A Theoretical Analysis of Domain-Heuristics for Arc-Consistency Algorithms

M.R.C. van Dongen (dongen@cs.ucc.ie)
Technical Report: TR0004
Department of Computer Science
University College Cork
College Road
Cork
Ireland

1 December 2000

Abstract

Arc-consistency algorithms are widely used to prune the search-space of constraint satisfaction problems (CSPs). Arc-consistency algorithms require *support-checks* to find out about the properties of CSPs. They use two kinds of heuristics to select their next support-check. The first kind operates at *arc-level* and selects the constraint that will be used for the next check. The second kind operates at *domain-level* and decides which values will be used for the next check.

It is our intention to investigate the effect of domain-heuristics by studying the average time-complexity of two arc-consistency algorithms which only differ in the domain-heuristics they use. We will assume that there are only two variables. We will discuss the consequences of this simplification. 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 present a detailed case study and present three good reasons why arc-consistency algorithms should give preference to double-support checks at domain-level.

For sufficiently large domain-sizes $\mathfrak{a}$ and $\mathfrak{b}$ our average time-complexity analysis provides the lower bound of approximately $2\mathfrak{a} + 2\mathfrak{b} + \mathbf{O}(1)$ for $\mathcal{L}$. We provide an upper bound for $\mathcal{D}$ which is strictly below $2\max(\mathfrak{a}, \mathfrak{b}) + 2$ if $\mathfrak{a} + \mathfrak{b} \geq 14$. We also derive the result that $\mathcal{D}$ spends strictly less than two checks more on average than *any* algorithm if $\mathfrak{a} + \mathfrak{b} \geq 14$. We believe that this is the first such result ever to have been reported.

The relevance of this work is that the double-support heuristic is promising and can be incorporated into any existing arc-consistency algorithm. Our result for the upper bound of $\mathcal{D}$ is informative about the possibilities and limitations of domain-heuristics. Finally, we think that our study is the first theoretical analysis of domain-heuristics for arc-consistency algorithms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Arc-consistency algorithms are widely used to prune the search-space of constraint satisfaction problems (CSPs). Arc-consistency algorithms require *support-checks* 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 notion of a *double-support check*. Empirical evidence already suggests that the double-support heuristic is efficient [van Dongen and Bowen, 2000].

We will define the algorithms $\mathcal{L}$ 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. Three reasons will be pointed out why arc-consistency algorithms should give preference to double-support checks at domain-level.

We will carry out an exact average time-complexity analysis for the case where the domains are not restricted to have a size of two. Our analysis will provide solid mathematical evidence that $\mathcal{D}$ is the better algorithm on average.

We will derive relatively simple exact formulae for the average time-complexity of both algorithms and derive simple expressions for their upper and lower bounds. To be more specific, we will demonstrate that $\mathcal{L}$ requires at least $1.9643785\mathsf{a} + 2\mathsf{b} + \mathbf{O}(1) + \mathbf{O}(\mathsf{b}2^{-\mathsf{a}}) + \mathbf{O}(\mathsf{a}2^{-\mathsf{b}})$ checks, where a and b are the sizes of the domains of the variables. We will also demonstrate that $\mathcal{D}$ requires a number of support-checks which is strictly less than $2\max(\mathsf{a},\mathsf{b}) + 2$ if $\mathsf{a} + \mathsf{b} \geq 14$. Our results demonstrate that $\mathcal{D}$ is the superior algorithm. Finally, we will provide the result that $\mathcal{D}$ requires strictly less on average than two checks more than any optimal algorithm if $\mathsf{a} + \mathsf{b} \geq 14$. This is the first such result ever to have been reported.

As part of our analysis we will compare the average time-complexity of the two algorithms under consideration and 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 report is organised as follows. In Chapter 2 we will provide basic definitions and review constraint satisfaction. A formal definition of the lexicographical and double-support algorithms will be presented in Chapter 3. In this chapter we shall also carry out our case-study for the case where the size of the domains is two. We will identify three reasons which, from an intuitive point of view, suggest that at domain-level arc-consistency algorithms should give preference to double-support checks. In Chapter 4 we shall carry out our average time-complexity analysis for the lexicographical algorithm. In Chapter 5 we shall do the same for the double-support algorithm. We shall compare the results of our average time-complexity analysis in Chapter 6. Our conclusions will be presented in Chapter 7.

To keep this paper completely self-contained, we have included an appendix which contains the mathematical background that is required for our time-complexity analysis.

Chapter 2

Constraint Satisfaction

This chapter provides our basic definitions and reviews constraint satisfaction. Its organisation is at follows. In Chapter 2.1 we shall provide our basic definitions. In Chapter 2.2 we shall review the related literature. As we already indicated, it is our intention to study arc-consistency algorithms for the case where there are only two variables in the CSP. In Chapter 2.3 we shall discuss the consequences of this simplification.

2.1 Basic Definitions

The constraint paradigm has been successfully applied for the representation and solution of many problems both inside and outside academia. 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.

We will only consider constraints between two variables at a time. Let (X, D, C) be a CSP, α and β two variables in X, $D(\alpha) = \{1, \ldots, a\}$ and $D(\beta) = \{1, \ldots, b\}$. A constraint between α and β restricts the values they are allowed to take simultaneously. In our setting, constraints are arrays/matrices and we will only consider constraints between two variables at a time. If M is the constraint between α and β then M 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. The tuple (i, j) in the Cartesian product of the domains of α and β is said to *satisfy* the constraint M iff $M_{ij} = 1$, where 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 $j \in D(\beta)$ iff $M_{ij} = 1$. Similarly, $j \in D(\beta)$ is said to be supported by $i \in D(\alpha)$ iff $M_{ij} = 1$.

Definition 1 (Arc-Consistency) *Let* (X, D, C) *be a* CSP. *The* CSP *is called* arc-consistent *iff* $(\forall x \in X)(|D(x)| \neq \emptyset)$ *and for each constraint, say* $M \in C$*, it holds that if* M *is between* α *and* β *then for every* $i \in D(\alpha)$ *there is a* $j \in D(\beta)$ *which supports* i *and vice versa.*

If (X, D, C) is a CSP then we will assume that it is such that $X = \{\alpha, \beta\}$ and $C = \{M\}$, where M is the constraint between α and β. Furthermore we will assume that $D(\alpha) = \{1, \ldots, a\} \neq \emptyset$ and $D(\beta) = \{1, \ldots, b\} \neq \emptyset$. In other words, we will only concern ourselves with CSPs where there are two variables, where the domains are non-empty, where there is

one constraint and where the constraint is between the two variables of the CSP. We shall discuss the consequences of our simplifications in Chapter 2.3.

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. Finally, we will assume that unless explicitly stated otherwise all matrices are a by b matrices.

Definition 2 (Row-Support) *The* row-support *of an* a *by* b *matrix* M *is the set*

$$\{\, i \in \{1, \ldots, a\} \mid (\exists j \in \{1, \ldots, b\})(M_{ij} = 1) \,\}.$$

The row-support of a matrix is the set containing the indices of its non-zero rows.

Definition 3 (Column-Support) *The* column-support *of an* a *by* b *matrix* M *is the set*

$$\{\, j \in \{1, \ldots, b\} \mid (\exists i \in \{1, \ldots, a\})(M_{ij} = 1) \,\}.$$

The column-support of a matrix is the set containing the indices of its non-zero columns.

Let M be an a by b matrix. We will say that a row i *supports* a column j if $M_{ij} = 1$. Similarly a column j is said to support a row i if $M_{ij} = 1$.

From now on we will also speak of the *support* of a matrix. By this we will mean the tuple (s_r, s_c), where s_r is the row-support and s_c is the column-support of that matrix.

An *arc-consistency algorithm* removes all the unsupported values from the domains of the variables of a CSP until this is no longer possible. For the case where there are two variables, arc-consistency algorithms compute the support of a matrix. A *support-check* is a test to find the value of an antry of a matrix. We will write $M^?_{ij}$ for the support-check to find the value of M_{ij}. An arc-consistency algorithms has to carry out a support-check, say $M^?_{ij}$, 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 support 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 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 are interested in the average time-complexity of arc-consistency algorithms.

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

Definition 4 (Average Time-Complexity) *Let* $\mathcal{A}$ *be an arc-consistency algorithm. The* average time-complexity *of* $\mathcal{A}$ over $\mathbb{M}^{ab}$ *is the function* $\text{avg}_{\mathcal{A}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$*, where*

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

Definition 5 (Stubborn Arc-Consistency Algorithm) *Let* $\mathcal{A}$ *be an arc-consistency algorithm, then* $\mathcal{A}$ *is called* stubborn *if it repeats 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.

Definition 6 (Trace) *Let* a *and* b *be positive integers,* M *an* a *by* b *zero-one matrix and* $\mathcal{A}$ *an arc-consistency algorithm. The* trace *of* M *w.r.t.* $\mathcal{A}$ *is the sequence of the form*

$$(i_1, j_1, M_{i_1 j_1}), (i_2, j_2, M_{i_2 j_2}), \ldots, (i_l, j_l, M_{i_l j_l}), \tag{2.1}$$

where $l = \text{checks}_{\mathcal{A}}(M)$ *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 (2.1) 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 a by b matrix has ab entries. Therefore, the lengths of the traces of non-stubborn algorithms are less than or equal to ab. An interesting property of traces of non-stubborn algorithms is the one formulated as the following theorem.

Theorem 1 (Trace Theorem) *Let* $\mathcal{A}$ *be a non-stubborn arc-consistency algorithm,* a *and* b *positive integers,* M *an* a *by* b *zero-one matrix and* t *the trace of* M *w.r.t.* $\mathcal{A}$*. If* l *is the length of* t *then the number of matrices whose trace is* t *is exactly* 2^{ab-l}*.*

Proof. Let the k-th member of t be $(i_k, j_k, M_{i_k j_k})$, for $1 \leq k \leq l$. We are looking for $|S|$, where

$$S = \{ M' \in \mathbb{M}^{ab} \mid (\forall k \in \{1, \ldots, l\})(M_{i_k j_k} = M'_{i_k j_k}) \}.$$

S contains exactly 2^{ab-l} members. □

The theorem will turn out to be convenient later on because it will allow us to determine the "savings" of a thread of an algorithm without too much effort.

2.2 Related Literature

In this section we shall review the related literature on arc-consistency algorithms.

Arc-consistency algorithms have been studied for quite some time. In 1977, Mackworth points out reasons why problems that are not arc-consistent are more difficult to solve with techniques based on backtracking and presents three arc-consistency algorithms called AC-1, AC-2 and AC-3 [Mackworth, 1977]. AC-3 is the most efficient of the three and in a joint paper with Freuder worst case time-complexity results for AC-3 are presented. The lower bound they present is $\Omega(ed^2)$ and their upper bound is $\mathbf{O}(ed^3)$, where e is the number of constraints and d is the maximum domain-size [Mackworth and Freuder, 1985]. Both bounds are linear in the number of constraints.

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 (α, β) and (β, α) 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 certain 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. In this paper, such heuristics will be called *arc-level* heuristics.

If (α, β) was the tuple which was removed from the queue then every value in the domain of α which is not supported by some value in the domain of β is removed from the domain of α. If values were removed from the domain of α then all pairs of the form (γ, α) are added to the queue for every constraint between γ and α in the CSP, except for the case where $\beta = \gamma$. The algorithm keeps on doing this until either the queue is empty in which case the CSP is arc-consistent or one of the domains becomes empty in which case support-checks can be saved because the CSP cannot be made arc-consistent.

Not only are there heuristics for AC-3 to remove members from the queue, but also are there heuristics which, given a constraint, select the values in the domains of the variables that will be used for the support-checks. Such heuristics we will call *domain-heuristics*.

Wallace and Freuder report on the effects of arc-level heuristics [Wallace and Freuder, 1992]. Wallace reports that the average time required by AC-3 is better than the average time required by AC-4 [Wallace, 1993]. AC-4 is an arc-consistency which has an optimal $\mathbf{O}(ed^2)$ worst case time-complexity [Mohr and Henderson, 1986].

The major drawback of AC-3 is that it cannot remember the support-checks it has already carried out and—as a consequence—repeats some of them. Bessière, Freuder and Régin present another class of arc-consistency algorithms called AC-7 [Bessière *et al.*, 1995]. 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.

The most prominent data-structures of AC-7 are a *deletion-stream* and a *seek-support-stream*. The purpose of the deletion-stream is to propagate the consequences of the removal of a value from the domain of one of the variables. The seek-support-stream contains tuples of the form $((\alpha, i), \beta)$, where α and β are variables and $i \in D(\alpha)$. A tuple $((\alpha, i), \beta)$ in the seek-support-stream represents the fact that support for the value $i \in D(\alpha)$ has to be found with some value in $D(\beta)$. Heuristics to select members from the seek-support-stream have effects on the number of support-checks that are required and the order in which they are carried out.

AC-7's heuristics for the selection of tuples from its seek-support-stream are a combination of arc-level and domain-heuristics. However, because AC-7 uses inference, not every removal of every tuple from the seek-support-stream will result in an actual support-check. Since a double-support heuristic is but a special example of a heuristic, the heuristics for a particular implementation of AC-7 could be such that the domain-level component would (partially) depend on a double-support heuristic.

In their paper, Bessière, Freuder and Régin present empirical 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, where MAC-i is a backtracking algorithm which uses AC-i to maintain arc-consistency during search [Sabin and Freuder, 1994]. Unfortunately it is not reported which members of the classes they use for their comparison, i.e. it is not

reported which heuristics they used for AC-3 and AC-7 and their experiments cannot be repeated to get the exact same results.

Van Dongen and Bowen present results of a comparison between AC-7 and an arc-consistency algorithm which is a cross-breed between Mackworth's AC-3 and Gaschnig's DEEB [van Dongen and Bowen, 2000; Gaschnig, 1978]. Their arc-consistency algorithm maintains a queue like AC-3 but processes its members slightly differently. If both (α, β) and (β, α) are in the queue then both pairs are treated as a unit so that they can be removed from the queue at the same time. If the unit is removed from the queue then it is tried to simultaneously find support for the values in the the domains of α and β. At this level a double-support heuristic is used to determine the support-checks. They used heuristics for AC-7 which selected the members of the seek-support-stream using a lexicographical rule. They compared applications of both algorithms to more than 30.000 random CSPs. The task of the algorithms was to compute the arc-consistent equivalent of the CSPs or decide that that was not possible. In their setting it turned out that AC-7 equipped with their lexicographical ordering heuristic performed worse than their own algorithm. It is important to point out that their results are even more interesting because their algorithm—like AC-3—has to repeat support-checks because it cannot remember them. The results are an indication that it is possible to use domain-heuristics to improve the performance of arc-consistency algorithms.

2.3 The General Problem

In this section we shall discuss the reasons for and the consequences of our decision to only study two-variable CSPs. Also we will make some general comments about the presentation of our algorithms further on in this paper.

One problem with our choice is that we have eliminated the effects that arc-level heuristics have on arc-consistency algorithms. Wallace and Freuder showed that arc-level heuristics have effects on performance [Wallace and Freuder, 1992]. Our study does not properly take the effects of arc-level 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 leafs 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 have studied a simper problem hoping that it would provide insight as to why the double-support heuristic was so successful in van Dongen and Bowen's setting [van Dongen and Bowen, 2000].

Our second reason to justify our decision to study two-variable CSPs is that we argue that at domain-level a double-support heuristic is a good choice and that it can be studied

independent of an arc-level heuristic. We assume that support-checks are not repeated.

Our reasoning is as follows. To compute an arc-consistent CSP we have to find out for each value in the domain of each of the variables if it is supported. We can only decide if a value, say $\mathtt{i} \in \mathsf{D}(\alpha)$, is inherently unsupportable if we find a constraint between α and another variable, say β, and spend checks for *each* of the values in $\mathsf{D}(\beta)$ that were not known to support $\mathtt{i}$. In other words, if $\mathtt{i}$ is inherently unsupportable then *any* domain-heuristic is a good choice. If $\mathtt{i}$ is inherently supportable then for each constraint between α and, say, β we should not only find a support as soon as possible but also find a support with a value in $\mathsf{D}(\beta)$ whose support-status was not yet known, i.e. we should spend double-support checks involving $\mathtt{i}$. After all, if a double-support check succeeds it will provide more information about which values are supported by the constraint. This information can be used for the purpose of making inference and for the purpose of "guiding" heuristics. In other words, regardless of $\mathtt{i}$'s inherent supportability a double-support heuristic is a good choice to complement any arc-level heuristic. Observe that our reasoning was independent of the choice of the arc-heuristic that was used. We can probably study the double-support heuristic by studying it for the case where the CSP is a two-variable CSP.

Chapter 3

Two Arc-Consistency Algorithms

In this chapter we shall introduce two arc-consistency algorithms and present a detailed case study where we shall compare the average time-complexity of the two algorithms for the case where the domain size of both variables is two. The two algorithms differ in the heuristic they use to select their support-checks. 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 $\mathcal{D}$. As part of our presentation we will point out three different reasons which, from an intuitive point of view, suggest that arc-consistency algorithms should give preference to double-support checks at domain-level. We shall see that for the problem under consideration $\mathcal{D}$ outperforms $\mathcal{L}$.

The remainder of this chapter is as follows. In Chapter 3.1 we shall define $\mathcal{L}$ and examine its average time-complexity for two by two matrices. In Chapter 3.2 we shall define $\mathcal{D}$ and examine its average time-complexity for two by two matrices. As part of the examination process we will point out three reasons which suggest that arc-consistency algorithms should give preference to double-support checks. Finally, in Chapter 3.3 we shall compare the two algorithms. Throughout this chapter we will denote the number of rows by $\mathfrak{a}$ and the number of columns by $\mathfrak{b}$.

3.1 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. We will not be concerned about the data-structures used in implementations of $\mathcal{L}$. Instead, it is our intention to present algorithms such that their essence becomes clear. In our presentation we use am ALGOL-ish pseudo-language which comes with a "forall $\nu \in \mathsf{S}$ do statements od" iteration-construct. The semantics of the construct are that for each member, say s, in S it assigns s to ν and carries out statements in S. The order in which the members of S are assigned to ν is the same as the lexicographical order on the members of S.

Definition 7 (Lexicographical Arc-Consistency Algorithm) *Let* $\mathfrak{a}$ *and* $\mathfrak{b}$ *be positive integers and* $\mathsf{M} \in \mathbb{M}^{\mathfrak{a}\mathfrak{b}}$. *The* lexicographical *arc-consistency algorithm is the algorithm* $\mathcal{L}$ *defined in Table 3.1.*

```
function 𝓛(M, a, b) do
    /* initialisation */
    row-support := ∅;
    column-support := ∅;
    forall i ∈ {1, ..., a} do
        forall j ∈ {1, ..., b} do
            C_ij := unknown;
        od;
    od;
    forall i ∈ {1, ..., a} do
        /* try to establish support for i */
        j := 0;
        while (j < b) and (i ∉ row-support) do
            /* find lexicographically smallest j that supports i */
            j := j + 1;
            C_ij := M_ij;
            if (C_ij = 1) then
                row-support := row-support ∪ {i};
                column-support := column-support ∪ {j};
            fi;
        od;
    od;
    forall j ∈ {1, ..., b} \ column-support do
        /* try to establish support for j */
        i := 0;
        while (i < a) and (j ∉ column-support) do
            /* find lexicographically smallest i that supports j */
            i := i + 1;
            if (C_ij = unknown) then
                if (M_ij = 1) then
                    column-support := column-support ∪ {j};
                fi;
            fi;
        od;
    od;
    return (row-support, column-support);
od;
```

Table 3.1: The Lexicographical Algorithm $\mathcal{L}$

$\mathcal{L}$ 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 a row, say 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, it tries to find it with the lexicographically smallest row that was not yet known to support i.

Example 1 ($\mathcal{L}$) *Let* $M \in \mathbb{M}^{33}$ *be the matrix whose first row is* $[\ 0\ \ 1\ \ 1\ ]$*, whose second row is* $[\ 0\ \ 0\ \ 0\ ]$*, and whose last row is* $[\ 1\ \ 1\ \ 0\ ]$*. In order to find the support of* M *the following support-checks are carried out by* $\mathcal{L}$ *in their order of appearance:* $M^?_{11}$, $M^?_{12}$, $M^?_{21}$, $M^?_{22}$, $M^?_{23}$, $M^?_{31}$, $M^?_{13}$. *The trace of* M *w.r.t.* $\mathcal{L}$ *is given by* $(1,1,0)$, $(1,2,1)$, $(2,1,0)$, $(2,2,0)$, $(2,3,0)$, $(3,1,1)$, $(1,3,1)$.

Figure (3.1) is a graphical representation of all traces w.r.t. $\mathcal{L}$. Each different path from the root to a leaf corresponds to a different trace w.r.t. $\mathcal{L}$. 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.

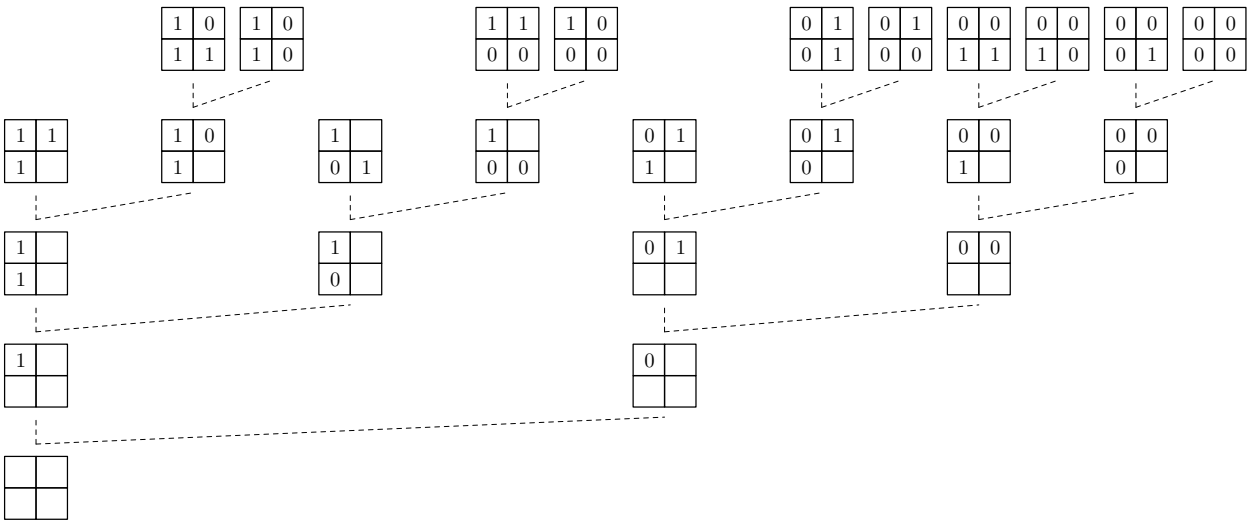

Figure 3.1: Traces of $\mathcal{L}$. Total Number of Support-Checks is $16 \times 4 - 6 \times 1 = 58$

The root of the tree is an artificial 0-th member of the traces. The nodes/leafs 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, say n, that goes straight up represents the fact that a support-check, say $M^?_{ij}$, was successful. A branch to the right of that same node n represents the fact that the same $M^?_{ij}$ was 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.

Example 2 (Trace) *The path in Figure (3.1) from the root of the tree to the second leaf from its right represents the trace* $t_1 := (1,1,0),(1,2,0)(2,1,0),(2,2,1)$. *The path from the root to the leftmost leaf corresponds to the trace* $t_2 := (1,1,1),(2,1,1),(2,2,1)$. *There is only one two by two zero-one matrix whose trace is* t_1. *There are two two by two zero-one matrices whose trace is given by* t_2.

Remember that we denote the number of rows by a and the number of columns by b. It is important to understand that there are $2^{ab} = 2^4 = 16$ different two by two zero-one matrices and that the traces of different matrices w.r.t. $\mathcal{L}$ 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 the matrices, where the savings of a matrix 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 the matrix w.r.t. $\mathcal{L}$.

It is interesting to notice that all traces of $\mathcal{L}$ have a length of at least three. Apparently $\mathcal{L}$ is not capable to determine its support in less 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.

$\mathcal{L}$ could only have terminated with two checks had both these checks been successful. If we focus on $\mathcal{L}$'s strategy 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 less 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. $\mathcal{L}$'s next check is $M^?_{21}$. Unfortunately, this check can only be used to learn *one* thing. *Regardless of whether the check* $M^?_{12}$ *succeeds of 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 $M^?_{22}$ 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 one 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. $\mathcal{L}$'s shortest traces 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. $\mathcal{L}$'s strategy therefore requires an average number of support-checks of $ab - \frac{3}{8} = 4 - \frac{3}{8} = 3\frac{5}{8}$. In the next section we shall see that better strategies than $\mathcal{L}$'s exist.

3.2 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 heuristic to select its support-checks based on the notion of a *double-support check*.

The organisation of the remainder of this section is as follows. First we shall define the notion of a double-support check and two other related notions. Next, we shall define $\mathcal{D}$ and analyse it for the two by two problem under consideration. As part of our analysis we will identify three reasons which, from an intuitive point of view, suggest that arc-consistency algorithms should prefer double-support checks to other checks.

Definition 8 (Zero-Support Check) *Let* M *be an* a *by* b *matrix. A support-check* $M^?_{ij}$ *is called a* zero-support check *if, just before the check was carried out, the row-support status of* i *and the column-support status of* j *were known.*

A zero-support check is a support-check from which nothing new can be learned about the support of a matrix. Good arc-consistency algorithms should therefore never carry out

zero-support checks.

Definition 9 (Single-Support Check) *Let* M *be an* $\mathtt{a}$ *by* $\mathtt{b}$ *matrix. A support-check* $\mathsf{M}^?_{\mathtt{ij}}$ *is called a* single-support check *if, just before the check was carried out, the row-support status of* $\mathtt{i}$ *was known and the column-support status of* $\mathtt{j}$ *was unknown or vice versa.*

A successful single-support check, say $\mathsf{M}^?_{\mathtt{ij}}$, leads to new knowledge about one thing. Either it leads to the knowledge that $\mathtt{i}$ 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 $\mathtt{j}$ is in the column-support of M where this was not known before the check was carried out.

Definition 10 (Double-Support Check) *Let* M *be an* $\mathtt{a}$ *by* $\mathtt{b}$ *matrix. A support-check* $\mathsf{M}^?_{\mathtt{ij}}$ *is called a* double-support check *if, just before the check was carried out, both the row-support status of of* $\mathtt{i}$ *and the column-support status of* $\mathtt{j}$ *were unknown.*

A successful double-support check, say $\mathsf{M}^?_{\mathtt{ij}}$, leads to new knowledge about two things. It leads to the knowledge that $\mathtt{i}$ is in the row-support of M and that $\mathtt{j}$ is in the column-support of M where none of these facts were known to be true just before the check was carried out.

Single-support checks, provided they are successful, lead to knowledge about one new thing at the price of one support-check. Double-support checks, provided they are successful, lead to knowledge about two new things at the price of one support-check. On average it is just as likely that a double-support check will succeed than 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 that of a single-support check. This is our first indication that arc-consistency algorithms should prefer double-support checks to single-support checks.

Our second indication that arc-consistency algorithms should prefer double-support checks to single-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.

Later in this section we will point out a third indication—more compelling than the previous two—that arc-consistency algorithms should prefer double-support checks over single-support checks.

Definition 11 (Double-Support Arc-Consistency Algorithm) *Let* $\mathtt{a}$ *and* $\mathtt{b}$ *be positive integers and* $\mathsf{M} \in \mathbb{M}^{\mathtt{ab}}$. *The* double-support *arc-consistency algorithm is the algorithm* $\mathcal{D}$ *defined in Table 3.2.*

$\mathcal{D}$'s strategy is a bit more complicated than $\mathcal{L}$'s. It will first try to 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 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.

```
function 𝒟(M, a, b) do
    /* initialisation */
    row-support := ∅;
    column-support := ∅;
    forall i ∈ {1, ..., a} do forall j ∈ {1, ..., b} do C_ij := unknown; od; od;
    forall i ∈ {1, ..., a} do
        /* try to establish support for i using double-support checks */
        j := 0;
        while (j < b) and (i ∉ row-support) do
            /* try to find lexicographically smallest j that supports i */
            j := j + 1;
            if (j ∉ column-support) then
                C_ij := M_ij;
                if (C_ij = 1) then
                    row-support := row-support ∪ {i};
                    column-support := column-support ∪ {j};
                fi;
            if;
        od;
    od;
    forall i ∈ {1, ..., a} \ row-support do
        /* try to establish support for i using single-support checks */
        j := 0;
        while (j < b) and (i ∉ row-support) do
            /* try to find lexicographically smallest j that supports i */
            j := j + 1;
            if (C_ij = unknown) then
                C_ij := M_ij;
                if (C_ij = 1) then
                    row-support := row-support ∪ {i};
                fi;
            fi;
        od;
    od;
    forall j ∈ {1, ..., b} \ column-support do
        /* try to establish support for j using single-support checks */
        i := 0;
        while (i < a) and (j ∉ column-support) do
            /* try to find lexicographically smallest i supporting j */
            i := i + 1;
            if (C_ij = unknown) then
                C_ij := M_ij;
                if (C_ij = 1) then
                    column-support := column-support ∪ {j};
                fi;
            fi;
        od;
    od;
    return ( row-support, column-support );
od;
```

Table 3.2: The Double-Support Algorithm $\mathcal{D}$

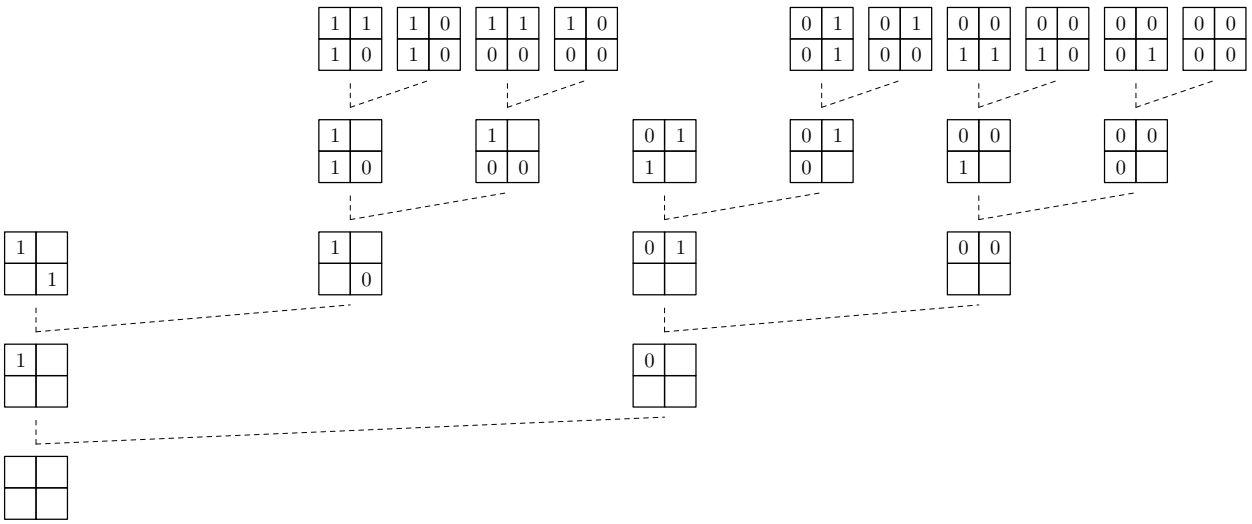

Figure 3.2: Traces of $\mathcal{D}$. Total Number of Support-Checks is $16 \times 4 - 4 \times 2 - 2 \times 1 = 54$.

We have depicted the traces of $\mathcal{D}$ in Figure (3.2). It may not be immediately obvious, but $\mathcal{D}$'s strategy is more efficient than $\mathcal{L}$'s. The reason for this is as follows. There are two traces whose length is shorter than $ab = 4$. There is one such trace whose length is $l_1 := 2$ and one such 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 on average when compared to $\mathcal{L}$.

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 $\mathcal{L}$.

$\mathcal{D}$'s strategy to prefer double-support checks over single-support checks leads to shorter traces. We can use the Trace Theorem to find that that the savings of a trace are of the form $(ab - l)2^{ab-l}$, 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 be able to find this trace had a big impact on the total savings of $\mathcal{D}$. The reason why $\mathcal{D}$ was able to find the short trace was because it was the result of a sequence of successful double-support checks and $\mathcal{D}$'s 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 over single-support checks.

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

In this section we have compared 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 the double-support algorithm was more efficient on average than the lexicographical algorithm for the problem under consideration.

We have been able to identify three reasons which, from an intuitive point of view, suggest that arc-consistency algorithms should prefer double-support checks to single-support checks. The first reason is that a double-support check has a pay-off which is twice as much. If a double-support check is successful two things are learned in return for only one support-check, as opposed to only one new thing for a successful single-support check. The second reason is that it is a necessary condition to maximise the number of successful double-support checks in order to minimise the total number of support-checks. The third and last reason is that the savings of a trace are of the form $(\mathtt{ab}-\mathtt{l})2^{\mathtt{ab}-\mathtt{l}}$, where $\mathtt{l}$ is the length of the trace. The shorter the trace, the bigger the savings. Only traces that contain many successful double support-checks can become very small and thus lead to big savings. To find many such traces requires a heuristic which gives preference to double-support checks.

In the following chapters we will provide solid mathematical evidence that $\mathcal{D}$'s strategy is superior to $\mathcal{L}$'s.

Chapter 4

Average Time-Complexity of $\mathcal{L}$

In this chapter we shall investigate the average time-complexity of the lexicographical algorithm. The organisation of this chapter is as follows. First we shall define the notions of *left* and *right* support-checks. Next, we shall determine the average time-complexity of $\mathcal{L}$ by computing the average number of left and right support-checks of $\mathcal{L}$ and use them to establish an exact formula for the average time-complexity of $\mathcal{L}$. Finally, we shall determine simple upper and lower bounds for the average time-complexity of $\mathcal{L}$.

$\mathcal{L}$ establishes its support in two phases. In its first phase $\mathcal{L}$ tries to establish its row-support. In its second phase $\mathcal{L}$ carries out the remaining work to find the column-support. In the following, we will call the checks that are carried out in the first phase the *left support-checks*. The checks that are carried out in the second phase will be called the *right support-checks*.

The following relates b and the sum of the left support-checks of a row of length b.

Lemma 1 (Left Support-Checks for Single Rows) *Let* b *be a positive integer,* $\mathsf{M} \in \mathbb{M}^{1\mathsf{b}}$ *a* 1 *by* b *matrix and* $\mathsf{L}_{\mathcal{L}}^{\mathsf{M}}$ *the number of support-checks required by* $\mathcal{L}$ *to determine the row-support of* M*. Then*

$$\sum_{\mathsf{M}\in\mathbb{M}^{1\mathsf{b}}} \mathsf{L}_{\mathcal{L}}^{\mathsf{M}} = 2^{\mathsf{b}+1} - 2.$$

Proof. First observe that a check which is spent on column $\mathsf{c} < \mathsf{d}$ can only be successful if the previous $\mathsf{c} - 1$ checks have all failed. Next observe that there are $2^{\mathsf{b}-\mathsf{c}}$ different cases where such a situation can arise. Finally, there are only two cases where b support-checks

have to be carried out. We can use these observations as follows:

$$\begin{aligned}
\sum_{M\in\mathbb{M}^{1b}} L_{\mathcal{L}}^{M} &= 2b + \sum_{c=1}^{b-1} c2^{b-c} \\
&= 2b + \sum_{c=1}^{b-1} (b-c)2^{c} \\
&= 2b + b(2^{b}-2) - \sum_{c=1}^{b-1} c2^{c} \\
&= b2^{b} - \sum_{c=1}^{b-1} c2^{c} \\
&= 2^{b+1} - 2.
\end{aligned}$$

The last equality holds because it follows from Lemma (6) (see Appendix) that $\sum_{c=1}^{b-1} c2^{c} = 2 + (b-2)2^{b}$. □

The following relates a, b and the sum of the left support-checks of all a by b matrices.

Lemma 2 (Left Support-Checks by $\mathcal{L}$) *Let a and b be positive integers. Let $L_{\mathcal{L}}^{M}$ be the number of support-checks required by $\mathcal{L}$ to determine the row-support of M. Then*

$$\sum_{M\in\mathbb{M}^{ab}} L_{\mathcal{L}}^{M} = a(2-2^{1-b})2^{ab}.$$

Proof. We can use Lemma (1) to count the number of checks that have to be spent on a row at a fixed position, say row r, in a matrix. This number is given by $2^{b+1}-2$. For each sequence of length b that consists of ones and zeroes there are exactly $2^{b(a-1)}$ different matrices where this sequence occurs in the row at the same fixed position r. Since there are a rows and since we can count the checks spent on each of the different rows independently, the total number of checks spent on a rows of length b is the same as a times the total number of checks spent on one row of length b. Therefore,

$$\begin{aligned}
\sum_{M\in\mathbb{M}^{ab}} L_{\mathcal{L}}^{M} &= (2^{b+1}-2)a2^{b(a-1)} \\
&= a(2-2^{-b})2^{ab},
\end{aligned}$$

which ends the proof. □

The following relates a, b and the sum of the right support-checks of all a by b matrices.

Lemma 3 (Right Support-Checks by $\mathcal{L}$) *Let a and b be positive integers, $M \in \mathbb{M}^{ab}$ an a by b matrix and $R_{\mathcal{L}}^{M}$ the number of support-checks required by $\mathcal{L}$ to determine the column-support of M after it has established the row-support of M. Then*

$$\sum_{M\in\mathbb{M}^{ab}} R_{\mathcal{L}}^{M} = 2^{ab}(2^{1-a}(1-b) + 2\sum_{c=2}^{b}(1-2^{-c})^{a}).$$

Proof. A support-check should only be carried out if there are no rows $r < r'$ s.t. $M_{r'c} = 1$ and $M_{r'c'} = 0$ for $1 \leq c' < c$ and no rows of the form $r'' < r$ s.t. $M_{r''c} = 1$. Therefore,

$$\begin{aligned}
\sum_{M \in \mathbb{M}^{a\,b}} R^M_{\mathcal{L}} &= \sum_{c=2}^{b} \sum_{r=1}^{a} 2^{a(b-c)} 2^{(r-1)(c-1)} (2^c - 2)(2^c - 1)^{a-r} \\
&= \sum_{c=2}^{b} \sum_{r=1}^{a} 2^{a(b-c)} 2^{(r-1)(c-1)} (2^c - 2)(2^c - 1)^{a-r} \\
&= \sum_{c=2}^{b} 2^{a(b-c)} (2^c - 2)(2^c - 1)^{a-1} \sum_{r=1}^{a} 2^{(r-1)(c-1)} (2^c - 1)^{1-r} \\
&= \sum_{c=2}^{b} 2^{a(b-c)+1} (2^{c-1} - 1)(2^c - 1)^{a-1} \sum_{r=0}^{a-1} (2^{c-1}/(2^c - 1))^r. \qquad (4.1)
\end{aligned}$$

Note that in the inner summation of Equation (4.1) the value of c is always greater than one. Therefore, $2^{c-1}/(2^c - 1) \neq 1$ for all the c that are under consideration. The inner summation turns out to be the sum of a geometric series and we can use Theorem 6 (see Appendix) to simplify it as follows:

$$\begin{aligned}
&\sum_{r=0}^{a-1} (2^{c-1}/(2^c - 1))^r \\
&= (1 - 2^{a(c-1)} (2^c - 1)^{-a})(1 - 2^{c-1} (2^c - 1)^{-1})^{-1} \\
&= (2^c - 1)(2^{c-1} - 1)^{-1} (1 - 2^{a(c-1)} (2^c - 1)^{-a})).
\end{aligned}$$

This allows us to continue to simplify Equation (4.1) as follows:

$$\begin{aligned}
&\sum_{c=2}^{b} 2^{a(b-c)+1} (2^{c-1} - 1)(2^c - 1)^{a-1} \sum_{r=0}^{a-1} (2^{c-1}/(2^c - 1))^r \\
&= \sum_{c=2}^{b} 2^{a(b-c)+1} (2^c - 1)^a (1 - 2^{a(c-1)} (2^c - 1)^{-a})) \\
&= 2^{ab} \sum_{c=2}^{b} (2(1 - 2^{-c})^a - 2^{1-a}) \\
&= 2^{ab} (2^{1-a}(1 - b) + 2 \sum_{c=2}^{b} (1 - 2^{-c})^a),
\end{aligned}$$

which concludes the proof of Lemma (3). □

We are finally in a position where we can determine the average time complexity of $\mathcal{L}$.

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 given by the function $\mathrm{avg}_{\mathcal{L}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$, where*

$$\mathrm{avg}_{\mathcal{L}}(a, b) = a(2 - 2^{1-b}) + (1 - b)2^{1-a} + 2 \sum_{c=2}^{b} (1 - 2^{-c})^a. \qquad (4.2)$$

Proof. Divide the result of the addition of the left and right support-checks of $\mathcal{L}$ by 2^{ab}. □

The following is an immediate consequence of Theorem 2.

Corollary 1 (Upper Bound for $\mathcal{L}$'s Average Time-Complexity) *Let* a *and* b *be positive integers. An upper bound for the average time complexity of* $\mathcal{L}$ *over* $\mathbb{M}^{ab}$ *is the function* $\mathrm{upb}_{\mathcal{L}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$*, where*

$$\mathrm{upb}_{\mathcal{L}}(a, b) = 2a + 2b - 2 - 2^{1-b}a + (1-b)2^{1-a}.$$

Proof. The corollary is a consequence of the fact that the summation in Equation (4.2) satisfies the following inequality:

$$\sum_{c=2}^{b} (1 - 2^{-c})^a \leq \sum_{c=2}^{b} 1^a = b - 1,$$

which completes the proof. □

In the following, we will use the Bernoulli Inequality (Theorem 8) to obtain a tight lower bound for the average time-complexity of $\mathcal{L}$. The proof of the corollary is rather longwinded.

Corollary 2 (Lower Bound for $\mathcal{L}$'s Average Time-Complexity) *Let* s *be a constant positive integer. If* s *is sufficiently much less than* a *then* $\mathrm{lwb}_{\mathcal{L}}(a, b) \leq \mathrm{avg}_{\mathcal{L}}(a, b)$*, where*

$$\mathrm{lwb}_{\mathcal{L}}(a, b) = 2a + 2b + a\phi(s) + \mathbf{O}(1) + \mathbf{O}(b2^{-a}) + \mathbf{O}(a2^{-b}),$$

where

$$\phi(s) = 2^{-s} - 2\sum_{k=0}^{s} \binom{s}{k} (-1)^k (2^{k+1} - 1)^{-1}.$$

Proof. We can use Corollary (3) to find that as a becomes large the following is true for any non-negative *constant* integer s:

$$(1 - 2^{-c})^a \geq (1 - 2^{-c})^s (1 - (a - s)2^{-c}).$$

The summation in Equation (4.2) therefore satisfies the following inequality:

$$\sum_{c=2}^{b} (1 - 2^{-c})^a \geq \sum_{c=2}^{b} (1 - 2^{-c})^s (1 - (a - s)2^{-c}).$$

We can use Newton's Binomial Theorem (Theorem 7) to rewrite this as follows:

$$\begin{aligned}
&\sum_{c=2}^{b} (1 - 2^{-c})^s (1 - (a - s)2^{-c}) \\
&= \sum_{c=2}^{b} \sum_{k=0}^{s} \binom{s}{k} (-1)^k 2^{-kc} (1 - (a - s)2^{-c}) \\
&= \sum_{c=2}^{b} \sum_{k=0}^{s} \binom{s}{k} (-1)^k (2^{-kc} + s2^{-kc-c}) - a \sum_{c=2}^{b} \sum_{k=0}^{s} \binom{s}{k} (-1)^k 2^{-kc-c}. \qquad (4.3)
\end{aligned}$$

We shall first simplify the first term of Equation (4.3) and then simplify its second term. Each simplification relies on the fact that we assume s to be a fixed positive integer sufficiently small w.r.t. a. This allows us to make simplifications like $s^n = \mathbf{O}(1)$ for any constant n. The simplification of the first term is as follows:

$$\begin{aligned}
&\sum_{c=2}^{b}\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{-kc}+s2^{-kc-c})\\
&= \sum_{k=0}^{s}\binom{s}{k}(-1)^k\sum_{c=2}^{b}(2^{-kc}+s2^{-kc-c})\\
&= \sum_{k=0}^{s}\binom{s}{k}(-1)^k\sum_{c=2}^{b}2^{-kc}+s\sum_{k=0}^{s}\binom{s}{k}(-1)^k\sum_{c=2}^{b}2^{-kc-c}\\
&= \sum_{k=0}^{s}\binom{s}{k}(-1)^k\sum_{c=2}^{b}2^{-kc}+s\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{-k-1}-2^{-bk-b})(2^{k+1}-1)^{-1},
\end{aligned}$$

where the simplification relies on Theorem 6. We can simplify this if we "put" some expressions in $\mathbf{O}(\cdot)$-terms.

$$\begin{aligned}
&\sum_{k=0}^{s}\binom{s}{k}(-1)^k\sum_{c=2}^{b}2^{-kc}+s\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{-k-1}-2^{-bk-b})(2^{k+1}-1)^{-1}\\
&= \mathbf{O}(1)+\mathbf{O}(2^{-b})+\sum_{k=0}^{s}\binom{s}{k}(-1)^k\sum_{c=2}^{b}2^{-kc}\\
&= (b-1)+\mathbf{O}(1)+\mathbf{O}(2^{-b})+\sum_{k=1}^{s}\binom{s}{k}(-1)^k\sum_{c=2}^{b}2^{-kc}\\
&= b+\mathbf{O}(1)+\mathbf{O}(2^{-b})+\sum_{k=1}^{s}\binom{s}{k}(-1)^k((2^{2k}-2^k)^{-1}-(2^{bk+k}-2^{bk})^{-1})\\
&= b+\mathbf{O}(1)+\mathbf{O}(2^{-b}).
\end{aligned}$$

This ends the simplification of the first term of Equation (4.3). The simplification of the second term of Equation (4.3) is as follows:

$$\begin{aligned}
&-a\sum_{c=2}^{b}\sum_{k=0}^{s}\binom{s}{k}(-1)^k2^{-kc-c}\\
&= -a\sum_{k=0}^{s}\binom{s}{k}(-1)^k\sum_{c=2}^{b}2^{-kc-c}\\
&= -a\sum_{k=0}^{s}\binom{s}{k}(-1)^k(\frac{2^{-k-1}}{2^{k+1}-1}-\frac{2^{-kb-b}}{2^{k+1}-1}).
\end{aligned}$$

Again, we can simplify this even further if we put some expressions in $\mathbf{O}(\cdot)$-terms. Our

simplification proceeds as follows:

$$\begin{aligned}
&-a\sum_{k=0}^{s}\binom{s}{k}(-1)^k\left(\frac{2^{-k-1}}{2^{k+1}-1}-\frac{2^{-kb-b}}{2^{k+1}-1}\right)\\
&= \mathbf{O}(a2^{-b})-a\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{2k+2}-2^{k+1})^{-1}\\
&= \mathbf{O}(a2^{-b})+a\sum_{k=0}^{s}\binom{s}{k}(-1)^k2^{-k-1}-a\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{k+1}-1)^{-1}\\
&= a2^{-s-1}+\mathbf{O}(a2^{-b})-a\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{k+1}-1)^{-1}.
\end{aligned}$$

Having simplified the first and the second term of Equation (4.3) we can simplify Equation (4.3) as follows:

$$\begin{aligned}
&\sum_{c=2}^{b}\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{-kc}+s2^{-kc-c})-a\sum_{c=2}^{b}\sum_{k=0}^{s}\binom{s}{k}(-1)^k2^{-kc-c}\\
&= b+\mathbf{O}(1)+\mathbf{O}(2^{-b})+a2^{-s-1}+\mathbf{O}(a2^{-b})-a\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{k+1}-1)^{-1}\\
&= b+\mathbf{O}(1)+\mathbf{O}(a2^{-b})+a2^{-s-1}-a\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{k+1}-1)^{-1}.
\end{aligned}$$

We can therefore derive the following expression for the lower bound of the average time-complexity of $\mathcal{L}$.

$$\begin{aligned}
\mathrm{avg}_{\mathcal{L}}(a,b) &= a(2-2^{1-b})+(1-b)2^{1-a}+2\sum_{c=2}^{b}(1-2^{-c})^a\\
&\geq 2a+2b+a\phi(s)+\mathbf{O}(1)+\mathbf{O}(b2^{-a})+\mathbf{O}(a2^{-b}) \qquad (4.4)\\
&= \mathrm{lwb}_{\mathcal{L}}(a,b),
\end{aligned}$$

where

$$\phi(s)=2^{-s}-2\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{k+1}-1)^{-1}, \qquad (4.5)$$

which concludes the proof of Corollary (2). □

We have studied the function ϕ from Equation (4.5) and have noticed that it becomes smaller as s becomes large. We have not been able to prove this. It seems that $\phi(2s)\approx\phi(s)/2$. For example, $\phi(20)\approx -0.1373920$, $\phi(40)\approx -0.0703735$, $\phi(80)\approx -0.0356215$ and so on. This seems to suggest that, provided $\lim_{s\to\infty}\phi(s)=0$, the lower bound for the average number of support-checks of $\mathcal{L}$ is of the form $2a+2b+\mathbf{O}(1)$ as a and b become large and are of the same magnitude.

Note that the formula for the lower bound can be used as follows. If we let $s = 80$ then we can show that the average number of consistency-checks required by $\mathcal{L}$ is bounded from below by

$$\begin{aligned}&(2 + \phi(s))a + 2b + \mathbf{O}(1) + \mathbf{O}(b2^{-a}) + \mathbf{O}(a2^{-b})\\ &\quad\approx\quad 1.9643785a + 2b + \mathbf{O}(1) + \mathbf{O}(b2^{-a}) + \mathbf{O}(a2^{-b}).\end{aligned}$$

We have used the same approach for greater values of s. In all the cases considered by us, the technique worked. The greater the s was chosen by us, the better the coefficient of a in the resulting expression approximated two.

The expression for $\text{lwb}_{\mathcal{L}}$ is interesting because if a and b are of the same magnitude and become large then $\text{lwb}_{\mathcal{L}}$ is "almost" of the form $2a + 2b + \mathbf{O}(1)$. Again, this is under the assumption that $\lim_{s\to\infty} \phi(s) = 0$. This seems to suggest that $\mathcal{L}$ requires or saves a constant number of support-checks on average and requires two checks for each of the members in the domains of each of the variables. Given that on average every second entry of a matrix is a one, this seems to suggest that $\mathcal{L}$ cannot use checks that are used to learn about its row-support to also learn about its column-support and vice versa. Otherwise, $\mathcal{L}$'s lower bound would not have included coefficients which are (approximately) 2 for *both* a *and* b. It is as if $\mathcal{L}$ carries out the checks to find its row-support on the one hand and the checks to find its column-support on the other almost completely independently from each other.

We can also explain the results we have obtained for the lower bound for $\mathcal{L}$ in the following way. As we already observed, $\mathcal{L}$ will first try to establish its row-support. In total $\mathcal{L}$ requires about $2a$ support-checks for this task on average. Most of the support-checks are only spent on the first few columns. If a and b are sufficiently large and are of the same magnitude then after it has established its row support, it is still not known for "almost" all columns whether they are in the column-support and for each of these "almost" b columns two support-checks are required. In total this means that on average $\mathcal{L}$ will require "almost" $2a + 2b$ support-checks.

Chapter 5

Average Time-Complexity of $\mathcal{D}$

In this section we shall derive the average time-complexity of $\mathcal{D}$. It will turn out that this is a bit easier than the complexity analysis carried out in the previous section. As part of our analysis we will demonstrate that if $a + b \geq 14$ then $\mathcal{D}$ requires strictly less than two checks more than any algorithm.

The organisation of this section is as follows. We shall first establish a recurrence equation for the average time-complexity of $\mathcal{D}$ and from it derive an upper and a lower bound for its average time-complexity.

Theorem 3 (Average Time Complexity of $\mathcal{D}$) *The average time complexity of $\mathcal{D}$ over $\mathbb{M}^{ab}$ is given by $\mathrm{avg}_{\mathcal{D}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$, where $\mathrm{avg}_{\mathcal{D}}(a, 0) = 0$, $\mathrm{avg}_{\mathcal{D}}(0, b) = 0$ and*

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

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

Proof. Let $\mathrm{tot}_{\mathcal{D}}(a, b) = \sum_{M \in \mathbb{M}^{ab}} \mathrm{checks}_{\mathcal{D}}(M)$. We shall first show how to obtain $\mathrm{avg}_{\mathcal{D}}(\cdot, \cdot)$ from $\mathrm{tot}_{\mathcal{D}}(\cdot, \cdot)$. Next we shall show how to obtain $\mathrm{tot}_{\mathcal{D}}(\cdot, \cdot)$.

The function $\mathrm{tot}_{\mathcal{D}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ is given by $\mathrm{tot}_{\mathcal{D}}(a, 0) = 0$, $\mathrm{tot}_{\mathcal{D}}(0, b) = 0$ and by

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

if $a \neq 0$ and $b \neq 0$. We can obtain $\mathrm{avg}_{\mathcal{D}}(s, t)$ from $\mathrm{tot}_{\mathcal{D}}(s, t)$ using the fact that $\mathrm{avg}_{\mathcal{D}}(s, t) = \mathrm{tot}_{\mathcal{D}}(s, t)/2^{st}$.

In the second and last part of the proof we have to demonstrate that the recurrence equation for $\mathrm{tot}_{\mathcal{D}}(\cdot, \cdot)$ is correct. The proof turns out to be easy when compared with the lexicographical case.

Note that $\mathrm{tot}_{\mathcal{D}}(1, b) = b2^b$ and $\mathrm{tot}_{\mathcal{D}}(a, 1) = a2^a$ and for both cases, the recurrence equation is satisfied. Assume that both a and b are greater than one, then:

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

The following four points explain the purpose of each of the terms in the equation.

1. The term $(2^{b+1}-2)2^{(a-1)b}$ corresponds to the effort spent to find the smallest j s.t. $M_{1j}=1$, where the effort is over all matrices in $\mathbb{M}^{ab}$. Lemma (1) states that the such effort spent on a row with b columns is $2^{b+1}-2$. The total number of checks is therefore given by $(2^{b+1}-2)2^{(a-1)b}$. The factor $2^{(a-1)b}$ accounts for the fact that the number of a by b matrices that have the same first row is $2^{(a-1)b}$.

2. If the first row was zero—there is one such case—then we have to compute $\mathrm{tot}_{\mathcal{D}}(a-1,b)$.

3. If the first row was non-zero—there are 2^b-1 such rows—we have to compute $\mathrm{tot}_{\mathcal{D}}(a-1,b-1)$. Let j be the smallest positive integer s.t. $M_{1j}=1$. The factor 2^{a-1} accounts for the column below M_{1j} that is not checked in the recursive chase.

4. "After" the recursive application the following two independent tasks have to be carried out.

   (a) For every column, say j, for which no one could be found in the recursive case the check $M^{?}_{1j}$ has to be carried out. The following is an illustration of this case.

   $$\begin{bmatrix} * & \cdots & * & 1 & * & \cdots & * \\ & & & & 0 & & \\ & & & & \vdots & & \\ & & & & 0 & & \end{bmatrix}$$

   The term $2^{(a-1)(b-1)}((b-2)2^b+2)$ counts the number of these checks.

   (b) If the first row is non-zero and if j is the column for which it was found out that $M_{1j}=1$, then for every row, say r, for which no one could be found in the recursive case, it has to be checked if $M_{rj}=1$. The following is an illustration of this case.

   $$\begin{bmatrix} 0 & \cdots & 0 & 1 & * & \cdots & * \\ 0 & \cdots & 0 & * & 0 & \cdots & 0 \end{bmatrix}$$

   The term $(a-1)(2^b-1)2^{(a-2)b+1}$ counts these checks. The factor $(a-1)$ corresponds to the number of rows whose index is greater than one and the factor 2^b-1 is the number of non-zero rows of length b. The factor $2^{(a-2)b+1}$ is made up of the factor 2^1 which accounts for M_{rj} and the factor $2^{(a-2)b}$ which accounts for the M_{ij}, where $i\neq 1$ and $i\neq r$.

This ends the proof of Theorem 3. □

Theorem 4 (Upper Bound for $\mathcal{D}$'s Average Time-Complexity) *Let* $\mathfrak{a}$ *and* $\mathfrak{b}$ *be positive integers s.t.* $\mathfrak{a}+\mathfrak{b} \geq 14$. *An upper bound for the average time-complexity of* $\mathcal{D}$ *over* $\mathbb{M}^{\mathfrak{a}\mathfrak{b}}$ *is given by* $\mathrm{upb}_{\mathcal{D}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$, *where*

$$\begin{aligned} \mathrm{upb}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) \;=\; & 2\max(\mathfrak{a},\mathfrak{b})+2 \\ & -(2\max(\mathfrak{a},\mathfrak{b})+\min(\mathfrak{a},\mathfrak{b}))2^{-\min(\mathfrak{a},\mathfrak{b})} \\ & -(2\min(\mathfrak{a},\mathfrak{b})+3\max(\mathfrak{a},\mathfrak{b}))2^{-\max(\mathfrak{a},\mathfrak{b})}. \end{aligned} \tag{5.1}$$

Proof. We shall prove this by induction on $\mathfrak{a}+\mathfrak{b}$. Let $\mathcal{P}(\mathfrak{i})$ be true iff $\mathfrak{i} \geq 14$ and there are two positive integers $\mathfrak{a}$ and $\mathfrak{b}$ s.t. $\mathfrak{i} = \mathfrak{a}+\mathfrak{b}$ and $\mathrm{avg}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) \leq \mathrm{upb}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b})$.

We shall first verify the cases where $\mathfrak{a}=1$ or $\mathfrak{b}=1$ and then tackle the more general case.

$$\begin{aligned} & \mathrm{avg}_{\mathcal{D}}(\mathfrak{a},1) - \mathrm{upb}_{\mathcal{D}}(\mathfrak{a},1) \\ &= \mathfrak{a} - (2\mathfrak{a}+2-(2\mathfrak{a}+1)/2-(2+3\mathfrak{a})2^{-\mathfrak{a}}) \\ &= 2^{1-\mathfrak{a}} + 3\mathfrak{a}/2^{-\mathfrak{a}} - 3/2. \end{aligned}$$

This means that if $\mathfrak{a}$ becomes greater than 2 then $\mathrm{avg}_{\mathcal{D}}(\mathfrak{a},1) \leq \mathrm{upb}_{\mathcal{D}}(\mathfrak{a},1)$. The case where $\mathfrak{b}=1$ is similar.

We have verified that for all integers $1<\mathfrak{a}$ and $1<\mathfrak{b}$ satisfying $14 \leq \mathfrak{a}+\mathfrak{b} \leq 15$ it is true that $\mathrm{avg}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) \leq \mathrm{upb}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b})$. In other words, $\mathcal{P}(14)$ and $\mathcal{P}(15)$ are true.

Assume that $\mathcal{P}(\mathfrak{i}-2)$ and $\mathcal{P}(\mathfrak{i}-1)$ are true for some integer $\mathfrak{i} \geq 16$. We must prove that $\mathcal{P}(\mathfrak{i})$ must hold as well. Let $\mathfrak{a}$ and $\mathfrak{b}$ be any integers satisfying the equations $\mathfrak{i} = \mathfrak{a}+\mathfrak{b}$. We already know that if $\mathfrak{a}=1$ or $\mathfrak{b}=1$ then $\mathcal{P}(\mathfrak{a}+\mathfrak{b})$ holds. Therefore, a proof for the case where $\mathfrak{a}>1$ and $\mathfrak{b}>1$ will suffice.

Let $\mathfrak{a}>1$ and $\mathfrak{b}>1$. We have depicted the "base-case" in Figure (5.1). The thick dark lines and thick dark arrows in the picture represent (some of) the points $(\mathfrak{a}',\mathfrak{b}')$ for which we know $\mathcal{P}(\mathfrak{a}'+\mathfrak{b}')$ to be true. We must prove that for every point $(\mathfrak{a}',\mathfrak{b}')$ in the "interior" area in the upper right of the picture it is also true that $\mathcal{P}(\mathfrak{a}'+\mathfrak{b}')$ is true. As already indicated we will use induction on $\mathfrak{a}+\mathfrak{b}$ to finalise our proof. The proof relies on the fact that if $(\mathfrak{a}_0,\mathfrak{b}_0)$ is a point in the "interior" and if $(\mathfrak{a}_0,\mathfrak{b}_0)$, $(\mathfrak{a}_1,\mathfrak{b}_1)$, ... is a sequence of points where $(\mathfrak{a}_\mathfrak{i}+\mathfrak{b}_\mathfrak{i})-(\mathfrak{a}_{\mathfrak{i}+1}+\mathfrak{b}_{\mathfrak{i}+1})=1$, for all $\mathfrak{i}\in\mathbb{N}$, then there has to be some positive integer $\mathfrak{i}$, s.t. $(\mathfrak{a}_\mathfrak{i},\mathfrak{b}_\mathfrak{i})$ is on one of the thick dark lines or on one of the thick dark arrows, i.e. there has to be an $(\mathfrak{a}_\mathfrak{i},\mathfrak{b}_\mathfrak{i})$ for which $\mathcal{P}(\mathfrak{a}_\mathfrak{i}+\mathfrak{b}_\mathfrak{i})$ holds.

There are two cases: either $1<\mathfrak{b}<\mathfrak{a}$ or $1<\mathfrak{a}\leq\mathfrak{b}$. Assume $1<\mathfrak{b}<\mathfrak{a}$ then

$$\begin{aligned} & \mathrm{avg}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) - \mathrm{upb}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) \\ &= 2+2^{2-\mathfrak{a}-\mathfrak{b}}+(\mathfrak{b}-2)2^{1-\mathfrak{a}}+(\mathfrak{a}-2)2^{1-\mathfrak{b}}-(\mathfrak{a}-1)2^{1-2\mathfrak{b}} \\ &\quad +2^{-\mathfrak{b}}\,\mathrm{avg}_{\mathcal{D}}(\mathfrak{a}-1,\mathfrak{b})+(1-2^{-\mathfrak{b}})\,\mathrm{avg}_{\mathcal{D}}(\mathfrak{a}-1,\mathfrak{b}-1)-\mathrm{upb}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) \\ &\leq 2+2^{2-\mathfrak{a}-\mathfrak{b}}+(\mathfrak{b}-2)2^{1-\mathfrak{a}}+(\mathfrak{a}-2)2^{1-\mathfrak{b}}-(\mathfrak{a}-1)2^{1-2\mathfrak{b}} \\ &\quad +2^{-\mathfrak{b}}\,\mathrm{upb}_{\mathcal{D}}(\mathfrak{a}-1,\mathfrak{b})+(1-2^{-\mathfrak{b}})\,\mathrm{upb}_{\mathcal{D}}(\mathfrak{a}-1,\mathfrak{b}-1)-\mathrm{upb}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) \\ &= (6-3\mathfrak{a})2^{-\mathfrak{a}}+(6-3\mathfrak{b})2^{-\mathfrak{b}}-(6-3\mathfrak{b})2^{-2\mathfrak{b}}. \end{aligned}$$

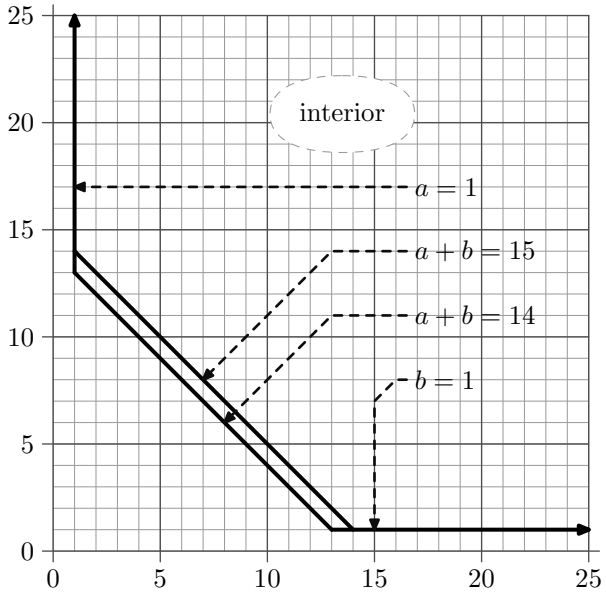

Figure 5.1: Base-Case for Induction on $a + b$.

Since $b \geq 2$ it must hold that $(6 - 3b)2^{-b} \leq (6 - 3b)2^{-2b}$. This allows us to continue our simplification as follows:

$$\begin{aligned}
&(6-3a)2^{-a} + (6-3b)2^{-b} - (6-3b)2^{-2b} \\
&\quad\leq\ (6-3a)2^{-a} + (6-3b)2^{-2b} - (6-3b)2^{-2b} \\
&\quad=\ (6-3a)2^{-a} \\
&\quad\leq\ 0,
\end{aligned}$$

where the last inequality follows from the fact that $a \geq 2$.

Assume that $1 < a \leq b$. We can use the same technique as for the case where $1 < b < a$ to derive the following:

$$\begin{aligned}
&\mathrm{avg}_{\mathcal{D}}(a,b) - \mathrm{upb}_{\mathcal{D}}(a,b) \\
&\quad\leq\ 2 + 2^{2-a-b} + (b-2)2^{1-a} + (a-2)2^{1-b} - (a-1)2^{1-2b} \\
&\qquad +2^{-b}\,\mathrm{upb}_{\mathcal{D}}(a-1,b) + (1-2^{-b})\,\mathrm{upb}_{\mathcal{D}}(a-1,b-1) - \mathrm{upb}_{\mathcal{D}}(a,b) \\
&\quad=\ (6-3a)2^{-a} + (4-3b)2^{-b} + (3b-6)2^{-2b} \\
&\quad\leq\ (6-3a)2^{-a} + (6-3b)2^{-b} + (3b-6)2^{-2b} \\
&\quad\leq\ 0.
\end{aligned}$$

Apparently $\mathcal{P}(i-2) \wedge \mathcal{P}(i-1) \implies \mathcal{P}(i)$. We have shown that $\mathcal{P}(14)$ and $\mathcal{P}(15)$ hold. We have furthermore shown that if $\mathcal{P}(i-2)$ and $\mathcal{P}(i-1)$ hold for some integer $i \geq 16$ then $\mathcal{P}(i)$ must hold as well. In other words, $\mathcal{P}(i)$ is true for every integer $i \geq 14$, which completes our proof. □

An important result that follows from Theorem 4 is that we can can prove that $\mathcal{D}$ is efficient. Before we provide this result we present the following lemma to shorten the proof.

Lemma 4 (Lower Bound) *Let* $\mathfrak{a}$ *and* $\mathfrak{b}$ *be positive integers and let* $\mathcal{A}$ *be any arc-consistency algorithm then*

$$\max(\mathfrak{a},\mathfrak{b})(2-2^{1-\min(\mathfrak{a},\mathfrak{b})}) \leq \mathrm{avg}_{\mathcal{A}}(\mathfrak{a},\mathfrak{b}).$$

Proof. Assume $\mathfrak{b} \leq \mathfrak{a}$. It is obvious that $\mathrm{avg}_{\mathcal{A}}(\mathfrak{a},\mathfrak{b})$ is at least as much as the average number of support-checks that have to be carried out to find the row-support. The average number of support-checks to find the row-support are at least $\mathfrak{a}(2-2^{1-\mathfrak{b}})$ (the number of left support-checks). The case for $\mathfrak{a} < \mathfrak{b}$ is analogous. □

Theorem 5 (Efficiency) *Let* $\mathcal{A}$ *be any arc-consistency algorithm and let* $\mathfrak{a}+\mathfrak{b} \geq 14$ *then*

$$\mathrm{avg}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) - \mathrm{avg}_{\mathcal{A}}(\mathfrak{a},\mathfrak{b}) \leq 2 - \min(\mathfrak{a},\mathfrak{b})2^{-\min(\mathfrak{a},\mathfrak{b})} - (2\min(\mathfrak{a},\mathfrak{b}) + 3\max(\mathfrak{a},\mathfrak{b}))2^{-\max(\mathfrak{a},\mathfrak{b})}.$$

Proof. If $\mathrm{avg}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) < \mathrm{avg}_{\mathcal{A}}(\mathfrak{a},\mathfrak{b})$ then the theorem is obviously true. If $\mathrm{avg}_{\mathcal{A}}(\mathfrak{a},\mathfrak{b}) \leq \mathrm{avg}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b})$ then it follows from Lemma (4) that

$$\begin{aligned}
&\max(\mathfrak{a},\mathfrak{b})(2-2^{1-\min(\mathfrak{a},\mathfrak{b})}) \\
&\quad\leq \mathrm{avg}_{\mathcal{A}}(\mathfrak{a},\mathfrak{b}) \\
&\quad\leq \mathrm{avg}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) \\
&\quad\leq \mathrm{upb}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}),
\end{aligned}$$

where $\mathrm{upb}_{\mathcal{D}}$ is the upper bound provided by Theorem 4. It then follows that

$$\begin{aligned}
&\mathrm{avg}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) - \mathrm{avg}_{\mathcal{A}}(\mathfrak{a},\mathfrak{b}) \\
&\quad\leq \mathrm{upb}_{\mathcal{D}}(\mathfrak{a},\mathfrak{b}) - \mathrm{avg}_{\mathcal{A}}(\mathfrak{a},\mathfrak{b}) \\
&\quad= 2 - \min(\mathfrak{a},\mathfrak{b})2^{-\min(\mathfrak{a},\mathfrak{b})} - (2\min(\mathfrak{a},\mathfrak{b}) + 3\max(\mathfrak{a},\mathfrak{b}))2^{-\max(\mathfrak{a},\mathfrak{b})},
\end{aligned}$$

which completes the proof. □

In other words, if $\mathcal{A}$ is any arc-consistency algorithm which is better than $\mathcal{D}$, then $\mathcal{D}$ will spend strictly less than two checks more on average than $\mathcal{A}$.

Chapter 6

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

In this section we shall compare the average number of support-checks required by $\mathcal{L}$ and $\mathcal{D}$. The comparison will consist of a theoretical evaluation of the average time-complexity of $\mathcal{L}$ and $\mathcal{D}$ and of a comparison of the average number of support-checks for some special cases.

The remainder of this section is organised as follows. In Chapter 6.1 we shall compare the results obtained from the average time-complexity analysis of $\mathcal{L}$ and $\mathcal{D}$ from a theoretical point of view. In Chapter 6.2 we shall compare the results of the average time required by $\mathcal{L}$ and $\mathcal{D}$ for the problem classes $\mathbb{M}^{nn}$, for $1 \leq n \leq 20$.

6.1 A Theoretical Comparison of $\mathcal{L}$ and $\mathcal{D}$

In this section we shall compare the results obtained in Chapter 4 and Chapter 5 from a theoretical point of view.

We already observed in Chapter 3.1 that the minimum number of support-checks required by $\mathcal{L}$ is $a+b-2$. In Chapter 5 we have derived an upper bound below $\max(a,b)+2$ for the average number of support-checks required by $\mathcal{D}$, provided that $a+b \geq 14$. If $a+b \geq 14$ and $a=b$ then 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 follows almost immediately from our average time-complexity analysis. It is the observation that $\mathcal{D}$ is a better algorithm than $\mathcal{L}$ because its upper bound is lower than the lower bound that we derived for $\mathcal{L}$ using Theorem 2 for the case where $s=80$. When a and b get large and are of the same magnitude then the difference is about $2\min(a,b)$ which is quite substantial.

We remarked that it was as if $\mathcal{L}$ carried out the checks to find its row-support on the one hand and the checks to find its column-support on the other completely independently of each other.

Our most important result is the observation that if $a+b \geq 14$ and if $\mathcal{A}$ is any arc-consistency algorithm then $\text{avg}_{\mathcal{D}}(a,b) - \text{avg}_{\mathcal{A}}(a,b) < 2$. To the best of our knowledge, this is the first such result that has been obtained.

6.2 A Comparison for Some Special Cases

In this section we shall compare the average time required by $\mathcal{L}$ and $\mathcal{D}$ for the first twenty cases where the number of rows and the number of columns are the same.

n	$\text{avg}_{\mathcal{L}}$	$\text{avg}_{\mathcal{D}}$	ratio	n	$\text{avg}_{\mathcal{L}}$	$\text{avg}_{\mathcal{D}}$	ratio
1	1.000	1.000	1.000	11	36.276	23.678	1.532
2	3.625	3.375	1.074	12	40.040	25.688	1.559
3	6.934	6.043	1.147	13	43.821	27.694	1.582
4	10.475	8.623	1.215	14	47.616	29.697	1.603
5	14.093	11.037	1.277	15	51.425	31.699	1.622
6	17.740	13.306	1.333	16	55.245	33.699	1.639
7	21.408	15.472	1.384	17	59.075	35.700	1.655
8	25.095	17.571	1.428	18	62.915	37.700	1.668
9	28.802	19.628	1.467	19	66.763	39.700	1.682
10	32.529	21.660	1.502	20	70.619	41.700	1.693

Table 6.1: Comparison of $\text{avg}_{\mathcal{L}}(n, n)$ and $\text{avg}_{\mathcal{D}}(n, n)$ for $n \in \{1, \ldots, 20\}$

Table 6.1 compares the average time-complexity of $\mathcal{L}$ and $\mathcal{D}$ for each of the problem-classes $\mathbb{M}^{nn}$, where $1 \leq n \leq 20$. The columns n corresponds to the class $\mathbb{M}^{nn}$. The columns $\text{avg}_{\mathcal{L}}$ lists the average number of support-checks required by $\mathcal{L}$. The columns $\text{avg}_{\mathcal{D}}$ lists the average number of support-checks required by $\mathcal{D}$. The columns ratio corresponds to the ratio between $\text{avg}_{\mathcal{L}}$ and $\text{avg}_{\mathcal{D}}$. The data in Table 6.1 have been obtained with the use of Theorem 2 and Theorem 3.

It is important to state that the computations have *not* been carried out using floating-point numbers but with arbitrary-precision integers. This is necessary to avoid the loss of precision due to the enormous differences in the ratios between the absolute values of the numbers occurring in the formulae for the average number of support-checks required by $\mathcal{L}$ and $\mathcal{D}$.

The same data as presented in Table 6.1 are also presented in the form of a graph in Figure (6.1). The horizontal axis represents the size of the problem classes. A number n on this axis corresponds to the class of n by n matrices. The vertical axis represents the average number of support-checks required by both algorithms. The solid line in the graph represents the average number of support-checks spent by $\mathcal{L}$. The dashed line in the graph represents the average number of support-checks spent by $\mathcal{D}$.

The figure clearly demonstrates what was stated before as Theorem 3, namely that $\text{avg}_{\mathcal{D}}$ is linear in the size of the problems. It furthermore demonstrates that already for small problem sizes does $\mathcal{D}$ become significantly better than $\mathcal{L}$ and remains to be so.

In Figure (6.2) we have depicted the graph of $\text{upb}_{\mathcal{D}} - \text{avg}_{\mathcal{D}}$ for the first twenty cases where the number of rows and columns are the same. The position where the graph becomes positive is where $n = 7$, i.e. $\text{upb}_{\mathcal{D}}(a, b) - \text{avg}_{\mathcal{D}}(a, b)$ becomes positive when $a + b = 14$ which conforms with our analysis in the previous chapter. As the size of the problem increases the upper bound seems to remain about 0.25 above the average. This seems to suggest that it is still possible to improve upon the upper bound.

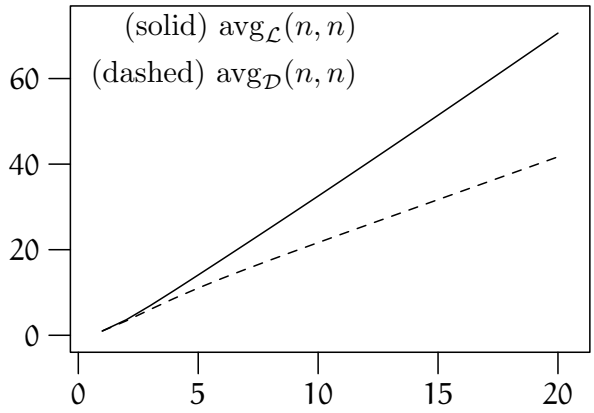

Figure 6.1: $\mathrm{avg}_{\mathcal{L}}$ and $\mathrm{avg}_{\mathcal{D}}$

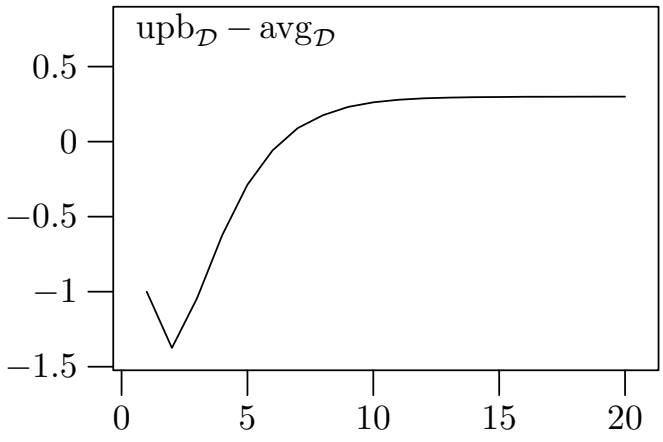

Figure 6.2: $\mathrm{upb}_{\mathcal{D}} - \mathrm{avg}_{\mathcal{D}}$

Chapter 7

Conclusions and Recommendations

In this work 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 carried out a careful average time-complexity analysis 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.

As part of our detailed case-study we have presented three good reasons why arc-consistency algorithms should give preference to double-support checks over other checks. The first reason is that a double-support check has a higher pay-off. If a double-support check is successful two things are learned in return for only one support-check as opposed to only one new thing for a successful single-support check. The second reason is that it is a necessary condition to maximise the number of successful double-support checks in order to minimise the total number of support-checks. The third and last reason is that the savings of a trace are of the form $(ab - l)2^{ab-l}$, where l is the length of the trace.

Our average time-complexity analysis has provided the lower bound $1.9643785a + 2b + \mathbf{O}(1) + \mathbf{O}(b2^{-a}) + \mathbf{O}(a2^{-b})$ for $\text{avg}_{\mathcal{L}}(a, b)$ for sufficiently large a and an upper bound of $2\max(a, b) + 2 - (2\max(a, b) + \min(a, b))2^{-\min(a,b)} - (2\min(a, b) + 3\max(a, b))2^{-\max(a,b)}$ for $\text{avg}_{\mathcal{D}}(a, b)$, provided that $a + b \geq 14$.

Two results follow immediately from the lower bound for $\mathcal{L}$ and the upper bound for $\mathcal{D}$. Our first result is that it clearly shows that $\mathcal{D}$ is the better algorithm of the two. Our second result is the result that if $\mathcal{A}$ is any arc-consistency algorithm and if $a + b \geq 14$ then $\text{avg}_{\mathcal{D}}(a, b) - \text{avg}_{\mathcal{A}}(a, b) < 2$. To the best of our knowledge this is the first such result ever to have been reported.

We think that the work that we started 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 think that it should be worthwhile to tackle the more complicated problem of analysing the case where the

constraints are not required to contain only two variables. Finally, we think that it should be interesting to implement an instance of AC-7 which is equipped with heuristics which partially rely on our double-support heuristic.

Acknowledgements

The author would like to thank Sabin Tabirca for help with part of the proofs. Also he would like to thank Jim Bowen for detailed and useful suggestions on an early draft.

Bibliography

[Bessière *et al.*, 1995] C. Bessière, E.C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In C.S. Mellish, editor, *IJCAI'95*, volume 1, pages 592–598, Montréal, Québec, Canada, 1995. Morgan Kaufmann Publishers, Inc., San Mateo, California, USA.

[Gaschnig, 1978] 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.

[Mackworth and Freuder, 1985] 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.

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

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

[Sabin and Freuder, 1994] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *ECAI'94*, pages 125–129. John Wiley & Sons, 1994.

[van Dongen and Bowen, 2000] 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'00)*, 2000.

[Wallace and Freuder, 1992] 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.

[Wallace, 1993] R.J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In R. Bajcsy, editor, *IJCAI'93*, pages 239–245, 1993.

Appendices

Appendix A

Mathematical Background

This chapter presents the mathematical background that is required for the average time-complexity analysis in the previous chapters. The presentation is at the level of a computer-science undergraduate. It has been included to make this paper fully self-contained.

The contents of this chapter cover basic properties of geometric sequences, Bernoulli's Inequality, Newton's Binomial Theorem, and basic counting techniques.

Lemma 5 (Cardinality of $\mathbb{M}^{ab}$) *Let a and b be positive integers, then the cardinality of $\mathbb{M}^{ab}$ is given by 2^{ab}.*

Proof. For every entry in an a by b matrix there are two possible values. Since there are ab different ordered entries it follows that the total number of all a by b zero-one matrices is given by 2^{ab}. □

It is an immediate consequence of Lemma (5) that the number of non-zero a by b matrices is $2^{ab} - 1$.

A sequence of the form $\{ b_i \}_{i \in \mathbb{N}}$ is called a *geometric sequence* if b_i / b_{i+1} is constant for each $i \in \mathbb{N}$. The following relates geometric series and the sum of their first members.

Theorem 6 (Geometric Sequence) *Let $\{ b_i \}_{i \in \mathbb{N}}$ be a geometric sequence, n a non-negative integer, $a := b_0$ and $d := b_i / b_{i+1}$. If $|d| \neq 1$ then*

$$\sum_{r=0}^{n} b_i = a(1 - d^{n+1})/(1 - d). \tag{A.1}$$

Proof. By induction on n. Let $\mathcal{P}(n)$ be true iff Equation (A.1) is true for given n. If $n = 0$ then $\sum_{r=0}^{n} ad^r = a = a(1 - d^{0+1})/(1 - d)$. Therefore, $\mathcal{P}(0)$ holds. Assume that $\mathcal{P}(n)$ is true for some non-negative integer n, then

$$\begin{aligned}
\sum_{r=0}^{n+1} ad^r &= ad^{n+1} + \sum_{r=0}^{n} ad^r \\
&= ad^{n+1} + a(1 - d^{n+1})/(1 - d) \\
&= a(d^{n+1} - d^{n+2})/(1 - d) + a(1 - d^{n+1})/(1 - d) \\
&= a(1 - d^{n+2})/(1 - d),
\end{aligned}$$

i.e. $\mathcal{P}(n) \implies \mathcal{P}(n+1)$.

We have shown that $\mathcal{P}(0)$ is true. Furthermore, we have shown that for every non-negative integer n it follows that if $\mathcal{P}(n)$ is true then so must $\mathcal{P}(n+1)$. Together, the two can be used to show that $\mathcal{P}(0)$, that $\mathcal{P}(0+1)$, that $\mathcal{P}(0+1+1)$, that .... In other words, Equation (A.1) is true for every integer n. □

Definition 12 (Factorial) *Let* k *be a non-negative integer. The* factorial *of* k *is the number of permutations of* k *different objects. The factorial of* k *is denoted by* $k!$*.*

It can be shown that $k! = \prod_{i=1}^{k} i$. The proof relies on the fact that the number of permutations of zero objects is one and that the number of permutations of k different objects is k times the number of permutations of $k-1$ objects if $k > 0$.

Definition 13 (Binomial Coefficient) *Let* n *and* k *be positive integers s.t.* $n \geq k$*. The* binomial coefficient *of* n *and* k *is the number of selections of* k *unordered objects out of* n *different objects, i.e. the binomial coefficient of* n *and* k *is* $|\{ S \subseteq \{1, \ldots, n\} \,|\, |S| = k\}|$*. The binomial coefficient of* n *and* k *is denoted by* $\binom{n}{k}$*.*

It is not difficult to show that $\binom{n}{k} = n!/(k! \times (n-k)!)$. The proof is as follows. The number of *ordered* sequences of length k from n different objects is $n \times (n-1) \times \ldots (n-k+1)$, i.e. $n!/(n-k)!$. Since we are looking for the number of *unordered* selections, we have to divide $n!/(n-k)!$ by the number of permutations of k objects, which ends the proof.

The following well known result is known as Newton's Binomial Theorem.

Theorem 7 (Binomial Theorem) *Let* n *be any integer, then*

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}. \tag{A.2}$$

Proof. It is not too difficult to see that $(x+y)^n$ is of the form $\sum_{k=0}^{n} c_k x^k y^{n-k}$. One way of viewing $(x+y)^n$ is as a sum of products, where the products are all the 2^n possible sequences of length n s.t. each sequence contains xs and ys only. For example, $(x+y)^2$ is the sum of the members in the sequence xx, xy, yx, yy. For every non-negative $k \leq n$ there are exactly $\binom{n}{k}$ sequences where the number of xs is exactly k. Each and only each such sequence contributes $x^k y^{n-k}$ to the term $c_k x^k y^{n-k}$. Since there are $\binom{n}{k}$ such sequences, the coefficient c_k and $\binom{n}{k}$ are the same for every non-negative $k \leq n$. □

The following result—we shall only need the integer version—dates back to about the same time as Newton's Binomial Theorem.

Theorem 8 (Bernoulli's Inequality) *Let* $x \geq -1$ *be any real and* a *a positive integer, then*

$$(1+x)^a \geq 1 + ax. \tag{A.3}$$

Proof. By induction on a. Let $\mathcal{P}(a)$ be true iff Equation (A.3) is true for a. If $a = 1$ then

$$\begin{aligned}(1+x)^a &= (1+x)^1 \\ &= 1+1\times x \\ &= 1+a\times x \\ &\geq 1+ax.\end{aligned}$$

Therefore, $\mathcal{P}(1)$ is true. Assume that $\mathcal{P}(a)$ holds for some positive integer a, then

$$\begin{aligned}(1+x)^{a+1} &= (1+x)\times(1+x)^a \\ &\geq (1+x)\times(1+ax) \\ &= 1+ax+x+ax^2 \\ &= 1+(a+1)x+ax^2 \\ &\geq 1+(a+1)x.\end{aligned}$$

In other words, $\mathcal{P}(a) \implies \mathcal{P}(a+1)$. We have shown that $\mathcal{P}(1)$. Furthermore we have shown that for every positive a it is true that if $\mathcal{P}(a)$ is true then so is $\mathcal{P}(a+1)$. Therefore, we can show that Equation (A.3) is true for every positive integer a. □

The following is a direct consequence of the Bernoulli Inequality.

Corollary 3 (Bernoulli Inequality) *Let $x \geq -1$ be real and s and a be non-negative integers, s.t. $0 \leq s < a$, then*

$$(1+x)^a \geq (1+x)^s(1+(a-s)x).$$

Proof. Trivial. □

The following lemma will also prove its usefulness later on.

Lemma 6 *Let b be a positive integer. Then*

$$\sum_{c=1}^{b} c2^c = 2+(b-1)2^{b+1}.$$

Proof. Let $f(b) = \sum_{c=1}^{b} c2^c$ and $g(b) = 2+(b-1)2^{b+1}$. We have to prove that $f(b) = g(b)$ for all positive integers b. We have $f(1) = g(1) = 2$. For every positive b we have:

$$\begin{aligned}&f(b+1) - f(b) \\ &= (b+1)2^{b+1} \\ &= b2^{b+1} + 2^{b+1} \\ &= b2^{b+2} - (b-1)2^{b+1} \\ &= 2+((b+1)-1)2^{(b+1)-1} - (2+(b-1)2^{b+1}) \\ &= g(b+1) - g(b).\end{aligned}$$

Since $f(1) = g(1)$ and $f(b+1) - f(b) = g(b+1) - g(b)$ for every $b > 1$ we can prove by induction on b that $f(b) = g(b)$ for all $b > 0$. □