Reduced branching-factor algorithms for constraint satisfaction problems*

Igor Razgon and Amnon Meisels
Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel
{irazgon,am}@cs.bgu.ac.il

November 21, 2005

Abstract

This paper investigates connections between the worst-case complexity of CSP search algorithms and their practical efficiency. We prove that a well-known search algorithm FC-CBJ together with the Fail-First ordering heuristic has a worst-case complexity of $O^*((d-1)^n)$ rather than $O^*(d^n)$, where d and n are the maximal domain size and the number of variables of the given CSP, respectively. This result shows that heuristic methods designed only for practical purposes can be useful for reducing worst-case complexity.

To show that low-complexity CSP algorithms can be useful in practice, we take an existing "purely-theoretical" approach that solves CSP in $O^*(\lceil d/2 \rceil^n)$ and, based on this approach, design two algorithms 2FC and 2MAC that are an order of magnitude better than their prototypes, FC and MAC, respectively, on a number of benchmark domains. ¿From this result we infer that purely theoretical results can be very useful from a practical point of view.

1 Introduction

The constraint satisfaction problem (CSP) is a widely-recognized model which provides a convenient representation for various real-world problems. Nonetheless, there is another, not less important application of CSP in that

*The authors would like to acknowledge the Lynn and William Frankel Center for Computer Sciences and the Paul Ivanier Center for Robotics for financial support

it serves as the main framework for developing of sophisticated search algorithms.

Search algorithms are used for solving intractable problems. Examples of basic search algorithms include: chronological backtracking for CSP [5], the DPLL procedure for SAT [3], and forward chaining algorithm for planning [1]. However, all these algorithms suffer from one major drawback. Whenever they are applied in their "pure" form, they are unable to process in a reasonable time instances of "real-world" size. To be more efficient, these algorithms must be accompanied by appropriate speeding-up methods including techniques of pruning the search space, ordering heuristics, preprocessing techniques, learning techniques. Traditionally, most of these techniques were developed first for solving of CSPs. This connection can be observed especially distinctly for SAT. For example, Forward Checking [12] is a prototype of unit propagation; dynamic variable ordering heuristics for CSPs [12, 11] are prototypes of branching heuristics for SAT; Conflict-Directed Backjumping [18] is a prototype of clause learning [16]. Similar relations can be observed for methods of graph coloring [2].

During more than 20 years of CSP research, various aspects related to improving behavior of search algorithms were investigated. These aspects include:

- preprocessing algorithms that achieve a certain level of local consistency [17, 4];
- ordering heuristics that determine the search tree being explored [11, 23, 10];
- methods of maintaining consistency during search [24, 26];
- methods of intelligent backtracking that allow early identification of dead-ends [6];
- domain dependent methods [22, 20], symmetry breaking methods [8, 9];
- algorithms with reduced worst-case complexity [7].

Nevertheless, there are aspects that have received little (if any) attention. One of these aspects is the correlation between the worst-case complexity of backtrack algorithms and their practical efficiency. At first glance, it is not clear whether such a correlation exists because the runtime of backtrack algorithms for finding the first solution is usually much smaller than

their worst-case complexity. Moreover, an algorithm with "poor" complexity measure but with "wise" pruning heuristics might be much better from a practical point of view than an algorithm that has excellent worst case-complexity but searches in a brute-force manner. The present paper contributes to the investigation of the above correlation. The contributions can be divided into the following two parts.

Heuristic techniques can reduce worst-case complexity. The worst-case complexity of most "practical" backtrack algorithms is considered to be $O^*(d^n)$, where d and n are the maximal domain size and the number of variables of a given CSP, respectively. We refute this widely accepted conjecture by proving that the algorithm resulting from the combination of Forward Checking (FC) with Conflict-Directed Backjumping (CBJ) [18] and the Fail-First ordering heuristic (FF) [12] has the worst-case complexity of $O^*((d-1)^n)$. Furthermore, we show that all the components above are necessary for reducing complexity because combinations of FC only with CBJ (FC-CBJ [18]) or only with FF, result in an algorithm that has the worst-case complexity of $\Omega^*(d^n)$.

These results provide evidence that heuristic methods designed for practical purposes only can reduce worst-case complexity. More generally, the results indicate that a correlation exists between the practical usefulness of backtrack algorithms and their complexity measures. Actually, it is widely accepted that FC together with CBJ and FF works better than a combination FC with only FF or FC with only CBJ. The proposed complexity analysis shows exactly the same relation between these algorithms.

Backtrack algorithms with reduced complexity can be efficient in practice. Having demonstrated that heuristic techniques can be interesting from a theoretical point of view, we investigate the opposite direction. The point is that CSPs can be solved by a simple algorithm that has a worst-case complexity of $O(\lceil d/2 \rceil^n)$. This complexity can be achieved by defining a smaller search space, but it is commonly explored by an "impractical" brute-force procedure with an actual runtime close to its theoretical upper bound. We investigate the possibility of designing a "practical" algorithm that benefits from a reduced search space.

For this purpose, we design two algorithms which we term 2FC and 2MAC . Both of these algorithms have a worst-case complexity of $O(\lceil d/2 \rceil^n)$. 2FC explores the search space in a manner similar to Forward Checking (FC), and 2MAC works similarly to the Maintaining Arc-Consistency algo-

rithm (MAC). According to our experimental evaluation, there are a number of classes of CSPs where 2FC and 2MAC are an order of magnitude better than FC and MAC, respectively.

These algorithms indicate that techniques originally developed for "theoretical" purposes can be useful for designing of strong "practical" search algorithms.

The rest of the paper is organized as follows. Section 2 provides the necessary background. The complexity analysis of FC-CBJ with FF and the accompanying results are presented in Section 3. The design of algorithms 2FC and 2MAC is described in Section 4. Empirical evaluation of 2MAC is given in Section 5. Finally, Section 6 concludes the paper.

2 Preliminaries

2.1 Notations and terminology

The present paper considers only binary CSPs. A CSP Z consists of three components. The first component is a set of variables. Every variable has a domain of values. We denote a value val of a variable v by $\langle v, val \rangle$. The set of domains of variables comprises the second component of Z. The *constraint* between variables u and v is a subset of the Cartesian product of the domains of u and v. A pair of values $(\langle u, val_1 \rangle, \langle v, val_2 \rangle)$ is compatible if it belongs to the constraint between u and v. Otherwise the values are incompatible (*conflicting*). The set of all constraints comprises the third part of Z [5, 25]

A set P of values of different variables is *consistent* (*satisfies* all the constraints) if all the values of P are mutually compatible. In this case, we call P a *partial solution* of Z. If $\langle u, val \rangle \in P$, we say that P *assigns* u. Accordingly, $\langle u, val \rangle$ is the *assignment* of u in P. If P assigns all the variables, it is a *solution* of P. The task of a CSP is to find a solution of Z or to report that no solution exists.

Generally, not every partial solution is a subset of a full solution. If a partial solution P is not a subset of any solution, it is called a *nogood*. Note that sometimes in the literature, the notion of nogood has a broader meaning in that it includes also a set of assignments with inner conflicts. In the present work a nogood is a specific case of a partial solution, that is, a consistent set of assignments.

In order to present the results of the proposed research, we extend the notion of compatibility and consistency.

A value $\langle u, val \rangle$ is *compatible* with a partial solution P if the following two conditions hold:

- if u is assigned by P then $\langle u, val \rangle \in P$;
- $\langle u, val \rangle$ is compatible with all the assignments of P.

A value $\langle u, val \rangle$ is *consistent* with a subset S of the domain of a variable v if $\langle u, val \rangle$ is compatible with at least one value of S. If S contains all values of the domain of v then $\langle u, val \rangle$ is *consistent with variable* v.

Similarly, a partial solution P is *consistent* with a subset S of the domain of a variable v if it is compatible with at least one value of S. If S contains all values of the domain of v then P is *consistent with variable* v. Accordingly, a value $\langle u, val \rangle$ (or a partial solution P) is inconsistent with a subset S of the domain of v if it is incompatible with all values of S.

Finally, we present the notion of arc-consistency.

Definition 1 *A value $\langle v, val \rangle$ is arc-consistent if it is consistent with respect to every variable $u \neq v$.*

A CSP is arc-consistent if all its values are arc-consistent.

If a value $\langle v, val \rangle$ is not consistent with respect to some variable $u \neq v$, this value clearly cannot belong to any solution of the underlying CSP. Such a value can be removed from the domain of v. *Achieving arc-consistency* is a procedure that iteratively removes values that are not arc-consistent. At the termination of the procedure, one of two possible situations hold: either an arc-consistent CSP is obtained, or the domain of some variable is empty. In the latter case, it follows that the underlying CSP is insoluble.

An efficient implementation of a procedure for achieving AC is not trivial because the removal of a value $\langle v, val \rangle$ can cause another value, say, $\langle w, val' \rangle$ to be inconsistent with respect to v (if the only value of v compatible with $\langle w, val' \rangle$ is $\langle v, val \rangle$) [17].

2.2 The FC, FC-CBJ, and MAC algorithms

The Forward Checking algorithm (FC) [18] is a CSP search algorithm based on enumeration of partial solutions. It starts from the empty partial solution. In every iteration, FC selects an unassigned variable, assigns it with a value and appends the assignment to the current partial solution. The characteristic feature of FC is that whenever a new assignment is added to the current partial solutions, the values of unassigned variables that are incompatible with the new assignment are temporarily removed from the

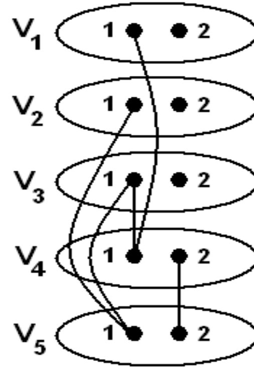

Figure 1: CSP used for illustration of work of search algorithms

domains of variables. Therefore, when we consider some state that occurs during the execution of FC, we frequently refer to the *current domain* of some variable v, having in mind the subset of values that were not removed from the domain of v.

If FC assigns all the variables of the underlying CSP, it returns a solution. However, during the iterative enlargement of the current partial solution, the current domain of some unassigned variable might be emptied. In this case, FC backtracks, that is, discards the last assignment of the current partial solution and replaces it by another assignment of the same variable. Note that when an assignment is discarded, all the values, removed because of incompatibility with the assignment, are restored to their current domains. It may also happen that FC cannot replace the discarded assignment by another one. In this case it backtracks again. Finally, it might happen that FC tries to backtrack, but the current partial solution is empty. In this case, FC reports insolubility [18].

Assume that FC solves the CSP in Figure 1 and consider a possible scenario of the execution. FC starts from the empty current partial solution. Then $\langle v_1, 1\rangle$ is appended to the current partial solution and $\langle v_4, 1\rangle$ is removed because of incompatibility with $\langle v_1, 1\rangle$. The next assignment appended to the current partial solution is $\langle v_2, 1\rangle$; the assignment causes the removal of $\langle v_5, 1\rangle$. The next appended assignment is $\langle v_3, 1\rangle$. Then FC adds to the current partial solution assignment $\langle v_4, 1\rangle$; as a result, $\langle v_5, 2\rangle$ is removed, the domain of v_5 is emptied and FC backtracks.

Performing backtrack, FC discards $\langle v_4, 2\rangle$ and removes it from the current domain of v_4. As a result, $\langle v_5, 2\rangle$ is restored to the current domain of v_5. The backtracking empties the domain of v_4, hence FC backtracks again,

discarding $\langle v_3, 1\rangle$ and restoring $\langle v_4, 2\rangle$. Note that $\langle v_4, 1\rangle$ is not restored because it was removed by incompatibility with $\langle v_1, 1\rangle$, which still belongs to the current partial solution Then $\langle v_3, 2\rangle$ is appended to the current partial solution. After that FC appends again $\langle v_4, 2\rangle$ which causes three consecutive backtracks that discard assignments $\langle v_4, 2\rangle$, $\langle v_3, 2\rangle$, and $\langle v_2, 1\rangle$. Then FC appends $\langle v_2, 2\rangle$ to the current partial solution. The assignment $\langle v_3, 1\rangle$ appended next to the current partial solution is discarded after a number of steps. After appending assignments $\langle v_3, 2\rangle$, $\langle v_4, 1\rangle$ and $\langle v_5, 1\rangle$, FC obtains a full solution, which is returned.

States of a search algorithm The execution of a CSP search algorithm can be represented as a sequence of atomic operations of updating of the current partial solution (addition or removal of assignments) accompanied by appropriate updating of the maintained data structures in order to preserve consistency. The information recorded in the data structures before the execution or after performing an atomic operation constitutes a *state* of a search algorithm. Thus a sequence of states is another possible representation of a search algorithm. We frequently use this representation in the present paper, in order to prove properties of the analyzed algorithms.

Forward Checking with Conflict-directed Backjumping (FC-CBJ) is a modification of FC that can backtrack more than 1 step backwards (backjump). The completeness of the enumeration is preserved by maintaining *conflict sets* of variables. In a given state of FC-CBJ, the conflict set of a variable v, denoted by $conf(v)$, contains all variables whose assignments in the current partial solution are "culprit" for removing of values from the current domain of v.

The detailed description of FC-CBJ is quite technical and long, hence we list only those features of the algorithm that are relevant to the theorems we are going to prove.

- Initially all conflict sets are empty.
- Whenever a value $\langle u, val\rangle$ is appended to the current partial solution and a value of an unassigned variable v is removed as a result of incompatibility with $\langle u, val\rangle$, u is added to $conf(v)$.
- Whenever the empty domain of a variable v causes a backtrack, FC-CBJ backjumps to the last assigned variable u that appears in $conf(v)$ and discards the assignment of this variable. The assignments of variables that were appended to the current partial solution after the assignment of u are canceled, as if they were not performed at all (of

course, with the restoring of values removed by these assignments). Note that removing an assignment of a variable from the current partial solution, FC-CBJ removes appearances of this variable from all conflict sets.

- Whenever the empty current domain of a variable v causes backtrack and the backtrack process discards the assignment of a variable u, $conf(u)$ is set to $conf(u) \cup conf(v) \setminus \{u\}$.

Assume that the CSP illustrated in Figure 1 is processed by FC-CBJ. In the beginning, the execution is similar to that of FC with the only difference that whenever new assignments are added to the current partial solution the conflict sets of the corresponding variables are updated. In particular adding assignment $\langle v_1, 1\rangle$ causes adding v_1 to $conf(v_4)$, v_2 is added to $conf(v_5)$ as a result of adding of $\langle v_2, 1\rangle$. Note that the assignment $\langle v_3, 1\rangle$ does not cause updating of conflict sets. Finally, the assignment $\langle v_4, 5\rangle$ causes v_4 to be added to $conf(v_5)$. The first backtrack of FC-CBJ is caused by the empty domain of v_5. At the time of backtrack, $conf(v_5) = \{v_2, v_4\}$, hence FC-CBJ jumps to v_4. Note that before the backtrack, $conf(v_4) = \{v_1\}$. After backtrack it is updated to $\{v_1, v_2\}$ as a result of the union operation with $conf(v_5)$ and the removal of v_4. Also, v_4 is removed from $conf(v_5)$.

The second backtrack occurs because the domain of v_4 becomes empty. Because the last variable in $conf(v_4)$ is v_2, FC-CBJ jumps over v_3 and discards the assignment of v_2. Thus FC-CBJ avoids an unnecessary assignment $\langle v_3, 2\rangle$ performed by FC after the second backtrack.

The last search algorithm we consider in this section is Maintaining Arc-Consistency (MAC) [24]. At the preprocessing stage this algorithm achieves arc-consistency for the underlying CSP. The search procedure is very similar to that of FC with the only difference that after every assignment $\langle u, val\rangle$, in addition to removal of values incompatible with $\langle u, val\rangle$, it removes every value $\langle v, val_1\rangle$ of an unassigned variable v, which is inconsistent with the current domain of another unassigned variable w. In other words, after every assignment, MAC achieves arc-consistency for the CSP created by the current domains of unassigned variables.

Let us demonstrate the execution of MAC on the CSP shown in Figure 1. The CSP itself is arc-consistent, so, no value is removed at the preprocessing stage. After adding assignment $\langle v_1, 1\rangle$ to the current partial solution and removing of $\langle v_4, 1\rangle$, $\langle v_5, 2\rangle$ becomes non-arc-consistent because it conflicts with $\langle v_4, 2\rangle$, the only values in the current domain of v_4. After removing of $\langle v_5, 2\rangle$, $\langle v_2, 1\rangle$ and $\langle v-3, 1\rangle$ become non-arc-consistent. The arc-consistency

achieving process terminates with the removal of the latter two values. The MAC assigns v_2 with 2, v_3 with 2, v_4 with 2, v_5 with 1, thus getting a full solution without backtracking. This example demonstrates that marinating arc-consistency during search could be very useful for saving computational effort.

2.3 The Fail-First ordering heuristics

CSP search algorithms do not specify explicitly the order of selection of variables to be assigned. This job is performed by ordering heuristics. One of the simplest and most successful ordering heuristics is called Fail-First (FF) [12]. Every time a new variable must be assigned, FF selects a variable with the smallest size of the current domain. The time complexity of FF is linear in the number of unassigned variables. The implementation of FF requires maintaining an array of domain sizes of variables which is updated dynamically when values are removed or restored. FF selects the minimal element among the entries of the array corresponding to the domains of unassigned variables.

Consider the execution of FC with FF on the CSP of Figure 1, assuming that in case of two or more variables with the smallest domain size, one is selected according to lexicographic order.

Initially, the current domains of all the variables are of equal size, so v_1 is assigned with 1. After removal of $\langle v_4, 1\rangle$ as a result of the assignment, v_4 becomes the variable with the smallest domain size, so $\langle v_4, 2\rangle$, the only remaining value is added to the current partial solution. The value $\langle v_5, 2\rangle$ is removed because of incompatibility with $\langle v_4, 2\rangle$, hence v_5 becomes the variable with the smallest domain size and the assignment $\langle v_5, 1\rangle$ is added to the current partial solution. The values $\langle v_2, 1\rangle$ and $\langle v_3, 1\rangle$ are incompatible with $\langle v_5, 1\rangle$, hence they are removed from the current domains of v_2 and v_3. The next two iterations add to the current partial solution $\langle v_2, 2\rangle$ and $\langle v_3, 2\rangle$. The obtained full solution is returned after that.

The above example demonstrates the strength of FF, because it allows to avoid backtracks during the processing of the given example CSP.

2.4 Complexity of backtrack algorithms

All complete CSP search algorithms (those that return a solution if one exists or report insolubility otherwise) have exponential time-complexity. Discussing aspects related to the complexity of backtracking algorithms, we follow two agreements:

- We express the time complexity by the O^* notation [27], which suppresses the polynomial factor. For example, instead of $O(n^2 * 2^n)$, we write $O^*(2^n)$. Note, that for a constant $d > 1$ $O^*((d-1)^n)$ is smaller than $O^*(d^n)$ because $O^*(d^n) = O^*((d/d-1)^n * (d-1)^n)$, where $(d/(d-1))^n$ is a growing exponential function that cannot be suppressed by the O^* notation. On the other hand, given constants d and k, $O^*(d^{n+k})$ is the same as $O^*(d^n)$ because $O^*(d^{n+k}) = O^*(d^k * d^n)$, where d^k can be suppressed as a constant.
- We express the time complexity of a CSP search algorithm by the number of partial solutions generated by the algorithm. This is a valid representation because the time complexity can be represented as the number of partial solutions multiplied by a polynomial factor which is ignored by the O^* notation.

The worst-case complexity of FC, FC-CBJ and MAC when applied to a CSP with n variables and maximum domain size d is $O^*(d^n)$. However, a CSP can be easily solved in $O^*(\lceil d/2 \rceil^n)$.

Let us refer to a CSP with at most two values in every domain as *2-CSP*. It is possible to solve 2-CSP efficiently by transforming it to 2-SAT [14]. This fact points to a method for solving a CSP with a worst-case complexity of $O^*(\lceil d/2 \rceil^n)$.

Assume for simplicity that d is even and that the domain of every variable has exactly d values and partition arbitrarily the domain of every variable into $d/2$ pairs. Then we can solve all the generated 2-CSPs obtained by taking exactly one partition class from every variable. If at least one of them is soluble, return a solution. Otherwise, report insolubility.

It is not difficult to show that the above method of solving is correct and that at most $(d/2)^n$ 2-CSPs are generated, which proves the required upper bound. The complete proof is a bit more technical and considers also the case when d is odd. In this case, one of the partition classes is a singleton. That is why we use $\lceil d/2 \rceil$ instead of $d/2$ in the complexity formula.

The complexity of $O^*(\lceil d/2 \rceil^n)$ can be further improved if we partition the domain into triplets instead of pairs. A sophisticated approach proposed by Eppstein [7] solves 3-CSP (a CSP with at most 3 values in every domain) approximately in $O^*(1.35^n)$. Taking into account that the partition creates at most $O(d/3)^n$ 3-CSPs, we get a complexity for the complete CSP of about $O((0.45 * d)^n)$.

It seems strange that these algorithms have not attracted more attention from researchers who investigate the practical aspects of constraint

satisfaction. One possible may be that these algorithms achieve better upper bounds by defining smaller search spaces but explore these search spaces in a brute-force manner. This make the actual time of their work close to their theoretical upper bound, while the actual runtime (for finding the first solution) of algorithms like FC and MAC is much smaller.

3 Reducing time complexity by heuristic methods

3.1 FC-CBJ combined with FF has complexity of $O^*((d-1)^n)$

In this section we will show that the use of heuristic techniques can decrease the complexity of a search algorithm. In particular we prove that the FC-CBJ algorithm [18] combined with the FF heuristic [12] has a worst-case complexity of $O^*((d-1)^n)$, where n and d are the number of variables and the maximal domain size, respectively. For the proof, we use the notion of a maximal partial solution.

Definition 2 *Let P be a partial solution explored by a search algorithm during the solving of a CSP Z. Then P is maximal if it is not a subset of any other partial solution visited by the algorithm during solving Z.*

We now prove a theorem that states an upper bound on the number of maximal solutions explored by FC-CBJ with FF. The overall complexity of FC-CBJ with FF will follow from this result.

Theorem 1 *FC-CBJ with FF applied to a CSP Z with $n \geq 2$ variables and maximal domain size d explores at most $M(n) = d * \Sigma_{i=0}^{n-2}(d-1)^i$ maximal partial solutions.*

In order to prove the theorem, we need to prove an additional lemma.

Lemma 1 *Let Z be a CSP with maximum domain size d. Consider a state S of FC-CBJ that occurs during processing of Z. Let P be the current partial solution maintained by FC-CBJ in this state. Assume that in P is not empty and that the current domain size of every unassigned variable is d. Let Z' be a CSP created by the current domains of unassigned variables. Assume that Z' is insoluble. Then, after visiting state S, the execution of FC-CBJ is continued as follows: FC-CBJ detects the insolubility of Z', discards all the values of P, reports insolubility of Z', and stops.*

Proof. Considering that d is the maximum possible domain size, we infer that the current domains of the unassigned variables are the same as their initial domains. It follows that all values of the original domains of the unassigned variables are compatible with P. Consequently, the conflict sets of all the unassigned variables are empty.

Observe that when processing Z', a variable assigned by P does not appear in any conflict set of a variable of Z'. This observation can be verified by induction on the sequence of events updating the conflict sets of Z'. Note that the observation holds *before* FC-CBJ starts to process Z', because all the conflict sets are empty (by the argumentation of the previous paragraph). Assuming that the observation holds for the first k events, let us consider the $k+1$-th one. Assume that v is the variable whose conflict set is updated. If this updating results in insertion of the currently assigned variable then the variable being inserted belongs to Z' which is not assigned by P. Otherwise, $conf(v)$ is united with the conflict set of another variable u of Z'. However, $conf(u)$ does not contain variables assigned by P by the induction assumption.

If Z' is insoluble, FC-CBJ will eventually discard P. This means that FC-CBJ will arrive at a state in which the current domain of a variable v of Z' is empty and $conf(v)$ does not contain any variable of Z'. On the other hand, $conf(v)$ will not contain any variable assigned by P. That is, the conflict set of v will be empty. Consequently, FC-CBJ will jump "over" all the assigned variables, report insolubility of Z, and stop. ■

Now we are ready to prove Theorem 1.

Proof of Theorem 1. We prove the theorem by induction on n. For the basic case assume that $n = 2$. Let v_1 and v_2 be the variables of Z and assume that v_1 is assigned first. Consider the situation that occurs when v_1 is assigned with a value $\langle v_1, val \rangle$. If the value is compatible with at least one value in the domain of v_2, FC-CBJ returns a solution. Otherwise, it instantiates $\langle v_1, val \rangle$ with another value of v_1 or exits if all values of v_1 have been explored. Thus, every value of v_1 participates in at most one partial solution. Keeping in mind that there are at most d such values, we get that at most d partial solutions are explored. Observe that $M(2) = d$. That is, the theorem holds for $n = 2$.

Assume that $n > 2$ and that the theorem holds for all CSPs having less than n variables. We consider two possible scenarios of execution of FC-CBJ.

According to the first scenario whenever the current partial solution is not empty (at least one variable has been already instantiated), FC-CBJ selects for instantiation a variable with the current domain size smaller than

d. Then FC-CBJ explores a search tree in which at most d edges leave the root node and at most $d-1$ edges leave any other node.

Note that when FC-CBJ has assigned all the variables but one, it does not execute branching on the last variable. If the domain of the last variable is not empty, FC-CBJ takes any available value and returns a full solution. Otherwise, it backtracks. It follows that in the search tree explored by FC-CBJ only the first $n-1$ levels can contain nodes with two or more leaving edges. The branching factor on the first level is d, but the branching factor of a node at any other of $n-2$ remaining levels is $d-1$. Consequently, the number of leaves of the search tree is at most $d*(d-1)^{n-2}$. Taking into account that the leaves of the search tree correspond to the maximal partial solutions, we see that in the considered case FC-CBJ explores at most $d*(d-1)^{n-2} \leq M(n)$ maximal partial solutions. Thus, the theorem holds in the case of the first scenario.

If the first scenario does not occur then FC-CBJ, having at least one variable instantiated, selects for assignment a variable with the current domain size d. Consider the first time when such a selection occurs and denote by P the current partial solution maintained by FC-CBJ in the considered state. Denote by Z' the CSP created by the current domains of variables unassigned by P. Proceeding with the execution, FC-CBJ solves Z'. If Z' is soluble then FC-CBJ finds a solution of Z', returns its union with P, and stops.

The case when Z' is insoluble is **the main point in the proof of the theorem**. Note that FC-CBJ uses FF. If a variable with the current domain size d is selected, the current domain sizes of the other unassigned variables are at least d. On the other hand, d is the maximal possible domain size, hence the current domain sizes of the other variables are exactly d. By Lemma 1, FC-CBJ stops after detecting insolubility of Z'. Note again that both FC-CBJ and FF contributed to the validity of this claim. The contribution of FF is ensuring that the current domains of all the unassigned variables are exactly d. The contribution of FC-CBJ is explained in the proof of Lemma 1.

Thus we have shown that whenever FC-CBJ selects a variable with the current domain size d given that the current partial solution is non-empty, the algorithm always stops after detecting the insolubility of Z'.

The number of maximal partial solutions visited by FC-CBJ in this case equals the number of maximal partial solutions explored before visiting P plus the number of maximal partial solution explored after visiting P.

Recall that we consider the first time during the execution when a variable with the current domain size d is selected, given that the current partial

solution is not empty. Therefore, before arriving at the considered state, FC-CBJ explores at most $d * (d-1)^{n-2}$ maximal partial solutions, according to the argumentation provided for the first scenario.

All maximal partial solutions explored after visiting P are visited during solving of Z'. Therefore, every maximal partial solution P_1 visited after exploring of P can be represented as $P_1 = P \cup P_2$, where P_2 is a maximal partial solution of Z' (non-maximality of P_2 contradicts maximality of P_1). Thus the number of maximal partial solutions explored after visiting of P equals the number of maximal partial solutions explored by FC-CBJ during solving of Z'.

Considering that P is not empty, it follows that Z' contains at most $n-1$ variables. By the induction assumption, FC-CBJ explores at most $M(n-1)$ of maximal partial solutions during solving of Z'. Thus the overall number of maximal partial solutions is $d * (d-1)^{n-2} + M(n-1) = M(n)$, what completes the proof for the second scenario. ■

Corollary 1 *FC-CBJ with FF explores $O^*((d-1)^n)$ maximal partial solutions.*

Proof. By definition of $M(n)$, $M(n) \leq dn(d-1)^{n-2} = O^*((d-1)^n)$. ■

We have shown that the number of maximal partial solutions explored by FC-CBJ with FF is bounded by $O^*((d-1)^n)$. Clearly, every partial solution is a subset of some maximal partial solution. On the other hand, every maximal partial solution serves as a subset of at most n partial solutions. Actually, every partial solution generated by FC-CBJ corresponds to a node of the search tree explored by FC-CBJ. Note that subsets of the given partial solution P correspond to the ancestors of P in the search tree. Taking into account that every path from the root to a leaf in the search tree has a length of at most n, we infer that P cannot have more than n ancestors. Consequently, the number of partial solutions explored by FC-CBJ is at most the number of maximal partial solutions multiplied by n. Thus we have proven the following theorem.

Theorem 2 *The complexity of FC-CBJ with the fail-first ordering heuristic is $O^*((d-1)^n)$.*

3.2 FC with FF and FC-CBJ have $O^*(d^n)$ complexity

It may seem that a combination of FC-CBJ with FF is far too complex to achieve the purpose of reducing complexity. We will show that this is not

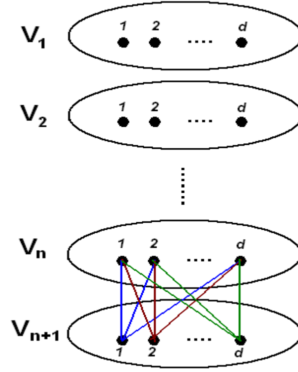

Figure 2: A hard CSP for FC with FF

so. In particular, we will show that both FC (without CBJ) with FF and FC-CBJ alone have a worst-case complexity of $\Omega^*(d^n)$.

Let us start by analyzing the complexity of FC with FF. We prove that for every n there is a CSP Z with $n+1$ variables and d values in every domain (d is an arbitrary number) such that in order to solve Z, the algorithm generates at least d^n partial solutions. Let $v_1, \ldots, v_{n+1}$ be the variables of the considered CSP. Every value of v_n conflicts with every value of v_{n+1}. There are no other conflicts in the CSP. This CSP is illustrated in Figure 2.

We assume that if during the search there are two or more variables with the smallest current domain size, FC with FF will select the first of them according to lexicographic ordering. This is a valid assumption, because we are going to refute the claim that FC with FF has a better complexity than $O^*(d^n)$. To refute the claim, it is sufficient to show that FC combined with FF and a *particular* ordering for equal-sized variables has $\Omega^*(d^n)$ complexity.

Observe that the source of insolubility of Z is that v_n and v_{n+1} have no pair of compatible values. However, FC with the heuristic described above will not be able to recognize the insolubility source, because it will assign first v_1, then v_2, and so on. Note that to refute Z, the algorithm will have to explore all the partial solutions on variables $v_1, \ldots, v_n$. Clearly, there are d^n such partial solutions. Denoting $n+1$ by m we obtain that for every m there is a CSP with m variables such that FC with FF explores at least d^{m-1} partial solutions in solving this CSP. That is, the complexity of FC with FF is $\Omega^*(d^{m-1}) = \Omega^*(1/d * d^m) = \Omega^*(d^m)$ as claimed.(Note that if this CSP is solved by FC-CBJ given the above variable ordering, the conflict set of v_n will be empty in the state where the current domain of v_n becomes empty. Therefore FC-CBJ will jump over all unconstrained variables and

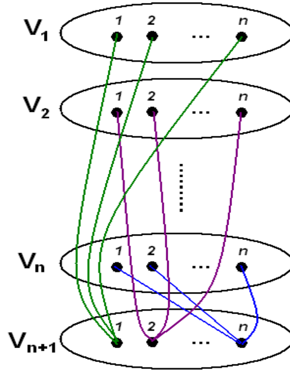

Figure 3: A hard CSP for FC-CBJ

stop.)

Let us now analyze the complexity of FC-CBJ. We show that FC-CBJ has $\Omega^*(d^n)$ complexity. As for the case of FC with FF, we show that there is a CSP Z with $n+1$ variables and maximum domain size d such that FC-CBJ with a certain ordering heuristic generates at least $O(d^n)$ partial solutions during the processing of Z. Note again, that we are free to choose an ordering heuristic for FC-CBJ because FC-CBJ does not specify any particular ordering heuristic.

Let Z be a CSP with $n+1$ variables $v_1, \dots v_{n+1}$. The domain of v_{n+1} contains n values, say, $val_1, \dots, val_n$. The domain of any other variable contains d values, where d is an arbitrary number greater than or equal to n. The only conflicts of Z are found between the values of v_{n+1} and the values of other variables. In particular, a value $\langle v_{n+1}, val_i \rangle$ conflicts with all the values of variable v_i. The CSP is illustrated in Figure 3. We assume that FC-CBJ orders variables lexicographically.

The source of insolubility of Z is that every value of v_{n+1} conflicts with the domain of some variable from $v_1, \dots, v_n$. However, FC-CBJ is unable to recognize the insolubility source because it assigns v_{n+1} last, according to the specified ordering heuristic. Observe that all maximal partial solutions generated by FC-CBJ are of the form $\{\langle v_1, val_{i_1}, \rangle, \dots, \langle v_n, val_{i_n} \rangle\}$, because no assignment to a proper subset of $\{v_1, \dots, v_n\}$ can discard all the values of v_{n+1}. Clearly, there are d^n partial solutions of the above form and we shall show that FC-CBJ explores all of them, which proves its $\Omega^*(d^n)$ complexity.

To show that FC-CBJ explores all the partial solutions of the form $\{\langle v_1, val_{i_1}, \rangle, \dots, \langle v_n, val_{i_n} \rangle\}$, it is sufficient to show that FC-CBJ never backjumps more than 1 step backwards when applied to Z. First, we show that

FC-CBJ never backjumps when exploring a maximal partial solution. Actually, an assignment of every v_i conflicts only with $\langle v_{n+1}, val_i \rangle$. That is, every assignment of a maximal partial solution conflicts with the unique value of v_{n+1}. This means that when the current domain of v_{n+1} is emptied, all the variables of $\{v_1, \dots, v_n\}$ appear in $conf(v_{n+1})$. Therefore, FC-CBJ will discard the assignment of v_n.

When the assignment of v_n is discarded, it is only because the current domain of v_{n+1} was emptied. Therefore, the set $\{v_1, \dots, v_n\} \setminus \{v_n\}$ is added to the conflict set of v_n and, when the domain of v_n is emptied, FC-CBJ has no choice but to backtrack to v_{n-1}. Tracing further the execution of FC-CBJ, we observe that whenever an assignment of v_i is discarded, the set $\{v_1, \dots, v_{i-1}\}$ is added to $conf(v_i)$. Hence, when the current domain of v_i is emptied, FC-CBJ backtracks to v_{i-1}. This argumentation shows that FC-CBJ applied to Z with the lexicographic ordering heuristic never backjumps and thus explores at least d^n partial solutions.

4 Algorithms with the branching factor of $\lceil d/2 \rceil$

4.1 Overview of the method

In this section we design CSP algorithms utilizing the approach shown in Section 2.4. These algorithms have a worst-case complexity of $O^*(\lceil d/2 \rceil^n)$. Recall that the approach described in Section 2.4 is impractical because it does not guarantee that when all variables are assigned, the resulting 2-CSP has a solution. Consequently, this algorithm can potentially perform exhaustive search which results in a non-reasonable runtime even for small CSPs.

To avoid the exhaustive search, we modify the search algorithm so that whenever it assigns all the variables, the resulting 2-CSP is soluble. In particular, the search process is organized as follows:

- Variables are assigned one by one. A variable is assigned with two values if its current domain contains at least two values. Otherwise, the variable is assigned with one value. In other words, at every step of the algorithm, a variable v is assigned with a nonempty subset A of its current domain, where $|A| \leq 2$.
- Whenever a variable is assigned with a set of values A, the algorithm removes all the values of unassigned variables that are incompatible with all the values of A.

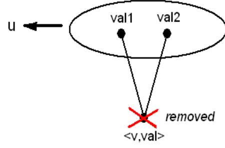

Figure 4: Removal of a value

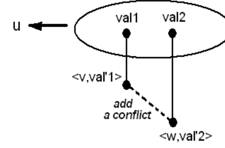

Figure 5: Adding a conflict

This situation is illustrated in Figure 4. In this figure a variable u is assigned with two values: val_1 and val_2. Both values are incompatible with a value $\langle v, val \rangle$, hence the latter must be removed.

- The algorithm backtracks when the current domain of some variable becomes empty.
- Whenever a variable u is assigned with a set $\{\langle u, val_1 \rangle, \langle u, val_2 \rangle\}$, a new conflict is added between any pair $(\langle v, val'_1 \rangle, \langle w, val'_2 \rangle)$ of values of different unassigned variables v and w such that $\langle v, val'_1 \rangle$ conflicts with $\langle u, val_1 \rangle$ and $\langle w, val'_2 \rangle$ conflicts with $\langle u, val_2 \rangle$. The addition of a conflict between values of unassigned variables is depicted in Figure 5.
.

The proposed search procedure resembles FC with the main difference that a variable can be assigned with two values. Therefore we call the algorithm 2FC. The main property of 2FC is that whenever all variables are assigned, the resulting 2-CSP is *guaranteed* to have a solution.

The rest of the section provides a precise description of 2FC, proves the correctness of 2FC, and transforms it into 2MAC.

4.2 The 2FC algorithm

The 2FC algorithm maintains the following auxiliary data structures: $CurSol$, *assigned*, *validity*, and *constraint*.

$CurSol$ is an ordered list of pairs of assignments. It contains the current 2CSP maintained by 2FC. We will further prove that this 2CSP is always soluble.

The Boolean array *assigned* has one entry per variable. $assigned[v] = true$ means that the variable v is assigned by the current partial solution. Otherwise, $assigned[v] = false$. We refer to former variables as *assigned* and to latter variables as *unassigned*

During execution of FC, values are frequently removed from their domains and restored back. To remember which values are removed and which

are not, the algorithm maintains an array *validity* with one entry per value of the underlying CSP. If $validity[\langle v, val \rangle] =' VALID'$, the value is not removed. For the removed values, there are two possible ways to mark the corresponding entry of *validity*. These are discussed during the description of the pseudocode. We call the set of values of a variable v that are not removed at a particular moment of execution of FC *the current domain* of v.

Along with removing and restoring values, the 2FC algorithm also add and removes conflicts. The current set of conflicts is maintained in the array *constraint*. For every unordered pair of values $\langle v_1, val_1 \rangle$ and $\langle v_1, val_2 \rangle$, the corresponding entry $constraint[\{\langle v_1, val_1 \rangle, \langle v_2, val_2 \rangle]$ contains information about the compatibility of these values. Initially, the value of an entry can be either $'CONSISTENT'$ or $'INCONSISTENT'$, the former means that the corresponding values are compatible and the latter means that the corresponding values are incompatible. The entries whose content is $'INCONSISTENT'$ are not changed during execution of the algorithm, but the entries whose content is $'CONSISTENT'$ can change dynamically. The way the entries of *constraint* are updated is analogous to the technique of updating of the entries of *validity*. These are discussed during the pseudocode description.

Algorithm 1 2FC

```
1:  Let Z be the CSP being solved
2:  CurSol ← ∅
3:  for every variable v of V(Z) do
4:     assigned[v] ← false
5:     for every value ⟨v, val⟩ do
6:        validity[⟨v, val⟩] =′ VALID′
7:     end for
8:  end for
9:  R = 2fc − rec()
10: if R =′ SUCCESS′ then
11:    Extract a solution from CurSol
12: else
13:    Report insolubility
14: end if
```

Algorithm 1 presents the initial part of 2FC. The structures *CurSol*, *assigned* and *validity* are initialized and the function $2fc - rec$ is applied.

The function $2fc - rec$ is the main search engine of 2FC We assume that the above structures are global, so that they can be read and updated by function $2fc - rec$. If the function obtains $CurSol$ that assigns all the variables with pairs of their values, it returns $'SUCCESS'$; otherwise, it returns $'FAIL'$. In case $'SUCCESS'$ is returned, 2FC extracts a solution from $CurSol$ (line 11 of Algorithm 1). As we will show further, the 2CSP contained in $CurSol$ is always soluble, therefore such extraction is always possible. In case $2fc-rec$ returns $'FAIL'$, the underlying CSP is insoluble, hence 2FC reports insolubility.

Function $2fc-rec$ is presented in Algorithm 2 The initial part of $2fc-rec$ (lines 1-6) checks two stopping conditions. If all the variables are assigned, the function propagates the $'SUCCESS'$ message to the upper level. If there is a variable with an empty current domain, the $'FAIL'$ message is propagated.

When $2fc-rec$ executes line 7, it is guaranteed that there are unassigned variables and the current domains of all unassigned variables are not empty. An unassigned variable v is selected in line 7. The cycle in lines 8-29 scans over all values of the current domain of v.

Lines 9-14 of Algorithm 2 describe the selection of values to be assigned to the current variable v. If the current domain of variable v contains at least two values then two values are selected. Otherwise, $2fc - rec$ selects the only value that remains in the domain. For a uniform presentation of these two cases, we state that a variable v is assigned with a set A of values which may be either of size 1 or of size two.

After set A is selected, it is appended to $CurSol$ (line 15), the array *assigned* is updated (line 16), appropriate conflicts are added in case $|A| = 2$ (lines 17-19),and the values of unassigned variables conflicting with A are removed (line 20). The addition of conflicts and the removal of values are performed by functions $2fc-add-conflicts$ (Algorithm 3) and $2fc-lookahead$ (Algorithm 4), respectively. In order to add a conflict between compatible values $\langle u, val'_1 \rangle$ and $\langle w, val'_2 \rangle$, $|CurSol|$ is inserted into the corresponding entry of the *constraint* array (line 6 of Algorithm 3). The length of $CurSol$ serves as an indication of the conflicts that were added as a result of assigning A to v. This indication is necessary because when the assignment is discarded, the conflicts caused by it must be discarded too. The same technique is used for removal of values (line 4 of Algorithm 4).

In line 21, $2fc - rec$ is applied recursively. Further execution depends on the answer R returned by the recursive call. If $R =' SUCCESS'$, this message is propagated to the upper level. Otherwise, $2fc - rec$ restores all the values that were removed because of incompatibility with $\langle v, val \rangle$ (func-

Algorithm 2 $2fc - rec()$

1: **if** all variables are assigned **then**
2: Return $'SUCCESS$
3: **end if**
4: **if** there is a variable v with the empty current domain **then**
5: Return $'FAIL'$
6: **end if**
7: Select an unassigned variable v
8: **while** the current domain of v is not empty **do**
9: **if** the current domain of v contains at least two values **then**
10: Select from the current domain of v two values $\langle v, val_1 \rangle$ and $\langle v, val_2 \rangle$
11: $A \leftarrow \{\langle v, val_1 \rangle, \langle v, val_2 \rangle\}$
12: **else**
13: Let A be the singleton containing the only value of the current domain of v
14: **end if**
15: Append A to $CurSol$
16: $assigned[v] = true$
17: **if** $|A| = 2$ **then**
18: $2fc - add - conflicts(\langle v, val_1 \rangle, \langle v, val_2 \rangle)$
19: **end if**
20: $2fc - lookahead(A)$
21: $R = 2fc - rec()$
22: **if** $R =' SUCCESS'$ **then**
23: Return R
24: **else**
25: $2fc - restore - val()$
26: remove the last entry from $CurSol$
27: Set $validity[\langle v, val \rangle] =' INVALID'$ for every $\langle v, val \rangle \in A$
28: **end if**
29: **end while**
30: Set $validity[\langle v, val \rangle]$ to $'VALID'$ for all $\langle v, val \rangle$ where $validity[\langle v, val \rangle] =' INVALID'$
31: Return $'FAIL'$

tion $2fc-restore-val$ is described in Algorithm 5), removes the last entry from $CurSol$ and writes $'INVALID'$ to the *validity* entries corresponding to all the values of A. Note the way the algorithm distinguishes between the values removed by inference and the values removed because of incompatibility with the last assignment. When a value is removed by inference, 2FC writes $'INVALID'$ in the corresponding *validity* entry; otherwise the entry is set tot he length of $CurSol$.

Further execution of $2fc-rec$ depends on the size of the current domain of v. If the size is greater than 0, another value is selected in the next iteration of the cycle. Otherwise, the function exits from the cycle, restores $'VALID'$ for all entries marked by $'INVALID'$ in the cycle and propagates the $'FAIL'$ message to the upper level.

Algorithm 3 $2fc-add-conflicts(\langle v, val_1\rangle, \langle v, val_2\rangle)$

1: **for** every unassigned variable u **do**
2:   **for** every valid value $\langle u, val'_1\rangle$ **do**
3:     **for** every unassigned variable $w \neq u$ **do**
4:       **for** every valid value $\langle w, val'_2\rangle$ compatible with $\langle u, val'_1\rangle$ **do**
5:         **if** $\langle u, val'_1\rangle$ is incompatible with $\langle v, val_1\rangle$ and $\langle u, val'_2\rangle$ is incompatible with $\langle v, val_1\rangle$ **then**
6:           $constraint[\{\langle u, val'_1\rangle\langle w, val'_2\rangle\}] \leftarrow |CurSol|$
7:         **end if**
8:       **end for**
9:     **end for**
10:   **end for**
11: **end for**

Algorithm 4 $2fc-lookahead(A)$

1: **for** every variable u with $assigned[u] = false$ **do**
2:   **for** every value $\langle u, val'\rangle$ with $validity[\langle u, val'\rangle] =' VALID'$ **do**
3:     **if** every value of A is incompatible with $\langle u, val'\rangle$ **then**
4:       $validity[\langle u, val'\rangle] \leftarrow |CurSol|$
5:     **end if**
6:   **end for**
7: **end for**

Algorithm 5 $2fc - restore - val()$

```
1: for every entry constraint[x] = |CurSol| do
2:    constraint[x] ← 'CONSISTENT'
3: end for
4: for every variable u with assigned[u] = false do
5:    for every value ⟨u, val'⟩ with validity[⟨u, val'⟩] = |CurSol| do
6:       validity[⟨u, val'⟩] ← 'VALID'
7:    end for
8: end for
```

4.3 Correctness proof for 2FC

To prove that 2FC is correct, we have to prove that the algorithm terminates and also that it is sound and complete. Termination easily follows from the fact that the search scheme of 2FC is analogous to that of FC which is known to terminate from [15]. Soundness means that whenever 2FC assigns all the variables, the resulting 2-CSP will have a solution. Completeness means that whenever a CSP is soluble, 2FC, being applied to the CSP, returns a solution.

We start with the proof of soundness. First, we extend our terminology by introducing the notions of directional arc-consistency and directional path-consistency. The definitions are taken from [5] but are slightly modified.

Definition 3 *A CSP Z is called directionally arc-consistent with respect to an order* $v_1, \dots, v_n$ *of its variables if every value* $\langle v_i, val\rangle$ *is consistent with a domain of a variable* v_k *whenever* $k < i$.

Definition 4 *A CSP Z is called directionally path-consistent with respect to the order* $v_1, \dots, v_n$ *of its variables if every pair of compatible values* $\{\langle v_i, val_i\rangle, \langle v_k, val_k\rangle\}$ *is consistent with the domain of a variable* v_l *whenever* $l < i$ *and* $l < k$.

Soundness is proved by two lemmas, one of which claims that a directionally arc and path-consistent 2-CSP with no empty domains is always soluble. The second lemma shows that in every state of 2FC the 2-CSP contained in $CurSol$ satisfies these properties and, consequently, is soluble.

Lemma 2 *Let Z be a 2-CSP with no empty domain which is directionally arc-consistent and directionally path consistent with respect to an order $v_1, \dots, v_n$ of variables of Z. Then Z is soluble.* [1]

Proof. The proof is by induction on n, the number of variables of Z. It is trivial for $n = 1$. For $n > 1$, assign v_n with a value val_n that belongs to its domain. Let Z' be a 2-CSP obtained from Z by removing v_n and deleting from the domains of the rest of variables all values that are incompatible with $\langle v_n, val_n \rangle$. Observe the following properties of Z'.

- The domains of all variables of Z' are not empty. Actually, an empty domain of some variable v in Z' would mean that $\langle v_n, val_n \rangle$ conflicts with all the values of the domain of v in Z, which is in contradiction to the directional arc-consistency of Z.

- Z' is directionally arc-consistent with respect to the order $v_1, \dots, v_{n-1}$. Assume by contradiction that a value $\langle v_k, val \rangle$ is inconsistent with the domain of v_i, $i < k$. If the domain of v_i in Z' is the same as in Z, $\langle v_k, val \rangle$ is inconsistent with v_i in Z, which is in contradiction to our assumption about directional arc-consistency of Z. Otherwise, one of the values of the domain of v_i is incompatible with $\langle v_n, val_n \rangle$, the other is incompatible with $\langle v_k, val \rangle$, while $\langle v_n, val_n \rangle$ and $\langle v_k, val \rangle$ are compatible. In this case we get a contradiction with our assumption about directional path-consistency of Z.

- Z' is directionally path-consistent. Otherwise, if we have two compatible values $\langle v_k, val_k \rangle$ and $\langle v_l, val_l \rangle$ that conflict with the domain of some variable v_i, $(i < k, l)$, the same situation occurs in Z, which is in contradiction to directional path-consistency of Z.

Thus Z' satisfies all conditions of the lemma and has $n - 1$ variables. Therefore Z' is soluble by the induction assumption. Let S be a solution of Z'. Note that all values of Z' are compatible with $\langle v_n, val_n \rangle$. Therefore $S \cup \{\langle v_n, val_n \rangle\}$ is a solution of Z. ■.

Lemma 3 *2FC is sound.*

Proof. We show that in every state visited by 2FC, $CurSol$ contains a 2-CSP satisfying the conditions of Lemma 2. Soundness will follow from the statement of Lemma 2.

[1] Note that the suggested sufficient condition of solubility of 2-CSPs is weaker than path-consistency whose sufficiency is proved in [14].

It is clear that in every state of 2FC, the CSP maintained by $CurSol$ has no variable with an empty domain, for otherwise it would mean that a variable has been assigned with an empty set. Thus, it remains to show that the 2-CSP recorded by $CurSol$ is directional arc and path-consistent.

Let $v_1, \ldots v_n$ be the variables assigned by $CurSol$ and enumerated in the chronological order of their assignment. We apply induction on n. (We assume that the constraints are given by the *constraint* array of the considered state).

The statement is trivial for $n = 1$. For $n > 1$, by the induction assumption the first $n - 1$ variables create a 2-CSP satisfying the conditions. Thus, arc or path-consistency can be broken only by the values of the domain of v_n. Breaking of directional arc-consistency means that a value $\langle v_n, val \rangle$ conflicts with the domain of some v_i, $(i < n)$. However in this case $\langle v_n, val \rangle$ would have been removed by $2fc - lookahead$ applied after assignment of v_i. [2]. Thus the considered 2-CSP is directionally arc-consistent.

For directional path-consistency, take $i < k < n$ and assume that a value $\langle v_k, val_1 \rangle$ is compatible with $\langle v_n, val_2 \rangle$ and $\{\langle v_k, val_1 \rangle, \langle v_n, val_2 \rangle\}$ conflicts with the domain of v_i. This situation is impossible, because after the assignment of v_i, the $2fc - add - conflicts$ function would add a conflict between $\langle v_k, val_1 \rangle$ and $\langle v_n, val_2 \rangle$.

We have shown that for every state of 2FC, the 2-CSP contained in $CurSol$ is soluble with respect to the *current* CSP. This CSP is, of course, soluble with respect to the *original* CSP because no original conflict is removed by 2FC. ■

Lemma 4 *2FC is complete.*

Proof. We prove completeness by induction on the number n of variables.

Completeness follows immediately for $n = 1$. For $n > 1$, let v be the variable that 2FC selects to be assigned first. If the underlying CSP is soluble then 2FC eventually assigns v with a set of values S, one of which belongs to a solution. Then 2FC removes from the domains of the rest of the variables all values that are inconsistent with S and adds conflicts between pairs of compatible values that conflict with S. Note that the removed values can neither be in the same solution with any value of S nor be pairs of values that are made incompatible. Therefore 2FC is applied recursively to a soluble CSP where it finds a solution by the induction assumption. ■

[2]Note that it is impossible that conflicts with v_i appear when v_i has been already assigned because 2FC adds conflicts only between unassigned variables

This completes the proof of the following theorem.

Theorem 3 *2FC is correct.*

4.4 The 2MAC algorithm

The transformation of 2FC into 2MAC is similar to the transformation of FC into MAC. First, arc-consistency of the underlying CSP must be achieved at the preprocessing stage. Next, the call to function $2fc-lookahead(A)$ (line 20 of Algorithm 2) must be replaced by a call to function $2maintain_mac(A)$, which is presented in Algorithm 6. The replacement is done in order to maintain arc-consistency of the CSP created by the current domains of unassigned variables.

The function $2maintain-ac$ achieves arc-consistency using the AC-3 algorithm [5]. It is not only an efficient algorithm but it is also quite simple. The main data structure of the algorithm is the queue *ac_queue* of values to be removed. The initialization of ac_queue is divided into 3 stages. First, *ac_queue* is emptied (line 1). Then (lines 2-8) the values of the unassigned variables that are incompatible with all the values of the current assignment are inserted into *ac_queue*. Finally, (lines 9-15), the function inserts into *ac_queue* the values of unassigned variables that conflict with other unassigned variables. A cycle in lines 16-26 scans the queue from beginning to end, dequeues an element from it, say $\langle v, val \rangle$, removes it from the current domain of v, and adds to the queue the values that have lost their arc-consistency with respect to v as a result of the deletion of $\langle v, val \rangle$.

Finally, we emphasize again that $2maintain-ac$ considers two values to be compatible if and only if the corresponding entry in the *constraint* array equals $'CONSISTENT'$.

The soundness of 2MAC can be proven analogously to the soundness proof of 2FC. The completeness proof must take into account the additional values removed by $2maintain-ac$ that are not removed by $2fc-lookahead$. Observe that every such a value conflicts with the current domain of some variable at the time of its removal. Clearly, such a value cannot participate in any solution of the CSP created by the current domains of unassigned variables and *constraint* array. Combining this argument with the inductive reasoning used in the proof of Lemma 4 results in a completeness proof.

4.5 Optimal design of 2MAC

Traditionally, during search, AC is achieved for a CSP created by the current domains of unassigned variables. However, an alternative form of achieving

Algorithm 6 $2maintain - ac(A)$

```
1: ac_queue ← ∅
2: for every variable u with assigned[u] = false do
3:    for every value ⟨u, val′⟩ with validity[⟨u, val′⟩] =′ VALID′ do
4:       if ⟨v, val⟩ is incompatible with all the values of A then
5:          enqueue(ac_queue, ⟨u, val′⟩)
6:       end if
7:    end for
8: end for
9: for every variable unassigned v do
10:   for every value ⟨v, val⟩ do
11:      if ⟨v, val⟩ conflicts with the domain of another unassigned variable
         u then
12:         enqueue(ac_queue, ⟨v, val⟩)
13:      end if
14:   end for
15: end for
16: while ac_queue is not empty do
17:   ⟨u, val′⟩ = dequeue(ac_queue)
18:   validity[⟨u, val′⟩] ← |CurSol|
19:   for every variable w except u do
20:      for every value ⟨w, val′′⟩ do
21:         if ⟨w, val′′⟩ is in conflict with the domain of u then
22:            enqueue(ac_queue, ⟨w, val′′⟩)
23:         end if
24:      end for
25:   end for
26: end while
```

AC is possible, where AC is achieved for a CSP including all the variables of the original CSP. The domain of a variable v in this CSP is its current domain whenever v is unassigned. Otherwise, the domain includes all the values that v is assigned with.

This alternative form of maintaining AC is not considered when every variable is assigned with one value because all values incompatible with the assignments are removed from the domains of unassigned variables and new incompatible values cannot appear during achieving AC. That is, incorporating assigned variables into the process of achieving of AC results in no additional pruning. However, it is not clear whether the same is true when a variable is assigned two values.

When a variable v is assigned two values, say $\langle v, val_1 \rangle$ and $\langle v, val_2 \rangle$, the values of current domains of unassigned variables may be incompatible with any one of them (not with both of them). Therefore it might seem that the alternative form of achieving AC can result in additional pruning. However, below we show that this is not true.

In a state of 2MAC, we consider two CSPs, say Z_1 and Z_2. Z_1 is created by the current domains of unassigned variables, the conflicts between the values are determined by the *constraint* array. In addition to the variables of Z_1, Z_2 includes all assigned variables. The domains of assigned variables are the set they are assigned with in $CurSol$. As in Z_1, the conflicts are determined by the *constraint* array. The theorem proven below states that the process of achieving arc-consistency removes from the domains of unassigned variables in Z_1 the same values that are removed from the domains of these variables in Z_2. It follows from the theorem that the form of achieving AC in 2MAC is "optimal" in the sense that assigned variables do not help to refine domains of unassigned variables.

We start by extending our terminology.

Definition 5 *A pair of variables u and v are path-consistent with respect to a variable w if for every pair of compatible values $\langle u, val_1 \rangle$ and $\langle v, val_2 \rangle$, there is a value $\langle w, val_3 \rangle$ compatible with both of them.*

Let Z be a CSP. We denote by $AC(Z)$ the CSP obtained from Z by a procedure achieving arc-consistency. (for example $AC(Z)$ can be obtained as a result of applying $preprocess - ac$ to Z).

Definition 6 *A projection of a CSP Z to a set of variables S denoted by $Pr(Z, S)$ is the CSP obtained by removing from Z all the variables that are not in S.*

Now we prove a theorem that will be directly applied to show the "optimality" of 2MAC.

Theorem 4 *Let Z be a CSP. Denote by A the subset of variables of Z with domain sizes of at most 2. Denote $V(Z) \setminus A$ by U. Assume that Z has the following properties:*

- *the domains of all variables are not empty;*
- *$Pr(Z, A)$ is directionally arc-consistent and directionally path-consistent with respect to the same order;*
- *every value $\langle v, val \rangle$ of every variable $v \in U$ is consistent with every variable of A;*
- *every pair of variables v and w of U are path-consistent with every variable of A;*
- *let u and v be variables of A such that u precedes v in the order with respect to which $Pr(Z, A)$ is directionally arc and path-consistent; let w be a variable of U; then v and w are path-consistent with respect to u.*

The above conditions imply that $Pr(AC(Z), U) = AC(Pr(Z, U))$.

Proof. Recall that achieving AC for Z means iterative removal of values that are not consistent with the domains of other variables. What is interesting is that the order of removing such inconsistent values is irrelevant since at the end of the removal process one obtains the same CSP regardless of the order by which the values have been removed. We utilize this property to prove the present theorem.

We prove the theorem by induction on $|A|$. For $|A| = 1$, we have exactly one variable, say v, with the domain size of at most 2.

Assume that in order to achieve arc-consistency for Z, we achieve first arc-consistency for $Pr(Z, V(Z) \setminus \{v\})$. That is we remove at the first stage all values $\langle u, val \rangle$ for $u \neq v$ that are inconsistent with the domains of variables of $V(Z) \setminus \{v\}$. At the second stage, we achieve arc-consistency for the whole of Z. According to the above discussion, this order does not affect the final CSP.

Suppose we have achieved AC for $Pr(Z, V(Z) \setminus \{v\})$ and now we are going to execute the second stage, that is, to achieve AC for the whole of Z. We will show that during the second stage no value will be removed from the domains of variables of $V(Z) \setminus \{v\}$.

A natural question that now arises is what causes a value $\langle u, val \rangle$ of variable $u \in V(Z) \setminus \{v\}$ to be removed? There are two explanations. One, $\langle u, val \rangle$ may be inconsistent with the domain of v. This is infeasible because we assumed that all values of $V(Z) \setminus A$ are consistent with the domains of A. To understand the second reason, let $\{\langle v, val_1 \rangle, \langle v, val_2 \rangle\}$ be the domain of v. Then, it is possible that $\langle v, val_1 \rangle$ is inconsistent with some variable w and $\langle u, val \rangle$ is inconsistent with $\langle v, val_2 \rangle$. Thus after removal of $\langle v, val_2 \rangle$, the value $\langle u, val \rangle$ must be removed.

We show that this situation also cannot occur, because in this case $\langle u, val \rangle$ is inconsistent with the domain of w and thus must be removed during the first stage of achieving AC. If there is a value $\langle w, val_3 \rangle$ that is compatible with $\langle u, val \rangle$, then these two values are not path-consistent with the domain of v, which contradicts the construction of Z. Thus we have shown that for $|A| = 1$ the theorem holds.

Assume now that $|A| = m > 1$ and that the theorem holds for $|A| < m$. Let $v_1, \dots v_m$ be the variables of A enumerated in the order according to which $Pr(Z, A)$ is directionally arc and path-consistent. Assume again that we achieve arc-consistency in two stages: in the first stage for $Pr(Z, V(Z) \setminus \{v_1\})$ and in the second stage for the whole of Z. Note that $Pr(Z, V(Z) \setminus \{v_1\})$ satisfies the four conditions stated by the present theorem, hence the theorem holds for this CSP by the induction assumption. Let Z_1 be the CSP obtained after achieving arc-consistency for $Pr(Z, V(Z) \setminus \{v_1\})$. We will show that for achieving arc-consistency for Z_1, one removes no value from the domains of $V(Z) \setminus \{v_1\}$ and, of course, from the domains of $V(Z) \setminus A$.

Below we show that every value of $V(Z) \setminus \{v_1\}$ is consistent with respect to v_1 and that every pair of variables of $V(Z) \setminus \{v_1\}$ is path-consistent with respect to v_1. Then the same argumentation we used for the case $|A| = 1$ applies here as well.

Let $w \in V(Z) \setminus \{v_1\}$. If $w \in A$ then any $\langle w, val \rangle$ is consistent with respect to v_1 because v_1 precedes w in the order determining the direction of arc and path-consistency of $Pr(Z, A)$. If $w \in U$ then any $\langle w, val \rangle$ is consistent with respect to v_1 as well according to one of the conditions of the present theorem.

Consider two variables v and w of $V(Z) \setminus \{v_1\}$. If both belong to A then $\{v_1\}$ clearly precedes them in the order determining arc and path-consistency of $Pr(Z, A)$. Then v and w are path-consistent with v_1 according to the definition of directional path-consistency.

If both v and w belong to U, they are path-consistent with respect to u as well according to one of the conditions of the present theorem.

Finally, if one of them, say v belongs to A and the other belongs to U

then v_1 clearly precedes v in the order determining arc and path-consistency of $Pr(Z, A)$. Path-consistency of v and w with respect to v_1 follows from the last condition of the theorem.

Thus to prove that no value of the variables of $V(Z) \setminus \{v_1\}$ is removed during the second stage of achieving AC, we can apply analogous argumentation to that used for the case where $|A| = 1$ ■

Now we formulate and prove the theorem that states the "optimality" of 2MAC.

Theorem 5 *Consider a state of 2MAC that occurs just after application of the* $2maintain-ac$ *procedure. Let* Z *be a CSP with the same set of variables as in the underlying CSP. The domains of the assigned variables are the sets assigned to them. The domains of the unassigned variables are their current domains in the considered state. The conflicts are determined by the constraint array of the considered state. Let* U *be the set of unassigned variables of the given state. Then* $Pr(AC(Z), U) = Pr(Z, U)$.

Proof. Denote $V(Z) \setminus U$ by A. Observe that Z satisfies all the conditions of Theorem 4. Then, it follows from Theorem 4 that $Pr(AC(Z), U) = AC(Pr(Z, U))$. Taking into account that in the considered state $Pr(Z, U) = AC(Pr(Z, U))$, the theorem follows. ■

4.6 Experimental evaluation

To evaluate the proposed pruning technique, we compare 2MAC with MAC [24]. [3] Two measures of performance are used: the number of nodes visited and the runtime. Runtime is preferable to the number of consistency checks because the latter is proportional to runtime but does not take into account additional computational overhead. Every measure is obtained as an average of 50 runs.

Both of the algorithms order variables by the Fail-First heuristic [12] which selects a variable with the smallest current domain size. MAC orders the values within each variable by the min-conflict heuristic [10]. 2MAC selects a pair of values by the following heuristic. The first value is selected by min-conflict heuristic. Then the same heuristic is applied to the remaining values in order to select the second value.

First, we apply the compared algorithms to the Graph k-Coloring problem. We generate random Graph k-Coloring problems by specifying the

[3] The results of comparing 2FC to FC look similar [21]. They are omitted for the sake of brevity

number of vertices, the number of colors and the density. The resulting CSP has the set of variables corresponding to the set of vertices, the domain of every variable corresponds to the set of colors. The pairs of variables that correspond to the pairs of adjacent vertices are constrained by the inequality constraint.

We compare algorithms on three sets of instances. For every set of instances we fix the number of vertices and the number of colors and vary the density (the probability for an edge between two given vertices). The fixed parameters are selected as follows. In the first set of instances, the phase transition region falls in the area of small densities. In the second set of instances, the phase transition occurs for medium densities. In the last set the hardest instances have a high density. The results are presented in Figures 6- 11.

The results for every set of parameters are represented by two graphs: one on the left side one compares the number of nodes visited, the other on the right side compares the runtime. The solid-line graphs represent the behavior of 2MAC while the dotted-line graphs represent the behavior of standard MAC.

For every set of instances, we show the values of density that lie close to the phase-transition region for that set of instances. For the values of density that lie farther, both algorithms finish with almost no backtracks, hence their comparison is uninteresting.

It is easy to see that 2MAC works much better than MAC on hard instances of Graph k-Coloring problems. In particular, the performance of 2MAC, when measured by the number of nodes visited, is more than 10 times better over the whole range of densities. Also, according to the runtime measures, 2MAC runs from 4.5 to 9 times faster than MAC.

The next problem for which 2MAC and MAC are compared is the Subgraph Isomorphism Problem defined as follows. Given two graphs G_1 and G_2, the task is to determine whether G_2 is a subgraph of G_1. To encode the problem as a CSP, we assume that G_1 and G_2 have the same number of vertices (denoted by n) and that the vertices of the graphs are labeled by $\{1, \dots, n\}$.

The Subgraph Isomorphism problem is represented as a CSP Z as follows. The variables of Z are denoted by $V_1, \dots, V_n$. The values of every domain of Z are denoted by $\{1, \dots, n\}$. The constraints are defined by the following two rules:

- for every pair of nonadjacent vertices i and j of G_2, the variables V_i and V_j are connected by the inequality constraint;

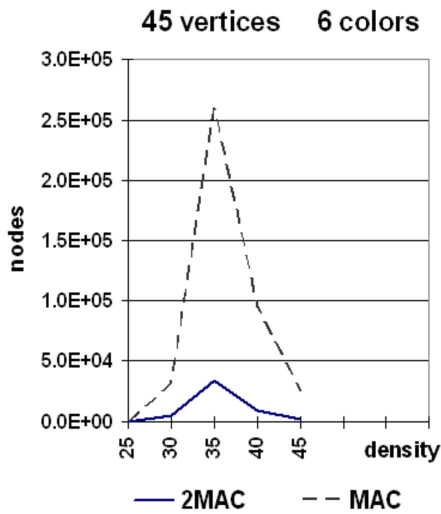

Figure 6: Graph k-Coloring: nodes visited for hard problems of low density

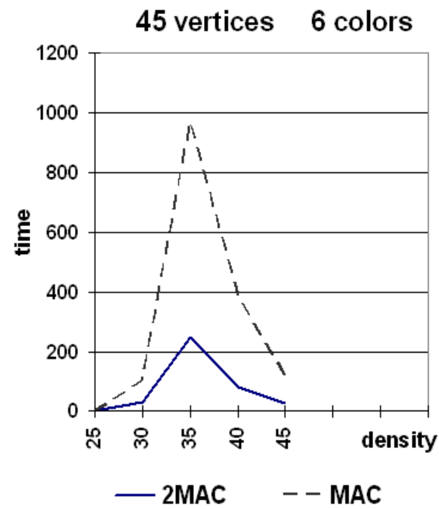

Figure 7: Graph k-Coloring: runtime for hard problems of low density

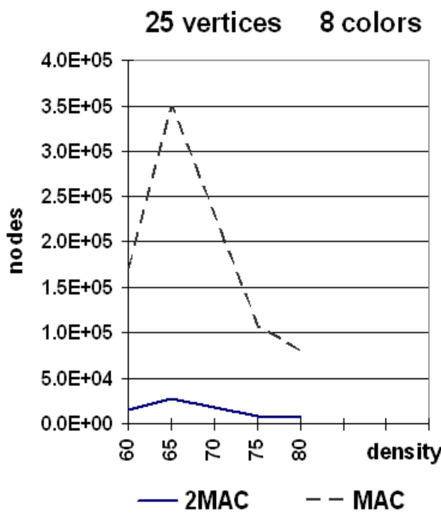

Figure 8: Graph k-Coloring: nodes visited for hard problems of medium density

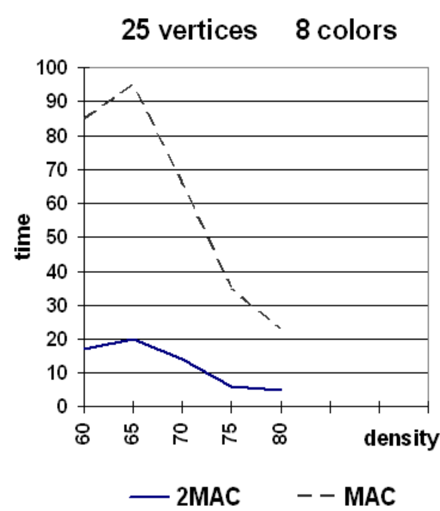

Figure 9: Graph k-Coloring: runtime for hard problems of medium density

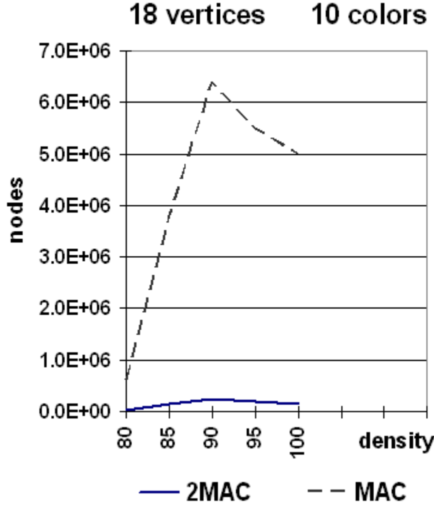

Figure 10: Graph k-Coloring: nodes visited for hard problems of high density

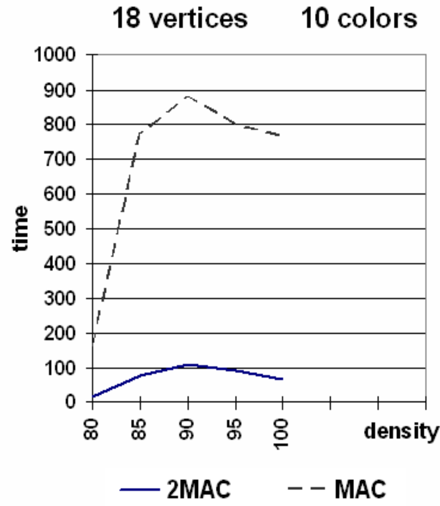

Figure 11: Graph k-Coloring: runtime for hard problems of high density

- for every pair i and j of adjacent vertices of G_2, $\langle V_i, k\rangle$ and $\langle V_j, l\rangle$ are compatible if and only if the vertices k and l are adjacent in G_1.

It is easy to verify that the CSP Z has a solution if and only if G_2 is a subgraph of G_1. Any solution of Z is a mapping from vertices of G_2 to the vertices of G_1. A vertex i of G_2 is mapped to a vertex k of G_1 if V_i is assigned the value k in the solution.

We generate the instances randomly given the number of vertices, the density of G_1, and the density of G_2. We test these algorithms on three sets of instances. For every set of instances we fix the number of vertices of the graphs as well as the density of G_1 and vary the density of G_2. In the first set the phase transition region falls in the region of small values of density of G_2. In the second set of instances, the phase transition occurs for medium densities of G_2. In the last set, the hardest instances occur when G_2 has a high density. The results of the experiments are presented in Figures 12-17.

As can be seen from the results, 2MAC again performs much better than MAC. The factor of improvement in the number of nodes visited varies from about 2.6 for graphs with a low and average density to about 12.5 for graphs with a high density. The factor of runtime improvement varies from about 1.7 for graphs with a low and average density to about 7 for graphs with a

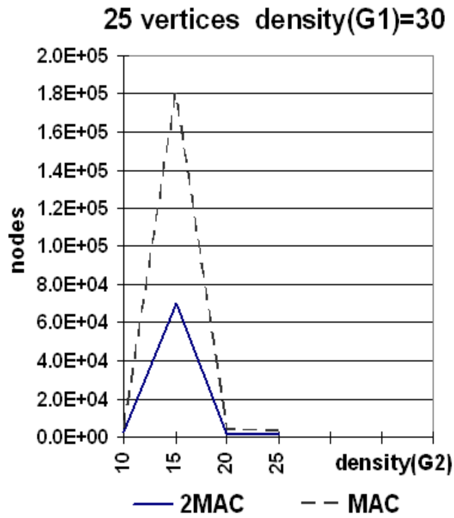

Figure 12: Subgraph Isomorphism: nodes visited for hard instances of low density

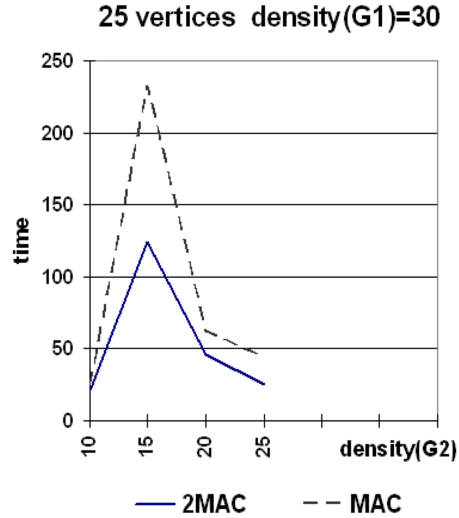

Figure 13: Subgraph Isomorphism: runtime for hard instances of low density

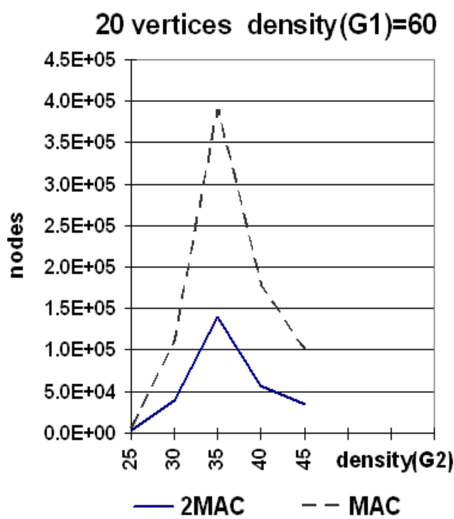

Figure 14: Subgraph Isomorphism: nodes visited for hard instances of medium density

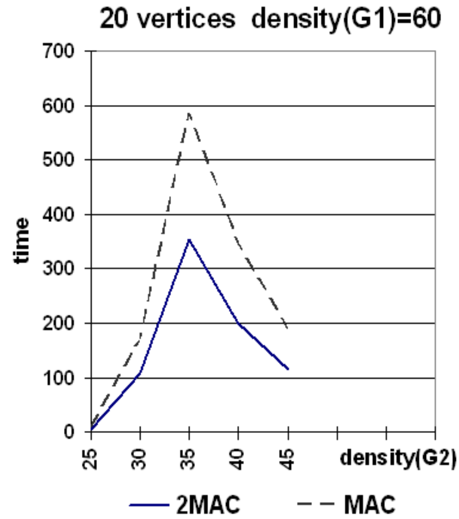

Figure 15: Subgraph Isomorphism: runtime for hard instances of medium density

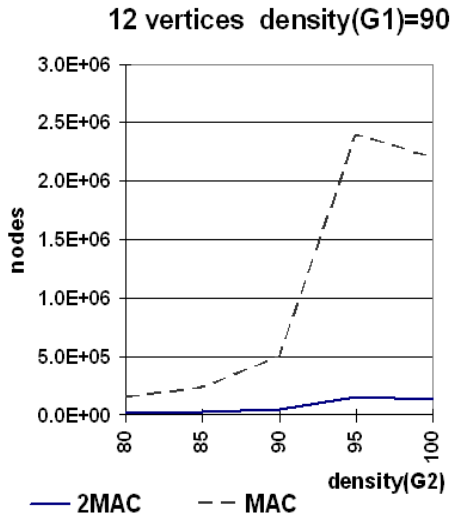

Figure 16: Subgraph Isomorphism: nodes visited for hard instances of high density

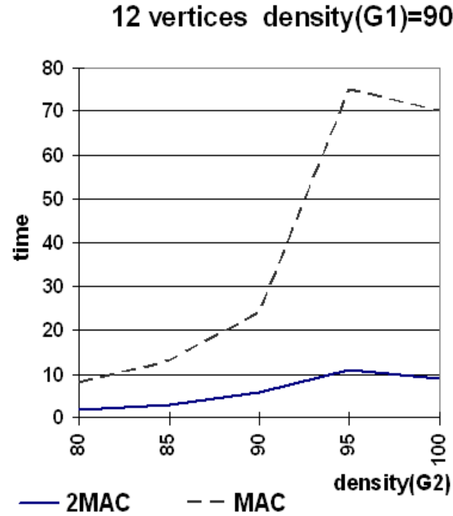

Figure 17: Subgraph Isomorphism: runtime for hard instances of high density

high density.

Observe that the CSP classes used for the experimental evaluation have an interesting common property. *The number of different constraint relations connecting the variables of CSPs of these classes is exactly 2.* In the Graph k-Coloring problem, two variables can be connected by either the universal constraint (all pairs of values are allowed) or by the inequality constraint. In the Subgraph Isomorphism problem, two variables are connected by either the inequality constraint or by the constraint corresponding to the edges of G_1.

This situation suggests that it would be interesting to compare 2MAC and MAC on a more general CSP class with the above property. For this purpose, we design a class of CSPs similar to the random CSPs proposed by Prosser [19]. A CSP is generated randomly given the number of variables n, the domain size d, the density p_1 (the probability that a given pair of two variables are constrained) and the tightness p_2 (the probability that a given pair of constrained variables is conflicting). The difference is that according to [18], a unique constraint relation must be generated for every pair of constrained variables. We generate just one constraint relation, say C. Then the constraint between two variables is either the universal one or

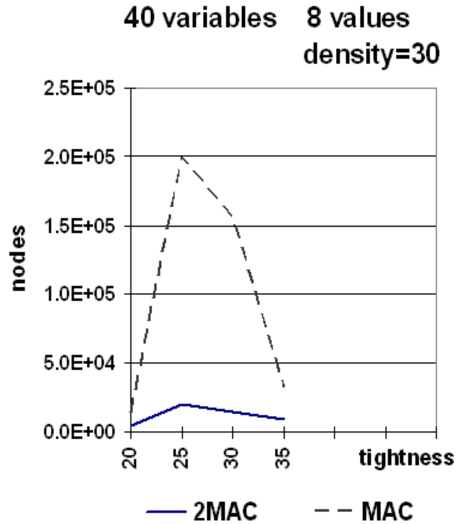

Figure 18: Random regular CSPs: nodes visited for hard problems of low density

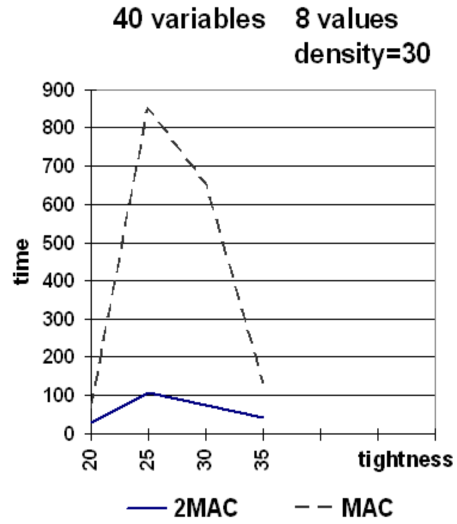

Figure 19: Random regular CSPs: runtime for hard problems of low density

C. The additional requirement of C is that there must be no equal pairs. Thus, p_2 is the probability that a given pair of non-equal values is conflicting.

We term the resulting class of CSPs *random regular CSPs*. We generate the sets of instances for the experiments by fixing n, d, and p_1, and varying p_2. Similarly to the previous two problems, we generate three sets of instances with the first transition region occurring for small, average and high values of p_2, respectively. The results of the experiments are presented in Figures 18- 23.

The results of experiments on random regular CSPs support the conjecture that 2MAC performs better than MAC on problems with a small number of constraint relations. In particular, 2MAC runs an order of magnitude faster than MAC over almost the whole range of the considered CSPs. The improvement measured for the number of nodes visited is even more impressive. For problems the last two sets of instances, 2MAC works two orders of magnitude better than MAC in the phase transition region.

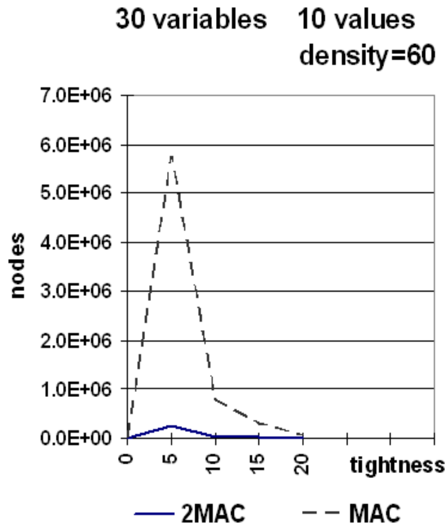

Figure 20: Random regular CSPs: nodes visited for hard problems of medium density

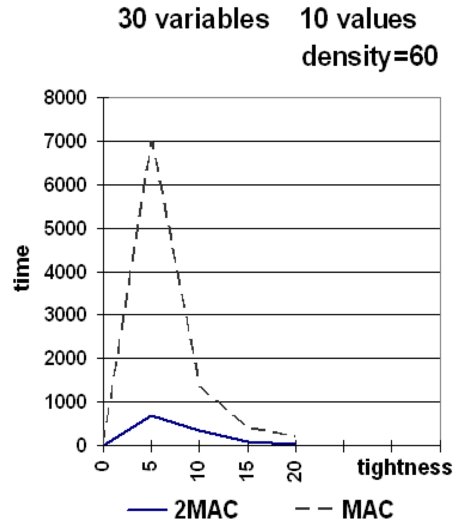

Figure 21: Random regular CSPs: runtime for hard problems of medium density

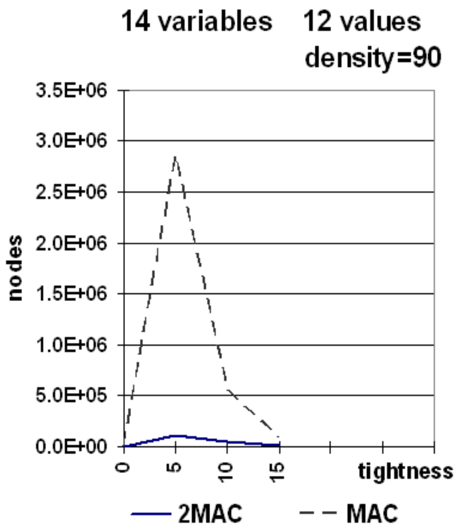

Figure 22: Random regular CSPs: nodes visited for hard problems of high density

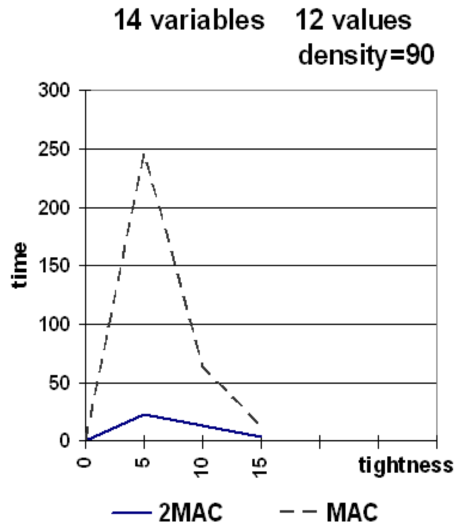

Figure 23: Random regular CSPs: runtime for hard problems of high density

5 Conclusion

In this paper we studied aspects of correlation between worst-case complexity of backtrack algorithms and their practical efficiency. The contributions of the paper are twofold. First, we developed a novel algorithmic scheme based on assigning a variable with two values rather than one, thus achieving a complexity of $O(\lceil d/2 \rceil^n)$. Based on this scheme we developed two algorithms, which we termed 2FC and 2MAC. The 2MAC algorithm is shown empirically to be an order of magnitude better than MAC on a number of CSP classes. This successful performance indicates that a "purely theoretical" approach can be utilized to obtain a useful "practical" algorithm.

The other contribution of the paper is the providing of new insight into existing algorithmic schemes. We proved that FC-CBJ combined with the well-known FF heuristic [12] has a reduced worst-case complexity of $O^*((d-1)^n)$. This statement is quite surprising because both FC-CBJ and FF were designed for practical purposes only with no intention to reduce worst-case complexity. Thus, it is indicated that a particular combination of heuristic methods can be useful from a theoretical point of view.

There are three main directions for further developing the methods presented in the paper: improving the proposed algorithms, applying them to other search problems, and theoretical investigation of the proposed approaches. These three directions are discussed below.

1. The most promising strategy for improving the proposed algorithms seems to be to combine 2FC or 2MAC with a method for replacing pairs of variables by single variables. To understand this strategy, consider a CSP in which there are two variables, say v_1 and v_2, that have more than 90 % of pairs of values forbidden. In this situation, it seems worthwhile to replace v_1 and v_2 by a single variable v with the values corresponding to the allowed pairs of values of v_1 and v_2 (of course, with appropriate updating of constraints with other variables). This replacement could work well when combined with 2FC and 2MAC, because these algorithms add conflicts between unassigned variables during their execution. As a result, some pairs of variables can become very constrained. Replacing such pairs of variables according to the strategy shown above, could reduce the search space.

2. The proposed algorithms could be applied to other search problems such as graph coloring problems. The potential applicability is indicated by good performance demonstrated by 2MAC on Graph k-

Coloring problems. For this purpose, algorithms proposed in the paper must be adopted to the optimization version of the problem.

Another possible application of the proposed algorithms is search for robust solutions [13]. Note that 2-CSPs returned by 2FC and 2MAC potentially have many solutions that can be generated efficiently. Therefore a 2-CSP could be used as a representation of a robust solutions. Accordingly, 2FC and 2MAC could be used as algorithms for computing of robust solutions.

3. From a theoretical point of view, the proposed methods could be applied to reducing the worst-case complexity of algorithms that solve CSPs. Currently, the fastest CSP algorithm takes about $O^*((0.45{*}d)^n)$ [7]. The complexity of 2FC or 2MAC is $O^*((0.5 * d)^n)$ but this bound is not tight because it does not take into account additional pruning imposed by the algorithm. To increase the pruning effect, 2FC or 2MAC could be transformed into 2MAC-CBJ and combined with some ordering heuristic. A particular combination of these methods could improve the best current complexity bound.

   Theoretical investigations could also help to develop new ordering heuristics that are useful for real-world CSP instances. To this purpose a deeper study of the complexity of FC-CBJ is of particular interest. To see this, consider again the example in Section 3.2. that demonstrated that for any n there is d such that FC-CBJ has a worst-case complexity of $\Omega((d^n)$ for CSPs with n variables and maximum domain size d. We conjecture that if d is small with respect to n, the complexity of FC-CBJ is smaller than $O(d^n)$. In particular, we suggest that the complexity of FC-CBJ can be expressed in the form $O(d^{n/f(d)})$, where $f(d) = 1$ if d is comparable with n and greater than 1 otherwise. Studying the properties of $f(d)$ and understanding how an ordering heuristic like FF influence these properties could help to design new efficient ordering heuristics.

References

[1] F. Bacchus and Y. Teh. Making forward chaining relevant. In *AIPS 98*, pages 54–61, 1998.

[2] M. Caramia and P. Dell'Olmo. Constraint propagation in graph coloring. *Journal of Heuristics*, 8:83–107, 2002.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5:394–397, 1962.

[4] R. Debruyne and C. Bessierre. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.

[5] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[6] R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136:147–188, 2002.

[7] D. Eppstein. Improved algorithms for 3-coloring, 3-edge coloring and constraint satisfaction. In *SODA-2001*, pages 329–337, 2001.

[8] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *CP2001*, pages 93–108. Springer, November 2001.

[9] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *CP2001*, pages 93–108. Springer, November 2001.

[10] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95*, pages 572–578, Montreal, Canada, 1995.

[11] I. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics. In *CP-96*, pages 179–193, 1996.

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

[13] E. Hebrard, B. Hnich, and T. Walsh. Robust solutions for constraint satisfaction and optimization. In *ECAI 2004*, pages 186–190, 2004.

[14] P. Jeavons, D. Cohen, and M. Cooper. Constraints, consistency, and closure. *Artificial Intelligence*, 101:251–265, 1998.

[15] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In Chris Mellish, editor, *IJCAI'95*, Montreal, 1995.

[16] J. Marques-Silva and K. Sakallah. Grasp : a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided design*, pages 220–227, 1996.

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

[18] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[19] P. Prosser. An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[20] C.-G. Quimper, A. Lopez-Ortiz, P. vanBeek, and A. Golynski. Improved algorithms for the global cardinality constraint. In *Principles and Practice of Constraint Programming-CP2004*, pages 542–556, Toronto, Canada, sep 2004. Springer.

[21] I. Razgon and A. Meisels. A CSP search algorithm with reduced branching factor. In *Proceedings of CSCLP 2005*, pages 13–27, 2005.

[22] J.-C. Regin. A filtering algorithm for constraints of difference in csps. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 362–367. American Association for Artificial Intelligence, 1994.

[23] R.Wallace. Analysis of heuristic synergies. In *CSCLP 2005*, pages 1–13, 2005.

[24] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *PPCP'94*, pages 10–20, 1994.

[25] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[26] Edward P. K. Tsang. No more "partial" and "full looking ahead". *Artificial Intelligence*, 98(1-2):351–361, 1998.

[27] G. Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial Optimization: "Eureka, you shrink", LNCS 2570*, pages 185–207, 2003.