Domain-Heuristics for Arc-Consistency Algorithms

M.R.C. van Dongen

CS Department UCC/Cork Constraint Computation Centre,* Western Road, Cork, Ireland,
dongen@cs.ucc.ie

Abstract. Arc-consistency algorithms are widely used to prune the search-space of Constraint Satisfaction Problems (CSPs). They use *support-checks* (also known as consistency-checks) to find out about the properties of CSPs. They use *arc-heuristics* to select the next constraint and *domain-heuristics* to select the next values for their next support-check. We will investigate the effects of domain-heuristics by studying the average time-complexity of two arc-consistency algorithms which use different domain-heuristics. We will assume that there are only two variables. The first algorithm, called $\mathcal{L}$, uses a lexicographical heuristic. The second algorithm, called $\mathcal{D}$, uses a heuristic based on the notion of a *double-support check*. We will discuss the consequences of our simplification about the number of variables in the CSP and we will carry out a case-study for the case where the domain-sizes of the variables is two. We will present relatively simple formulae for the *exact* average time-complexity of both algorithms as well as simple bounds. As a and b become large $\mathcal{L}$ will require about $2a + 2b - 2\log(a) - 0.665492$ checks on average, where a and b are the domain-sizes and $\log(\cdot)$ is the base-2 logarithm. $\mathcal{D}$ requires an average number of support-checks which is below $2\max(a, b) + 2$ if $a + b \geq 14$. Our results demonstrate that $\mathcal{D}$ is the superior algorithm. Finally, we will provide the result that on average $\mathcal{D}$ requires strictly fewer than two checks more than any other algorithm if $a + b \geq 14$.

1 Introduction

Arc-consistency algorithms are widely used to prune the search-space of Constraint Satisfaction Problems (CSPs). Arc-consistency algorithms require *support-checks* (also known as consistency-checks in the constraint literature) to find out about the properties of CSPs. They use *arc-heuristics* and *domain-heuristics* to select their next support-check. Arc-heuristics operate at *arc-level* and selects the constraint that will be used for the next check. Domain-heuristics operate at *domain-level*. Given a constraint, they decide which values will be used for the next check. Certain kinds of arc-consistency algorithms use heuristics which are—in essence—a combination of arc-heuristics and domain-heuristics.

We will investigate the effect of domain-heuristics by studying the average time-complexity of two arc-consistency algorithms which use different domain-heuristics. We will assume that there are only two variables. The first algorithm, called $\mathcal{L}$, uses a lexicographical heuristic. The second algorithm, called $\mathcal{D}$, uses a heuristic based on the

* The Cork Constraint Computation Centre is is supported by Science Foundation Ireland.

notion of a *double-support check*. Experimental evidence already exists to suggest that this *double-support heuristic* is efficient.

We will define $\mathcal{L}$ and $\mathcal{D}$ and present a detailed case-study for the case where the size of the domains of the variables is two. We will show that for the case-study $\mathcal{D}$ is superior on average.

We will present relatively simple *exact* formulae for the average time-complexity of both algorithms as well as simple bounds. As a and b become large $\mathcal{L}$ will require about $2a + 2b - 2\log(a) - 0.665492$ checks on average, where a and b are the domain-sizes and $\log(\cdot)$ is the base-2 logarithm. $\mathcal{D}$ will require an average number of support-checks which below $2\max(a, b) + 2$ if $a + b \geq 14$. Therefore, $\mathcal{D}$ is the superior algorithm. Finally, we will provide an "optimality" result that on average $\mathcal{D}$ requires strictly fewer than two checks more than any other algorithm if $a + b \geq 14$.

As part of our analysis we will discuss the consequences of our simplifications about the number of variables in the CSP.

The relevance of this work is that the double-support heuristic can be incorporated into any existing arc-consistency algorithm. Our optimality result is informative about the possibilities and limitations of domain-heuristics.

The remainder of this paper is organised as follows. In Section 2 we shall provide basic definitions and review constraint satisfaction. A formal definition of the lexicographical and double-support algorithms will be presented in Section 3. In that section we shall also carry out our case-study. In Section 4 we shall present our average time-complexity results and shall compare these results in Section 5. Our conclusions will be presented in Section 6.

2 Constraint Satisfaction

2.1 Basic Definitions

A *Constraint Satisfaction Problem* (or CSP) is a tuple $(\, X, D, C\,)$, where X is a set containing the variables of the CSP, D is a function which maps each of the variables to its domain, and C is a set containing the constraints of the CSP.

Let $(\, X, D, C\,)$ be a CSP. For the purpose of this paper we will assume that $X = \{\, \alpha, \beta \,\}$, that $\emptyset \subset D(\alpha) \subseteq \{\, 1, \ldots, a \,\}$, that $\emptyset \subset D(\beta) \subseteq \{\, 1, \ldots, b \,\}$, and that $C = \{\, M \,\}$, where M is a constraint between α and β. We shall discuss the consequences of our simplifications in Section 2.3.

A constraint M between α and β is an a by b zero-one matrix, i.e. a matrix with a rows and b columns whose entries are either zero or one. A tuple $(\, i, j\,)$ in the Cartesian product of the domains of α and β is said to *satisfy* a constraint M if $M_{ij} = 1$. Here M_{ij} is the j-th column of the i-th row of M. A value $i \in D(\alpha)$ is said to be *supported* by $j \in D(\beta)$ if $M_{ij} = 1$. Similarly, $j \in D(\beta)$ is said to be supported by $i \in D(\alpha)$ if $M_{ij} = 1$. M is called *arc-consistent* if for every $i \in D(\alpha)$ there is a $j \in D(\beta)$ which supports i and vice versa.

We will denote the set of all a by b zero-one matrices by $\mathbb{M}^{ab}$. We will call matrices, rows of matrices and columns of matrices *non-zero* if they contain more than zero ones, and call them *zero* otherwise.

The *row-support/column-support* of a matrix is the set containing the indices of its non-zero rows/columns. An *arc-consistency algorithm* removes all the unsupported values from the domains of the variables of a CSP until this is no longer possible. A *support-check* is a test to find the value of an entry of a matrix. We will write $M_{ij}^?$ for the support-check to find the value of M_{ij}. An arc-consistency algorithm has to carry out a support-check $M_{ij}^?$ to find out about the value of M_{ij}. The time-complexity of arc-consistency algorithms is expressed in the number of support-checks they require to find the supports of their arguments.

If we assume that support-checks are not duplicated then at most ab support-checks are needed by any arc-consistency algorithm for any a by b matrix. For a zero a by b matrix each of these ab checks is required. The worst-case time-complexity is therefore exactly ab for any arc-consistency algorithm. In this paper we shall be concerned with the average time-complexity of arc-consistency algorithms.

If $\mathcal{A}$ is an arc-consistency algorithm and M an a by b matrix, then we will write $\mathrm{checks}_{\mathcal{A}}(M)$ for the number of support-checks required by $\mathcal{A}$ to compute the row and column-support of M.

Let $\mathcal{A}$ be an arc-consistency algorithm. The *average time-complexity* of $\mathcal{A}$ *over* $\mathbb{M}^{ab}$ is the function $\mathrm{avg}_{\mathcal{A}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$, where

$$\mathrm{avg}_{\mathcal{A}}(a, b) = \sum_{M \in \mathbb{M}^{ab}} \mathrm{checks}_{\mathcal{A}}(M)/2^{ab}.$$

Note that it is implicit in our definition of average time-complexity that it is just as likely for a check to succeed as it is for it to fail. $\mathcal{A}$ is called *non-repetitive* if it does not repeat support-checks.

A support-check $M_{ij}^?$ is said to *succeed* if $M_{ij} = 1$ and said to *fail* otherwise. If a support-check succeeds it is called *successful* and *unsuccessful* otherwise.

Let a and b be positive integers, let M be an a by b zero-one matrix, and let $\mathcal{A}$ be an arc-consistency algorithm. The *trace* of M with respect to $\mathcal{A}$ is the sequence

$$(\, i_1, j_1, M_{i_1 j_1} \,), (\, i_2, j_2, M_{i_2 j_2} \,), \ldots, (\, i_l, j_l, M_{i_l j_l} \,), \tag{1}$$

where $l = \mathrm{checks}_{\mathcal{A}}(M)$ and $M_{i_k j_k}^?$ is the k-th support-check carried out by $\mathcal{A}$, for $1 \leq k \leq l$. The *length* of the trace in Equation (1) is defined as l. Its k*-th member* is defined by $(\, i_k, j_k, M_{i_k j_k} \,)$, for $1 \leq k \leq l$.

An interesting property of traces of non-repetitive algorithms is the one formulated as the following theorem which is easy to prove.

Theorem 1 (Trace Theorem). *Let $\mathcal{A}$ be a non-repetitive arc-consistency algorithm, let M be an a by b zero-one matrix, let t be the trace of M with respect to $\mathcal{A}$, and let l be the length of t. There are 2^{ab-l} members of $\mathbb{M}^{ab}$ whose traces with respect to $\mathcal{A}$ are equal to t.*

The theorem will turn out to be convenient later on because it will allow us to determine the "savings" of traces of non-repetitive arc-consistency algorithms without too much effort.

$M_{ij}^?$ is called a *single-support check* if, just before the check was carried out, the row-support status of i was known and the column-support status of j was unknown, or vice versa. A successful single-support check $M^?_{ij}$ leads to new knowledge about one thing. Either it leads to the knowledge that i is in the row-support of M where this was not known before the check was carried out, or it leads to the knowledge that j is in the column-support of M where this was not known before the check was carried out. $M^?_{ij}$ is called a *double-support check* if, just before the check was carried out, both the row-support status of of i and the column-support status of j were unknown. A successful double-support check $M^?_{ij}$ leads to new knowledge about *two* things. It leads to the knowledge that i is in the row-support of M and that j is in the column-support of M where neither of these facts were known to be true just before the check was carried out. A domain-heuristic is called a *double-support heuristic* if it prefers double-support checks to other checks.

On average it is just as likely that a double-support check will succeed as it is that a single-support check will succeed—in both cases one out of two checks will succeed on average. The potential payoff of a double-support check is twice as large that of a single-support check. This is our first indication that at domain-level arc-consistency algorithms should prefer double-support checks to single-support checks.

Our second indication that arc-consistency algorithms should prefer double-support checks is the insight that in order to minimise the total number of support-checks it is a necessary condition to maximise the number of successful double-support checks.

In the following section we shall point out a third indication—more compelling than the previous two—that arc-consistency algorithms should prefer double-support checks to single-support checks. Our average time-complexity results will demonstrate that the double-support heuristic is superior to lexicographical heuristic which is usually used.

2.2 Related Literature

In 1977, Mackworth presented an arc-consistency algorithms called AC-3 [5]. Together with Freuder he presented a lower bound of $\mathbf{\Omega}(ed^2)$ and an upper bound of $\mathbf{O}(ed^3)$ for the worst-case time-complexity [6]. Here, e is the number of constraints and d is the maximum domain-size.

AC-3, as presented by Mackworth, is not an algorithm as such; it is a *class* of algorithms which have certain data-structures in common and treat them similarly. The most prominent data-structure used by AC-3 is a *queue* which initially contains each of the pairs $(\,\alpha, \beta\,)$ and $(\,\beta, \alpha\,)$ for which there exists a constraint between α and β. The basic machinery of AC-3 is such that *any* tuple can be removed from the queue. For a "real" implementation this means that heuristics determine the choice of the tuple that was removed from the queue. By selecting a member from the queue, these heuristics determine the constraint that will be used for the next support-checks. Such heuristics will be called *arc-heuristics*.

Not only are there arc-heuristics for AC-3, but also there are heuristics which, given a constraint, select the values in the domains of the variables that will be used for the next support-check. Such heuristics we will call *domain-heuristics*.

Experimental results from Wallace and Freuder clearly indicate that arc-heuristics influence the average performance of arc-consistency algorithms [10]. Gent *et al* have made similar observations [4].

Bessière, Freuder, and Régin present another class of arc-consistency algorithms called AC-7 [1]. AC-7 is an instance of the AC-INFERENCE schema, where support-checks are saved by making inference. In the case of AC-7 inference is made at domain-level, where it is exploited that $M_{ij} = M_{ji}^T$, where $\cdot^T$ denotes transposition. AC-7 has an optimal upper bound of $\mathbf{O}(ed^2)$ for its worst-case time-complexity and has been reported to behave well on average.

In their paper, Bessière, Freuder, and Régin present experimental results that the AC-7 approach is superior to the AC-3 approach. They present results of applications of MAC-3 and MAC-7 to real-world problems. Here MAC-i is a backtracking algorithm which uses AC-i to maintain arc-consistency during search [7].

In [9] van Dongen and Bowen present results from an experimental comparison between AC-7 and AC-3_b which is a cross-breed between Mackworth's AC-3 and Gaschnig's DEE [3]. At the domain-level AC-3_b uses a double-support heuristic. In their setting, AC-7 was equipped with a lexicographical arc and domain-heuristic. AC-3_b has the same worst-case time-complexity as AC-3. In van Dongen and Bowen's setting it turned out that AC-3_b was more efficient than AC-7 for the majority of their $30,420$ random problems. Also AC-3_b was more efficient on average. These are surprising results because AC-3_b—unlike AC-7—has to repeat support-checks because it cannot remember them. The results are an indication that domain-heuristics can improve arc-consistency algorithms.

2.3 The General Problem

In this section we shall discuss the reasons for, and the consequences of, our decision to study only two-variable CSPs.

One problem with our choice is that we have eliminated the effects that arc-heuristics have on arc-consistency algorithms. Wallace and Freuder, and Gent *et al* showed that arc-heuristics have effects on the performance of arc-consistency algorithms [10, 4]. Our study does not properly take the effects of arc-heuristics into account. However, later in this section we will argue that a double-support heuristic should be used at domain-level.

Another problem with our simplification is that we cannot properly extrapolate average results for two-variable CSPs to the case where arc-consistency algorithms are used as part of MAC--algorithms. For example, in the case of a two-variable CSP, on average about one out of every two support-checks will succeed. This is not true in MAC--search because most time is spent at the leaves of the search-tree and most support-checks in that region will fail. A solution would be to refine our analysis to the case where different ratios of support-checks succeed.

We justify our decision to study two-variable CSPs by two reasons. Our first reason is that at the moment the general problem is too complicated. We study a simpler problem hoping that it will provide insight in the effects of domain-heuristics in general and the successfulness of the double-support heuristic in particular.

Our second reason to justify our decision to study two-variable CSPs is that some arc-consistency algorithms find support for the values in the domains of two variables at the same time. For such algorithms a good domain-heuristic is important and it can be found by studying two-variable CSPs only.

3 Two Arc-Consistency Algorithms

3.1 Introduction

In this section we shall introduce two arc-consistency algorithms and present a detailed case-study to compare the average time-complexity of these algorithms for the case where the domain-sizes of both variables is two. The two algorithms differ in their domain-heuristic. The algorithms under consideration are a *lexicographical algorithm* and a *double-support algorithm*. The lexicographical algorithm will be called $\mathcal{L}$. The double-support algorithm will be called $\mathcal{D}$. We shall see that for the problem under consideration $\mathcal{D}$ outperforms $\mathcal{L}$.

3.2 The Lexicographical Algorithm $\mathcal{L}$

In this section we shall define the *lexicographical arc-consistency algorithm* called $\mathcal{L}$ and discuss its application to two by two matrices. We shall first define $\mathcal{L}$ and then discuss the application.

$\mathcal{L}$ does not repeat support-checks. $\mathcal{L}$ first tries to establish its row-support. It does this for each row in the lexicographical order on the rows. When it seeks support for row i it tries to find the lexicographical smallest column which supports i. After $\mathcal{L}$ has computed its row-support, it tries to find support for those columns whose support-status is not yet known. It does this in the lexicographical order on the columns. When $\mathcal{L}$ tries to find support for a column j, it tries to find it with the lexicographically smallest row that was not yet known to support j.

Figure 1 is a graphical representation of all traces with respect to $\mathcal{L}$. Each different path from the root to a leaf corresponds to a different trace with respect to $\mathcal{L}$. Each trace of length l is represented in the tree by some unique path that connects the root and some leaf via $l - 1$ internal nodes. The root of the tree is an artificial 0-th member of the traces. The nodes/leaves at distance l from the root correspond to the l-th members of the traces, for $0 \leq l \leq ab = 4$.

Nodes in the tree are decision points. They represent the support-checks which are carried out by $\mathcal{L}$. A branch of a node n that goes straight up represents the fact that a support-check, say $M^?_{ij}$, is successful. A branch to the right of that same node n represents the fact that the same $M^?_{ij}$ is unsuccessful. The successful and unsuccessful support-checks are also represented at node-level. The i-th row of the j-th column of a node does not contain a number if the check $M^?_{ij}$ has not been carried out. Otherwise, it contains the number M_{ij}. It is only by studying the nodes that it can be found out which support-checks have been carried out so far.

Remember that we denote the number of rows by a and the number of columns by b. Note that there are $2^{ab} = 2^4 = 16$ different two by two zero-one matrices, and that traces of different matrices with respect to $\mathcal{L}$ can be the same. To determine the average time-complexity of $\mathcal{L}$ we have to add the lengths of the traces of each of the matrices and divide the result by 2^{ab}. Alternatively, we can compute the average number of support-checks if we subtract from ab the sum of the *average savings* of each of the matrices. Here, the *savings* of a matrix M are given by $ab - l$ and its *average savings* are given by $(ab - l)/2^{ab}$, where l is the length of the trace of M with respect to $\mathcal{L}$. Similarly, we define the *(average) savings* of a trace to be the sum of the (average) savings of all the matrices that have that trace.

It is interesting to notice that all traces of $\mathcal{L}$ have a length of at least three. Apparently, $\mathcal{L}$ is not able to determine its support in fewer than three support-checks—not even if a matrix does not contain any zero at all. It is not difficult to see that $\mathcal{L}$ will always require at least $a + b - 1$ support-checks regardless of the values of a and b.

$\mathcal{L}$ could only have terminated with two checks had both these checks been successful. If we focus on strategy $\mathcal{L}$ uses for its second support-check for the case where its first support-check was successful we shall find the reason why it cannot accomplish its task in fewer than three checks. After $\mathcal{L}$ has successfully carried out its first check $M^?_{11}$ it needs to learn only *two* things. It needs to know if 2 is in the row-support *and* it needs to know if 2 is in the column-support. The next check of $\mathcal{L}$ is $M^?_{21}$. Unfortunately, this check can only be used to learn *one* thing. *Regardless of whether the check $M^?_{12}$ succeeds or fails*, another check *has* to be carried out.

If we consider the case where the check $M^?_{22}$ was carried out as the second support-check we shall find a more efficient way of establishing the support. The check $M^?_{22}$ is a double-support check. It offers the potential of learning about *two* new things. If the check is successful then it offers the knowledge that 2 is in the row-support *and* that 2 is in the column-support. Since this was all that had to be found out the check $M^?_{22}$ offers the potential of termination after two support-checks. What is more, one out of every two such checks will succeed. Only if the check $M^?_{22}$ fails do more checks have to be carried out. Had the check $M^?_{22}$ been used as the second support-check, checks could have been saved on average.

Remember that the same trace in the tree can correspond to different matrices. The Trace Theorem states that if l is the length of a trace then there are exactly 2^{ab-l} matrices which have the same trace. The shortest traces of $\mathcal{L}$ are of length $l_1 = 3$. $\mathcal{L}$ finds exactly $s_1 = 3$ traces whose lengths are l_1. The remaining l_2 traces all have length $l_2 = ab$. Therefore, $\mathcal{L}$ saves $(s_1 \times (ab - l_1) \times 2^{ab-l_1} + s_2 \times (ab - l_2) \times 2^{ab-l_2})/2^{ab} = (3 \times (4 - 3) \times 2^{4-3} + 0)/2^4 = 3 \times 1 \times 2^1/2^4 = 3/8$ support-checks on average. $\mathcal{L}$ therefore requires an average number of support-checks of $ab - \frac{3}{8} = 4 - \frac{3}{8} = 3\frac{5}{8}$. In the following section we shall see that better strategies than $\mathcal{L}$'s exist.

3.3 The Double-Support Algorithm $\mathcal{D}$

In this section we shall introduce a second arc-consistency algorithm and analyse its average time-complexity for the special case where the number of rows a and the number of columns b are both two. The algorithm will be called $\mathcal{D}$. It uses a double-support heuristic as its domain-heuristic.

$\mathcal{D}$'s strategy is a bit more complicated than $\mathcal{L}$'s. It will use double-support checks to find support for its rows in the lexicographical order on the rows. It does this by finding for every row the lexicographically smallest column whose support-status is not yet known. When there are no more double-support checks left, $\mathcal{D}$ will use single-support

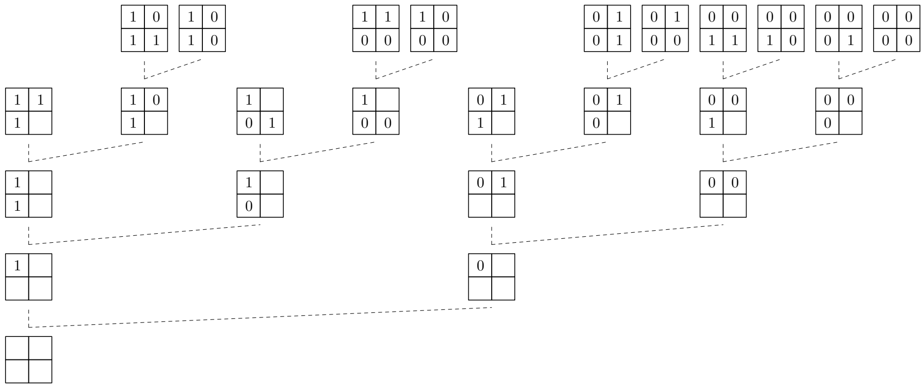

Fig. 1. Traces of $\mathcal{L}$. Total number of support-checks is 58

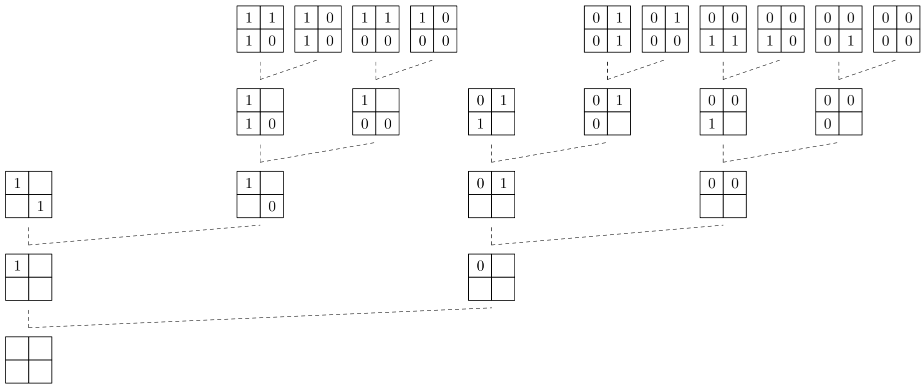

Fig. 2. Traces of $\mathcal{D}$. Total number of support-checks is 54

checks to find support for those rows whose support-status is not yet known and then find support for those columns whose support status is still not yet known. When it seeks support for a row/column, it tries to find it with the lexicographically smallest column/row that is not yet known to support that row/column.

We have depicted the traces of $\mathcal{D}$ in Figure 2. It may not be immediately obvious, but $\mathcal{D}$ is more efficient than $\mathcal{L}$. The reason for this is as follows. There are two traces whose length is shorter than $ab = 4$. There is $s_1 = 1$ trace whose length is $l_1 = 2$ and there is $s_2 = 1$ trace whose length is $l_2 = 3$. The remaining s_3 traces each have a length of $l_3 = ab$. Using the Trace Theorem we can use these findings to determine the number of support-checks that are saved on average. The average number of savings of $\mathcal{D}$ are given by $(s_1 \times (ab - l_1) \times 2^{ab-l_1} + s_2 \times (ab - l_2) \times 2^{ab-l_2} + s_3 \times (ab - l_3) \times 2^{ab-l_3})/2^{ab} = (2 \times 2^2 + 1 \times 2^1 + 0)/2^4 = 5/8$. This saves $1/4$ checks more on average than $\mathcal{L}$.

It is important to observe that l_1 has a length of only two and that it is the result of a sequence of two successful double-support checks. It is this trace which contributed the most to the savings. As a matter of fact, this trace by itself saved more than the *total* savings of $\mathcal{L}$.

The strategy used by $\mathcal{D}$ to prefer double-support checks to single-support checks leads to shorter traces. We can use the Trace Theorem to find that that the average savings of a trace are $(ab - l)2^{-l}$, where l is length of the trace. The double-support algorithm was able to produce a trace that was smaller than any of those produced by the lexicographical algorithm. To find this trace had a big impact on the total savings of $\mathcal{D}$. $\mathcal{D}$ was only able to find the short trace because it was the result of a sequence of successful double-support checks and its heuristic forces it to use as many double-support checks as it can. Traces which contain many successful double-support checks contribute much to the total average savings. This is our third and last indication that arc-consistency algorithms should prefer double-support checks to single-support checks.

3.4 A First Comparison of $\mathcal{L}$ and $\mathcal{D}$

In this section we have studied the average time-complexity of the lexicographical algorithm $\mathcal{L}$ and the double-support algorithm $\mathcal{D}$ for the case where the size of the domains is two. We have found that for the problem under consideration $\mathcal{D}$ was more efficient on average than $\mathcal{L}$.

4 Average Time-Complexity Results

In this section we shall present average time-complexity results for $\mathcal{L}$ and $\mathcal{D}$. The reader is referred to [8] for proof and further information.

Theorem 2 (Average Time Complexity of $\mathcal{L}$). *Let a and b be positive integers. The average time complexity of $\mathcal{L}$ over $\mathbb{M}^{ab}$ is* exactly $\operatorname{avg}_{\mathcal{L}}(a, b)$, *where*

$$\operatorname{avg}_{\mathcal{L}}(a, b) = a(2 - 2^{1-b}) + (1 - b)2^{1-a} + 2\sum_{c=2}^{b}(1 - 2^{-c})^a.$$

Following [2, Page 59] we obtain the following accurate estimate.

Corollary 1.
$$\mathrm{avg}_{\mathcal{L}}(a, b) \approx 2a + 2b - 2\log(a) - 0.665492.$$

Here, $\log(\cdot)$ is the base-2 logarithm. This estimate is already good for relatively small a and b.

We can also explain the results we have obtained for the bound in Corollay 1 in the following way. $\mathcal{L}$ has to establish support for each of its a rows and b columns except for the l columns which were found to support a row when $\mathcal{L}$ was establishing its row-support. Therefore, $\mathcal{L}$ requires about $2a + 2(b - l)$. To find l turns out to be easy because on average $a/2$ rows will be supported by the first column. From the remaining $a/2$ rows on average $a/4$ rows will be supported by the second column, ..., from the remaining 1 rows on average $1/2$ will find support with the $\log_2(a)$-th column, i.e. $l \approx \log_2(a)$. This informal reasoning demonstrates that on average $\mathcal{L}$ will require about $2a + 2b - 2\log_2(a)$ support-checks and this is almost exactly what we found in Corollay 1.

Theorem 3 (Average Time Complexity of $\mathcal{D}$). *Let a and b be non-negative integers. The average time complexity of $\mathcal{D}$ over $\mathbb{M}^{ab}$ is* exactly $\mathrm{avg}_{\mathcal{D}}(a, b)$*, where* $\mathrm{avg}_{\mathcal{D}}(a, 0) = \mathrm{avg}_{\mathcal{D}}(0, b) = 0$*, and*

$$\begin{aligned}\mathrm{avg}_{\mathcal{D}}(a, b) = 2 &+ (b-2)2^{1-a} + (a-2)2^{1-b} + 2^{2-a-b} \\ &- (a-1)2^{1-2b} + 2^{-b}\,\mathrm{avg}_{\mathcal{D}}(a-1, b) \\ &+ (1 - 2^{-b})\,\mathrm{avg}_{\mathcal{D}}(a-1, b-1)\end{aligned}$$

if $a \neq 0$ and $b \neq 0$.

Let a and b be positive integers such that $a + b \geq 14$. The following upper bound for $\mathrm{avg}_{\mathcal{D}}(a, b)$ is presented in [8]:

$$\begin{aligned}\mathrm{avg}_{\mathcal{D}}(a, b) < 2\max(a, b) + 2 & \\ &- (2\max(a, b) + \min(a, b))2^{-\min(a,b)} \\ &- (2\min(a, b) + 3\max(a, b))2^{-\max(a,b)}.\end{aligned}$$

An important result that follows from the upper bound for $\mathrm{avg}_{\mathcal{D}}(a, b)$ is that $\mathcal{D}$ is "almost optimal." The proof relies on the fact that *any* arc-consistency algorithm will require at least about $2\max(a, b)$ checks on average. Therefore, if $14 \leq a + b$, then $\mathrm{avg}_{\mathcal{D}}(a, b) - \mathrm{avg}_{\mathcal{A}}(a, b) < 2$ for any arc-consistency algorithm $\mathcal{A}$, hence the optimality claim.

5 Comparison of $\mathcal{L}$ and $\mathcal{D}$

In this section we shall briefly compare the average time-complexity of $\mathcal{L}$ and $\mathcal{D}$. We already observed in Section 3.2 that the minimum number of support-checks required by $\mathcal{L}$ is $a+b-1$. In Section 4 we have presented the bound $\mathrm{avg}_{\mathcal{D}}(a, b) < \max(a, b)+2$, provided that $a + b \geq 14$. If $a + b \geq 14$ and $a = b$ then the *minimum* number of support-checks required by $\mathcal{L}$ is almost the same as the *average* number of support-checks required by $\mathcal{D}$!

Our next observation sharpens the previous observation. It is the observation that on average $\mathcal{D}$ is a better algorithm than $\mathcal{L}$ because its upper bound is lower than the bound that we derived for $\mathcal{L}$. When a and b get large and are of the same magnitude then the difference is about $a + b - 2\log((a + b)/2)$ which is quite substantial.

Another important result is the observation that $\mathcal{D}$ is almost "optimal." If $a+b \geq 14$ and if $\mathcal{A}$ is any arc-consistency algorithm then $\mathrm{avg}_{\mathcal{D}}(a, b) - \mathrm{avg}_{\mathcal{A}}(a, b) < 2$. To the best of our knowledge, this is the first such result for arc-consistency algorithms.

6 Conclusions and Recommendations

In this paper we have studied two domain-heuristics for arc-consistency algorithms for the special case where there are two variables. We have defined two arc-consistency algorithms which differ only in the domain-heuristic they use. The first algorithm, called $\mathcal{L}$, uses a lexicographical heuristic. The second algorithm, called $\mathcal{D}$, uses a heuristic which gives preference to double-support checks. We have presented a detailed case-study of the algorithms $\mathcal{L}$ and $\mathcal{D}$ for the case where the size of the domains of the variables is two. Finally, we have presented average time-complexity results for $\mathcal{L}$ and $\mathcal{D}$.

We have defined the notion of a *trace* and have demonstrated the usefulness of this notion. In particular we have shown that the average savings of a trace are $(ab - l)2^{-l}$, where l is the length of the trace and a and b are the sizes of the domains of the variables.

Our average time-complexity results have demonstrated that $\mathcal{D}$ is more efficient than $\mathcal{A}$. Furthermore, we have demonstrated that $\mathcal{D}$ is almost optimal in a certain sense; if $a + b \geq 14$ then $\mathrm{avg}_{\mathcal{D}}(a, b) - \mathrm{avg}_{\mathcal{A}}(a, b) < 2$ for any arc-consistency algorithm $\mathcal{A}$.

The work that was started here should be continued in the form of a refinement of our analysis for the case where only every m-th out of every n-th support-check succeeds. This will provide an indication of the usefulness of the two heuristics under consideration when they are used as part of a MAC--algorithm. Furthermore, we believe that it should be worthwhile to tackle the more complicated problems where there are more than two variables in the CSP and where the constraints involve more than two variables. Finally, we believe that it should be interesting to implement an arc-consistency algorithm which does not repeat support-checks and which comes equipped with our double-support heuristic as its domain-heuristic.

References

1. C. Bessière, E.C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, volume 1, pages 592–598, Montréal, Québec, Canada, 1995. Morgan Kaufmann Publishers, Inc., San Mateo, California, USA.
2. P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Mellin transform asymptotics. Technical Report Research Report 2956, INRIA, 1996.

3. J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceeding of the Second Biennial Conference, Canadian Society for the Computational Studies of Intelligence*, pages 268–277, 1978.
4. I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'1997)*, pages 327–340. Springer, 1997.
5. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
6. A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–73, 1985.
7. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley & Sons, 1994.
8. M.R.C. van Dongen. *Constraints, Varieties, and Algorithms*. PhD thesis, Department of Computer Science, University College, Cork, Ireland, 2002.
9. M.R.C. van Dongen and J.A. Bowen. Improving arc-consistency algorithms with double-support checks. In *Proceedings of the Eleventh Irish Conference on Artificial Intelligence and Cognitive Science (AICS'2000)*, pages 140–149, 2000.
10. R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *AI/GI/VI '92*, pages 163–169, Vancouver, British Columbia, Canada, 1992.