

Algorithms and Constraint Programming^{*}

Fabrizio Grandoni¹ and Giuseppe F. Italiano²

¹ Dipartimento di Informatica, Università di Roma “La Sapienza”, via Salaria 113, 00198 Roma, Italy. grandoni@di.uniroma1.it

² Dipartimento di Informatica, Sistemi e Produzione, Università di Roma “Tor Vergata”, via del Politecnico 1, 00133 Roma, Italy. italiano@disp.uniroma2.it

Abstract. Constraint Programming is a powerful programming paradigm with a great impact on a number of important areas such as logic programming [45], concurrent programming [42], artificial intelligence [12], and combinatorial optimization [46]. We believe that constraint programming is also a rich source of many challenging algorithmic problems, and cooperations between the constraint programming and the algorithms communities could be beneficial to both areas.

1 Introduction

Given a set of variables \mathcal{X} , and a set of constraints \mathcal{C} forbidding some partial assignments of variables, the NP-hard *Constraint Satisfaction Problem* (CSP) is to find an assignment of all the variables which satisfies all the constraints [37].

One of the most common ways to solve CSPs is via *backtracking*: given a partial assignment of variables, extend it by instantiating some other variable in a way compatible with the previous assignments. If this is not possible, backtrack and try a different partial assignment. This standard approach can be improved in several ways:

- **(improved search)** instead of backtracking to the previously instantiated variable, one can backtrack to the variable generating the conflict (*backjumping*), and try to avoid such conflict later (*backchecking* and *backmarking*).
- **(domain filtering)** consistency properties which feasible assignments need to satisfy can be used to filter out part of the values in the do-

^{*} This work has been partially supported by the Sixth Framework Programme of the EU under Contract Number 507613 (Network of Excellence “EuroNGI: Designing and Engineering of the Next Generation Internet”) and by MIUR, the Italian Ministry of Education, University and Research, under Project ALGO-NEXT (“Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments”).

mains, thus reducing the search space. This can be done in a preprocessing step, or during the search, both for specific and for arbitrary sets of constraints.

- **(variables/values ordering)** The order in which variables and values are considered during the search can heavily affect the time needed to find a solution. There are several heuristics to find a convenient, static or dynamic, ordering of variables and values (*fail-first*, *succeed-first*, *most-constrained*, etc.).

In this paper we focus on the last two strategies, and we show how algorithmic techniques can be helpful. In Section 3 we will describe two polynomial-time filtering algorithms. The first one, which is based on matching, can be used for filtering of the well-known `alldifferent` constraint. The second one uses techniques from dynamic algorithms to speed up the filtering based on *inverse-consistency*, a consistency property which can be applied to arbitrary sets of binary constraints.

In Section 4 we will present an exact (exponential-time) algorithm to solve any CSP asymptotically faster than with trivial enumeration. As we will see, the improved running time is heavily based on the way variables and values are instantiated. However, in this case the approach is not heuristic: the running time is guaranteed on any instance.

2 Preliminaries

Let $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$ be a set of n variables. Given $x \in \mathcal{X}$, by $D(x)$ we denote the domain of x . From now on we will assume that each domain is discrete and finite.

An *assignment* of a variable $x \in \mathcal{X}$ is a pair (x, a) , with $a \in D(x)$, whose meaning is that x is assigned value a . A *constraint* C is a set of assignments of different variables:

$$C = \{(x_{i(1)}, a_1), (x_{i(2)}, a_2) \dots (x_{i(h)}, a_h)\}.$$

Constraint C is said to be *satisfied* by an assignment of the variables if there exists one variable $x_{i(j)}$ such that $x_{i(j)} \neq a_j$, and it is said to be *violated* otherwise.

Remark 1. For ease of presentation, in this paper we use the explicit representation of constraints above. However, implicit representations are more common in practice.

Given \mathcal{X} and a set \mathcal{C} of constraints, the *Constraint Satisfaction Problem* (CSP) is to find an assignment of values to variables (*solution*) such

that all the constraints are satisfied. We only mention that there are two relevant variants of this problem:

- list all the solutions;
- find the best solution according to some objective function.

Some of the techniques we are going to describe partially extend to such cases.

A (d, p) -CSP is a CSP where each domain contains at most d values, and each constraint involves at most p variables. Without loss of generality, we can consider $(d, 2)$ -CSPs only (also called *binary* CSPs), as the following simple lemma shows.

Lemma 1. *Each (d, p) -CSP instance can be transformed into an equivalent $(\max\{d, p\}, 2)$ -CSP instance in polynomial time.*

Proof. Duplicate the variables \mathcal{X} and add a variable c for each constraint $C = \{(x_{i(1)}, a_1), (x_{i(2)}, a_2), \dots, (x_{i(h)}, a_h)\}$, $h \leq p$. The domain of c is

$$D(c) = \{(x_{i(1)} \neq a_1), (x_{i(2)} \neq a_2), \dots, (x_{i(h)} \neq a_h)\}.$$

Intuitively, assigning the value $(x_{i(j)} \neq a_j)$ to c means that constraint C is satisfied thanks to the fact that $x_{i(j)} \neq a_j$. For each such c we also add h constraints C_1, C_2, \dots, C_h of the following form:

$$C_j = \{(c, (x_{i(j)} \neq a_j)), (x_{i(j)}, a_j)\}.$$

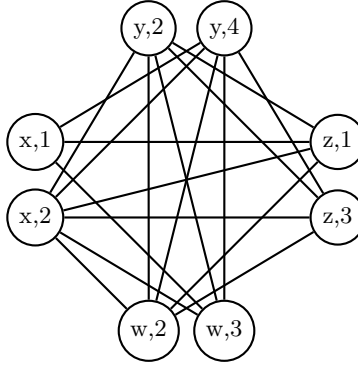
The intuitive explanation of C_j is that it cannot happen that $x_{i(j)} = a_j$ and, at the same time, constraint C is satisfied thanks to the fact that $x_{i(j)} \neq a_j$. It is not hard to see that the new problem is satisfiable if and only if the original problem is. \square

Remark 2. Dealing with non-binary constraints directly, without passing through their binary equivalent, might be convenient in some applications [44].

It is also worth to mention that there is a nice duality between variables and constraints.

Lemma 2. *Each (d, p) -CSP on n variables and m constraints can be transformed in polynomial-time into an equivalent (p, d) -CSP on m variables and n constraints.*

Figure 1 Example of consistency-graph. A solution is given by the assignments $\{(x, 2), (y, 2), (z, 1), (w, 2)\}$. The assignment $(x, 1)$ is arc consistent, while it is not path-inverse consistent.



Proof. For each constraint $C = \{(x_{i(1)}, a_1), (x_{i(2)}, a_2) \dots (x_{i(h)}, a_h)\}$, $h \leq p$, create a variable c of domain:

$$D(c) = \{(x_{i(1)} \neq a_1), (x_{i(2)} \neq a_2), \dots, (x_{i(h)} \neq a_h)\}.$$

The interpretation of c is as in Lemma 1. Now consider any variable x of the original problem. If there exists $a \in D(x)$ such that the assignment (x, a) does not conflict with any constraint, do not add any constraint for x . Note that in such case, if there is a solution, there is a solution with $x = a$. Otherwise, for each $i \in D(x) = \{1, 2, \dots, d(x)\}$, take a variable c_i such that $(x \neq i) \in D(c_i)$. Add the constraint

$$X = \{(c_1, (x \neq 1)), (c_2, (x \neq 2)), \dots, (c_{d(x)}, (x \neq d(x)))\}.$$

The intuitive explanation of X is that x must take some value in its domain. It is not hard to see that the original problem is satisfiable if and only if the original problem is. \square

Note that each $(d, 2)$ -CSP can be represented via a *consistency graph* which has a node for each possible assignment (x, a) and an edge between each pair of compatible assignments (anti-edges correspond to constraints or to multiple assignments of the same variable). Any solution corresponds to an n -clique in such graph (see Figure 1).

3 Polynomial algorithms and domain filtering

An assignment (x, a) is *consistent* if it belongs to some solution, and *inconsistent* otherwise. Deciding whether an assignment is inconsistent is an NP-hard problem (otherwise one could solve CSP in polynomial time [37]). However, it is sometimes possible to filter out (part of the) inconsistent assignments efficiently. In this section we give two examples of how algorithmic techniques can be used in the filtering process. In Section 3.1 we discuss the filtering of the well-known (non-binary) **alldifferent** constraint, which makes use of matching algorithms. In Section 3.2 we consider the filtering based on ℓ -*inverse-consistent*, suitable for any set of binary constraints, and we present a faster algorithm employing standard techniques from dynamic algorithms.

3.1 Alldifferent filtering via matching

In this paper we focus on binary constraints. However, there are families of non-binary constraints which appear very frequently in the applications. So it makes sense to design faster and more accurate filtering algorithms for them.

A relevant example is the **alldifferent** constraint, which requires that a set of variables take values different from each other. The **alldifferent** constraint is very powerful. For example with only 3 such constraints on a proper set of variables one can naturally model the well-known *n-queens problem*: place n queens on a $n \times n$ chessboard such that no two queens threaten each other (a queen threatens any other queen on the same row, column, diagonal and anti-diagonal).

The **alldifferent** constraint has the great advantage that the consistency of the assignments can be decided in polynomial time, with the following procedure by Regin [40]. Consider the bipartite graph B , which has the variables on the left side, the values on the right side, and one edge between x and a for each possible assignment (x, a) . Without loss of generality, let us assume the values available are at least as many as the variables (otherwise the problem has trivially no solution). Then any feasible solution to the original problem corresponds to a perfect bipartite matching in B , that is a matching where all the variables are matched. Luckily, we do not need to compute explicitly all the perfect bipartite matchings to determine whether a given assignment (x, a) belongs to any one of them. In fact, let M be any perfect bipartite matching. Such matching can be computed in time $O(m'\sqrt{n'})$, where n' is the number of nodes and m' the number of edges of B [29]. Let us direct all the edges in M

from right to left, and all the other edges from left to right. Then an edge $(x, a) \notin M$ belongs to some other perfect matching M' if and only if

- (x, a) belongs to an oriented cycle or
- it belongs to an even-length oriented path, starting in a free node on the right side.

We can check the two properties above for all the edges in linear time $O(n' + m')$. Altogether, we can find the subset of consistent assignments in time $O(m'\sqrt{n'})$.

In many applications the variables range over intervals. If such intervals are very large, the approach above becomes unpractical. However, there is a convenient alternative in such case: computing the largest subinterval for each variable such that both endpoints correspond to consistent assignments (*narrowing* of the intervals). Puget [38] observed that the bipartite graph B corresponding to the `alldifferent` constraint is *convex* if the variables range over intervals. Thus one can compute a perfect bipartite matching in $O(n \log n)$ time via the matching algorithm by Glover [25] for convex bipartite graphs. Puget use this observation to narrow the `alldifferent` constraint within the same time bound. Later Mehlhorn and Thiel [34] described an algorithm which takes linear time plus the time to sort the intervals endpoints. Their algorithm makes use of the union-find data structure by Gabow and Tarjan [24]. This improves on the result by Puget in all the cases where sorting can be done in linear time.

3.2 Inverse consistency and decremental clique problem

Most of the filtering techniques designed for arbitrary binary constraints are based on some kind of *local consistency property* \mathcal{P} , which all the consistent assignments need to satisfy. Enforcing \mathcal{P} -consistency is a typical dynamic process: an assignment (x, a) which is initially \mathcal{P} -consistent may become inconsistent because of the removal of some other assignment (y, b) . Thus the same (x, a) might be checked several times. Using the information gathered during the previous consistency-checks can speed up the following checks. This is typically what happens in dynamic algorithms, and so it makes sense trying to apply the techniques developed in that area to speed up the filtering process (for references on dynamic algorithms, see e.g. [13]).

Maybe the simplest and most studied local consistency property is *arc-consistency* [33]. An assignment (x, a) is arc-consistent if, for every

other variable y , there is at least one assignment (y, b) compatible with (x, a) . The assignment (y, b) is a *support* for (x, a) on variable y . Clearly, if an assignment is not arc-consistent, it cannot be consistent (unless x is the unique variable). Arc-consistency can be naturally generalized. An assignment (x, a) is *path-inverse consistent* [17] if it is arc-consistent and, for every two other distinct variables y and z , there are assignments (y, b) and (z, c) which are mutually compatible and compatible with (x, a) . We say that $\{(y, b), (z, c)\}$ is a *support* for (x, a) on $\{y, z\}$. The ℓ -inverse consistency [17] is the natural generalization of arc-consistency ($\ell = 2$) and path-inverse consistency ($\ell = 3$) to arbitrary (fixed) values of $\ell \leq n$.

There is a long series of papers on arc-consistency [3,4,5,33,35]. The currently fastest algorithm has running time $O(ed^2)$, where e denotes the number of distinct pairs of variables involved in some constraint. For long time the fastest known ℓ -inverse-consistency-based filtering algorithm, for $\ell \geq 3$, was the $O(en^{\ell-2}d^\ell)$ algorithm by Debruyne [10].

Remark 3. The quantity $en^{\ell-2}$, corresponding to the number of distinct subsets of ℓ pairwise constrained variables, can be replaced by the tighter $e^{\ell/2}$, given by a combinatorial lemma by Erdős [19].

The algorithm in [10] is based on a rather simple dynamic strategy to check whether an assignment is ℓ -inverse consistent. Roughly, the idea is to sort the candidate supports of any assignment (x, a) on any subset of other $(\ell - 1)$ distinct variables in an arbitrary way, and then follow such order while searching for supports for (x, a) . Moreover, the last supports found are maintained: this way, if a support for (x, a) is deleted and a new one is needed, the already discarded candidates are not reconsidered any more.

The algorithm in [10] was conjectured to be asymptotically the fastest possible. In this section we review an asymptotically faster algorithm for $\ell \geq 3$ [27], based on standard techniques from dynamic algorithms. For the sake of simplicity, let us consider the case of path-inverse-consistency ($\ell = 3$). The same approach extends to larger values of ℓ . Consider any triple of pairwise constrained variables $\{x, y, z\}$, and let $G_{x,y,z}$ be the graph whose nodes are the assignments of x, y and z , and whose edges are the pairs of compatible assignments (i.e. $G_{x,y,z}$ is the restriction of the consistency graph to variables x, y , and z). Any assignment (x, a) is path-inverse-consistent with respect to variables y and z if and only if (x, a) belongs to at least one triangle of $G_{x,y,z}$. More precisely, the number of supports for (x, a) on $\{y, z\}$ is exactly the number of triangles of $G_{x,y,z}$ which contain (x, a) .

Thus a natural approach to enforce path-inverse-consistency is to count all the supports for (x, a) on $\{y, z\}$ initially, and then update the count each time a support is deleted. If we scan all the candidate supports, the initial counting costs $O(d^3)$. Since there can be at most $O(d)$ deletions, and listing all the triangles that contain a given deleted value costs $O(d^2)$, the overall cost of this approach is $O(d^3)$. Since the graphs $G_{x,y,z}$ are $O(e^{1.5})$, this approach has cost $O(e^{1.5}d^3)$, the same as with Debruyne's algorithm.

We next show how to speed up both the initial counting and the later updating by using fast matrix multiplication and lazy updating, two techniques which are widely used in dynamic algorithms [14,15]. By A we denote the adjacency matrix of G . Given an assignment $i = (x, a)$, the number of triangles in which i is contained is $t(i) = (1/2)A^3[i, i]$. Hence, as first observed by Itai and Rodeh [31], the quantities $t(i)$'s can be computed in time $O(d^\omega)$, where $\omega < 2.376$ is fast square matrix multiplication exponent [8].

It remains to show how to maintain the counting under deletion of nodes. We use the following idea. In time $O(d^\omega)$ we can also compute for each edge $\{i, j\}$, the number $t(i, j)$ of triangles which contain $\{i, j\}$. The number of triangles $t(i)$ containing i is one half times the sum of the $t(i, j)$'s over all the edges $\{i, j\}$ incident to i . Now suppose we remove a neighbor j of i . Then, in order to update $t(i)$, it is sufficient to subtract $t(i, j)$. Suppose that we later remove another neighbor k of i . This time subtracting $t(i, k)$ is not correct any more, since we could subtract the triangle $\{i, j, k\}$ twice. However, we can subtract $t(i, k)$ and then add one if $\{i, j, k\}$ is a triangle. This argument can be generalized. Suppose we already deleted a subset of nodes D , and now we wish to delete a neighbor j of i . Then, in order to update $t(i)$, we first subtract $t(i, j)$ and then we add one for each $k \in D$ such that $\{i, j, k\}$ is a triangle. This costs $O(|D|)$ for each update. Altogether maintaining the counting costs $O(d|D|)$ per deletion of node.

When $|D|$ becomes too large, say $|D| \geq d^\epsilon$ for some $\epsilon \in (0, 1)$, this approach is not convenient any more. However in that case we can update all the $t(i, j)$'s, and empty D . In order to update the $t(i, j)$'s, we need to compute all the 2-length paths passing through a node in D . This costs $O(d^{\omega\epsilon+2(1-\epsilon)})$, that is the time to multiply a $d \times d^\epsilon$ matrix by a $d^\epsilon \times d$, where the multiplication is performed by decomposing the rectangular matrices in square pieces, and using square matrix multiplication. Since we perform such update every d^ϵ deletions, the amortized cost per deletion is $O(d^{2+\epsilon(\omega-3)})$. Balancing the terms $O(d^{2+\epsilon(\omega-3)})$ and

$O(d^{1+\epsilon})$, one obtains an overall $O(d^{1+1/(4-\omega)}) = O(d^{1.616})$ amortized cost per deletion. Using more sophisticated rectangular matrix multiplication algorithms [30] the running time can be reduced to $O(d^{2.575})$. This leads to the following result.

Theorem 1. *Path-inverse consistency can be enforced in time $O(e^{1.5}d^{2.575})$.*

Remark 4. Depending on the size and density of the matrices involved, it might be convenient in practice to use matrix-multiplication algorithms different from the fastest asymptotic ones.

Another important consistency property is *max-restricted path consistency*. The same basic approach as above allows one to reduce the time to enforce max-restricted path consistency from $O(e^{1.5}d^3)$ [11] to $O(e^{1.5}d^{2.575})$ [26].

4 Exact algorithms in variables/values ordering

The classical approach to solve (exactly) NP-hard problems is via heuristics. Although heuristics are very useful in practice, they suffer from few drawbacks. First, they do not guarantee worst-case running times (better than the trivial bounds). For example, the worst-case running time to solve a $(d, 2)$ -CSP instance on n variables is (implicitly) assumed to be $\Omega(d^n)$, that is the time bound achieved with exhaustive search. This can be problematic in critical applications, where the worst-case running time matters. Moreover, since the relative performance of heuristics can be assessed only empirically, it is often difficult to compare different heuristic approaches for the same problem.

A potential way to overcome those limits is offered by the design of *exact algorithms*, an area which attracted growing interest in the last decade. The aim of exact algorithms is to solve NP-hard problems in the minimum possible (exponential) worst-case running time. Exact algorithms have several merits:

- The measure of performance is theoretically well-defined: comparing different algorithms is easy.
- The running time is guaranteed on any input, not only on inputs tested experimentally.
- A reduction of the base of the exponential running time, say, from $O(2^n)$ to $O(2^{0.9n})$, increases the size of the instances solvable within a given amount of time by a constant *multiplicative* factor; running a given exponential algorithm on a faster computer can enlarge the mentioned size only by a constant *additive* factor.

- The design and analysis of exact algorithms leads to a deeper insight in NP-hard problems, with a positive long-term impact on the applications.

There are exact algorithms faster than trivial approaches for a number of problems such as: TSP [18,28], maximum independent set [1,22,41], minimum dominating set [20], coloring [2,7], satisfiability [6,9], maximum cut [47], feedback vertex set [39], Steiner tree [16,36], treewidth [23], and many others. For more references, see e.g. [21,32,43,48,49].

To show the potentialities of exact algorithms, we will describe an exact deterministic algorithm which solves any $(d, 2)$ -CSP on n variables in time $O((1 + \lfloor d/3 \rfloor 1.3645)^n)$, thus breaking the $\Omega(d^n)$ barrier given by trivial enumeration.

In order to achieve such running time, we first describe a faster algorithm to solve $(3, 2)$ -CSPs, and we later show how to use it to speed up the solution of arbitrary $(d, 2)$ -CSPs. Note that $(3, 2)$ -CSP is an interesting problem in its own, since it includes as special cases important problems like 3-coloring and 3-SAT via Lemma 2.

We need the following two observations.

Lemma 3. (reduction) [2] *Consider a variable x of a $(d, 2)$ -CSP such that $|D(x)| \leq 2$. Then there is a polynomial-time computable equivalent $(d, 2)$ -CSP with one less variable.*

Proof. If $D(x) = \{a\}$, it is sufficient to remove variable x , and each value conflicting with (x, a) in the domains of the other variables. So, let us assume $D(x) = \{a, b\}$. In such case remove x and add the following set of constraints: for every (y, a') conflicting with (x, a) and for every (z, b') conflicting with (x, b) , $y \neq z$, add the constraint $\{(y, a'), (z, b')\}$. In fact, setting $y = a'$ and $z = b'$ would rule out any possible assignment for x . On the other direction, any solution to the new problem can be extended to a solution for the original problem by assigning either value a or b to x . \square

Lemma 4. (domination) *Consider any $(d, 2)$ -CSP. Let $a, b \in D(x)$, for some variable x , and let A and B be the set of assignments of other variables conflicting with (x, a) and (x, b) , respectively. If $A \subseteq B$, then an equivalent CSP is obtained by removing b from $D(x)$.*

Proof. Suppose there is a solution where $x = b$. Then, by switching x to a , the solution remains feasible. \square

Remark 5. The two properties above cannot be applied if the aim is to compute all the solutions, or the best solution according to some objective function.

We are now ready to describe our improved algorithm for (3, 2)-CSP, which consists of the following steps.

1. **(filtering)** Exhaustively apply arc-consistency and domination to reduce the domains.
2. **(base)** If there a variable with an empty domain, return *no*. Otherwise, if there is at most one variable, return *yes*.
3. **(reduction)** If there is a domain $D(x)$ of cardinality at most 2, remove variable x according to Lemma 3, and branch one the problem obtained.
4. **(branch 1)** If there is a constraint $\{(x, a), (y, b)\}$, where (y, b) is not involved in other constraints, branch by either *selecting* a (i.e., restricting $D(x)$ to $\{a\}$) or *discarding* a (i.e., removing a from $D(x)$).
5. **(branch 2)** Otherwise, take a pair (x, a) involved in constraints with the maximum possible number of distinct variables. Branch by either selecting or discarding a .

By *branching* on a set of subproblems, we mean solve them recursively, and return *yes* if and only if the answer of any one of the subproblems is *yes*.

Lemma 5. *The algorithm above solves any (3, 2)-CSP instance in worst-case time $O(1.466^n)$.*

Proof. We define an instance *ground* if the algorithm solves it without branching on two subproblems (hence in polynomial-time). By $P(\mathcal{X}, \mathcal{C})$ we denote the total number of ground instances solved to solve a given (3, 2)-CSP instance $(\mathcal{X}, \mathcal{C})$. Let $P(n)$ be the maximum of $P(\mathcal{X}, \mathcal{C})$ over all the instances on n variables. We will show that $P(n) \leq 1.4656^n$. The claim follows by observing that generating each subproblem costs only polynomial time, excluding the cost of the recursive calls, and the total number of subproblems generated is within a polynomial from $P(n)$. Hence the total running time is $O(1.4656^n n^{O(1)}) = O(1.466^n)$.

We proceed by induction on n . Trivially, $P(0) = P(1) = 1$ satisfy the claim. Now assume the claim is true up to $(n - 1) \geq 1$ variables, and consider any instance $(\mathcal{X}, \mathcal{C})$ on $n \geq 2$ variables. We distinguish different cases, depending on the step where the algorithm branches:

(base) We do not generate any subproblem

$$P(\mathcal{X}, \mathcal{C}) \leq 1 \leq 1.4656^n.$$

(reduction) We generate exactly one subproblem with one less variable:

$$P(\mathcal{X}, \mathcal{C}) \leq P(n-1) \leq 1.4656^{n-1} \leq 1.4656^n.$$

(branch 1) In both subproblems variable x is removed by Step 3. When we select a , we remove b from $D(y)$ by arc-consistency, and hence variable y by Step 3. When we discard a , we remove a value $c \in D(x) \setminus \{b\}$ by dominance, being c dominated by b . So also in that case y is later removed by Step 3. Altogether

$$P(\mathcal{X}, \mathcal{C}) \leq P(n-2) + P(n-2) \leq 2 \cdot 1.4656^{n-2} \leq 1.4656^n.$$

(branch 2) By basically the same arguments as above, if (x, a) is involved in constraints with at least two other variables y and z , when we branch by discarding a , we remove at least variable x , while when we branch by selecting a , we remove at least variables x , y and z . Hence

$$P(\mathcal{X}, \mathcal{C}) \leq 1.4656^{n-1} + 1.4656^{n-3} \leq 1.4656^n.$$

Otherwise, by Steps 1 and 4, and by a simple case analysis, there must be a set of 6 constraints involving x and y of the following form: $\{(x, a), (y, b')\}$, $\{(x, a), (y, c')\}$, $\{(x, b), (y, b')\}$, $\{(x, c), (y, c')\}$, $\{(x, b), (y, a')\}$, and $\{(x, c), (y, a')\}$. When we select a , we remove variable x , values b' and c' from $D(y)$ by arc-consistency, and later variable y by Step 3. When we discard a , we remove variable x , value a' by dominance, and later variable y by Step 3. Thus

$$P(\mathcal{X}, \mathcal{C}) \leq 2 \cdot 1.4656^{n-2} \leq 1.4656^n.$$

The claim follows. \square

By using similar, but more sophisticated, arguments, Beigel and Eppstein showed that any $(3, 2)$ -CSP on n variables can be solved in worst-case time $O(1.36443^n)$ [2].

Consider now the following algorithm to solve any $(d, 2)$ -CSP, which can be interpreted as a derandomization of a result in [2]. For each variable x , partition $D(x)$ in $\lceil d/3 \rceil$ subsets of cardinality at most 3, and branch by restricting the domain of x to each one of the mentioned subsets. When all the domains are restricted in that way, solve the instance obtained with the mentioned $O(1.36443^n)$ algorithm for $(3, 2)$ -CSP. Return *yes* if and only if one of the subproblems generated is feasible.

Theorem 2. *Any $(d, 2)$ -CSP, $d \geq 3$, can be solved in $O(\alpha^n)$ worst-case time, where $\alpha = \alpha(d) = \min\{\lceil d/3 \rceil 1.3645, 1 + \lfloor d/3 \rfloor 1.3645\}$.*

Proof. For the sake of simplicity, assume all the domains have size d . This can be achieved by adding dummy variables. Consider the algorithm above. Its running time is trivially

$$O(\lceil d/3 \rceil^n 1.36443^n n^{O(1)}) = O(\lceil d/3 \rceil^n 1.3645^n).$$

When d is not a multiple of 3 a better time bound is achieved by observing that the partition of each $D(x)$ contains $\lfloor d/3 \rfloor$ sub-domains of size 3, and one sub-domain of size $d \pmod{3} \in \{1, 2\}$. Hence the algorithm generates $\binom{n}{i}$ problems containing i domains of cardinality at most 2, and $n - i$ domains of cardinality 3. The variables corresponding to the i small domains can be removed without branching. Hence the running time is

$$O\left(\sum_i \binom{n}{i} \lfloor d/3 \rfloor^{n-i} 1.36443^{n-i} n^{O(1)}\right) = O((1 + \lfloor d/3 \rfloor 1.3645)^n).$$

□

Remark 6. A different algorithm is obtained by partitioning the domains in sub-domains of size 2 instead of 3, and then branching as in the algorithm above. Since each subproblem created can be solved in polynomial time, the overall running time is $O(\lceil d/2 \rceil^n n^{O(1)})$. This improves on the previous result for $d = 4$ and $d = 10$.

References

1. R. Beigel. Finding maximum independent sets in sparse and general graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 856–857, 1999.
2. R. Beigel and D. Eppstein. 3-coloring in time $O(1.3289^n)$. *Journal of Algorithms*, 54:168–204, 2005.
3. C. Bessière. Arc-consistency and arc-consistency again. In *National Conference on Artificial Intelligence (AAAI)*, pages 179–190, 1994.
4. C. Bessière and M. O. Cordier. Arc-consistency and arc-consistency again. In *National Conference on Artificial Intelligence (AAAI)*, pages 108–113, 1993.
5. C. Bessière, E. Freuder, and J. C. Regin. Using inference to reduce arc consistency computation. In *International Joint Conference on Artificial Intelligence*, pages 592–598, 1995.
6. T. Brueggemann and W. Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science*, 329:303–313, 2004.
7. J. M. Byskov. Enumerating maximal independent sets with applications to graph colouring. *Operations Research Letters*, 32:547–556, 2004.
8. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

9. E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schning. A deterministic $(2 - 2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.
10. R. Debruyne. A property of path inverse consistency leading to an optimal PIC algorithm. In *European Conference on Artificial Intelligence*, pages 88–92, 2000.
11. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Principles and Practice of Constraint Programming (CP)*, pages 312–326, 1997.
12. R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.
13. C. Demetrescu, I. Finocchi, G. F. Italiano, “Dynamic Graphs”, Chapter 22, *Handbook of Data Structures and Applications*, CRC Press, 2004.
14. C. Demetrescu, G. F. Italiano, “Trade-Offs for Fully Dynamic Transitive Closure on DAGs: Breaking Through the $O(n^2)$ Barrier”, *Journal of the ACM*, vol. 52, no. 2, March 2005, 147–156.
15. C. Demetrescu, G. F. Italiano, “Dynamic Shortest Paths and Transitive Closure: Algorithmic Techniques and Data Structures”, *Journal of Discrete Algorithms*, vol 4 (3), September 2006.
16. S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1971/72.
17. C. D. Elfe and E. C. Freuder. Neighborhood inverse consistency preprocessing. In *National Conference on Artificial Intelligence (AAAI)/Innovative Applications of Artificial Intelligence*, volume 1, pages 202–208, 1996.
18. D. Eppstein. The traveling salesman problem for cubic graphs. In *Workshop on Algorithms and Data Structures (WADS)*, pages 307–318, 2003.
19. P. Erdős. On the number of complete subgraphs contained in certain graphs. *Magyar Tudományos Akadémia Matematikai Kutató Intézetének Közleményei*, 7:459–464, 1962.
20. F. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: domination - a case study. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 191–203, 2005.
21. F. Fomin, F. Grandoni, and D. Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the European Association for Theoretical Computer Science*, 87:47–77, 2005.
22. F. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: A simple $o(2^{0.288n})$ independent set algorithm. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 18–25, 2006.
23. F. V. Fomin, D. Kratsch, and I. Todinca. Exact algorithms for treewidth and minimum fill-in. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 568–580, 2004.
24. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *ACM Symposium on the Theory of Computing (STOC)*, pages 246–251, 1983.
25. F. Glover. Maximum matching in convex bipartite graph. *Naval Research Logistic Quarterly*, 14:313–316, 1967.
26. F. Grandoni and G. F. Italiano. Improved algorithms for max-restricted path consistency. In *Principles and Practice of Constraint Programming (CP)*, pages 858–862, 2003.
27. F. Grandoni and G. F. Italiano. Decremental clique problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 142–153, 2004.

28. M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of SIAM*, 10:196–210, 1962.
29. J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.
30. X. Huang and V. Pan. Fast rectangular matrix multiplication and applications. *Journal of Complexity*, 14(2):257–299, 1998.
31. A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.
32. K. Iwama. Worst-case upper bounds for k-SAT. *Bulletin of the European Association for Theoretical Computer Science*, 82:61–71, 2004.
33. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
34. K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In R. Dechter, editor, *Principles and Practice of Constraint Programming (CP)*, pages 306–319, 2000.
35. R. Mohr and T. C. Henderson. Arc and path consistency revised. *Artificial Intelligence*, 28:225–233, 1986.
36. D. Mölle, S. Richter, and P. Rossmanith. A faster algorithm for the Steiner tree problem. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 561–570, 2006.
37. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.
38. J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *National Conference on Artificial Intelligence (AAAI)/Innovative Applications of Artificial Intelligence*, pages 359–366, 1998.
39. I. Razgon. Exact computation of maximum induced forest. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, 2006. To appear.
40. J. C. Regin. A filtering algorithm for constraints of difference in CSP. In *National Conference on Artificial Intelligence (AAAI)*, volume 1, pages 362–367, 1994.
41. J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440, 1986.
42. V. A. Saraswat. *Concurrent Logic Programming Languages*. PhD thesis, Carnegie-Mellon University, 1987.
43. U. Schöning. Algorithmics in exponential time. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 36–43, 2005.
44. K. Stergiou. *Representation and Reasoning with Non-Binary Constraints*. PhD thesis, University of Strathclyde, Glasgow, Scotland, 2001.
45. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.
46. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
47. R. Williams. A new algorithm for optimal constraint satisfaction and its implications. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1227–1237, 2004.
48. G. J. Woeginger. Exact algorithms for np-hard problems: A survey. In M. Jünger, G. Reinelt, and G. Rinaldi, editors, *International Workshop on Combinatorial Optimization – Eureka, You Shrink*, number 2570 in Lecture Notes in Computer Science, pages 185–207. Springer-Verlag, 2003.
49. G. J. Woeginger. Space and time complexity of exact algorithms: Some open problems. In *International Workshop on Parameterized and Exact Computation (IWPEC)*, pages 281–290, 2004.