Binary vs. Non-Binary Constraints *

Fahiem Bacchus
Department of Computer Science
University of Toronto
Toronto, Canada
fbacchus@cs.toronto.edu

Xinguang Chen
Department of Computing Science
University of Alberta
Alberta, Canada
xinguang@cs.ualberta.ca

Peter van Beek
Department of Computer Science
University of Waterloo
Ontario, Canada
vanbeek@uwaterloo.ca

Toby Walsh
Department of Computer Science
University of York
York, England
tw@cs.york.ac.uk

* This paper includes results that first appeared in [1, 4, 23]. This research has been supported in part by the Canadian Government through their NSERC and IRIS programs, and by the EPSRC Advanced Research Fellowship program.

Abstract

There are two well known transformations from non-binary constraints to binary constraints applicable to constraint satisfaction problems (CSPs) with finite domains: the dual transformation and the hidden (variable) transformation. We perform a detailed formal comparison of these two transformations. Our comparison focuses on two backtracking algorithms that maintain a local consistency property at each node in their search tree: the forward checking and maintaining arc consistency algorithms. We first compare local consistency techniques such as arc consistency in terms of their inferential power when they are applied to the original (non-binary) formulation and to each of its binary transformations. For example, we prove that enforcing arc consistency on the original formulation is equivalent to enforcing it on the hidden transformation. We then extend these results to the two backtracking algorithms. We are able to give either a theoretical bound on how much one formulation is better than another, or examples that show such a bound does not exist. For example, we prove that the performance of the forward checking algorithm applied to the hidden transformation of a problem is within a polynomial bound of the performance of the same algorithm applied to the dual transformation of the problem. Our results can be used to help decide if applying one of these transformations to all (or part) of a constraint satisfaction model would be beneficial.

1 Introduction

To model a problem as a constraint satisfaction problem (CSP), we specify a search space using a set of variables each of which can be assigned a value from some finite domain of values. To specify the assignments that solve the problem, the model includes constraints that restrict the set of acceptable assignments. Each constraint is over some subset of the variables and imposes a restriction on the simultaneous values these variables may take. In general, there are many possible ways of modeling a problem as a CSP. Each model might contain a different set of variables, domains, and constraints. The choice of model can have a large impact on the time it takes to find a solution [3, 17, 21], and various modeling techniques have been developed, including adding redundant and symmetry-breaking constraints [9, 22, 25], adding hidden variables [6, 24], aggregating or grouping variables together [3, 7], and transforming a CSP model into an equivalent representation over a different set of variables [7, 11, 19, 26].

One important modeling decision is the arity of the constraints used. Constraints can be either binary over pairs of variables, or non-binary over three or more variables. Although a problem may be naturally modeled with non-binary constraints, these constraints can be easily (and automatically) transformed into binary constraints. Many CSP search algorithms are designed specifically for binary constraints, and furthermore, like all modeling decisions, the choice of binary or non-binary constraints can have a significant impact on the time it takes to solve the CSP.

In general, there is much research that remains to be done on the question of which modeling techniques one should choose when attacking a particular problem. In this paper, we formally study the effectiveness of two modeling techniques that can be used to transform a general (non-binary) CSP model into an equivalent binary CSP: the dual and hidden transformations [7, 19]. Our results give some guidance on the question of choosing between binary and non-binary constraints. Further, the dual and hidden transformations can be seen as extensions of the widely used techniques of aggregating variables together or adding hidden variables to reduce the arity of constraints and thus our results also provide information about these modeling techniques.

The choice of a CSP model also depends on the algorithm that will be used to solve the model. We focus here on backtracking search algorithms that maintain a local consistency property at each node in their search tree. Various types of local consistency have been defined, and algorithms developed for enforcing them (e.g, [5, 13, 14]). Algorithms that maintain a local consistency property during backtracking search (e.g., [8, 10, 15, 16, 20]) can detect dead-ends sooner and thus have the potential of significantly reducing the size of the tree they have to search. Such algorithms have demonstrated significant empirical advantages and are the algorithms of choice in practice. Hence, they are the most relevant objects of study.

We compare the performance of local consistency techniques and backtracking algorithms on three different models of a problem: the original formulation, the dual transformation, and the hidden transformation. For the local consistency techniques, we establish whether a local consistency property on one model is stronger than or equivalent to a local consistency property on another. Among other results, we prove that arc consistency on the original formulation is equivalent to arc consistency on the hidden transformation, but that arc consistency on the dual transformation is stronger than arc consistency on the original formulation. For backtracking algorithms, we give either a theoretical bound on how much better one model can be over another when using a given algorithm, or we give examples to show that no such polynomial bound exists. For example, we prove that the performance of an algorithm that maintains arc consistency when applied to the original formulation is equal to its performance when applied to the hidden transformation. As another example, we also show that the performance of the forward checking algorithm on the hidden transformation is never more than a polynomial factor worse than its performance on the dual, but that its performance on the dual can be an exponential factor worse than its performance on the hidden. Hence we have good theoretical reasons to prefer using the forward checking algorithm on the hidden transformation rather than on the dual transformation. In this way, our results can provide general guidelines as to which transformation, if any, should be applied to a non-binary CSP.

2 Background

In this section, we formally define constraint satisfaction problems and the dual and hidden transformations. In addition we briefly review local consistency techniques and the search tree explored by backtracking algorithms.

2.1 Basic definitions

Definition 1 (Constraint Satisfaction Problem (CSP)) A constraint satisfaction problem, $\mathcal{P}$, is a tuple $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ whose components are defined below.

- $\mathcal{V} = \{x_1, \ldots, x_n\}$ is a finite set of n variables.
- $\mathcal{D} = \{dom(x_1), \ldots, dom(x_n)\}$ is a set of domains. Each variable $x \in \mathcal{V}$ has a corresponding finite domain of possible values, $dom(x)$.
- $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of m constraints. Each constraint $C \in \mathcal{C}$ is a pair $(vars(C), rel(C))$ defined as follows.
  1. $vars(C)$ is an ordered subset of the variables, called the constraint *scheme*. The size of $vars(C)$ is known as the *arity* of the constraint. A *binary* constraint has arity equal to 2; a *non-binary* constraint has arity greater than 2.
  2. $rel(C)$ is a set of tuples over $vars(C)$, called the constraint *relation*, that specifies the allowed combinations of values for the variables in $vars(C)$. A *tuple* over an ordered 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 viewed as a set of variable-value assignments $\{x_1 \leftarrow a_1, \ldots, x_k \leftarrow a_k\}$.

Throughout the paper, we use n, d, m, and r to denote the number of variables, the size of the largest domain, the number of constraints, and the arity of the largest constraint scheme in the CSP, respectively. As well, we assume throughout that for any variable $x \in \mathcal{V}$, there is at least one constraint $C \in \mathcal{C}$ such that $x \in vars(C)$.

Example 1 Propositional satisfiability (SAT) problems can be formulated as CSPs. Consider a SAT problem with 6 propositions, $x_1, \ldots, x_6$, and 4 clauses, (1) $x_1 \vee x_3 \vee x_6$, (2) $\neg x_1 \vee \neg x_3 \vee x_4$, (3) $x_4 \vee \neg x_5 \vee x_6$ and (4) $x_2 \vee x_4 \vee \neg x_5$. In one CSP representation of this 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, $C_1(x_1, x_3, x_6)$, $C_2(x_1, x_3, x_4)$, $C_3(x_4, x_5, x_6)$ and $C_4(x_2, x_4, x_5)$. Each constraint specifies the value combinations that will make its corresponding clause true. For example, $C_4(x_2, x_4, x_5)$, the constraint associated with the clause $x_2 \vee x_4 \vee \neg x_5$, allows all tuples over the variables x_2, x_4, and x_5 except the falsifying assignment $(0, 0, 1)$.

We use the notation $vars(t)$ to denote the set of variables a tuple t is over. If X is any subset of $vars(t)$ then $t[X]$ is used to denote the tuple over X that is obtained by restricting t to the variables in X. Given a constraint C and a subset of its variables $S \subseteq vars(C)$, the projection $\pi_S C$ is a new constraint, where $vars(\pi_S C) = S$ and $rel(\pi_S C) = \{t[S] \mid t \in rel(C)\}$.

An *assignment* to a set of variables X is simply a tuple over X. An assignment t is *consistent* if, for all constraints C such that $vars(C) \subseteq vars(t)$, $t[vars(C)] \in rel(C)$. A *solution* to a CSP is a consistent assignment to all of the variables in the CSP. If no solution exists, the CSP is said to be *insoluble*.

Local consistency is an important concept in CSPs. Local consistencies are properties of CSPs that are defined over "local" parts of the CSP, e.g., properties defined over subsets of the variables and constraints of the CSP. Many local consistency properties on CSPs have been defined (see [5] for a large collection).

Local consistency properties are generally neither necessary nor sufficient conditions for a CSP to be soluble. For example, it is quite possible for a CSP that is not arc consistent to have solutions, and for an arc consistent CSP to be insoluble. The importance of local consistency properties arise instead from the existence of (typically polynomial) algorithms for *enforcing* these properties.

We say that a local consistency property $\mathcal{LC}$ can be enforced if there exists a (computable) function from CSPs to new CSPs, such that if $\mathcal{P}$ is a CSP then $\mathcal{LC}(\mathcal{P})$ is a new CSP with the *same set of solutions* (and thus it must necessarily have the same set of variables).[1] We call applying this function to a CSP *enforcing* the local consistency. Furthermore, we require that $\mathcal{LC}(\mathcal{P})$ satisfy the property $\mathcal{LC}$, i.e., enforcing $\mathcal{LC}$ must yield a CSP satisfying $\mathcal{LC}$, and that if $\mathcal{P}$ satisfies $\mathcal{LC}$ then $\mathcal{LC}(\mathcal{P}) = \mathcal{P}$, i.e., enforcing a local consistency on a CSP that already satisfies it does not change the CSP. The reason for enforcing a local consistency property is that often $\mathcal{LC}(\mathcal{P})$ is easier to solve than $\mathcal{P}$.

In this paper we further restrict our attention to local consistency properties whose enforcement involves only three types of transformations to the CSP: (1) the domains of some of the variables might be reduced, (2) some constraint relations might be reduced

[1] Note that we use the notation $\mathcal{LC}$ to denote the property of a problem $\mathcal{P}$, and $\mathcal{LC}(\mathcal{P})$ to denote the problem resulting from enforcing the property $\mathcal{LC}$.

(i.e., elements of $rel(C)$ might be removed), and (3) some new constraints might be added to the CSP.[2] Note however, in all cases the variables of the CSP are unchanged and their domain of values can only be reduced.

A CSP is said to be *empty* if at least one of its variables has an empty domain or at least one of its constraints has an empty relation. An empty CSP is obviously insoluble. Given a local consistency property $\mathcal{LC}$, we say that a CSP $\mathcal{P}$ is *not empty after enforcing* $\mathcal{LC}$ if $\mathcal{LC}(\mathcal{P})$ is not empty.

One of the most important local consistency properties is arc consistency [13, 14].

Definition 2 (Arc Consistency) Let $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a CSP. Given a constraint C and a variable $x \in vars(C)$, a value $a \in dom(x)$ has a *support* 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 each value a of each variable $x \in vars(C)$ has a support in C. The entire CSP, $\mathcal{P}$, is *arc consistent* iff it has non-empty domains and each of its constraints is arc consistent.

Arc consistency can be enforced on a CSP by repeatedly removing unsupported values from the domains of its variables to create a subdomain.

Definition 3 (Subdomain) A subdomain $\mathcal{D}'$ of a CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, is a set of domains, $\{dom^{\mathcal{D}'}(x_1),\ldots, dom^{\mathcal{D}'}(x_n)\}$, where $dom^{\mathcal{D}'}(x_i) \subseteq dom(x_i)$, for each $x_i \in \mathcal{V}$. We say a subdomain is *empty* if it contains at least one empty domain. We say a subdomain $\mathcal{D}'$ is arc consistent iff the CSP $\mathcal{P}' = (\mathcal{V}, \mathcal{D}', \mathcal{C}')$ is arc consistent, where $\mathcal{C}'$ are the original constraints $\mathcal{C}$ reduced so they contain only tuples over $\mathcal{D}'$.[3]

Definition 4 (Arc Consistency Closure) An algorithm that enforces arc consistency computes the maximum arc consistent subdomain, and when applied to a CSP, $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, it gives rise to a new arc consistent CSP called the *arc consistency closure* of $\mathcal{P}$, which we denote by $ac(\mathcal{P})$. We have that $ac(\mathcal{P}) = (\mathcal{V}, \mathcal{D}^{ac(\mathcal{P})}, \mathcal{C}^{ac(\mathcal{P})})$, where $\mathcal{D}^{ac(\mathcal{P})}$ is the maximum arc consistent subdomain of $\mathcal{P}$, and $\mathcal{C}^{ac(\mathcal{P})}$ are the original constraints $\mathcal{C}$ reduced so that they contain only tuples over the subdomain $\mathcal{D}^{ac(\mathcal{P})}$.

Constraint satisfaction problems are often solved using backtracking search (for a detailed presentation see, for example, [12, 25]). A backtracking search may be seen as traversing a *search tree*. In this approach we identify tuples (assignments of values to variables) with nodes: the empty tuple is the root of the tree, the first level nodes are 1-tuples (an assignment of a value to a single variable), the second level nodes are 2-tuples (a first level assignment extended by selecting an unassigned variable, called the current variable, and assigning it a value from its domain), and so on. We say that a backtracking algorithm *visits* a node in the search tree if at some stage of the

[2] Only in Section 3.3 will we consider local consistency properties that might add new constraints.

[3] According to Definition 1, the tuples of each constraint must only contain values that are in the domains of the variables. A constraint can be reduced by deleting from the relation all tuples that contain a value a that was removed in the process of enforcing arc consistency. However, arc consistency algorithms do not normally physically remove tuples from the constraint relations of $\mathcal{P}$ as this requires that the relations be represented extensionally. Nevertheless, it is always implicit that the constraint relations are tuples over the reduced variable domains.

algorithm's execution the algorithm tries to extend the tuple of assignments at 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 search tree visited by a backtracking algorithm.

The chronological backtracking algorithm (BT) is the starting point for all of the more sophisticated backtracking algorithms. BT checks a constraint only if all the variables in its scheme have been instantiated. In contrast, the more widely used backtracking algorithms enforce a local consistency property at each node visited in the backtracking search. The consistency enforcement algorithm is applied to the *induced CSP*. This is the original CSP reduced by the current assignment.

Definition 5 (Induced CSP) Given an assignment t of some of the variables of a CSP $\mathcal{P}$, the CSP induced by t, denoted by $\mathcal{P}|_t$, is the same as $\mathcal{P}$ except that the domain of each variable $x \in vars(t)$ contains only one value $t[x]$, the value that has been assigned to x by t, and the constraints are reduced so that they contain only tuples over the reduced domains.

If the induced CSP is empty after enforcing the local consistency, the instantiation of the current variable cannot be extended to a solution and it should be uninstantiated; otherwise, the instantiation of the current variable is accepted and the search continues to the next level. The forward checking algorithm (FC) [10, 15, 25] enforces arc consistency only on the constraints which have exactly one uninstantiated variable. By comparison, on a problem that is not empty after enforcing arc consistency, the maintaining arc consistency or really-full lookahead algorithms [8, 16, 20], as their names suggest, enforce full arc consistency on the induced CSP.

2.2 Dual and hidden transformations

The dual and hidden transformations are two general methods for converting a non-binary CSP into an equivalent binary CSP. The dual transformation comes from the relational database community and was introduced to the CSP community by Dechter and Pearl [7].

The hidden transformation, on the other hand, arose from the work of the philosopher Peirce. In particular, Rossi et al. [19] credit Peirce [18] with first showing that binary relations have the same expressive power as non-binary relations. Peirce's method for representing non-binary relations with a collection of binary relations forms the foundation of the hidden transformation.

In the dual transformation, the constraints of the original formulation become variables in the new representation. We refer to these variables, which represent the original constraints, as the *dual variables*, and the variables in the original CSP as the *ordinary variables*. The domain of each dual variable is exactly the set of tuples that are in the original constraint relation. There is a binary constraint, called a *dual constraint*, between two dual variables iff the two original constraints share some variables. A dual constraint prohibits pairs of tuples that do not agree on the shared variables.

Definition 6 (Dual Transformation) Given a CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, its dual transformation $dual(\mathcal{P}) = (\mathcal{V}^{dual(\mathcal{P})}, \mathcal{D}^{dual(\mathcal{P})}, \mathcal{C}^{dual(\mathcal{P})})$ is defined as follows.

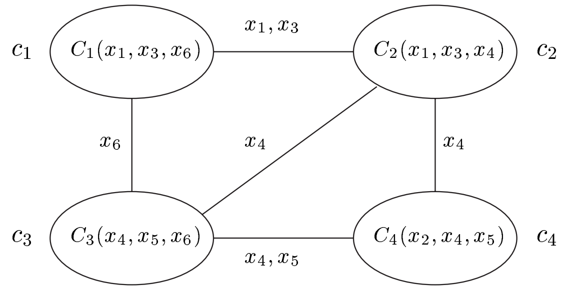

Figure 1: The dual transformation of the CSP in Example 1.

- $\mathcal{V}^{dual(\mathcal{P})} = \{c_1, \ldots, c_m\}$ where $c_1, \ldots, c_m$ are called *dual variables*. For each constraint C_i of $\mathcal{P}$ there is a unique corresponding dual variable c_i. We use $vars(c_i)$ and $rel(c_i)$ to denote the corresponding sets $vars(C_i)$ and $rel(C_i)$ (given that the context is not ambiguous).
- $\mathcal{D}^{dual(\mathcal{P})} = \{dom(c_1), \ldots, dom(c_m)\}$ 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 $vars(C_i)$. An assignment of a value t to a dual variable c_i, $c_i \leftarrow t$, can thus be viewed as being a sequence of assignments to the ordinary variables $x \in vars(c_i)$ where each such ordinary variable is assigned the value $t[x]$.
- $\mathcal{C}^{dual(\mathcal{P})}$ is a set of binary constraints over $\mathcal{V}^{dual(\mathcal{P})}$ called the *dual constraints*. There is a dual constraint between dual variables c_i and c_j if $S = vars(c_i) \cap vars(c_j) \neq \emptyset$. In this dual constraint a tuple $t_i \in dom(c_i)$ is compatible with a tuple $t_j \in dom(c_j)$ iff $t_i[S] = t_j[S]$, i.e., the two tuples have the same values over their common variables.

It is important to note that in our definition all of the constraints of $\mathcal{P}$ are converted to dual variables, even the binary and unary constraints.

Example 2 In the dual transformation of the CSP given in Example 1, there are 4 dual variables, $c_1, \ldots, c_4$, one for each constraint in the original formulation as shown in Figure 1. For example, the dual variable c_1 corresponds to the non-binary constraint $C_1(x_1, x_3, x_6)$ and the domain of c_1 contains all possible tuples except $(0, 0, 0)$. As an example of a dual constraint, the constraint between c_1 ($C_1(x_1, x_3, x_6)$) and c_2 ($C_2(x_1, x_3, x_4)$) requires that the first and second arguments of the tuples assigned to c_1 and c_2 agree. Hence, $\{c_1 \leftarrow (0, 0, 1)\}$ is compatible with $\{c_2 \leftarrow (0, 0, 0)\}$, but $\{c_1 \leftarrow (0, 0, 1)\}$ is incompatible with $\{c_2 \leftarrow (0, 1, 0)\}$.

In the hidden transformation, the set of variables consists of all the ordinary variables in the original formulation with their original domains plus all the dual variables as defined by the dual transformation. There is a binary constraint, called a *hidden constraint*, between a dual variable and each of the ordinary variables in the constraint

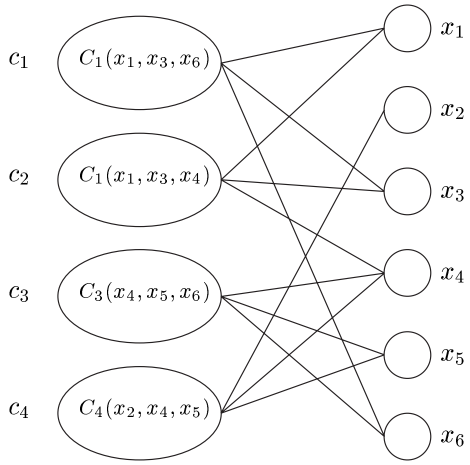

Figure 2: The hidden transformation of the CSP in Example 1.

represented by the dual variable. A hidden constraint enforces the condition that a value of the ordinary variable must be the same as the value assigned to it by the tuple that is the value of the dual variable.

Definition 7 (Hidden Transformation) Given a CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, its hidden transformation $hidden(\mathcal{P})$ = $(\mathcal{V}^{hidden(\mathcal{P})}, \mathcal{D}^{hidden(\mathcal{P})}, \mathcal{C}^{hidden(\mathcal{P})})$ is defined as follows.

- $\mathcal{V}^{hidden(\mathcal{P})} = \{x_1, \ldots, x_n\} \cup \{c_1, \ldots, c_m\}$, where $x_1, \ldots, x_n$ is the original set of variables in $\mathcal{V}$ (called the *ordinary variables*) and $c_1, \ldots, c_m$ are dual variables generated from the constraints in $\mathcal{C}$. There is a unique dual variable c_i corresponding to each constraint $C_i \in \mathcal{C}$. When dealing with the hidden transformation, the dual variables are sometimes called hidden variables [6].

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

- $\mathcal{C}^{hidden(\mathcal{P})}$ is a set of binary constraints over $\mathcal{V}^{hidden(\mathcal{P})}$ called the *hidden constraints*. For each dual variable c, there is a hidden constraint between c and each of the ordinary variables $x \in vars(c)$. This constraint specifies that a tuple $t \in dom(c)$ is compatible with a value $a \in dom(x)$ iff $t[x] = a$.

The hidden transformation has some special properties. The constraint graph of the hidden transformation is a bipartite graph, as ordinary variables are only constrained with dual variables, and vice versa, and the hidden constraints are one-way functional constraints, in which a tuple in the domain of a dual variable is compatible with at most one value in the domain of the ordinary variable. The dual transformation can in fact be built from the hidden transformation by composing the hidden constraints between the dual variables and the ordinary variables to obtain dual constraints between the dual variables, and then discarding the hidden constraints and ordinary variables [23]. Note that we need not add hidden variables for binary constraints. However, as we obtain similar results if hidden variables are only introduced for ternary and higher arity constraints, we do not consider this further.

Example 3 In the hidden transformation of the CSP given in Example 1, there are 10 variables (6 ordinary variables and 4 dual variables), as shown in Figure 2. As an example of a hidden constraint, the constraint between c_1 ($C_1(x_1, x_3, x_6)$) and x_1 requires that the first argument of the tuple assigned to c_1 agrees with the value assigned to x_1. Hence, $\{c_1 \leftarrow (0,0,1)\}$ is compatible with $\{x_1 \leftarrow 0\}$, and $\{c_1 \leftarrow (0,0,1)\}$ is incompatible with $\{x_1 \leftarrow 1\}$.

In the following, we call a CSP instance $\mathcal{P}$ the *original formulation* with respect to its dual transformation and hidden transformation. Because we usually deal with more than one formulation of $\mathcal{P}$ simultaneously, we use the notation $\mathcal{V}^{f(\mathcal{P})}$, $\mathcal{D}^{f(\mathcal{P})}$ and $\mathcal{C}^{f(\mathcal{P})}$ to denote the set of variables, the set of domains and the set of constraints in $\mathcal{P}$ after it has been reformulated by some transformation f (to become a new CSP $f(\mathcal{P})$). Also, we use $dom^{f(\mathcal{P})}(x)$ to denote the domain of variable x in $f(\mathcal{P})$.

3 Local Consistency Techniques

In this section, we compare the strength of arc consistency on the original formulation, and on the dual and hidden transformations. We show that arc consistency on the original formulation and the hidden transformation are equivalent, but arc consistency on the dual transformation is stronger. We then compare several stronger local consistency properties defined over the binary constraints in the dual and hidden formulations. We establish a hierarchy, with respect to a simple ordering relation, for the various combinations of local consistency and problem formulation.

Debruyne and Bessière [5] compare local consistency properties defined on binary CSPs. They define a local consistency property $\mathcal{LC}_1$ to be stronger than another $\mathcal{LC}_2$ ($\mathcal{LC}_1 \succeq_{DB} \mathcal{LC}_2$) iff in any CSP instance in which $\mathcal{LC}_1$ holds, then $\mathcal{LC}_2$ holds, and $\mathcal{LC}_1$ to be strictly stronger than $\mathcal{LC}_2$ ($\mathcal{LC}_1 \succ_{DB} \mathcal{LC}_2$) if $\mathcal{LC}_1 \succeq_{DB} \mathcal{LC}_2$ and not $\mathcal{LC}_2 \succeq_{DB} \mathcal{LC}_1$.

However, their definition of ordering among different local consistency properties does not provide sufficient discrimination for our purposes as we wish to simultaneously compare changes in problem formulation and local consistency properties. To this end we define the following ordering relation.

Definition 8 Given two local consistency properties $\mathcal{LC}_1$ and $\mathcal{LC}_2$, and two transformations $\mathcal{A}$ and $\mathcal{B}$ for CSP problems (perhaps identity transformations), $\mathcal{LC}_1$ on $\mathcal{A}$ is *tighter* than $\mathcal{LC}_2$ on $\mathcal{B}$, written $\mathcal{LC}_1(\mathcal{A}) \succeq \mathcal{LC}_2(\mathcal{B})$, iff

> given any problem $\mathcal{P}$, if $\mathcal{A}(\mathcal{P})$ is not empty after enforcing $\mathcal{LC}_1$, then $\mathcal{B}(\mathcal{P})$ is also not empty after enforcing $\mathcal{LC}_2$.

$\mathcal{LC}_1$ on $\mathcal{A}$ is *strictly tighter* than $\mathcal{LC}_2$ on $\mathcal{B}$, $\mathcal{LC}_1(\mathcal{A}) \succ \mathcal{LC}_2(\mathcal{B})$, iff $\mathcal{LC}_1(\mathcal{A}) \succeq \mathcal{LC}_2(\mathcal{B})$ and not $\mathcal{LC}_2(\mathcal{B}) \succeq \mathcal{LC}_1(\mathcal{A})$. And $\mathcal{LC}_1$ on $\mathcal{A}$ is *equivalent* to $\mathcal{LC}_2$ on $\mathcal{B}$, $\mathcal{LC}_1(\mathcal{A}) \simeq \mathcal{LC}_2(\mathcal{B})$, iff $\mathcal{LC}_1(\mathcal{A}) \succeq \mathcal{LC}_2(\mathcal{B})$ and $\mathcal{LC}_2(\mathcal{B}) \succeq \mathcal{LC}_1(\mathcal{A})$. If the transformations are identical; i.e., $\mathcal{A} = \mathcal{B}$, then we simply write $\mathcal{LC}_1 \succeq \mathcal{LC}_2$.

The ordering relation we define is motivated by our desire to examine backtracking search algorithms in which some form of local consistency is maintained during search. In such algorithms it is the occurrence of an empty subproblem at a node of the search tree that justifies backtracking. Thus if $\mathcal{LC}_1$ is tighter than $\mathcal{LC}_2$ it follows (using the contrapositive form of the definition: if $\mathcal{P}$ is empty after enforcing $\mathcal{LC}_2$, then $\mathcal{P}$ is also empty after enforcing $\mathcal{LC}_1$) that when an algorithm that maintains $\mathcal{LC}_2$ backtracks at a node, then an algorithm that maintains $\mathcal{LC}_1$ would backtrack at that node as well.

Debruyne and Bessière's ordering relation is defined by whether or not a problem satisfies some local consistency property, whereas our ordering relation is defined by whether or not a non-empty subproblem satisfies some local consistency property. This distinction is important. For example, there exist problems for which the dual transformation is arc consistent but the original formulation is not, and problems where the original formulation is arc consistent while the dual is not. Thus, under Debruyne and Bessière's ordering (suitably modified to deal with two different problem formulations) arc consistency on these two formulations would be incomparable. Under our ordering relation, however, arc consistency on the dual transformation can be shown to be strictly tighter than arc consistency on the original formulation. As the following lemma demonstrates, Debruyne and Bessière's ordering relation is stronger than ours. The relation $\succ_{DB}$ is therefore unable to make as fine distinctions between different local consistencies as the relation $\succeq$.

Lemma 1 *$\mathcal{LC}_1 \succeq_{DB} \mathcal{LC}_2$ implies $\mathcal{LC}_1 \succeq \mathcal{LC}_2$.*

Proof: Let $\mathcal{P}$ be a problem which is not empty after enforcing $\mathcal{LC}_1$. Then there is a non-empty subdomain of $\mathcal{P}$ in which $\mathcal{LC}_1$ holds and hence $\mathcal{LC}_2$ holds, since $\mathcal{LC}_1 \succ_{DB} \mathcal{LC}_2$. Therefore $\mathcal{P}$ is also not empty after enforcing $\mathcal{LC}_2$, and $\mathcal{LC}_1 \succeq \mathcal{LC}_2$. ∎

Note also that $\mathcal{LC}_1 \succ_{DB} \mathcal{LC}_2$ implies $\mathcal{LC}_1 \succeq \mathcal{LC}_2$, but not necessarily $\mathcal{LC}_1 \succ \mathcal{LC}_2$. In particular, $\mathcal{LC}_2 \succeq_{DB} \mathcal{LC}_1$ failing to hold does not imply that $\mathcal{LC}_2 \succeq \mathcal{LC}_1$ also fails to hold, as an $\succeq$ ordering might exist between $\mathcal{LC}_2$ and $\mathcal{LC}_1$ even though no $\succeq_{DB}$ ordering exists.

3.1 Arc consistency on the hidden transformation

Consider a CSP $\mathcal{P}$ with 4 variables, $x_1, \ldots, x_4$, each with domain $\{0, 1, 2\}$ and constraints given by, $x_1 + x_2 < x_3$, $x_1 + x_3 < x_4$ and $x_2 + x_3 < x_4$. Figure 3 shows the mappings between $\mathcal{P}$, its arc consistency closure $ac(\mathcal{P})$, its hidden transformation $hidden(\mathcal{P})$, and the arc consistency closure of its hidden transformation $ac(hidden(\mathcal{P}))$. It turns out that $ac(hidden(\mathcal{P}))$ is the same as the hidden transformation of the arc consistency closure $hidden(ac(\mathcal{P}))$. In this example, an ordinary variable has the same domain in $ac(\mathcal{P})$ and $ac(hidden(\mathcal{P}))$ and the domain of a dual variable in $ac(hidden(\mathcal{P}))$ is the same as the set of tuples in the corresponding constraint that remain after enforcing arc consistency on the original formulation. We show in the following that these properties are true in general.

Lemma 2 *If $\mathcal{P}$ is arc consistent; i.e., $\mathcal{P} = ac(\mathcal{P})$, then $hidden(\mathcal{P})$ is empty if and only if $\mathcal{P}$ is empty.*

Proof: If $\mathcal{P}$ is empty then either it has an empty variable domain or an empty constraint relation or both. In either of these cases $hidden(\mathcal{P})$ will have an empty variable domain. That is, for any $\mathcal{P}$, if $\mathcal{P}$ is empty, then $hidden(\mathcal{P})$ is empty. (We do not need arc consistency for this direction.)

If $hidden(\mathcal{P})$ is empty then we have one of two cases. (1) $hidden(\mathcal{P})$ has an empty variable domain. In this case $\mathcal{P}$ must also be empty. Otherwise, (2) there are no empty variable domains but $hidden(\mathcal{P})$ has an empty constraint relation. We claim that case (2) cannot occur if $\mathcal{P}$ is arc consistent. Let $C_{x,c}$ be any of the hidden constraints, and say that it is a constraint between an ordinary variable x and a dual variable c. $dom(x)$ is not empty so it must contain at least one value a. Furthermore, since $\mathcal{P}$ is arc consistent there must be a tuple t in $dom(c)$ such that the pair (x, t) is a support for a in $C_{x,c}$. That is, $rel(C_{x,c})$ must contain at least the pair (x, t). So none of the constraints of $hidden(\mathcal{P})$ can be empty. ∎

Theorem 1 *Given a CSP $\mathcal{P}$,*

1. *$\mathcal{P}$ is arc consistent if and only if $hidden(\mathcal{P})$ is arc consistent,*
2. *$hidden(ac(\mathcal{P})) = ac(hidden(\mathcal{P}))$, and*
3. *arc consistency on $\mathcal{P}$ is equivalent to arc consistency on $hidden(\mathcal{P})$; i.e., $ac \simeq ac(hidden)$.*

Proof: (1) If the original formulation $\mathcal{P}$ is not arc consistent, then there is at least one 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. Hence, in $hidden(\mathcal{P})$, $x \leftarrow a$ will not have a support in the hidden constraint between the ordinary variable x and the corresponding dual variable c, and $hidden(\mathcal{P})$ is not arc consistent either. On the other hand, suppose $hidden(\mathcal{P})$ is not arc consistent. Since by definition for every constraint C, $rel(C)$ contains only tuples whose values are in the product of the domains of $vars(C)$, each tuple in the domain of a dual variable must have a support in a hidden constraint between the dual variable and an ordinary variable. Hence, for $hidden(\mathcal{P})$ not to be arc consistent there must be a value a of an ordinary variable x and a dual variable c such that $\{x \leftarrow a\}$ does not have a support in the hidden constraint between x and c; thus $\{x \leftarrow a\}$ cannot have a support in the corresponding original constraint C. Therefore, the original formulation $\mathcal{P}$ is not arc consistent either.

(2) First, $hidden(\mathcal{P})$, $hidden(ac(\mathcal{P}))$, and $ac(hidden(\mathcal{P}))$ all have the same set of variables (ordinary and dual) and the same constraint schemes since (a) enforcing arc consistency does not alter the variables or the constraint schemes of a problem and (b)

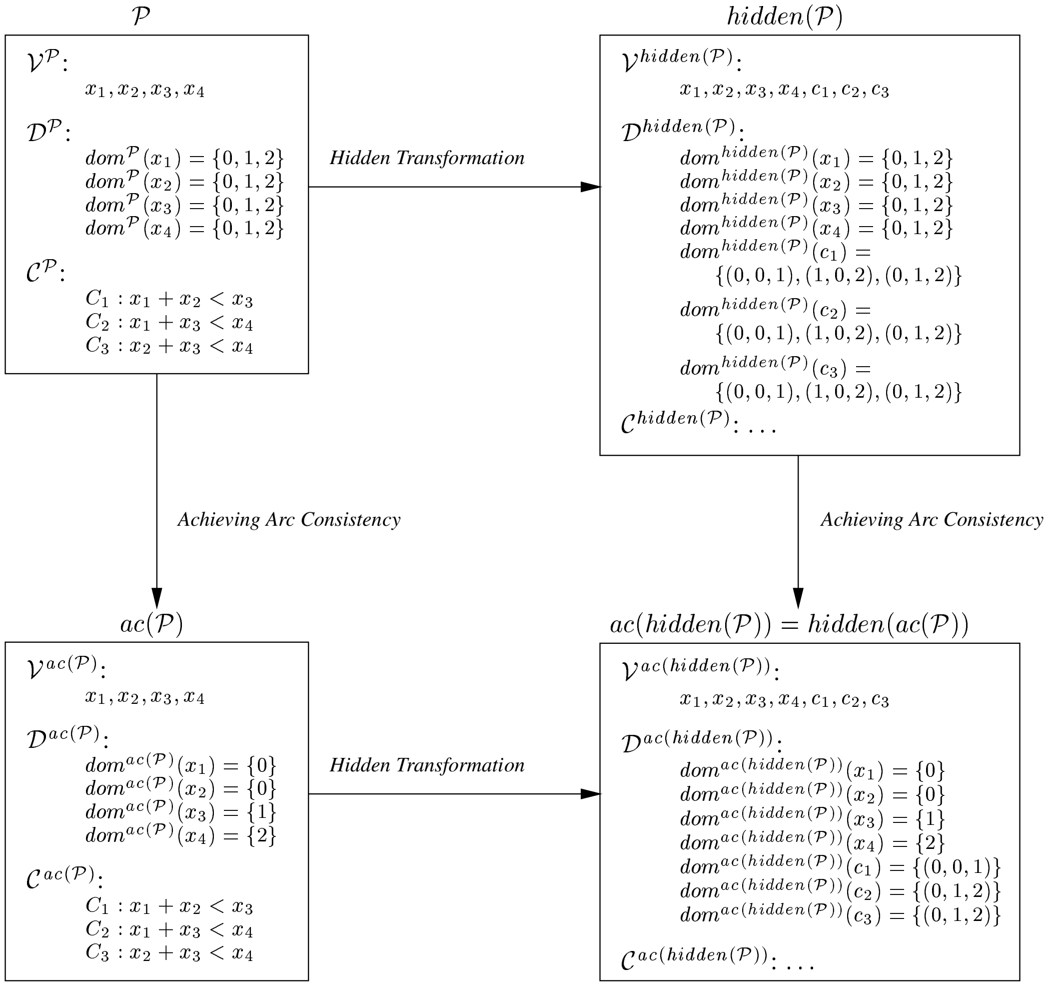

Figure 3: An example to show the mappings between an original CSP, its hidden transformation, its arc consistency closure, the arc consistency closure of its hidden transformation, and the hidden transformation of its arc consistency closure.

the variables of the hidden are completely determined by the variables and the schemes of the constraints of the original problem.

Second, the set of domains of $hidden(ac(\mathcal{P}))$, $\mathcal{D}^{hidden(ac(\mathcal{P}))}$, is a subdomain of $hidden(\mathcal{P})$: enforcing ac on $\mathcal{P}$ reduces the variable domains and the constraint relations, which simply has the effect, after applying the hidden transformation, of reducing the domains of the ordinary and dual variables from their state in $hidden(\mathcal{P})$. Furthermore, by (1) $hidden(ac(\mathcal{P}))$ must be arc consistent. Thus $\mathcal{D}^{hidden(ac(\mathcal{P}))}$ is also a subdomain of $ac(hidden(\mathcal{P}))$ as $\mathcal{D}^{ac(hidden(\mathcal{P}))}$ is the (unique) maximal arc consistent subdomain of $hidden(\mathcal{P})$. This means that for every variable q (ordinary or dual) $dom^{hidden(ac(\mathcal{P}))}(q) \subseteq dom^{ac(hidden(\mathcal{P}))}(q)$.

Third, we show for every variable q, $dom^{ac(hidden(\mathcal{P}))}(q) \subseteq dom^{hidden(ac(\mathcal{P}))}(q)$, and hence that $\mathcal{D}^{hidden(ac(\mathcal{P}))} = \mathcal{D}^{ac(hidden(\mathcal{P}))}$. There are two cases to consider. (a) q is an ordinary variable. Since, $ac(hidden(\mathcal{P}))$ is arc consistent, each value $a \in dom^{ac(hidden(\mathcal{P}))}(q)$ must have a supporting tuple in every dual variable c that q is constrained with. Furthermore, these supporting tuples must themselves have supports in every ordinary variable that c is constrained with. In other words, a has a support in each constraint by a tuple consisting of values from $\mathcal{D}^{ac(hidden(\mathcal{P}))}$. Thus the set of domains of the ordinary variables in $ac(hidden(\mathcal{P}))$ are an arc consistent subdomain of $\mathcal{P}$, and by maximality we must have that $dom^{ac(hidden(\mathcal{P}))}(q) \subseteq dom^{ac(\mathcal{P})}(q)$. Furthermore $dom^{ac(\mathcal{P})}(q) = dom^{hidden(ac(\mathcal{P}))}(q)$ by the construction of the hidden. Thus for ordinary variables $dom^{ac(hidden(\mathcal{P}))}(q) \subseteq dom^{hidden(ac(\mathcal{P}))}(q)$. (b) q is a dual variable. From (a) we know that every tuple $t \in dom^{ac(hidden(\mathcal{P}))}(q)$ consists of values of ordinary variables taken from $\mathcal{D}^{ac(\mathcal{P})}$. Hence, in $ac(\mathcal{P})$, t will be in the constraint relation of the constraint corresponding to q, and thus t will also be in $dom^{hidden(ac(\mathcal{P}))}(q)$.

Finally, in any hidden formulation the hidden constraints have the same intension. Thus given that the variable domains $\mathcal{D}^{hidden(ac(\mathcal{P}))}$ and $\mathcal{D}^{ac(hidden(\mathcal{P}))}$ are identical all of the constraint relations (extensions) will be the same in $hidden(ac(\mathcal{P}))$ and $ac(hidden(\mathcal{P}))$. (The constraint schemes are determined by the dual variables which also agree in these two formulations). Hence, $hidden(ac(\mathcal{P}))$ and $ac(hidden(\mathcal{P}))$ have the same set of variables, domains for these variables, and constraints. That is, they are syntactically identical.

(3) Since $ac(\mathcal{P})$ is arc consistent, $ac(\mathcal{P})$ is empty iff $ac(hidden(\mathcal{P}))$ is empty by (2) and Lemma 2. ∎

Corollary 1 *In any CSP $\mathcal{P}$, for each of the ordinary variables x in $\mathcal{P}$, $dom^{ac(\mathcal{P})}(x) = dom^{hidden(ac(\mathcal{P}))}(x) = dom^{ac(hidden(\mathcal{P}))}(x)$.*

Proof: The first equality follows from the construction of the hidden; the second follows from (2) of Theorem 1. ∎

3.2 Arc consistency on the dual transformation

We have proven that the original formulation is arc consistent if and only if its hidden transformation is arc consistent. However, such an equivalence does not hold for the dual transformation.

Example 4 Consider a CSP $\mathcal{P}$ with four Boolean variables and constraints:

$$\begin{array}{rcl} C_1(x_1, x_2, x_3) & = & \{(0,0,0), (1,1,1)\}, \\ C_2(x_2, x_3, x_4) & = & \{(0,0,0), (1,1,1)\}, \\ C_3(x_1, x_3, x_4) & = & \{(0,0,1), (1,1,0)\}. \end{array}$$

The original formulation $\mathcal{P}$ 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 empty after enforcing arc consistency on the dual transformation. Thus $dual(\mathcal{P})$ is not arc consistent and $ac(dual(\mathcal{P}))$ is empty; i.e., $ac \not\succeq ac(dual)$.

Example 5 Consider a CSP $\mathcal{P}$ with three Boolean variables and constraints:

$$\begin{array}{rcl} C_1(x_1, x_2) & = & \{(1,1)\}, \\ C_2(x_2, x_3) & = & \{(1,1)\}, \\ C_3(x_1, x_3) & = & \{(1,1)\}. \end{array}$$

The dual transformation $dual(\mathcal{P})$ is arc consistent. However, the original formulation is not arc consistent, because the value 0 for each of the variables will be removed from its domain when enforcing arc consistency.

We can show that if the dual transformation is not empty after enforcing arc consistency, then the original formulation is not empty either after enforcing arc consistency; i.e., $ac(dual) \succeq ac$. Together with Example 4 this shows that $ac(dual) \succ ac$.

Lemma 3 *If a subdomain $\mathcal{D}'$ of a dual transformation $dual(\mathcal{P})$ is arc consistent, then for each pair of dual variables c_i and c_j in $dual(\mathcal{P})$ such that $S = vars(c_i) \cap vars(c_j) \neq \emptyset$, and for each $x \in S$, $\pi_{\{x\}} dom^{\mathcal{D}'}(c_i) = \pi_{\{x\}} dom^{\mathcal{D}'}(c_j)$.*

Proof: For each $x \in S$ and each tuple $t \in \pi_{\{x\}} dom^{\mathcal{D}'}(c_i)$ there is a tuple $t_i \in dom^{\mathcal{D}'}(c_i)$ such that $t_i[x] = t$. Because $\mathcal{D}'$ is arc consistent, there must also be a tuple $t_j \in dom^{\mathcal{D}'}(c_j)$ such that $t_i[S] = t_j[S]$. Thus $t = t_i[x] = t_j[x]$. Because $t \in \pi_{\{x\}} dom^{\mathcal{D}'}(c_j)$, we have $\pi_{\{x\}} dom^{\mathcal{D}'}(c_i) \subseteq \pi_{\{x\}} dom^{\mathcal{D}'}(c_j)$. Similarly, we can show that $\pi_{\{x\}} dom^{\mathcal{D}'}(c_j) \subseteq \pi_{\{x\}} dom^{\mathcal{D}'}(c_i)$. Therefore, $\pi_{\{x\}} dom^{\mathcal{D}'}(c_i) = \pi_{\{x\}} dom^{\mathcal{D}'}(c_j)$. ∎

From the arc consistency closure of $dual(\mathcal{P}))$, $ac(dual(\mathcal{P}))$, we can construct a subdomain for the original formulation $\mathcal{P}$ (in Theorem 2 below we show that this subdomain is in fact an arc consistent subdomain of $\mathcal{P}$).

Definition 9 Let $\mathcal{D}^{dualac(\mathcal{P})}$ denote the subdomain for the ordinary variables in $\mathcal{P}$ that is constructed from the domains for the dual variables in $ac(dual(\mathcal{P}))$ as follows: for each ordinary variable x in $\mathcal{P}$, choose a dual variable c in $ac(dual(\mathcal{P}))$ such that

$x \in vars(c)$ and set $dom^{\mathcal{D}^{dualac(\mathcal{P})}}(x)$ to be $\pi_{\{x\}} dom^{ac(dual(\mathcal{P}))}(c)$. Note that each $dom^{\mathcal{D}^{dualac(\mathcal{P})}}(x)$ is well defined as (1) by our assumption that each variable is constrained by at least one constraint, it is always possible to choose one such c, and (2) by Lemma 3, if there is more than one such c the result does not depend on the dual variable we choose.

For example, the dual transformation of the CSP in Example 5 is arc consistent. Hence $\mathcal{D}^{ac(dual(\mathcal{P}))}$ is $\{dom(c_1) = \{(1,1)\}, dom(c_2) = \{(1,1)\}, dom(c_3) = \{(1,1)\}\}$, and $\mathcal{D}^{dualac(\mathcal{P})}$ is $\{dom(x_1) = \{1\}, dom(x_2) = \{1\}, dom(x_3) = \{1\}\}$.

Theorem 2 *Given a CSP $\mathcal{P}$, arc consistency on $dual(\mathcal{P})$ is strictly tighter than arc consistency on $\mathcal{P}$; i.e., $ac(dual) \succ ac$.*

Proof: We show that if $ac(dual(\mathcal{P}))$ is not empty, then neither is $ac(\mathcal{P})$; i.e. $ac(dual) \succeq ac$. Because the domain of each dual variable in $ac(dual(\mathcal{P}))$ is not empty, its projection over an ordinary variable cannot be empty either. So there is no empty domain in $\mathcal{D}^{dualac(\mathcal{P})}$. In $\mathcal{P}$, for each ordinary variable x, each value $a \in dom^{\mathcal{D}^{dualac(\mathcal{P})}}(x)$, and each constraint C, where $x \in vars(C)$, let a be the projection of the tuple t of the corresponding dual variable c. For each of the variables $y \in vars(C)$, $t[y] \in dom^{\mathcal{D}^{dualac(\mathcal{P})}}(y)$. Thus t is a support for $\{x \leftarrow a\}$ in constraint C, and furthermore t is a tuple of values all of which come from $\mathcal{D}^{dualac(\mathcal{P})}$. Therefore, $\mathcal{D}^{dualac(\mathcal{P})}$ is a non-empty arc consistent subdomain of $\mathcal{P}$ and thus $ac(\mathcal{P})$ is not empty.

Example 4 shows that arc consistency on the dual transformation may be *strictly* tighter than arc consistency on the original formulation; i.e., $ac \not\succeq dual(ac)$. Therefore, $ac(dual) \succ ac$. ∎

By combining Theorem 1 and Theorem 2, we can make the following comparison between arc consistency on the dual transformation and on the hidden transformation.

Corollary 2 *Given a CSP $\mathcal{P}$, arc consistency on $dual(\mathcal{P})$ is strictly tighter than arc consistency on $hidden(\mathcal{P})$; i.e., $ac(dual) \succ ac(hidden)$.*

3.3 Beyond arc consistency

Because the dual and hidden transformations are binary CSPs, we can enforce local consistency properties that are defined only over binary constraints. Following [5], a binary CSP is (i,j)-consistent if it is not empty and any consistent assignment over i variables can be extended to a consistent assignment involving j additional variables. A CSP is arc consistent (AC) if it is (1,1)-consistent. A CSP is path consistent (PC) if it is (2,1)-consistent. A CSP is strongly path consistent (SPC) if it is (i,1)-consistent for each $1 \leq i \leq 2$. A CSP is path inverse consistent (PIC) if it is (1,2)-consistent. A CSP is neighborhood inverse consistent (NIC) if any instantiation of a single variable x can be extended to a consistent assignment over all the variables that are constrained with x, called the *neighborhood* of x. A CSP is restricted path consistent (RPC) if it is arc consistent and whenever there is a value of a variable that is consistent with just one value of an adjoining variable, every other variable has a value that is compatible with this pair of values. A CSP is singleton arc consistent (SAC) if it is not empty, and the CSP induced by any instantiation of a single variable is not empty after enforcing arc consistency.

Debruyne and Bessière [5] demonstrated that, SPC $\succ_{DB}$ SAC $\succ_{DB}$ PIC $\succ_{DB}$ RPC $\succ_{DB}$ AC, and NIC $\succ_{DB}$ PIC, where "$\succ_{DB}$" is the ordering relation defined in their paper, as discussed in Section 3. Thus, by Lemma 1 we immediately have that SPC $\succeq$ SAC $\succeq$ PIC $\succeq$ RPC $\succeq$ AC, and NIC $\succeq$ PIC.

Theorem 3 *Given a CSP $\mathcal{P}$, neighborhood inverse consistency on $hidden(\mathcal{P})$ is equivalent to arc consistency on $hidden(\mathcal{P})$; i.e., $nic(hidden) \simeq ac(hidden)$.*

Proof: Since $nic \succeq ac$, we immediately have $nic(hidden) \succeq ac(hidden)$. Conversely, suppose $ac(hidden(\mathcal{P}))$ is not empty. For a dual variable c, its neighborhood in $ac(hidden(\mathcal{P}))$ is $vars(c)$. Thus an instantiation of c with a tuple t from its domain in $ac(hidden(\mathcal{P}))$ can be extended to a consistent assignment of its neighboring variables, 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 has constraints with dual variables. An instantiation of x with a value a from its domain in $ac(hidden(\mathcal{P}))$ can be extended to a consistent assignment including all its neighborhood, where for each of the dual 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 transformation is not empty after enforcing neighborhood inverse consistency; i.e., $nic(hidden(\mathcal{P}))$ is not empty. In fact, $ac(hidden(\mathcal{P}))$ is already neighborhood inverse consistent. ∎

Because neighborhood inverse consistency on the hidden transformation collapses down onto arc consistency those consistencies that are weaker than neighborhood inverse consistency but tighter than arc consistency, e.g., path inverse consistency and restricted path consistency, are also equivalent to arc consistency. That is, $nic(hidden) \simeq pic(hidden) \simeq rpc(hidden) \simeq ac(hidden)$ (and by the equivalence $ac(hidden) \simeq ac$ established in Theorem 1, each of these local consistencies on the hidden transformation is in turn also equivalent to arc consistency on the original formulation).

However, neighborhood inverse consistency on the dual transformation does not collapse. It is strictly tighter than arc consistency. In fact, the even weaker restricted path consistency is strictly tighter than arc consistency on the dual.

Example 6 Consider a CSP with three Boolean variables and constraints:

$$\begin{aligned} C_1(x_1, x_2) &= \{(0,0),(1,1)\}, \\ C_2(x_2, x_3) &= \{(0,0),(1,1)\}, \\ C_3(x_1, x_3) &= \{(0,1),(1,0)\}. \end{aligned}$$

The dual transformation of this problem is arc consistent but not restricted path consistent (RPC). Furthermore, enforcing RPC on the dual transformation yields an empty CSP. Thus we have that $rpc(dual) \succ ac(dual)$; i.e., rpc is strictly tighter on the dual.

Along with the previous orderings this immediately gives that both $pic(dual) \succ ac(dual)$, and $nic(dual) \succ ac(dual)$.

Although neighborhood inverse consistency and path inverse consistency on the hidden transformation do not provide any additional power over arc consistency, the same is not true for singleton arc consistency.

Example 7 Consider a CSP with three parity constraints: even($x_1+x_2+x_3$), even($x_1+x_3+x_4$), and even($x_1+x_2+x_4$). If x_1 is set to 1, and the other variables have domain $\{0, 1\}$ then the hidden transformation is arc consistent but not singleton arc consistent. Furthermore, enforcing singleton arc consistency on this problem yields an empty CSP. Thus we have $sac(hidden) \succ ac(hidden)$ and $sac(hidden) \succ nic(hidden)$ (since $nic(hidden) \simeq ac(hidden)$).

Theorem 4 *Given a CSP $\mathcal{P}$, singleton arc consistency on $dual(\mathcal{P})$ is tighter than singleton arc consistency on $hidden(\mathcal{P})$; i.e., $sac(dual) \succeq sac(hidden)$.*

Proof: First we define a function τ from the arc consistent subdomains of $dual(\mathcal{P})$ to subdomains of $hidden(\mathcal{P})$. Let $\mathcal{D}$ be an arc consistent subdomain of $dual(\mathcal{P})$. In $\tau(\mathcal{D})$ each dual variable c will have the same domain as it did in $\mathcal{D}$, $dom^{\mathcal{D}}(c) = dom^{\tau(\mathcal{D})}(c)$, and the domain of each ordinary variable x, $dom^{\tau(\mathcal{D})}(x)$, is set to be equal to $\pi_{\{x\}} dom^{\mathcal{D}}(c)$ for some dual variable c such that $x \in vars(c)$. τ is well defined, as from Lemma 3 we know that since $\mathcal{D}$ is an arc consistent subdomain of $dual(\mathcal{P})$, $dom^{\tau(\mathcal{D})}(x)$ is independent of which dual variable c we choose to project.

τ has three relevant properties. (1) $\tau(\mathcal{D})$ is an arc consistent subdomain of $hidden(\mathcal{P})$. For each ordinary variable x we have that $dom^{\tau(\mathcal{D})}(x) = \pi_{\{x\}} dom^{\mathcal{D}}(c)$ for every dual variable c that x is constrained with. Thus, every value of x has a support in each of the dual variables it is constrained with (take one of the tuples whose projection was that value), and every tuple t of every dual variable c has a support in each of the ordinary variables it is constrained with (take the projection of t on that variable). (2) If $\mathcal{D}'$ is a subdomain of $\mathcal{D}$, then $\tau(\mathcal{D}')$ is a subdomain of $\tau(\mathcal{D})$. This is obvious from the definition of τ. (3) $\tau(\mathcal{D})$ is an empty subdomain, if and only if $\mathcal{D}$ is an empty subdomain.[4] The only non-trivial case is when $\tau(\mathcal{D})$ contains an empty domain for an ordinary variable x. But in that case there must be a dual variable c with $x \in vars(c)$ such that $\pi_{\{x\}} dom^{\mathcal{D}}(c) = \emptyset$. And this can only be the case if $dom^{\mathcal{D}}(c)$ is itself empty; i.e., $\mathcal{D}$ contains an empty domain.

Now we show that if $sac(dual(\mathcal{P}))$ is not empty then neither is $sac(hidden(\mathcal{P}))$; i.e., $sac(dual) \succeq sac(hidden)$. Let $\mathcal{D}_d = \mathcal{D}^{sac(dual(\mathcal{P}))}$ and $\mathcal{D}_h = \tau(\mathcal{D}_d)$. Our claim is that $\mathcal{D}_h$ is a non-empty singleton arc consistent subdomain of $hidden(\mathcal{P})$.

First, since $sac(dual(\mathcal{P}))$ is arc consistent and non-empty, $\mathcal{D}^{sac(dual(\mathcal{P}))} = \mathcal{D}_d$ is an arc consistent and non-empty subdomain of $dual(\mathcal{P})$. Thus, $\mathcal{D}_d$ is a non-empty member of the domain of the function τ and by (3) $\mathcal{D}_h = \tau(\mathcal{D}_d)$ is non-empty and we need only prove that it is singleton arc consistent.

Let c be a dual variable, and t a tuple in $dom^{\mathcal{D}_h}(c)$. We must show that $\mathcal{D}_h|_{c \leftarrow t}$ (i.e., $\mathcal{D}_h$ in which $dom^{\mathcal{D}_h}(c)$ has been reduced to the singleton $\{t\}$) contains a non-empty

[4] By Definition 3 a subdomain is empty if it contains an empty domain for some variable.

arc consistent subdomain. However, since $\mathcal{D}_d$ is singleton arc consistent, $\mathcal{D}_d|_{c\leftarrow t}$ contains a non-empty arc consistent subdomain $ac(\mathcal{D}_d|_{c\leftarrow t})$. $\tau(ac(\mathcal{D}_d|_{c\leftarrow t}))$ is easily seen to be a subdomain of $\mathcal{D}_h|_{c\leftarrow t}$, and thus by (1) and (3) $\tau(ac(\mathcal{D}_d|_{c\leftarrow t}))$ must be a non-empty arc consistent subdomain of $\mathcal{D}_h|_{c\leftarrow t}$. On the other hand, let x be an ordinary variable and a a value in its domain. To show that $\mathcal{D}_h|_{x\leftarrow a}$ contains a non-empty arc consistent subdomain, we choose any dual variable c that x is constrained with, and a tuple t from the domain of c such that $t[x] = a$. Now if we consider $\mathcal{D}_d|_{c\leftarrow t}$ and its non-empty arc consistent subdomain $ac(\mathcal{D}_d|_{c\leftarrow t})$, we can similarly show that $\tau(ac(\mathcal{D}_d|_{c\leftarrow t}))$ is a non-empty arc consistent subdomain of $\mathcal{D}_h|_{x\leftarrow a}$. ∎

In the hidden transformation, for each pair of dual variables c_i and c_j, where $vars(c_i) \cap vars(c_j) \neq \emptyset$, enforcing strong path consistency adds a constraint between c_i and c_j. This constraint ensures that a tuple from c_i agrees with a tuple from c_j on the shared ordinary variables. The constraint is identical to the dual constraint between c_i and c_j in the dual transformation. Thus, strong path consistency on the hidden transformation must be as strong as on the dual. In fact, we can show their equivalence.

Theorem 5 *Given a CSP $\mathcal{P}$, strong path consistency on $hidden(\mathcal{P})$ is equivalent to strong path consistency on $dual(\mathcal{P})$; i.e., $spc(hidden) \simeq spc(dual)$.*

Proof: See [4]. ∎

Figure 4 summarizes our results. If there is a directed path between consistency properties $\mathcal{A}$ and $\mathcal{B}$, then $\mathcal{A}$ is tighter than $\mathcal{B}$. If the path contains a strictly tighter than link then $\mathcal{A}$ is strictly tighter than $\mathcal{B}$. Note that some of the relationships between consistency properties are not completely characterized. For example, it is an open question whether or not $sac(hidden) \succeq sac(dual)$.

4 Backtracking Algorithms

In this section, we compare the performance of three backtracking algorithms—the chronological backtracking algorithm, the forward checking algorithm, and the maintaining (generalized) arc consistency algorithm—when solving the original formulation and the dual and hidden transformations of a problem. Our results are proven under the assumption that a backtracking algorithm finds all solutions to a problem.

Given two backtracking algorithms and two formulations of a problem we identify whether or not the relation "one combination can be only polynomially worse than another combination" holds. To formalize this relation we must first specify quantitative measures of the size of a CSP and the cost of solving a CSP using a given algorithm.

We denote by $\mathrm{size}(\mathcal{P})$ the size of a CSP $\mathcal{P}$. Consistent with real-world practice, we assume that the domains of the variables are represented extensionally and that the constraints are represented intensionally. Thus, to specify the variables, domains, and constraints of a (possibly transformed) CSP $\mathcal{P}$ takes $O(n + nd + mr)$ space, where n denotes the number of variables, d the size of the largest domain, m the number of constraints, and r the arity of the largest constraint scheme in $\mathcal{P}$. Since the dual and the hidden are transformations of an original (non-binary) formulation $\mathcal{P}$, we can also

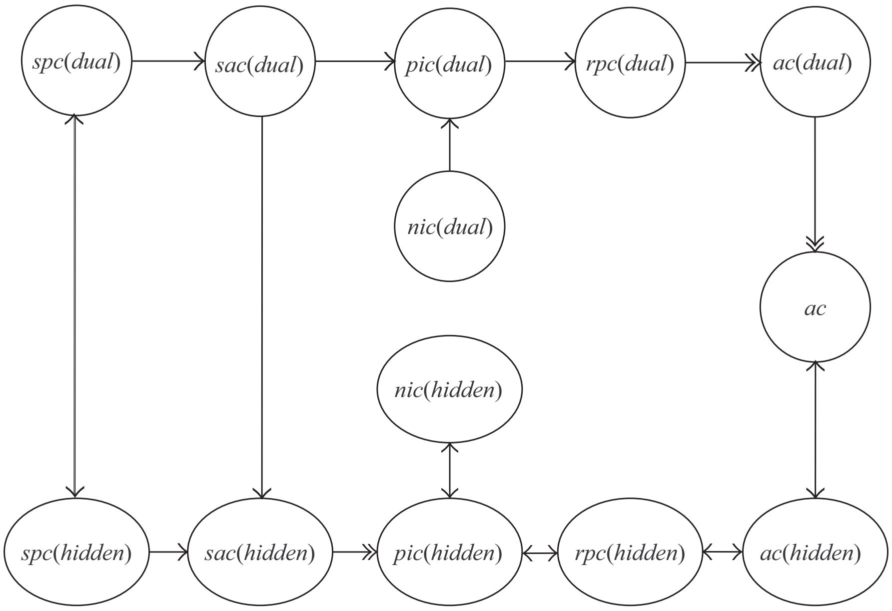

Figure 4: The hierarchy of relations between consistencies on the original, dual, and hidden formulations. A bi-directional arrow is equivalence, $\simeq$, a double headed arrow is the strictly tighter relation, $\succ$, and an ordinary arrow is the tighter relation, $\succeq$.

specify their sizes in terms of the parameters of $\mathcal{P}$. In particular, let n, d, m, and r be the parameters of an original (non-binary) formulation $\mathcal{P}$. In the worst case, the transformation $dual(\mathcal{P})$ takes $O(m + md^r + m^2)$ space and the transformation $hidden(\mathcal{P})$ takes $O((n + m) + (nd + md^r) + mr)$ space. Thus, the dual and hidden transformations can require space that grows exponentially with the arity of the constraints in the original formulation. In practice, one would certainly want to limit the arity of the constraints to which these transformations are applied.

To solve a CSP with a backtracking algorithm, one must specify the variable ordering the algorithm uses to determine which variable to instantiate next. It is well known that the variable ordering used can have an exponential effect on the cost of solving a CSP. Thus an exponentially difference in performance between two algorithm/formulation pairs is always trivially possible under particular variable orderings. Hence, to formalize a sensible notion of "only polynomially worse" we must do so in a way that is independent of any particular variable ordering. In our definitions we achieve this independence by quantifying over all possible orderings.

Formally, we define a variable ordering function ν to be a mapping from a tuple t (a node making a, possibly empty, set of variable-value assignments) and a CSP $\mathcal{P}$ to a new variable not in $vars(t)$. We say that a variable ordering ν is an ordering for a problem $\mathcal{P}$ if it is defined over the variables of $\mathcal{P}$. In addition, we say that an algorithm A uses the variable ordering ν if ν characterizes the choices made by A at the various nodes A visits as it does its backtracking search; i.e., A next instantiates the variable x when it is at node n if and only if $\nu(n) = x$.

We denote by $\mathrm{cost}(\mathsf{A}, \mathcal{P}, \nu)$ the cost of solving a CSP $\mathcal{P}$ using an algorithm A and a variable ordering ν. This cost is determined by the number of nodes visited by the algorithm and the cost at each node. In turn, the cost at each node is determined by the cost of enforcing the local consistency property maintained by the backtracking algorithm and the cost, if any, of determining the next variable to instantiate (the cost of the function ν).

Definition 10 Let A-$\mathcal{A}$ denote algorithm A applied to problems transformed by a transformation $\mathcal{A}$. Given two backtracking algorithms A and B (possibly identical), and two transformations $\mathcal{A}$ and $\mathcal{B}$ for CSP problems (perhaps identity transformations),

1. A-$\mathcal{A}$ can be *only polynomially worse* than B-$\mathcal{B}$, written A-$\mathcal{A} \underset{\text{poly}}{\prec} \mathsf{B}$-$\mathcal{B}$, iff given any CSP $\mathcal{P}$ and variable ordering $\nu_{\mathcal{B}}$ for $\mathcal{B}(\mathcal{P})$, there is a variable ordering $\nu_{\mathcal{A}}$ for $\mathcal{A}(\mathcal{P})$ such that,

$$\frac{\mathrm{cost}(\mathsf{A}, \mathcal{A}(\mathcal{P}), \nu_{\mathcal{A}})}{\mathrm{cost}(\mathsf{B}, \mathcal{B}(\mathcal{P}), \nu_{\mathcal{B}})} \leq \text{polynomial function of } \min(\mathrm{size}(\mathcal{A}(\mathcal{P})), \mathrm{size}(\mathcal{B}(\mathcal{P})).$$

   That is, the cost of solving $\mathcal{A}(\mathcal{P})$ using A and $\nu_{\mathcal{A}}$ is at *most* a polynomial factor worse than the cost of solving $\mathcal{B}(\mathcal{P})$ using B and $\nu_{\mathcal{B}}$.

2. A-$\mathcal{A}$ can be *superpolynomially worse* than B-$\mathcal{B}$, written A-$\mathcal{A} \underset{\text{superpoly}}{\prec} \mathsf{B}$-$\mathcal{B}$, iff A-$\mathcal{A} \underset{\text{poly}}{\not\prec} \mathsf{B}$-$\mathcal{B}$ (i.e., the negation of polynomially worse), iff there exists a CSP $\mathcal{P}$ and a variable ordering $\nu_{\mathcal{B}}$ for $\mathcal{B}(\mathcal{P})$, such that for any variable ordering $\nu_{\mathcal{A}}$ for

$\mathcal{A}(\mathcal{P})$,

$$\frac{\mathrm{cost}(\mathsf{A}, \mathcal{A}(\mathcal{P}), \nu_{\mathcal{A}})}{\mathrm{cost}(\mathsf{B}, \mathcal{B}(\mathcal{P}), \nu_{\mathcal{B}})} > \text{superpolynomial function of } \min(\mathrm{size}(\mathcal{A}(\mathcal{P})), \mathrm{size}(\mathcal{B}(\mathcal{P}))).$$

That is, the cost of solving $\mathcal{A}(\mathcal{P})$ using A and $\nu_{\mathcal{A}}$ is at *least* a superpolynomial factor worse than the cost of solving $\mathcal{B}(\mathcal{P})$ using B and $\nu_{\mathcal{B}}$.

To prove a relationship $\mathsf{A}\text{-}\mathcal{A} \underset{\text{poly}}{\preceq} \mathsf{B}\text{-}\mathcal{B}$, we (1) establish that the number of nodes visited by A on $\mathcal{A}$ is at most a polynomial factor more than the number of nodes visited by B on $\mathcal{B}$, (2) establish that the time complexity of enforcing the local consistency property maintained by A at each node is at most a polynomial factor worse than the time complexity of enforcing the local consistency property maintained by B at each node, and (3) establish that the time complexity of $\nu_{\mathcal{A}}$ is at most a polynomial factor worse than the time complexity of $\nu_{\mathcal{B}}$. In turn, to prove condition (1), we establish a correspondence between the nodes in the ordered search tree generated by $\nu_{\mathcal{A}}$ that are visited by A and the nodes in the ordered search tree generated by $\nu_{\mathcal{B}}$ that are visited by B. In the definition of $\mathsf{A}\text{-}\mathcal{A} \underset{\text{poly}}{\preceq} \mathsf{B}\text{-}\mathcal{B}$, the existential (there is a variable ordering $\nu_{\mathcal{A}}$) occurs within the scope of two universal quantifiers (any CSP $\mathcal{P}$ and any variable ordering $\nu_{\mathcal{B}}$) and thus $\nu_{\mathcal{A}}$ can depend on both $\mathcal{P}$ and $\nu_{\mathcal{B}}$. To prove condition (3), we show how to construct $\nu_{\mathcal{A}}$ given only polynomial access to $\nu_{\mathcal{B}}$ and polynomial additional computation.

4.1 Discussion

Every variable ordering ν for a CSP instance $\mathcal{P}$ generates a complete ordered search tree for $\mathcal{P}$. Starting with the root of the tree being the empty tuple, at every node n we apply ν to that node to obtain the variable $x = \nu(n)$ that is to be instantiated by the children of n. Then the children of n will be all of the possible extensions of n that can be made by assigning x values from its domain. This process is continued recursively until we reach nodes that assign all of the variables of the CSP instance. Our formalization of variable orderings encompasses both static variable orderings (in which ν returns the same variable at every node that assigns the same number of variables) and dynamic orderings (in which ν can return a different variable at each node). Further, it assumes a static (possibly heuristically derived) of the *values* of each variable. The children of a node are thus ordered in the search tree left to right following this ordering.[5]

If we apply A to solve $\mathcal{P}$ using ν as the variable ordering, then A will search in this ordered search tree. Due to the constraints imposed by ν on A's operation, A cannot visit a node (assignment) not in this ordered search tree. Depending on how A operates, it will visit some subset of the nodes in this search tree. We refer to the sub-tree (of the ordered search tree) visited by a backtracking algorithm on the original formulation as

[5]Our results would go through unchanged if the value ordering is dynamic assuming that all children of a given node are instantiations of the same variable. All that would be needed is for ν to return in addition to the next variable an ordering over its domain. However, we avoid doing this since it would make the notation unnecessarily complicated.

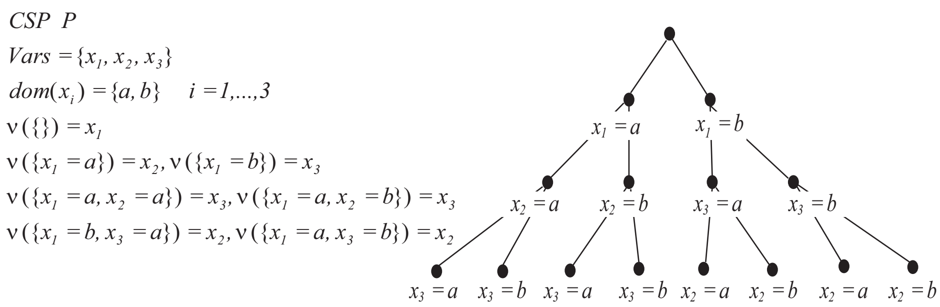

Figure 5: The ordered search tree generated by a variable ordering.

the *original search tree*, the sub-tree visited when solving the dual transformation as the *dual search tree*, and the one visited when solving the hidden transformation as the *hidden search tree*. All of these sub-trees are defined with respect to particular variable orderings (each of which generates a particular ordered search tree). Figure 5 shows an example of a CSP instance $\mathcal{P}$, variable ordering ν for $\mathcal{P}$, and the complete search tree generated by ν.

As we have defined them, a variable ordering can play either a descriptive or a prescriptive role. Say that we run A on $\mathcal{A}(\mathcal{P})$ and we use some heuristic function to compute the next variable to instantiate at every node of the search tree. This heuristic function could use various pieces of the state of the program in computing its answer. For example, in an algorithm like FC, the minimum remaining values heuristic uses the domain sizes of the uninstantiated variables in computing the next variable. In this case, ν plays a descriptive role. After A has run on $\mathcal{A}(\mathcal{P})$ there will be some set of variable ordering decisions that it has made that can be captured by specifying a variable ordering function ν, and we can say that A has used ν when solving $\mathcal{A}(\mathcal{P})$. Note that there will in general be many different variable orderings that describe A's behavior on $\mathcal{A}(\mathcal{P})$. In particular, A will only visit a subset of the possible tuples that could be generated from the variables and values of $\mathcal{A}(\mathcal{P})$, and ν need only agree with A on those tuples A actually visits. How ν maps the other tuples to variables can be decided arbitrarily.

On the other hand, ν can also be used in a prescriptive role. If ν can be computed externally to A, then whenever A visits a node n it can invoke the computation of $\nu(n)$ to tell it what variable to instantiate next.

In our definitions the variable ordering for B-$\mathcal{B}$ is descriptive while the variable ordering for A-$\mathcal{A}$ is prescriptive. In particular, we have that B achieves some level of performance when solving $\mathcal{B}(\mathcal{P})$, and that the variable ordering it used to achieve this performance is described by $\nu_{\mathcal{B}}$. Then when we use A to solve $\mathcal{A}(\mathcal{P})$ we assume that there is a variable ordering $\nu_{\mathcal{A}}$ that can be computed externally to A to prescribe the variable ordering it should use when solving $\mathcal{A}(\mathcal{P})$. The definitions specify conditions on A's performance on $\mathcal{A}(\mathcal{P})$ under all possible $\nu_{\mathcal{A}}$.

The relation A-$\mathcal{A} \underset{\text{superpoly}}{\prec}$ B-$\mathcal{B}$ means that there is a problem $\mathcal{P}$ such that when B solves $\mathcal{B}(\mathcal{P})$ using a variable ordering described by $\nu_{\mathcal{B}}$ it achieves a performance that is super-polynomially better than that of A on $\mathcal{A}(\mathcal{P})$ no matter what variable ordering is prescribed for A to use.

The relation A-$\mathcal{A} \underset{\text{poly}}{\preceq}$ B-$\mathcal{B}$ means that for any problem $\mathcal{P}$ the performance achieved by B when solving $\mathcal{B}(\mathcal{P})$ using a variable ordering described by $\nu_{\mathcal{B}}$ can always be matched within a polynomial by A when solving $\mathcal{A}(\mathcal{P})$ using a prescribed variable ordering $\nu_{\mathcal{A}}$.

Two potentially problematic cases arise from the fact that each algorithm utilizes a different variable ordering. First, if B used an exponential computation to compute its variable ordering, then it would seem to be unfair to compare A's performance with it—B might have unmatchable performance simply due to its superior variable ordering. Second, if A used exponential resources to compute its variable ordering, then it would seem to be unfair to say that it was polynomially as good as B—it could be that B needed only polynomial resources to compute its ordering and yet A needed exponential resources to achieve similar performance. Both of these problems are resolved by our use of a ratio of costs as the metric for comparison. In particular, in the first case A would also be allowed to use exponential resources to compute its variable ordering, and in the second case if A used exponentially more resources than B in computing its variable ordering then the "only polynomially worse" relation would not hold.

We have used quantification as a mechanism for removing any dependence on a particular variable ordering in our definitions. Quantification allows us to achieve a number of useful properties.

First, we need to compare the performance of algorithms on problems that contain different sets of variables. For example, the original formulation and the dual transformation contain completely different sets of variables. Hence, it is not possible to simply assume that the same variable ordering is used in each algorithm, as is commonly done. By quantifying over possible variable orderings we have the freedom to allow each algorithm to employ a different variable ordering.

Second, since different variable orderings can yield such tremendous differences in practice, it is not desirable to fix the variable ordering used by an algorithm independently of the problem. By quantifying over all possible variable orderings we do not need to fix the variable ordering.

Finally, another seemingly plausible way of comparing algorithm and problem formulation combinations is to compare their performance when they are both using the *best* possible variable ordering. That is, to look at $\frac{\text{cost}(\mathsf{A},\mathcal{A}(\mathcal{P}),\nu_{\mathcal{A}})}{\text{cost}(\mathsf{B},\mathcal{B}(\mathcal{P}),\nu_{\mathcal{B}})}$ under the condition that $\nu_{\mathcal{A}}$ is the best possible variable ordering for A on problem $\mathcal{A}(\mathcal{P})$, and similarly for $\nu_{\mathcal{B}}$. However, the practical impact of such an approach would be limited since determining the best possible variable ordering for a given algorithm and problem combination is at least as hard as solving the problem itself. With quantification we achieve something that is both stronger and more useful. In particular, it is easy to see that if A-$\mathcal{A} \underset{\text{poly}}{\preceq}$ B-$\mathcal{B}$ holds, it then also holds if we restrict our attention the best possible variable ordering for each combination. The advantage of our stronger formulation is that it tells us something about many different variable orderings, not only the best ones, and thus our results have a much greater practical impact. For example, if we have A-$\mathcal{A} \underset{\text{poly}}{\preceq}$ B-$\mathcal{B}$ then no matter what variable ordering we use for B we know that there exists a variable ordering for A that will achieve a comparable performance. Importantly, the variable ordering for A need not be the best possible; in fact, in most of our proofs of this relation we show a way of constructing the ordering for A from the ordering for B.

4.2 Forward checking algorithm (FC)

In this section, we compare the performance of the forward checking backtracking algorithm (FC) [10, 15, 25] on the three models.

We can make things simpler by restricting the class of variable orderings that we need to consider for FC-hidden (FC applied to the hidden transformation). In particular, we assume that any variable ordering for FC-hidden always instantiates all of the ordinary variables prior to instantiating any dual variable. Due to the following result this restriction does not affect either of the two relations $\underset{\text{poly}}{\prec}$ or $\underset{\text{superpoly}}{\prec}$.

Theorem 6 *Given a CSP $\mathcal{P}$ and a variable ordering ν for the hidden transformation $hidden(\mathcal{P})$, we can construct (in polynomial time given polynomial time access to ν) a new variable ordering ν' that instantiates all of the ordinary variables of $hidden(\mathcal{P})$ prior to instantiating any dual variable such that FC-hidden using ν' visits at most $O(rd)$ times as many nodes as it visits when using ν, where d is the size of the largest domain and r is the arity of the largest constraint scheme in $\mathcal{P}$.*

Proof: See [4]. ∎

In fact we can go even further, and assume that FC-hidden explores a search tree containing the ordinary variables only. Using one of the above restricted variable orderings, FC-hidden will have instantiated all ordinary variables prior to instantiating any dual variable. Due to the nature of the constraints in the hidden, once all of the ordinary variables have been consistently instantiated, there will be only one consistent tuple in the domain of each dual variable; FC will prune away all of the other, inconsistent, tuples. FC will then proceed to descend in a backtrack free manner down the single remaining branch to instantiate all of the dual variables. Thus we can stop the search once all of the ordinary variables have been assigned—we already have a solution. That is, we need only consider the top part of the search tree where the ordinary variables are being instantiated, and we can consider FC-hidden and FC-orig to be exploring the same search tree consisting of all of the ordinary variables.

We now present examples that allow us to prove some relationships between the three problem formulations.

Example 8 Consider a non-binary CSP with only one constraint over n Boolean variables, $C(x_1, \ldots, x_n) = \{x_1 = x_2 = \cdots = x_n\}$. FC applied to this formulation visits $O(2^n)$ nodes and performs $O(2^n)$ consistency checks to find all solutions irrespective of the variable ordering used. There are only two nodes in the search tree for FC-dual, representing the two possible solutions. FC-hidden visits $O(n)$ nodes and performs $O(n)$ consistency checks.

Theorem 7 *FC-orig can be super-polynomially worse than FC-dual and FC-hidden; i.e., FC-orig* $\underset{\text{superpoly}}{\prec}$ *FC-dual and FC-orig* $\underset{\text{superpoly}}{\prec}$ *FC-hidden.*

Proof: See Example 8. ∎

Example 9 Consider the non-binary CSP with n Boolean variables $x_1, \ldots, x_n$ and n constraints given by $\{x_1\}$, $\{\neg x_1 \vee x_2\}$, $\{\neg x_1 \vee \neg x_2 \vee x_3\}, \cdots$, $\{\neg x_1 \vee \cdots \vee \neg x_{n-1} \vee x_n\}$. FC applied to this formulation visits n nodes and performs $2n$ consistency checks when using the static variable ordering $x_1, \ldots, x_n$. FC-hidden performs at least $2^n - 1$ consistency checks irrespective of the variable ordering used, as the domain of the dual variable associated with the constraint $\{\neg x_1 \vee \cdots \vee \neg x_{n-1} \vee x_n\}$ costs that much to filter once any one of the ordinary variables is instantiated.

Theorem 8 *FC-hidden can be super-polynomially worse than FC-orig; i.e., FC-hidden* $\underset{\text{superpoly}}{\prec}$ *FC-orig.*

Proof: See Example 9. ∎

Example 10 Consider a CSP with $2n$ variables, $x_1, \ldots, x_{2n}$, each with domain $\{1, \ldots, n\}$, and n constraints,

$$
\begin{array}{rcl}
C_1(x_1, x_2, x_{n+1}) & = & \{x_1 = x_2\}, \\
C_2(x_2, x_3, x_{n+2}) & = & \{x_2 = x_3\}, \\
 & \cdots & \\
C_{n-1}(x_{n-1}, x_n, x_{2n-1}) & = & \{x_{n-1} = x_n\}, \\
C_n(x_1, x_n, x_{2n}) & = & \{x_1 \neq x_n\}.
\end{array}
$$

The problem is insoluble because the first $n-1$ constraints force $x_1 = x_n$ and the last constraint forces $x_1 \neq x_n$. Note that in each of the above constraints, the variable x_{n+i} merely increases the arity and the number of tuples of the constraint. Given the lexicographic static variable ordering $x_1, \ldots, x_{2n}$, FC-orig and FC-hidden will search n paths, $\{x_1 \leftarrow 1, \ldots, x_n \leftarrow 1\}$, $\{x_1 \leftarrow 2, \ldots, x_n \leftarrow 2\}, \ldots$, $\{x_1 \leftarrow n, \ldots, x_n \leftarrow n\}$: at each node, there is only one value in the domain of the current variable that is consistent with every uninstantiated variable. Thus FC-orig and FC-hidden visit $O(n^3)$ nodes to conclude that the problem is insoluble. However, irrespective of the variable ordering used, along those paths where all of the x_i are set to the same value, FC-dual has to instantiate at least $\log(n) - 1$ dual variables to reach a dead-end. In particular, it must instantiate enough of the dual variables $c_1, \ldots, c_{n-1}$ to allow it to conclude that $x_1 = x_n$, which will then yield a contradiction with the last dual variable c_n. However, even under the restriction that each of the variables x_i get the same value, FC-dual must additionally "instantiate" a variable from $x_{n+1}, \ldots, x_{2n}$ at each stage that has no influence on the failure. This will cause it to backtrack uselessly to try n different ways of setting each dual variable using different values for these variables

The best variable ordering strategy for FC-dual along these paths is to repeatedly split the set of ordinary variables $x_1, \ldots, x_n$ by instantiating the dual variable over the mid-point variables, so as to most quickly derive a relation between x_1 and x_n.

For example, FC-dual would first branch on the dual variable corresponding to the constraint $C(x_{\frac{n}{2}}, x_{\frac{n}{2}+1}, x_{n+\frac{n}{2}})$, thus instantiating the two mid-point variables in the sequence $x_1, \ldots, x_n$. In the next two branches it would split the subproblems involving $x_1, \ldots, x_{\frac{n}{2}}$ and $x_{\frac{n}{2}+1}, \ldots, x_n$. Continuing this way it can instantiate all of the variables $x_1, \ldots, x_n$ by instantiating $O(\log(n))$ dual variables and thus conclude that $x_1 = x_n$ to obtain a contradiction. Once failure has been detected FC must then backtrack and try the other $O(n)$ consistent values of each dual variable. Hence, FC-dual has to explore at least $O(n^{\log(n)-1})$ nodes.

Theorem 9 *FC-dual can be super-polynomially worse than FC-orig and FC-hidden; i.e., FC-dual* $\underset{\text{superpoly}}{\prec}$ *FC-orig and FC-dual* $\underset{\text{superpoly}}{\prec}$ *FC-hidden.*

Proof: See Example 10. ∎

Now we turn to the relation between FC-dual and FC-hidden. In Example 8, we observe that FC-hidden visits $O(n)$ times as many nodes as FC-dual does. As we now show, this bound is true in general. We then show that FC-hidden $\underset{\text{poly}}{\prec}$ FC-dual.

Let $\mathcal{P}$ be any CSP instance, and let ν_{dual} be any variable ordering for $dual(\mathcal{P})$. We must show that there exists a variable ordering ν_{hidden} such that the performance of FC on $hidden(\mathcal{P})$ using ν_{hidden} is within a polynomial of its performance on $dual(\mathcal{P})$ using ν_{dual}. First, we show how to construct ν_{hidden} using ν_{dual}, then we show that under this variable ordering a polynomial bound is achieved.

Let $n = \{z_1 \leftarrow a_1, \ldots, z_k \leftarrow a_k\}$ be a sequence of assignments to ordinary variables; i.e., a possible node in the search tree explored by FC-hidden. We need to compute $\nu_{hidden}(n)$; i.e., the next variable to be assigned by FC-hidden when and if it visits n. This will be the variable instantiated by the children of n. Once we can compute $\nu_{hidden}(n)$ for any node n, we will have determined the function ν_{hidden}, and hence the ordered search tree generated by ν_{hidden} and searched by FC-hidden. To do this we establish a correspondence between the nodes in the ordered search tree generated by ν_{dual}, T_{dual}, and nodes that would be in the ordered search tree generated by ν_{hidden}, T_{hidden}. Using Theorem 6 we can consider T_{hidden} to be an ordered search tree over only the ordinary variables (i.e., the original variables of $\mathcal{P}$).

The correspondence is based on the simple observation that every assignment to a dual variable c by FC-dual corresponds to a sequence of assignments to the ordinary variables in $vars(c)$. Each node n_d in T_{dual} is a sequence of assignments to dual variables. Let this sequence of assignments be $\{c_1 \leftarrow t_1, \ldots, c_k \leftarrow t_k\}$, where t_i is a tuple of values in the domain of the dual variable c_i. Each dual variable represents a constraint (from the original formulation $\mathcal{P}$) over some set of ordinary variables. Let $vars(c_i) = \{z_1^{c_i}, \ldots, z_{r_i}^{c_i}\}$ be the set of ordinary variables associated with the dual variable c_i with r_i being the arity of c_i. Assigning c_i a value assigns a value to every variable in $vars(c_i)$. Given a lexicographic ordering of the ordinary variables, $c_i \leftarrow t_i$ thus corresponds to a sequence of assignments to the ordinary variables in $vars(c_i)$: $\{z_1 \leftarrow t_i[z_1], \ldots, z_{r_i} \leftarrow t_i[z_{r_i}]\}$, where $t[x]$ is the value that tuple t assigns to variable x. Thus we can convert each node n_d in T_{dual}, $n_d = \{c_1 \leftarrow t_1, \ldots, c_k \leftarrow t_k\}$ into a sequence of assignments to ordinary variables $\{z_1^{c_1} \leftarrow t_1[z_1^{c_1}], \ldots, z_{r_1}^{c_1} \leftarrow t_1[z_{r_1}^{c_1}], \ldots, z_1^{c_k} \leftarrow t_k[z_1^{c_k}], \ldots, z_{r_k}^{c_k} \leftarrow t_k[z_{r_k}^{c_k}]\}$.

If no ordinary variable is assigned two different values in this sequence, then for each variable in the sequence we can delete all but its first assignment. This will yield a non-repeating sequence of assignments to ordinary variables. The fundamental property of ν_{hidden} is that each such non-repeating sequence generated by the nodes of T_{dual} will be a node in the ordered search tree T_{hidden}. We define a function, $d{\to}h$ from nodes of T_{dual} to the nodes of T_{hidden}. Given a node n_d of T_{dual}, $d{\to}h(n_d)$ is the non-repeating sequence of assignments to ordinary variables generated as just described. If n_d contains two different assignments to an ordinary variable, then we leave $d{\to}h(n_d)$ undefined (in this case there is no corresponding node in T_{hidden}).

One thing to note about the function $d{\to}h$ is that if n_1 and n_2 are nodes of T_{dual} such that $d{\to}h$ is defined on both nodes and n_1 is an ancestor of n_2 then $d{\to}h(n_1)$ will be a sub-sequence of or will be equal to $d{\to}h(n_2)$. This means that either $d{\to}h(n_1)$ is the same node as $d{\to}h(n_2)$ or that $d{\to}h(n_1)$ is an ancestor of $d{\to}h(n_2)$ in T_{hidden}.

Example 11 For example, the sequence of assignments $n_d = \{c_1(x_1, x_2, x_5) \leftarrow (0, 1, 0),\ c_2(x_3, x_4, x_5) \leftarrow (1, 0, 0),\ c_3(x_4, x_5) \leftarrow (0, 0)\}$, yields the sequence of assignments to ordinary variables $\{x_1 \leftarrow 0, x_2 \leftarrow 1, x_5 \leftarrow 0, x_3 \leftarrow 1, x_4 \leftarrow 0, x_5 \leftarrow 0, x_4 \leftarrow 0, x_5 \leftarrow 0\}$. No variable is assigned two different values in this sequence, so $d{\to}h(n_d) = \{x_1 \leftarrow 0, x_2 \leftarrow 1, x_5 \leftarrow 0, x_3 \leftarrow 1, x_4 \leftarrow 0\}$ is the non-repeating sequence, and this sequence of assignments will be a node in T_{hidden}. On the other hand, the sequence $n'_d = \{c_1(x_1, x_2) \leftarrow (0, 1), c_2(x_1, x_3) \leftarrow (1, 0)\}$ has two different assignments to x_1 so $d{\to}h(n'_d)$ remains undefined, and n'_d does not have a corresponding node in T_{hidden}.

To compute $\nu_{hidden}(n)$ for some node n (which could be the empty set of assignments) we start by setting $n_d = \emptyset$; i.e. the root of T_{dual}. At each iteration $d{\to}h(n_d)$ will be a prefix of (or sometimes equal to) n and we will extend n_d so that it covers more of n until $d{\to}h(n_d)$ is a super-sequence of n. To extend n_d we examine $\nu_{dual}(n_d)$. Let $\nu_{dual}(n_d) = c$. c is the next dual variable assigned by the children of n_d. Let $\vec{z} = \{z_1, \ldots, z_k\}$ be the lexicographically ordered sequence of ordinary variables that will be *newly* assigned by c (these are the variables in $vars(c)$ that have not already been assigned in n_d), and let $\vec{y} = \{y_1, \ldots, y_i\}$ be the sequence of variables assigned by n that come after $d{\to}h(n_d)$. (Note that either of, or both of, these sequences could be empty.) There are three possibilities.

1. These two sequences do not match (i.e., neither is a prefix of the other). In this case n does not have a matching node in T_{dual}, and we can terminate the process and choose $\nu_{hidden}(n)$ arbitrarily.

2. $\vec{z}$ is a sub-sequence of or is equal to $\vec{y}$. In this case n assigns a value to each of the variables z_i. In addition, all of the other variables in $vars(c)$ (the ones that are not newly assigned by c) will have been assigned by previous assignments in n. Thus there is only one tuple of values that can be assigned to c that will be compatible with n. Let this tuple be t. We check that t is in $dom(c)$. If it is, then n_d will have a child that makes the new assignment $c \leftarrow t$, and we extend n_d by including that new assignment (i.e., we move n_d to this single matching child). $d{\to}h(n_d)$ will still be a prefix of n and we can continue to the next iteration.

Otherwise, if $t \notin dom(c)$, then n has no matching node in T_{dual} and we can terminate the process and choose $\nu_{hidden}(n)$ arbitrarily.

3. $\vec{z}$ is a strict super-sequence of $\vec{y}$. In this case we set $\nu_{hidden}(n)$ to be the first ordinary variable z_i in $\vec{z}$ that is unassigned by n.

The time complexity of the above process for computing ν_{hidden} is at most a polynomial factor worse than the time complexity of ν_{dual}, as it only requires polynomial time access to ν_{dual}. (In particular, we do not need to do any search in T_{dual} to compute ν_{hidden}, rather we only need to follow one path of T_{dual}.)[6]

Given a problem $hidden(\mathcal{P})$, the variable ordering ν_{hidden} generates an ordered search tree T_{hidden}. Using $d{\rightarrow}h$, we can define an "inverse" function $h{\rightarrow}d$ that maps nodes of T_{hidden} to nodes of T_{dual}. The function is used in the proofs of Lemma 4 and Theorem 11. If n is a node in T_{hidden} (i.e., a non-repeating sequence of assignments to ordinary variables that occurs in T_{hidden}) then $h{\rightarrow}d(n)$ is the first node of T_{dual} in a breadth-first ordering of T_{dual} that makes all of the same assignments as n. Formally, $h{\rightarrow}d(n)$ is defined to be the *shallowest (closest to the root) and leftmost* node $n_d \in T_{dual}$ such that $d{\rightarrow}h(n_d)$ is a super-sequence of n (not necessarily proper). If there is no such node in T_{dual} then $h{\rightarrow}d(n)$ is undefined.

Example 12 Consider the ordered search tree T_{dual} shown in Figure 6 and the corresponding ordered search tree T_{hidden} generated by ν_{hidden} shown in Figure 7. $h{\rightarrow}d(\{x_1 \leftarrow a, x_2 \leftarrow a, x_3 \leftarrow b\}) = n_3$ (nodes in the diagram include all assignments made along the arcs from the node to the root of the tree). Similarly, $h{\rightarrow}d(x_1 \leftarrow a, x_2 \leftarrow a, x_3 \leftarrow b, x_4 \leftarrow b) = n_3$. So additional assignments in T_{hidden} do not move to new nodes in T_{dual} if the dual node contains extra ordinary variables. On the other hand, $h{\rightarrow}d(x_1 \leftarrow a, x_2 \leftarrow a, x_3 \leftarrow a, x_4 \leftarrow a) = n_2$, while $h{\rightarrow}d(x_1 \leftarrow a, x_2 \leftarrow a, x_3 \leftarrow a, x_4 \leftarrow a, x_5 \leftarrow b) = n_8$. So an additional assignment can move down through many nodes in T_{dual}, as many as is needed to find the first descendant that assigns the new variable. In this case we had to move down two levels to find a node assigning x_5.

Lemma 4 *The number of nodes in T_{hidden} that are mapped to the same node in T_{dual} by the function $h{\rightarrow}d$ is at most r, the maximum arity of a constraint in the original formulation.*

Proof: Say that the nodes $n_1, \ldots, n_k$ in T_{hidden} are all different and yet $h{\rightarrow}d(n_1) = \cdots = h{\rightarrow}d(n_k) = n_d$. By the definition of $h{\rightarrow}d$, $d{\rightarrow}h(n_d)$ must be a super-sequence of all of these nodes. Hence, they must in fact all be of different lengths, and we can consider them to be arranged in increasing length. We also see that n_i must be a sub-sequence of n_j if $j \geq i$. $d{\rightarrow}h(n_d)$ might be equal to n_k, but it must be a proper super-sequence of all of the other nodes. Furthermore, if p_n is n_d's parent (in T_{dual}) then

[6] This is an important, albeit technical point. The number of tuples in the domain of a dual variable can be exponential in n. So a node n_d might have an exponential number of child nodes. However, in this procedure we need never search the child nodes to find the correct extension of n_d. In case (2) we always know the tuple of assignments that must be the extension and we can test whether or not this extension exists with a single constraint check. In case (3) we can compute the next variable to assign directly from the constraint's scheme without looking at the possible assignments to the constraint.

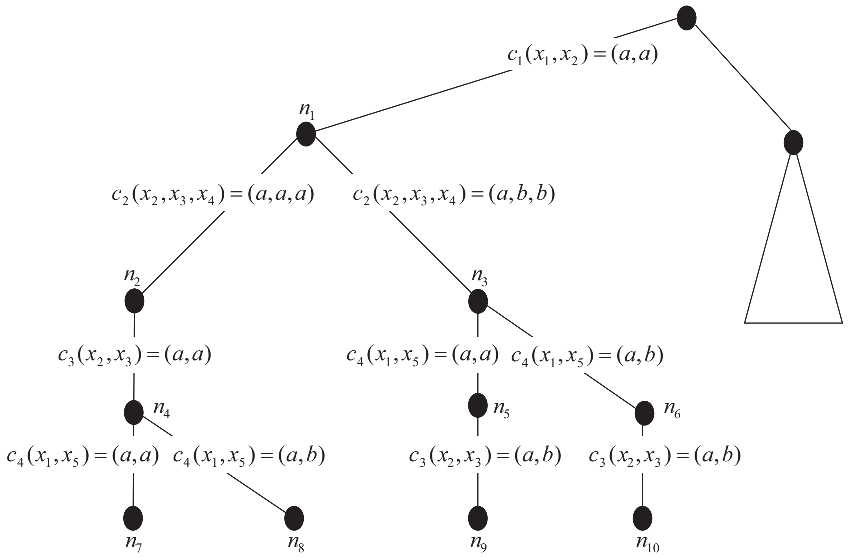

Figure 6: A sample ordered search tree for FC-dual

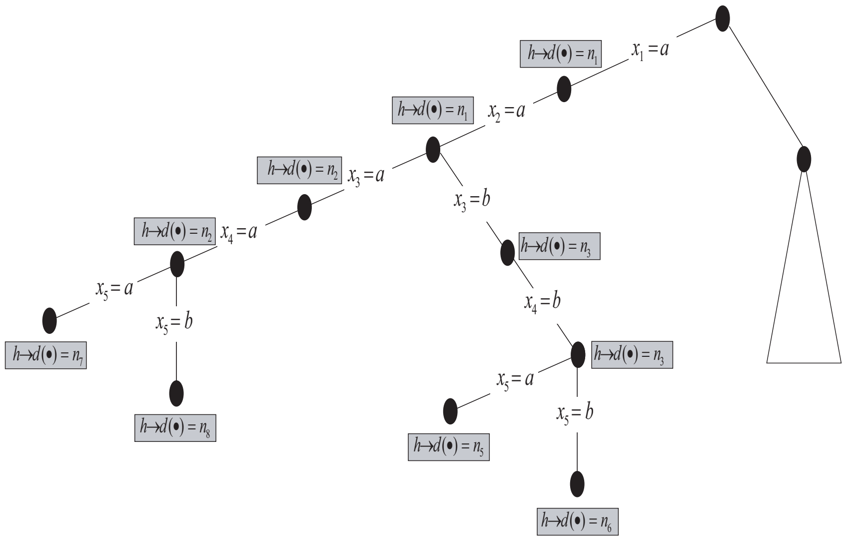

Figure 7: The ordered search tree for FC-hidden that arises from Figure 6. The nodes that are the values of the function $h{\to}d$ refer to the ordered search tree of Figure 6.

$d{\rightarrow}h(p_n)$ must be a proper sub-sequence of all of these nodes. (Otherwise, $h{\rightarrow}d(n_1)$ would not be n_d as p_d would have been a shallower node satisfying $h{\rightarrow}d$'s definition.) Since n_d instantiates only one more dual variable than p_d, $d{\rightarrow}h(n_d)$ can be at most r assignments (to ordinary variables) longer than $d{\rightarrow}h(p_d)$. Putting these constraints together we see that the sequence of nodes $n_1, \ldots, n_k$ can only be at most r long. ∎

The final component we need to prove that FC-hidden $\precsim_{\text{poly}}$ FC-dual is to recall a characterization of the nodes visited by FC that is due to Kondrak and van Beek [12]. Given a CSP $\mathcal{P}$ and a tuple of assignments t, we say that t is *consistent with a variable* if t can be extended to a consistent assignment including that variable. It is easy to see that if an assignment t is consistent with every variable, any subtuple $t' \subseteq t$ is also consistent with every variable.

Theorem 10 (Kondrak and van Beek [12]) *FC visits a node n (in the ordered search tree it is exploring) if and only if n is consistent and its parent is consistent with every variable.*

From this result it follows that if a node n is visited by FC and then FC subsequently visits a child of n then n must be consistent with every variable. That is, all parent nodes in the search tree explored by FC must be consistent with every variable.

Theorem 11 *FC-hidden can be only polynomially worse than FC-dual; i.e., FC-hidden $\precsim_{\text{poly}}$ FC-dual.*

Proof: Using the formalism just developed we must first show that the number of nodes visited by FC-hidden (FC solving $hidden(\mathcal{P})$) using ν_{hidden} is within a polynomial of the number of nodes visited by FC-dual (FC solving $dual(\mathcal{P})$) using ν_{dual}.

That FC-hidden uses ν_{hidden} means that it searches in the tree T_{hidden}, and similarly FC-dual searches in the tree T_{dual}. If n is a parent node in the sub-tree of T_{hidden} that is visited by FC, then by Theorem 10 n must be consistent with every variable. We claim that for every such parent node $h{\rightarrow}d(n)$ is defined, and that FC-dual visits $h{\rightarrow}d(n)$ in T_{dual}. We prove this claim by induction.

The base of the induction is when n is the root of T_{hidden}. In this case $h{\rightarrow}d(n)$ is the root of T_{dual}, and FC-dual visits it. Let n be a node that is consistent with every variable, and let p be n's parent. p is also consistent with every variable, as we observed above. Thus by induction $h{\rightarrow}d(p) = p_d$ is defined, and FC-dual visits it. Note that all of p_d's ancestors must map to proper sub-sequences of p (and n) under $d{\rightarrow}h$, since p_d is the shallowest node to map to a super-sequence of p. We have two cases.

1. $d{\rightarrow}h(p_d)$ is a proper super-sequence of p. n must next assign the ordinary variable $\nu_{hidden}(p)$, which by construction must be the first ordinary variable not assigned by p in the sequence $d{\rightarrow}h(p_d)$. Furthermore, all of p_d's siblings assign values to the same dual variable. Thus they all assign the same sequence of new ordinary variables as p_d, and in particular p_d and all of its siblings must assign all of the ordinary variables assigned by n.

   Let the last dual variable assigned by p_d be c. Since n is consistent with every variable, there must be at least one tuple t in $dom(c)$ that is consistent with n.

Furthermore, since p_d's parent makes fewer assignments to ordinary variables than does n, p_d's parent must also be consistent with $c \leftarrow t$. (Remember that the dual constraints only require agreement on the shared ordinary variables). Hence, p_d's parent must have a child node making the assignment $c \leftarrow t$, this child node is itself consistent, and FC-dual must visit this child node. All such child nodes (siblings of p_d) yield super-sequences of n under the mapping $d \mapsto h$ and they are the shallowest nodes to do so: their parent maps to a proper subsequence of n. We may take the leftmost such child to be $h \mapsto d(n)$, and we have also shown that FC-dual visits $h \mapsto d(n)$.

2. $d \mapsto h(p_d) = p$. In this case p and thus n assigns every ordinary variable in the dual variables of p_d. Starting at p_d we can descend in the tree T_{dual}. At each stage we examine the dual variable c assigned by the children of p_d. If n assigns all of the variables of c, then there can be at most one tuple t in $dom(c)$ that is consistent with n. Furthermore, since n is consistent with every variable, including c, such a tuple must exist. Since p_d makes fewer assignments to ordinary variables than n it too must be consistent with every variable, with $c \leftarrow t$, and no other value in $dom(c)$. Thus since FC-dual visits p_d it will visit the child n_d of p_d making the assignment $c \leftarrow t$ and no other children. Now we set p_d to be n_d and repeat the argument until we set p_d to be a node that makes a super-set (or the same set) of the assignments made by n. At this point, we are back to case one above.

So we have that for each parent node n in the sub-tree visited by FC-hidden, there is a corresponding node $h \mapsto d(n)$ in the sub-tree visited by FC-dual. Furthermore, at most r nodes can be mapped to the same node by $h \mapsto d$ (by Lemma 4), so there are at most r times as many parent nodes visited by FC-hidden as nodes visited by FC-dual. Now, since each parent node in T_{hidden} can have at most d children (only ordinary variables are instantiated by FC-hidden and d is the maximum number of values these variables possess) the total number of nodes visited by FC-hidden is at most d times the number of parent nodes visited, and at most rd times the number of nodes FC-dual visits. To complete the proof, we note that both FC-dual and FC-hidden filter only dual variables and at most m of them when forward checking at a node and that therefore the number of consistency checks performed at each node are polynomially related. ∎

We summarize the relations between FC on the different formulations in Figure 8.

4.3 Maintaining (generalized) arc consistency algorithm (MAC)

In this section, we compare the performance of an algorithm that maintains (generalized) arc consistency or really-full lookahead [8, 16, 20] on the three models. We refer to the algorithm as MAC; namely, maintaining generalized arc consistency algorithm.

We begin by characterizing the nodes visited by MAC. The algorithm enforces arc consistency on the CSP induced by the current assignment (see Definition 5). Given a CSP and an assignment t, we say t is arc consistent if the CSP induced by t is not empty after enforcing arc consistency. It is easy to see that if an assignment t is arc consistent, any subtuple $t' \subseteq t$ is also arc consistent.

Theorem 12 *MAC will visit a node n (in the ordered search tree it is exploring) if and only if n's parent is arc consistent and the value assigned to the current variable by n has not been removed from its domain when enforcing arc consistency on n's parent.*

Proof: We prove this by induction on the length of n. The claim is vacuously true when n is the empty sequence of assignments. Say that n is of length k, and let p be n's parent. If n is visited then it is clear that p must have yielded a non-empty CSP after MAC enforced arc consistency on it; i.e., p must have been arc consistent, and the new assignment made at n must have survived arc consistency. Conversely, suppose p was arc consistent and that n survives enforcing arc consistency at p. Since p is itself arc consistent, it must have had an arc consistent parent, and it must have survived the enforcement of arc consistency at its parent. Hence, by induction MAC will have visited p. But then once MAC visits p and enforces arc consistency there, it will have a non-empty CSP in which n survived. Thus it must continue on to visit n. ∎

Note this also means that MAC will visit a node n (in the ordered search tree it is exploring) if n is arc consistent (as then its parent would be arc consistent, and it would have survived the enforcement of arc consistency at its parent).

Just as in the case of FC-hidden, MAC-hidden does not need to instantiate all the variables in order to find a solution. A variable ordering for MAC-hidden that instantiates all the ordinary variables first is at worst polynomially bounded compared to any other variable ordering strategy.

Theorem 13 *Given a CSP $\mathcal{P}$ and a variable ordering ν for the hidden transformation $hidden(\mathcal{P})$, we can construct (in polynomial time given polynomial time access to ν) a new variable ordering ν' that instantiates all of the ordinary variables of $hidden(\mathcal{P})$ prior to instantiating any dual variable such that MAC-hidden using ν' visits at most $O(r)$ times as many nodes as it visits when using ν, where r is the arity of the largest constraint scheme in $\mathcal{P}$.*

Proof: See [4]. ∎

Given the above, we can again assume that MAC-hidden only instantiates the ordinary variables during its search.

We know from Theorem 1 that arc consistency on the hidden transformation is equivalent to arc consistency on the original formulation. Because MAC-orig and MAC-hidden explore the same search tree, we expect they should visit the same nodes.

Lemma 5 *Given an assignment t of a CSP $\mathcal{P}$, $ac(\mathcal{P}|_t)$ is empty iff $ac(hidden(\mathcal{P})|_t)$ is empty. Furthermore, for each ordinary variable x, x has the same domain in $ac(\mathcal{P}|_t)$ and $ac(hidden(\mathcal{P})|_t)$.*

Proof: Since $ac(\mathcal{P}|_t)$ is arc consistent, $ac(\mathcal{P}|_t)$ is empty iff $hidden(ac(\mathcal{P}|_t))$ is empty (by Lemma 2) iff $ac(hidden(\mathcal{P}|_t))$ is empty (by Theorem 1). Furthermore, for each ordinary variable x we have that $dom^{ac(\mathcal{P}|_t)}(x) = dom^{ac(hidden(\mathcal{P}|_t))}$ (by Corollary 1). Now consider the two problems $hidden(\mathcal{P}|_t))$ and $hidden(\mathcal{P})|_t$. In $\mathcal{P}|_t$ (Definition 5) the domain of each ordinary variable $x \in vars(t)$ is reduced to the singleton value assigned to that variable by t, $\{t[x]\}$. The domains of the other variables are unaffected. However, all of the constraints of $\mathcal{P}|_t$ have also been reduced so that they contain only tuples over the reduced variable domains. Thus the hidden transformation $hidden(\mathcal{P}|_t)$ will contain dual variables in which any tuple incompatible with t has been removed. In $hidden(\mathcal{P})|_t$ on the other hand, the dual variables will not be reduced, they will contain the same set of tuples as the original constraints of $\mathcal{P}$. However, every tuple in the domain of a dual variable c in $hidden(\mathcal{P})|_t$ that is not in the domain of c in $hidden(\mathcal{P}|_t)$ must be arc inconsistent. It assigns an ordinary variable a value that no longer exists in the domain of that variable. So enforcing arc consistency will remove all of these tuples. Clearly if we sequence arc consistency so that we remove these values first, we will first transform $hidden(\mathcal{P})|_t$ to $hidden(\mathcal{P}|_t)$ after which the continuation of arc consistency enforcement must yield the same final CSP. That is, $ac(hidden(\mathcal{P}|_t)) = ac(hidden(\mathcal{P})|_t)$, and thus $ac(\mathcal{P}|_t)$ is empty iff $ac(hidden(\mathcal{P})|_t)$ is empty. Furthermore, for every variable $dom^{\,ac(hidden(\mathcal{P}|_t))} = dom^{ac(hidden(\mathcal{P})|_t)}$, and thus for every ordinary variable $dom^{\,ac(\mathcal{P}|_t)}(x) = dom^{ac(hidden(\mathcal{P})|_t)}$. ∎

Theorem 14 *MAC-orig can be only polynomially worse than MAC-hidden, and vice versa; i.e., MAC-orig* $\preceq_{\text{poly}}$ *MAC-hidden and MAC-hidden* $\preceq_{\text{poly}}$ *MAC-orig.*

Proof: Since MAC-orig and MAC-hidden search over the same variables, the same variable ordering can be used by both formulations. Let ν be the variable ordering used and let T_ν be the ordered search tree induced by ν for a CSP $\mathcal{P}$. Both MAC-orig and MAC-hidden will search in T_ν. We show that they both visit the same nodes in T_ν. The parent p of node n is arc consistent in $\mathcal{P}$ iff $ac(\mathcal{P}|_p)$ is not empty (by definition) iff $ac(hidden(\mathcal{P})|_p)$ is not empty (by Lemma 5) iff p is arc consistent in $hidden(\mathcal{P})$. Furthermore, once we enforce arc consistency at p in $\mathcal{P}$ the domains of all of the uninstantiated (ordinary) variables will be the same as their domains in $ac(\mathcal{P}|_p)$ which will be the same as their domains in $ac(hidden(\mathcal{P})|_p)$ (by Lemma 5). Thus n will survive arc consistency enforcement at p in $\mathcal{P}$ iff it survives arc consistency enforcement at p in $hidden(\mathcal{P})$. Hence, by Theorem 12, MAC will visit n in $\mathcal{P}$ iff MAC visits n in $hidden(\mathcal{P})$.

To complete the proof, we note that arc consistency on a CSP $\mathcal{P}$ takes $O(md^r)$ time in the worst case and arc consistency on $hidden(\mathcal{P})$ takes $O(mrd^{r+1})$ time [2], where d, m, and r denote the size of the largest domain, the number of constraints, and the arity of the largest constraint scheme in $\mathcal{P}$, respectively. ∎

The following example shows that MAC-orig and MAC-hidden can be exponentially better than MAC-dual.

Example 13 Consider a CSP with $n + n(n-1)/2$ variables $x_1, \ldots, x_n$, and $y_1, \ldots, y_{n(n-1)/2}$, each with domain $\{1, \ldots, n-1\}$, and $n(n-1)/2$ constraints,

$$\begin{aligned}
C(x_1, x_2, y_1) &= \{x_1 \neq x_2\}, \\
C(x_1, x_3, y_2) &= \{x_1 \neq x_3\}, \\
&\cdots \\
C(x_{n-1}, x_n, y_{n(n-1)/2}) &= \{x_{n-1} \neq x_n.\}
\end{aligned}$$

It is a pigeon-hole problem with an extra variable y_i in each constraint. The pigeon-hole problem is insoluble but highly locally consistent. MAC-orig and MAC-hidden have to instantiate at least $n-2$ variables before an induced CSP is empty after enforcing arc consistency and they visit $O(n!)$ nodes to conclude the problem is insoluble. It can be seen that MAC-dual has the same pruning power as MAC-orig because each pair of original constraints share at most one variable. However, at each node of the dual search tree, MAC-dual has to additionally instantiate a variable y_i, which has no influence on the failure. As a result, MAC-dual has to explore a factor of $O(n^n)$ more nodes and is thus exponentially worse than MAC-orig and MAC-hidden.

The following example shows the converse: if two original constraints share more than one variable, arc consistency on the dual is tighter than on the original formulation, and MAC-dual can be super-polynomially better than MAC-orig and MAC-hidden.

Example 14 Consider a CSP with $4n+2$ variables, $x_1, \ldots, x_{4n+2}$, each with domain $\{1, \ldots, n\}$, and $2n+1$ constraints,

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

$(x_1 + x_2 \bmod 2) = 0$ implies $(x_3 + x_4 \bmod 2) = 1$ implies $(x_5 + x_6 \bmod 2) = 0$ implies ... implies $(x_{4n+1} + x_{4n+2} \bmod 2) = 0$. But then the last constraint also implies $(x_1 + x_2 \bmod 2) = 1$. Thus the problem is insoluble. When enforcing arc consistency at a node in the original search tree, no values 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 formulation 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 $O(\log(n))$ and the number of nodes explored by MAC-orig and MAC-hidden is $O(n^{\log(n)})$. In the dual transformation, on the other hand, the dual constraints form a cycle in the constraint graph. Once a dual variable is instantiated, the cycle is broken so that the induced CSP 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. MAC-dual is therefore super-polynomially better than MAC-orig and MAC-hidden.

Theorem 15 *MAC-dual can be super-polynomially worse than MAC-orig and MAC-hidden; i.e., MAC-dual* $\underset{\text{superpoly}}{\prec}$ *MAC-orig and MAC-dual* $\underset{\text{superpoly}}{\prec}$ *MAC-hidden.*

Proof: See Example 13. ∎

Theorem 16 *MAC-orig and MAC-hidden can be super-polynomially worse than MAC-dual; i.e., MAC-orig* $\underset{\text{superpoly}}{\prec}$ *MAC-dual and MAC-hidden* $\underset{\text{superpoly}}{\prec}$ *MAC-dual.*

Proof: See Example 14. ∎

MAC-dual can be super-polynomially better because it enforces a stronger consistency on the dual transformation and MAC-hidden can be super-polynomially better because it makes fewer instantiations at each stage during the backtracking search.

Figure 8 summarizes our results. For completeness, we summarize in the diagram our results for the chronological backtracking algorithm (BT). However, for reasons of length, we do not present the proofs of these results. Such proofs, using an alternative formalization, can be found in [4]. As can be seen, BT-dual can be only polynomially worse than BT-hidden, and vice versa. On the other hand, BT-dual and BT-hidden can be super-polynomially worse than BT-orig, and vice versa.

Also included in the diagram are results due to Kondrak and van Beek [12] between different algorithms applied to the same problem formulation. For example, consider the relation FC-hidden $\underset{\text{poly}}{\prec}$ BT-hidden. Since the same problem is being solved, Kondrak and van Beek's result that FC always visits the same or fewer nodes than BT, can be directly applied.[7] Then, since arc consistency is $O(md^2)$ and forward checking is $O(md)$ for binary problems with m constraints and domain size d, it follows that FC-hidden $\underset{\text{poly}}{\prec}$ BT-hidden. Interestingly, although it is easily shown that MAC-orig always visits the same or fewer nodes than FC-orig, we have that MAC-orig $\underset{\text{superpoly}}{\prec}$ FC-orig, since there exist problems where MAC-orig can perform exponentially more constraint checks than FC-orig.

Furthermore, we can use properties of $\underset{\text{poly}}{\prec}$ to draw additional conclusions from the diagram. Whenever we are comparing formulations that are all polynomially related in size the relation $\underset{\text{poly}}{\prec}$ is transitive. Thus, for example, since the hidden and dual transformations (although exponentially larger than the original formulation) are polynomially related in size, from MAC-hidden $\underset{\text{poly}}{\prec}$ FC-hidden and FC-hidden $\underset{\text{poly}}{\prec}$ FC-dual, we can conclude that MAC-hidden $\underset{\text{poly}}{\prec}$ FC-dual. Similarly, from MAC-dual $\underset{\text{poly}}{\prec}$ FC-dual and FC-dual $\underset{\text{poly}}{\prec}$ BT-dual, we can conclude that MAC-dual $\underset{\text{poly}}{\prec}$ BT-dual. Note that the $\underset{\text{superpoly}}{\prec}$ relation is not transitive. So, for example, we cannot conclude that since FC-hidden $\underset{\text{superpoly}}{\prec}$ FC-orig and FC-orig $\underset{\text{superpoly}}{\prec}$ FC-dual we have that FC-hidden $\underset{\text{superpoly}}{\prec}$ FC-dual. In fact, as shown in Theorem 11, FC-hidden $\underset{\text{poly}}{\prec}$ FC-dual.

5 Conclusion

We compared three possible models for a constraint satisfaction problem—the original formulation, the dual transformation, and the hidden transformation—with respect to the effectiveness of various local consistency properties and the performance of three different backtracking algorithms. To our knowledge, this is the first comprehensive attempt to evaluate constraint modeling techniques in a formal way.

We studied arc consistency on the original formulation, and its dual and hidden transformations. We showed that arc consistency on the dual transformation is tighter

[7] Kondrak and van Beek prove their results for static variable orderings. However, their results also hold when the algorithm searches a fixed ordered search tree, as is allowed by the definition of $\underset{\text{poly}}{\prec}$.

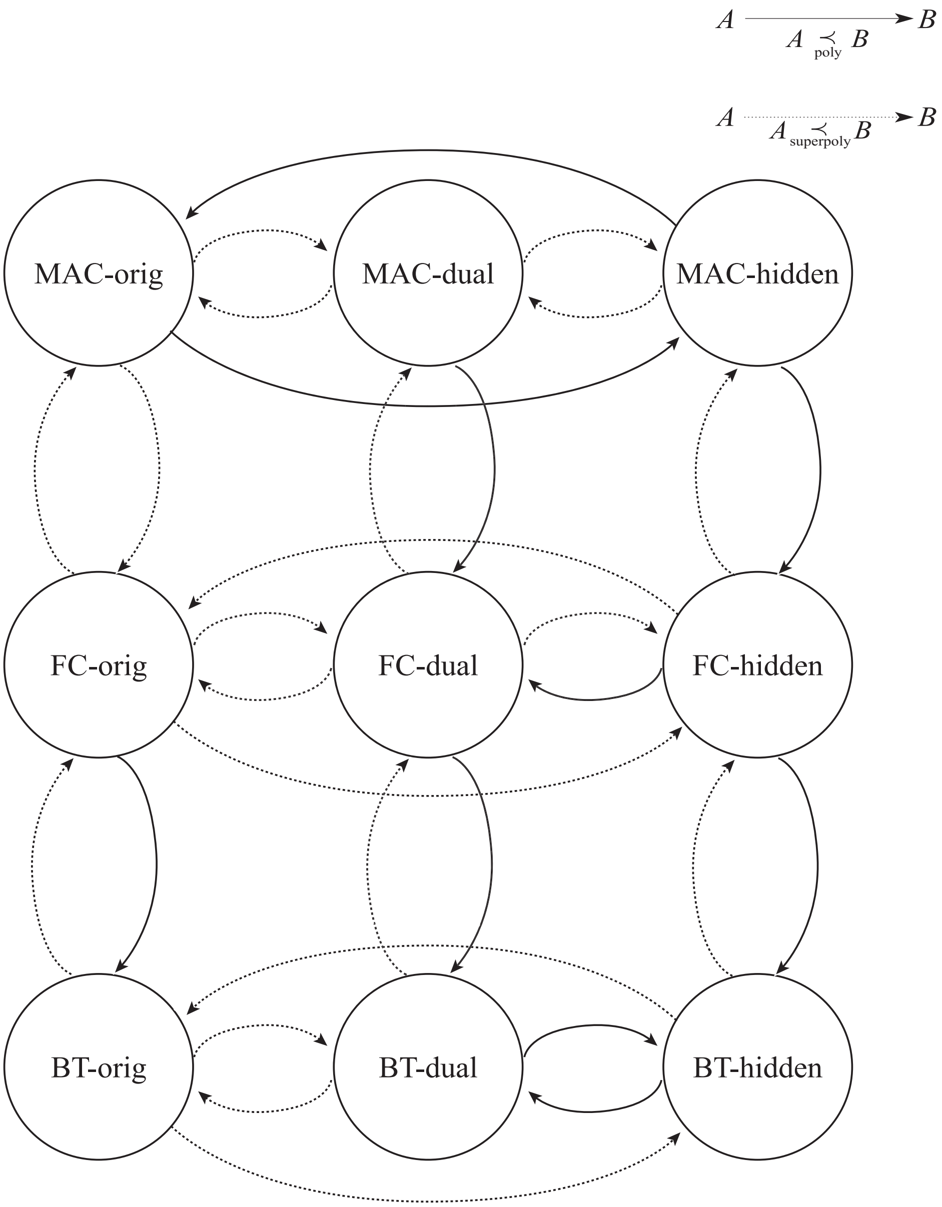

Figure 8: Summary of the relations between the combinations of algorithms and formulations. A solid directed edge from A-$\mathcal{A}$ to B-$\mathcal{B}$ means A-$\mathcal{A} \underset{\text{poly}}{\prec} \mathsf{B}$-$\mathcal{B}$ and a dashed directed edge means A-$\mathcal{A} \underset{\text{superpoly}}{\prec} \mathsf{B}$-$\mathcal{B}$.

than arc consistency on the original formulation, which itself is equivalent to arc consistency on the hidden transformation. We then considered local consistencies that are defined over binary constraints. For example, we showed that singleton arc consistency on the dual is tighter than singleton arc consistency on the hidden.

We then compared the performance of three different backtracking algorithms on a non-binary CSP and on its dual and hidden transformations. Considering the forward checking algorithm, FC-dual can be super-polynomially worse than FC-orig and FC-hidden, and FC-orig can be super-polynomially worse than FC-dual and FC-hidden. However, the cost to solve FC-hidden can be at most a polynomial factor worse than the cost to solve FC-dual. Turning to the algorithm that maintains arc consistency, MAC-orig and MAC-hidden visit the same nodes and have the same cost at each node, while MAC-dual can be super-polynomially worse than MAC-orig and MAC-hidden because it may have to make many more instantiations at each node of the search tree. Furthermore, MAC-orig and MAC-hidden can be super-polynomially worse than MAC-dual because MAC-dual enforces a stronger consistency property than MAC-orig or MAC-hidden do.

Our results can be used by practitioners to help build efficient models for real-world constraint satisfaction problems. Our objective is to provide some general guidelines as to whether or not, or under which conditions, the dual or hidden transformation should be applied to a non-binary CSP. For example, if the performance of formulation $\mathcal{A}$ is bounded by a polynomial from formulation $\mathcal{B}$ but can be super-polynomially better than $\mathcal{B}$, then we are assured that the performance of $\mathcal{A}$ cannot be much worse than that of $\mathcal{B}$, and that furthermore $\mathcal{A}$ has the potential to provide a dramatic improvement over $\mathcal{B}$. Thus, $\mathcal{A}$ may be preferred in the hope that it can provide super-polynomial savings over $\mathcal{B}$ and given that in the worst case, it cannot lose too much. On the other hand, if two formulations are equivalent for a certain backtracking algorithm, there is little to be gained from developing both models.

References

[1] 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*, pages 311–318, Madison, Wisconsin, 1998.

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

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

[4] X. Chen. *A Theoretical Comparison of Selected CSP Solving and Modeling Techniques*. PhD thesis, University of Alberta, Canada, 2000.

[5] R. Debruyne and C. Bessière. Domain filtering consistencies. *J. Artificial Intelligence Research*, 14:205–230, 2001.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

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

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

[17] B. A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5:16–23, 1990.

[18] 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.

[19] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. Technical Report ACT-AI-222-89, MCC, Austin, Texas, 1989. A shorter version appears in *ECAI-90*, pages 550-556.

[20] 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.

[21] H. Simonis. Standard models for finite domain constraint solving. Tutorial presented at PACT'97 Conference, London, UK, 1997.

[22] B. Smith, S. C Brailsford, P. M. Hubbard, and H. P. Williams. The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996.

[23] 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.

[24] 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.

[25] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[26] R. Weigel, C. Bliek, and B. Faltings. On reformulation of constraint satisfaction problems. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 254–258, Brighton, United Kingdom, 1998.