure 5.12. As we can see, BT-dual and BT-hidden are always comparable to each other. BT-dual is never worse than BT-hidden whereas BT-hidden is at most $O(n)$ times worse than BT-dual. BT-dual and BT-hidden are superior to BT-orig if the domains of the dual (hidden) variables are small, *i.e.*, the constraints are tight. On the other hand, BT-orig is better when the maximum arity of the constraints is bounded.

5.3 Forward Checking Algorithm (FC)

In this section, We compare the performance of the forward checking algorithm (FC) [60, 80] on the three formulations. Following Van Hentenryck [114], we say that a *k-ary* constraint, $k \geq 2$, is *forward checkable* if $k-1$ of its variables have been instantiated and the remaining variable is uninstantiated. In that case, the uninstantiated variable is called the forward checked variable. At each node in the search tree, the instantiation of the current variable causes some (possibly empty) set of constraints to become forward checkable. For each newly forward checkable constraint, FC forward checks the remaining uninstantiated variable, *i.e.*, the forward checked variable in the constraint. For each remaining value in the domain of the forward checked variable, FC checks whether or not the instantiation of the forward checked variable with that value along with the instantiations in the current partial solution satisfies the constraint, and the inconsistent values are temporarily removed from the domain

of the forward checked variable. The consistency check fails if a domain wipe out is encountered and the instantiation to the current variable is retracted.

Following Kondrak and van Beek [69], given a CSP and a partial solution t, we say t is consistent with a variable if t can be extended to a consistent partial solution including that variable, and we say t is consistent with all the variables if t is consistent with each of the variables. It is easy to see that if a partial solution t is consistent with all the variables, a subtuple $t' \subseteq t$ is also consistent with all the variables. Kondrak and van Beek have shown that:

Theorem 5.13 *[69] For binary CSPs, FC visits a node if only if it is consistent and its parent is consistent with all the variables.*

In the following discussion, we denote FC applied on the original problem as *FC-orig*, FC applied on the dual problem as *FC-dual*, and FC applied on the hidden problem as *FC-hidden*.

5.3.1 FC-hidden

For a hidden problem, FC-hidden does not have to instantiate all the variables in order to find a solution. Once FC-hidden encounters a state in which for each of the hidden constraints, at least one of its variables has been instantiated, due to the forward checking, each of the uninstantiated variables has only one value remaining in the domain and a solution of the problem can be assembled in a backtrack free manner. For example, a partial solution over all the ordinary variables that is consistent with all the hidden variables, can be extend to a unique solution of the problem, since there is only one remaining tuple in the domain of each hidden variable. On the other hand, once all the hidden variables have been instantiated and there is no domain wipe-out, the domain of each ordinary variable has been reduced to exactly one value.

We will show in the sequel that a variable ordering for FC-hidden that instantiates all the ordinary variables is only bounded worse than any other variable ordering strategy. Given a variable ordering in the hidden problem, $y_1, \ldots, y_l$, in which y_i may be an ordinary variable or a hidden variable. We can construct a new ordering for the ordinary variables only in the hidden problem. If y_i is a hidden variable, since the instantiation of a hidden variable is equivalent to the instantiations of several ordinary variables, in the new ordering, all the uninstantiated ordinary variables $x_{i,1}, \ldots, x_{i,r_i}$ in the scheme of y_i are chosen to be instantiated (breaking ties arbitrarily). If y_i is an ordinary variable and in the new ordering y_i has not

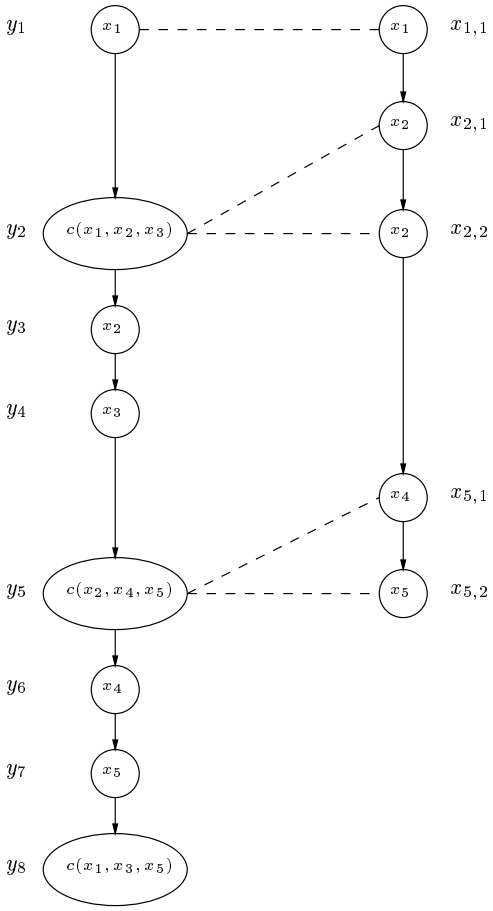


Figure 5.13: The correspondence between the variables in the original ordering and the variables in the new ordering for the hidden problem.

been instantiated, y_i is instantiated in the new ordering and we denote it as $x_{i,1}$. For example, suppose the variables in the hidden problem are originally ordered as $x_1, c(x_1, x_2, x_3), x_2, x_3, c(x_2, x_4, x_5), x_4, x_5, c(x_1, x_3, x_5)$, the new variable ordering for the hidden problem is x_1, x_2, x_3, x_4 and x_5, as shown in Figure 5.13. Under the above orderings, each of the variables $x_{i,j}$ in the new ordering corresponds to a unique hidden or ordinary variable y_i in the original ordering. Note that, in the new ordering, x_{i,r_i} may not always be followed immediately by $x_{i+1,1}$, because $x_{i+1,1}$ may not exist. To distinguish between the search trees generated under the above two orderings, we denote the search tree explored by FC-hidden under the original ordering as the *original hidden tree*, and the search tree explored under the new ordering as the *new hidden search tree.*

Observation 5.3 *If a partial solution t over some ordinary variables of the hidden problem is consistent with all the variables, t can be extended to a unique partial solution $allhidden(t)$ including all the hidden variables c such that $vars(c) \subseteq vars(t)$. If all the ordinary variables in the scheme of the hidden variable c have been instantiated in t, there is only one tuple $t[vars(c)]$ in the domain of c that is compatible with t, and thus $allhidden(t)[c]$ is set to be $t[vars(c)]$. Furthermore, if a node t at the level of x_{i,r_i} in the new hidden search tree is consistent with all the variables, t corresponds to a unique node $hidden(t)$ at the level of y_i in the original search tree, where $hidden(t) = allhidden(t)[\{y_1, \ldots, y_i\}]$* [3].

Note that $allhidden(t)$ is an extension of t (which is a partial solution on ordinary variables) to include all the hidden variables that are "instantiated" by t. $hidden(t)$ is also a subtuple of $allhidden(t)$, and only includes the instantiations of the variables $y_1, \ldots, y_i$. For example, in Figure 5.13, given a partial solution $t = \{(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$, $allhidden(t)$ is $\{(x_1, x_2, x_3, c(x_1, x_2, x_3) \leftarrow (0, 0, 2, (0, 0, 2))\}$ and $hidden(t)$ is $\{(x_1, c(x_1, x_2, x_3)) \leftarrow (0, (0, 0, 2))\}$.

Lemma 5.14 *If a node t at the level of x_{i,r_i} in the new hidden search tree is consistent with all the variables, then $allhidden(t)$ is consistent with all the variables. Furthermore, its correspondence node $hidden(t)$ at the level of y_i in the original hidden search tree is consistent with all the variables.*

Proof: Suppose $allhidden(t)$ is not consistent with a future variable y. If y is an ordinary variable, there exists two hidden variables c and c' such that $(allhidden(t)[c])[y] \neq$

[3] From the construction of the new variable ordering, $y_1, \ldots, y_i$ must have been instantiated in $allhidden(t)$.

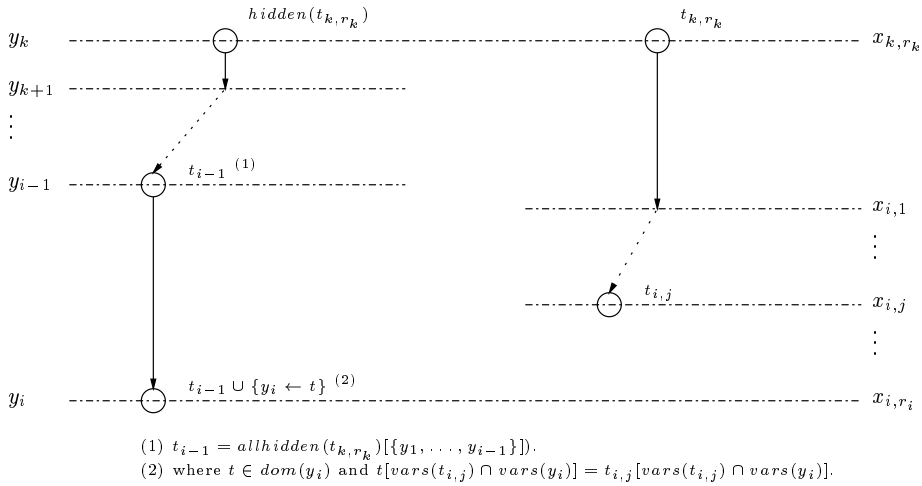


Figure 5.14: A node $t_{i,j}$ visited by FC-hidden at the level of $x_{i,j}$ in the new hidden search tree corresponds to a unique node visited by FC-hidden at the level of y_i in the original hidden search tree.

$(allhidden(t)[c'])[y]$ (otherwise, all the hidden variables instantiated in $allhidden(t)$ have the same value over y, thus y is consistent with $allhidden(t)$). Because $allhidden(t)[c]$ is set to be $t[vars(c)]$ and $allhidden(t)[c']$ is set to be $t[vars(c')]$, they must have the same value over y. That is a contradiction. If y is a hidden variable, because all the ordinary variables instantiated in $allhidden(t)$ are instantiated with the same values in t, that means t is inconsistent with the hidden variable y. This is also a contradiction. Because $hidden(t)$ is a subtuple of $allhidden(t)$, $hidden(t)$ is consistent with all the variables. ∎

Thus the total number of the nodes at the level of x_{i,r_i} that are consistent with all the variables in the new hidden search tree is bounded by the total number of the nodes at the level of y_i in the original search tree that are consistent with all the variables .

Lemma 5.15 *If a node $t_{i,j}$ at the level of $x_{i,j}$, $1 \leq j \leq r_i$, in the new hidden search tree is consistent with all the variables, $t_{i,j}$ corresponds to a node visited by FC-hidden at the level of y_i in the original hidden search tree. Furthermore, for two distinct nodes at the level of $x_{i,j}$ in the new hidden search tree, their correspondences in the original hidden search tree are different.*

Proof: If y_i is an ordinary variable, then j is equal to 1 and $x_{i,j}$ is equal to y_i. From Lemma 5.14, $hidden(t_{i,j})$ at the level of y_i in the original hidden search tree is

consistent with all the variables, and thus it is visited by FC-hidden. Now suppose y_i is a hidden variable. Because $t_{i,j}$ is consistent with all the variables, let t_{k,r_k} denote $t_{i,j}$'s ancestor at the level of x_{k,r_k}, where $x_{i,1}$ immediately follows x_{k,r_k}[4]. We know that t_{k,r_k} is also consistent with all the variables. From Lemma 5.14, t_{k,r_k}'s correspondence $hidden(t_{k,r_k})$ at the level of y_k in the original hidden search tree is consistent with all the variables. Thus FC-hidden will visit $hidden(t_{k,r_k})$ and extend it to the variables $y_{k+1}, \ldots, y_{i-1}$ and y_i. From the construction of the new ordering, for each of the variables $y_{k+1}, \ldots, y_{i-1}$, if it is an ordinary variable, it must have been instantiated in t_{k,r_k}, and if it is a hidden variable, all the ordinary variables in its scheme must have been instantiated in t_{k,r_k}. Thus the node $t_{i-1} = allhidden(t_{k,r_k})[\{y_1, \ldots, y_{i-1}\}]$ at the level of y_{i-1} in the original hidden search tree is consistent with all the variables. Thus FC-hidden will visit t_{i-1} and extend it to the level of y_i. Because $t_{i,j}$ is consistent with all the hidden variables, there is a tuple t in the domain of y_i to be consistent with $t_{i,j}$, that is, $t[vars(t_{i,j}) \cap vars(y_i)] = t_{i,j}[vars(t_{i,j}) \cap vars(y_i)]$. Because for each of the ordinary variables x instantiated at the node t_{i-1}, x has the same instantiation in t_{i-1} and $t_{i,j}$, thus $\{y_i \leftarrow t\}$ is consistent with t_{i-1}. Therefore, FC-hidden visits the node $t_{i-1} \cup \{y_i \leftarrow t\}$ at the level of y_i in the original hidden search tree. Let $t_{i,j}$ corresponds to the node $t_{i-1} \cup \{y_i \leftarrow t\}$. Given two distinct nodes at the level of $x_{i,j}$ in the new hidden search tree, they have different values over the part $vars(t_{i,j}) \cap vars(y_i)$, and thus they must be compatible with different tuples in the domain of y_i. Thus, their correspondences at the level of y_i in the original hidden search tree are different. ∎

Theorem 5.16 *Given a CSP instance and a variable ordering for the hidden problem, we can construct a variable ordering for the ordinary variables in the hidden problem, such that FC-hidden under the new variable ordering visits at most $O(rd)$ times as many nodes as it visits under the original variable ordering.*

Proof: Because the total number of the nodes that are consistent with all the variables at the level of $x_{i,j}$ in the new hidden search tree is bounded by the total number of the nodes at the level of y_i in the original search tree, the total number of the nodes that are consistent with all the variables in the new hidden search tree is at most $O(r)$ times as many as the total number of the nodes that are consistent with all the variables in the original hidden search tree. Note that FC-hidden visits a node only if its parent is consistent with all the variables, and each node may have at most d children. Thus the total number of the nodes visited by FC-hidden in the new

[4]Because in the construction of the new ordering, x_{i,r_i} may not always be followed by $x_{i+1,1}$.

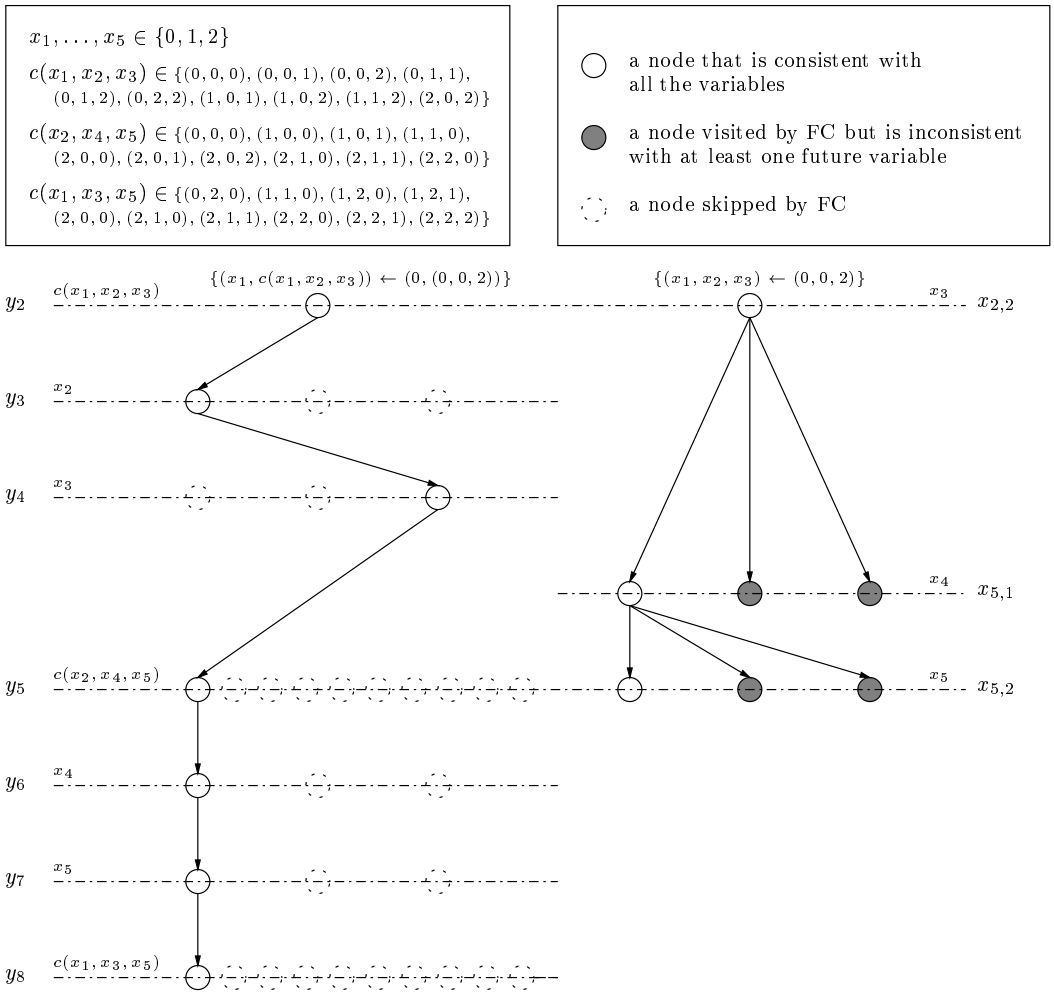


Figure 5.15: The comparison of the search tree explored by FC-hidden under the original variable ordering and new variable ordering to solve the CSP in Example 2.1.

hidden search tree is at most $O(rd)$ times as many as the total number of the nodes visited in the original hidden search tree. ∎

Example 5.8 *We use FC-hidden to solve the CSP in Example 2.1, under the above two variable orderings, as shown in Figure 5.15. For example, in the new hidden search tree, the node* $t_{5,1} = \{(x_1, x_2, x_3, x_4) \leftarrow (0, 0, 2, 0)\}$ *at the level of* $x_{5,1}$ *is consistent with all the variables, thus its parent* $t_{2,2} = \{(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$ *at the level of* $x_{2,2}$ *is also consistent with all the variables. From Lemma 5.14,* $t_{2,2}$ *corresponds to the node* $hidden(t_{2,2}) = \{(x_1, c(x_1, x_2, x_3)) \leftarrow (0, (0, 0, 2))\}$ *at the level of* y_2 *in the original hidden search tree. Furthermore,* $hidden(t_{2,2})$ *has a descendant*

$t_4 = \{(x_1, c(x_1, x_2, x_3), x_2, x_3) \leftarrow (0, (0, 0, 2), 0, 2)\}$ *which is consistent with all the variables at the level of* y_4 *in the original hidden search tree, and* $t_{5,1}$ *corresponds to one of* t_4*'s children,* $\{x_1, c(x_1, x_2, x_3), x_2, x_3, c(x_2, x_4, x_5)) \leftarrow (0, (0, 0, 2), 0, 2, (0, 0, 0))\}$ *visited by FC-hidden at the level of* y_5 *in the original hidden search tree.*

As a special case, when only all the hidden variables are instantiated by FC-hidden, we can show that FC-hidden is equivalent to BT-dual. Because FC-hidden and BT-dual explore the same search tree, *i.e.*, the search tree consists of the partial solutions on the dual or hidden variables, we assume that they use the same variable ordering.

Theorem 5.17 *If all the hidden variables are instantiated first in the hidden problem, FC-hidden visits exactly the same nodes as BT-dual.*

Proof: Suppose that FC-hidden visits a node t. From Theorem 5.13, t's parent $p(t)$ is consistent with all the ordinary variables. Thus, for any ordinary variable x, and for any two hidden variables c_i and c_j, where $c_i, c_j \in vars(p(t))$ and $x \in vars(c_i) \cap vars(c_j)$, $(p(t)[c_i])[x] = (p(t)[c_j])[x]$. Thus $(p(t)[c_i])[vars(c_i) \cap vars(c_j)] = (p(t)[c_j])[vars(c_i) \cap vars(c_j)]$. That means, $p(t)$ is a consistent node in the dual search tree. From Theorem 5.1, BT-dual will visit t. Suppose that BT-dual visits a node t. Thus, t's parent $p(t)$ is consistent in the dual problem. For any ordinary variable x, and for any two hidden variables c_i and c_j, where $c_i, c_j \in vars(p(t))$ and $x \in vars(c_i) \cap vars(c_j)$, $p(t)[c_i]$ and $p(t)[c_j]$ should have the same value on x. That means, in the hidden problem, $p(t)$ is consistent with all the ordinary variables. Therefore, FC-hidden will visit t. ∎

So in general the variable ordering for FC-hidden that instantiates all the ordinary variables is only bounded worse than any other variable ordering strategy for FC-hidden. From now on, we assume that FC-hidden will only instantiate the ordinary variables. Therefore, FC-hidden and FC-orig explore the same search tree consisting of all the ordinary variables.

5.3.2 FC-orig, FC-dual and FC-hidden

Example 5.9 *Consider a non-binary CSP with only one constraint over n Boolean variables,* $C(x_1, \ldots, x_n) = \{(0, \ldots, 0), (1, \ldots 1)\}$. *FC applied on this problem will explore* $O(2^n)$ *nodes and perform* $O(n2^n)$ *constraint checks to find all solutions. There are only two nodes in the dual search tree, representing two solutions of the problem.*

FC-hidden(instantiating all the ordinary variables first) will visit $O(n)$ nodes and perform $O(n)$ checks.

Theorem 5.18 *[7] There is a CSP instance in which FC-dual and FC-hidden are always exponentially better than FC-orig no matter what variable ordering is used in the original problem.*

Proof: It is true from the CSP in Example 5.9. ∎

Example 5.10 *Consider a CSP of $2n+1$ variables, $x_1, \ldots, x_{2n+1}$ and each variable has n values, $1, \ldots, n$. There are n constraints,*

$$\begin{aligned} C(x_1, x_2, x_{n+1}) &= \{x_1 = x_2\} \\ C(x_2, x_3, x_{n+2}) &= \{x_2 = x_3\} \\ &\ldots \\ C(x_{n-1}, x_n, x_{2n}) &= \{x_{n-1} = x_n\} \\ C(x_1, x_n, x_{2n+1}) &= \{x_1 \neq x_n\} \end{aligned}$$

This problem is insoluble because it enforces x_1 to be equal to $x_2, \ldots,$ and x_n and it also prohibits x_1 and x_n to have the same value. Note in each of the above constraints, variable x_{n+i} does not enforce anything but increase the arity and the number of tuples of the constraint. Given a static variable ordering, $x_1, \ldots, x_{2n+1}$, FC-orig and FC-hidden go along n paths, $\{(x_1 \leftarrow 0, \ldots, x_n \leftarrow 0\}$, ..., and $\{x_1 \leftarrow n, \ldots, x_n \leftarrow n\}$. At each stage, there is only one value consistent with all the future variables in the domain of the current variable. Thus FC-orig and FC-hidden visit $O(n)$ nodes to conclude that the problem is insoluble. However, by any variable ordering strategy, FC-dual has to instantiate at least $log(n) - 1$ dual variables to reach a dead-end. At each stage of FC-dual, it additionally instantiates one variable from $x_{n+1}, \ldots x_{2n+1}$, which has no influence on the failure. The best variable ordering strategy for FC-dual is to break the problem into two subproblems at each step in the backtrack search, where one of the two subproblems in insoluble. For example, FC-dual first branches on the dual variable corresponding to the constraint $C(x_{\frac{n}{2}}, x_{\frac{n}{2}+1}, x_{n+\frac{n}{2}})$. For each of the n^2 values in the domain of the dual variable, the current instantiation will result in one of the two subproblems to be insoluble, one consisting of the dual variables for the constraints $C(x_1, x_2, x_{n+1}), \ldots, C(x_{\frac{n}{2}-1}, x_{\frac{n}{2}}, x_{n+\frac{n}{2}-1})$, and the other consisting of the dual variables for the constraints $C(x_{\frac{n}{2}+1}, x_{\frac{n}{2}+2}, x_{n+\frac{n}{2}+1}), \ldots, C(x_1, x_n, x_{2n+1})$. Then the backtrack search will focus on the insoluble subproblem. So FC-dual has to explore at least $O(n^{log(n)-1})$ nodes.

Theorem 5.19 *[7] There is a CSP instance in which FC-orig and FC-hidden are always exponentially better than FC-dual no matter what variable ordering is used in the dual problem.*

Can FC-dual also be exponentially better than FC-hidden? In Example 5.9, we notice that FC-hidden visits $O(2n)$ times as many nodes as FC-dual does. This bound is generally true as we will show in the sequel. Given a variable ordering for FC-dual, $c_1, \ldots, c_m$, we can arrange the ordinary variables in the hidden problem in the same way as we have done in the case of BT-dual and BT-orig. That is, the instantiation of the dual variable c_i is equivalent to the instantiations of the ordinary variables $x_{i,1}, \ldots, x_{i,r_i}$, where $x_{i,j} \in vars(c_i)$ and $x_{i,j} \notin \bigcup_{k=1}^{i-1} vars(c_k)$. An example of such a variable ordering arrangement is shown in Figure 5.2. Under the above variable orderings, each ordinary variable $x_{i,j}$ in the hidden problem corresponds to a unique dual variable c_i in the dual problem. However, not all the dual variables have some correspondence in the ordinary variables. Therefore, in the hidden problem, x_{i,r_i} may not always be followed by $x_{i+1,1}$ (because $x_{i+1,1}$ may not exist).

Observation 5.4 *If a partial solution t over some ordinary variables in the hidden problem is consistent with all the hidden variables, t corresponds to a unique partial solution in the dual problem, $alldual(t)$, including all the dual variables c such that $vars(c) \subseteq vars(t)$. Because all the ordinary variables in the scheme of the hidden variable c have been instantiated, there is only one tuple $t[vars(c)]$ in the domain of c to be compatible with t, and thus $alldual(t)[c]$ is set to $t[vars(c)]$. Furthermore, if a node t at the level of x_{i,r_i} in the hidden search tree is consistent with all the variables, t corresponds to a unique node $dual(t)$ at the level of c_i in the dual search tree, where $dual(t) = alldual(t)[\{c_1, \ldots, c_i\}]$*[5].

For example, under the variable orderings shown in Figure 5.2, given a partial solution $t = \{(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$, $alldual(t)$ is $\{c(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$ and $dual(t)$ is also $\{c(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$. The condition that t is consistent with all the hidden variables cannot be relaxed. Otherwise, suppose t is not consistent with a hidden variable c_j for $\leq j \leq i$, *i.e.*, $t[vars(c_j)] \notin dom(c_j)$, then $t[vars(c_j)]$ is not a valid tuple in the domain of the dual variable c_j and thus $dual(t)$ is not a valid node in the dual search tree.

Lemma 5.20 *Under the above orderings, if a node t at the level of x_{i,r_i} in the hidden*

[5]From the construction of the variable ordering, $c_1, \ldots, c_i$ must have been instantiated in $alldual(t)$.

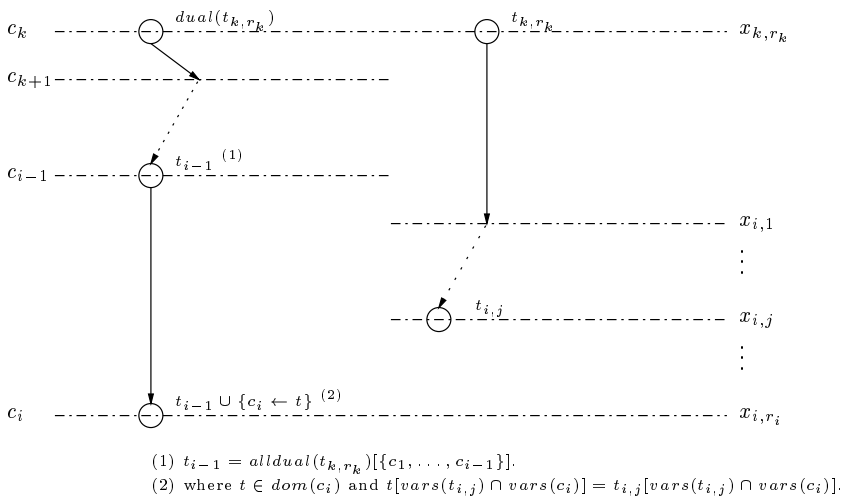


Figure 5.16: A node $t_{i,j}$ visited by FC-hidden at the level of $x_{i,j}$ in the hidden search tree corresponds to a unique node visited by FC-dual at the level of c_i in the dual search tree.

search tree is consistent with all the variables, then in the dual problem, $alldual(t)$ is consistent with all the dual variables. Furthermore, its correspondence $dual(t)$ at the level of c_i in the dual search tree is consistent with all the dual variables.

Proof: First, we prove that $alldual(t)$ is consistent in the dual problem. Suppose $alldual(t)$ does not satisfy the dual constraint between two dual variables c and c', then $t[vars(c)]$ is not compatible with $t[vars(c')]$. That is, there is an ordinary variable $x \in vars(c) \cap vars(c')$ such that $(t[vars(c)])[x] \neq (t[vars(c')])[x]$. This could not happen because both $(t[vars(c)])[x]$ and $(t[vars(c')])[x]$ are equal to $t[x]$. That is a contradiction. Thus, $alldual(t)$ is consistent. Now we prove that $alldual(t)$ is consistent with all the dual variables. Because in the hidden problem t is consistent with all the hidden variables, for any hidden variable c, there must exist a tuple t_c in the domain of c such that $t[vars(t) \cap vars(c)] = t_c[vars(t) \cap vars(c)]$. For each of the dual variables c' instantiated in $alldual(t)$, because $vars(c') \subseteq vars(t)$, thus $t[vars(c') \cap vars(c)] = t_c[vars(c') \cap vars(c)]$. Note that $alldual(t)[c']$ is set to be $t[vars(c')]$, that is, $\{c \leftarrow t_c\}$ is compatible with $alldual(t)[c']$. Therefore, $alldual(t)$ is consistent with the dual variable c and thus it is consistent with all the dual variables. Because $dual(t)$ is a subtuple of $alldual(t)$, thus $dual(t)$ is also consistent with all the dual variables. ∎

Thus the total number of the nodes at the level of x_{i,r_i} in the hidden search tree that are consistent with all the hidden variables is bounded by the total number of

the nodes at the level of c_i in the dual search tree that are consistent with all the dual variables.

Theorem 5.21 *Given a CSP and a variable ordering for the dual problem, there is a variable ordering on the hidden problem such that FC-hidden visits at most $O(rd)$ times nodes as many as FC-dual does.*

Proof: Now we consider a node $t_{i,j}$ at the level $x_{i,j}$ for $1 \leq j \leq r_i$ in the hidden search tree that is consistent with all the hidden variables. $t_{i,j}$'s ancestor t_{k,r_k} at the level of x_{k,r_k} is consistent with all the variables, where x_{k,r_k} is immediately followed by $x_{i,1}$. From Lemma 5.20, x_{k,r_k}'s correspondence $dual(x_{k,r_k})$ at the level of c_k in the dual search tree is consistent with all the dual variables. Thus FC-dual will visit $dual(x_{k,r_k})$ and extend it to the levels of $c_{k+1}, \ldots, c_{i-1}$ and c_i. Furthermore, because $alldual(t_{k,r_k})$ is consistent with all the dual variables, and for each of the dual variables $c_1, \ldots, c_{i-1}$, it must have been instantiated in $alldual(t_{k,r_k})$, the node $t_{i-1} = alldual(t_{k,r_k})[\{c_1, \ldots, c_{i-1}\}]$ at the level of c_{i-1} in the dual search tree must be consistent with all the dual variables. Thus FC-dual will visit t_{i-1} and extend it to the level of c_i. Because $t_{i,j}$ is consistent with all the hidden variables, there is a tuple t in the domain of c_i such that $t_{i,j}[vars(t_{i,j}) \cap vars(c_i)] = t[vars(t_{i,j}) \cap vars(c_i)]$. For each of the dual variables c_l where $1 \leq l \leq i-1$, note that $t_{i-1}[c_l] = alldual(t_{k,r_k})[c_l] = t_{r,r_k}[vars(c_l)] = t_{i,j}[vars(c_l)]$. That means, in the dual problem, $\{c_i \leftarrow t\}$ is compatible with each of the instantiations $\{c_l \leftarrow t_{i-1}[c_l]\}$, for $1 \leq l \leq i-1$. Thus FC-dual will visit node $t_{i-1} \cup \{c_i \leftarrow t\}$. Therefore $t_{i,j}$ corresponds a node visited by FC-dual at the level of c_i in the dual search tree, as shown in Figure 5.16. Furthermore, given two distinct nodes at the level of $x_{i,j}$ in the hidden search tree, because they have different values on the part $vars(t_{i,j}) \cap vars(c_i)$, they must be compatible with different tuples in the domain of c_i. Thus their correspondences in the dual search tree are different. Therefore, the total number of the nodes in the hidden search tree that are consistent with all the hidden variables is at most $O(r)$ times as many as the total number of the nodes visited by FC-dual in the dual search tree. From Theorem 5.13, FC-hidden visits a node only if its parent is consistent with all the variables, and each node may have at most d children. The total number of the nodes visited by FC-hidden in the hidden search tree is at most $O(rd)$ times as many as the total number of the nodes visited by FC-dual in the dual search tree. ∎

In Example 5.9, FC-hidden visits $O(rd)$ times nodes as many as FC-dual does.

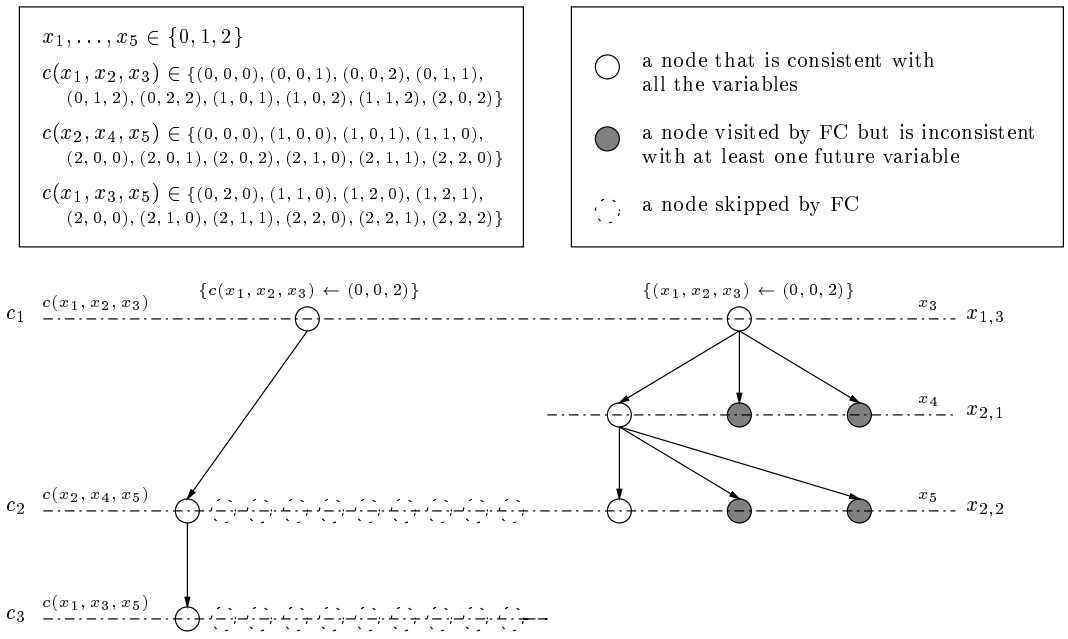


Figure 5.17: The comparison of FC-dual and FC-hidden to solve the CSP in Example 2.1.

Thus the above bound is tight. Thus, FC-hidden may be exponentially better than FC-dual and it can be only bounded worse than FC-dual.

Example 5.11 *We use FC-dual and FC-hidden to solve the CSP in Example 2.1, as shown in Figure 5.17. For example, in the hidden search tree, the node* $t_{2,1} = \{(x_1, x_2, x_3, x_4) \leftarrow (0, 0, 2, 0)\}$ *at the level of* $x_{2,1}$ *is consistent with all the variables in the hidden problem, its parent* $t_{1,3} = \{(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$ *at the level of* $x_{1,3}$ *is also consistent with all the variables. From Lemma 5.20,* $t_{1,3}$ *corresponds to node* $dual(t_{1,3}) = \{c(x_1, x_2, x_3) \leftarrow (0, 0, 2)\}$ *at the level of* c_1 *in the dual search tree, which is consistent with all the variables in the dual problem, and thus* $t_{2,1}$ *corresponds to node* $\{(c(x_1, x_2, x_3), c(x_2, x_4, x_5)) \leftarrow (0, 0, 2), (0, 0, 0)\}$ *visited by FC-dual at the level of* c_2 *in the dual search tree.*

5.3.3 FC+

Example 5.12 *Consider a non-binary CSP with n variables,* $x_1, \ldots x_n$ *and all the variables have the same domain,* $\{0, 1, 2\}$*. There are n+1 constraints,*

$$C(x_1, x_2) = \{(0, 0), (1, 1), (2, 2)\};$$
$$C(x_2, x_3) = \{(0, 0), (1, 1), (2, 2)\};$$

$$
\begin{aligned}
C(x_i, x_{i+1}) &= \{(0,0),(0,1),(1,0),(1,1),(2,2)\},\ i = 3 \cdots n-1; \\
C(x_1, x_2, x_n) &= \{(0,0,0),(1,1,1),(2,2,2)\}; \\
C(x_2, x_3, x_n) &= \{(0,0,1),(1,1,0),(2,2,2)\}.
\end{aligned}
$$

The only solution to this problem is $\{x_1 = 2, \ldots, x_n = 2\}$. *FC-orig is able to detect that every node instantiating* x_1, x_2, x_3 *with value 0 or 1 is incompatible with* x_n *and it will explore* $O(n)$ *nodes to find the solution. However, under the default variable ordering, FC-dual and FC-hidden are unable to detect such dead-end and have to explore* $O(2^n)$ *nodes.*

In the above example, if FC-hidden instantiates x_n right after the instantiations of x_1, x_2, and x_3, FC-hidden can detect a dead-end because neither of the values in the domain of x_n is consistent with the hidden variables for $C(x_1, x_2, x_n)$ and $C(x_2, x_3, x_n)$ simultaneously. Generally, given the execution of FC-orig, we can arrange the instantiation order for FC-hidden in the following way. If a node t in the hidden search tree is consistent with all the hidden variables, and t in the original search tree (note that FC-hidden only instantiates ordinary variables) is consistent with all the variables[6], FC-hidden extends t with the same variable that FC-orig used to extend t in its execution. Otherwise, if t in the hidden search tree is consistent with all the hidden variables, but t in the original search tree is not consistent with one future variable x, *i.e.*, for each of the values $a \in dom(x)$, $t \cup \{x \leftarrow a\}$ violates a constraint C, FC-hidden extends t to variable x. Then FC-hidden will detect none of the values in the domain of x is consistent with the corresponding hidden variable c. By such an arrangement, FC-hidden will visit at most $O(d)$ times as many nodes as FC-orig does. However, the above variable ordering for FC-hidden cannot be obtained without knowing the complete execution of FC-orig, because FC-hidden should know which of the future variables causes the dead-end in FC-orig. Without such an oracle, FC-hidden has to examine (instantiate) the forward checked variable in each of the forward checkable constraints. Thus FC-hidden has to instantiate more variables than FC-orig does at an early stage in the backtrack search, and because each of the forward checked variables may have more than one value in its domain, these extra instantiations by FC-hidden may cause exponential overhead over FC-orig.

However, there is a way to improve FC-hidden by doing more constraint propagation besides the forward checking. Following Bacchus and van Beek in [7], after forward checking prunes the domain of any hidden variable, we additionally prune

[6]Neither of the above two conditions implies the other.

the domains of any uninstantiated ordinary variables constrained by that hidden variable so as to remove values whose support has been lost. As usual we backtrack if there is a domain wipe out of any future variable. From another point of view, in Example 5.9, FC-orig does not forward check the domain of any future variable until $n-1$ variables have been instantiated. An extension to FC-orig is to make constraint checks as early as possible. Each time a variable has been instantiated, for each of the constraints involving this variable and for each of the uninstantiated variables involving this constraint, to remove all the values from its domain, which do not have a valid support in the constraint with respect to the current partial solution. In fact, these two approaches do the same work such as they explore exactly the same nodes. Bacchus and van Beek denote the new algorithm as *FC+* [7].

Because FC+ performs more checking than FC and FC-hidden, intuitively, FC+ should always visit no more nodes than FC-hidden and FC-orig.

Theorem 5.22 *Given a CSP and a variable ordering, FC+ always visits fewer nodes than FC-hidden and FC-orig.*

Proof: It is straightforward that FC-hidden and FC-orig visit all the nodes that FC+ visits. ∎

By Theorem 5.21, given a CSP instance P and any variable ordering for its dual problem, there is a variable ordering on the hidden problem such that FC+ visits at most $O(rd)$ times as many nodes as FC-dual does. This bound can be further improved if P is arc consistent.

Lemma 5.23 *If a non-binary CSP is arc consistent, FC+ applied on its hidden problem visits a node t only if t is consistent with all the variables.*

Proof: Suppose FC+ visits a node t and t is inconsistent with a hidden variable c. Let the current variable instantiated by t be x and $p(t)$ denote t's parent in the search tree. There must exist a hidden constraint between x and c, *i.e.*, $x \in vars(c)$, otherwise $p(t)$ is not consistent with c either and thus FC+ will not visit t. If $p(t)$ is the root of the search tree, then t instantiates only one variable x. Because the original CSP is arc consistent, from Theorem 4.3, the hidden problem is also arc consistent. Therefore, $\{x \leftarrow t[x]\}$ has a support in the hidden constraint between x and c and thus t can be extended to a consistent partial solution including c. That is a contradiction. Suppose $p(t)$ is not the root of the search tree, *i.e.*, $vars(p(t)) \neq \emptyset$. Because t is not consistent with c, for each of the tuples $t_c \in dom(c)$,

$t_c[vars(c) \cap vars(t)] \neq t[vars(c) \cap vars(t)]$. Thus, for each of the tuples $t_c \in dom(c)$ such that $t_c[vars(p(t)) \cap vars(t)] = t[vars(p(t)) \cap vars(t)]$, that is, t_c remains in the domain of c at the node $p(t)$, we have $t_c[x] \neq t[x]$. Therefore $\{x \leftarrow t[x]\}$ does not have a valid support in the domain of c at the node $p(t)$. $t[x]$ will be removed from the domain of x in the second phase of the consistency checks in FC+ (from the hidden variables to the ordinary variables) at the node $p(t)$ and FC+ will not visit t. That is a contradiction. ∎

Theorem 5.24 *Given an arc consistent CSP instance and any variable ordering for its dual problem, there is a variable ordering on the hidden problem such that FC+ visits at most $O(r)$ times as many nodes as FC-dual does.*

Proof: From Lemma 5.20, the number of the nodes visited by FC+ at each level of the hidden search tree is bounded by a factor $O(r)$ from the number of the nodes visited by FC-dual in the dual search tree. ∎

FC+ is an enhancement to FC-orig and FC-hidden to provide a tradeoff between a possibly exponential saving and a bounded more constraint checks. On the other hand, we can improve the original formulation by adding some redundant constraints to achieve the same effect. When FC+ visits a node t and let the current variable be x. For each constraint C involving x and for each uninstantiated variable x' involving C, FC+ will remove all the values from the domain of x' that do not have valid supports in C with respect to the current partial solution t. Let $S = (vars(t) \cap vars(C)) \cup \{x'\}$, The same pruning effect can be achieved by FC-orig if we add a redundant constraint $\pi_S C$, because $\pi_S C$ is forward checkable at the current node t and a valid support in $\pi_S C$ for a value in the domain of the forward checked variable x' can be extended to a valid support in C for that value. Thus, by adding some redundant constraints in the original problem, FC-orig can achieve the same improvement. Given a non-binary CSP P, for each constraint C, we add a redundant constraint $\pi_S C$ for each subset of the variables $S \subseteq vars(C)$. Let $proj(P)$ denote the resulting CSP and *FC-proj* denote FC applied on $proj(P)$.

Theorem 5.25 *Given a CSP P and any variable ordering, FC+ visits exactly the same nodes as FC-proj.*

Proof: Because we had added all possible projections of a constraint in $proj(P)$, FC-proj should perform more consistency checks than FC+ and visits no more nodes than FC+. Suppose both FC+ and FC-proj visit a node t and let the current variable be

x. We will show that if value a is not removed from the domain of an uninstantiated variable x' by FC+, the value cannot be removed from the domain by FC-proj either. Because a is not removed by FC+, for any constraint C involving both the current variable x and the uninstantiated variable x', there is a tuple t' in $rel(C)$ such that $t'[x'] = a$ and $t'[vars(C) \cap vars(t)] = t[vars(C) \cap vars(t)]$. Thus, in $proj(P)$, for each subset of variables $S \subset vars(C)$, where $x' \in S$ and $vars(S) \subseteq vars(t) \cup \{x'\}$, (that is, $\pi_S C$ is forward checkable at the current node and x' is the forward checked variable in $\pi_S C$), value a cannot be removed from the domain of x' when FC forward checks the constraint $\pi_S C$, because $\pi_S C$ allows the tuple $t'[S]$, where $(t'[S])[x'] = a$. Thus FC-proj makes no more domain prunings than FC+. Therefore, they visit exactly the same nodes in the backtrack search. ∎

As we can see in the above proof, not all the projections contribute to the domain prunings. If a dynamic variable ordering is used, we have to add all possible projections for each constraint to establish the above equivalence. For each of the constraints C, there are in total $2^{|vars(C)|} - 1$ possible projections over C. Thus adding all the projections is not practical for a problem having some high arity constraints. When a static variable ordering is used to solve the problem, suppose the variable are instantiated in the order $x_1, \ldots, x_n$, and for a constraint $C(x_{i_1}, \ldots, x_{i_r})$, where the order $x_{i_1}, \ldots, x_{i_r}$ conforms the above static variable ordering, we only need to add the projections of C over the sets of variables, $\{x_{i_1}\}$, $\{x_{i_1}, x_{i_2}\}$, $\ldots, \{x_{i_1}, x_{i_r}\}$, $\{x_{i_1}, x_{i_2}, x_{i_3}\}$, $\ldots, \{x_{i_1}, x_{i_2}, x_{i_r}\}$, $\ldots$, and $\{x_{i_1}, \ldots, x_{i_r}\}$. Thus given a static variable ordering, for each of the constraints C, the number of the projections is reduced to $O(|vars(C)|^2)$.

Now we can present a hierarchy of the above relations in Figure 5.18. In the above figure, we identify three relations between two identities (formulation+algorithm) $\mathcal{A}$ and $\mathcal{B}$. (1) $\mathcal{A}$ is never worse than $\mathcal{B}$. For example, FC+ is never worse than FC-orig and FC-hidden. Furthermore, $\mathcal{A}$ is equivalent to $\mathcal{B}$ if $\mathcal{A}$ is never worse than $\mathcal{B}$, and vice versa. For example, BT-dual is equivalent to FC-hidden if FC-hidden instantiates all the hidden variables first. Actually, BT-dual and FC-hidden under the above variable ordering visit exactly the same nodes. (2) $\mathcal{A}$ is bounded worse than $\mathcal{B}$. For example, FC-hidden visits at most $O(rd)$ times as many nodes as BT-dual does. (3) $\mathcal{A}$ may be exponentially worse than $\mathcal{B}$. This relation is usually established from a CSP instance in which there is a variable ordering for $\mathcal{B}$ such that $\mathcal{A}$ is exponentially worse than $\mathcal{B}$ no matter what variable ordering is used in $\mathcal{A}$.

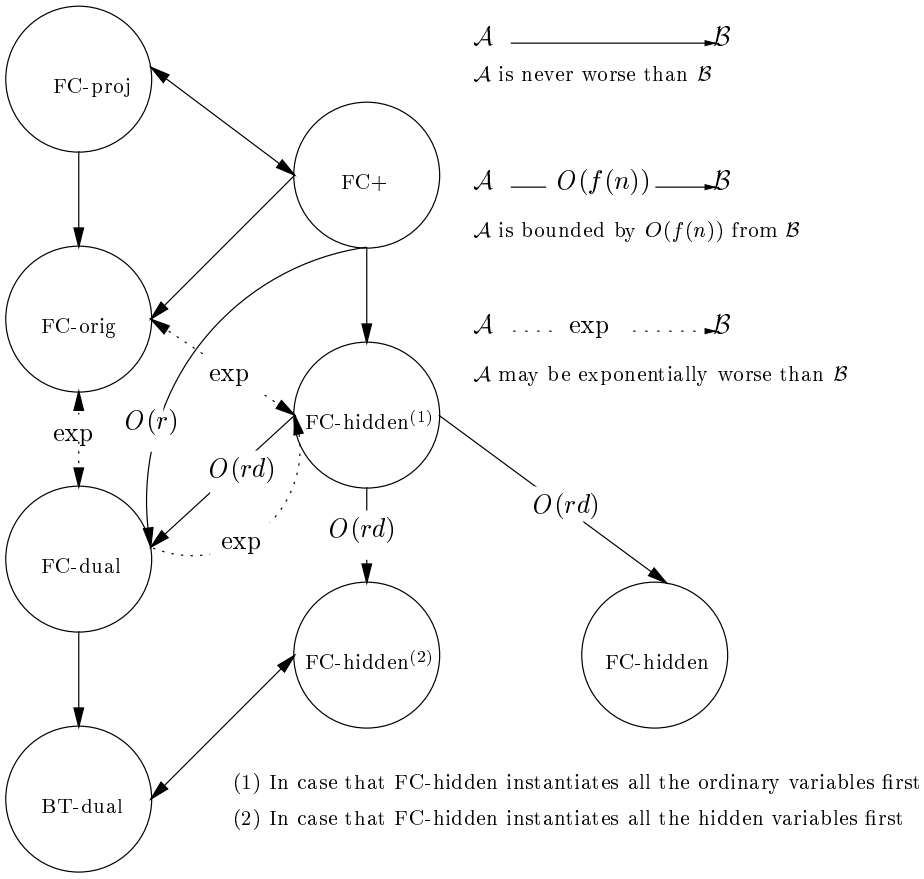


Figure 5.18: The relations between FC-orig, FC-dual, FC-hidden and FC+ .

5.4 Maintaining Arc Consistency Algorithm (GAC or MAC)

The maintaining arc consistency algorithm is called MAC in the CSP community. For general CSPs, we refer to the algorithm as *GAC*, namely, generalized maintaining arc consistency algorithm. At each node in the search tree, GAC achieves arc consistency on the subproblem induced by the current partial solution (see Definition 3.3). If, as a result, one of the uninstantiated variables experiences a domain wipe out, the instantiation of the current variables will lead to an insoluble subproblem and thus it should be retracted. If the induced subproblem is not empty after enforcing arc consistency, the instantiation to the current variable is accepted and GAC extends the current node to a future variable.

To recapitulate, the induced problem has exactly the same set of variables and the same set of constraints as the original problem, where the domain of each instantiated variable is restricted to contain one value.

Definition 5.2 *A partial solution t is* arc consistent *if the CSP induced by t is not empty after enforcing arc consistency.*

If the induced subproblem at the current node is not empty after enforcing arc consistency, the node is called an *arc consistent node.*

Lemma 5.26 *Given two partial solutions t and t' of a CSP P, where $t' \subseteq t$, if t is arc consistent, then t' is arc consistent.*

Proof: Because $P|_t$ has more restrictive domains than $P|_{t'}$, an arc consistent subdomain of $P|_t$ is also an arc consistent subdomain of $P|_{t'}$, and because $P|_t$ is not empty after achieving arc consistency, $P|_{t'}$ is not empty either after achieving arc consistency. Therefore t' is also arc consistent. ∎

Theorem 5.27 *GAC (MAC) visits a node t only if t's parent is arc consistent; GAC (MAC) visits a node t if t is arc consistent.*

Proof: Because GAC will not continue to node t if the CSP induced by t's parent is empty after achieving arc consistency, GAC visits t only if t's parent is arc consistent. We prove the second part by induction on the depth of the search tree. The hypothesis is trivial for the case of 1. Suppose it is true for the case of k, and suppose there is an arc consistent node t at level $k+1$. From Lemma 5.26, t's parent at level k

is arc consistent. Thus GAC visits its parent. Because t is arc consistent, the value assigned to the current variable by t cannot be removed from its domain when GAC enforces arc consistency on t's parent. As a consequence, GAC will visit t. ∎

A sufficient and necessary condition for GAC visiting a node t is: t's parent is arc consistent and the value assigned to the current variable by t has not been removed from its domain when enforcing arc consistency on t's parent. In the following discussion, we denote GAC applied on the original problem as *GAC-orig*, MAC applied on the dual problem as *MAC-dual*, and MAC applied on the hidden problem as *MAC-hidden*.

5.4.1 MAC-hidden

Similar to the case of FC-hidden, MAC-hidden does not need to instantiate all the variables in order to find a solution. Once MAC-hidden encounters a state in which for each of the hidden constraints, at least one of its variables has been instantiated, each of the uninstantiated variables has at most one value remaining in the domain and a solution can be assembled in a backtrack free manner. We will show in the sequel that a variable ordering for MAC-hidden that instantiates all the ordinary variables first is only bounded worse than any other variable ordering strategy.

Given a variable ordering in the hidden problem, $y_1, \ldots, y_l$, where y_i may be an ordinary variable or a hidden variable, in the same way as we have done in the case of FC-hidden, we can construct a new variable ordering that instantiates all the ordinary variables first. That is, if y_i is a hidden variable, in the new ordering, all the uninstantiated ordinary variables $x_{i,1}, \ldots, x_{i,r_i}$ in the scheme of y_i are chosen to be instantiated, otherwise if y_i is an ordinary variable and it has not been instantiated in the new ordering, y_i is chosen to be instantiated and we denote it as $x_{i,1}$. Note that in the new ordering, x_{i,r_i} may not always be immediately followed by $x_{i+1,1}$. We call the search tree explored by MAC-hidden under the original variable ordering as the *original hidden search tree*, and the search tree explored under the new variable ordering as the *new hidden search tree*.

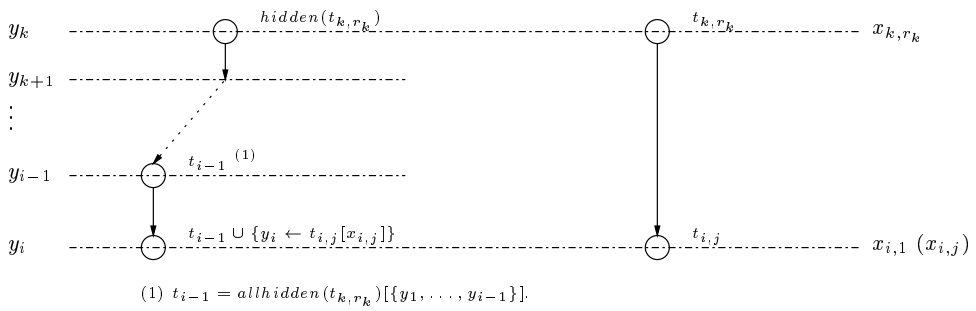
If a partial solution t over some ordinary variables is consistent with all the variables, t can be extended to a partial solution $allhidden(t)$ which additionally instantiates all the hidden variables c where $vars(c) \subseteq vars(t)$. Furthermore, if a node t at the level of x_{i,r_i} in the new hidden search tree is consistent with all the variables, t corresponds to a unique node $hidden(t)$ at the level of y_i in the original hidden search tree, where $hidden(t) = allhidden(t)[\{y_1, \ldots, y_i\}]$.

Lemma 5.28 *Given a CSP P and the above variable orderings, if a node t at the level of x_{i,r_i} in the new hidden search tree is arc consistent, then $allhidden(t)$ is arc consistent and the node $hidden(t)$ at the level of y_i in the original hidden search tree is also arc consistent. Furthermore, $ac(hidden(P)|_t) = ac(hidden(P)|_{allhidden(t)}) = ac(hidden(P)|_{hidden(t)})$.*

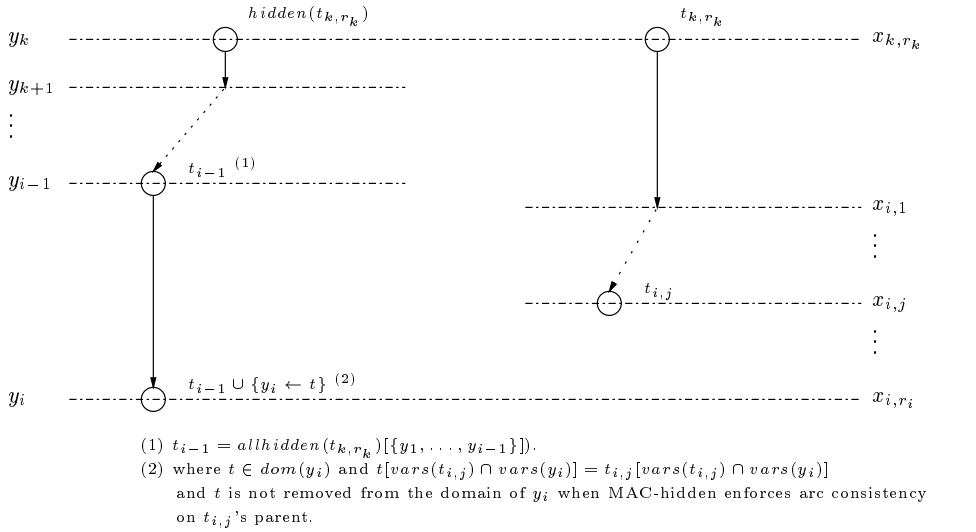
Proof: Because t is arc consistent, for each hidden variable c, there is a tuple t_c in the domain of c such that $t_c[vars(c) \cap vars(t)] = t[vars(c) \cap vars(t)]$ (otherwise, all the tuples in the domain of c will be removed when enforcing arc consistency on $hidden(P)|_t$). That is, t is consistent with c. Therefore t is consistent with all the variables, and thus $allhidden(t)$ and $hidden(t)$ do exist. Note that $hidden(P)|_t$ and $hidden(P)|_{allhidden(t)}$ have the same domains over the ordinary variables, whereas in $hidden(P)|_{allhidden(t)}$, for each of the hidden variables c such that $vars(c) \subseteq vars(t)$, the domain of c is set to have only one tuple $t[vars(c)]$. When enforcing arc consistency on $hidden(P)|_t$, for each of the hidden variables c such that $vars(c) \subseteq vars(t)$, and for each of the tuples t_c in the domain of c, if $t_c \neq t[vars(c)]$, t_c will be removed from the domain. Thus $ac(hidden(P)|_t)$ has the same domains as $ac(hidden(P)|_{allhidden(t)})$. Since $hidden(P)|_t$ is not empty after achieving arc consistency, $hidden(P)|_{allhidden(t)}$ is not empty either after achieving arc consistency and thus $allhidden(t)$ is arc consistent. Because $hidden(P)|_{allhidden(t)}$ has more restrictive domains than $hidden(P)|_{hidden(t)}$, $ac(hidden(P)|_{hidden(t)})$ is not empty either. Therefore, $hidden(t)$ is also arc consistent. For each of the ordinary variables x instantiated in t, either there is an ordinary variable y_i instantiated in $hidden(t)$ such that y_i is equal to x and y_i is instantiated with the value $t[x]$ in $hidden(t)$, or there is a hidden variable y_i instantiated in $hidden(t)$, such that $x \in vars(y_i)$ and $(hidden(t)[y_i])[x] = t[x]$. In either case, when achieving arc consistency on $hidden(P)|_{hidden(t)}$, the domain of x contains only one value $t[x]$. On the other hand, for each of the hidden variables c instantiated in $hidden(t)$, because $vars(c) \subseteq vars(t)$, when enforcing arc consistency on $hidden(P)|_t$, all the tuples t_c in the domain of c where $t_c \neq t[vars(c)]$ will be removed from its domain. Therefore $ac(hidden(P)|_{hidden(t)})$ has the same domains as $ac(hidden(P)|_t)$. We have, $ac(hidden(P)|_t) = ac(hidden(P)|_{allhidden(t)}) = ac(hidden(P)|_{hidden(t)})$. ∎

Lemma 5.29 *Given three partial solutions t_1, t_2 and t_3 of a CSP P, where $t_1 \subseteq t_2 \subseteq t_3$, if $ac(P|_{t_1}) = ac(P|_{t_3})$, then $ac(P|_{t_1}) = ac(P|_{t_2}) = ac(P|_{t_3})$.*

Proof: Because $t_1 \subseteq t_2 \subseteq t_3$, an arc consistent subdomain of $P|_{t_3}$ is also an arc consistent subdomain of $P|_{t_2}$, and an arc consistent subdomain of $P|_{t_2}$ is also an arc



(1)



(2)

Figure 5.19: A node $t_{i,j}$ visited by MAC-hidden at the level of $x_{i,j}$ in the new hidden search tree corresponds to a unique node visited by MAC-hidden at the level of y_i in the original hidden search tree.

consistent subdomain of $P|_{t_1}$. Because $ac(P|_{t_1})$ and $ac(P|_{t_3})$ have the same domains, we have $ac(P|_{t_1}) = ac(P|_{t_2}) = ac(P|_{t_3})$. ∎

Theorem 5.30 *Given a CSP and a variable ordering for the hidden problem, we can construct a variable ordering for the ordinary variables in the hidden problem, such that MAC-hidden under the new variable ordering visits at most $O(r)$ times as many nodes as it does under the original variable ordering.*

Proof: Suppose MAC-hidden visits a node $t_{i,j}$ at the level of $x_{i,j}$ in the new hidden search tree. Let x_{k,r_k} be immediately followed by $x_{i,1}$ in the new variable ordering

and let t_{k,r_k} denote $t_{i,j}$'s ancestor at the level of x_{k,r_k}.

(1) If y_i is an ordinary variable, then j is equal to 1 and $x_{i,j}$ is equal to y_i. From Theorem 5.27, t_{k,r_k} is arc consistent and its correspondence $hidden(t_{k,r_k})$ at the level of y_k in the original hidden search tree is also arc consistent. Furthermore, in the original hidden search tree, each of the variables $y_{k+1}, \ldots, y_{i-1}$ is instantiated in $allhidden(t_{k,r_k})$, let t_{i-1} denote the node $allhidden(t_{k,r_k})[\{y_1, \ldots, y_{i-1}\}]$, which is an extension of $hidden(t_{k,r_k})$. From Lemma 5.29, $ac(hidden(P)|_{t_{i-1}})$ and $ac(hidden(P)|_{hidden(t_{k,r_k})})$ have the same domains and thus $ac(hidden(P)|_{t_{i-1}})$ and $ac(hidden(P)|_{t_{k,r_k}})$ have the same domains. Thus, t_{i-1} at the level of y_{i-1} in the original hidden search tree is arc consistent, and MAC-hidden will visit t_{i-1} and extend it to the level of y_i. Because the value $t_{i,j}[x_{i,j}]$ is not removed from the domain of $x_{i,j}$ (*i.e.*, y_i) when enforcing arc consistency on the node t_{k,r_k}, $t_{i,j}[x_{i,j}]$ cannot be removed from the domain of y_i when enforcing arc consistency on the node t_{i-1}. Therefore, in the original hidden search tree, MAC-hidden will visit the node $t_{i-1} \cup \{y_i \leftarrow t_{i,j}[x_{i,j}]\}$ at the level of y_i. Let $t_{i,j}$ correspond to the node $t_{i-1} \cup \{y_i \leftarrow t_{i,j}[x_{i,j}]\}$, as shown in part (1) in Figure 5.19. Two distinct nodes at the level of $x_{i,j}$ in the new hidden search tree have different values over $x_{i,j}$, and thus they correspond to different nodes at the level of y_i in the original hidden search tree.

(2) Otherwise y_i is a hidden variable. The node t_{i-1} is introduced in the same way as the above. Because the value $t_{i,j}[x_{i,j}]$ is not removed from the domain of $x_{i,j}$ when MAC-hidden enforces arc consistency on $t_{i,j}$'s parent, there is an unpruned tuple t in the domain of y_i such that $t[vars(t_{i,j}) \cap vars(y_i)] = t_{i,j}[vars(t_{i,j}) \cap vars(y_i)]$. Thus, t cannot be removed from its domain when MAC-hidden enforces arc consistency on the node t_{k,r_k} at the level of x_{k,r_k} in the new hidden search tree (because t_{k,r_k} is an ancestor of $t_{i,j}$), and it cannot be pruned either when MAC-hidden enforces arc consistency on the node t_{i-1} in the original hidden search tree (because $ac(hidden(P)|_{t_{i-1}})$ and $ac(hidden(P)|_{t_{k,r_k}})$ have the same domains). Therefore, MAC-hidden will visit the node $t_{i-1} \cup \{y_i \leftarrow t\}$ at the level of y_i in the original hidden search tree. Let $t_{i,j}$ correspond to $t_{i-1} \cup \{y_i \leftarrow t\}$, as shown in part (2) in Figure 5.19. Because two distinct nodes at the level of $x_{i,j}$ have different supports from y_i, which do not agree on the part $vars(t_{i,j}) \cap vars(y_i)$, their correspondences at the level of y_i in the original hidden search tree are different. Thus the total number of the nodes visited by MAC-hidden in the new hidden search tree is bounded by a factor $O(r)$ from the total number of the nodes visited by MAC-hidden in the original hidden search tree. ∎

5.4.2 GAC-orig and MAC-hidden

We know from Theorem 4.3, arc consistency on the hidden problem is equivalent to arc consistency on the original problem. Because GAC-orig and MAC-hidden explore the same search tree, intuitively, they should visit exactly the same nodes.

Lemma 5.31 *Given a partial solution t of a CSP P, $ac(P|_t)$ is not empty if and only if $ac(hidden(P)|_t)$ is not empty. Furthermore, for each ordinary variable x, x has the same domain in $ac(P|_t)$ and $ac(hidden(P)|_t)$.*

Proof: From Theorem 4.3, $ac(P|_t)$ is not empty if and only if $ac(hidden(P|_t))$ is not empty and for each ordinary variable x, x has the same domain in $ac(P|_t)$ and $ac(hidden(P|_t))$. Note that $hidden(P)|_t$ and $hidden(P|_t)$ have the same domains for the ordinary variables. For each hidden variable c, the domain of c in $hidden(P)|_t$ contains all the tuples t_c in the corresponding constraint C, whereas its domain in $hidden(P|_t)$ contains only the tuples t_c in C such that $t_c[vars(t) \cap vars(c)] = t[vars(t) \cap vars(c)]$. However, for each of the tuples t_c in the domain of c in $hidden(P)|_t$, if $t_c[vars(t) \cap vars(c)] \neq t[vars(t) \cap vars(c)]$, t_c does not have a support from at least one of the ordinary variables $x \in vars(c)$, and thus t_c will be removed from the domain when achieving arc consistency on $hidden(P)|_t$. Therefore, $ac(hidden(P)|_t)$ and $ac(hidden(P|_t))$ have the same set of domains. Thus, $ac(P|_t)$ is not empty if and only if $ac(hidden(P)|_t)$ is not empty and they have the same domains for the ordinary variables. ∎

Theorem 5.32 *Given a CSP and any variable ordering, GAC-orig visits exactly the same nodes as MAC-hidden does.*

Proof: If a node t is arc consistent in the original problem, t is also arc consistent in the hidden problem. From Theorem 5.27, GAC visits a node t if and only if t's parent is arc consistent and the value assigned to the current variable by t has not be removed from its domain when enforcing arc consistency on t's parent. From Lemma 5.31, GAC-orig and MAC-hidden visit exactly the same nodes. ∎

5.4.3 GAC-orig and MAC-dual

The following examples show that GAC-orig (and MAC-hidden) may be exponentially better or worse than MAC-dual.

Example 5.13 *Consider a CSP of* $n(n+1)/2$ *variables,* $x_1, \ldots, x_n$, $x_{1,2}, \ldots, x_{1,n}$, $\ldots, x_{n-1,n}$, *and each variable has* $n-1$ *values,* $1, \ldots, n-1$. *There are* $n(n-1)/2$ *constraints,*

$$\begin{aligned} C(x_1, x_2, x_{1,2}) &= \{x_1 \neq x_2\}, \\ C(x_1, x_3, x_{1,3}) &= \{x_1 \neq x_3\}, \\ &\ldots \\ C(x_{n-1}, x_n, x_{n-1,n}) &= \{x_{n-1} \neq x_n.\} \end{aligned}$$

It is essentially a pigeon-hole problem except that we pad an extra variable $x_{i,j}$ *in each constraint. The pigeon-hole problem is insoluble but highly consistent[110]. By any variable ordering, GAC-orig has to instantiate* $n-2$ *variables to encounter a dead-end and it visits* $O(n^{\log n})$ *nodes to conclude the problem is insoluble. Because any two constraints overlap at most one ordinary variable, from Theorem 4.9, arc consistency on the dual representation is equivalent to arc consistency on the original problem. However, at each node of the dual search tree, MAC-dual has to additionally instantiate a variable* $x_{i,j}$, *which has no influence on the failure. So MAC-dual has to explore* $O(n^n)$ *nodes. Thus MAC-dual is exponentially worse than GAC-orig.*

In the above example, MAC-dual has the same pruning power as GAC-orig because each pair of the original constraints share at most one variable. However, MAC-dual has to do one useless instantiation at each node in the search tree. As a result, these extra instantiations cause MAC-dual to be exponentially worse than GAC-orig. The following example shows the converse: if two original constraints share more than one variable, arc consistency on the dual is stronger than arc consistency on the original problem, and MAC-dual may be exponentially better than GAC-orig.

Example 5.14 *Consider a CSP of* $4n+2$ *variables,* $x_1, \ldots, x_{4n+2}$ *and each variable has* n *values,* $1, \ldots, n$. *There are* $2n+1$ *constraints,*

$$\begin{aligned} C(x_1, x_2, x_3, x_4) &= \{(x_1 + x_2 \ mod\ 2) \neq (x_3 + x_4 \ mod\ 2)\} \\ C(x_3, x_4, x_5, x_6) &= \{(x_3 + x_4 \ mod\ 2) \neq (x_5 + x_6 \ mod\ 2)\} \\ &\ldots \\ C(x_{4n-1}, x_{4n}, x_{4n+1}, x_{4n+2}) &= \{(x_{4n-1} + x_{4n} \ mod\ 2) \neq (x_{4n+1} + x_{4n+2} \ mod\ 2)\} \\ C(x_{4n+1}, x_{4n+2}, x_1, x_2) &= \{(x_{4n+1} + x_{4n+2} \ mod\ 2) \neq (x_1 + x_2 \ mod\ 2)\} \end{aligned}$$

Because $(x_1 + x_2 \ mod\ 2) = 0$ *implies* $(x_3 + x_4 \ mod\ 2) = 1$, $(x_5 + x_6 \ mod\ 2) = 0$, ..., *and* $(x_{4n+1} + x_{4n+2} \ mod\ 2) = 0$ *and then* $(x_1 + x_2 \ mod\ 2) = 1$. *Thus the problem is insoluble. When enforcing arc consistency at a node in the original search tree, none will be removed from the domain of an ordinary variable unless the variable is the last uninstantiated variable in a constraint. The best variable ordering strategy in the original problem is to divide the problem in half by first branching on the variables* x_1, x_2, x_{2n+1} *and* x_{2n+2}. *Then we can branch on an insoluble subproblem consisting of* $x_3, \ldots, x_{2n}$, *or* $x_{2n+3}, \ldots, x_{4n+2}$. *By this divide-and-conquer approach, the maximum depth of the original search tree is about* $O(log(n))$ *and the total number of the nodes explored by GAC-orig is* $O(n^{log(n)})$. *In the dual problem, the dual constraints form a cycle in the constraint graph. Once a dual variable is instantiated, the cycle is broken so that the induced subproblem is empty after enforcing arc consistency. Thus MAC-dual only needs to instantiate one variable to conclude the problem is insoluble and it visits* $O(n^4)$ *nodes. Therefore, MAC-dual is exponentially better than GAC-orig.*

Theorem 5.33 *There is a CSP instance in which GAC-orig and MAC-hidden are always exponentially better than MAC-dual no matter what variable ordering is used in the dual problem.*

Proof: It is true from the CSP in Example 5.13. ∎

Theorem 5.34 *There is a CSP instance in which MAC-dual is always exponentially better than GAC-orig and MAC-hidden no matter what variable ordering strategies are used for them.*

Proof: It is true from the CSP in Example 5.14. ∎

From Theorem 4.9, we know that given a CSP where any two original constraints overlap on at most one ordinary variable, achieving arc consistency on the dual is equivalent to achieving arc consistency on the original problem. We will show that GAC-orig is only bounded worse than MAC-dual in such an instance. Given an ordering of the dual variables, $c_1, \ldots, c_m$, we can arrange the ordinary variables in the original problem in the same way as we have done in the case of BT-orig and BT-dual. That is, the instantiation to the dual variable c_i is equivalent to the instantiations of the ordinary variables $x_{i,1}, \ldots, x_{i,r_i}$, where $x_{i,j} \in vars(c_i)$ and $x_{i,j} \notin \bigcup_{k=1}^{i-1} vars(c_k)$. An example of such a variable ordering arrangement is shown in Figure 5.2. Under the above variable orderings, each ordinary variable $x_{i,j}$ in the original problem corresponds to a unique dual variable c_i in the dual problem. However, not all the

dual variables have some correspondences in the ordinary variables. Therefore, in the original problem, x_{i,r_i} may not be followed by $x_{i+1,1}$ (since $x_{i+1,1}$ may not exist).

Observation 5.5 *If a partial solution t in the original problem is consistent, t corresponds to a unique partial solution $alldual(t)$ in the dual problem including all the dual variables c such that $vars(c) \subseteq vars(t)$. Because t is consistent, for each dual variable c where $vars(c) \subseteq vars(t)$, the tuple $t[vars(c)]$ is included in the domain of c, and thus $alldual(t)[c]$ is set to be $t[vars(c)]$. Furthermore, if a node t at the level of x_{i,r_i} in the original search tree is consistent, t corresponds to a unique node $dual(t)$ at the level of c_i in the dual search tree, where $dual(t) = alldual(t)[\{c_1, \ldots, c_i\}]$.*

Note that in the comparison of FC-hidden and FC-dual, we have used the notations $alldual(t)$ and $dual(t)$, we still use these notations in comparing MAC-orig and MAC-dual because they are exactly the same under the two situations.

Lemma 5.35 *Given a CSP P in which for any two constraints C and C' of P, $vars(C) \cap vars(C')$ contains at most one ordinary variable, and given the above variable orderings, if a node t at the level of x_{i,r_i} in the original search tree is arc consistent, then $alldual(t)$ and $dual(t)$ are also arc consistent. Furthermore, for each original constraint C, if a tuple t_c is not (implicitly) removed from the constraint C when enforcing arc consistency on $P|_t$, the tuple t_c cannot be removed from the domain of the corresponding dual variable c when enforcing arc consistency on $dual(P)|_{alldual(t)}$ and $dual(P)|_{dual(t)}$.*

Proof: In the original problem, for each of the constraints C where $vars(C) \subseteq vars(t)$, and for each of the variables $x \in vars(C)$, the domain of x in $P|_t$ contains only one value $t[x]$. Because $ac(P|_t)$ is not empty, thus $t[vars(C)] \in rel(C)$. So t is consistent in the original problem. Thus $alldual(t)$ and $dual(t)$ do exist. Because $ac(P|_t)$ is not empty, from Theorem 4.7, $ac(dual(P|_t))$ is not empty. Note that $dual(P|_t)$ and $dual(P)|_{alldual(t)}$ have the same domain for each dual variable c, where $vars(c) \subseteq vars(t)$ (the domain of c contains only one tuple $t[vars(c)]$), or $vars(c) \cap vars(t) = \emptyset$ (the domain of c contains all the tuples in the corresponding constraint). For each dual variable c, where $vars(c) \cap vars(t) \neq \emptyset$ and $vars(c) \not\subseteq vars(t)$, that means, c is not instantiated in $alldual(t)$, but it is constrained with at least one of the dual variables instantiated in $alldual(t)$. The domain of c in $dual(P)|_{alldual(t)}$ contains all the tuples in the corresponding constraint, whereas its domain is $dual(P|_t)$ only contains those tuples t_c in the corresponding constraint such that $t_c[vars(c) \cap vars(t)] = t[vars(c) \cap vars(t)]$. However, for each of the tuples t_c in the domain of

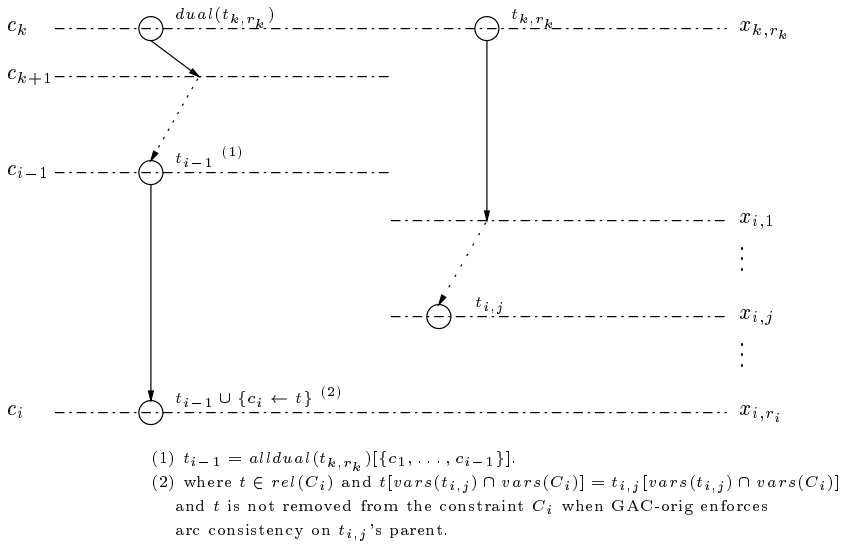


Figure 5.20: A node $t_{i,j}$ visited by GAC-orig at the level of $x_{i,j}$ in the original search tree corresponds to a unique node visited by MAC-dual at the level of c_i in the dual search tree.

c in $dual(P)|_{alldual(t)}$ such that $t_c[vars(c) \cap vars(t)] \neq t[vars(c) \cap vars(t)]$, because $vars(t) = \bigcup_{k=1}^{i} vars(c_k)$ and $\{c_1, \ldots, c_i\} \subseteq alldual(t)$, there is one dual variable c_j for $1 \leq j \leq i$, such that $\{c \leftarrow t_c\}$ is not compatible with $\{c_j \leftarrow t[vars(c_j)]\}$. Because $t[vars(c_j)]$ is the only tuple in the domain of c_j in $dual(P)|_{alldual(t)}$, t_c will be removed from its domain when enforcing arc consistency on the dual constraint between c and c_j. Thus $ac(dual(P|_t))$ and $ac(dual(P)|_{alldual(t)})$ have exactly the same domains for the dual variables. Because $ac(P|_t)$ is not empty, $ac(dual(P)|_{alldual(t)})$ is not empty either. Therefore, $alldual(t)$ is arc consistent. Furthermore, from Theorem 4.9, $dual(ac(P|_t)) = ac(dual(P)|_{alldual(t)})$. That is, if a tuple t_c is not (implicitly) removed from an original constraint C in $ac(P|_t)$, t_c cannot be removed from the domain of the corresponding dual variable c in $ac(dual(P)|_{alldual(t)})$. Because $dual(t)$ is a subtuple of $alldual(t)$, thus an arc consistent subdomain of $dual(P)|_{alldualt}$ is also an arc consistent subdomain of $dual(P)|_{dual(t)}$. Therefore, $dual(t)$ is arc consistent and if a tuple t_c is not implicitly removed from an original constraint C in $ac(P|_t)$, t_c cannot be removed from the domain of the corresponding dual variable c in $ac(dual(P)|_{dual(t)})$. ∎

Thus the total number of the arc consistent nodes at the level of x_{i,r_i} in the original search tree is bounded by the total number of the arc consistent nodes at the level of c_i in the dual search tree.

Theorem 5.36 *Let P be a CSP instance such that for any two constraints C and C' of P, $vars(C) \cap vars(C')$ contains at most one ordinary variable. Given any variable ordering for its dual representation, there is a variable ordering on the original problem such that GAC-orig visits at most $O(r)$ times as many nodes as MAC-dual does.*

Proof: Suppose GAC-orig visits a node $t_{i,j}$ at the level of $x_{i,j}$ in the original search tree, let t_{k,r_k} denote $t_{i,j}$'s ancestor at the level of x_{k,r_k}, where x_{k,r_k} is followed by $x_{i,1}$ in the variable ordering for the original problem. We know that t_{k,r_k} is arc consistent. From Lemma 5.35, its correspondence $dual(t_{k,r_k})$ is an arc consistent node at the level of c_k in the dual search tree. Thus MAC-dual will visit $dual(t_{k,r_k})$ and extend it to the levels of $c_{k+1}, \ldots, c_{i-1}$ and c_i. Furthermore, because $alldual(t_{k,r_k})$ is arc consistent, and each of the variables $c_1, \ldots, c_{i-1}$ must have been instantiated in $alldual(t_{k,r_k})$, the node $t_{i-1} = alldual(t_{k,r_k})[\{c_1, \ldots, c_{i-1}\}]$ is an arc consistent node at the level of c_{i-1} in the dual search tree. Thus MAC-dual will visit t_{i-1} and extend it to the level of c_i. Because the value $t_{i,j}[x_{i,j}]$ was not removed from the domain of $x_{i,j}$ when GAC-orig enforces arc consistency on $t_{i,j}$'s parent in the original problem, there is a tuple $t \in rel(C_i)$ such that $t[vars(t_{i,j}) \cap vars(C_i)] = t_{i,j}[vars(t_{i,j}) \cap vars(C_i)]$ and t is not (implicitly) removed the constraint C_i when enforcing arc consistency on $t_{i,j}$'s parent, *i.e.*, t is a valid support for $\{x_{i,j} \leftarrow t_{i,j}[x_{i,j}]\}$. Thus t will not be (implicitly) removed from the constraint C_i when GAC-orig enforces arc consistency on the node t_{k,k_r} (because t_{k,r_k} is an ancestor of $t_{i,j}$). From Lemma 5.35, t cannot be removed from the domain of the dual variable c_i in $ac(dual(P)|_{alldual(t_{k,r_k})})$. Because t_{i-1} is a subtuple of $alldual(t_{k,r_k})$, thus t cannot be removed from the domain of c_i in $ac(dual(P)|_{t_{i-1}})$. Therefore, MAC-dual will visit the node $t_{i-1} \cup \{c_i \leftarrow t\}$ at the level of c_i in the dual search tree. Let $t_{i,j}$ correspond to $t_{i-1} \cup \{c_i \leftarrow t\}$, as shown in Figure 5.20. Given two distinct nodes visited by GAC-orig at the level of $x_{i,j}$ in the original search tree, because they have the different instantiations on the part $vars(t_{i,j}) \cap vars(C_i)$, their correspondences at the level of c_i in the dual search tree are different. Therefore, the total number of the nodes visited by GAC-orig in the original search tree is bounded by a factor $O(r)$ from the total number of the nodes visited by MAC-dual in the dual search tree. ∎

5.4.4 Combined Formulation

MAC-dual may be exponentially better because it enforces a stronger consistency on the dual representation and MAC-hidden may be exponentially better because

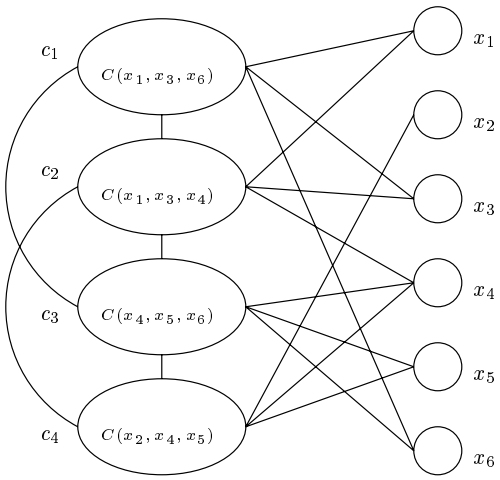


Figure 5.21: The combined formulation of the CSP in Example 1.1.

it makes less instantiations at each stage during the backtrack search. We observe that the advantages of the dual representation and the hidden representation can be combined into a new problem formulation.

Definition 5.3 (combined representation) *Given a CSP instance* $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, *its combined representation* $comb(P) = (\mathcal{V}^{comb(P)}, \mathcal{D}^{comb(P)}, \mathcal{C}^{comb(P)})$ *is defined as:*

- $\mathcal{V}^{comb(P)} = \mathcal{V}^{hidden(P)}$, i.e., *the set of variables consists of the ordinary variables from the original problem and the hidden variables corresponding to the constraints in the original problem,*
- $\mathcal{D}^{comb(P)} = \mathcal{D}^{hidden(P)}$, i.e., *the domain of an ordinary variable is the same as it in the original problem and the domain of a hidden variable consists of all the tuples in its corresponding constraint,*
- $\mathcal{C}^{comb(P)} = \mathcal{C}^{hidden(P)} \cup \mathcal{C}^{dual(P)}$. *The set of constraints includes all the hidden constraints in the hidden representation and all the dual constraints in the dual representation.*

The combined representation of the the CSP in Example 1.1 is shown in Figure 5.21, which is essentially the combination of the hidden representation as shown in Figure 4.2 and the dual representation as shown in Figure 4.1. Note that the combined representation is a binary CSP, in which an ordinary variable only constrains with the

hidden variables, whereas a hidden variable may have constraints with the ordinary variables and other hidden variables.

In the following, we denote MAC applied on the combined representation as *MAC-comb*. Similar to the case of MAC-hidden, we assume that all the ordinary variables are instantiated first in the combined representation. Thus, MAC-comb explores the same search tree as GAC-orig does.

Theorem 5.37 *Given a CSP P, $ac(comb(P))$ is not empty if and only if $ac(dual(P))$ is not empty. Furthermore, a hidden variable has the same domain in $ac(comb(P))$ and $ac(dual(P))$.*

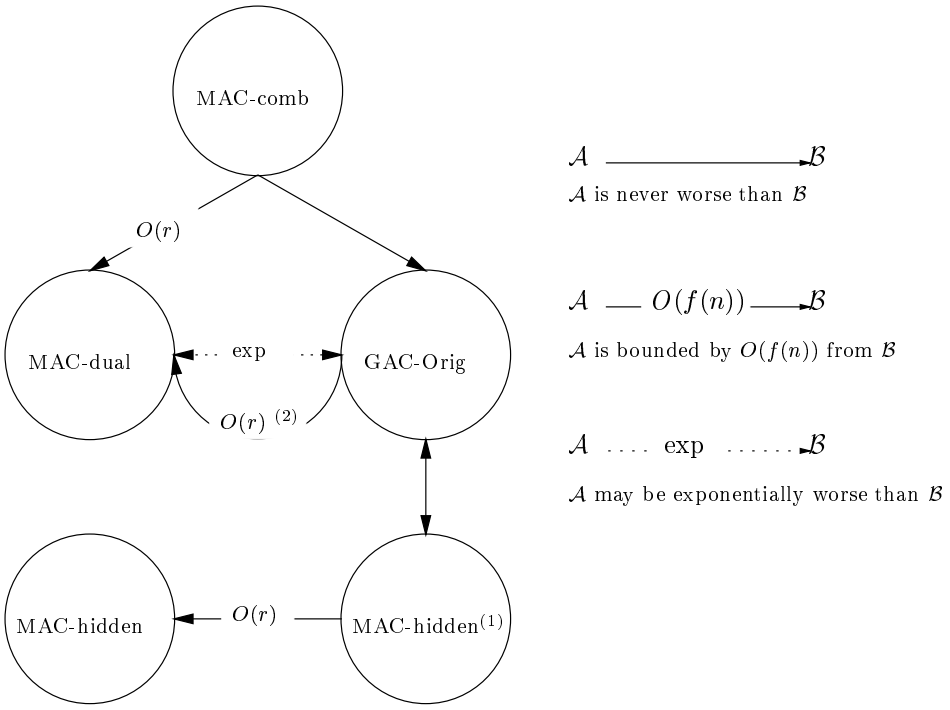
Proof: Given a CSP P, because the dual problem is a subproblem in the combined representation, if $ac(dual(P))$ is empty, then $ac(comb(P))$ is empty too. On the other hand, if $ac(dual(P))$ is not empty, from $ac(dual(P))$, we can construct an arc consistent subdomain for the original problem, $\mathcal{D}^{dualac(P)}$ (see page 97). It is easy to verify that $\mathcal{D}^{dualac(P)} \cup \mathcal{D}^{ac(dual(P))}$ is an arc consistent subdomain for $comb(P)$. ∎

Theorem 5.38 *Given a CSP and a variable ordering for the original problem, there is a variable ordering for the combined representation such that MAC-comb always visits no more nodes than GAC-orig does. On the other hand, there exists a CSP instance in which MAC-comb is exponentially better than GAC-orig no matter what variable ordering is used in the original problem.*

Proof: Because MAC-comb and GAC-orig explore the same search tree and at each node in the search tree, MAC-comb enforces a stronger consistency than GAC-orig does, GAC-orig visits all the nodes that MAC-comb visits. Since MAC-comb enforces a more powerful consistency, sometimes this will be paid off. For example, GAC-orig is exponentially worse than MAC-comb when solving the CSP in Example 5.14.

Theorem 5.39 *Given a CSP and a variable ordering for the dual problem, there is a variable ordering for the combined representation such that MAC-comb visits at most $O(r)$ times as many nodes as MAC-dual visits. On the other hand, there exists a CSP instance in which MAC-dual is exponentially worse than MAC-comb no matter what variable ordering is used in the dual problem.*

Proof: Since arc consistency on the combined representation is equivalent to arc consistency on the dual problem, in the same way as we have done in proving Theorem 5.36, we can show that MAC-comb is only bounded worse than MAC-dual.



(1) In the case that MAC-hidden instantiated all the ordinary variables first.

(2) In the case that each pair of the original constraints share at most one variable.

Figure 5.22: The relations between GAC-orig, MAC-dual, MAC-hidden and MAC-comb.

Because MAC-comb makes a weaker instantiation at each node in the search tree than MAC-dual does, MAC-comb can avoid some useless instantiations that MAC-dual has to commit. For example, MAC-comb is exponentially better than MAC-dual when solving the problem in Example 5.13. ∎

We summarize the relations between GAC (MAC) on different formulations in Figure 5.22. MAC-comb is on the top in the hierarchy in terms of the size of the search tree, but it performs more work at each node and in practice it may not have the best run time performance.

5.5 Discussion

In the above, we have theoretically studied the relations between the original problem, the dual problem and the hidden problem with respect to the selected backtracking algorithms. The results are summarized in Figure 5.1. Furthermore, given three

algorithm+formulation combinations $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ in the figure, we can make the following inferences:

- The relations $\mathcal{A}$ may be exponentially worse than $\mathcal{B}$ and $\mathcal{A}$ is bounded worse than $\mathcal{B}$, cannot hold simultaneously. For example, FC-hidden is bounded worse than FC-dual, thus there is no such instance to show that FC-dual can be exponentially better than FC-hidden.

- If $\mathcal{A}$ is polynomially bounded worse than $\mathcal{B}$, and $\mathcal{B}$ is polynomially bounded worse than $\mathcal{C}$, then $\mathcal{A}$ is polynomially bounded worse than $\mathcal{C}$. For example, because FC-proj and FC+ always visit the same nodes and FC+ is bounded worse than FC-dual, thus FC-proj is only bounded worse than FC-dual.

- If $\mathcal{A}$ is bounded worse than $\mathcal{B}$, and $\mathcal{C}$ may be exponentially worse than $\mathcal{B}$, then $\mathcal{C}$ may be exponentially worse than $\mathcal{A}$. For example, MAC-hidden always visits fewer nodes than FC-hidden, which itself is bounded worse than FC-dual. Then we can conclude that MAC-hidden is only bounded worse than FC-dual, whereas MAC-hidden may be exponentially better than FC-dual.

However, if $\mathcal{A}$ may be exponentially worse than $\mathcal{B}$, and $\mathcal{B}$ may be exponentially worse than $\mathcal{C}$, we cannot conjecture that $\mathcal{A}$ may be exponentially worse than $\mathcal{C}$. For example, FC-hidden may be exponentially worse than FC-orig, and FC-orig may be exponentially worse than FC-dual, but FC-hidden is only bounded worse than FC-dual.

Although the above relations are established in terms of the number of the nodes visited by the algorithm, they are also valid in the case that the number of constraint checks performed is considered if the number of constraint checks performed by the algorithm at each node in the search tree can be bounded by a polynomial [7]. For example, if $\mathcal{A}$ may be exponentially worse than $\mathcal{B}$ in terms of the number of nodes visited by the algorithms, it still holds that $\mathcal{A}$ is exponentially worse than $\mathcal{B}$ in terms of the constraint checks, because the backtracking algorithm performs a polynomial number of constraint checks at each node in the search tree. Otherwise, if $\mathcal{A}$ is bounded worse than $\mathcal{B}$ in term of the number of the nodes visited by the algorithms, the number of the constraint checks performed in $\mathcal{A}$ is also bounded in a polynomial factor by the number of the constraints checks performed in $\mathcal{B}$. Furthermore, due to the special properties of the dual and hidden transformations, some general methods

[7] However, the worst case complexity of achieving arc consistency on general CSPs is not always bounded by a polynomial.

to exploit such properties are available to speed up the constraint checking and constraint propagation in the dual problem and the hidden problem. A measure based on the constraint checks performed by the algorithm can hardly reflect those efforts.

Nevertheless, the above relations are the worst case analyses and in practice they do not precisely reflect the actual performances. Our objective is to provide some general guidelines in determining whether or under which conditions the dual or hidden transformation should be applied on a non-binary CSP. For example, if $\mathcal{A}$ is just bounded worse than $\mathcal{B}$, but it may be exponentially better than $\mathcal{B}$, we are ensured that the performance of $\mathcal{A}$ could not be much worse than the performance of $\mathcal{B}$, but $\mathcal{A}$ has the potential to provide a dramatic improvement over $\mathcal{B}$. Thus, if we are solving a large problem, $\mathcal{A}$ is preferred in the hope that $\mathcal{A}$ can provide exponential savings over $\mathcal{B}$ and in the worst case, it cannot lose too much.

For example, consider a crossword puzzle problem. We know that there exists three possible formulations for the problem, an original formulation in which each letter is represented by a variable and each word is represented by a non-binary constraint, the dual formulation in which each word is represented by a variable and the binary constraints specify that two words agree on their intersecting letter, and the hidden formulation which represents both the letters and the words by variables. There are few constraints in the original formulation, *i.e.*, the CSP is sparse, and each of the constraints is very tight compared to all possible combinations of 26 letters. If we apply FC to solve the problem, intuitively, the original formulation would not be a good choice because a non-binary constraint can be forward checked only if all but one of its variables has been instantiated. From Theorem 5.21, we know that FC-hidden is only bounded worse than FC-dual, thus the hidden formulation is at least comparable to the dual formulation. If MAC is applied to solve the problem, from Theorem 5.32, we know that GAC-orig visits the same nodes as MAC-hidden does if they use the same static variable ordering. Because each pair of constraints in the original formulation share at most one variable, from Theorem 5.36, we know that GAC-orig is only bounded worse than MAC-dual. Thus, the original formulation and the hidden formulation are the winner in the case of MAC. Furthermore, the efficiency of constraint propagation in all the three formulations can be improved by the use of some propagators. For example, in the original formulation, to find a support (for a revised value) in a non-binary constraint, the generic method is to list all possible combinations (*e.g.*, 26^{10}) of the values in the current domains of the variables in the constraint. Because the constraint is very tight, it is rare to encounter a valid support in the list. However, we can simply go through the dictionary and

check each word whether it is a valid support for that value. Since arc consistency on the dual formulation does not have more pruning power than the one on the original formulation and the hidden formulation, it is not worthwhile to combine the dual and hidden together and apply MAC on the combined formulation.

5.6 Summary

In this chapter, we examined theoretically how well some backtracking algorithms perform on a non-binary CSP and its dual and hidden transformations. Given two algorithm+formulation combinations $\mathcal{A}$ and $\mathcal{B}$, we identify one of two mutually exclusive relations between $\mathcal{A}$ and $\mathcal{B}$, either $\mathcal{A}$ may be exponentially worse than $\mathcal{B}$, or $\mathcal{A}$ can always be (polynomially) bounded worse than $\mathcal{B}$. We mean that $\mathcal{A}$ may be exponentially worse than $\mathcal{B}$ if there is a CSP instance and a variable ordering for $\mathcal{B}$ such that the performance of the algorithm on $\mathcal{A}$ is exponentially worse than its performance on $\mathcal{B}$ no matter what variable ordering is used in $\mathcal{A}$, and we mean $\mathcal{A}$ is bounded worse than $\mathcal{B}$ if for any CSP instance and for any variable ordering in $\mathcal{B}$, we can figure out a variable ordering for $\mathcal{A}$ such that the performance of the algorithm on $\mathcal{A}$ is bounded by a polynomial factor from its performance on $\mathcal{B}$.

For the chronological backtracking algorithm, BT-orig may be exponentially worse than BT-dual and BT-hidden, and BT-dual and BT-hidden may be exponentially worse than BT-orig. However, BT-dual will always visit no more nodes than BT-hidden does, and BT-hidden can visit at most $O(rd)$ times as many nodes as BT-dual does. Moreover, if the maximum arity of the constraints in the original problem is bounded by a constant r, BT-orig visits at most $O(d^{r+1})$ as many nodes as BT-dual or BT-hidden does, and if the maximum number of the tuples in the constraints of the original problem is bounded by a constant M, BT-dual or BT-hidden can visit at most $O(mdM)$ times as many nodes as BT-orig does.

For the forward checking algorithm, FC-dual may be exponentially worse than FC-orig and FC-hidden, and FC-orig may be exponentially worse than FC-dual and FC-hidden. However, FC-hidden can visit at most $O(rd)$ times as many nodes as FC-dual does. Both FC-orig and FC-hidden can be improved by doing more constraint checks at each node in the search tree. For example, Bacchus and van Beek introduce an algorithm called FC+ as an improvement to FC-orig and FC-hidden [7]. FC+ never visits more nodes than FC-orig or FC-hidden does, and FC+ can visit at most $O(r)$ times as many nodes as FC-dual (if the original problem is arc consistent). Furthermore, the original formulation can be improved by adding all the projections

of the non-binary constraints. FC applied on the new formulation is called FC-proj, and we have shown that FC-proj and FC+ visit exactly the same nodes. That means, improving the algorithm and improving the formulation may have the same effect.

For the maintaining arc consistency algorithm, GAC-orig and MAC-hidden visit exactly the same nodes, while MAC-dual may be exponentially worse than GAC-orig and MAC-hidden because MAC-dual makes more instantiations at each node of the search tree, and GAC-orig and MAC-hidden may be exponentially worse than MAC-dual because MAC-dual enforces a stronger consistency in the backtrack search than GAC-orig or MAC-hidden does. If any pair of the constraints in the original problem have at most one common variable, we know that arc consistency on the dual is equivalent to arc consistency on the original and the hidden problem. In that case, we can show that GAC-orig and MAC-hidden visit at most $O(r)$ times as many nodes as MAC-dual does. The dual and hidden problem can be combined into a new binary formulation and we denote MAC applied on the new problem as MAC-comb. MAC-comb never visits more nodes than GAC-orig or MAC-hidden does, and it visits at most $O(r)$ times as many nodes as MAC-dual does.

Although all the above relations are based on the number of the nodes visited by a backtracking algorithm, they are still valid if the number of the constraint checks is considered. Our study can provide some general guidelines to determine whether or under which conditions the dual or hidden transformation can be applied on a non-binary CSP.

Chapter 6

Future Work and Conclusion

6.1 Future Work

In Chapter 3, we present two seemingly contradictory scenarios, one is that an algorithm doing more looking ahead cannot benefit more from a look-back enhancement, and the other is that GAC can still be (significantly) improved by the use of CBJ. The first one is of theoretical interest as we have shown that the use of a dynamic variable ordering or maintaining strong k-consistency will weaken the effects of the backjumping technique. The second one has practical value as our experiments show that GAC-CBJ outperforms GAC by orders of magnitude on some real world problems. A missed part in the picture is the linkage between the theoretical justifications and the empirical observations; *i.e.*, from a practical point of view, we are more interested in using these theoretical results to explain and predict whether or under which condition a look-ahead algorithm will be improved from a backjumping enhancement. For example, FC-CBJ is known to be better than FC on a wide range of problem domains, but the improvement of GAC-CBJ over GAC can only be observed in sparse random CSPs and some real world problems. The theoretical results say that if a CSP is highly consistent, CBJ can hardly generate effective backjumps. One possible approach is to find a way to characterize the degree of consistency in a CSP, similar to the "constrainedness" property used in phase transition study [55]. For example, because a dense CSP usually has a higher level of consistency than a sparse CSP, GAC-CBJ shows improvement on sparse problems. Adding redundant constraints to a CSP has the effect of achieving some degree of consistency and thus increases the consistency level of the formulation. Therefore, the benefit of CBJ will be diminished by the use of redundant constraints. One solution is to find a parameter indicating the consistency level of a CSP, and tie each of the look-ahead algorithms

with a threshold point of the parameter, where to solve a CSP with a consistency level above the point, the CBJ enhancement is more likely useless to the look-ahead algorithm, and it will bring improvement on instances with consistency level below the threshold point.

Our work in the comparison of the dual and hidden transformations can be extended in several directions. To study the consistency properties of non-binary CSPs, the dual transformation provides a good starting point. As we know, strong k-consistency is not so useful for non-binary CSPs. One obvious paradox is that if all the constraints in a CSP have arity greater than k, the CSP is strongly k-consistent. Dechter and van Beek have proposed the concept of *relational k-consistency* as the generalization of k-consistency for non-binary CSPs [39]. However, relational consistency has some serious drawbacks to be used in practice [1]. As we know, arc consistency on the dual problem is stronger than arc consistency on the original problem, and in fact it is even stronger than relational arc consistency. Therefore, arc consistency on the dual transformation will induce a new consistency property on the original non-binary problem. For example, we can call it *dual arc consistency*. Subsequently, we can define more consistency properties, such as *dual k-consistency*, *dual (i,j)-consistency*, *dual neighborhood inverse consistency*, and so on, to enrich the family of consistencies for general CSPs.

In Chapter 5, we have thoroughly compared the performance of several backtracking algorithms on the three possible formulations for any problem. However, a "pure" form of the dual or hidden transformation is rarely used in modeling a problem. Instead, they are often used in the form of partial conversions, and combined with other modeling techniques, such as exploiting meta values.

One drawback of the dual and hidden transformations is that they are only applicable to sparse CSPs. For a dense CSP, if every constraint is transformed into a dual variable or a hidden variable, the transformation will have too many variables to be solved by backtracking algorithms. A partial conversion means a subset of the constraints in an original formulation become dual variables, or a subset of the variables in a constraint are aggregated into a hidden variable. For example, in temporal reasoning, an interval-based representation of temporal information can sometimes be viewed as a partial (dual) conversion of a point-base representation (see Figure 4.4), which can often be processed very quickly. One extension of our theoretical results is to formalize various partial conversions and evaluate how they will affect

[1] For example, even the complexity of achieving relational arc consistency in a non-binary CSP may be exponential if the maximum arity of the constraints is not bounded.

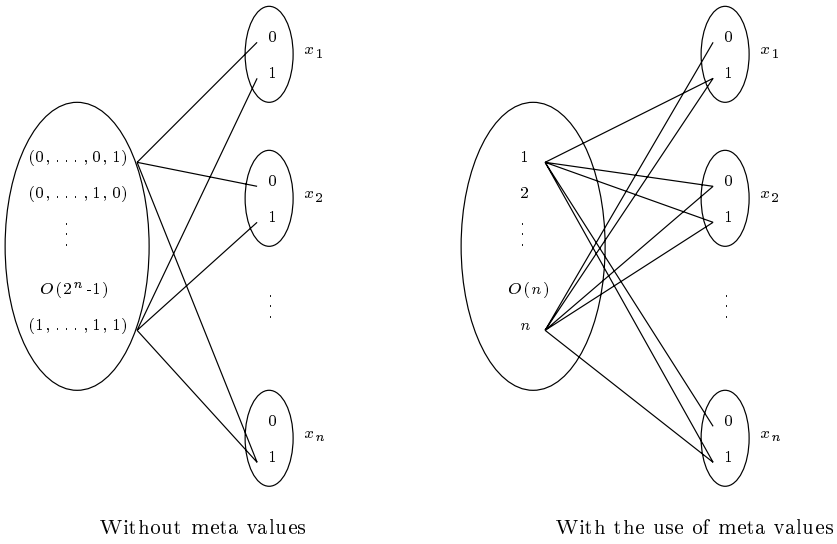


Figure 6.1: Use of meta values in the hidden variable representation.

the effectiveness of consistencies and the performance of backtracking algorithms.

The dual and hidden transformations use a very restricted representation for non-binary constraints in which each tuple in a non-binary constraint is represented by a value in the domain of the dual or hidden variable. This usually results in some exponentially large domains. In practice, there often exists more flexible representations for some classes of non-binary constraints. For instance, in Example 1.4, three hidden variables are used to represent the global equation constraint, and each of the hidden variables has only two values. Furthermore, with the help of identifying meta values of the dual or hidden variables, their domains may be dramatically condensed. For example, consider a constraint over n Boolean variables, $x_1 \vee \cdots \vee x_n$. If it is represented by a "pure" hidden variable, the domain of the hidden variable will contain $2^n - 1$ tuples, which is usually too large to be used in practice. However, the domain of the hidden variable can be condensed by the use of meta values. One scheme is to represent the constraint with one hidden variable c whose domain contains n values, $\{1, \ldots, n\}$, and add one constraint between c and each of the ordinary variables x_i to specify the following relation, $c = i \longrightarrow x_i = 1$, as shown in Figure 6.1. Note that the constraints in the second representation are not hidden constraints anymore because they are not one-way functional constraints. Further work should include identifying meta values and more flexible representation schemes for some classes of constraints and evaluating their effects on the effectiveness of consistencies and the performance of backtracking algorithms.

6.2 Conclusion

Our work aims at improving the efficiency of solving constraint satisfaction problems with respect to improving the backtracking algorithms and improving the formulations. We studied the relations between look-ahead algorithms and the backjumping technique and evaluated the dual and hidden transformation techniques. Throughout the dissertation, we did not restrict ourself to binary CSPs.

The theoretical contributions in this dissertation include the following. We partially explained why look-ahead algorithms will benefit less from backjumping enhancement. We introduced the concept of backjump level to characterize the execution of backjumping algorithms, the concept of k-proof-tree to characterize the strong k-consistency achievement algorithms, and the concept of induced CSP to characterize the maintaining strong k-consistency algorithm. We evaluated two modeling techniques, the dual transformation and the hidden transformation, with respect to the effectiveness of various consistency properties and the performance of some backtracking algorithms. To our knowledge, this is the first comprehensive approach to evaluating modeling techniques in a purely theoretical way.

The practical contributions in this dissertation include the following. We proposed a new algorithm, GAC-CBJ, and our experiments show that GAC-CBJ significantly improves GAC on some harder real world problems, and it is only 10% slower than GAC on relatively easy problems. The theoretical results in the comparison of the dual and hidden transformations also have practical interests. For example, we know that GAC applied on an original formulation always visits the same nodes as MAC applied on its hidden transformation, and MAC applied on the dual transformation may be significantly better than GAC applied on the original problem only if there are two constraints in the original problem that share more than one variable. These results can be used by practitioners to more effectively find an efficient model for real-world problems.

Bibliography

[1] J. Aerts. A survey of optimization algorithms for job shop scheduling. Technical Report COSOR 97-19, Eindhoven University of Technology, 1997.

[2] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17:57–73, 1993.

[3] J. F. Allen. Maintaining knowledge about temporal intervals. *Comm. ACM*, 26:832–843, 1983.

[4] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[5] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[6] F. Bacchus and A. Grove. Looking forward in constraint satisfaction algorithms. 1999. Available from: http://logos.uwaterloo.ca/fbacchus/on-line.html.

[7] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, WI, 1998.

[8] F. Bacchus and P. van Run. Dynamic variable ordering in CSPs. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 258–275, Cassis, France, 1995. Available as: Springer Lecture Notes in Computer Science 976.

[9] P. Baptiste and C. Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 600–606, Montreal, Quebec, 1995.

[10] R. J. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 46–60, Cambridge, Mass., 1996.

[11] R. J. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 202–208, Providence, Rhode Island, 1997.

[12] C. A. Baykan and M. S. Fox. An investigation of opportunistic constraint satisfaction in space planning. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1035–1038, Milan, Italy, 1987.

[13] B. W. Benson Jr. and E. C. Freuder. Interchangeability preprocessing can improve forward checking search. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 28–30, Vienna, 1992.

[14] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[15] C. Bessière, E. C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 592–598, Montreal, Quebec, 1995.

[16] C. Bessière and J.-C. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 61–75, Cambridge, Mass., 1996.

[17] C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 398–404, Nagoya, Japan, 1997.

[18] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Comm. ACM*, 18:651–656, 1975.

[19] J. E. Borrett. *Formulation Selection for Constraint Satisfaction Problems: A Heuristic Approach.* PhD thesis, University of Essex, United Kingdom, 1998.

[20] S. Bressan. Database query optimization and evaluation as constraint satisfaction problem solving. In *Proceedings of the Post-ILPS'94 Workshop on Constraints and Databases*, Ithaca, NY, 1994. Available on the World Wide Web at URL: http://www.research.att.com/~sudarsha/ILPS94-CDBWorkshop.html.

[21] P. Brucker, B. Jurisch, and A. Krämer. The job-shop problem and immediate selection. *Annals of Operations Research*, 50:73–114, 1994.

[22] P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49:107–127, 1994.

[23] P. Burke and P. Prosser. The distributed asynchronous scheduler. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, pages 309–339. Morgan Kaurmann Publishers, 1994.

[24] A. Colmerauer. An introduction to Prolog III. *Comm. ACM*, 33:69–90, 1990.

[25] M. C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[26] M. C. Cooper, D. A. Cohen, and P. G. Jeavons. Characterising tractable constraints. *Artificial Intelligence*, 65:347–361, 1994.

[27] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.

[28] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 412–417, Nagoya, Japan, 1997.

[29] A. Dechter and R. Dechter. Removing redundancies in constraint networks. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 105–109, Seattle, Wash., 1987.

[30] R. Dechter. Learning while searching in constraint-satisfaction problems. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 178–183, Philadelphia, Pennsylvania, 1986.

[31] R. Dechter. Decomposing a relation into a tree of binary relations. *J. of Computer and System Sciences*, 41, 1990.

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

[33] R. Dechter. On the expressiveness of networks with hidden variables. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 556–562, Boston, Mass., 1990.

[34] R. Dechter. Constraint networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence, 2nd Edition*, pages 276–285. John Wiley & Sons, 1992.

[35] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.

[36] R. Dechter and D. Frost. Backtracking algorithms for constraint satisfaction problems – a tutorial survey. Department of Information and Computer Science Technical Report R56, University of California, Irvine, 1998. Available at ftp://ftp.ics.uci.edu/pub/CSP-repository/papers/R56.ps.

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

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

[39] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173:283–308, 1997.

[40] M. Dincbas, P. van Hentenryck, P. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings fo the International Conference on Fifth Generation Computer Systems*, pages 693–702, 1988.

[41] ECRC. Eclipse user manual, 1999. Avaliable from http://www.ecrc.de/eclipse/eclipse.html.

[42] E. C. Freuder. Synthesizing constraint expressions. *Comm. ACM*, 21:958–966, 1978.

[43] E. C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29:24–32, 1982.

[44] E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32:755–761, 1985.

[45] E. C. Freuder. Complexity of k-tree structured constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 4–9, Boston, Mass., 1990.

[46] E. C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 227–233, Anaheim, Calif., 1991.

[47] E. C. Freuder. Constraint programming position paper for ACM workshop on strategic directions in computing research. *ACM Computing Surveys*, 28A(4), December, 1996. Available on the World Wide Web at URL: http://www.cs.unh.edu/ccc/grail.html.

[48] D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 301–306, Seattle, Wash., 1994.

[49] D. Frost and R. Dechter. In search of the best search: An empirical evaluation. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 294–300, Seattle, Wash., 1994.

[50] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 572–578, Montreal, Quebec, 1995.

[51] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[52] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, page 457, Cambridge, Mass., 1977.

[53] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*, pages 268–277, Toronto, Ont., 1978.

[54] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, 1979.

[55] I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 246–252, Portland, Oregon, 1996.

[56] L. Getoor, G. Ottosson, M. Fromherz, and B. Carlson. Effective redundant constraints for online scheduling. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 302–307, Providence, Rhode Island, 1997.

[57] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.

[58] M. L. Ginsberg and D. A. McAllester. GSAT and dynamic backtracking. In *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming*, pages 243–265, Rosario, Orcas Island, Washington, 1994.

[59] S. A. Grant and B. M. Smith. The phase transition behaviour of maintaining arc consistency. Technical Report 95.25, University of Leeds, School of Computer Studies, 1995.

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

[61] ILOG. Ilog optimization suite: White paper, 1998. Avaliable from http://www.ilog.com.

[62] P. Jeavons, D. Cohen, and M. Gyssens. Closure properties of constraints. Technical Report CSD-TR-96-15, Computer Science Department, University of London,, 1996.

[63] P. Jeavons, D. Cohen, and M. Gyssens. A test for tractability. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 267–281, Cambridge, Mass., 1996. Available as: Springer Lecture Notes in Computer Science 1118.

[64] P. G. Jeavons and M. C. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, 79:327–339, 1995.

[65] P. Jégou. Decomposition of domains based on the micro-structure of finite constraint satisfaction problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 731–736, Washington, DC, 1993.

[66] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *KR-96*, pages 374–384, 1996.

[67] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1194–1201, Portland, Oregon, 1996.

[68] G. Kondrak. A theoretical evaluation of selected backtracking algorithms. Technical Report TR94-10, University of Alberta, 1994.

[69] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[70] P. B. Ladkin and R. D. Maddux. On binary constraint problems. *J. ACM*, 41:435–469, 1994.

[71] P. B. Ladkin and A. Reinefeld. Effective solution of qualitative interval constraint problems. *Artificial Intelligence*, 57:105–124, 1992.

[72] J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[73] C. Le Pape. Constraint propagation in planning and scheduling. CIPE technical report, Stanford University, 1991.

[74] H. Levy. A contraction algorithm for finding small cycle cutsets. *J. of Algorithms*, 9:470–493, 1988.

[75] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[76] A. K. Mackworth. On reading sketch maps. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 598–606, Cambridge, Mass., 1977.

[77] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence, 2nd Edition*, pages 285–293. John Wiley & Sons, 1992.

[78] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[79] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[80] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inform. Sci.*, 19:229–250, 1979.

[81] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–206, 1992.

[82] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[83] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656, Munchen, FRG, 1988.

[84] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95–132, 1974.

[85] B. A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[86] B. A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5:16–23, 1990.

[87] B. Nudel. Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics. *Artificial Intelligence*, 21:135–178, 1983.

[88] C. S. Peirce. In C. Hartshorne and P. Weiss, editors, *Collected Papers, Vol. III*. Harvard University Press, 1933. Cited in: F. Rossi, C. Petrie, and V. Dhar, 1989.

[89] G. Pesant and M. Gendreau. A view of local search in constraint programming. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 353–366, Cambridge, Mass., 1996.

[90] L. Proll and B. M. Smith. ILP and constraint programming approaches to a template design problem. Technical Report 97-16, School of Computer Studies, University of Leeds, 1997.

[91] P. Prosser. Domain filtering can degrade intelligent backtrackng search. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 262–267, 1993.

[92] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[93] P. Prosser. MAC-CBJ: Maintaining arc consistency with conflict-directed backjumping. Technical Report Research Report 177, University of Strathclyde, 1995.

[94] J.-C. Regin. A filtering algorithm for constraints of difference in CSP. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, Wash., 1994.

[95] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 550–556, Stockholm, Sweden, 1990.

[96] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 125–129, Amsterdam, 1994.

[97] P. Saint-Dizier. Logic programming for language processing. In *International Conference on Logic Programming*, Paris, France, 1991.

[98] T. Schiex, J.-C. Régin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 216–221, Portland, Oregon, 1996.

[99] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3:1–15, 1994.

[100] R. Seidel. A new method for solving constraint satisfaction problems. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 338–342, Vancouver, B.C., 1981.

[101] B. Selman, H. A. Kautz, and D. A. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 50–54, Nagoya, Japan, 1997. Avaiable from http://www.research.att.com/~selman/challenge.

[102] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, San Jose, Calif., 1992.

[103] B. M. Smith and S. A. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 646–651, Montreal, 1995.

[104] S. F. Smith and C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 139–144, Washington, D.C., 1993.

[105] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[106] K. Stergiou and T. Walsh. Encodings of non-binary constraint satisfaction problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 163–168, Orlando, Florida, 1999.

[107] Mozart Programming System. Tutorial of Oz. Avaliable from http://www.mozart-oz.org/documentation/.

[108] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[109] E. P. K. Tsang and N. Foster. Solution synthesis in the constraint satisfaction problem. Technical Report CSM-142, Departmant of Computer Science, University of Essex, 1990.

[110] P. van Beek. On the inherent level of local consistency in constraint networks. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 368–373, Seattle, Wash., 1994.

[111] P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 585–590, Orlando, Florida, 1999.

[112] P. van Beek and R. Dechter. Constraint tightness versus global consistency. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 572–582, Bonn, Germany, 1994.

[113] P. van Beek and R. Dechter. On the minimality and global consistency of row-convex constraint networks. *J. ACM*, 42(3):543–561, 1995.

[114] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[115] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[116] M. Vilain. A system for reasoning about time. In *Proceedings of the Second National Conference on Artificial Intelligence*, pages 197–201, Pittsburgh, Pa., 1982.

[117] M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 377–382, Philadelphia, Pa., 1986.

[118] R. J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistent in CSPs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 239–245, Chambéry, France, 1993.

[119] D. Waltz. Understanding line drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[120] R. Weigel, C. Bliek, and B. Faltings. On reformulation of constraint satisfaction problems. In *Proceedings of the 13th European Conference on Artificial Intelligence*, Brighton, United Kingdom, 1998.

[121] R. Weigel and B. Faltings. Structuring techniques for constraint satisfaction problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 418–423, Nagoya, Japan, 1997.

[122] J. Zhou. A constraint program for solving the job-shop problem. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 510–524, Cambridge, Mass., 1996.

Appendix A

Glossary