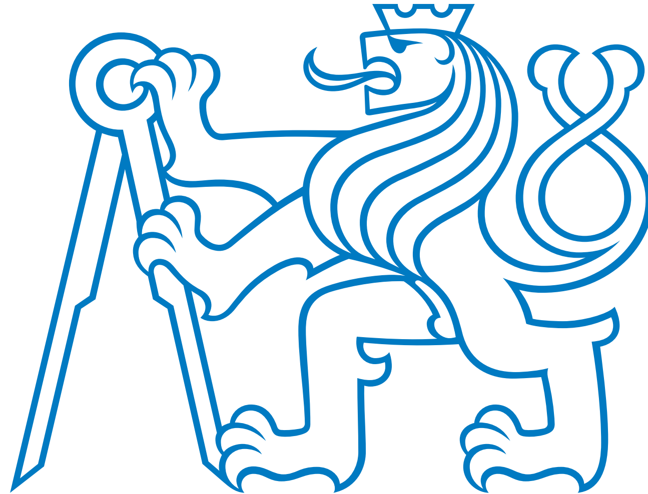
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science



MASTER THESIS

Minimizing Convex Piecewise-Affine Functions by Local Consistency Techniques

Tomáš Dlask

Supervisor: Doc. Ing. Tomáš Werner, PhD.

May 2018

Author Statement

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 21st May 2018 ....

signature

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Dlask** Jméno: **Tomáš** Osobní číslo: **420114**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Minimalizace konvexních po částech afinních funkcí pomocí lokálních konzistencí

Název diplomové práce anglicky:

Pokyny pro vypracování:

1. Prostudujte algoritmy na minimalizaci horní meze na optimální hodnotu VCSP (Valued Constraint Satisfaction Problem) pomocí technik lokální konzistence [2], [3], [4].
2. Prostudujte zobecnění jednoho z těchto algoritmů (max-plus difúze [3, xVI]) na minimalizaci obecných konvexních po částech afinních funkcí [1].
3. Navrhněte podobně zobecnění algoritmu Augmenting DAG [3, xVII] / Virtual Arc Consistency [2].
4. Dokažte správnost algoritmu. Určete složitost iterace algoritmu a diskutujte složitost algoritmu.
5. Implementujte algoritmus v jazyce C/C++. Výstupem bude knihovna, která bude kromě řešení instance problému podporovat i rychlé přepočítání optima po malé změně úlohy.
6. Algoritmus otestujte na instancích VCSP (příp. jiných problémů doporučených vedoucím).

Práce má výzkumný charakter. Ve spolupráci s vedoucím diskutujte a řeštee potíže, které se při vývoji algoritmu vyskytnou.

Seznam doporučené literatury:

T Werner. On Coordinate Minimization of Piecewise-Ane Functions. Research report CTU-CMP-2017-05.
2. MC Cooper, S de Givry, M Sanchez, T Schiex, M Zytnicki, TWerner. Soft Arc Consistency Revisited. Articial Intelligence 174(7-8):449-478, 2010.
3. TWerner. A Linear Programming Approach to Max-sum Problem: A Review. IEEE Trans. on Pattern Recognition and Machine Intelligence (PAMI) 29(7), 2007.
4. T Werner. A Linear Programming Approach to Max-sum Problem: A Review. Research report CTU-CMP-2005-25.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Tomáš Werner, Ph.D., Strojové učení FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **18.12.2017** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce: **30.09.2019**

doc. Ing. Tomáš Werner, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

© ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Abstrakt

Cílem této práce je vytvořit algoritmus pro minimalizaci konvexních po částech afinních funkcí zadaných jako součet maxim afinních funkcí, přičemž se zaměřujeme na řídké instance velkého rozsahu. K dosáhnutí tohoto cíle zobecňujeme algoritmus Augmenting DAG, který minimalizuje horní mez max-plus problému, a používáme podmínku lokální konzistence, která je relaxací podmínky pro globální optimalitu. V této práci představujeme minimalizační algoritmus a teorii s ním spojenou, dokazujeme správnost tohoto algoritmu a navrhujeme jeho verzi pro celočíselnou aritmetiku, kterou jsme implementovali v C++. Poté při testování na velkých řídkých instancích experimentálně ukazujeme, že náš algoritmus je schopen dosáhnout výsledků, které jsou blízko optima.

Klíčová slova: optimalizace nediferencovatelných funkcí, konvexní po částech afinní funkce, lokální konzistence, binární max-plus problém

Abstract

The aim of this work is to develop an algorithm minimizing convex piecewise-affine functions given in the form of a sum of pointwise maxima of affine functions. To make this possible for very large sparse instances, we generalize the Augmenting DAG algorithm (previously proposed to minimize an upper bound on the max-sum problem) and use the notion of local consistency, which relaxes the global optimality condition. We develop the algorithm, the related theory, prove its correctness, and propose a version using integer arithmetic which we implement in C++. We then experimentally show that the algorithm can provide near-optimal results, when tested on sparse large instances.

Keywords: non-differentiable optimization, convex piecewise-affine function, local consistency, binary max-sum problem

Acknowledgements

I would like to thank my supervisor doc. Ing. Tomáš Werner, PhD. for his guidance, frequent consultations and useful advice concerning the thesis. I also wish to express my sincere gratitude to my family and friends for their unceasing support throughout my studies.

Contents

Introduction **1**

1 Max-sum Problem **5**

1.1 Linear Programming Relaxation 6
1.2 Equivalent Max-sum Problems 7
1.3 Upper Bound 8
1.4 Tightness of the Bound 10
1.5 Minimality of the Upper Bound 11
1.6 Arc Consistency and its Closure 12
1.7 Upper Bound Minimization Algorithms 14
1.7.1 Exact Algorithms 14
1.7.2 Approximate Algorithms 15

2 Convex Piecewise-Affine Functions and Local Consistency **17**

2.1 Convex Piecewise-Affine Functions 17
2.1.1 Transformation from a Binary Max-sum Problem 18
2.2 Local Consistency 20
2.2.1 Local Consistency for Minimizing MAF 20
2.2.2 Local Consistency for Minimizing SMAF 28

3 Minimization Algorithm **31**

3.1 Description 31
3.1.1 Inputs and Variables 31
3.1.2 Local Consistency Algorithm 32
3.1.3 Finding Decreasing Direction 34
3.1.4 Line Search 38
3.1.5 Update of $\mathbf{x}$ 44
3.1.6 The Whole Algorithm 46

3.1.7 Using ϵ-consistency 46
3.2 Correctness 47
3.2.1 Correctness of Local Consistency Algorithm 47
3.2.2 Correctness of Decreasing Direction 49
3.2.3 Correctness of Line Search 52
3.2.4 Finite Number of Iterations 55
3.3 Integer Version of the Algorithm 60
3.3.1 Changes in the Algorithm 61
3.3.2 Assuring Consistency and Dealing with Zero Step Size 62
3.3.3 Optimality Criteria 64

4 Implementation and Complexity **67**
4.1 Implementation Details 67
4.1.1 Storing Input Data 67
4.1.2 Efficiency Improvements 68
4.2 Time and Space Complexity 70
4.2.1 Space Complexity 70
4.2.2 Time Complexity 70
4.3 Interface 73
4.3.1 Input and Output File Formats 73
4.3.2 The Code and its Usage 74

5 Experiments **76**
5.1 Two-dimensional Grammars 76
5.1.1 Examples of Two-dimensional Grammars 77
5.2 Tested Instances 81
5.3 Results 83
5.3.1 Lines Grammar 83
5.3.2 Rectangles Grammar 85
5.3.3 Pi Grammar 85
5.3.4 Curve Grammar 87
5.3.5 Small Changes 91
5.4 Further Analysis 92
5.4.1 Dependence on Initial Value of ϵ 92
5.4.2 Typical Progress of Objective Value 93

6 Conclusion and Future Research **99**

6.1 Conclusion 99

6.2 Future Research 100

A Attached Files **106**

List of Tables

2.1 Assignments performed in Example 2.3. 26

2.2 Run of Algorithm 4 on Example 2.3. 27

3.1 Global variables of the algorithm. 32

5.1 Vertical relations between labels in *lines* grammar. 79

5.2 Horizontal relations between labels in *lines* grammar. 79

5.3 Sizes of the individual tested instances. 82

5.4 Overview of all the numerical results and runtimes on all instances. . 90

List of Figures

1.1 Example of a binary max-sum problem. 6
1.2 Equivalent problem to the previous example. 8
1.3 The CSP corresponding to Figure 1.1. 11
1.4 The CSP corresponding to Figure 1.2. 11

2.1 The DAG corresponding to the run of the CSP solving procedure in Example 2.3. 27

3.1 Example case of searching the initial value for t by first non-decreasing hit strategy. 40
3.2 Comparison of initialization methods of t. 40
3.3 Example of enforcing that no cluster maximum increases with step size t. 42
3.4 Flowchart on deciding optimality. 66

4.1 Processing coordinates in queue. 71
4.2 Input file format. 73
4.3 Output file format. 74

5.1 Examples of images, first two are generated by the *lines* grammar. . . 78
5.2 Examples of labelled images generated by various grammars. 80
5.3 Examples of images generated by various grammars. 81
5.4 Input images and solutions for the *lines* grammar. 84
5.5 Input images and solutions for the *rectangles* grammar. 86
5.6 Input images and solutions for the *pi* grammar. 88
5.7 Input images and solutions for the *curve* grammar. 89
5.8 Results corresponding to modified instances. 95
5.9 Comparison of results for various initialization values of ϵ. 96
5.10 Progress of minimization, instance `lines200_high`. 97
5.11 Progress of minimization, instance `rect100sharp_med`. 98

Used notation

Symbol	Meaning
$\mathbb{R}_{-\infty}$	the set of all real numbers and minus infinity, $\mathbb{R}_{-\infty} = \mathbb{R} \cup \{-\infty\}$
$\binom{A}{2}$	the set of all 2-element subsets of set A
$\boldsymbol{g}, \mathbf{x}, \boldsymbol{\alpha}$	vectors
$[m]$	the set $\{1, 2, ..., m\}$
$[a, b]$	closed interval (includes both a and b), i.e. $\{x \mid a \leq x \leq b\}$
$[a, b)$	half-open interval (includes a but not b), i.e. $\{x \mid a \leq x < b\}$
$[\![P]\!]$	Iverson bracket (evaluates to 1 if P is true, otherwise evaluates to 0)
conv A	convex hull of the set A, $A \subseteq \mathbb{R}^n$
$\sum_{i\in[n]} A_i$	Minkowski sum of sets $A_i \subseteq \mathbb{R}^n$
inf A	infimum of a set $A \subseteq \mathbb{R}$ ($\inf \emptyset = \infty$, for a finite set A: $\inf A = \min A$)
a mod b	modulo function, assumes $a \in \mathbb{Z}_0^+$, $b \in \mathbb{N}$

Text Flow

Symbol	Meaning
□	end of an example
■	end of a proof

Used abbreviations

Abbreviation	Full name
AC	Arc Consistency
API	Application Programming Interface
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
FH	First Hit strategy
FIH	First-increasing Hit strategy
FNDH	First Non-decreasing Hit strategy
ILP	Integer Linear Program
LHS	Left-hand Side
LP	Linear Program
MAF	Maximum of Affine Functions
RHS	Right-hand Side
SMAF	Sum of Maxima of Affine Functions
WLOG	Without Loss of Generality
w.r.t.	With Respect to

Introduction

Motivation and Related Work

We consider the problem of minimizing a convex piecewise-affine function, given either as the point-wise maximum of affine function (which we abbreviate as MAF) or as the sum of point-wise maxima of affine functions (SMAF). This problem can be formulated as a linear program (LP), therefore it can be solved in polynomial time. However, for very large sparse instances, on which we focus, solving this LP is in practice impossible because the space and time complexity of general LP solvers is highly superlinear and thus prohibitive. It does not help that the problem is sparse because these methods do not maintain sparsity during their run, therefore large enough instances simply do not fit in the computer memory. Alternatively, minimizing MAF or SMAF can be seen as an instance of convex non-differentiable minimization and thus one could apply subgradient methods which have linear space complexity – however, these methods have been experimentally observed to be very slow.

A special example of sparse large-scale convex piecewise-affine minimization is the minimization of an upper bound on the max-sum labelling problem. In the binary form of this problem, we aim to minimize a sum of unary and binary functions of discrete variables. One powerful approach to this NP-hard combinatorial optimization problem is LP relaxation. The dual of this LP relaxation leads to minimizing an upper bound on the true optimum by equivalent transformations of the problem, which in fact means minimizing a special form of MAF or SMAF.

A number of algorithms to minimize the above mentioned upper bound have been proposed. These algorithms could be divided into two groups – exact (resp. global) algorithms and approximate (resp. local) algorithms. The exact algorithms always find the true optimum. Those are for example the LP solving methods, subgradient methods or smoothing methods. The approximate algorithms do not find a global minimum of the upper bound, but only a local minimum, where "locality" is not meant as usually with respect to the Euclidean metric, but has a different meaning. Max-sum diffusion, which is described in Kovalevsky and Koval (approx. 1975) and listed in Werner (2007), is an example of a simple approximate algorithm. There are multiple algorithms based on it, e.g. TRW-S from Kolmogorov (2006). The approximate minimization is also performed in the Augmenting DAG algorithm

that was introduced in Koval' and Schlesinger (1976) or in Virtual Arc Consistency algorithm from Cooper et al. (2010).

These approximate algorithms were observed by Kappes et al. (2013) to be significantly faster than the global methods and be able to find local minima not far from the global ones for sparse instances. This is not surprising because it has been proved in Průša and Werner (2015) and Průša and Werner (2017) that finding a global minimum of the upper bound for a binary max-sum problems with 3 or more labels is not easier than solving a general linear program.

The local minima found by the local methods are of the same nature – they are local with respect to separately changing the individual variables or blocks of variables. The fact that these algorithms may not find a global minimum corresponds with the well-known fact that coordinate-wise minimization may not find a global minimum of a convex non-differentiable function. However, naively applying coordinate-wise minimization to the upper bound leads to very poor local minima, whereas the algorithms above are its clever modifications that usually yield good local minima.

The local minima found by the approximate algorithms are closely related to local consistencies or constraint propagation in CSP, in particular to arc consistency. These terms are explained for example in Bessiere (2006).

One can naturally ask whether the above local methods can be generalized from minimizing the upper bound (i.e. a special form of MAF or SMAF) to minimizing arbitrary MAF or SMAF. This has been outlined in Werner (2017) for max-sum diffusion. Here, the concept of arc consistency in the CSP was modified to a suitable local consistency in MAF. This in fact corresponds to the sign relaxation of the global optimality condition for MAF, which says that the subdifferential at the considered point must contain the zero vector. Then, max-sum diffusion was generalized to a simple algorithm that finds a local minimum of an arbitrary MAF.

Contribution

Our aim in this thesis is to generalize the Augmenting DAG algorithm for arbitrary convex-piecewise affine function in the form of MAF or SMAF. To do that, we first focus on the MAF form – we show that if a point is not locally consistent in the sense of Werner (2017), we can efficiently (faster than by solving a linear program) find an improving direction in which the function decreases. Then, we can perform a step in this direction and repeat this process iteratively until a locally consistent point is reached.

In more detail, we focus on the more general form, SMAF. We use a similar notion of local consistency and propose an algorithm that finds a decreasing direction

of the function at any point that is not locally consistent and also provide a fast method of line search.

To provide better results in terms of both the final function value and runtime, we introduce a relaxed notion of local consistency, called local ϵ-consistency. This relaxation improves the results and also allows us to prove the correctness of the algorithm.

Additionally, we introduce a version of the algorithm that uses only integer arithmetic and present theorems concerning the optimality of its results. The implementation of this version in C++ is enclosed and tested on instances generated using two-dimensional grammars, experimentally showing the capability to reach near-optimal results. Furthermore, the runtime of our algorithm was significantly lower than the runtime of the chosen LP solver on each tested instance.

Structure

In Chapter 1, we define the binary max-sum problem and the related theory, namely what is the upper bound on quality of such problem, how it can be minimized and in what way is the upper bound connected to arc consistency. We also survey existing algorithms that minimize the upper bound of such problem.

Chapter 2 presents the convex piecewise-affine functions and the forms with which we will deal. We show that the upper bound of a binary max-sum problem can be viewed as a convex piecewise-affine function and we also show multiple transformation algorithms. This chapter also introduces the notion of local consistency and in what way it can be used to minimize convex piecewise-affine functions given as either maximum of affine functions or sum of maxima.

Then, Chapter 3 presents our minimization algorithm – we use the previously defined local consistency, which is its core idea and follow with a detailed description of the algorithm. After the algorithm is presented, we also introduce the generalised version with local ϵ-consistency and prove its correctness for real numbers and also show how to deal with issues caused by limited precision of numbers in our computers. We also present theorems concerning the optimality of the results.

In Chapter 4, we discuss how the algorithm can be implemented efficiently along with simple speed-ups, mainly based on pre-calculation and aggregation of values. After the implementation is described, we calculate the asymptotic space complexity of the whole algorithm and asymptotic time complexity of one iteration. We also present the implementation details of the interface – namely the format of the input and output file and the usage of the enclosed implementation.

Chapter 5 introduces two-dimensional grammars and describes how the max-sum problem of finding the nearest image from the grammar can be used to produce large sparse instances of convex piecewise-affine functions. We created such instances

and run our algorithm on them. The results of the algorithm are compared to the results of an LP solver in the sense of both optimality and runtime. It is also shown how the algorithm can recalculate its result after a small change in the input data and we analyse the runtime of the algorithm in detail.

The thesis is concluded in Chapter 6, where we also summarise the possible areas of further research that were encountered during the work.

Chapter 1

Max-sum Problem

This chapter defines the binary max-sum problem and presents the basic terms that are related to it followed by the description of algorithms solving it – this first section is primarily based on Werner (2007) and Werner (2005), which summarize the previous work based on Schlesinger (1976). The term max-sum problem is also known as Weighted CSP or Valued CSP and it is used so e.g. in Cooper et al. (2010).

Definition 1.1. The *binary max-sum labelling problem* is defined by a tuple $(G, X, \boldsymbol{g})$, where

- $G = (T, E)$ is an undirected graph with *objects* T and *object pairs*[1] $E \subseteq \binom{T}{2}$,
- X is a finite set of *labels*,
- $\boldsymbol{g}$ is a vector consisting of elements $g_t(x) \in \mathbb{R}_{-\infty}$ for each $t \in T$ and $x \in X$ and elements $g_{t,t'}(x, x') \in \mathbb{R}_{-\infty}$ for each pair $\{t, t'\} \in E$ and labels $x, x' \in X$. For all pairs, it holds that $g_{t,t'}(x, x') = g_{t',t}(x', x)$.

The task of a max-sum labelling problem is to assign each object $t \in T$ a label $x_t \in X$ while maximizing the criterion function

$$F(\boldsymbol{x}|\boldsymbol{g}) = \sum_{t \in T} g_t(x_t) + \sum_{\{t,t'\} \in E} g_{t,t'}(x_t, x_{t'}) \tag{1.1}$$

whose value is called the *quality of a labelling* $\boldsymbol{x} \in X^T$. In other words, we want to maximize

$$\max_{\boldsymbol{x} \in X^T} F(\boldsymbol{x}|\boldsymbol{g}). \tag{1.2}$$

[1]The terms *node* and *edge* are reserved for different concepts, therefore the constituents of the graph G are called differently to avoid ambiguity.

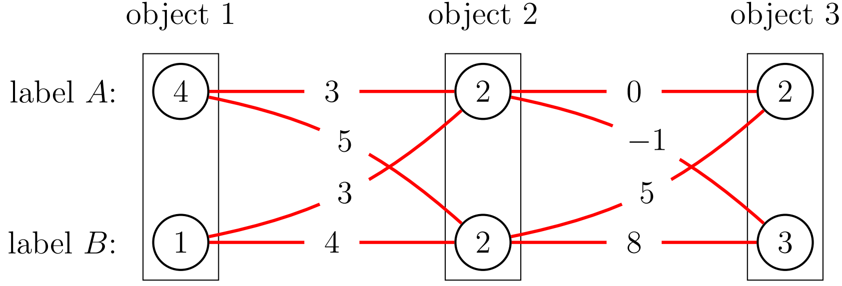


Figure 1.1: Example of a binary max-sum problem.

Definition 1.2. This setting allows us to define another undirected graph $H = (T \times X, E_X)$, where $E_X = \{\{(t,x),(t',x')\} \mid \{t,t'\} \in E, x, x' \in X\}$. The elements of $T \times X$ will be called *nodes* and the elements of E_X *edges*.

Definition 1.3. For a pair of objects $t, t' \in T$ such that $\{t,t'\} \in E$ and a label $x \in X$, the *pencil* (t,t',x) is defined as the set of edges that connect the node (t,x) with nodes (t',x'), where $x' \in X$, i.e.

$$P(t,t',x) = \{\{(t,x),(t',x')\} \mid x' \in X\}. \tag{1.3}$$

Using the constituents of graph H, we can observe that the length of the vector $\boldsymbol{g}$ equals $|I|$, where $I = (T \times X) \cup E_X$, i.e. the vector has $|E| \cdot |X|^2 + |T| \cdot |X|$ components.

Example 1.1. In the Figure 1.1, there is a small instance of a binary max-sum problem with $X = \{A, B\}$, $T = \{1,2,3\}$ and $E = \{\{1,2\},\{2,3\}\}$. The values of the $\boldsymbol{g}$ vector are written in the figure, for example $g_1(A) = 4$, $g_{2,3}(A,B) = -1$. The quality of labelling $\boldsymbol{x} = (B,B,A)$ equals

$$g_1(B) + g_2(B) + g_3(A) + g_{1,2}(B,B) + g_{2,3}(B,A) = 14. \tag{1.4}$$

□

1.1 Linear Programming Relaxation

The above maximization problem 1.2 can be formulated as an ILP whose variables are values $\mu_t(x)$ and $\mu_{t,t'}(x,x')$ that are binary. It is a straightforward idea to relax this condition and create an LP with the same variables that create a *fractional labelling*.

Definition 1.4. The conditions for vector $\boldsymbol{\mu}$, which consists of variables $\mu_t(x)$ and $\mu_{t,t'}(x,x')$, to be a *fractional labelling* are

$$\sum_{x'} \mu_{t,t'}(x,x') = \mu_t(x), \qquad \forall \{t,t'\} \in E,\ \forall x \in X \tag{1.5a}$$

$$\sum_{x} \mu_t(x) = 1, \qquad \forall t \in T \tag{1.5b}$$

$$\boldsymbol{\mu} \geq \mathbf{0}, \tag{1.5c}$$

where $\mu_{t,t'}(x,x') = \mu_{t',t}(x',x)$. The set of all $\boldsymbol{\mu}$ satisfying these conditions will be denoted as $\Lambda_{G,X}$.

If the vector $\boldsymbol{\mu}$ contains only values 0 or 1, then it is a "decided" labelling because there is a corresponding labelling where $x_t = x$ holds if and only if $\mu_t(x) = 1$. It can be easily seen that the $\mu_{t,t'}(x,x')$ values are also binary in this case.

The *relaxed max-sum problem* is the linear program

$$\max_{\boldsymbol{\mu} \in \Lambda_{G,X}} \left[\sum_{\substack{t \in T \\ x \in X}} g_t(x)\, \mu_t(x) + \sum_{\substack{\{t,t'\} \in E \\ x,x' \in X}} g_{t,t'}(x_t, x_{t'})\, \mu_{t,t'}(x_t, x_{t'}) \right], \tag{1.6}$$

where the objective function is in fact the scalar product of vectors $\boldsymbol{\mu}$ and $\boldsymbol{g}$, therefore it can be rewritten into a simpler form

$$\max_{\boldsymbol{\mu} \in \Lambda_{G,X}} \boldsymbol{g}^T \boldsymbol{\mu}. \tag{1.7}$$

1.2 Equivalent Max-sum Problems

Definition 1.5. Two max-sum problems $(G, X, \boldsymbol{g})$ and $(G, X, \boldsymbol{g'})$ are called *equivalent* (denoted as $\boldsymbol{g} \sim \boldsymbol{g'}$) if the corresponding quality functions are identical, that is

$$F(\boldsymbol{x}|\boldsymbol{g}) = F(\boldsymbol{x}|\boldsymbol{g'}), \quad \forall \boldsymbol{x} \in X^T. \tag{1.8}$$

Observe that if we assign a value $\varphi_{t,t'}(x)$ to each pencil (t, t', x) of a given binary max-sum problem $(G, X, \boldsymbol{g})$ and create a new vector $\boldsymbol{g}^{\varphi}$ in the following way:

$$g^{\varphi}_t(x) = g_t(x) + \sum_{\{t,t'\} \in E} \varphi_{t,t'}(x) \tag{1.9a}$$

$$g^{\varphi}_{t,t'}(x,x') = g_{t,t'}(x,x') - \varphi_{t,t'}(x) - \varphi_{t',t}(x') \tag{1.9b}$$

then, $(G, X, \boldsymbol{g})$ and $(G, X, \boldsymbol{g}^{\varphi})$ are equivalent. This can be proven by expressing $F(\boldsymbol{x}|\boldsymbol{g}^{\varphi})$ using $\boldsymbol{g}$ and $\varphi_{t,t'}(x)$ because the $\varphi_{t,t'}(x)$ terms cancel out.

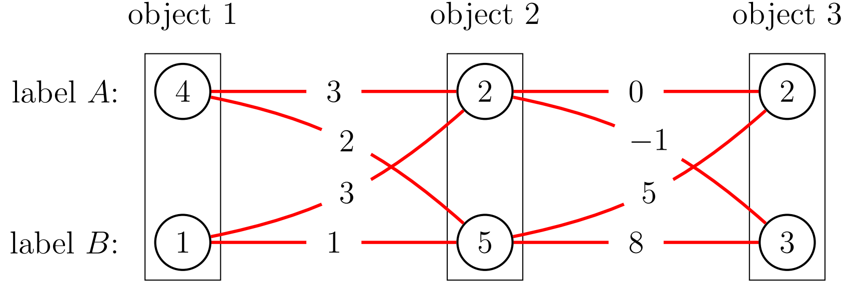


Figure 1.2: Equivalent problem to the previous example.

However, not all problems that are in one equivalence class are necessarily tied to each other by the shown transformation. It holds in general for connected graphs G and vectors $\boldsymbol{g}$ with finite values.

Notice that when we defined the $\boldsymbol{g}$ values, writing either $g_{t,t'}(x, x')$ or $g_{t',t}(x', x)$ was the same, because both were just simpler notations of $g(\{(t, x), (t', x')\})$, i.e. the assigned value to the given edge. On the other hand, $\varphi_{t,t'}(x)$ and $\varphi_{t',t}(x')$ are different variables and can generally have different values.

Example 1.2. The problem shown in Figure 1.2 is equivalent to the one in Figure 1.1. It was obtained using a transformation vector $\boldsymbol{\varphi}$ which contains only zeros, except for $\varphi_{2,1}(B) = 3$. Because of this, the value of the corresponding node and the values of the edges in pencil $P(2, 1, B)$ change, i.e.

$$g_2^{\boldsymbol{\varphi}}(B) = g_2(B) + \varphi_{2,1}(B) + \varphi_{2,3}(B) = 2 + 3 + 0 = 5 \tag{1.10a}$$

$$g_{2,1}^{\boldsymbol{\varphi}}(B, A) = g_{2,1}(B, A) - \varphi_{2,1}(B) - \varphi_{1,2}(A) = 5 - 3 - 0 = 2 \tag{1.10b}$$

$$g_{2,1}^{\boldsymbol{\varphi}}(B, B) = g_{2,1}(B, B) - \varphi_{2,1}(B) - \varphi_{1,2}(B) = 4 - 3 - 0 = 1. \tag{1.10c}$$

□

1.3 Upper Bound

It is easy to formulate an upper bound on the maximum quality of a solution. This upper bound was introduced by Schlesinger (1976) and is therefore sometimes called *Schlesinger's upper bound* or the *height*.

Definition 1.6. The *height* of a binary max-sum problem $(G, X, \boldsymbol{g})$ is defined as

$$U(\boldsymbol{g}) = \sum_{t \in T} u_t + \sum_{\{t,t'\} \in E} u_{t,t'}, \tag{1.11}$$

where

$$u_t = \max_{x \in X} g_t(x), \qquad u_{t,t'} = \max_{\{x,x'\} \in X} g_{t,t'}(x, x'). \tag{1.12}$$

The term u_t is called the *height of object* t and $u_{t,t'}$ is the *height of pair* $\{t,t'\}$.

Proposition 1.1. *The height of a max-sum problem is an upper bound on its quality,*

$$F(\boldsymbol{x}|\boldsymbol{g}) \leq U(\boldsymbol{g}). \tag{1.13}$$

Proof. It is truly an upper bound because $u_t \geq g_t(x)$ for all $t \in T$, $x \in X$ and $u_{t,t'} \geq g_{t,t'}(x,x')$ for all $\{t,t'\} \in E$, $x,x' \in X$ and exactly one node from each object and exactly one edge from each pair is used in any labelling. ■

As opposed to the optimal quality of a binary max-sum problem, the height of a problem is not invariant to equivalent transformations and it is therefore possible to minimize the height by transforming the problem. This can be formulated as the linear program

$$\min_{\boldsymbol{u},\boldsymbol{\varphi}} \sum_{t \in T} u_t + \sum_{\{t,t'\} \in E} u_{t,t'} \tag{1.14a}$$

$$\varphi_{t't}(x) \in \mathbb{R} \tag{1.14b}$$

$$u_t \in \mathbb{R} \tag{1.14c}$$

$$u_{t,t'} \in \mathbb{R} \tag{1.14d}$$

$$u_t \geq g_t(x) + \sum_{\{t,t'\} \in E} \varphi_{t,t'}(x) \tag{1.14e}$$

$$u_{t,t'} \geq g_{t,t'}(x,x') - \varphi_{t,t'}(x) - \varphi_{t',t}(x') \tag{1.14f}$$

which is actually the dual to the LP relaxation (1.7).

To find the minimum height of a problem, we can formulate the criterion of the previous LP also differently, for example

$$(|T| + |E|) \min_{\boldsymbol{\varphi}} \max \big\{ \max_{t \in T} u_t(x), \max_{\{t,t'\} \in E} u_{t,t'}(x,x') \big\}, \tag{1.15}$$

where the constant $(|T| + |E|)$ can be removed from the criterion function during optimization. The criterion can be also written in the form

$$\min_{\boldsymbol{\varphi} \,|\, g^{\boldsymbol{\varphi}}_{t,t'}(x,x') \leq 0} \sum_{t \in T} u_t(x), \tag{1.16}$$

which requires an additional constraint on $g^{\boldsymbol{\varphi}}_{t,t'}(x,x')$ values.

1.4 Tightness of the Bound

Definition 1.7. Given a binary max-sum problem $(G, X, \boldsymbol{g})$, node (t, x) is a *maximal node* if $u_t = g_t(x)$. Likewise, edge $\{(t, x), (t', x')\}$ is a *maximal edge* if $u_{t,t'} = g_{t,t'}(x, x')$.

Whether a node is maximal or not can be encoded into a binary vector $\bar{\boldsymbol{g}}$ by setting

$$\bar{g}_t(x) = [\![g_t(x) = u_t]\!] \tag{1.17a}$$

$$\bar{g}_{t,t'}(x, x') = [\![g_{t,t'}(x, x') = u_{t,t'}]\!]. \tag{1.17b}$$

The binary max-sum problem $(G, X, \bar{\boldsymbol{g}})$ is essentially a constraint satisfaction problem, in which we search for a labelling (or fractional labelling) that uses only those edges and nodes that correspond to ones in the vector $\bar{\boldsymbol{g}}$.

Theorem 1.1. *The CSP* $(G, X, \bar{\boldsymbol{g}})$ *corresponding to a binary max-sum problem* $(G, X, \boldsymbol{g})$ *is satisfiable iff*

$$\max_{\boldsymbol{x} \in X^T} F(\boldsymbol{x}|\boldsymbol{g}) = U(\boldsymbol{g}). \tag{1.18}$$

Proof. To show that the previous claim holds is easy because the satisfiability of the CSP implies that there is a labelling $\boldsymbol{x}$ that uses only the maximal nodes and edges. And for this labelling, it holds that the $\boldsymbol{g}$ value of each used node (resp. edge) corresponds to the maximum in the corresponding object t (resp. pair $\{t, t'\}$) that equals u_t (resp. $u_{t,t'}$), therefore $F(\boldsymbol{x}|\boldsymbol{g}) = U(\boldsymbol{g})$ and the quality of this labelling is the highest because it uses only the maximal edges and maximal nodes.

For the inverse implication, we can choose the labelling $\boldsymbol{x}^*$ that maximizes $F(\boldsymbol{x}^*|\boldsymbol{g})$, which equals $U(\boldsymbol{g})$. So it must necessarily hold that this labelling uses only the maximal nodes and edges (otherwise, its quality would be lower than $U(\boldsymbol{g})$), therefore the corresponding CSP is satisfiable. ∎

Proposition 1.2. *If condition* (1.18) *is satisfied, then the upper bound is minimal.*

Example 1.3. The CSP corresponding to the problem shown in Figure 1.1 is depicted in Figure 1.3. The maximal nodes in the figure are filled and the non-maximal nodes are white. The maximal edges are drawn with full lines, whereas the non-maximal edges are dotted. Because the CSP is satisfiable, it means that there is a labelling that uses only the maximal nodes and edges, therefore the original problem has minimum height. The CSP thus revealed the optimal labelling of the original problem as $\boldsymbol{x} = (A, B, B)$.

On the other hand, the problem shown in Figure 1.2 does not have minimum height, because the corresponding CSP, which is shown in Figure 1.4, is not satisfiable. □

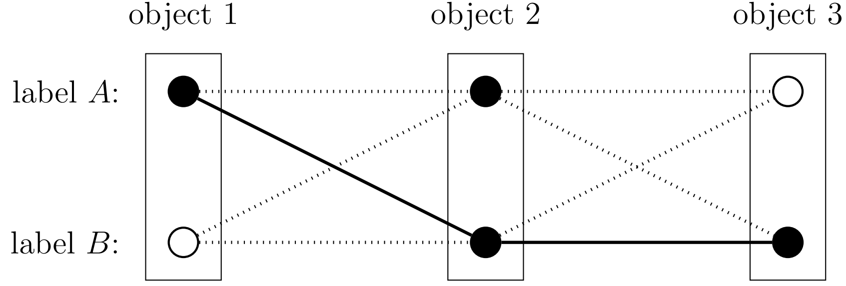


Figure 1.3: The CSP corresponding to Figure 1.1.

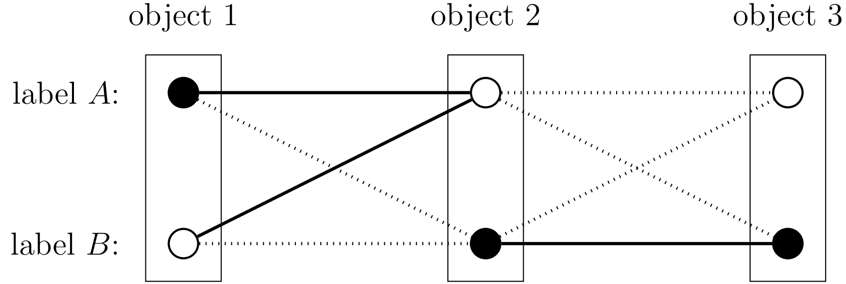


Figure 1.4: The CSP corresponding to Figure 1.2.

1.5 Minimality of the Upper Bound

Due to duality of the corresponding linear programs (1.7) and (1.14), $\boldsymbol{g}^T\boldsymbol{\mu} \leq U(\boldsymbol{g})$. Knowing this and using the complementary slackness condition, it can be seen that $\boldsymbol{g}^T\boldsymbol{\mu} = U(\boldsymbol{g})$ iff

$$\mu_t(x)\,(u_t - g_t(x)) = 0, \qquad \forall t \in T, x \in X \tag{1.19a}$$

$$\mu_{t,t'}(x,x')\,(u_{t,t'} - g_{t,t'}(x,x')) = 0, \qquad \forall \{t,t'\} \in E,\ x,x' \in X. \tag{1.19b}$$

The constraints (1.19) can be reformulated using the $\bar{\boldsymbol{g}}$ vector from (1.17) as

$$\mu_t(x)\,(1 - \bar{g}_t(x)) = 0 \tag{1.20a}$$

$$\mu_{t,t'}(x,x')\,(1 - \bar{g}_{t,t'}(x,x')) = 0. \tag{1.20b}$$

Theorem 1.2. *For a max-sum problem $(G, X, \boldsymbol{g})$ and a fractional labelling $\boldsymbol{\mu} \in \Lambda_{G,X}$, the following statements are equivalent:*

(a) $(G, X, \boldsymbol{g})$ has the minimal height of all its equivalents and $\boldsymbol{\mu}$ has the highest quality.

(b) $\boldsymbol{g}^T\boldsymbol{\mu} = U(\boldsymbol{g})$

(c) $\boldsymbol{\mu}$ is zero on non-maximal nodes and edges.

The relation between *(a)* and *(b)* in Theorem 1.2 follows from strong duality, whereas the relation between *(b)* and *(c)* is given by complementary slackness conditions (1.20).

The set of vectors $\boldsymbol{\alpha}$ satisfying conditions (1.5) and (1.20) will be denoted as

$$\bar{\Lambda}_{G,X}(\bar{\boldsymbol{g}}) = \{\boldsymbol{\mu} \in \Lambda_{G,X} \mid (1 - \bar{\boldsymbol{g}})^T \boldsymbol{\mu} = 0\}. \tag{1.21}$$

Such $\boldsymbol{\mu}$ does not necessarily need to always exist, i.e. the set $\bar{\Lambda}_{G,X}(\bar{\boldsymbol{g}})$ could be empty.

Definition 1.8. We are given a CSP $(G, X, \bar{\boldsymbol{g}})$. This instance is called *relaxed-satisfiable* if $\bar{\Lambda}_{G,X}(\bar{\boldsymbol{g}}) \neq \emptyset$.

Theorem 1.3. *The height of $(G, X, \boldsymbol{g})$ is minimal of all its equivalents if and only if $(G, X, \bar{\boldsymbol{g}})$ is relaxed-satisfiable.*

1.6 Arc Consistency and its Closure

Definition 1.9. We say that a CSP $(G, X, \bar{\boldsymbol{g}})$ is *arc consistent* if

$$\bigvee_{x' \in X} \bar{g}_{t,t'}(x, x') = \bar{g}_t(x), \quad \forall \{t, t'\} \in E,\, x \in X, \tag{1.22}$$

where $\bar{g}_{t,t'}(x, x')$ is viewed as true iff it equals 1 (likewise with $\bar{g}_t(x)$).

The main idea of arc consistency in this case is that if a node (or an edge) has value 1 in the $\bar{\boldsymbol{g}}$ vector, then it can be viewed as allowed to be used in a solution. However, if a node (resp. an edge) has value 1 and all its outgoing edges that are in one pencil have value 0 (resp. one of the nodes that it connects has value 0), then there is no solution that would use this node (resp. edge), so it is actually forbidden to use it even though its corresponding value in the $\bar{\boldsymbol{g}}$ vector is 1. Arc consistent CSPs do not have this issue.

For the following definition, we need to define what is a subproblem of a CSP. Given a CSP $(G, X, \bar{\boldsymbol{g}})$, we say that the CSP $(G, X, \bar{\boldsymbol{g}}')$ is its *subproblem* iff it holds that $\bar{\boldsymbol{g}}' \leq \bar{\boldsymbol{g}}$, where the inequality is evaluated element-wise. It is obvious that a CSP could have multiple subproblems.

Given multiple subproblems $(G, X, \bar{\boldsymbol{g}}^1), \cdots, (G, X, \bar{\boldsymbol{g}}^n)$ of the same CSP, then the *union* of these subproblems is the CSP $(G, X, \bar{\boldsymbol{g}}^U)$, where $\bar{\boldsymbol{g}}^U$ satisfies

$$\bar{g}^U_t(x) = \bigvee_{i=1}^{n} \bar{g}^i_t(x) \qquad \forall t \in T,\ x \in X \tag{1.23a}$$

$$\bar{g}^U_{t,t'}(x, x') = \bigvee_{i=1}^{n} \bar{g}^i_{t,t'}(x, x') \qquad \forall \{t, t'\} \in E,\ x, x' \in X. \tag{1.23b}$$

Definition 1.10. The *AC closure* of a CSP is the union of all its arc consistent subproblems.

Example 1.4. Neither the CSP shown in Figure 1.3 nor the one in Figure 1.4 is arc consistent, but the CSP corresponding to Figure 1.3 has a non-empty AC closure, as opposed to the other CSP, whose AC closure is empty. □

Now, we observed that the AC closure of a CSP can be empty – we say that the AC closure is empty iff for some object (or pair), there is no node (or edge) with its $\bar{\boldsymbol{g}}$ value equal to 1[2]. In this case, it can be easily seen that a vector $\boldsymbol{\mu}$ satisfying conditions (1.5) and (1.20) cannot exist, i.e. $\bar{\Lambda}_{G,X}(\bar{\boldsymbol{g}}) = \emptyset$. This observation allows to formulate Theorem 1.4 whose proof is in Werner (2007).

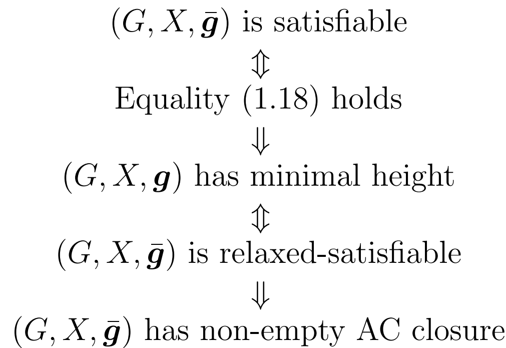
Theorem 1.4. *If a CSP is relaxed-satisfiable, then its AC closure is non-empty.*

Finding out whether the AC closure of a CSP is empty can be done in polynomial time by an arc consistency algorithm. This procedure is derived from the condition on arc consistency (1.22). Initially, we are given a CSP $(G, X, \bar{\boldsymbol{g}})$. Now, we will be changing the vector $\bar{\boldsymbol{g}}$ until the corresponding CSP is arc consistent.

The arc consistency procedure looks for values $\{t, t'\} \in E$, $x \in X$ such that the condition (1.22) is violated. There are only two ways in which it could be violated. The first one is when the disjunction on the LHS evaluates to true and $\bar{g}_t(x)$ is false. In this case, the elements of the disjunction on the LHS will be set to zero, so that the condition is satisfied for this particular pencil. The second case is when the LHS evaluates to false but $\bar{g}_t(x)$ is true – then, we need to set $\bar{g}_t(x)$ to zero. This procedure will always surely terminate after a finite amount of steps because the number of non-zero components of $\bar{\boldsymbol{g}}$ vector always decreases after each step and it is a finite non-negative value.

The Proposition 1.2 along with Theorems 1.1, 1.3 and 1.4 together give the following chain of implications and equivalences.

[2]For connected graphs G, this means $\bar{\boldsymbol{g}} = \mathbf{0}$.



1.7 Upper Bound Minimization Algorithms

Algorithms for upper bound minimization can be divided into two categories – algorithms that find the true minimum of the upper bound (1.14) and algorithms that do not guarantee to find its minimum, but instead find an equivalent transformation such that the corresponding CSP $(G, X, \bar{\boldsymbol{g}}^{\varphi})$ has a non-empty AC closure. By Theorems 1.3 and 1.4, having a non-empty AC closure is a necessary condition for having minimum height.

1.7.1 Exact Algorithms

The exact algorithms guarantee to find the optimum in all cases, however the price for the optimality is their longer runtime. Kappes et al. (2013) experimentally shows that exact methods are significantly slower than approximate algorithms in typical practical applications.

Linear Programming

Of course, the problem of height minimization can be optimally solved by general LP solvers because it is directly given as an LP. Thus, we could for example use the simplex method or interior point methods. However, solving large-scale problems using these methods may need huge memory and could be time-demanding.

Subgradient Methods

Subgradient method of convex function minimization is described e.g. in Boyd et al. (2003). Its core idea is the same as with gradient descent but we do not use the gradient to find a decreasing direction. Instead, we use a subgradient and the subgradient method converges to a global optimum of a convex function that can be non-differentiable. Subgradient methods for minimization of the height of a max-sum problem are used for example by Komodakis et al. (2007) or Schlesinger and Giginjak (2007).

Smoothing Methods

It is also possible to replace the original minimized function by its smooth approximation, this was done for example in Weiss et al. (2012), Johnson et al. (2007) or Ravikumar et al. (2008). The replacement results in a differentiable function which can be minimized by any method of convex differentiable minimization.

1.7.2 Approximate Algorithms

As indicated above, the exact algorithms are more time-demanding and that is the reason why algorithms that enforce only the weaker condition of arc consistency are used. These algorithms can be seen as a special version of coordinate-wise minimization. It is known that the coordinate-wise minimization is guaranteed to converge to a global minimum only if the objective function is convex and differentiable. If it is convex but non-differentiable, it may converge to a local[3] minimum.

Max-sum diffusion and Augmenting DAG algorithms contradict the common wisdom in optimization that coordinate-wise minimization applied to convex non-differentiable functions typically converges to very poor local minima.

Max-sum Diffusion

The procedure of max-sum diffusion was introduced in Kovalevsky and Koval (approx. 1975) and its pseudocode is shown in Algorithm 1. In one step of the algorithm, the $\varphi_{t,t'}(x)$ value is chosen such that the transformed values $g_t^{\boldsymbol{\varphi}}(x)$ and $u_{t,t'}^{\boldsymbol{\varphi}}(x)$ are equal[4]. This step is performed for each pencil of the problem until the values converge.

Algorithm 1: Max-sum diffusion

```
1 while did not converge do
2     for (t, t', x) ∈ P do
3         φ_{t,t'}(x) ← φ_{t,t'}(x) + 1/2 (g_t^φ(x) + max_{x'∈X} g_{t,t'}^φ(x, x'));
```

1 **while** *did not converge* **do**
2   **for** $(t, t', x) \in P$ **do**
3         $\varphi_{t,t'}(x) \leftarrow \varphi_{t,t'}(x) + \frac{1}{2}\left(g_t^{\boldsymbol{\varphi}}(x) + \max_{x' \in X} g_{t,t'}^{\boldsymbol{\varphi}}(x, x')\right)$;

There is no proof of convergence of this algorithm, but it was at least experimentally shown to converge. Max-sum diffusion can be viewed as a form of coordinate descent method because the upper bound is lowered to its minimum with respect to each updated $\varphi_{t,t'}(x)$ at a time. This fact can be easily seen from the form (1.14).

[3]Local is meant not as a topological neighbourhood, but only with respect to movements along the individual coordinates.

[4]Note that $u_{t,t'}^{\boldsymbol{\varphi}}(x) = \max_{x' \in X} g_{t,t'}^{\boldsymbol{\varphi}}(x, x')$.

Augmenting DAG Algorithm

This algorithm uses the graph presented in Definition 1.2, in which each node and edge is assigned an auxiliary variable that initially stores whether the given node or edge is maximal. Then, the arc consistency algorithm is applied to this graph. If a maximal edge has a non-maximal node on its end, then the edge is marked as dead (i.e. it was killed by the non-maximal node). Likewise, if all edges in a pencil from a given maximal node are non-maximal, then the node is marked as dead (i.e. it was killed by the pencil). Whenever a node or an edge is killed, the pointer to the cause is stored, i.e. a dead edge points at one of its end points and a dead node points at the edges that are in the pencil that caused its death. For next iterations of the algorithm, dead nodes and edges are viewed as non-maximal and can cause further deletions. If it happens that in a given object $t^* \in T$, all nodes (t, x), $x \in X$ are either non-maximal or dead, then the arc consistency procedure can be stopped.

Next, the algorithm searches for a direction $\Delta\boldsymbol{\varphi}$ and step size λ so that the equivalent transformation by $\lambda\Delta\boldsymbol{\varphi}$ would decrease the height of the object t^* and would not increase the height of any other object while satisfying constraints for edges $g^{\boldsymbol{\varphi}}_{t,t'}(x, x') \leq 0$[5].

The procedure explained in the previous paragraphs is then iteratively repeated and after each iteration, the value of $\lambda\Delta\boldsymbol{\varphi}$ is accumulated in a vector $\boldsymbol{\varphi}$, i.e.

$$\boldsymbol{\varphi} \leftarrow \boldsymbol{\varphi} + \lambda\Delta\boldsymbol{\varphi}. \tag{1.24}$$

In this way, the Augmenting DAG algorithm gradually decreases the upper bound by equivalent transformations and ends when the arc consistency algorithm cannot mark any edge or node as killed and every object $t \in T$ still has at least one maximal node.

It holds for both Augmenting DAG and max-sum diffusion that after the algorithm ends, if the maxima of $g^{\boldsymbol{\varphi}}_t(x)$ over x are unique for each $t \in T$ and the same holds for the $g^{\boldsymbol{\varphi}}_{t,t'}(x, x')$ values for all $x, x' \in X$ and $\{t, t'\} \in E$, then the optimal labelling can be simply obtained as the maximizing labels of $g^{\boldsymbol{\varphi}}_t(x)$. In such case, we are sure that the found upper bound is minimal.

However, if the maxima are not unique, the result of both algorithms is at least arc consistent, which is a necessary condition for optimality, but not a sufficient one.

[5] This algorithm uses the criterion function in the form (1.16).

Chapter 2

Convex Piecewise-Affine Functions and Local Consistency

In this chapter, two forms of convex piecewise-affine functions are defined, which is followed by algorithms that transform the upper bound minimization of a max-sum problem to the shown forms of functions. After that, we introduce the notion of local consistency, which is first shown in the simpler form and then also for the other form.

2.1 Convex Piecewise-Affine Functions

A convex piecewise-affine function can be given in various forms, one of them is the pointwise maximum of affine functions (MAF), i.e.

$$f(\mathbf{x}) = \max_{i=1}^{m}(\mathbf{a}_i^T\mathbf{x} + b_i) = \max_{i=1}^{m} f_i(\mathbf{x}), \tag{2.1}$$

where $m \in \mathbb{N}$, $b_1, ..., b_m \in \mathbb{R}$, $\mathbf{a}_1, ..., \mathbf{a}_m \in \mathbb{R}^n$, and $\mathbf{x} \in \mathbb{R}^n$. The affine functions f_i are called the *subfunctions* of f. It can be shown that any convex piecewise-affine function can be defined as a MAF.

Another form of convex piecewise affine function that we will use is the sum of pointwise maxima of affine functions (SMAF), i.e.

$$f(\mathbf{x}) = \sum_{i=1}^{l} \max_{j=1}^{m_i}(\mathbf{a}_{i,j}^T\mathbf{x} + b_{i,j}) = \sum_{i=1}^{l} \max_{j=1}^{m_i} f_{i,j}(\mathbf{x}), \tag{2.2}$$

where $\mathbf{m} \in \mathbb{N}^l$ and for each $i \in [l]$, we are given scalars $b_{i,1}, ..., b_{i,m_i} \in \mathbb{R}$ and vectors $\mathbf{a}_{i,1}, ..., \mathbf{a}_{i,m_i} \in \mathbb{R}^n$. The affine functions $f_{i,j}$ are called the *subfunctions* of f. Additionally, we say that the subfunction $f_{i,j}$ belongs to *cluster* i.

2.1.1 Transformation from a Binary Max-sum Problem

If a binary max-sum problem $(G, X, \mathbf{g})$ is given, we can transform its height minimization problem to minimization of a MAF or a SMAF. It can be easily seen from the form (1.9) that $g^{\boldsymbol{\varphi}}_{t,t'}(x, x')$ and $g^{\boldsymbol{\varphi}}_t(x)$ are affine functions of $\boldsymbol{\varphi}$, because all $g_t(x)$ and $g_{t,t'}(x, x')$ are constants.

Viewing height minimization as a general MAF or SMAF allows us to simplify the problem structure. It means that we can for example index the variables in $\mathbf{x}$ just using one number instead of the $\boldsymbol{\varphi}$ variables that are indexed by label, edge and its direction. The structure of the graph G is basically encoded in the vectors $\mathbf{a}$ and the values from $\boldsymbol{g}$ are transformed to $\mathbf{b}$.

Algorithm 2: Transformation algorithm of a binary max-sum problem to MAF

```
1  Function transformToMAF(G, X, g) is
2      a ← 0;
3      b ← 0;
4      i ← 0;
5      for (t, x) ∈ T × X do
6          if g_t(x) > −∞ then
7              i ← i + 1;
8              b_i ← g_t(x);
9              for {t, t'} ∈ E do
10                 a_{i,Φ(φ_{t,t'}(x))} ← 1;
11     for {t, t'} ∈ E do
12         for (x, x') ∈ X^2 do
13             if g_{t,t'}(x, x') > −∞ then
14                 i ← i + 1;
15                 b_i ← g_{t,t'}(x, x');
16                 a_{i,Φ(φ_{t,t'}(x))} ← −1;
17                 a_{i,Φ(φ_{t,'t}(x'))} ← −1;
18     m ← i;
19     return m, a, b;
```

As it was already mentioned in Section 1.3, the problem of height minimization of a binary max-sum problem can be expressed in many forms. The form (1.15) is clearly a MAF minimization.

In the previously mentioned formulation, we minimize the maximum of at most $|T| \cdot |X| + |E| \cdot |X|^2$ elements that correspond to functions. Notice that the functions with corresponding g value equal to $-\infty$ can be disregarded and not added among

the resulting functions. The amount of variables is $2|E| \cdot |X|$.

Algorithm 2 shows how the height is transformed into a MAF. It is assumed that there is a bijection Φ that maps each variable $\varphi_{t,t'}(x')$ for $\{t, t'\} \in E$, $x \in X$ to a number from the set $[2|E| \cdot |X|]$.

Similarly, the original form of the dual LP (1.14) is obviously a SMAF minimization. We minimize the sum of $l = |T| + |E|$ maxima. Again as in the previous case, the functions with corresponding g value equal to $-\infty$ can be disregarded and not added into the set of functions. The amount of variables is again $2|E| \cdot |X|$.

Algorithm 3: Transformation algorithm of a binary max-sum problem to SMAF

```
1  Function transformToSMAF(G, X, g) is
2      a ← 0;
3      b ← 0;
4      m ← 0;
5      l ← |T| + |E|;
6      i ← 0;
7      for t ∈ T do
8          i ← i + 1;
9          j ← 0;
10         for x ∈ X do
11             if g_t(x) > −∞ then
12                 j ← j + 1;
13                 b_{i,j} ← g_t(x);
14                 for {t, t'} ∈ E do
15                     a_{i,j,Φ(φ_{t,t'}(x))} ← 1;
16         m_i ← j;
17     for {t, t'} ∈ E do
18         i ← i + 1;
19         j ← 0;
20         for (x, x') ∈ X^2 do
21             if g_{t,t'}(x, x') > −∞ then
22                 j ← j + 1;
23                 b_{i,j} ← g_{t,t'}(x, x');
24                 a_{i,j,Φ(φ_{t,t'}(x))} ← −1;
25                 a_{i,j,Φ(φ_{t',t}(x'))} ← −1;
26         m_i ← j;
27     return l, m, a, b;
```

Algorithm 3 shows how the upper bound is transformed into a SMAF. It is

again assumed that there is a function Φ with the same characteristics as in the MAF case.

2.2 Local Consistency

Before we introduce the notion of local consistency, we recall the definitions of subgradient and subdifferential. Similarly as gradient can be utilized to identify local extremes of differentiable functions, subdifferential provides the corresponding condition for non-differentiable functions.

Definition 2.1. Vector $\mathbf{v} \in \mathbb{R}^n$ is a *subgradient* of a convex function $f : \mathbb{R}^n \to \mathbb{R}$ at point $\mathbf{x} \in \mathbb{R}^n$ if

$$f(\mathbf{x}') \geq f(\mathbf{x}) + \mathbf{v}^T(\mathbf{x}' - \mathbf{x}) \tag{2.3}$$

holds for all $\mathbf{x}' \in \mathbb{R}^n$.

Definition 2.2. The *subdifferential* of a convex function $f : \mathbb{R}^n \to \mathbb{R}$ at point $\mathbf{x} \in \mathbb{R}^n$ is the set of all its subgradients at $\mathbf{x}$.

The subgradient can be viewed as a generalized gradient, because if f is actually differentiable at some point $\mathbf{x}$, then it has only one subgradient at $\mathbf{x}$, which is the function's gradient at $\mathbf{x}$.

Example 2.1. Let $f(x) = |x|$, then $\partial f(2) = \{1\}$ and $\partial f(0) = [-1, 1]$. □

Example 2.2. Let $f(x_1, x_2) = |x_1| + |x_2|$, then $\partial f(0, 0) = [-1, 1] \times [-1, 1]$. □

It can be easily seen that a point $\mathbf{x} \in \mathbb{R}^n$ is a minimizer of a convex function $f : \mathbb{R}^n \to \mathbb{R}$ if and only if $\mathbf{0} \in \partial f(\mathbf{x})$. That can be checked by substituting $\mathbf{v} = \mathbf{0}$ into the definition of subdifferential, which yields

$$f(\mathbf{x}') \geq f(\mathbf{x}), \forall \mathbf{x}' \in \mathbb{R}^n, \tag{2.4}$$

thus $\mathbf{x}$ is a minimum of f.

2.2.1 Local Consistency for Minimizing MAF

Definition 2.3. Given a MAF $f : \mathbb{R}^n \to \mathbb{R}$, we say that a subfunction f_i, $i \in [m]$ of f is *active at point* $\mathbf{x} \in \mathbb{R}^n$ if $f_i(\mathbf{x}) = f(\mathbf{x})$. Using the notion of active subfunctions, we can define the set of indices of active subfunctions at $\mathbf{x}$ as

$$I(\mathbf{x}) = \{i \in [m] \mid f_i(\mathbf{x}) = f(\mathbf{x})\}. \tag{2.5}$$

Proposition 2.1. *The subdifferential of a MAF $f : \mathbb{R}^n \to \mathbb{R}$ at point $\mathbf{x} \in \mathbb{R}^n$ is*

$$\partial f(\mathbf{x}) = \operatorname{conv}\{\mathbf{a}_i \mid i \in I(\mathbf{x})\}. \tag{2.6}$$

Now, to find out if a point $\mathbf{x} \in \mathbb{R}^n$ is a minimizer of a MAF f, it is enough to determine whether $\mathbf{0} \in \operatorname{conv}\{\mathbf{a}_i \mid i \in I(\mathbf{x})\}$. That is equivalent to the decision whether the LP

$$\sum_{i \in I(\mathbf{x})} \lambda_i = 1 \tag{2.7a}$$

$$\lambda_i \geq 0,\ \forall i \in I(\mathbf{x}) \tag{2.7b}$$

$$\sum_{i \in I(\mathbf{x})} \lambda_i a_{i,j} = 0,\ \forall j \in [n] \tag{2.7c}$$

is satisfiable. The variables $\lambda_i \in \mathbb{R}$, $i \in I(\mathbf{x})$ represent the coefficients of the convex combination. However, calculating this LP could be demanding for large problems, so we will use the sign relaxation introduced in Werner (2017).

The relaxation defines a new variable $\sigma_i \in \{0, 1\}$ to each λ_i as $\sigma_i = \operatorname{sign}(\lambda_i)$. Then, the condition (2.7a) in the previous LP implies that at least one σ_i should be 1. Condition (2.7c) implies that for all $j \in [n]$, if there is $i \in I(\mathbf{x})$ such that $\sigma_i = 1$ and $a_{i,j} > 0$, then there must be $i' \in I(\mathbf{x})$ such that $\sigma_{i'} = 1$ and $a_{i',j} < 0$ (and vice versa). These conditions are formally given as

$$\exists i \in I(\mathbf{x}) : \sigma_i = 1 \tag{2.8a}$$

$$(\exists i \in I(\mathbf{x}) : \sigma_i a_{i,j} > 0) \Leftrightarrow (\exists i' \in I(\mathbf{x}) : \sigma_{i'} a_{i',j} < 0), \forall j \in [n] \tag{2.8b}$$

and they are always satisfiable when the original LP (2.7) is satisfiable, but the opposite does not hold, i.e. when the LP is not satisfiable, then the CSP could be satisfiable.

We can solve the CSP defined by constraints (2.8) by generalized arc consistency. Practically, it means that if for some $j \in [n]$, all $a_{i,j}$, $i \in I(\mathbf{x})$, are non-negative and at least one is positive, then the variables σ_i that correspond to $a_{i,j} > 0$ must be zero. Similar reasoning can be done with the inverse implication of (2.8b). Eventually, there will either be no σ_i that will be forced to zero and we could therefore set at least one variable to 1 to satisfy condition (2.8a). On the other hand, if all σ_i are forced to be zero, the CSP is not satisfiable.

This procedure can be written more formally as follows:

1. For all $i \in I(\mathbf{x})$, set $\sigma_i \leftarrow 1$.
2. If $\exists j \in [n]$ such that $(\exists i \in I(\mathbf{x}) : \sigma_i a_{i,j} > 0) \land (\nexists i' \in I(\mathbf{x}) : \sigma_{i'} a_{i',j} < 0)$, set $\sigma_i \leftarrow 0$.

3. If $\exists j \in [n]$ such that $(\nexists i \in I(\mathbf{x}) : \sigma_i a_{i,j} > 0) \wedge (\exists i' \in I(\mathbf{x}) : \sigma_{i'} a_{i',j} < 0)$, set $\sigma_{i'} \leftarrow 0$.

4. If $\forall j \in [n]$ holds that $(\exists i \in I(\mathbf{x}) : \sigma_i a_{i,j} > 0) \Leftrightarrow (\exists i' \in I(\mathbf{x}) : \sigma_{i'} a_{i',j} < 0)$, terminate. Otherwise go to 2.

Observe that whenever some σ_i is set to zero, it is an assignment which is enforced by the constraint (2.8b), so the previously described procedure will find out whether the CSP is satisfiable – if $\boldsymbol{\sigma} \neq \mathbf{0}$ after it terminates, then the CSP is satisfiable. If $\boldsymbol{\sigma} = \mathbf{0}$, it is not.

The procedure can be viewed as an arc consistency algorithm applied on the CSP. It is known and it was proven in Apt (1999) that such algorithms end always with the same result, which means that if there are multiple values j, i, i' that satisfy the condition in step 1 and 2, we can choose arbitrary one and perform the update of $\boldsymbol{\sigma}$. It was also shown for this particular case in Werner (2017).

Also observe that the procedure must terminate after a finite amount of steps, because the value $\sum_{i \in I(\mathbf{x})} \sigma_i$ can be viewed as its variant, i.e. a variable that monotonously decreases during the run of the procedure. It has initial value $|I(\mathbf{x})|$, which is finite, and in each step, the procedure either terminates or lowers the variant by 1. Eventually, the variant will either decrease to 0 and the procedure terminates, or it ends beforehand.

Definition 2.4. Given a MAF $f : \mathbb{R}^n \to \mathbb{R}$, a point $\mathbf{x} \in \mathbb{R}^n$ is called *locally consistent* with respect to f if the CSP (2.8) is satisfiable.

Decreasing Direction

If the corresponding CSP (2.8) is not satisfiable for given input values determined by a MAF f, then we can conclude that the point $\mathbf{x}$ is not a minimizer and it is possible to find a direction $\Delta\mathbf{x}$, in which the function f decreases from this point. If the CSP is satisfiable, we cannot be sure whether the point $\mathbf{x}$ was a minimizer just from the CSP. But only locally consistent points can be minima of a MAF.

To give the reader some intuition and necessary insight for the next sections, we will show a way of obtaining the direction $\Delta\mathbf{x}$ which decreases a given MAF f from a point $\mathbf{x}$ that is not locally consistent. To find such direction, we will store additional information during the run of the previous procedure. Namely, we will store a pointer $p(i)$ that will be initialized to value ALIVE for each $i \in I(\mathbf{x})$ and if the corresponding σ_i is set to 0, then we set $p(i) \leftarrow j$, where j is the variable that was inconsistent. Next, we will also store the order in which the σ values were set to zero, i.e. create a sequence o that contains the indices i (resp. i') of subfunctions. So, $o(l) = i$ iff $\sum_{i' \in I(\mathbf{x})} \sigma_{i'} = l$ holds right before σ_i is set to 0. This is a downwards

counter – first subfunction whose σ_i is set to 0 satisfies $o(|I(\mathbf{x})|) = i$ and the last subfunction whose σ_i is set to 0 satisfies $o(1) = i$.

For convenience, we will also define an inverse structure o^{-1} such that $o^{-1}(i) = l$ iff $o(l) = i$, which contains the reverse order in which the sigma values were set to zero.

Observe that if the point was not locally consistent, all σ_i values were set to 0 and each σ_i was set to this value exactly once, therefore the p and o values are defined unambiguously. But they might have different values if the procedure was run with a different choice of indices j, i, i'.

Now, we will describe an algorithm that produces a vector $\Delta\mathbf{x}$ such that all the active subfunctions decrease in this direction, that means finding a $\Delta\mathbf{x} \in \mathbb{R}^n$ that satisfies

$$\mathbf{a}_i^T \Delta\mathbf{x} < 0, \quad \forall i \in I(\mathbf{x}). \tag{2.9}$$

This could be again performed by an LP solver, but the time and space complexity of the LP would be too high for large-scale problems, so we use a simpler algorithm. The algorithm will use a directed graph whose nodes will be the active subfunctions and there exists an edge from f_i to $f_{i'}$ iff $a_{i,p(i)} a_{i',p(i)} < 0$.

Theorem 2.1. *The graph is acyclic and the sequence o stores a topological order of its nodes.*

Proof. Only acyclic graphs have a topological order, therefore it is enough to prove that o stores a topological order. We will do so by contradiction.

Assume that there is an edge from node f_s to f_t and $o^{-1}(s) > o^{-1}(t)$. It holds that $a_{s,p(s)} a_{t,p(s)} < 0$ because there is an edge. And it also holds that $\sigma_s \leftarrow 0$ was performed before $\sigma_t \leftarrow 0$, because $o^{-1}(s) > o^{-1}(t)$, so at the time when σ_s was set to zero, $\sigma_t = 1$. We will analyse the situation right before the coordinate $p(s) = j$ was found inconsistent.

If $a_{s,j} > 0$, then the condition in step 2 must have been satisfied. But it could not have been, because there is $i' = t$ with $a_{t,j} < 0$ and $\sigma_t = 1$.

So, necessarily, the second condition must have caused σ_s to be set to zero, i.e. $i' = s$ and $a_{s,j} < 0$. But there is $i = t$ with $a_{t,j} > 0$, thus even the second condition could not have set σ_s to zero. Therefore we conclude that if the sequence o was created in accordance with the previously specified procedure, it is a topological ordering of the graph. ■

It is also important to see that for any $j \in [n]$ it holds that all $a_{i,j}$ with $p(i) = j$ have the same sign. This fact can be proved by contradiction in a very similar manner as the previous proof, where we would base our reasoning on the order of nullifying the corresponding elements of the $\boldsymbol{\sigma}$ vector.

This can be used to formulate the main idea of $\Delta\mathbf{x}$ calculation – if for some coordinate j, we have $a_{i,p(i)} > 0$ for all i that satisfy $p(i) = j$, then we can decrease $x_{p(i)}$ and the corresponding subfunctions with $p(i) = j$ will decrease. Vice versa, if the sign is always negative, we can increase $x_{p(i)}$. Notice that if such changes are performed, only the successors[1] of the subfunctions with the given $p(i)$ value can increase and no previous subfunction (w.r.t. the topological order) increases. And because the graph is acyclic, we can thus arbitrarily decrease the value of any subfunction. In Algorithm 4, we search for $\Delta\mathbf{x}$ such that

$$\mathbf{a}_i^T \Delta\mathbf{x} \leq -1, \forall i \in I(\mathbf{x}), \tag{2.10}$$

so that (2.9) is satisfied. The choice of the constant -1 is arbitrary and choosing a different negative number would only scale the result.

Algorithm 4: Direction calculation for MAF

1. $\Delta\mathbf{x} \leftarrow \mathbf{0}$;
2. **for** $l = 1$ *to* $|I(\mathbf{x})|$ **do**
3. $i \leftarrow o(l)$;
4.   **if** $\mathbf{a}_i^T \Delta\mathbf{x} > -1$ **then**
5. $\Delta x_{p(i)} \leftarrow \Delta x_{p(i)} - (1 + \mathbf{a}_i^T \mathbf{x})/a_{i,p(i)}$;
6. **return** $\Delta\mathbf{x}$;

Algorithm 4 iterates over the active subfunctions in the topological order and if for some subfunction f_i, the inequality is violated, we need to change $\Delta x_{p(i)}$. Eventually, the algorithm returns $\Delta\mathbf{x}$ such that (2.10) is satisfied. To show that this truly holds can be simply done by induction.

Proposition 2.2. *After the l-th subfunction $f_{o(l)}$ is processed by Algorithm 4, it holds that*

$$\mathbf{a}_{o(l')}^T \Delta\mathbf{x} \leq -1, \quad \forall l' \in [l]. \tag{2.11}$$

Proof. When the first subfunction $f_{o(1)}$ is processed, $\Delta\mathbf{x} = 0$ and thus $\mathbf{a}_{o(1)}^T \Delta\mathbf{x} = 0$, so the condition in the loop on line 4 is satisfied and $\Delta x_{p(o(1))}$ is set to

$$-(1 + \mathbf{a}_{o(1)}^T \mathbf{x})/a_{o(1),p(o(1))}, \tag{2.12}$$

and therefore,

$$\mathbf{a}_{o(1)}^T \Delta\mathbf{x} = a_{o(1),p(o(1))} \Delta x_{p(1)} = -a_{o(1),p(o(1))} \cdot (1 + 0)/a_{o(1),p(o(1))} = -1 \leq -1. \tag{2.13}$$

After that, when the $(l+1)$-th subfunction is processed, $l > 0$, it could happen that $\mathbf{a}_i^T \Delta\mathbf{x} \leq -1$, $i = o(l+1)$, and no update is necessary – then, the claim holds

[1] Successors are understood as the end nodes of the edges leading from a specified node.

even for $l+1$. But if the update is necessary, we are decreasing (resp. increasing) $\Delta x_{p(i)}$ if $a_{i,p(i)} > 0$ (resp. $a_{i,p(i)} < 0$). Denoting $\Delta\mathbf{x}^{\text{prev}}$ as the previous value of $\Delta\mathbf{x}$ before the update, we obtain

$$\mathbf{a}_i^T \Delta\mathbf{x} = \mathbf{a}_i^T \Delta\mathbf{x}^{\text{prev}} - a_{i,p(i)}(1 + \mathbf{a}_i^T \mathbf{x}^{\text{prev}})/a_{i,p(i)} = -1 \leq -1, \tag{2.14}$$

which means that the subfunction f_i now satisfies (2.10). It is important to notice that some values $\mathbf{a}_{i'}^T \Delta\mathbf{x}$, $i' \in I(\mathbf{x})$ change their value, i.e. those that satisfy $a_{i',p(i)} \neq 0$. Observe that if the sign of $a_{i',p(i)}$ is the same as the sign of $a_{i,p(i)}$, then the value decreases and therefore, (2.10) definitely still holds for such i' if $o^{-1}(i') < o^{-1}(i)$. On the other hand, if the sign is the opposite, it must hold that $o^{-1}(i') > o^{-1}(i)$, so the subfunction $f_{i'}$ will be processed in future. ■

Because Algorithm 4 processes all active subfunctions, condition (2.10) will hold for all the active subfunctions after it terminates. The inactive subfunctions may increase or decrease, but a small step with size $t > 0$ in the $\Delta\mathbf{x}$ direction can be performed so that no inactive subfunction becomes active, which means that the function value will decrease by at least t.

Ways of calculating the step size will be discussed in detail in Section 3.1.4, where we even deal with the more general case for SMAF.

But for now, it is enough to understand that the previously shown procedure can be used to find a locally consistent point of a MAF if a line search method was provided. We could iteratively repeat the following procedure until such point is reached: first, determine whether a point is locally consistent, then (if it is not) calculate the direction $\Delta\mathbf{x}$ and perform a step in this direction.

Examples

These examples show how the constraints from the CSP (2.8), respectively from the LP (2.7) are enforced and what are the possible outcomes of the arc consistency procedure. These examples assume that there is a MAF f and a point $\mathbf{x}$ which is tested for minimality.

Example 2.3. Let $I(\mathbf{x}) = [5]$, $n = 3$ and $a_{i,j}$ values form the matrix

$$\mathbf{A} = \begin{bmatrix} -1 & 2 & 0 \\ 1 & 0 & 1 \\ -2 & 2 & -1 \\ 0 & 0 & -2 \\ 0 & 0 & -1 \end{bmatrix}. \tag{2.15}$$

Now, let us use the previously described procedure to find out whether the CSP (2.8) is satisfiable. First, we can observe that for $j = 2$, there is $i = 1$ with $a_{1,2}\sigma_1 > 0$, but

Iteration	Condition				Assignments		
	in step	j	i	i'	$\boldsymbol{\sigma}$	$\mathbf{p}$	$\mathbf{o}$
1	2	2	1	–	$\sigma_1 \leftarrow 0$	$p(1) \leftarrow 2$	$o(5) \leftarrow 1$
2	2	2	3	–	$\sigma_3 \leftarrow 0$	$p(3) \leftarrow 2$	$o(4) \leftarrow 3$
3	2	1	2	–	$\sigma_2 \leftarrow 0$	$p(2) \leftarrow 1$	$o(3) \leftarrow 2$
4	3	3	–	4	$\sigma_4 \leftarrow 0$	$p(4) \leftarrow 3$	$o(2) \leftarrow 4$
5	3	3	–	5	$\sigma_5 \leftarrow 0$	$p(5) \leftarrow 3$	$o(1) \leftarrow 5$

Table 2.1: Assignments performed in Example 2.3.

there is no i' with $a_{i',2}\sigma_{i'} < 0$, which means that we set $\sigma_1 \leftarrow 0$, $p(1) \leftarrow 2$, $o(5) \leftarrow 1$. The same can be done with $j = 2$ and $i = 3$, so we set $\sigma_3 \leftarrow 0$, $p(3) \leftarrow 2$, $o(4) \leftarrow 3$. Next, we see that the coordinate $j = 1$ is inconsistent, because with $i = 2$, $a_{2,1}\sigma_2 > 0$, but there is no i' with $a_{i',2}\sigma_{i'} < 0$, so $\sigma_2 \leftarrow 0$, $p(2) \leftarrow 1$, $o(3) \leftarrow 2$. Doing this, the coordinate $j = 3$ became inconsistent, because for $i' = 4$, it holds that $a_{4,3}\sigma_4 < 0$, but there is no i such that $a_{i,3}\sigma_i > 0$, thus $\sigma_4 \leftarrow 0$, $p(4) \leftarrow 3$, $o(2) \leftarrow 4$. Finally, the same is done with $i = 5$ and $j = 3$. The assignments are more clearly shown in Table 2.1.

In this setting, all $\boldsymbol{\sigma}$ were set to zero and the CSP is therefore unsatisfiable. This also implies that the original LP (2.7) is unsatisfiable, so we conclude that the point $\mathbf{x}$ is not a minimum and it is neither locally consistent.

Observe that similar reasoning as with setting σ_i to zero can be also performed directly with the λ_i values, where we would distinguish possibly non-zero λ_i values and λ_i values forced to zero. For example, we can observe from the matrix $\mathbf{A}$ that necessarily $\lambda_1 = 0$ and $\lambda_3 = 0$ because otherwise the second element of the convex combination would be positive (and it needs to be zero from the condition (2.7c)). This reasoning can be followed in the same sense as before but we already know that it would result in unsatisfiability – all λ_i values forced to zero, which violates the condition (2.7a).

Because point $\mathbf{x}$ was not locally consistent, we can find the direction $\Delta\mathbf{x}$ in which all the active subfunctions decrease. We employ Algorithm 4. Initially, $\Delta\mathbf{x}$ is set to $\mathbf{0}$. When the first subfunction f_5 from the topological order is processed, $\mathbf{a}_5^T\Delta\mathbf{x} = 0$, so we set $\Delta x_{p(5)} \leftarrow 1$. Then, we process f_4 and see that $\mathbf{a}_4^T\Delta\mathbf{x} = -2 \leq -1$, so we do not update $\Delta\mathbf{x}$. After that, f_2 is processed and $\mathbf{a}_2^T\Delta\mathbf{x} = 1 > -1$, thus $\Delta x_{p(2)}$ is decreased by 2. Then, we process f_3 and because $\mathbf{a}_3^T\Delta\mathbf{x} = 3 > -1$, we decrease $\Delta x_{p(3)}$ by 2. The last processed subfunction is f_1, which already satisfies $\mathbf{a}_1^T\Delta\mathbf{x} = -2 \leq -1$, so no update is needed and the algorithm terminates.

The iterative development of both $\Delta\mathbf{x}$ and $\mathbf{a}_i^T\Delta\mathbf{x}$ values is in Table 2.2, where we can see that all $\mathbf{a}_i^T\Delta\mathbf{x}$ are eventually lower than 0, so the corresponding subfunctions decrease in the $\Delta\mathbf{x}$ direction. The corresponding directed acyclic graph is shown in Figure 2.1, where it can be obviously seen that $(f_{o(1)}, ..., f_{o(5)}) = (f_5, f_4, f_2, f_3, f_1)$

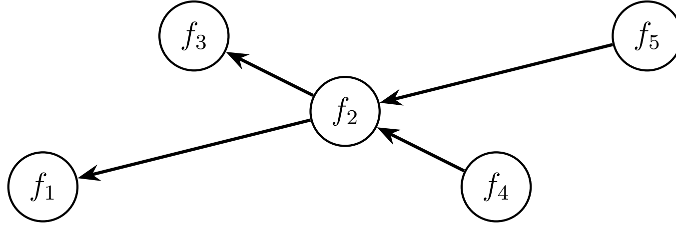


Figure 2.1: The DAG corresponding to the run of the CSP solving procedure in Example 2.3.

$\Delta\mathbf{x}$	$\mathbf{A}\Delta\mathbf{x}$	processed subfunctions
$(0, 0, 0)$	$(0, 0, 0, 0, 0)$	–
$(0, 0, 1)$	$(0, 1, -1, -2, -1)$	5
$(0, 0, 1)$	$(0, 1, -1, -2, -1)$	5, 4
$(-2, 0, 1)$	$(2, -1, 3, -2, -1)$	5, 4, 2
$(-2, -2, 1)$	$(-2, -1, -1, -2, -1)$	5, 4, 2, 3
$(-2, -2, 1)$	$(-2, -1, -1, -2, -1)$	5, 4, 2, 3, 1

Table 2.2: Run of Algorithm 4 on Example 2.3.

is a topological ordering of its nodes.

□

Example 2.4. Let $I(\mathbf{x}) = [3]$, $n = 3$ and $a_{i,j}$ values form the matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 2 & 0 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{bmatrix}. \tag{2.16}$$

We initially set $\boldsymbol{\sigma} \leftarrow 1$ and observe that for $j = 1$ and $i = 3$, there is $a_{3,1}\sigma_3 > 0$, but no i' with $a_{i',1}\sigma_{i'} < 0$, which forces $\sigma_3 \leftarrow 0$. But after setting this value, no more sigma values are changed and we can keep $\sigma_1 = 1$ and $\sigma_2 = 1$, so that the whole CSP (2.8) is satisfied. At this moment, we cannot be sure whether the corresponding LP is satisfiable and it is therefore not known whether the point $\mathbf{x}$ was a minimum.

Solving the LP would give the solution $\boldsymbol{\lambda} = (1/3, 2/3, 0)$, which means that $\mathbf{x}$ was a minimum.

But if the matrix was a little different, namely

$$\mathbf{A} = \begin{bmatrix} 0 & 2 & 2 \\ 0 & -1 & -2 \\ 1 & 0 & 0 \end{bmatrix}, \tag{2.17}$$

then the corresponding CSP would be satisfiable, yet the LP would not be satisfiable and thus $\mathbf{x}$ would not be a minimum. □

2.2.2 Local Consistency for Minimizing SMAF

Now, we will focus on minimizing SMAF using local consistency. First, we will find out what is the subdifferential of a SMAF, for which we need to define a few terms.

If we are given a SMAF $f : \mathbb{R}^n \to \mathbb{R}$, we say that a subfunction $f_{i,j}$ is *active in point* $\mathbf{x} \in \mathbb{R}^n$ if $f_{i,j}(\mathbf{x}) = \max_{j' \in [m_i]} f_{i,j'}(\mathbf{x})$. Then, we can define the set of active subfunctions in cluster i as

$$S_i(\mathbf{x}) = \{j \in [m_i] \mid f_{i,j}(\mathbf{x}) = \max_{j' \in [m_i]} f_{i,j'}(\mathbf{x})\} \tag{2.18}$$

and the set of all active subfunctions as

$$I(\mathbf{x}) = \{(i,j) \mid i \in [l], j \in S_i(\mathbf{x})\}. \tag{2.19}$$

Recall that the *Minkowski sum* of sets $A_1, ..., A_n \subseteq \mathbb{R}^n$ is the set

$$\sum_{i \in [n]} A_i = \left\{ \sum_{i \in [n]} \mathbf{a}_i \mid \mathbf{a}_i \in A_i \right\}. \tag{2.20}$$

It is known that the subdifferential of a function $f : \mathbb{R}^n \to \mathbb{R}$ defined as $f(\mathbf{x}) = \sum_{i \in [l]} f_i(\mathbf{x})$ is the Minkowski sum of the subdifferentials of the convex functions f_i, i.e.

$$\partial f(\mathbf{x}) = \sum_{i \in [l]} \partial f_i(\mathbf{x}). \tag{2.21}$$

Next, we can use the Proposition 2.1 and express the subdifferential of a SMAF $f : \mathbb{R}^n \to \mathbb{R}$ in point $\mathbf{x} \in \mathbb{R}^n$ as

$$\partial f(\mathbf{x}) = \sum_{i \in [l]} \operatorname{conv}\{\mathbf{a}_{i,j} \mid j \in S_i(\mathbf{x})\}. \tag{2.22}$$

Finding out whether a given point $\mathbf{x}$ is a minimum of SMAF f would require to determine the satisfiability of the LP

$$\sum_{j \in S_i(\mathbf{x})} \lambda_{i,j} = 1,\ \forall i \in [l] \tag{2.23a}$$

$$\lambda_{i,j} \geq 0,\ \forall i \in [l], j \in S_i(\mathbf{x}) \tag{2.23b}$$

$$\sum_{i \in [l]} \sum_{j \in S_i(\mathbf{x})} \lambda_{i,j} a_{i,j,k} = 0,\ \forall k \in [n] \tag{2.23c}$$

which just formalizes the condition $\mathbf{0} \in \partial f(\mathbf{x})$.

Observe that we could apply the same approach as in the MAF case and create

a CSP relaxation of this LP with variables $\sigma_{i,j} \in \{0,1\}$. This relaxation is formalized as

$$\forall i \in [l]\, \exists j \in S_i(\mathbf{x}) : \sigma_{i,j} = 1 \tag{2.24a}$$

$$(\exists (i,j) \in I(\mathbf{x}) : \sigma_{i,j} a_{i,j,k} > 0) \Leftrightarrow (\exists (i',j') \in I(\mathbf{x}) : \sigma_{i',j'} a_{i',j',k} < 0), \forall k \in [n] \tag{2.24b}$$

The second condition of the CSP is basically the same as in the MAF case, we treat the subfunctions independently of their cluster and look for inconsistencies in any variable. However, the first condition is different – we do not need to force all $\sigma_{i,j}$ to zero but it is enough to do that for a single cluster i to find out that the CSP is not satisfiable. Similarly as with MAF, we can state a procedure that will determine whether the CSP (2.24) is satisfiable:

1. For all $(i,j) \in I(\mathbf{x})$, set $\sigma_{i,j} \leftarrow 1$.
2. If $\exists k \in [n]$ such that $(\exists (i,j) \in I(\mathbf{x}) : \sigma_{i,j} a_{i,j,k} > 0) \wedge (\nexists (i',j') \in I(\mathbf{x}) : \sigma_{i',j'} a_{i',j',k} < 0)$, set $\sigma_{i,j} \leftarrow 0$.
3. If $\exists k \in [n]$ such that $(\nexists (i,j) \in I(\mathbf{x}) : \sigma_{i,j} a_{i,j,k} > 0) \wedge (\exists (i',j') \in I(\mathbf{x}) : \sigma_{i',j'} a_{i',j',k} < 0)$, set $\sigma_{i',j'} \leftarrow 0$.
4. If $\forall k \in [n]$ holds that $(\exists (i,j) \in I(\mathbf{x}) : \sigma_{i,j} a_{i,j,k} > 0) \Leftrightarrow (\exists (i',j') \in I(\mathbf{x}) : \sigma_{i',j'} a_{i',j',k} < 0)$, terminate.
5. If $\exists i \in [l]$ such that $\forall j \in S_i(\mathbf{x}) : \sigma_{i,j} = 0$, terminate. Otherwise, go to 2.

Definition 2.5. Given a SMAF $f : \mathbb{R}^n \to \mathbb{R}$, a point $\mathbf{x} \in \mathbb{R}^n$ is called *locally consistent* with respect to f if the CSP (2.24) is satisfiable.

It is clear that this procedure finds the solution to the CSP (2.24) whenever one exists and if it does not, it will detect it too. It naturally leads from the definition and the previous reasoning that if for some $i^* \in [l]$, all values $\sigma_{i^*,j}$ are forced to 0, then we can terminate the algorithm and conclude that the point is not locally consistent.

The choice of values i, j, k (resp. also i', j') during the procedure above could influence the final values of $\sigma_{i,j}$, and therefore also the cluster i^* which contains only zero $\sigma_{i^*,j}$ (if the corresponding CSP is not satisfiable). But similarly as the corresponding procedure for MAF, this one also always terminates with the same result in the sense of the CSP satisfiability regardless of the choice of the constants in the existential quantifiers.

The relation of locally consistent points of a SMAF to its minima is the same as in the MAF case. If a point $\mathbf{x}$ is not a locally consistent point of SMAF f, then we can find a direction $\Delta\mathbf{x}$, in which the function f decreases from this point and it

is therefore not a minimizer. On the other hand, if the point is a minimum, then it is necessarily locally consistent. This property will be used and proven as a part of the main minimization algorithm

Chapter 3

Minimization Algorithm

In this chapter, we present the whole algorithm for SMAF minimization in detail, show its generalization for ϵ-consistency and prove its correctness. We also show its modified version for finite-precision arithmetic. Finally, we formulate conditions that can in some cases decide whether the true optimum was reached.

3.1 Description

In this section, we will describe the algorithm based on the notion of local consistency that was outlined in the previous chapter. The task of this algorithm is to find a locally consistent point of a given SMAF f. In contrast to Chapter 2, here we give all the details and a complete pseudocode of the algorithm.

The algorithm works iteratively. In each iteration, it finds out whether the current point $\mathbf{x}$ is locally consistent. If it is, then the algorithm halts and returns this value. If it is not, then we are able to find a direction in which the function decreases and then update $\mathbf{x}$ by performing a step in this direction.

3.1.1 Inputs and Variables

The algorithm takes a SMAF f on its input, which is defined by the values $l \in \mathbb{N}$, $\mathbf{m} \in \mathbb{N}^l$ and for each $i \in [l]$, we have $b_{i,1}, ..., b_{i,m_i} \in \mathbb{R}$ and vectors $\mathbf{a}_{i,1}, ..., \mathbf{a}_{i,m_i} \in \mathbb{R}^n$, as it was shown in formula (2.2). Next, the algorithm uses a few global variables, whose meanings are in Table 3.1. All these mentioned inputs and listed variables are viewed as global and can be accessed in any function. How these variables are initialized can be seen in Algorithm 5.

Variable	Content
$\mathbf{x}$	current point
$\Delta\mathbf{x}$	decreasing direction from the current point
$\mathbf{y}$	current subfunction values, i.e. $y_{i,j} = \mathbf{a}_{i,j}^T\mathbf{x} + b_{i,j}$
$\mathbf{c}$	slopes of the subfunctions w.r.t. $\Delta\mathbf{x}$ direction, i.e. $c_{i,j} = \mathbf{a}_{i,j}^T\Delta\mathbf{x}$
$\mathbf{d}$	in-degree of a node in the augmenting DAG
$\mathbf{r}$	whether a node is in the augmenting DAG
$\mathbf{p}$	the statuses of subfunctions
Q	queue of the coordinates that are suspected of being inconsistent
i^*	index of the cluster in which no subfunction is alive

Table 3.1: Global variables of the algorithm.

Algorithm 5: Initialization of the minimization algorithm

```
1  Function initialize() is
2      Δx ← 0;
3      x ← 0;
4      y ← b;
5      c ← 0;
6      d ← 0;
7      r ← FALSE;
8      p ← INACTIVE;
9      Q ← empty queue;
10     for (i, j) ∈ I(x) do
11         p(i, j) ← ALIVE;
12         for k ∈ N(f_{i,j}) do
13             Q.push(k);
```

3.1.2 Local Consistency Algorithm

The local consistency algorithm calculates whether the current point is locally consistent. Its core idea is the same as in the procedure shown in Section 2.2.2 but it is optimized in the sense that it can partially re-use the results from the previous iteration. The output of the local consistency algorithm is a boolean value (denotes whether the current point is locally consistent) and a structure that allows us to calculate the decreasing direction (in case that the current point is not locally consistent).

Instead of storing the order of subfunctions, pointers to coordinates, σ values and whether the subfunction is active, we can simply keep only the status of each subfunction in the variables $p(i, j)$. The status value can be divided into three equivalence classes. First, the subfunction $f_{i,j}$ can be inactive and thus $p(i, j) =$ INACTIVE.

Second, the subfunction can be active and not yet killed; this status is called ALIVE and corresponds to having $\sigma_{i,j} = 1$ in the previous section. Note that "killing" a subfunction corresponds to setting $\sigma_{i,j} \leftarrow 0$ in previous procedure. The third equivalence class are the statuses of subfunctions that are active, but already killed. In this case, the status contains the index of the corresponding variable that caused the subfunction to be killed (i.e. the variable that was inconsistent), similarly as before.

Definition 3.1. In Algorithm 6, a variable x_k (resp. coordinate k) is called *inconsistent* iff there is at least one $f_{i,j}$ with $p(i,j) = \text{ALIVE}$ and $a_{i,j,k} \neq 0$ and there is no $f_{i',j'}$ with $p(i',j') = \text{ALIVE}$ and $a_{i',j',k}\, a_{i,j,k} < 0$.

Definition 3.2. For a given subfunction $f_{i,j}$, the set $N(f_{i,j})$ is defined as the set of coordinates on which its value depends, i.e.

$$N(f_{i,j}) = \{k \in [n] \mid a_{i,j,k} \neq 0\}. \tag{3.1}$$

Similarly, we can also define the corresponding set for a given variable x_k as

$$N(x_k) = \{(i,j) \mid i \in [l], j \in [m_i], a_{i,j,k} \neq 0\}. \tag{3.2}$$

The previous definition can be viewed as the relation of adjacency on an undirected bipartite graph whose nodes are all the subfunctions $f_{i,j}$ and all variables x_k, where an edge connects node $f_{i,j}$ with node x_k iff $a_{i,j,k} \neq 0$. Then, $N(f_{i,j})$ are the indices of neighbouring variables x_k and vice versa.

Initially, before the calculations start, all the subfunctions are viewed as inactive and before the first iteration, the ones that are active are marked so on line 11 in Algorithm 5 and the queue Q is also filled with possibly inconsistent coordinates on line 13 – in this case, this corresponds to the coordinates on which at least one subfunction depends.

After the initialization is done, we employ Algorithm 6 to find out whether the current point is locally consistent. The algorithm is in its core analogous to the procedure which was in the Section 2.2.2 tailored to SMAF. It means that we are looking for inconsistent coordinates and if one is found, then we mark the corresponding subfunctions that caused the inconsistency with the variable but we can kill multiple subfunctions at a time, as opposed to the previous procedure. Killing subfunctions means that their $p(i,j)$ value is set from ALIVE to the coordinate index.

As we said in Section 2.2.2, the algorithm can be terminated if at some point there is a cluster i in which no subfunction has status ALIVE. In this case, the algorithm returns FALSE, which means that the current point is not locally consistent. For a locally consistent point, the algorithm returns TRUE.

The pseudocode of the algorithm is optimized in the sense that it does not try all coordinates in each iteration of the main while loop, but instead stores the

potentially inconsistent ones in the queue Q. This queue was originally initialized in Algorithm 5 and it will be later periodically re-filled in Algorithm 10. When an inconsistent coordinate is discovered, the queue is again updated so that it still contains all the coordinates that might have become inconsistent.

Algorithm 6: The local consistency algorithm that calculates whether the active functions are consistent.

```
1  Function calculateConsistency() is
2      if ∃i ∈ [l], ∄j ∈ [m_i] : p(i, j) = ALIVE then
3          i* ← i;
4          return FALSE;
5      while Q is not empty do
6          k ← Q.pop();
7          if variable x_k is inconsistent then
8              for (i, j) ∈ N(x_k) do
9                  if p(i, j) = ALIVE then
10                     p(i, j) ← k;
11                     for k' ∈ N(f_{i,j}) do
12                         Q.push(k');
13             if ∃i ∈ [l], ∄j ∈ [m_i] : p(i, j) = ALIVE then
14                 i* ← i;
15                 return FALSE;
16     return TRUE;
```

3.1.3 Finding Decreasing Direction

In this section, we will present an algorithm that finds a direction in which the function f decreases. Throughout this section, it is assumed that Algorithm 6 decided that the current point is not locally consistent. In the opposite case, the consistent point was already reached and we do not need to find a decreasing direction.

We will look for a decreasing direction $\Delta\mathbf{x} \in \mathbb{R}^n$ that satisfies

$$f_{i,j}(\mathbf{x} + \Delta\mathbf{x}) \leq f_{i,j}(\mathbf{x}), \qquad \forall (i,j) \in I(\mathbf{x}), \tag{3.3a}$$

$$f_{i^*,j}(\mathbf{x} + \Delta\mathbf{x}) \leq f_{i^*,j}(\mathbf{x}) - 1, \qquad \forall j \in S_{i^*}(\mathbf{x}), \tag{3.3b}$$

where i^* is the cluster for which there is no subfunction $f_{i^*,j}$ with $p(i^*, j) = \mathsf{ALIVE}$. In Algorithm 7, we will show how to find such $\Delta\mathbf{x}$.

The previously stated condition requires that no active subfunction should increase in the direction and additionally, the subfunctions from cluster i^* should

decrease. The choice of constant -1 in the second inequality is arbitrary and choosing any other negative constant would only scale the problem. Obviously, there are infinitely many $\Delta\mathbf{x} \in \mathbb{R}^n$ that satisfy the conditions and we are interested in finding one of them.

The previous local consistency algorithm fills in the status values $\mathbf{p}$, which define a directed graph whose nodes are the subfunctions $f_{i,j}$ such that $p(i,j) \in [n]$ and there is an edge in the graph from subfunction $f_{i,j}$ to subfunction $f_{i',j'}$ iff $a_{i,j,p(i,j)}a_{i',j',p(i,j)} < 0$. More formally, it is a directed graph $G_\Delta = (N_\Delta, E_\Delta)$, where

$$N_\Delta = \{f_{i,j} \mid p(i,j) \in [n]\}, \tag{3.4a}$$
$$E_\Delta = \{(f_{i,j}, f_{i',j'}) \mid a_{i,j,p(i,j)}a_{i',j',p(i,j)} < 0\}. \tag{3.4b}$$

The meaning of the edges in the graph is based on the idea that all the corresponding $a_{i,j,p(i,j)}$ are non-zero and changing the value of $x_{p(i,j)}$ in one direction can lower the value of $f_{i,j}(\mathbf{x})$. More specifically, if $a_{i,j,p(i,j)} > 0$, then we can decrease the $f_{i,j}(\mathbf{x})$ value by lowering $x_{p(i,j)}$ and if $a_{i,j,p(i,j)} < 0$, then we can do it by increasing $x_{p(i,j)}$. If we change $x_{p(i,j)}$, the value of some other subfunctions $f_{i',j'}$ may increase because their coefficient $a_{i',j',p(i,j)}$ has the opposite sign than $a_{i,j,p(i,j)}$, which is exactly the definition of the edges in the graph.

It will be proven in the next sections that this graph is actually a DAG and it is therefore possible to arbitrarily decrease the value of any $f_{i,j}$ that is its node – we could change $x_{p(i,j)}$ to decrease it arbitrarily. If there are outgoing edges from the corresponding node, then some other subfunctions increased their value and if we do not want that, we can again lower them by their corresponding coordinates stored in $\mathbf{p}$. Eventually, we will decrease the subfunctions whose corresponding nodes in the DAG do not have any outgoing edges, i.e. we will finish the procedure successfully.

Therefore, we can use the DAG to find the direction $\Delta\mathbf{x}$ that satisfies the above mentioned conditions. First, we will identify a subgraph of the DAG, the so-called augmenting DAG. It is defined as all the nodes that are reachable from the nodes that are in the cluster i^* and are active. The edges between these nodes are kept without changes. The formal definition follows.

Definition 3.3. The *augmenting DAG* is a node-induced subgraph G_A of the graph $G_\Delta = (N_\Delta, E_\Delta)$ defined in (3.4). A node $f_{i,j}$ is in the augmenting DAG if and only if there exists $j' \in S_{i^*}(\mathbf{x})$ such that there is a directed path in the graph G_Δ from $f_{i^*,j'}$ to $f_{i,j}$.

Next, we can process the nodes of the augmenting DAG in a topological order and for each corresponding subfunction check whether it satisfies the conditions for $\Delta\mathbf{x}$ and if it does not, update the value of the corresponding status variable. This procedure is shown in Algorithm 7.

Algorithm 7 works exactly as described above. It first finds out which nodes are reachable from the nodes $f_{i^*,j}$, $j \in S_{i^*}(\mathbf{x})$ and if a node is reachable, its $\mathbf{r}$ value is set to TRUE. In this way, we identify which nodes from N_Δ are in the augmenting DAG. Additionally, we calculate the in-degrees of the nodes in the augmenting DAG and accumulate them in the structure $\mathbf{d}$. The exploration of the graph in Algorithm 7 is performed by a recursive function that is described in Algorithm 8.

Next, using these structures, we can simply identify all the nodes with zero in-degree and put them into the queue Q_f – notice that only the nodes from cluster i^* could have zero in-degree in the augmenting DAG. Finally, we traverse the nodes of the augmenting DAG in a topological order and whenever a node is processed, we decrease the in-degree of its successors. If the successor has in-degree equal to zero, it is put into the queue Q_f.

Algorithm 7: Direction calculation for SMAF

```
1  Function calculateDirection() is
2      for j ∈ S_{i*}(x) do
3          if r(f_{i*,j}) = FALSE then
4              r(f_{i*,j}) ← TRUE;
5              explore(f_{i*,j});
6      Q_f ← empty queue of pairs;
7      for j ∈ S_{i*}(x) do
8          if d(f_{i*,j}) = 0 then
9              Q_f.push(i*, j);
10     while Q_f is not empty do
11         (i, j) ← Q_f.pop();
12         r(f_{i,j}) ← FALSE;
13         k ← p(i, j);
14         prev ← Δx_k;
15         if c_{i,j} > −[[i = i*]] then
16             Δx_k ← Δx_k − ([[i = i*]] + c_{i,j})/a_{i,j,k};
17         for (i', j') ∈ N(x_k) do
18             c_{i',j'} ← c_{i',j'} + (Δx_k − prev)a_{i',j',k};
19             if (p(i', j') ≠ INACTIVE) ∧ ((f_{i,j}, f_{i',j'}) ∈ E_Δ) then
20                 d(f_{i',j'}) ← d(f_{i',j'}) − 1;
21                 if d(f_{i',j'}) = 0 then
22                     Q_f.push(i', j');
23                 else if d(f_{i',j'}) < 0 then
24                     d(f_{i',j'}) = 0;
```

While processing a node (i.e. a subfunction), we ensure that the conditions (3.3) are satisfied for it, these conditions are equivalent to

$$\mathbf{a}_{i,j}^T \Delta \mathbf{x} \leq 0, \quad \forall (i,j) \in I(\mathbf{x}) \tag{3.5a}$$

$$\mathbf{a}_{i^*,j}^T \Delta \mathbf{x} \leq -1, \quad \forall j \in S_{i^*}(\mathbf{x}). \tag{3.5b}$$

And these can be rewritten as one condition

$$c_{i,j} \leq -[\![i = i^*]\!], \forall (i,j) \in I(\mathbf{x}), \tag{3.6}$$

where $c_{i,j} = \mathbf{a}_{i,j}^T \Delta \mathbf{x}$. Algorithm 7 stores these $c_{i,j}$ values pre-calculated in order not to evaluate the scalar product every time.

If the condition (3.6) is violated for the currently processed subfunction $f_{i,j}$, then $\Delta x_{p(i,j)}$ is changed in order to satisfy the condition. Then, the values of $c_{i',j'}$ that correspond to subfunctions $f_{i',j'}$ that depend on $x_{p(i,j)}$ need to be updated so that they still contain the correct values $\mathbf{a}_{i',j'}^T \Delta \mathbf{x}$. During this update, the in-degree of successors of $f_{i,j}$ in the augmenting DAG is lowered, as mentioned above.

Notice that the algorithm does not enforce the inequality for subfunctions that are ALIVE. These functions satisfy the conditions automatically, due to the way of creating the $\mathbf{p}$ structure, as we will show later. However, the subfunctions that are INACTIVE do not need to satisfy any conditions.

Algorithm 8: Recursive exploration of reachable nodes in the DAG

```
1 Function explore(f_{i,j}) is
2     for (i', j') ∈ N(x_{p(f_{i,j})}) do
3         if (p(i', j') ≠ INACTIVE) ∧ ((f_{i,j}, f_{i',j'}) ∈ E_Δ) then
4             d(f_{i',j'}) ← d(f_{i',j'}) + 1;
5             if r(f_{i',j'}) = FALSE then
6                 r(f_{i',j'}) ← TRUE
7                 explore(f_{i',j'});
```

The algorithm assumes that on input, all the in-degrees stored in structure $\mathbf{d}$ are zero and all the values in the $\mathbf{r}$ structure are FALSE. These are also the initial values, which are changed during the runtime of the algorithm. When the calculations are ended, the values in the structures are returned to their previous values. In case of $\mathbf{r}$, this is done while traversing the augmenting DAG, when the values of reached nodes are set back to FALSE. The structure with in-degrees is gradually lowered down to -1 for each visited node, which is then set back to 0 for the purposes of the next iteration.

In a similar manner, it assumes that the initial values of all elements of $\Delta \mathbf{x}$ and $\mathbf{c}$ are zero. The values of these are set to zero after their values are used in the

step size calculation, i.e. Algorithm 9. It is efficient to store the indices of potentially non-zero values of $\Delta\mathbf{x}$ in a stack for further efficient use.

3.1.4 Line Search

When the direction $\Delta\mathbf{x}$ is calculated, we are interested in finding the length of the step t that would decrease the value of the function f, i.e.

$$f(\mathbf{x} + t\Delta\mathbf{x}) < f(\mathbf{x}) \tag{3.7}$$

and also guarantee that the algorithm will converge with the chosen method of the step size calculation.

This problem is actually only one-dimensional because

$$f(\mathbf{x} + t\Delta\mathbf{x}) = \sum_{i=1}^{l} \max_{j=1}^{m_i}(\mathbf{a}_{i,j}^T(\mathbf{x} + t\Delta\mathbf{x}) + b_{i,j}) = \sum_{i=1}^{l} \max_{j=1}^{m_i}(t \cdot c_{i,j} + y_{i,j}) = g(t), \tag{3.8}$$

where $c_{i,j} = \mathbf{a}_{i,j}^T\Delta\mathbf{x}$ and $y_{i,j} = \mathbf{a}_{i,j}^T\mathbf{x} + b_{i,j}$ are scalars and g is a convex piecewise affine function with one variable t. We could be interested in the optimal length of the step t. That is the one that minimizes function g, i.e.

$$t^* \in \operatorname*{argmin}_t g(t). \tag{3.9}$$

After this value would be obtained, we would not need to enforce other constraints and directly use it. However, a linear-time algorithm for finding such t^* is not known and that is why we resort to heuristic methods.

More specifically, we will find a step size t so that in all clusters $i \in [l]$, no subfunction increases above the value of the previous maximum, i.e.

$$\forall i \in [l], \forall j \in [m_i]: \; f_{i,j}(\mathbf{x} + t\Delta\mathbf{x}) \leq \max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}). \tag{3.10}$$

We apply an additional constraint on the subfunctions in cluster i^*, where the maximum value of a subfunction should be strictly lower than the previous maximum. There are multiple ways of choosing $t > 0$ that satisfy the above constraints. The shown methods consist of two parts – first, use a strategy for initializing the step size t such that the maximum in cluster i^* decreases and then enforce the constraint (3.10).

In all the initialization strategies, we first find the index $j_{i^*}^{\text{slowest}}$ such that it primarily has the maximal $f_{i^*,j}(\mathbf{x})$ value among $S_{i^*}(\mathbf{x})$ and if there are multiple maxima, then the one with the largest $c_{i^*,j}$ is chosen[1]. After this index is found,

[1] If there are still multiple subfunctions that have the maximal $c_{i^*,j}$, then they are identical from

we call the corresponding subfunction in the cluster i^* as the *slowest decreasing*. The slope of this subfunction is at most -1, due to inequalities (3.3), therefore for infinitesimally small t it will hold that

$$\max_{j=1}^{m_{i^*}} f_{i^*,j}(\mathbf{x}+t\Delta\mathbf{x}) \leq \max_{j=1}^{m_{i^*}} f_{i^*,j}(\mathbf{x}) - t, \tag{3.11}$$

but if t is too large, other subfunctions may become active at the point $\mathbf{x}+t\Delta\mathbf{x}$. This fact itself would not limit us in increasing t further if the newly active functions are still decreasing. But if they were increasing, we should stop.

Yet, re-calculating active functions for each step size would be time-consuming, therefore we introduce the heuristic methods.

First Non-decreasing Hit Strategy

In this initialization strategy, we will calculate the intersection points t' of the slowest decreasing subfunction with the non-decreasing subfunctions[2] from the cluster i^* and then choose the smallest t'. This approach guarantees that no value of a non-decreasing previously inactive subfunction will exceed the value of $f_{i^*,j_{i^*}^{\text{slowest}}}$. That is why we call this method first non-decreasing hit – because at the value of the returned t, the slowest decreasing subfunction has the same value as the first encountered non-decreasing subfunction (first in the sense that it corresponds to the smallest such t).

This idea is illustrated in Figure 3.1, which shows the values $f_{i^*,j}(\mathbf{x}+t\Delta\mathbf{x})$ of the individual subfunctions based on the step size t and also the maximum value $\max_{j=1}^{m_{i^*}} f_{i^*,j}(\mathbf{x}+t\Delta\mathbf{x})$ at each point. The slowest decreasing subfunction is shown as the red one, whereas the other ones are blue. Observe that at point $\mathbf{x}$ (which corresponds to $t=0$), there was also another active subfunction, which decreases faster. Then, there are the other subfunctions that were not active – from these, the non-decreasing ones are filtered and their intersection points with the slowest decreasing are calculated as t_1, t_2, t_3. In this case, we would initialize the value of t to t_1 because it is the minimum value among $\{t_1, t_2, t_3\}$.

The above initialization procedure of the line search formally corresponds to

$$t_{\text{FND}} \leftarrow \inf \left\{ \frac{f_{i^*,j_{i^*}^{\text{slowest}}}(\mathbf{x}) - f_{i^*,j}(\mathbf{x})}{c_{i^*,j} - c_{i^*,j_{i^*}^{\text{slowest}}}} \;\middle|\; j \in [m_{i^*}], c_{i^*,j} \geq 0 \right\}. \tag{3.12}$$

Notice that if there is no subfunction that is non-decreasing in the direction $\Delta\mathbf{x}$, then we set t as infinity. This is why we use infimum in the initialization – if the set

the point of view of line search and the choice is arbitrary. Even though all $f_{i^*,j}(\mathbf{x})$ for $j \in S_{i^*}(\mathbf{x})$ are equal, this general approach is necessary due to the ϵ generalizations that will come later.

[2] Notice that all the non-decreasing subfunctions in cluster i^* are necessarily inactive.

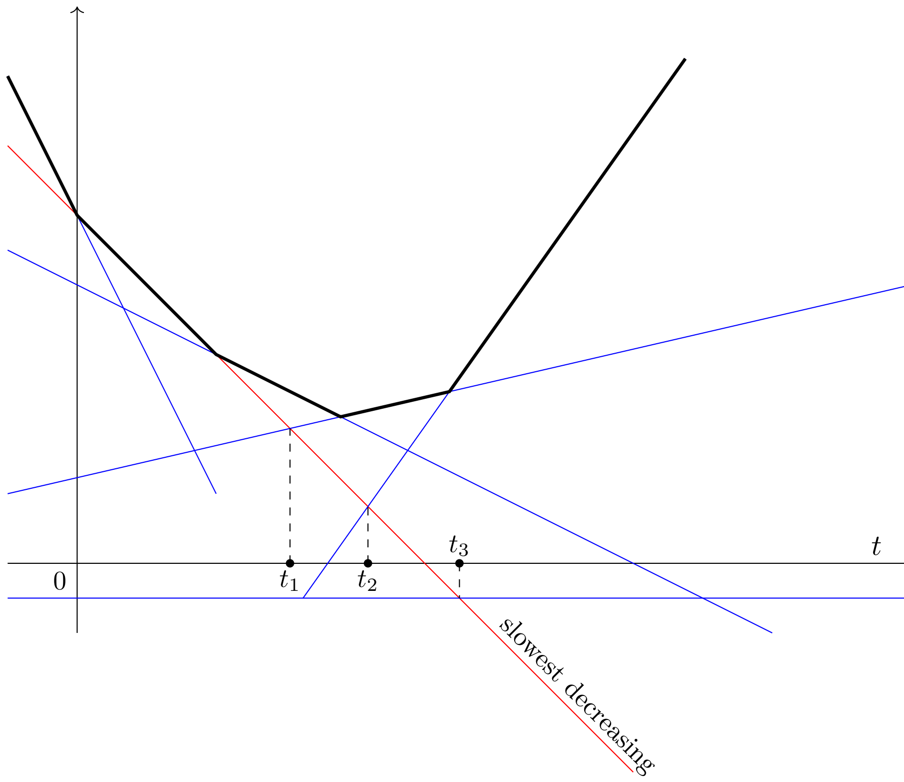


Figure 3.1: Example case of searching the initial value for t by first non-decreasing hit strategy.

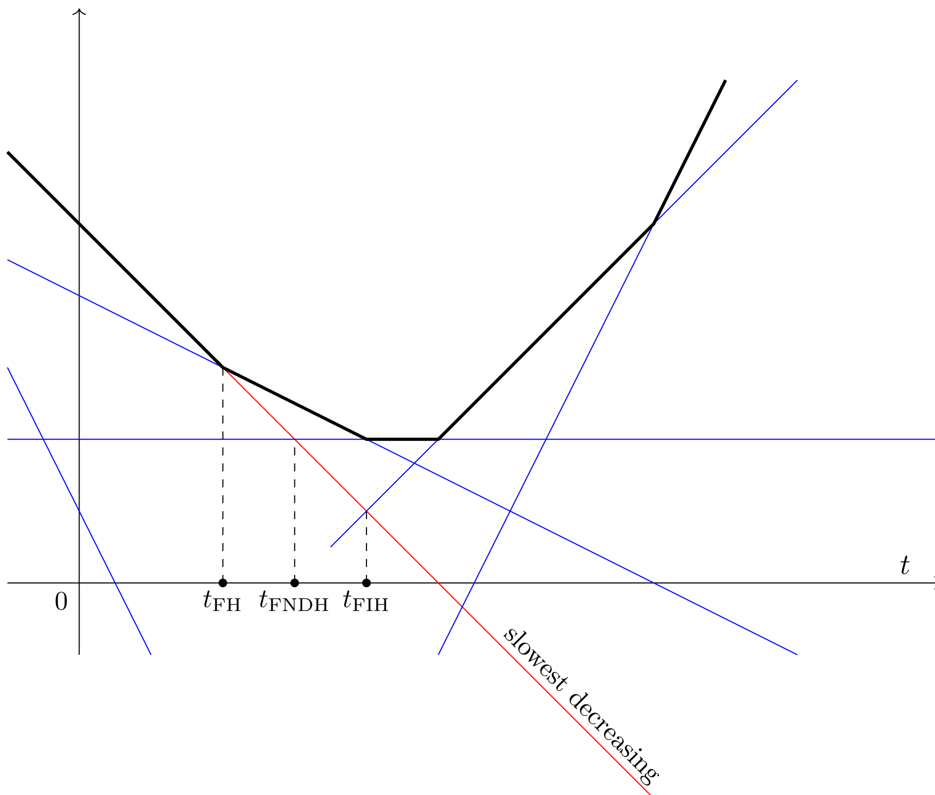


Figure 3.2: Comparison of initialization methods of t.

over which the infimum is taken is empty, the result is infinity, otherwise it is the minimum among the values in the set.

First-hit Strategy

Analogously to the previously described method, we could also use the first-hit strategy, which forbids any subfunction that was previously inactive to exceed the value of the slowest decreasing function, i.e.

$$t_{\mathrm{FH}} \leftarrow \inf \left\{ \frac{f_{i^*,j_{i^*}^{\mathrm{slowest}}}(\mathbf{x}) - f_{i^*,j}(\mathbf{x})}{c_{i^*,j} - c_{i^*,j_{i^*}^{\mathrm{slowest}}}} \;\middle|\; j \in [m_{i^*}], p(f_{i^*,j}) = \mathsf{INACTIVE} \right\}. \tag{3.13}$$

First Increasing Hit Strategy

A slightly more complicated strategy of finding t is to initialize it to analogously hit the first increasing (therefore inactive) subfunction, i.e. initialize

$$t_{\mathrm{FIH}} \leftarrow \inf \left\{ \frac{f_{i^*,j_{i^*}^{\mathrm{slowest}}}(\mathbf{x}) - f_{i^*,j}(\mathbf{x})}{c_{i^*,j} - c_{i^*,j_{i^*}^{\mathrm{slowest}}}} \;\middle|\; j \in [m_{i^*}], c_{i^*,j} > 0 \right\}. \tag{3.14}$$

However, this strategy is different from the above ones in the sense that if $t_{\mathrm{FNDH}} = \infty$ or $t_{\mathrm{FH}} = \infty$, then we know that the value of $\max_{j=1}^{m_{i^*}} f_{i^*,j}(\mathbf{x} + t\Delta\mathbf{x})$ is not bounded and with large-enough choice of t, it could be made arbitrarily small. This does not necessarily hold whenever $t_{\mathrm{FIH}} = \infty$, because there could be constant subfunctions that would bound the cluster maximum from below. We will discuss this in detail later.

Comparison of Strategies

It is easy to compare the t values of the heuristic methods, since

$$c_{i^*,j} > 0 \;\Rightarrow\; c_{i^*,j} \geq 0 \;\Rightarrow\; p(i^*, j) = \mathsf{INACTIVE} \tag{3.15}$$

and knowing that $\inf A \geq \inf B$ for $A \subseteq B$, we can say that

$$t_{\mathrm{FIH}} \geq t_{\mathrm{FNDH}} \geq t_{\mathrm{FH}}. \tag{3.16}$$

It is however not possible to compare the corresponding values $g(t_{\mathrm{FIH}})$, $g(t_{\mathrm{FNDH}})$ and $g(t_{\mathrm{FH}})$ generally for $l \geq 2$ because the sum of the maxima in other clusters will influence the final function value. For each of the presented heuristic strategies, there is a case in which it lowers the function g more than the others.

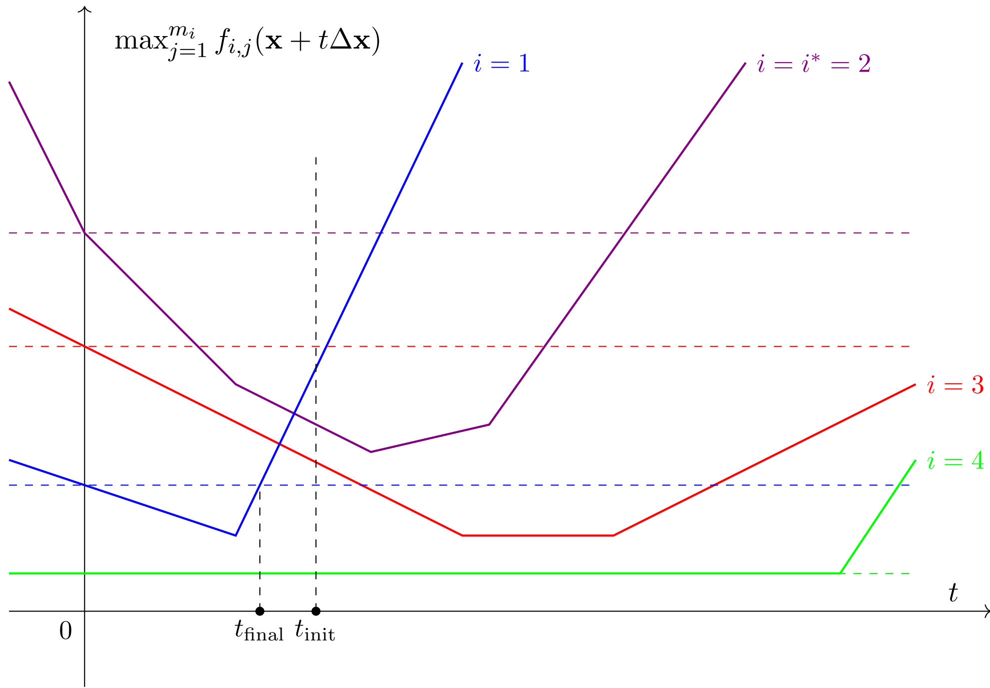


Figure 3.3: Example of enforcing that no cluster maximum increases with step size t.

However, if we are processing a SMAF with $l = 1$ (which is actually a MAF), then it holds that

$$g(t^*) \leq g(t_{\text{FIH}}) \leq g(t_{\text{FNDH}}) \leq g(t_{\text{FH}}). \tag{3.17}$$

The reasoning is simple because the point t_{FIH} is the smallest positive one, in which an increasing subfunction could become active. Therefore, the function g is non-increasing on the interval $[0, t_{\text{FIH}}]$. However, the slowest decreasing function could have become inactive at this point e.g. due to a previously inactive subfunction that decreases slower and therefore the value t_{FIH} is not necessarily optimal even in this case. Figure 3.2 shows the value of $\max_{j=1}^{m_{i^*}} f_{i^*,j}(\mathbf{x} + t\Delta\mathbf{x})$ for various values of t with black colour. The values of other subfunctions are shown in blue, except for the slowest decreasing one, which is emphasised by red.

Assuring Non-increasing Cluster Maxima

After the value t was initialized by one of the previous strategies, we need to assure that the condition (3.10) holds. So we go through the subfunctions $f_{i,j}$ that have possibly non-zero value $c_{i,j} = \mathbf{a}_{i,j}^T \Delta\mathbf{x}$ and if the corresponding value $c_{i,j}$ is positive, it means that the subfunction $f_{i,j}$ increases in the $\Delta\mathbf{x}$ direction and we need to make sure that the value of the maximum in cluster i would not increase after the update. It could therefore lower the step size.

This is shown in Figure 3.3, where the maxima of $l = 4$ clusters are shown

based on the value of t. Their original maxima for $t = 0$ are shown by dashed lines in corresponding colour. For cluster $i^* = 2$, it obviously by definition holds that the value of the maximum is not higher in t_{init}, which was in this case calculated by the first non-decreasing hit strategy. The maxima of clusters 3 and 4 are also not increased at this point, but the maximum in cluster 1 has higher value in $t = t_{\text{init}}$ than in $t = 0$, therefore we lower the value of t to t_{final}, which is the final value of the step size in this case. If there were more clusters violating this condition, we would need to check all of them and possibly decrease the value of t multiple times.

Algorithm 9: Line search algorithm for SMAF

1 **Function** calculateStepSize() **is**

2 $\quad j_{i^*}^{\text{slowest}} \leftarrow \underset{j}{\operatorname{argmax}} \left\{ c_{i^*,j} \;\middle|\; j \in \underset{j' \in S_{i^*}(\mathbf{x})}{\operatorname{argmax}} f_{i^*,j'}(\mathbf{x}) \right\};$

3 $\quad t \leftarrow \inf \left\{ \frac{f_{i^*,j_{i^*}^{\text{slowest}}}(\mathbf{x}) - f_{i^*,j}(\mathbf{x})}{c_{i^*,j} - c_{i^*,j_{i^*}^{\text{slowest}}}} \;\middle|\; j \in [m_{i^*}], c_{i^*,j} \geq 0 \right\};$

4 $\quad$ **for** $i \in \{i' \mid (i', j) \in N(x_k), \Delta x_k \neq 0\}$ **do**

5 $\quad\quad$ **for** $j \in [m_i]$ **do**

6 $\quad\quad\quad$ **if** $c_{i,j} > 0$ **then**

7 $\quad\quad\quad\quad t \leftarrow \min \left\{ t, \frac{1}{c_{i,j}} \left(\max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}) - f_{i,j}(\mathbf{x}) \right) \right\};$

8 $\quad$ **return** t;

In Algorithm 9, the whole procedure is formalized. On line 2, we use the first non-decreasing hit strategy to initialize the value of t. Then, we iterate over the subfunctions that could be increasing and for the ones that are, we assert that the cluster maximum does not increase. That is guaranteed by the update on line 7.

If the t returned by Algorithm 9 is infinite, it means that the function is unbounded and can be decreased arbitrarily in this direction. Note that even when a finite value of t is returned by Algorithm 9, it does not guarantee that the function f is bounded (even in the direction $\Delta\mathbf{x}$). This could happen if there are subfunctions that would cause some maxima to increase, but other maxima decrease more "steeply" so that the whole function decreases.

If we used the first-hit strategy in the algorithm, it would work in the same fashion. But, for the first increasing hit strategy initialization, it would need an additional modification to run correctly. Observe that if the value $t = t_{\text{FIH}}$ was initialized to infinity and then was not updated in the subsequent for loops on lines 4–7, the algorithm could incorrectly mark the input function as unbounded. It would happen in the case when there is no increasing subfunction in any cluster, but in each of them, there is at least one constant subfunction. So, to guarantee the correctness of the algorithm with FIH initialization, whenever it would like to return infinity, we

need to do an additional assignment

$$t \leftarrow \inf \left\{ \frac{f_{i^*,j_{i^*}^{\text{slowest}}}(\mathbf{x}) - f_{i^*,j}(\mathbf{x})}{c_{i^*,j} - c_{i^*,j_{i^*}^{\text{slowest}}}} \,\middle|\, j \in [m_{i^*}], c_{i^*,j} = 0 \right\} \tag{3.18}$$

that would lower the t_{FIH} to a finite value if there exists a constant subfunction in the i^* cluster and thus ensure the correctness of the result.

We have chosen the first non-decreasing hit strategy in our implementation because it does not require to iterate over the subfunctions in the cluster i^* again in cases of suspected unboundedness.

3.1.5 Update of x

By now, we have already introduced the local consistency algorithm which finds out whether a point $\mathbf{x}$ is locally consistent. If it is not, we can use the output of the local consistency algorithm and construct a decreasing direction vector and finally calculate the step size. Using the decreasing direction vector and the step size, we can perform the step that updates the $\mathbf{x}$ value and repeat this procedure iteratively from scratch.

Unfortunately, this straightforward update would throw away all the previous calculations which were performed in the local consistency algorithm and maybe repeat some of them, which would lead to inefficiency and slowdown of the algorithm.

To improve the efficiency of the algorithm, we introduce inter-iteration warm-starting, which is based on preserving some $\mathbf{p}$ values, so that they do not need to be calculated again. We will include the warm-starting in the update procedure – i.e. the procedure that performs the step and prepares the structures for the next iteration. This procedure is formalized in Algorithm 10.

After both the direction $\Delta\mathbf{x}$ and the step size t were found, we would like to update the value of the current point $\mathbf{x}$ and the subfunction values $\mathbf{y}$, namely

$$\mathbf{x} \leftarrow \mathbf{x} + t\Delta\mathbf{x} \tag{3.19a}$$

$$\mathbf{y} \leftarrow \mathbf{y} + t\mathbf{c}. \tag{3.19b}$$

But it would be highly inefficient to update the whole vectors, so we will update only those elements of $\mathbf{x}$ that correspond to non-zero $\Delta\mathbf{x}$ values. Similar procedure can be done with $\mathbf{y}$, where it is enough to only update those $y_{i,j}$ that are in a cluster that contains a subfunction that depends on a non-zero variable in $\Delta\mathbf{x}$. Notice that the maxima of only these clusters may change, the other maxima will stay the same even after the update.

After the update of $\mathbf{x}$, some active functions may become inactive and vice versa. Algorithm 11 is called for those that become newly active (i.e. were inactive in

the previous iteration and now are active) and Algorithm 12 is called for those that become inactive (i.e. were active in the previous iteration and now are inactive). Both these procedures also update the queue Q that contains the possibly inconsistent variables. After this is done, the content of the queue Q and statuses stored in $\mathbf{p}$ are ready to be processed by the local consistency algorithm.

Algorithm 10: Performs step of specified size

1. **Function** performStep(t) **is**
2. $\quad K \leftarrow \{k \mid \Delta x_k \neq 0\}$;
3. $\quad I \leftarrow \{i \mid k \in K, (i,j) \in N(x_k)\}$;
4. $\quad$ **for** $i \in I$ **do**
5. $\quad\quad$ **for** $j \in [m_i]$ **do**
6. $\quad\quad\quad y_{i,j} \leftarrow y_{i,j} + c_{i,j}t$;
7. $\quad\quad\quad c_{i,j} \leftarrow 0$;
8. $\quad\quad$ **for** $j \in S_i(\mathbf{x} + t\Delta\mathbf{x}) - S_i(\mathbf{x})$ **do**
9. $\quad\quad\quad$ activate($f_{i,j}$);
10. $\quad\quad$ **for** $j \in S_i(\mathbf{x}) - S_i(\mathbf{x} + t\Delta\mathbf{x})$ **do**
11. $\quad\quad\quad$ deactivate($f_{i,j}$);
12. $\quad$ **for** $k \in K$ **do**
13. $\quad\quad x_k \leftarrow x_k + \Delta x_k t$;
14. $\quad\quad \Delta x_k \leftarrow 0$;

Algorithm 11: Activate subfunction $f_{i,j}$

1. **Function** activate($f_{i,j}$) **is**
2. $\quad p(i,j) \leftarrow$ ALIVE;
3. $\quad$ **for** $k \in N(f_{i,j})$ **do**
4. $\quad\quad Q$.push(k);
5. $\quad\quad$ **for** $(i',j') \in N(x_k)$ **do**
6. $\quad\quad\quad$ **if** $(p(i',j') = k) \wedge (a_{i,j,k}a_{i',j',k} < 0)$ **then**
7. $\quad\quad\quad\quad$ activate($f_{i',j'}$);

Algorithm 12: Deactivate subfunction $f_{i,j}$

1. **Function** deactivate($f_{i,j}$) **is**
2. $\quad p(i,j) \leftarrow$ INACTIVE;
3. $\quad$ **for** $k \in N(f_{i,j})$ **do**
4. $\quad\quad Q$.push(k);

In case of activating a previously inactive function, it could happen that a previously inconsistent variable is now consistent, so it is necessary to set the ALIVE

status to some subfunctions that were previously killed, which may again resurrect other subfunctions etc. On the other hand, deactivating a subfunction cannot cause other subfunctions to change their status.

3.1.6 The Whole Algorithm

Now we have all the building blocks of the whole minimization, which is shown in Algorithm 13. First, the global variables are initialized and then the main minimizing loop is started. If the current point $\mathbf{x}$ is locally consistent, it is returned as a result. Otherwise, we continue with calculating the decreasing direction $\Delta\mathbf{x}$ and step size t. If the step size is infinite, then the input function is not bounded from below and we terminate the calculation. If it is a finite value, we perform the step and continue with the next iteration.

Algorithm 13: SMAF minimization algorithm

```
1  Function minimize(l, m, a, b) is
2      initialize();
3      loop
4          if calculateConsistency() then
5              return x;
6          calculateDirection();
7          t ← calculateStepSize();
8          if t = ∞ then
9              return unbounded input;
10         performStep(t);
```

3.1.7 Using ϵ-consistency

It is possible to relax the notion of active subfunctions and introduce ϵ-active subfunctions, so that the definition would result in searching for locally ϵ-consistent points. This relaxation speeds up the convergence and helps the algorithm reach points with lower function value. Additionally, we will later show that having $\epsilon > 0$ is a necessary condition for proving the correctness of the algorithm in the above version.

Definition 3.4. We say that a subfunction $f_{i,j}$ is *ϵ-active* for a given $\epsilon \geq 0$ in a point $\mathbf{x} \in \mathbb{R}^n$ iff $f_{i,j}(\mathbf{x}) \geq \max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}) - \epsilon$. Next, we can define the set $S_i^\epsilon(\mathbf{x})$ of ϵ-active functions for each $i \in [l]$ as

$$S_i^\epsilon(\mathbf{x}) = \{j \in [m_i] \mid f_{i,j} \text{ is } \epsilon\text{-active in } \mathbf{x}\} \tag{3.20}$$

and similarly for the whole set

$$I^{\epsilon}(\mathbf{x}) = \{(i, j) \mid i \in [l], j \in S_i^{\epsilon}(\mathbf{x})\}. \tag{3.21}$$

Now, we can replace the occurrences of S_i with S_i^{ϵ} in the algorithm and look for a locally ϵ-consistent point without any other alterations. Once such point is found, we can lower ϵ to further optimize and possibly lower the function value. A similar relaxation was presented in Koval' and Schlesinger (1976), where it was used to minimize the height of a max-sum problem.

Capacity Scaling

The previously shown relaxation could be viewed as a generalized capacity scaling, which is an idea introduced by Edmonds and Karp (1972) as an improvement of the well-known algorithm from Ford Jr and Fulkerson (1955). The basic Ford-Fulkerson algorithm is used for finding maximum flow or minimum cut by iteratively finding an arbitrary augmenting path from source to sink and increasing the flow along this path.

Some improvements of Ford-Fulkerson algorithm were later introduced to improve its pseudopolynomial runtime. One of them is capacity scaling, which first uses only those edges along which the flow can be increased by at least some threshold value ϵ and ignores the other ones. Once the flow cannot be increased, the threshold ϵ is lowered, so it is allowed to also use the edges with lower capacity. Eventually, when the threshold reaches zero, we can terminate the algorithm with optimal solution.

The capacity scaling offers applications in computer vision, e.g. Juan and Boykov (2007) and can be used even in generalized setting of max-flow, e.g. Ahuja and Orlin (1995).

3.2 Correctness

In this section, we show that if the minimized SMAF f has a minimum, then the presented ϵ-version of the algorithm always finds the minimum or at least ends at an locally ϵ-consistent point in a finite number of iterations. This proof will be based mainly on showing the correctness of Algorithm 6 and finding a lower bound of the value by which the function f decreases in each iteration.

3.2.1 Correctness of Local Consistency Algorithm

Proposition 3.1. *The queue Q contains all the initially inconsistent variables right before Algorithm 6 is executed for the first time.*

Proof. Before the first iteration, all the variables on which at least one active subfunction depends are put to the queue Q using the initialize() function. The only variables that are not in the queue do not depend on any of the active subfunctions, therefore they cannot be inconsistent. ■

Lemma 3.1. *On line 5 in Algorithm 6, it never happens that Q fails to contain an inconsistent variable.*

Proof. Using Proposition 3.1, the queue contains them at the start of the first iteration. Then, the local consistency algorithm is run and whenever an inconsistent variable is found, all the subfunctions that depend on it and are still alive are marked as killed. The removal of these subfunctions from the alive ones might have made some new variables inconsistent. But only those on which at least one of the killed subfunctions depends – and exactly these variables are put back to the queue on line 12 of Algorithm 6. When the set of active subfunctions changes due to update of $\mathbf{x}$, all the variables on which the activated or deactivated functions depend are again added to the queue. This happens in Algorithm 11 on line 4 and in Algorithm 12 on line 4. The consistency of any other variables could not have changed. ■

Proposition 3.2. *After the local consistency algorithm terminates, for all variables x_k it holds that the sign of all $\{a_{i,j,k} \mid p(i,j) = k\}$ is the same.*

Proof. Assume that the claim does not hold and there are two subfunctions $f_{i,j}$, $f_{i',j'}$ with $p(i,j) = p(i',j') = k$, but $\operatorname{sign}(a_{i,j,k}) = -\operatorname{sign}(a_{i',j',k}) \neq 0$. The p values must have been set at some point during the run of the algorithm, because they are initially $\mathsf{INACTIVE}$. And setting the p value to a coordinate happens only when this coordinate is inconsistent, i.e. on line 10 of Algorithm 6. But this could not have happened during one execution of the block of code on lines 8–12 of the algorithm, because if both $f_{i,j}$ and $f_{i',j'}$ were active and alive, x_k would be consistent.

So, it must WLOG hold that $p(i,j) \leftarrow k$ was assigned at some point before $p(i',j')$ was set to its current value and $p(i,j)$ did not change since then. It must have been when $p(i',j') = \mathsf{INACTIVE}$ or $p(i',j') = k' \neq k$ but then $f_{i',j'}$ must necessarily became ALIVE using Algorithm 11 (otherwise, it would have kept its previous value). But at this point, because both $f_{i,j}$ and $f_{i',j'}$ depend on x_k, $p(i,j)$ should have been set to ALIVE by the function call on line 7 of Algorithm 11, so the claim holds. ■

Theorem 3.1. *If the queue Q initially contains all inconsistent variables, then Algorithm 6 returns true if and only if the current point is locally ϵ-consistent.*

Proof. The proof will be done by showing that the procedure shown in Section 2.2.2 returns vector $\boldsymbol{\sigma}$ that satisfies CSP (2.24) if and only if the Algorithm 6 returns true. The correctness of the CSP solving procedure is obvious.

It is important to notice that the inactive subfunctions are neither considered by Algorithm 6 nor by the CSP solving procedure and can be therefore viewed as non-existing for the functionality of the algorithms. Considering the other subfunctions, if a subfunction $f_{i,j}$ has $\sigma_{i,j} = 1$, it corresponds to $p(i, j) = \mathsf{ALIVE}$. Likewise, $\sigma_{i,j} = 0$ corresponds to $p(i, j) \in [n]$.

Now that we know that if a coordinate x_k is inconsistent, it means that either the condition in step 2 or in step 3 in the CSP solving procedure is satisfied. Killing the subfunctions with the non-zero coefficient $a_{i,j,k}$ would correspond to repetitive usage of the same rule in the CSP solving procedure. After the subfunctions are killed, we check whether there is at least one alive subfunction in each cluster, which corresponds to having at least one non-zero $\sigma_{i,j}$ in each cluster in the CSP solving procedure. Thus, it is easy to see that whenever the CSP solving procedure ends with vector $\boldsymbol{\sigma}$ that does not satisfy the CSP, Algorithm 6 would return FALSE because the condition on the clusters is equivalent and is checked whenever it could have been satisfied.

On the other hand, if the CSP solving procedure ends with $\boldsymbol{\sigma}$ that satisfies the CSP, it means that the conditions in step 2 or 3 are not satisfiable with the current $\boldsymbol{\sigma}$, which corresponds to no inconsistent variables in Algorithm 6. And if there are no inconsistent variables, the queue will be gradually emptied without any additions into it. So eventually, the queue Q will become empty and Algorithm 6 returns TRUE. ■

3.2.2 Correctness of Decreasing Direction

Theorem 3.2. *The graph defined in* (3.4) *is acyclic.*

Proof. This proof will be done by contradiction. Assume that there is a cycle $f_{i_1,j_1} \to f_{i_2,j_2} \to \cdots \to f_{i_q,j_q} \to f_{i_1,j_1}$, which is shortest in the sense that there is no subset of these nodes so that these nodes make a cycle themselves. The requirement on shortest cycle ensures along with Proposition 3.2 that the values $p(i_s, j_s)$ for all $s \in [q]$ are unique. If they were not and there would be $s', s'' \in [q]$, $s' < s''$, such that $p(i_{s'}, j_{s'}) = p(i_{s''}, j_{s''})$, it would have to hold that $\operatorname{sign}(a_{i_{s'},j_{s'},p(i_{s'},j_{s'})}) = \operatorname{sign}(a_{i_{s''},j_{s''},p(i_{s''},j_{s''})})$, which means that there would be an edge from $f_{i_{s'},j_{s'}}$ to $f_{i_{s^*},j_{s^*}}$, where $s^* = (s'' \bmod q) + 1$, and this edge could shorten the cycle by $s'' - s' > 0$ nodes.

Therefore, the cycle is the shortest and the $p(i_s, j_s)$ values are unique, so they must have been assigned at different times when each of the variables was inconsistent and the subfunction f_{i_s,j_s} was alive. Assume WLOG that the subfunction f_{i_1,j_1} is the first one among those in the cycle that was assigned its current value $p(i_1, j_1)$, which did not change since then. But at this time, the subfunction f_{i_2,j_2} must have been either killed (with a different variable than it has now) or not active

yet because otherwise the coordinate $p(i_1, j_1)$ would be consistent. In both cases, the value of $p(i_2, j_2)$ must be changed to obtain the current cycle. To do that, it must first become alive, which can happen only in the activating procedure on line 2 in Algorithm 11. But this procedure would set the value of $p(i_1, j_1)$ to ALIVE, because $a_{i_1,j_1,p(i_1,j_1)}a_{i_2,j_2,p(i_1,j_1)} < 0$ (otherwise, there would not be the edge from f_{i_1,j_1} to f_{i_2,j_2}). So the assignment to $p(i_1, j_1)$ was not the final one, which leads to a contradiction. ■

Proposition 3.3. *For a given subfunction $f_{i',j'}$ with $p(i', j') \in [n]$, there is no subfunction $f_{i,j}$ with $p(i, j) = \mathsf{ALIVE}$ that would increase its value if the value of variable $x_{p(i',j')}$ changes so that $f_{i',j'}$ decreases.*

Proof. Again, we will prove this by contradiction. Assume that there would be such subfunction with $a_{i,j,p(i',j')}a_{i',j',p(i',j')} < 0$. The value $p(i', j')$ must have been set at some point when the coordinate $p(i', j')$ was inconsistent and then the value of $p(i', j')$ did not change. So either $p(i, j) \in [n]$ or $p(i, j) = \mathsf{INACTIVE}$ (otherwise the coordinate $p(i', j')$ would be consistent). The value of $p(i, j)$ must have been set after that to ALIVE using the activate function. But that would also activate the subfunction $f_{i',j'}$, which is a contradiction.

Now look at the other case – $p(i, j)$ was at some point set as ALIVE and since then did not change its value, and after that, $p(i', j')$ was set to some coordinate on which $f_{i,j}$ also depends. The sign of $a_{i,j,p(i',j')}$ must be the opposite than the sign of $a_{i',j',p(i',j')}$ because otherwise, $f_{i,j}$ would decrease when $f_{i',j'}$ decreases due to change in $x_{p(i',j')}$. If the signs are opposite, then the variable $x_{p(i',j')}$ is consistent and we could not have set is in such a way. ■

Proposition 3.4. *Algorithm 7 processes the subfunctions $f_{i,j}$ in a topological order with respect to the augmenting DAG during the main while loop.*

Proof. First of all, the algorithm starts in each active subfunction from the cluster i^*, marks it as reached on line 4 and continues in the recursive exploration of the graph from this subfunction on line 5. If an already visited node is reached, its in-degree is increased by one on line 4 of Algorithm 8 and the recursive exploration halts. If it was not yet visited, the exploration continues from it by the recursive call on line 7. It is obvious that this procedure calculates the in-degree of each node in the graph correctly.

After that, we iterate over the active subfunctions from the cluster i^* and add those with zero in-degree to the queue Q_f – this is done on line 9 in Algorithm 7. These are the first subfunctions in the topological order, because there are no edges leading to them.

Then, the main while loop on lines 10–24 is entered. When a subfunction $f_{i,j}$ is processed, we also look at to which subfunctions $f_{i',j'}$ there is an edge in the graph

from $f_{i,j}$. And if there is an edge, the in-degree of $f_{i',j'}$ is lowered by one by the assignment on line 20. The in-degree $d(f_{i',j'})$ can now be viewed as a counter that indicates how many subfunctions must be processed before $f_{i',j'}$ to ensure that the ordering is topological. If $d(f_{i',j'}) = 0$, it means that all of the subfunctions that have an edge to $f_{i',j'}$ were already processed, which means that $f_{i',j'}$ can be processed at any time from now and the ordering is thus topological. ■

Lemma 3.2. *After Algorithm 7 processes the subfunction* f_{i_s,j_s}*, then*

$$c_{i_w,j_w} \leq -[\![i_w = i^*]\!], \quad \forall w \in [s], \tag{3.22}$$

where $f_{i_1,j_1}, f_{i_2,j_2}, ..., f_{i_q,j_q}$*,* $s \leq q$ *is the topological order of processing.*

Proof. This proof will be done by induction. First, we will show the basic step for $s = 1$. Then, necessarily $i_1 = i^*$ and because $c_{i_1,j_1} = 0$, we need to set

$$\Delta x_{p(i_1,j_1)} \leftarrow 0 - 1/a_{i_1,j_1,p(i_1,j_1)}, \tag{3.23}$$

which results in c_{i_1,j_1} being assigned value -1, therefore the condition is satisfied.

Second, we will show the inductive step for $s \geq 2$: assume that for all $w < s$, the condition holds and we would like to show how the algorithm behaves when it processes s-th subfunction. If the condition is satisfied initially, then no update is needed. On the other hand, if it is violated, the c_{i_s,j_s} has some higher value, let us denote it as $z = c_{i_s,j_s} > -[\![i_s = i^*]\!]$. This forces the update

$$\Delta x_k \leftarrow \Delta x_k - ([\![i_s = i^*]\!] + z)/a_{i_s,j_s,p(i_s,j_s)}, \tag{3.24}$$

which alters the value of c_{i_s,j_s} to

$$z - a_{i_s,j_s,p(i_s,j_s)}([\![i_s = i^*]\!] + z)/a_{i_s,j_s,p(i_s,j_s)} = z - [\![i_s = i^*]\!] - z = -[\![i_s = i^*]\!], \tag{3.25}$$

therefore the condition is now satisfied also for f_{i_s,j_s}. The condition was not violated for any of the previous $w < s$ because there is no edge from f_{i_s,j_s} to any of the previous subfunctions in the augmenting DAG, so they could not increase by the same reasoning as in the proof of Proposition 2.2. ■

Theorem 3.3. *If Algorithm 6 ends with the result that the current point* $\mathbf{x}$ *is inconsistent, Algorithm 7 finds a decreasing direction* $\Delta\mathbf{x} \in \mathbb{R}^n$ *that satisfies*

$$f_{i,j}(\mathbf{x} + \Delta\mathbf{x}) \leq f_{i,j}(\mathbf{x}), \qquad \forall (i,j) \in I(\mathbf{x}) \tag{3.26a}$$

$$f_{i^*,j}(\mathbf{x} + \Delta\mathbf{x}) \leq f_{i^*,j}(\mathbf{x}) - 1, \qquad \forall j \in S_{i^*}(\mathbf{x}), \tag{3.26b}$$

where i^* *is the cluster for which there is no subfunction* $f_{i^*,j}$ *with* $p(i^*, j) = \mathsf{ALIVE}$*.*

Proof. First of all, we will use Lemma 3.2, which implies that after the algorithm ends, all subfunctions that are nodes in the augmenting DAG satisfy the condition (3.6).

Second, Proposition 3.3 can be used to show that if a subfunction $f_{i,j}$ has status ALIVE, then $c_{i,j} \leq 0$ because in the direction algorithm, the corresponding variables that are stored in $\mathbf{p}$ are changed always so that the corresponding subfunction decreases. Since $i \neq i^*$, having $c_{i,j} \leq 0$ satisfies the condition (3.6). See that $i \neq i^*$ because $p(i,j) = \mathsf{ALIVE}$.

Finally, the subfunctions $f_{i,j}$ that are not in the augmenting DAG but satisfy $p(i,j) \in [n]$ also satisfy $c_{i,j} \leq 0$ because initially, $c_{i,j} = 0$ and because no edge leads to them, they could not have increased their value. This follows from the definition of the DAG. No such subfunction satisfies $i = i^*$ because it would have been in the augmenting DAG if $i = i^*$ held. Therefore, the condition (3.6) is again satisfied.

From the reasoning above, we can say that all active subfunctions satisfy the condition

$$\mathbf{a}_{i,j}^T \Delta\mathbf{x} \leq -[\![i = i^*]\!],\ \forall (i,j) \in I(\mathbf{x}), \tag{3.27}$$

since it can be easily seen that $\mathbf{a}_{i,j}^T \Delta\mathbf{x} = c_{i,j}$. This is equivalent to

$$\mathbf{a}_{i,j}^T (\Delta\mathbf{x} + \mathbf{x}) + b_{i,j} \leq \mathbf{a}_{i,j}^T \mathbf{x} + b_{i,j} - [\![i = i^*]\!], \quad \forall (i,j) \in I(\mathbf{x}), \tag{3.28}$$

where we can substitute $f_{i,j}(\mathbf{x}) = \mathbf{a}_{i,j}^T \mathbf{x} + b_{i,j}$ and obtain

$$f_{i,j}(\mathbf{x} + \Delta\mathbf{x}) \leq f_{i,j}(\mathbf{x}), \qquad \forall (i,j) \in I(\mathbf{x}) \tag{3.29a}$$

$$f_{i^*,j}(\mathbf{x} + \Delta\mathbf{x}) \leq f_{i^*,j}(\mathbf{x}) - 1, \qquad \forall j \in S_{i^*}(\mathbf{x}), \tag{3.29b}$$

which was the originally required condition for $\Delta\mathbf{x}$ to satisfy. ■

3.2.3 Correctness of Line Search

Proposition 3.5. *After the direction is calculated by Algorithm 7, for all subfunctions $f_{i,j}$ it holds that if $c_{i,j} > 0$, then*

$$f_{i,j}(\mathbf{x}) < \max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}) - \epsilon. \tag{3.30}$$

Proof. It leads from Theorem 3.3 that all ϵ-active subfunctions must have $c_{i',j'} \leq 0$, therefore if a subfunction does not satisfy that, it is necessarily not ϵ-active. ■

Lemma 3.3. *Given $\epsilon \geq 0$, the computation on line 3 of Algorithm 9 calculates $t > 0$ and any $0 < t' \leq t$ satisfies*

$$f_{i^*,j}(\mathbf{x} + t'\Delta\mathbf{x}) < \max_{j'=1}^{m_i} f_{i^*,j'}(\mathbf{x}), \quad \forall j \in [m_{i^*}]. \tag{3.31}$$

Proof. If there is at least one $c_{i^*,j} \geq 0$, then the corresponding $f_{i^*,j}$ satisfy

$$f_{i^*,j_{i^*}^{\text{slowest}}}(\mathbf{x}) - f_{i^*,j}(\mathbf{x}) > \epsilon \geq 0, \tag{3.32}$$

because $f_{i^*,j_{i^*}^{\text{slowest}}}(\mathbf{x}) = \max_{j'=1}^{m_i} f_{i^*,j'}(\mathbf{x})$ due to the definition of $j_{i^*}^{\text{slowest}}$ and Proposition 3.5. Next, $c_{i^*,j_{i^*}^{\text{slowest}}} \leq -1$ and $c_{i^*,j} \geq 0$, so $c_{i^*,j} - c_{i^*,j_{i^*}^{\text{slowest}}} \geq 1$. The value of the fraction in the brackets of the infimum is therefore always positive because both the nominator and denominator are positive numbers. Thus $t > 0$.

To show the second part of the lemma, we will distinguish two separate cases. First, if a subfunction $f_{i^*,j}$ decreases in the direction $\Delta\mathbf{x}$, then for any $t' > 0$, the inequality (3.31) is obviously satisfied. All the active subfunctions from the cluster i^* decrease, so they satisfy it as well.

Second, if a subfunction $f_{i^*,j}$ increases in the direction $\Delta\mathbf{x}$, then the initialization setting will not allow it to have a higher value than $f_{i^*,j_{i^*}^{\text{slowest}}}$ if $t' \leq t$ because it implies that

$$t' \leq t \leq \frac{f_{i^*,j_{i^*}^{\text{slowest}}}(\mathbf{x}) - f_{i^*,j}(\mathbf{x})}{c_{i^*,j} - c_{i^*,j_{i^*}^{\text{slowest}}}}, \tag{3.33}$$

and since $c_{i^*,j} - c_{i^*,j_{i^*}^{\text{slowest}}} \geq 1$, we can rewrite that as

$$f_{i^*,j}(\mathbf{x}) + t'\, c_{i^*,j} \leq f_{i^*,j_{i^*}^{\text{slowest}}}(\mathbf{x}) + t'\, c_{i^*,j_{i^*}^{\text{slowest}}}. \tag{3.34}$$

or equivalently as

$$f_{i^*,j}(\mathbf{x} + t'\Delta\mathbf{x}) \leq f_{i^*,j_{i^*}^{\text{slowest}}}(\mathbf{x} + t'\Delta\mathbf{x}). \tag{3.35}$$

In this last inequality, we can see that the value of any subfunction that increases in the $\Delta\mathbf{x}$ direction will be lower or equal to the value of the slowest decreasing subfunction for any $0 < t \leq t'$. Since the slowest decreasing subfunction actually decreases for any $t' > 0$, the other subfunctions must therefore have a strictly lower value than the previous maximum. ■

Lemma 3.4. *Given $\epsilon \geq 0$, the step size t returned by Algorithm 9 satisfies $t > 0$ and*

$$f_{i,j}(\mathbf{x} + t\Delta\mathbf{x}) \leq \max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}), \quad \forall i \in [l],\ \forall j \in [m_i] \tag{3.36}$$

Proof. The for-loops on lines 4 and 5 in Algorithm 9 go through all subfunctions $f_{i,j}$ with a potentially non-zero $c_{i,j}$ (i.e. $\mathbf{a}_{i,j}^T\Delta\mathbf{x}$) because only those subfunctions that depend on a non-zero $\Delta\mathbf{x}$ component could have non-zero $c_{i,j}$. If a subfunction $f_{i,j}$ decreases or is constant in the $\Delta\mathbf{x}$ direction, it could not violate the condition (3.36). Therefore we are interested only in those that increase in the $\Delta\mathbf{x}$ direction, i.e. have $c_{i,j} > 0$.

For these subfunctions, the algorithm enforces that

$$t \leq \frac{1}{c_{i,j}} \left(\max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}) - f_{i,j}(\mathbf{x}) \right), \tag{3.37}$$

which can be rewritten as

$$f_{i,j}(\mathbf{x}) + c_{i,j} t \leq \max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}), \tag{3.38}$$

where the LHS equals $f_{i,j}(\mathbf{x} + t\Delta\mathbf{x})$. Thus, they do not increase above the previous value of the maximum.

Additionally, Proposition 3.5 gives us

$$\max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}) - f_{i,j}(\mathbf{x}) > \epsilon \geq 0, \tag{3.39}$$

therefore the value t satisfying the defined condition can be found anywhere between

$$0 \leq t \leq \inf_{i,j|c_{i,j}>0} \frac{\max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}) - f_{i,j}(\mathbf{x})}{c_{i,j}}, \tag{3.40}$$

which allows for $t > 0$ because both the nominator and denominator on the RHS are positive numbers.

Furthermore, if $\epsilon > 0$, we could restrict the possible range of values even to

$$0 \leq t \leq \inf_{i,j|c_{i,j}>0} \frac{\epsilon}{c_{i,j}}, \tag{3.41}$$

which would also allow $t > 0$.

Observe that if there were no increasing subfunctions, the step size would not be limited by the mentioned condition at all and the lemma would hold trivially. ■

Theorem 3.4. *The value t returned by Algorithm 9 satisfies*

$$f(\mathbf{x} + t\Delta\mathbf{x}) < f(\mathbf{x}). \tag{3.42}$$

Proof. The algorithm initializes t as it is mentioned in Lemma 3.3, which satisfies $t > 0$ and for any $0 < t' \leq t$

$$f_{i^*,j}(\mathbf{x} + t'\Delta\mathbf{x}) < \max_{j'=1}^{m_i} f_{i^*,j'}(\mathbf{x}), \quad \forall j \in [m_{i^*}] \tag{3.43}$$

therefore also

$$\max_{j=1}^{m_{i^*}} f_{i^*,j}(\mathbf{x} + t'\Delta\mathbf{x}) < \max_{j'=1}^{m_i} f_{i^*,j'}(\mathbf{x}). \tag{3.44}$$

Next, the value of t can be lowered to t' during the subsequent loops so that it

satisfies

$$f_{i,j}(\mathbf{x} + t'\Delta\mathbf{x}) \le \max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}), \quad \forall i \in [l], \forall j \in [m_i] \tag{3.45}$$

while still being strictly greater than zero, as given by Lemma 3.4. Then, the value t' is returned. The previous inequality implies

$$\max_{j=1}^{m_i} f_{i,j}(\mathbf{x} + t'\Delta\mathbf{x}) \le \max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}), \quad \forall i \in [l] - \{i^*\}. \tag{3.46}$$

Now, we can add the inequalities (3.44) and (3.46) and obtain

$$\sum_{i=1}^{l} \max_{j=1}^{m_i} f_{i,j}(\mathbf{x} + t'\Delta\mathbf{x}) < \sum_{i=1}^{l} \max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}), \tag{3.47}$$

which is exactly the required formula. ■

3.2.4 Finite Number of Iterations

In this section, we will show that for any $\epsilon > 0$, there is a constant $\beta > 0$ such that the function value after each iteration decreases at least by β, which depends only on the inputs and not on the internal state of the algorithm. We require ϵ to be positive because otherwise, the step size could be infinitesimally small and there would be no bound.

Bound on Maximum Slope

Assume that we are performing an iteration of the algorithm and the current point is not locally ϵ-consistent, then we run Algorithm 7 to find the direction $\Delta\mathbf{x}$. We will find a bound on the absolute value of the elements of the $\mathbf{c}$ and $\Delta\mathbf{x}$ vectors. To do so, we will first find the lowest non-zero absolute value of an element from the vectors $\mathbf{a}$ and the highest absolute value of an element, formally

$$a^- = \min_{\substack{i,j,k \\ a_{i,j,k} \neq 0}} |a_{i,j,k}|, \tag{3.48a}$$

$$a^+ = \max_{i,j,k} |a_{i,j,k}|. \tag{3.48b}$$

Using these constants, we define their ratio as a value α,

$$\alpha = \frac{a^+}{a^-}, \tag{3.49}$$

which is greater or equal to 1 because $a^+ \ge a^-$. It is assumed that there is at least one non-zero value in the $\mathbf{a}$ vectors.

Next, we will define an infinite sequence of vectors $\mathbf{q}^1, \mathbf{q}^2, \mathbf{q}^3, ...$, where $\mathbf{q}^r \in \mathbb{N}^r$, which means that the vectors do not have the same length. The first vector is a scalar $\mathbf{q}^1 = 1 \in \mathbb{N}^1$ and the following vectors will be defined inductively, always based on the previous one. If we are given $\mathbf{q}^r$, then the elements of $\mathbf{q}^{r+1}$ satisfy

$$q_i^{r+1} = \begin{cases} q_1^r + 1 & \text{if } i = 1 \\ q_{i-1}^r + q_i^r & \text{if } 2 \leq i \leq r \\ q_r^r & \text{if } i = r+1 \end{cases} \tag{3.50}$$

Lemma 3.5. *After the r-th subfunction ($r \geq 1$) was processed in the main while loop of Algorithm 7, then the vectors $\mathbf{c}$ and $\Delta\mathbf{x}$ satisfy*

$$\sum_{k\in[n]} |\Delta x_k| \leq \frac{1}{a^-} \sum_{i=1}^{r} q_i^r \cdot \alpha^{i-1} \tag{3.51a}$$

$$\max_{i\in[l]} \max_{j\in[m_i]} |c_{i,j}| \leq \sum_{i=1}^{r} q_i^r \cdot \alpha^i. \tag{3.51b}$$

Proof. The claim will be proven by induction. Initially, before any subfunction is processed, the values are zero, i.e. $\mathbf{c} = \mathbf{0}$ and $\Delta\mathbf{x} = \mathbf{0}$.

In the base case, we let $r = 1$ and process $f_{i,j}$. As already said before, the first processed subfunction is from the cluster i^*, so $i = i^*$ and the condition on $c_{i,j}$ is not satisfied. Then, the value of $\Delta x_{p(i,j)}$ is set from 0 to $-1/a_{i,j,p(i,j)}$, which in absolute value satisfies

$$\sum_{k\in[n]} |\Delta x_k| = \frac{1}{|a_{i,j,p(i,j)}|} \leq \frac{1}{a^-} = \frac{1}{a^-} \sum_{i=1}^{1} q_i^1 \cdot \alpha^{i-1}. \tag{3.52}$$

The fraction $1/|a_{i,j,p(i,j)}|$ also corresponds to $\sum_{k\in[n]} |\Delta x_k|$ because the other components of $\Delta\mathbf{x}$ are zero. Next, some $\mathbf{c}$ values will be set to non-zero, but maximally to a^+ times the change in $\Delta x_{p(i,j)}$, i.e.

$$\max_{i\in[l]} \max_{j\in[m_i]} |c_{i,j}| \leq a^+ |\Delta x_{p(i,j)}| = \frac{a^+}{|a_{i,j,p(i,j)}|} \leq \frac{a^+}{a^-} = \alpha = \sum_{i=1}^{1} q_i^1 \cdot \alpha^i. \tag{3.53}$$

So we can conclude that for $r = 1$, the inequalities hold.

Now, we will continue with the inductive step. We would like to show that the inequalities hold for $r+1$ ($r \geq 1$) if we know that they hold for r. Before the $(r+1)$-th subfunction $f_{i',j'}$ is processed, the largest possible $|c_{i,j}|$ value is $\sum_{i=1}^{r} q_i^r \cdot \alpha^i$. Therefore the maximal possible change of $\Delta x_{p(i',j')}$ is

$$\Big(1 + \sum_{i=1}^{r} q_i^r \cdot \alpha^i\Big) / |a_{i',j',p(i',j')}|, \tag{3.54}$$

which happens if $i^* = i'$. So, the sum of absolute values of $\Delta\mathbf{x}$ components also increases at most by this value, i.e.

$$\sum_{k\in[n]} |\Delta x_k| \leq \Big(1 + \sum_{i=1}^{r} q_i^r \cdot \alpha^i\Big)/|a_{i',j',p(i',j')}| + \sum_{k\in[n]} |\Delta x_k^{\text{prev}}|, \tag{3.55}$$

where $\Delta\mathbf{x}^{\text{prev}}$ is the value of $\Delta\mathbf{x}$ before the update was performed in the $(r+1)$-th iteration. We can use the bound from the previous iteration and the fact that $|a_{i',j',p(i',j')}| \geq a^-$ and see that the sum of absolute values of $\Delta\mathbf{x}$ components after $(r+1)$-th iteration is bounded by

$$\sum_{k\in[n]} |\Delta x_k| \leq \frac{1}{a^-}\Big(1 + \sum_{i=1}^{r} q_i^r \cdot \alpha^i\Big) + \frac{1}{a^-}\sum_{i=1}^{r} q_i^r \cdot \alpha^{i-1} \tag{3.56a}$$

$$= \frac{1}{a^-}\Big(1 + \sum_{i=2}^{r+1} q_{i-1}^r \cdot \alpha^{i-1} + \sum_{i=2}^{r} q_i^r \cdot \alpha^{i-1} + q_1^r \cdot \alpha^0\Big) \tag{3.56b}$$

$$= \frac{1}{a^-}\Big((1 + q_1^r)\alpha^0 + \sum_{i=2}^{r}(q_{i-1}^r + q_i^r) \cdot \alpha^{i-1} + q_r^r \cdot \alpha^r\Big) \tag{3.56c}$$

$$= \frac{1}{a^-}\sum_{i=1}^{r+1} q_i^{r+1} \cdot \alpha^{i-1}, \tag{3.56d}$$

where the last term is the upper bound that was required by the lemma. Similarly as in the base case, any $c_{i,j}$ value could change its value by at most

$$a^+\Big(1 + \sum_{i=1}^{r} q_i^r \cdot \alpha^i\Big)/|a_{i',j',p(i',j')}|, \tag{3.57}$$

therefore after the update, it holds that

$$\max_{i\in[l]}\max_{j\in[m_i]} |c_{i,j}| \leq a^+\Big(1 + \sum_{i=1}^{r} q_i^r \cdot \alpha^i\Big)/|a_{i',j',p(i',j')}| + \max_{i\in[l]}\max_{j\in[m_i]} |c_{i,j}^{\text{prev}}| \tag{3.58a}$$

$$\leq \frac{a^+}{a^-}\Big(1 + \sum_{i=1}^{r} q_i^r \cdot \alpha^i\Big) + \sum_{i=1}^{r} q_i^r \cdot \alpha^i \tag{3.58b}$$

$$= \alpha + \sum_{i=2}^{r+1} q_{i-1}^r \cdot \alpha^i + \sum_{i=2}^{r} q_i^r \cdot \alpha^i + q_1^r \cdot \alpha \tag{3.58c}$$

$$= (q_1^r + 1)\alpha + \sum_{i=2}^{r}(q_i^r + q_{i-1}^r) \cdot \alpha^i + q_r^r \cdot \alpha^r \tag{3.58d}$$

$$= \sum_{i=1}^{r+1} q_i^{r+1} \cdot \alpha^i, \tag{3.58e}$$

which was the bound required by the lemma. ■

Observe that because the amount of processed subfunctions in the algorithm is at most $M = \sum_{i\in[l]} m_i$, which is the total amount of all subfunctions in the SMAF, it will always hold that

$$\sum_{k\in[n]} |\Delta x_k| \le \frac{1}{a^-} \sum_{i=1}^{M} q_i^M \cdot \alpha^{i-1} \tag{3.59a}$$

$$\max_{i\in[l]} \max_{j\in[m_i]} |c_{i,j}| \le \sum_{i=1}^{M} q_i^M \cdot \alpha^i. \tag{3.59b}$$

Additionally, notice that if all M subfunctions are in the augmenting DAG, then they are all active. But this means that the algorithm found a decreasing direction for the whole function f. This means that if the function f is bounded, it is correct to assume that there is at most $M-1$ subfunctions in the augmenting DAG.

To give the reader a better insight into the bounds (3.59), we will discuss their properties. It generally holds that $\alpha \ge 1$. If $\alpha > 1$, then the values of the bound obviously increase exponentially with M. But even for $\alpha = 1$, the bound is still exponential in M since

$$\sum_{i=1}^{M} q_i^M = 2^M - 1. \tag{3.60}$$

Example 3.1. To demonstrate that the values of both $\Delta \mathbf{x}$ and $\mathbf{c}$ could become exponentially large, consider the following example with $l = 1$, $m_1 = 5$, $n = 4$, $i^* = 1$:

$$\begin{aligned}
\mathbf{a}_{1,1} &= (-1,2,2,2) & p(1,1) &= 1 \\
\mathbf{a}_{1,2} &= (0,-1,2,2) & p(1,2) &= 2 \\
\mathbf{a}_{1,3} &= (0,0,-1,2) & p(1,3) &= 3 \\
\mathbf{a}_{1,4} &= (0,0,0,-1) & p(1,4) &= 4 \\
\mathbf{a}_{1,5} &= (2,2,2,2) & p(1,5) &= \mathsf{INACTIVE}
\end{aligned}$$

The algorithm will in this case return $\Delta \mathbf{x} = (27,9,3,1)$ and $\mathbf{c}_1 = (-1,-1,-1,-1,80)$. Because $\alpha = 2$ and there were 4 subfunctions in the augmenting DAG, the bounds in this case are

$$\sum_{k=1}^{4} |\Delta x_k| \le \sum_{i=1}^{4} q_i^4 \cdot 2^{i-1} = 40 \tag{3.62a}$$

$$\max_{j=1}^{5} |c_{1,j}| \le \sum_{i=1}^{4} q_i^4 \cdot 2^i = 80. \tag{3.62b}$$

This example could be scaled arbitrarily. □

Minimal Decrease of the Objective Function

In this section, we will derive the minimal decrease in one iteration of the algorithm using the bound on $|c_{i,j}|$ presented in the previous section. It was mentioned in the proof of Lemma 3.3 that the initialization of t returns a value $t > 0$. Then, this value can be lowered so that no maximum becomes higher than its previous value.

If the assignment on line 7 in Algorithm 9 does not lower t, then we can be sure that the the maximum of cluster i^* is lowered by at least[3]

$$\frac{\epsilon}{1 + \max_{j \in [m_{i^*}]} c_{i^*,j}}, \tag{3.63}$$

because of the minimum difference of active and inactive subfunction values, which was showed in Proposition 3.5. The reasoning for this is that all the active subfunctions decrease at least by $\mathbf{a}_{i^*,j}^T \mathbf{x} \leq -1$, so none of them will be higher. Next, the inactive subfunctions that decrease necessarily had their value lower than ϵ even in the original point and their value will further decrease. The inactive non-decreasing subfunctions also cannot have higher value because their value is not higher than the value of the slowest decreasing subfunction, as it was shown in the proof of Lemma 3.3. Because the second condition for update of t was not violated, no maximum increases by Lemma 3.4, therefore the function value of f decreases by the specified value using the same reasoning as in the proof of Theorem 3.4.

On the other hand, if the assignment on line 7 in Algorithm 9 actually forces the value of t to decrease below the specified value, then the function value in the cluster i^* decreases by at least

$$\frac{\epsilon}{1 + \max_{i,j} c_{i,j}}, \tag{3.64}$$

which follows from the update in the algorithm.

To conclude this section, in one case, the function f decreases by at least (3.63), and in the other case, the function decreases at least by (3.64). Using the bound on $c_{i,j}$ values allows us to formulate Theorem 3.5.

Theorem 3.5. *Given a SMAF f, the algorithm lowers the function value in each iteration at least by*

$$\beta = \frac{\epsilon}{1 + \sum_{i=1}^{M} q_i^M \cdot \alpha^i}, \tag{3.65}$$

where M is the total amount of subfunctions in the problem. That means

$$f(\mathbf{x} + t\Delta\mathbf{x}) \leq f(\mathbf{x}) - \beta. \tag{3.66}$$

[3]If $\max_{j \in [m_{i^*}]} c_{i^*,j} \leq -1$, then it is lowered by infinity, because the problem is unbounded.

Maximum Amount of Iterations

Assume that the function f has a minimum in point $\mathbf{x}^*$ with a finite value $f(\mathbf{x}^*)$ and that our algorithm was initialized to start in $\mathbf{x} = \mathbf{0}$ with function value $f(\mathbf{0}) = \sum_{i=1}^{l} \max_{j=1}^{m_i} b_{i,j}$, which is also finite.

Using the previously shown properties of the algorithm and assuming that the function f has a minimum, we can easily see that after at most

$$\left\lfloor \frac{f(\mathbf{0}) - f(\mathbf{x}^*)}{\beta} \right\rfloor = \left\lfloor \frac{f(\mathbf{0}) - f(\mathbf{x}^*)}{\epsilon} \Big(1 + \sum_{i=1}^{M} q_i^M \cdot \alpha^i\Big) \right\rfloor \tag{3.67}$$

steps, the algorithm reaches a locally ϵ-consistent point $\mathbf{x}$.

The worst-case bounds derived in this section are of course very loose for typical instances, for which the number of iterations is empirically observed to be much lower.

3.3 Integer Version of the Algorithm

The previous sections assume exact arithmetic, which is not the case of usual floating point number representation in computers. It would be enough to use exact rational arithmetic but it could be time-consuming and significantly prolong the runtime of the algorithm.

For example, after assigning $a \leftarrow a+b$ for $b \neq 0$, the value of a might not change when the floating point arithmetic with limited precision is used – this happens for sufficiently large a and small $|b|$. The risk exists in this algorithm – for example the step size t can be exponentially small w.r.t. the total amount of subfunctions, which is large. Theoretically, it could happen that for a small step size t, some components of $\mathbf{x}$ or $\mathbf{y}$ would not be updated and these errors would accumulate during the run of the algorithm, leading to a wrong result. Or, after a corrupt update, some subfunctions that should have become active would not.

In this section, we will present an altered version of the algorithm that can overcome the issues of limited precision. Namely, we will view all the inputs and all the variables of the algorithm as integers and therefore completely avoid issues with precision. This approach can be also easily scaled to use fixed-point arithmetic.

More formally, we now assume that $\mathbf{a}_{i,j} \in \mathbb{Z}^n$ and $b_{i,j} \in \mathbb{Z}$ and we would like to find a locally ϵ-consistent point $\mathbf{x} \in \mathbb{Z}^n$. Observe that in the continuous case, such a point always existed but now it no longer has to.

Example 3.2. Assume a one-dimensional case with $f(x) = \max\{-2x + 2, x - 3\} = \max\{f_{1,1}(x), f_{1,2}(x)\}$, then it has a minimum $x = 5/3 \in \mathbb{Q}$, but there is no locally

ϵ-consistent point $x \in \mathbb{Z}$ for any $\epsilon > 1$, because $|f_{1,1}(1) - f_{1,2}(1)| = 2$ and $|f_{1,1}(2) - f_{1,2}(2)| = 1$ and in other integer arguments, the difference is even larger. $\square$

3.3.1 Changes in the Algorithm

Because we are using integers, we can assume that the multiplication, addition, subtraction and comparison of numbers always returns the correct result[4]. The only problem is division, where we should decide whether the result should be rounded up, down or treated in a different manner.

It is easy to see that the local consistency algorithm along with the activate and deactivate subfunction can work with integer values without any changes. The first issue comes with the direction calculation algorithm. When $c_{i,j}$ is larger than $-\llbracket i = i^* \rrbracket$, we update

$$\Delta x_k \leftarrow \Delta x_k - (\llbracket i = i^* \rrbracket + c_{i,j})/a_{i,j,k} \tag{3.68}$$

on line 16 in Algorithm 7, which could result in a non-integer value of Δx_k. To avoid this, we should choose an integer value for Δx_k that would "fix" $c_{i,j}$ value to a correct one. The solution to this is to round the value of $|(\llbracket i = i^* \rrbracket + c_{i,j})/a_{i,j,k}|$ up to the closest higher integer (or keep it if it already is an integer) and then change the value of Δx_k by either adding it (if $a_{i,j,k} < 0$) or subtracting it (if $a_{i,j,k} > 0$). This can be written compactly as

$$\Delta x_k \leftarrow \Delta x_k - \operatorname{sign}(a_{i,j,k}) \left\lceil \frac{\llbracket i = i^* \rrbracket + c_{i,j}}{\operatorname{sign}(a_{i,j,k}) \cdot a_{i,j,k}} \right\rceil. \tag{3.69}$$

It can be shown that if the vector $\Delta \mathbf{x}$ is calculated in this manner, it still satisfies the required properties and the proof is analogous to the one for the continuous version.

Another part that heavily uses division is the line search procedure. In the presented algorithm, the step size t is initialized on line 3 in Algorithm 9. This initialization has the important property that on the interval $[0, t]$, the value of the maximum in cluster i^* is decreasing[5] and no non-decreasing subfunction can become 0-active[6] on the interval $[0, t)$. We will replace the previous assignment with

$$t \leftarrow \inf \left\{ \left\lfloor \frac{f_{i^*, j_{i^*}^{\text{slowest}}}(\mathbf{x}) - f_{i^*,j}(\mathbf{x})}{c_{i^*,j} - c_{i^*, j_{i^*}^{\text{slowest}}}} \right\rfloor \middle| j \in [m_{i^*}], c_{i^*,j} \geq 0 \right\}, \tag{3.70}$$

[4]Overflow could happen in these operations, however, we never observed this. Yet, to achieve complete correctness, we could check the results of such operations for overflow and if it happened, it could be treated in the same manner as zero step size.

[5]This is more discussed in Section 3.1.4 that compares various line search methods.

[6]By 0-active, we understand that it would be active as defined in Section 2.2.2. But notice that such subfunction could become ϵ-active on this interval given $\epsilon > 0$.

and then the previous claims still hold, but the value of t might become smaller, even zero. If $t = 0$, it is an issue that we will solve later.

Similarly, we can replace the update on line 7 with

$$t \leftarrow \min\left\{t, \left\lfloor \frac{1}{c_{i,j}} \left(\max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}) - f_{i,j}(\mathbf{x})\right)\right\rfloor\right\}, \tag{3.71}$$

where the value of t still satisfies that after the step is performed, the value of the maximum in any cluster does not increase but it could happen even here that the returned value is zero.

3.3.2 Assuring Consistency and Dealing with Zero Step Size

In the integer setting of the algorithm, there arise two potential issues. First, a locally ϵ-consistent point with integer coordinates may not exist for each $\epsilon > 0$. Second, calculated step size t may equal zero.

To deal with the first issue, we will treat ϵ as an output parameter of the algorithm. It will be initialized at the start of the algorithm to a large value and then, the algorithm is started with this ϵ. If it finds a locally ϵ-consistent point, we lower the value of ϵ to e.g. $\left\lfloor \frac{\epsilon}{2} \right\rfloor$ and continue minimizing from the current point. The same is done whenever $t = 0$ – we simply interrupt the current minimization and continue with lowered ϵ. The whole procedure ends when $\epsilon = 0$ and either a locally consistent point was found or $t = 0$. This procedure is formalized in Algorithm 14.

The algorithm uses function modifiedMinimize$(\epsilon, \mathbf{x})$ that tries to minimize the given SMAF with ϵ from the point $\mathbf{x}$. It uses the integer modifications that we presented before. If it finds a locally ϵ-consistent point, then it returns it. If it calculates $t = 0$, it stops the descent and returns the current point. Similarly as the original continuous version, it may detect unbounded input.

Algorithm 14: Integer SMAF minimization algorithm

```
1  Function minimizeInteger(l, m, a, b) is
2      x ← 0;
3      ε ← max_{i,j} b_{i,j} − min_{i,j} b_{i,j};
4      loop
5          result ← modifiedMinimize(ε, x);
6          if result = unbounded then
7              return unbounded;
8          x ← result;
9          if ε = 0 then
10             return x;
11         ε ← ⌊ε/2⌋;
```

Observe that the function modifiedMinimize always terminates after a finite number of iterations if the input is bounded, because if $t \neq 0$, then $t \geq 1$ and the same reasoning as in the proof of correctness can be done to show that there is a bound on minimal decrease of the function. If $t = 0$, then the calculations are stopped and ϵ decreased. Because the inner function always terminates, we can say that Algorithm 14 also always terminates because the value of ϵ will in at most

$$\left\lfloor \log_2 \left(\max_{i,j} b_{i,j} - \min_{i,j} b_{i,j} \right) \right\rfloor \tag{3.72}$$

main loop iterations become zero.

But it is clear that the point $\mathbf{x}$ returned by Algorithm 14 is not necessarily ϵ-consistent for $\epsilon = 0$. It is only in the case when the last run of modifiedMinimize was terminated because of the point being consistent. If it was terminated because of $t = 0$, then it is not locally 0-consistent. Thus, we need to find an $\epsilon \geq 0$ such that the returned point $\mathbf{x}$ is locally ϵ-consistent.

We will define a set

$$E = \left\{ \max_{j'=1}^{m_i} f_{i,j'}(\mathbf{x}) - f_{i,j}(\mathbf{x}) \,\middle|\, i \in [l], j \in [m_i] \right\}, \tag{3.73}$$

which contains all the "differences" of all subfunctions to the maxima in their corresponding clusters. This set contains at most $M - l + 1$ elements (because all the values corresponding to 0-active subfunctions are zeros). Observe that if the function f is bounded, then there will be an $\epsilon \in E$ such that the returned point $\mathbf{x}$ is locally ϵ-consistent – obviously, it would have to hold at least for $\epsilon = \max E$, when all the subfunctions would be ϵ-active. Otherwise, the function f would be unbounded. Notice that there are infinitely many ϵ for which the point $\mathbf{x}$ is locally ϵ-consistent and we will be interested in finding the smallest one from them.

Deciding whether a point is locally ϵ-consistent depends only on which subfunctions are ϵ-active, therefore it is enough to try all the values of ϵ that are in the set E.

Proposition 3.6. *For any $\epsilon' \geq 0$, there is an $\epsilon \in E$ such that the set of ϵ'-active subfunctions is the same as the set of ϵ-active subfunctions, i.e.*

$$I^{\epsilon'}(\mathbf{x}) = I^{\epsilon}(\mathbf{x}). \tag{3.74}$$

This proposition allows to formulate Algorithm 15 that first constructs the set E, sorts the values that it contains and tries them in ascending order and for each, it finds out whether the point is ϵ-consistent or not. If it is, then it returns the current ϵ that is surely the minimal one. Otherwise, it activates subfunctions that become

ϵ-active with the next one and continues.

Algorithm 15: Finding smallest ϵ

```
1  Function findSmallestEpsilon(l, m, a, b, x) is
2      E ← { max_{j'=1}^{m_i} f_{i,j'}(x) − f_{i,j}(x) | i ∈ [l], j ∈ [m_i] };
3      ε_1, ..., ε_R ← sort(E);
4      p ← INACTIVE;
5      for (i, j) ∈ I^{ε_1}(x) do
6          activate(f_{i,j});
7      for r = 1 to R do
8          consistent ← calculateConsistency();
9          if consistent then
10             return ε_r;
11         for (i, j) ∈ I^{ε_{r+1}}(x) − I^{ε_r}(x) do
12             activate(f_{i,j});
```

3.3.3 Optimality Criteria

We have also looked into the properties of the integer version of the algorithm that allowed us to formulate theorems that can in some cases determine whether the returned point is the optimum or only a locally ϵ-consistent point.

Theorem 3.6. *If the integer version of the algorithm ends with a point* $\mathbf{x}$ *and* $\epsilon > 0$*, then* $\mathbf{x}$ *is not a minimizer of function* f*.*

Proof. If the algorithm ended with $\epsilon > 0$, it means that the point $\mathbf{x}$ is not locally 0-consistent, therefore there exists a decreasing direction which can lower the function value – and this was not performed because the calculated step size was zero. The point is therefore not the optimum. ■

Next, we will define the vector $\boldsymbol{\psi} \in \mathbb{R}^n$ that will contain the sum of coefficients $a_{i,j,k}$ of all subfunctions $f_{i,j}$ with $p(i,j) = \mathsf{ALIVE}$ for each coordinate $k \in [n]$. Formally,

$$\psi_k = \sum_{i \in [l]} \sum_{\substack{j \in [m_i] \\ p(i,j) = \mathsf{ALIVE}}} a_{i,j,k}. \tag{3.75}$$

Theorem 3.7. *Assume that the integer version of the algorithm ended with a point* $\mathbf{x}$ *and* $\epsilon = 0$*, and after the local consistency algorithm was run for the last time, it holds that*

- *there is exactly one subfunction* $f_{i,j}$ *that is alive in each cluster,*

- $\psi_k = 0$ *for all* $k \in [n]$

then $\mathbf{x}$ *is a minimizer of function* f.

Proof. The previous theorem will be proven by contradiction. We will assume that there is a direction $\Delta\mathbf{x}$ such that the function f decreases in this direction.

Notice that the subfunctions that are active, but killed, can be made inactive by finding a decreasing direction for them. This is possible because they define a DAG and all of them could be decreased at least by an infinitesimally small amount to become inactive with $\epsilon = 0$ without any other subfunction becoming alive[7]. Therefore, we can assume that we have done such step and only the subfunctions that remained alive are active now. Notice that this step did not change the value of the function because no maximum changed its value. If this point is shown to be optimal, then the previous was surely too.

So, we have only one subfunction $f_{i,j(i)}$ in each cluster i that is alive and its slope in the $\Delta\mathbf{x}$ direction is $\mathbf{a}_{i,j(i)}^T\Delta\mathbf{x}$. Because the function f is decreasing in this direction, it must hold that

$$\sum_{i\in[l]} \mathbf{a}_{i,j(i)}^T \Delta\mathbf{x} < 0, \tag{3.76}$$

which means that the overall slope of the subfunctions should be negative for the function f to decrease. Otherwise, it would be non-decreasing in this direction. But, it holds that

$$\begin{aligned}
\sum_{i\in[l]} \mathbf{a}_{i,j(i)}^T \Delta\mathbf{x} &= \sum_{i\in[l]}\sum_{k\in[n]} a_{i,j(i),k}\Delta x_k = \sum_{k\in[n]}\sum_{i\in[l]} a_{i,j(i),k}\Delta x_k &\text{(3.77a)}\\
&= \sum_{k\in[n]}\sum_{i\in[l]} \sum_{\substack{j\in[m_i]\\ p(i,j)=\mathsf{ALIVE}}} a_{i,j,k}\Delta x_k &\text{(3.77b)}\\
&= \sum_{k\in[n]} \Delta x_k \sum_{i\in[l]} \sum_{\substack{j\in[m_i]\\ p(i,j)=\mathsf{ALIVE}}} a_{i,j,k} = \sum_{k\in[n]} \Delta x_k \psi_k = 0 &\text{(3.77c)}
\end{aligned}$$

which leads to a contradiction. There could be no direction $\Delta\mathbf{x}$ in which the function f decreases from the point, therefore it has the optimal value. ■

Theorem 3.8. *Assume that the integer version of the algorithm ended with a point* $\mathbf{x}$ *and* $\epsilon = 0$, *and after the local consistency algorithm was run for the last time, it holds that*

- *there is exactly one subfunction* $f_{i,j}$ *that is alive in each cluster,*
- *there is* $k \in [n]$ *with* $\psi_k \neq 0$

[7]This is theoretically possible only with unlimited precision, but the used arithmetic does not influence the optimality of a point $\mathbf{x}$.

then $\mathbf{x}$ *is not a minimizer of* f*.*

Proof. Following the same reasoning as in the previous proof, we can show that such point $\mathbf{x}$ is not an optimum by finding a direction $\Delta\mathbf{x}$ for which it holds that the value of the function f decreases.

Let us define the direction as $\Delta\mathbf{x} = -\boldsymbol{\psi}$, then it holds that

$$\sum_{i\in[l]} \mathbf{a}_{i,j(i)}^T \Delta\mathbf{x} = \sum_{k\in[n]} \Delta x_k \sum_{i\in[l]} a_{i,j(i),k} = \sum_{k\in[n]} -\psi_k^2 < 0, \tag{3.78}$$

where the last inequality is strict because there is $\psi_k \neq 0$. Therefore, we have found the decreasing direction and the current point is therefore not optimal. ■

The above theorems are summarised in Figure 3.4, where it can be seen that the case when $\epsilon = 0$ and there is a cluster with multiple alive subfunctions is not covered. But we will see later that even the first two theorems are enough to determine the optimality of most instances we will deal with.

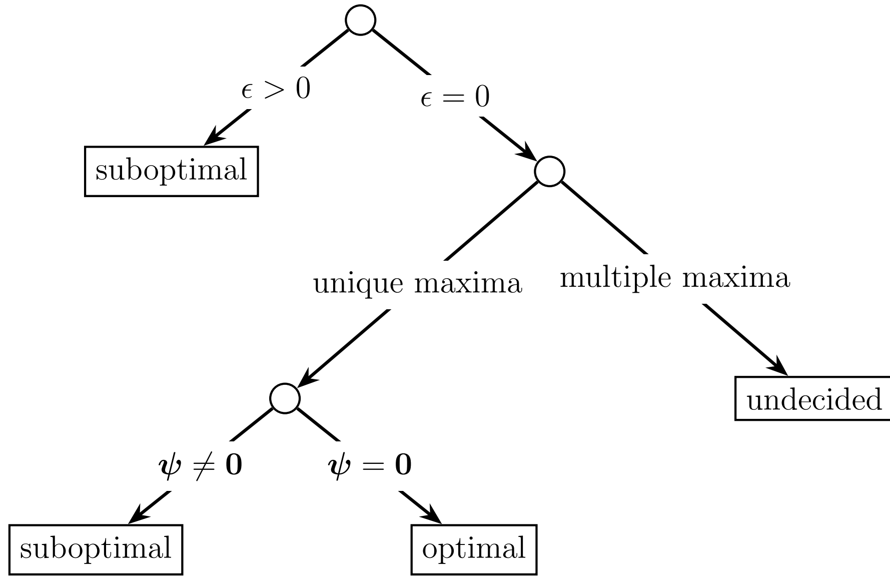


Figure 3.4: Flowchart on deciding optimality.

Chapter 4

Implementation and Complexity

In this chapter, we will first describe how the input data should be stored and accessed along with practical improvements of the formal pseudocode that will improve its efficiency. Next, we will calculate the space complexity of the presented implementation and time complexity of one iteration. Finally, we present the actual interface of the implemented minimization algorithm in C++ and the structure of the input and output data files.

4.1 Implementation Details

Here, we give the implementation details of the previously presented pseudocode and used data structures.

4.1.1 Storing Input Data

Concerning the input data, it is straightforward how to store the vectors $\mathbf{m}$ and $\mathbf{b}$ but it is necessary to deal with efficiently storing the vectors $\mathbf{a}$. As it was presented in Algorithm 3, the vectors are likely to be sparse. Therefore, we should store each one as a list of non-zero $a_{i,j,k}$ values along with the coordinates k that these values correspond to. Observe that this kind of storage automatically allows efficient querying of $N(f_{i,j})$ elements and the corresponding coefficients a, which is frequently done.

Similarly, we could store for each coordinate k the non-zero coefficients $a_{i,j,k}$ in a list along with the (i, j) indices. This structure will be used for querying $N(x_k)$.

Later in this section, we will show that it is never necessary to access the values $a_{i,j,k}$ randomly, but always in the scope of iterating over $N(x_k)$ or $N(f_{i,j})$.

4.1.2 Efficiency Improvements

First of all, it is important to note that on each place in the algorithm where the value of a subfunction is queried in the current point, we should not calculate it every time but use the pre-defined vector $\mathbf{y}$ that satisfies $y_{i,j} = f_{i,j}(\mathbf{x})$ and is updated after each iteration. In addition, we keep a vector $\boldsymbol{\tau} \in \mathbb{R}^l$ such that

$$\tau_i = \max_{j=1}^{m_i} f_{i,j}(\mathbf{x}), \tag{4.1}$$

so that we can also replace the occurrences of cluster maxima and not evaluate them each time.

Local Consistency Algorithm

In the local consistency algorithm, we introduce some auxiliary variables for efficiency. One of these vectors, $\boldsymbol{\varrho} \in \mathbb{N}^l$, contains in each ϱ_i the amount of remaining subfunctions that are alive in cluster i,

$$\varrho_i = \sum_{j=1}^{m_i} [\![p(i,j) = \mathsf{ALIVE}]\!]. \tag{4.2}$$

The initial values of this vector are zeros but whenever a subfunction $f_{i,j}$ becomes alive, we need to increase the corresponding component and do the opposite when it becomes inactive or killed. The same condition that is checked on lines 2 and 13 in Algorithm 6 is equivalent to checking whether some $\varrho_i = 0$, where i are the clusters of the subfunctions that were just killed. Updating the vector does not create any overhead in the sense of asymptotic time complexity because in each assignment to the vector $\mathbf{p}$, we will update at most one value in $\boldsymbol{\varrho}$.

Next, to see whether a coordinate is consistent, we do not need to iterate over all the subfunctions but instead, we can use vectors $\boldsymbol{\gamma}^+, \boldsymbol{\gamma}^- \in \mathbb{N}^n$, where γ_k^+ (resp. γ_k^-) is the amount of subfunctions $f_{i,j}$ that are alive and their corresponding coefficient $a_{i,j,k}$ is positive (resp. negative). Formally,

$$\gamma_k^+ = \sum_{i=1}^{l} \sum_{j=1}^{m_i} [\![(a_{i,j,k} > 0) \land (p(i,j) = \mathsf{ALIVE})]\!] \tag{4.3a}$$

$$\gamma_k^- = \sum_{i=1}^{l} \sum_{j=1}^{m_i} [\![(a_{i,j,k} < 0) \land (p(i,j) = \mathsf{ALIVE})]\!]. \tag{4.3b}$$

Testing for local consistency of a single variable x_k is then equivalent to checking

$$(\gamma_k^+ = 0) \iff (\gamma_k^- = 0). \tag{4.4}$$

Similarly as with $\boldsymbol{\varrho}$, the values of $\boldsymbol{\gamma}^+$ and $\boldsymbol{\gamma}^-$ need to be updated whenever a subfunction becomes alive, i.e. evaluate

$$\gamma_k^+ \leftarrow \gamma_k^+ + 1, \quad \forall k \in N(f_{i,j}), a_{i,j,k} > 0 \tag{4.5a}$$
$$\gamma_k^- \leftarrow \gamma_k^+ + 1, \quad \forall k \in N(f_{i,j}), a_{i,j,k} < 0 \tag{4.5b}$$

and whenever a subfunction is killed or becomes inactive, it is necessary to subtract 1 from the corresponding values.

It is also important to mention that the queue Q that is used in the local consistency algorithm should be implemented so that it stores each value k at most once – i.e. if the function Q.push(k) is called and the queue already contains k, then no action should be executed.

The final remark to the local consistency algorithm is that we should have a vector $\bar{\mathbf{a}}$ with values $\bar{a}_{i,j} \in \mathbb{R} \cup \{\mathsf{UNDEFINED}\}$ for each subfunction $f_{i,j}$. Initially, all $\bar{a}_{i,j} = \mathsf{UNDEFINED}$, but whenever some $p(i,j)$ value is set to a coordinate from $[n]$, then we set $\bar{a}_{i,j} \leftarrow a_{i,j,p(i,j)}$. The benefit of this approach is that we can eliminate all randomly accessed values $a_{i,j,k}$ from the algorithm. In the pseudocode, all these values are used only while iterating through the corresponding sets $N(x_k)$ or $N(f_{i,j})$, except for some occurrences of $a_{i,j,p(i,j)}$, which can be replaced by $\bar{a}_{i,j}$. Therefore, we do not require anywhere to access a random $a_{i,j,k}$ value but only obtain these values while querying the sets N.

Decreasing Direction, Line Search, and Update

When Algorithm 7 calculates the direction $\Delta\mathbf{x}$, the DAG or the augmenting DAG should not be explicitly created, except for storing which subfunctions are in the augmenting DAG (which is stored in $\mathbf{r}$) and their in-degrees (which are stored in $\mathbf{d}$). The existence of particular edges is then determined implicitly by their definition.

During the creation of $\Delta\mathbf{x}$ vector, whenever a value is assigned to some Δx_k, we push the coordinate k into a queue Q_x, which is initially empty. And after that, when the set $N(x_k)$ is traversed, we put all the clusters i' into a queue Q_i, which was also empty at the start of direction calculation algorithm. These queues should be implemented in the same manner as the queue in the local consistency algorithm, i.e. they will contain each value at most once. Eventually, the queue Q_x will contain all the non-zero coordinates of $\Delta\mathbf{x}$ and Q_i will contain all the clusters i such that this cluster contains at least one subfunction $f_{i,j}$ that depends on a non-zero coordinate of $\Delta\mathbf{x}$.

Now, when the step size is calculated, we can replace the occurrences of $f_{i,j}(\mathbf{x})$ with $y_{i,j}$ and do the same with the cluster maxima, which can be replaced by $\boldsymbol{\tau}$. Additionally, we can read the cluster indices i from the queue Q_i to find all potentially

increasing subfunctions, as it is done in the for-loops of the line search defined in Algorithm 9.

After that, in the update procedure, we iterate over the clusters stored in Q_i again to update the corresponding values in $\mathbf{y}$ and $\boldsymbol{\tau}$ and during that, we empty the queue. Finally, the current point $\mathbf{x}$ is updated but we only need to recalculate the coordinates that are stored in the queue Q_x, which is emptied during this process.

4.2 Time and Space Complexity

In this section, we will discuss the asymptotic space complexity of the algorithm and the asymptotic time complexity of one iteration. We will assume that the vectors $\mathbf{a}_{i,j}$ are sparse, which formally means that the size of sets $N(f_{i,j})$, respectively $N(x_k)$ does not exceed a small constant K, i.e.

$$K \geq \max\left(\max_{i=1}^{l}\max_{j=1}^{m_i}|N(f_{i,j})|,\ \max_{k=1}^{n}|N(x_k)|\right), \tag{4.6}$$

thus $|N(f_{i,j})| \in O(1)$ for all subfunctions $f_{i,j}$ and $|N(x_k)| \in O(1)$ for all $k \in [n]$. We will come to the result that the asymptotic time complexity of each iteration is $O(n + M)$, which is also the asymptotic space complexity of the algorithm. We denote the total number of subfunctions in the SMAF as M.

4.2.1 Space Complexity

Among all the variables that the algorithm uses, there are some whose size is proportional to the total number of subfunctions. These are for example $\mathbf{y}$, $\mathbf{c}$, $\mathbf{d}$, $\mathbf{r}$, $\mathbf{p}$ and Q_f. There are also structures whose size is proportional to the amount of variables n, e.g. $\mathbf{x}$, $\Delta\mathbf{x}$ and Q. There are no structures whose size would be asymptotically larger than n or M.

The input is also stored as a sparse matrix, so the total space complexity of it is not higher than $O(nK + MK) = O(n + M)$. Thus, the overall asymptotic space complexity is $O(n + M)$.

4.2.2 Time Complexity

Local Consistency Algorithm

As mentioned previously in this chapter, the first condition in Algorithm 6 can be checked in $O(l)$ using the precalculated values $\boldsymbol{\varrho}$.

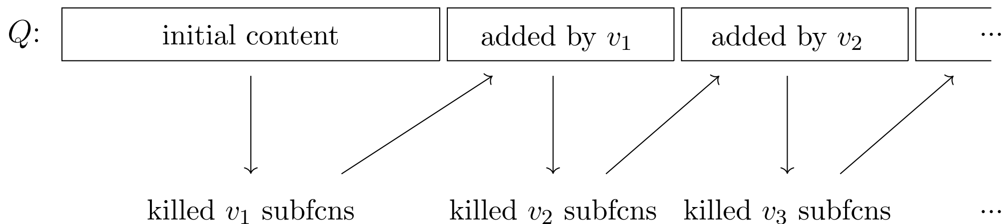


Figure 4.1: Processing coordinates in queue.

Next, we would like to estimate an upper bound on the number of loops driven by the queue Q. Assume that the queue is initialized with a non-zero amount of variables, then it could happen that even before all these variables are processed, the termination condition is satisfied in less than n loops. However, if all the initial elements of the queue are processed and the algorithm did not end yet, we can say that we have killed v_1 subfunctions until now. That means that there is at most some $K \cdot v_1$ variables that were added while the subfunctions were being killed. Then, we can use the same reasoning and see whether the algorithm ends before all these variables are processed. If it does not, we can denote v_2 as the number of subfunctions that were killed due to the previously added variables.

This idea is shown visually in Figure 4.1. In the figure, there is some initial content of the queue during which v_1 subfunctions are killed. Because these subfunctions are killed, they add at most $K \cdot v_1$ variables to the queue that kill v_2 subfunctions etc. But it is important that the algorithm ends at some point. Necessarily, the total amount of killed subfunctions until this point is lower than the total amount of subfunctions, i.e.

$$\sum_i v_i \leq M. \tag{4.7}$$

Because each killed subfunction added at most constant amount of variables into the queue, the total amount of added variables during the run of the algorithm is $O(KM)$. Assuming that there could have initially been at most n variables in the queue, there are at most $O(KM + n) = O(M + n)$ iterations of the while loop.

Each iteration of the while loop takes constant time because checking consistency is easy after introducing vectors $\boldsymbol{\gamma}^+$ and $\boldsymbol{\gamma}^-$, there is only constant amount of subfunctions that depend on the inconsistent variable and each of these subfunctions depends on constant amount of variables. The checking of the terminating condition can be done for the clusters of the subfunctions that are just being killed.

Therefore, the local consistency algorithm has asymptotic time complexity $O(n + M)$.

Decreasing Direction Calculation

In the calculation of $\Delta\mathbf{x}$ direction, the augmenting DAG is traversed in order to find the degrees of its nodes. Obviously, there can be at most M nodes in the DAG, therefore the maximum number of edges is $O(KM) = O(M)$, which means that the first traversal is at most linear w.r.t. the number of subfunctions.

Then, we go through the augmenting DAG again to calculate the direction. Because the size of augmenting DAG is at most $O(M)$ and each subfunction is processed in $O(K) = O(1)$ time, we can conclude that the whole process of calculating $\Delta\mathbf{x}$ has time complexity $O(M)$.

Line Search and Update

During the line search, we iterate over the subfunctions in the cluster i^* at most twice and then over all clusters that may contain a subfunction with non-zero slope. This procedure can be performed in $O(M)$ time because in the worst case, we would iterate over all the subfunctions.

The same can be said about the update – we might have to update all the subfunctions (and also all the values of the maxima) and all components of $\mathbf{x}$ and then iterate over the updated values to find out which subfunctions are active for the next iteration. This can be done in $O(2M+n) = O(M+n)$ time. It is important to note that the deactivate function runs in constant time $O(K)$ and the accumulated runtime of activate function will not exceed $O(M)$ because each subfunction's status could be set to ALIVE at most once and recursive resurrecting will not run more than once for each subfunction.

To conclude this section, we have estimated the time complexity of one iteration of the algorithm to be $O(n+M)$, where n is the number of variables and M is the total number of subfunctions in the SMAF.

Complexity of the Whole Algorithm

In Section 3.2.4, we introduced an upper bound that showed that the maximum number of the iterations could theoretically be exponential in the size of the input. It was out of scope of this thesis to find a better bound on its convergence rate and we do not know whether a better bound exists. However, as said above, the number of iterations was experimentally shown to be much lower than the exponential bound.

$$
\begin{array}{llllllllll}
l & n & K \\
m_1 & m_2 & \dots & m_l \\
e_{1,1} & k_1^{1,1} & a_{1,1,k_1^{1,1}} & k_2^{1,1} & a_{1,1,k_2^{1,1}} & \dots & k_{e_{1,1}}^{1,1} & a_{1,1,k_{e_{1,1}}^{1,1}} & b_{1,1} \\
e_{1,2} & k_1^{1,2} & a_{1,2,k_1^{1,2}} & k_2^{1,2} & a_{1,2,k_2^{1,2}} & \dots & k_{e_{1,2}}^{1,2} & a_{1,2,k_{e_{1,2}}^{1,2}} & b_{1,2} \\
\vdots \\
e_{1,m_1} & k_1^{1,m_1} & a_{1,m_1,k_1^{1,m_1}} & k_2^{1,m_1} & a_{1,m_1,k_2^{1,m_1}} & \dots & k_{e_{1,m_1}}^{1,m_1} & a_{1,m_1,k_{e_{1,m_1}}^{1,m_1}} & b_{1,m_1} \\
e_{2,1} & k_1^{2,1} & a_{2,1,k_1^{2,1}} & k_2^{2,1} & a_{2,1,k_2^{2,1}} & \dots & k_{e_{2,1}}^{2,1} & a_{2,1,k_{e_{2,1}}^{2,1}} & b_{2,1} \\
\vdots \\
e_{l,m_l} & k_1^{l,m_l} & a_{l,m_l,k_1^{l,m_l}} & k_2^{l,m_l} & a_{l,m_l,k_2^{l,m_l}} & \dots & k_{e_{l,m_l}}^{l,m_l} & a_{l,m_l,k_{e_{l,m_l}}^{l,m_l}} & b_{l,m_l}
\end{array}
$$

Figure 4.2: Input file format.

4.3 Interface

In this section, we describe the input format of the file that defines a SMAF and also the output format of the file that returns the result of the algorithm. We also briefly comment on our implementation in C++, namely its division into classes and description of their functionalities.

4.3.1 Input and Output File Formats

This section describes the way the input and output files are structured to allow the user to interpret them in own custom applications.

Input Format

The input format was designed so that it would be easy and efficient to load the data to an internal representation. The input file is a text file with content according to Figure 4.2.

On the first line, there should be the number of clusters l, number of variables n and a constant K that is an upper bound on the number of non-zero elements in any $\mathbf{a}_{i,j}$ and also the number of subfunctions that depend on a variable x_k.

The second line contains the sizes of clusters, i.e. how many subfunctions there are in each cluster.

Next, there are $M = \sum_{i\in[l]} m_i$ lines which define the individual subfunctions. The order of the lines is straightforward – we first define the m_1 subfunctions of the first cluster, then the m_2 subfunctions of the second one etc.

$l\ n\ \epsilon$
$x_1\ x_2\ \cdots\ x_n$
$h_1\ h_2\ \cdots\ h_l$

Figure 4.3: Output file format.

When defining a subfunction $f_{i,j}$, the first number on the corresponding line is $e_{i,j}$, which equals to $|N(f_{i,j})|$, i.e. the number of variables that this subfunction depends on. This number is followed by exactly $e_{i,j}$ pairs of values, where the first number of the pair is a coordinate k and the second number is the corresponding $a_{i,j,k}$ value. Obviously, only coordinates with non-zero $a_{i,j,k}$ values are listed and after all of them are indexed, the line is ended with the value of $b_{i,j}$. Notice that in the input files, coordinates are not numbered from 1 to n but instead from 0 to $n-1$.

Output Format

The output format is much simpler when compared to the input format. On the first line, it contains the input number of clusters l, number of variables n, and ϵ such that the returned point $\mathbf{x}$ is locally ϵ-consistent. On the second line, the values of $\mathbf{x}$ are listed. On the third line, there are l numbers h_1, ..., h_l that contain the index j of the subfunction in the corresponding cluster which has status ALIVE. The values of $\mathbf{p}$ are taken from the result of the local consistency algorithm which was run on ϵ-active subfunctions. If there are multiple alive subfunctions in a cluster, $h_i = -1$. Formally, we define

$$h_i = \begin{cases} \sum\limits_{j \in [m_i]} j \cdot [\![p(f_{i,j}) = \mathsf{ALIVE}]\!] & \text{if there is only one } j \text{ with } p(f_{i,j}) = \mathsf{ALIVE} \\ -1 & \text{otherwise} \end{cases}. \tag{4.8}$$

Similarly as with the variable indices in the input, the subfunctions are numbered from 0 to $m_i - 1$ instead of 1 to m_i in the actual input.

4.3.2 The Code and its Usage

The implementation is divided into 3 classes. Class `Instance` stores the input instance of SMAF and provides the data to the minimization algorithm, this class is initialized using a path to a source file in the format shown in Figure 4.2.

The minimization algorithm is implemented in class `Solver`, which is initialized by an object of class `Instance`. The minimization process then can be started by invoking the `solve()` function, which accepts the initial ϵ as its optional parameter – if it is not specified, then it uses the initialization shown on line 3 in Algorithm 14. By

default, the program writes the information concerning every iteration on standard output but this can be turned off by setting the verbose flag to false.

After the `solve()` function terminates, the user can call the `result()` function that reports the reached function value and the minimum ϵ value for which the current point is locally ϵ-consistent. It also uses the theorems presented in Section 3.3.3 to decide the optimality of the result (if it is possible). The function `result()` also accepts a path to a text file as its optional parameter. If the path is specified, then the output file in the form shown in Figure 4.3 is created and the reached point $\mathbf{x}$ is stored there. It is also possible to change the values of vector $\mathbf{b}$ by invoking the function `changeB(i, j, d)` that sets $b_{i,j} \leftarrow b_{i,j} + d$. After this change is done, the result can be re-optimized by the `solve()` function that would start from the previously found point.

The third class is `Queue`, which is an implementation of the queue that contains only unique elements and is used in multiple places throughout the algorithm. Except for the classes, we also attach an example code `main.cpp` that shows the usage of our implementation.

Chapter 5

Experiments

In this chapter, we introduce two-dimensional grammars and show how they can be used to create large instances of SMAF that correspond to Schlesinger's upper bound of a binary max-sum problem. After we describe how the instances are created, we use our algorithm to minimize the given function and compare our result with the optimal value. We also present meaningful modifications of the presented instances, where the recalculation of optimum after a small change is used. Finally, we analyse the initial setting of ϵ and the minimization process.

5.1 Two-dimensional Grammars

To create large instances of convex piecewise-affine functions, we will use the notion of two-dimensional grammars. Notice that the notion has changed its meaning recently – it is nowadays viewed as a generalisation of the "classical" grammars, which is a set of production rules for creation of string of characters. This is described in detail in Průša (2004), however we will use the "older" meaning defined in Schlesinger (1976).

In this notion, the pixels of images are assigned labels and there are rules on which labels could be in adjacent pixels. Formally, we will consider only grey-scale images, which will be viewed as matrices of size $h \times w$ with values from the interval $[0, 1]$.

Definition 5.1. A *two-dimensional grammar* is a binary max-sum problem $(G, X, \boldsymbol{g})$ and a function $c : X \to \{0, 1\}$, where

- $G = (T, E)$ is a grid graph, i.e.

$$T = [h] \times [w], \tag{5.1a}$$
$$E = \{\{(i, j), (i', j')\} \mid i, i' \in [h], j, j' \in [w], |i - i'| + |j - j'| = 1\}, \tag{5.1b}$$

- $g_t(x) = 0$ for all $t \in T$ and $x \in X$,
- $g_{t,t'}(x, x') \in \{0, -\infty\}$ for all $\{t, t'\} \in E$ and $x, x' \in X$.

The function c in the definition above can be interpreted as a function that assigns each label either black or white colour. For the purposes of the next sections, assume that black pixels have value 1 and white ones have value 0.

Notice that for the above mentioned binary max-sum problem, it holds for all labellings $\boldsymbol{x} \in X^T$ that $F(\boldsymbol{x}|\boldsymbol{g}) \in \{0, -\infty\}$.

We say that a black and white image $\mathbf{O} \in \{0,1\}^{[h]\times[w]}$ is *generated* by a given grammar if there exists a labelling $\boldsymbol{x} \in X^T$ such that $o_{i,j} = c(x_{(i,j)})$ for all $(i,j) \in T$ and $F(\boldsymbol{x}|\boldsymbol{g}) = 0$.

Nearest Image Generated by Given Grammar

If we are given an arbitrary grey-scale image $\mathbf{O} \in [0,1]^{[h]\times[w]}$ and a two-dimensional grammar, then we can formulate the task of looking for the nearest image $\mathbf{O}^*$ of the same size such that $\mathbf{O}^*$ is generated by the given grammar and it is the "closest" one to $\mathbf{O}$ in means of minimizing

$$\sum_{i\in[h]} \sum_{j\in[w]} |o^*_{i,j} - o_{i,j}|. \tag{5.2}$$

As already shown in Werner (2007), it is possible to express this problem as a binary max-sum problem. The corresponding graph G and the binary functions $g_{t,t'}$ will be the same as in the grammar but the unary functions are given as

$$g_{(i,j)}(x) = 1 - |c(x) - o_{i,j}| \tag{5.3}$$

for all $(i,j) \in T$.

5.1.1 Examples of Two-dimensional Grammars

In this section, we introduce four grammars – *lines, rectangles, pi* and *curves.* On the *lines* grammar, we show in detail how to define the problem of nearest image generated by the grammar.

Lines Grammar

We define a grammar called *lines* for images whose matrix contains only vertical and horizontal lines that stretch from top to bottom or from the left border to the

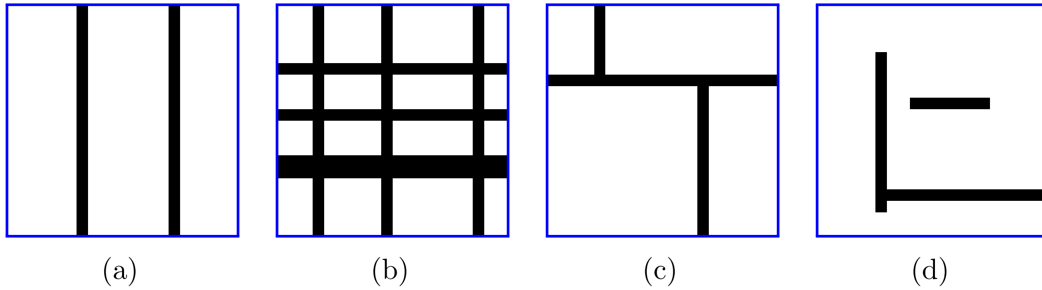


Figure 5.1: Examples of images, first two are generated by the *lines* grammar.

right border. More formally, we would like to say that an image $\mathbf{O}$ with size $h \times w$ is generated by the grammar if there exist sets $U \subseteq [h]$ and $V \subseteq [w]$ such that

$$o_{i,j} = [\![i \in U \lor j \in V]\!], \quad \forall (i,j) \in [h] \times [w]. \tag{5.4}$$

The set U (resp. V) in the previous definition can be viewed as the indices of rows (resp. columns) that contain the line. For example, the images shown in Figures 5.1a and 5.1b are generated by this grammar, whereas the images in the Figures 5.1c and 5.1d are not.

There will be four labels, $X = \{H, V, P, I\}$ and we should find a labelling $\boldsymbol{x} \in X^{[h]\times[w]}$ that assigns each pixel a label from the set X. The meaning of the labels is as follows. If a vertical and a horizontal line pass through a pixel, then it should have the label I (intersect). If only horizontal (resp. vertical) line passes through a pixel, then it should have label H (resp. V). If no line passes through it, then it is marked with P (empty). The labelled pixels of an image generated by the grammar are shown in Figure 5.2a.

The colour assigned to label P is white and the colours of the other labels will be black, i.e. $c(x) = [\![x \neq P]\!]$. The values of the unary functions are given by equation (5.3).

Now, we only need to properly define the binary functions $g_{t,t'}$ between the neighbouring pixels so that the result is generated by the *lines* grammar. That can be done by easy reasoning based on which labels could be neighbours – for example, above label V, there could be label I or another label V, but there cannot be label H or P. The allowed neighbouring relations are in Tables 5.1 and 5.2. Using these tables, we can simply forbid certain pairs of labels to neighbour in a certain direction by setting the $g_{t,t'}$ values to either 0 (if the pair is allowed) or $-\infty$ (if the pair is not allowed), formally

$$g_{t,t'}(x_t, x_{t'}) = \begin{cases} 0 & \text{if the pair } x_t, x_{t'} \text{ is allowed in the direction given by } t, t' \\ -\infty & \text{otherwise} \end{cases}.$$

Finally, we managed to transform the problem of searching the "closest" image

		Upper label $x_{(i,j)}$			
		P	H	V	I
Lower label $x_{(i+1,j)}$	P	✓	✓	X	X
	H	✓	✓	X	X
	V	X	X	✓	✓
	I	X	X	✓	✓

Table 5.1: Vertical relations between labels in *lines* grammar.

		Right label $x_{(i,j+1)}$			
		P	H	V	I
Left label $x_{(i,j)}$	P	✓	X	✓	X
	H	X	✓	X	✓
	V	✓	X	✓	X
	I	X	✓	X	✓

Table 5.2: Horizontal relations between labels in *lines* grammar.

generated by the *lines* grammar into an equivalent problem formalized as a binary max-sum problem. We can similarly define also other problems, resp. grammars.

Rectangles Grammar

In the *rectangles* grammar, we allow black shapes of rectangular form which are not allowed to overlap or touch each other. This grammar is formalized using 10 labels – empty space (P), left upper corner of a rectangle (LeUp), left lower corner (LeLo), right upper (RiUp), right lower (RiLo), upper line boundary of the rectangle (UpB), lower boundary (LoB), left boundary (LeB), right boundary (RiB) and the inside (I) of the rectangle.

An example of a rectangle with labelled pixels is in Figure 5.2b. We do not explicitly write the colouring function c or all the allowed options for vertical or horizontal neighbouring because all of the allowed options are in Figure 5.2b, where also the colours of labels can be seen. An example of an image generated by this grammar is in Figure 5.3a.

Pi Grammar

The *pi* grammar is similar as *rectangles*, except that the colours corresponding to the inner pixels of a rectangle (label I) and the lower boundary (label LoB) are white. This defines a grammar that contains images in which there are symbols that resemble the shape of letter Π. Figure 5.3b is an example of an image generated by this grammar.

P	V	P	P	P	V	V
P	V	P	P	P	V	V
P	V	P	P	P	V	V
H	I	H	H	H	I	I
H	I	H	H	H	I	I
P	V	P	P	P	V	V

(a) Example of an image generated by *lines* grammar.

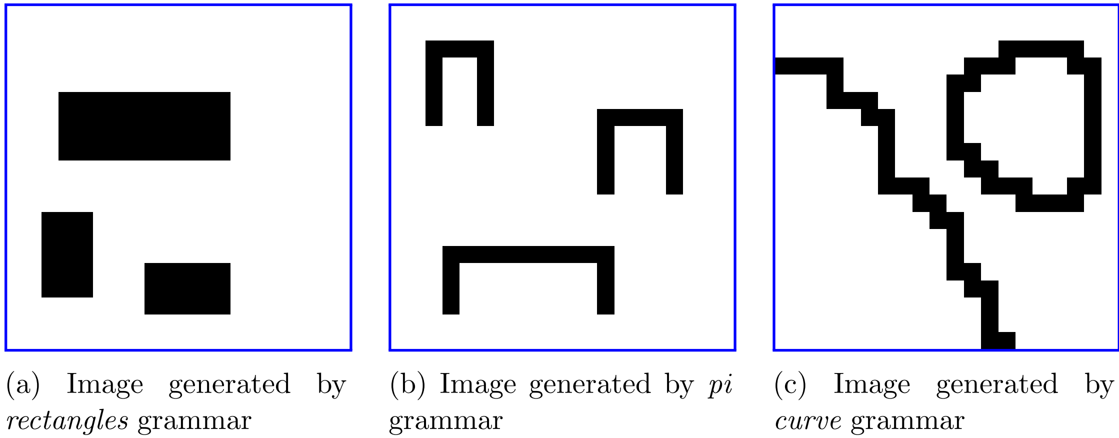
P	P	P	P	P	P	P
P	LeUp	UpB	UpB	UpB	RiUp	P
P	LeB	I	I	I	RiB	P
P	LeB	I	I	I	RiB	P
P	LeLo	LoB	LoB	LoB	RiLo	P
P	P	P	P	P	P	P

(b) Example of an image generated by *rectangles* grammar.

P	P	P	P	P	P	P	P	P	P
H	H	B	P	P	A	H	H	B	P
P	P	V	P	P	V	P	P	V	P
P	P	V	P	P	C	H	H	D	P
P	P	C	B	P	P	P	P	P	P
P	P	P	C	B	P	P	P	A	H
P	P	P	P	V	P	P	A	D	P

(c) Example of an image generated by *curve* grammar.

Figure 5.2: Examples of labelled images generated by various grammars.



(a) Image generated by *rectangles* grammar

(b) Image generated by *pi* grammar

(c) Image generated by *curve* grammar

Figure 5.3: Examples of images generated by various grammars.

Curve Grammar

The last grammar that will be introduced is the *curve* grammar, consisting of non-intersecting curves, where a curve is a sequence of pixels so that the next one is in the 4-neighbourhood of the first one. A line can also begin or end at the boundary of the image and an example of an image generated by this grammar is in Figure 5.3c. Additionally, in the 4-neighbourhood of a pixel that is black, there can be at most 2 other black pixels – one that is ”previous” in the curve sequence and one that is its ”successor” in the line sequence. If the line begins at the boundary, there is only one neighbouring black pixel.

To define this grammar, we used 6 labels – horizontal part of line (label H), vertical part of line (label V), bend from lower vertical to right horizontal (label A), bend from lower vertical to left horizontal (label B), bend from upper vertical to right horizontal (label C), bend from upper vertical to left horizontal (label D). All the allowed options are shown in Figure 5.2c. Notice that label combination AD is allowed in both horizontally and vertically neighbouring pixels to allow zig-zag shape. Similarly, BC (vertically) and CB (horizontally) combinations are allowed for the other direction of zig-zag.

5.2 Tested Instances

To obtain large instances of SMAF, we first generated baseline images that were generated by one of the previously defined grammars, then independent Gaussian noise was added to the pixels in the images to obtain the input image $\mathbf{O}$. After that, we transformed the problem of searching the ”closest” image generated by the grammar to the binary max-sum problem, basically in the same way as stated in the beginning of Section 5.1. Finally, we created the SMAF that corresponds to the height minimization of the binary max-sum problem defined by the image, which is

Instance	LP parameters		SMAF parameters		
	variables	constraints	clusters	subfunctions	dimension
lines200	756400	796800	119600	796800	636800
lines100	188200	198400	29800	198400	158400
lines50diamond	46600	49200	7400	49200	39200
rect100round	425800	416800	29800	416800	396000
rect100sharp	425800	416800	29800	416800	396000
pi50prec	105400	103400	7400	103400	98000
pi50rough	105400	103400	7400	103400	98000
curve500	7735000	8736000	749000	8736000	6986000

Table 5.3: Sizes of the individual tested instances.

the process defined in Algorithm 3. To also create "harder" inputs, we sometimes used baseline images that are not generated by the given grammar.

The Gaussian noise had zero mean and we tried three settings concerning the standard deviation of the noise – we used low noise with $\sigma = 0.12$, medium noise with $\sigma = 0.4$ and high noise with $\sigma = 1.2$. If any value of a pixel got out of the $[0, 1]$ interval, its value was trimmed to the nearer border.

We used 8 different baseline images, to which different amounts of noise were added, which resulted in 24 instances of SMAF in total. Table 5.3 contains the parameters of the instances which define their size, i.e. both the parameters of the corresponding upper bound minimizing LP in the form of equation (1.14) and the parameters corresponding to the SMAF that is minimized. Note that these parameters do not depend on the amount of noise in the image but only on the used grammar (i.e. the amount of labels and the rules on neighbouring labels) and the image size.

The first part of the name of an instance consists of the name of the grammar by which the resulting image should be generated (i.e. `lines`, `rect` for *rectangles*, `pi` and `curve`). This is followed by the size of the corresponding image – the images are square, so it is enough to mention only one number. The name of the instance may also contain an additional keyword for better identification. Finally, the actual input instances are extended by the keywords `_low`, `_med` and `_high` that define how much noise was added.

The algorithm is run on these instances and after it finishes the minimization and finds a locally ϵ-consistent point, we look on the returned values h_i, as defined in Section 4.3.1. If each cluster corresponding to a pixel has a unique maximum, we can reconstruct the solution by setting the corresponding maximum labels. On the other hand, if there are more maxima in a cluster, then we mark the object as undecided, which will be denoted by red pixels in the following section.

To find the optimal solutions to these instances, we calculated a continuous LP in form (1.14) using Gurobi Optimizer version 7.5.2. We also compared the runtime

of our implementation with the Gurobi Optimizer[1]. As described in Gurobi Optimization (2016), the LP solver by default uses a concurrent solver which runs various optimization algorithms on more processor cores at the same time in parallel and returns the solution when the fastest algorithm terminates. Therefore, the runtime of Gurobi could be higher if it comitted to a single algorithm and used only one processor core at time.

It is necessary to mention that in order to be able to find integer solutions, we multiplied the values in the resulting $\mathbf{b}$ (resp. $\boldsymbol{g}$) vector by a large integer.

5.3 Results

5.3.1 Lines Grammar

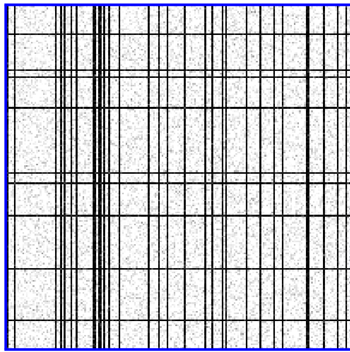
In the Figure 5.4, there are the results of the experiments with the *lines* grammar. In Figure 5.4j, there is the baseline image to which the noise was added, which created the input images in Figures 5.4a, 5.4d and 5.4g. After the minimization algorithm was applied, it found a solution, where each cluster contained exactly one subfunction that was alive (after the local consistency algorithm was run), which allowed to reconstruct the solution in Figure 5.4j. This solution is exactly the baseline image to which the noise was added and the result was the same in all the input images, the only difference is that with higher noise, more iterations were needed.

The same experiment was performed also with different image sizes, the input images in Figures 5.4c, 5.4f and 5.4i have the size 100 $\times$ 100 and used baseline image shown in Figure 5.4l. As in the previous case, the algorithm always found the solution that corresponds exactly to the baseline image.

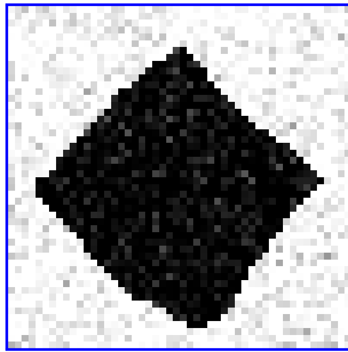
In the previous cases, the image to which the noise was added was always in the grammar that we were searching, so we also tried to use a baseline image that was also used in Werner (2007) which contains an inaccurately drawn square. This input image is in the original source called "diamond". To this image, we added all levels of noise, producing Figures 5.4b, 5.4e and 5.4h and searched for the nearest image generated by the *lines* grammar. In the case of low and medium noise, the algorithm surprisingly found solutions in which all the maxima were unique. The result to the input with medium noise is in Figure 5.4k. The result corresponding to the low noise was not the same, but very similar – the "cross" was a little bit thinner. In the case of high noise, no maximum of any cluster was unique, which prevented us from any reconstruction.

Table 5.4 shows the comparison between the optimal (minimal) values of the upper bound of various SMAF instances and the minimized values returned by the

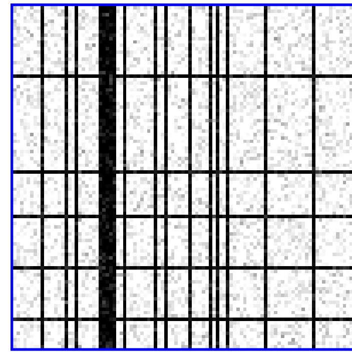
[1]The experiments were done on a computer with Intel Core i7 7500U CPU and 8 GB RAM.



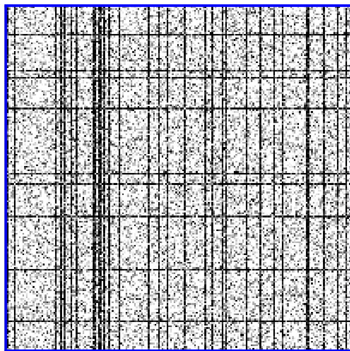
(a) Input image with low noise, size 200 × 200



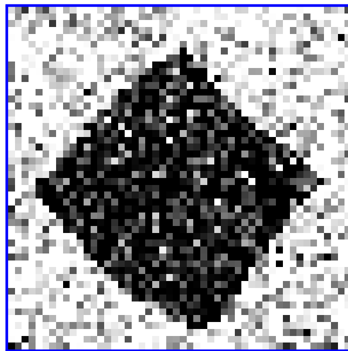
(b) Input with baseline diamond image, low noise



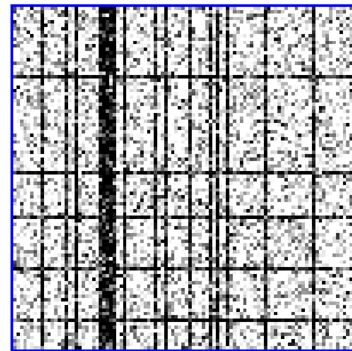
(c) Input image with low noise, size 100 × 100



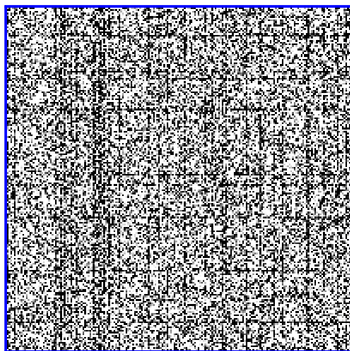
(d) Input image with med. noise, size 200 × 200



(e) Input with baseline diamond image, med. noise

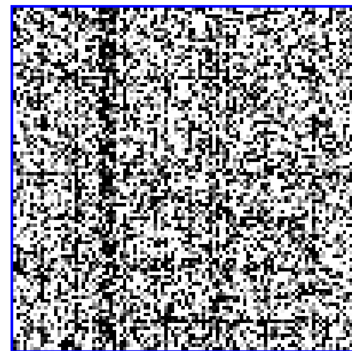


(f) Input image with med. noise, size 100 × 100

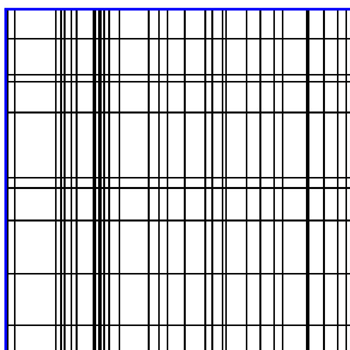


(g) Input image with high noise, size 200 × 200

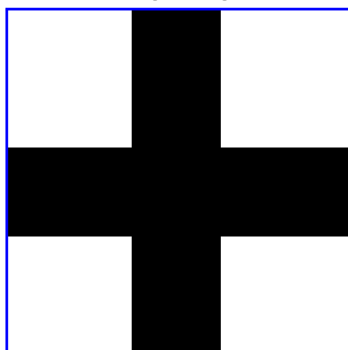
(h) Input with baseline diamond image, high noise



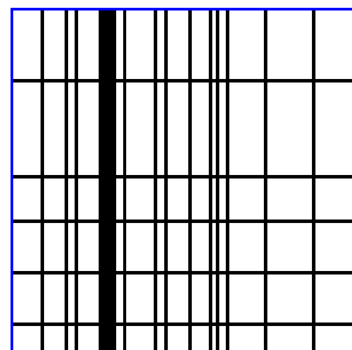
(i) Input image with high noise, size 100 × 100



(j) Both result and baseline image, size 200 × 200



(k) Result corresponding to the diamond input 5.4e



(l) Both result and baseline image, size 100 × 100

Figure 5.4: Input images and solutions for the *lines* grammar.

algorithm. The table also contains values that denote in which cases the height was truly optimal – we express the difference between the optimal value $f(\mathbf{x}^*)$ and the found function value $f(\mathbf{x})$ as a value

$$R = \frac{f(\mathbf{x}) - f(\mathbf{x}^*)}{f(\mathbf{x}^*)}, \tag{5.5}$$

which calculates the difference as a multiple of the optimal value. The table also contains the returned value of ϵ and the corresponding runtimes.

Our algorithm was significantly faster on the tested *lines* instances compared to the LP solver. On instances that are "obvious" to human perception, it produced the solution immediately. The other instances took more time.

5.3.2 Rectangles Grammar

Similar experiments to the previous ones were also performed with the *rectangles* grammar – we created a baseline image that is generated by the grammar (`rect100sharp`) and another one that is not (`rect100round`) and then added noise with various variance to create the individual instances.

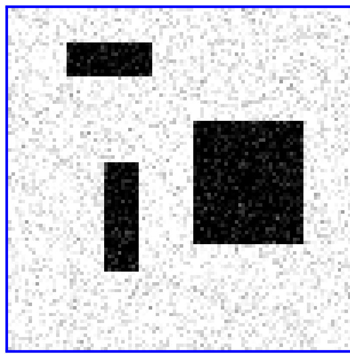
The generated images and the corresponding results are in Figure 5.5. The red pixels in the solutions mean that the cluster corresponding to the red pixel had multiple maxima alive, therefore it was not possible to determine the label.

The qualitative results are again in Table 5.4 – this time, there were fewer optima reached and the non-optimal values were also more distant from the optimum than in the case of *lines* grammar. Similarly as last time, the runtimes of our algorithm are significantly smaller than the runtimes of the LP solver.

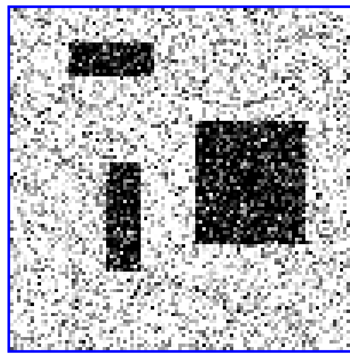
5.3.3 Pi Grammar

In case of the *pi* grammar, an image which is not generated by the grammar (called `pi50rough`) and also an image that originally is generated by the grammar (called `pi50prec`) were used. The input and output images are shown in Figure 5.6.

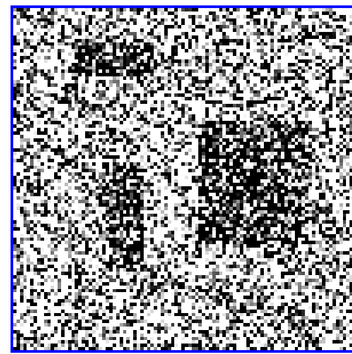
In the cases of low and medium noise, there are few undecided points, which are interestingly located only in the upper part of the image. The reason for this can be easily discovered after finding out which labels correspond to the clusters' maxima. In case of Figures 5.6d and 5.6j, the lowest undecided line has two maxima – it could be an empty space P or the lower boundary LoB of a pi sign. The upper lines have three maxima – it could be empty space P (if the pixel below was empty), lower boundary LoB (if the pixel below was empty) or interior point I (if the pixel below was a lower boundary or also an interior point). So, it can be basically viewed as that it is known that the red points in these images should be white, but it is



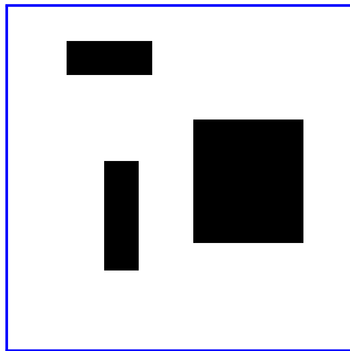
(a) Input rect100prec_low



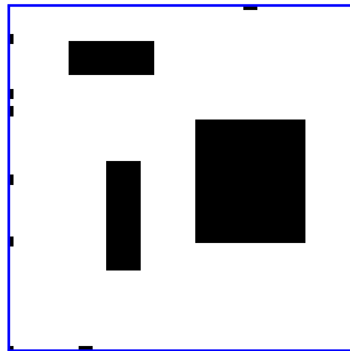
(b) Input rect100prec_med



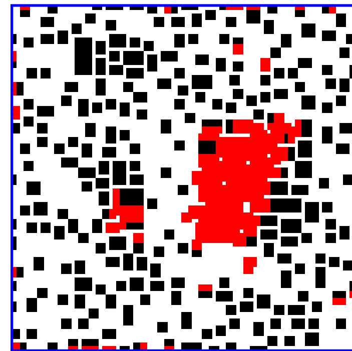
(c) Input rect100prec_high



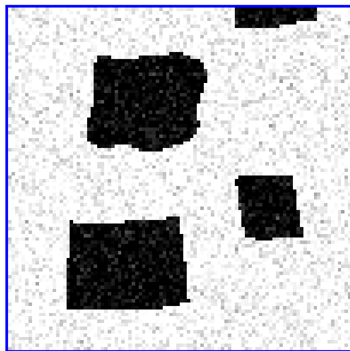
(d) Result of rect100sharp_low



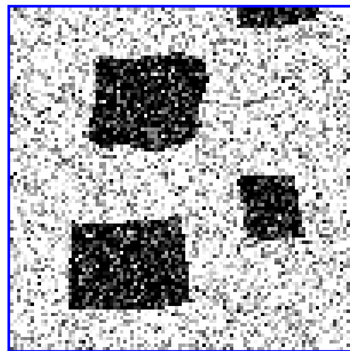
(e) Result of rect100sharp_med



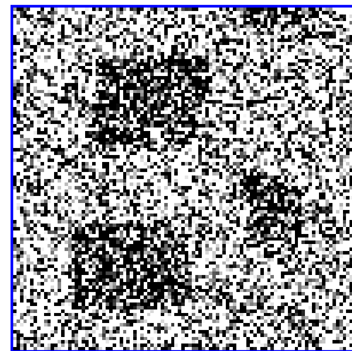
(f) Result of rect100sharp_high



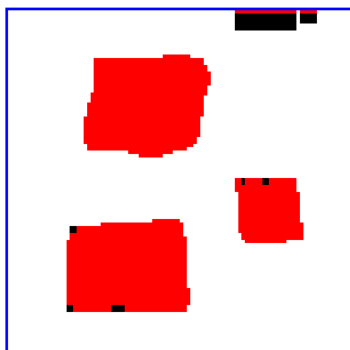
(g) Input rect100round_low



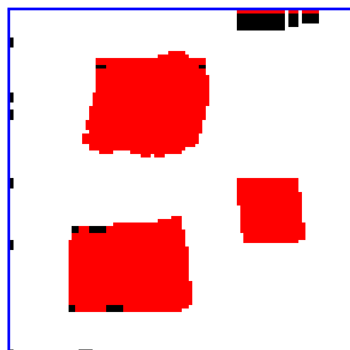
(h) Input rect100round_med



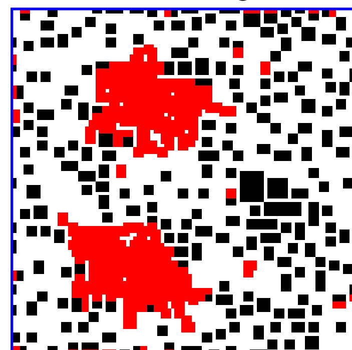
(i) Input rect100round_high



(j) Result of rect100round_low



(k) Result of rect100round_med



(l) Result of rect100round_high

Figure 5.5: Input images and solutions for the *rectangles* grammar.

not known if they should be empty or if there could be a large pi sign whose black pixels are out of the picture and therefore the upper part of the image could be inside such sign. This does not happen in the lower part of the image because the upper boundary of the pi sign would be visible there.

On the other hand, in the case of the images with medium noise, i.e. the results that are in Figures 5.6e and 5.6k, few pixels in the uppermost row are so dark due to the noise that they should be black and maxima of their clusters correspond to the LeLo and RiLo labels, i.e. the end of either the left or the right vertical line of the pi symbol. The pixels between them are either empty (label P) or the lower boundary (LoB) of a pi symbol – this would be based on the choice of the left/right direction of the other labels. So, we found multiple labelings with the highest quality.

In the results corresponding to the input with high noise, the undecided pixels are not so structured, likely because the input does not resemble an image from the pi grammar at all.

The numerical results on these instances are listed in Table 5.4, where it can be seen that the runtime of our algorithm was again significantly lower than the runtime of the LP solver and in 4 of the 6 cases, we reached the optimum.

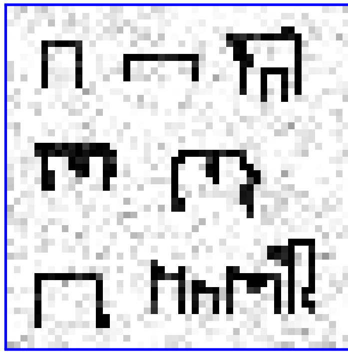
5.3.4 Curve Grammar

In the *curve* grammar, we generated the largest instance, which was made up from an image with size 500×500 that is originally generated by the grammar. Again, we added various amounts of noise to the image and tried to minimize the height of the corresponding binary max-sum problem. The input images and the results are shown in Figure 5.7.

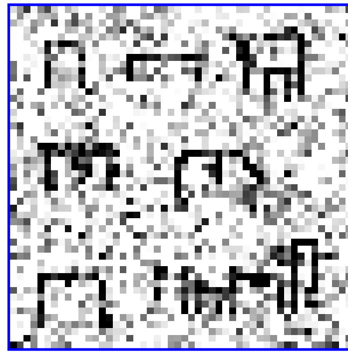
Similarly as in the previous cases, the comparison of runtimes along with the optimality of results is shown in Table 5.4. This time, Gurobi could not run the concurrent solver because it required too much memory. Neither dual simplex method nor the barrier method could be used even individually because of the same reason. That is why the runtimes in this case are calculated by the primal simplex method. In case of `curve500_high`, the calculations ran more than 2 weeks before out of memory error occurred. That is why the optimum of this instance was not calculated.

On the other hand, our algorithm terminated with a much slower runtime on these instances and reached the optimal value in the case with medium and low noise. By Theorem 3.6, we know that on the instance with high noise, suboptimal value was reached even without knowing the optimal value of the problem.

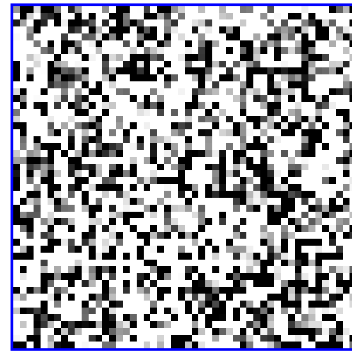
The runtimes in Table 5.4 are measured only for the iterative minimization procedure, we did not measure the runtime of loading the input file or saving the result. Similarly for the LP solver, we did not measure the time required for the set-up of the LP, but only the minimization process itself.



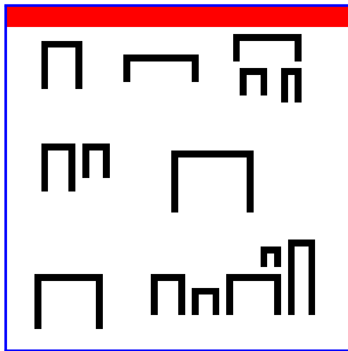
(a) Input `pi50rough` with low noise



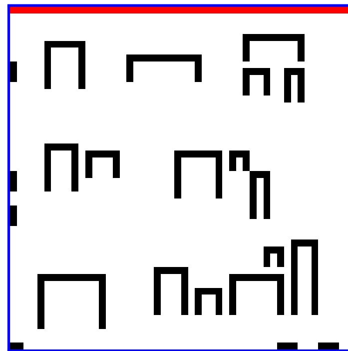
(b) Input `pi50rough` with medium noise



(c) Input `pi50rough` with high noise

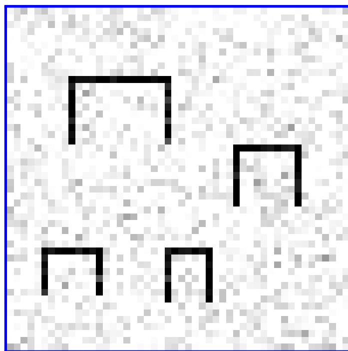


(d) Result of `pi50rough` with low noise

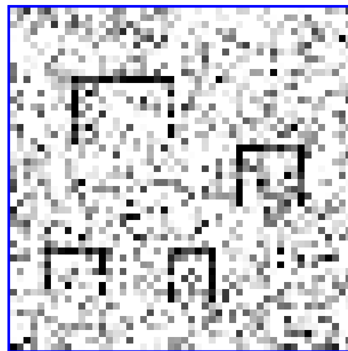


(e) Result of `pi50rough` with medium noise

(f) Result of `pi50rough` with high noise

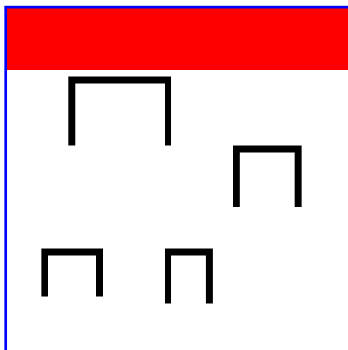


(g) Input `pi50prec` with low noise

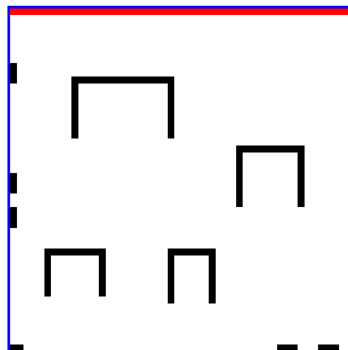


(h) Input `pi50prec` with medium noise

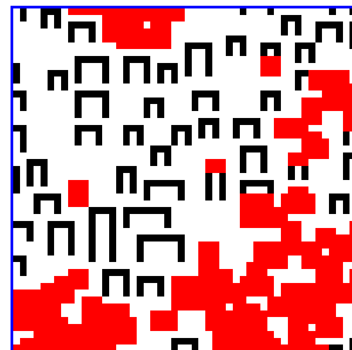
(i) Input `pi50prec` with high noise



(j) Result of `pi50prec` with low noise

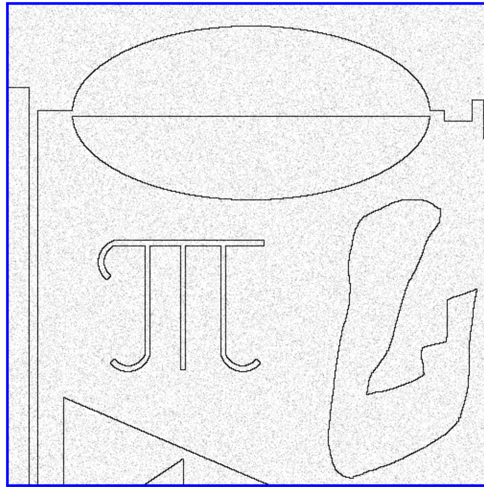


(k) Result of `pi50prec` with medium noise

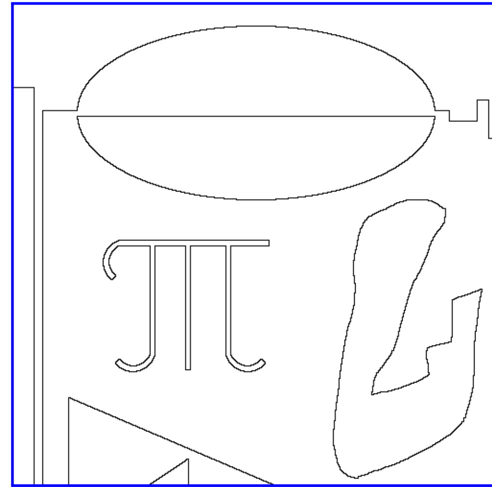


(l) Result of `pi50prec` with high noise

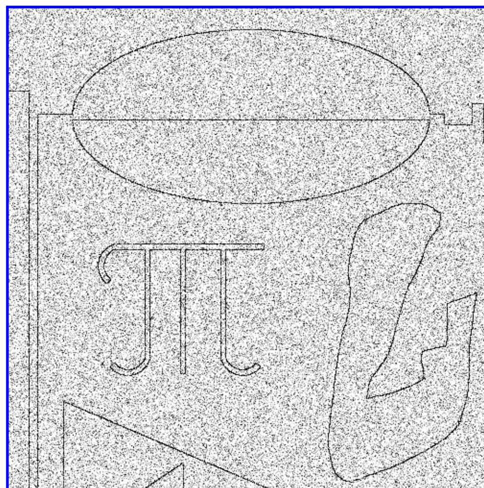
Figure 5.6: Input images and solutions for the *pi* grammar.



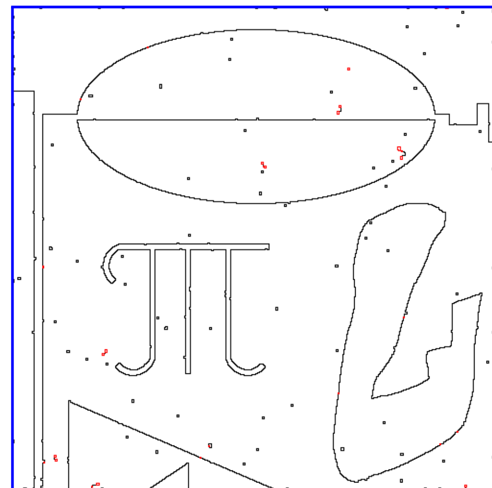
(a) Input `curve500` with low noise



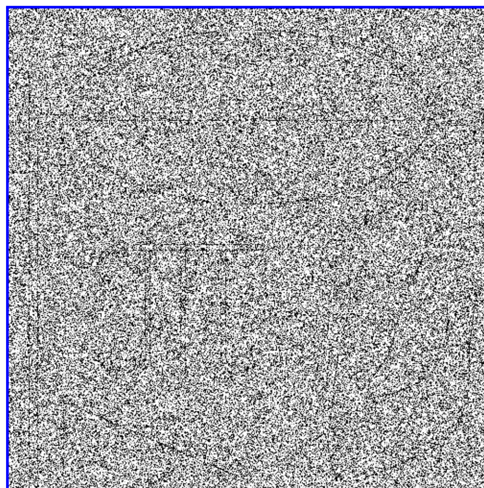
(b) Result of `curve500` with low noise



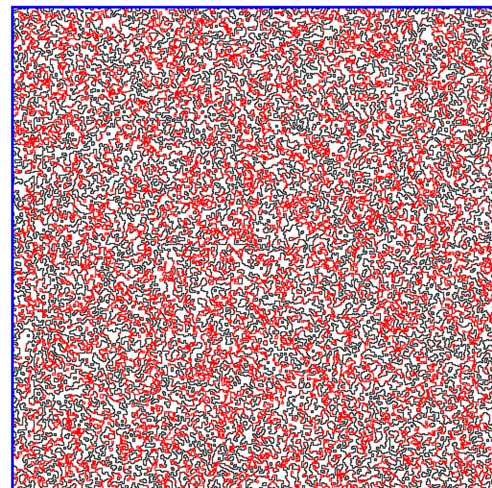
(c) Input `curve500` with medium noise



(d) Result of `curve500` with medium noise



(e) Input `curve500` with high noise



(f) Result of `curve500` with high noise

Figure 5.7: Input images and solutions for the *curve* grammar.

Instance of SMAF	Optimum	Minimized to	LP runtime	Our runtime	ϵ	R
lines200_high	14194617352192	14194617352192	14815.0 s	220.4 s	0	0
lines200_med	18093229211648	18093229211648	1983.0 s	1.1 s	0	0
lines200_low	20396852117504	20396852117504	1539.5 s	< 1 s	0	0
lines100_high	3553387085824	3553387085824	925.7 s	24.1 s	0	0
lines100_med	4519050084352	4519050084352	826.4 s	< 1 s	0	0
lines100_low	5098038099968	5098038099968	849.8 s	< 1 s	0	0
lines50diamond_high	918061842432	918061842662	44.6 s	< 1 s	3	$2.5 \cdot 10^{-10}$
lines50diamond_med	939794628608	939794628608	23.6 s	1.9 s	0	0
lines50diamond_low	1025186463744	1025186463744	22.5 s	1.5 s	0	0
rect100sharp_high	3862483641786.583	3862948035702	485.6 s	5.4 s	82	$1.2 \cdot 10^{-4}$
rect100sharp_med	4522357293056	4522357293056	6585.6 s	< 1 s	0	0
rect100sharp_low	5098293952512	5098293952512	4338.2 s	< 1 s	0	0
rect100round_high	3858045642353.665	3858719475728	5427.7 s	17.3 s	2453	$1.7 \cdot 10^{-4}$
rect100round_med	4469794039592.592	4470672873012	7427.2 s	61.4 s	9191	$2.0 \cdot 10^{-4}$
rect100round_low	5019918069332.757	5020900572035	5314.2 s	72.5 s	40231	$2.0 \cdot 10^{-4}$
pi50rough_high	1001347577719.467	1001419353514	170.6 s	< 1 s	3	$7.2 \cdot 10^{-5}$
pi50rough_med	1113698861056	1113698861056	168.7 s	< 1 s	0	0
pi50rough_low	1233886642176	1233886642176	188.4 s	< 1 s	0	0
pi50prec_high	1006988427264	1007074368367	105.6 s	< 1 s	3	$8.5 \cdot 10^{-5}$
pi50prec_med	1139051331584	1139051331584	183.5 s	< 1 s	0	0
pi50prec_low	1276033105920	1276033105920	180.5 s	< 1 s	0	0
curve500_high	n/a	104703708253010	$> 1.5 \cdot 10^{6}$ s	220.9 s	32	> 0
curve500_med	112950824140800	112950824140800	32524.1 s	26.3 s	0	0
curve500_low	127430662553600	127430662553600	39570.3 s	4.1 s	0	0

Table 5.4: Overview of all the numerical results and runtimes on all instances.

Except for a few instances, we are able to determine the optimality of the solutions just by applying Theorems 3.6 and 3.7 and we would not even have to know the optimal values from the LP. In the instances `pi50rough` and `pi50prec` with medium and low noise, no previously shown theorem can decide the optimality of the result, but it could be deduced from the reasoning in Section 5.3.3 because we know that these instances of SMAF correspond to the upper bound of a binary max-sum problem. Therefore, the only case that would remain unresolved if we did not know the true optimum would be the `curve500_med` instance.

5.3.5 Small Changes

We have also experimented with changing the values in the $\mathbf{b}$ vector to re-optimize the problem for a slightly different setting.

Pi Grammar Instances

In this case, the possibility to do small changes of the problem become useful – we can help the solver to decide which solution to take. For example, in the case of `pi50prec_low` or `pi50rough_low`, we can change e.g. the value $b_{i,j}$ corresponding to the empty label of the pixel in the upper left corner and increase it by 1. After that, the algorithm does not need to change the current value of $\mathbf{x}$ in this case because it still is locally consistent but the ambiguous labels disappear because in the upper left corner, there is only one maximum – the empty label. Therefore the I or LoB labels become inconsistent and are killed, which means that the previously red areas in Figures 5.6d and 5.6j now become white because the only remaining subfunctions that are alive correspond to empty label. The result corresponding to the modified instance `pi50prec_low` is in Figure 5.8a.

The same thing can be done also for the instances with medium noise, which would result in an image exactly like Figures 5.6e, resp. 5.6k, but in the uppermost row, there would be few separated black pixels. The result corresponding to the modified instance `pi50prec_med` is in Figure 5.8b.

Rectangles Grammar Instances

In the results of the *rectangles* grammar instances with high noise, there was a lot of small rectangles, i.e. in Figures 5.5f and 5.5l. In order to try to improve the result, we tried to introduce a regularizer on the amount of rectangles.

The regularizer is the decrease of the $b_{i,j}$ values corresponding to the corner labels of the rectangles by a value ω. In this way, the optimization criterion (5.2)

becomes

$$4\,\omega \cdot \text{rectangles}(\mathbf{O}^*) + \sum_{i\in[h]} \sum_{j\in[w]} |o^*_{i,j} - o_{i,j}|, \tag{5.6}$$

where rectangles($\mathbf{O}^*$) is the amount of rectangles in the image $\mathbf{O}^*$.

We tried to set $\omega = 1$ or $\omega = 10$ and decrease the corresponding $b_{i,j}$ values by this value[2].

In the case of `rect100prec_high`, the smaller ω resulted in Figure 5.8c, which contains fewer rectangles than the original result without regularization. The larger value of ω returned the result in Figure 5.8d, which almost exactly corresponds to the baseline image, except that the upper left rectangle is narrower by 1 pixel from each side and the left long rectangle is wider by 2 pixels on the left side.

The same worked also for the instance `rect100round_high`, whose corresponding result for $\omega = 1$ is in Figure 5.8e, which again contains fewer rectangles. Figure 5.8f shows the result for $\omega = 10$ – notice that the result contains four rectangles in the places that can be recognized e.g. in Figure 5.5g.

The recalculation of the problem from the previously locally consistent point is beneficial in the means of runtime. In all these cases, the runtime of the algorithm after the small change was performed was smaller than if the algorithm was completely restarted. This could be also beneficial in cases where the regularization constant is searched and results for multiple regularizations are compared.

5.4 Further Analysis

Here, we first analyse the dependence of the number of iterations and the final function value on the initial value of ϵ. Then, we look into one iteration and its progress during a single run.

5.4.1 Dependence on Initial Value of ϵ

As we said above, we introduce ϵ-consistency in order to make the calculations faster. Now, we will show how it is sped up based on the initial value of ϵ. We analysed the speed-up by the number of iterations required for the algorithm to terminate.

We observed that the number of iterations does not always increase when initial ϵ is lowered and that the instances can be roughly divided into two groups. In the first one, the number of iterations is initially almost constant with increasing value of initial ϵ but after a region of values of the initializer is crossed, the number of iterations decreases rapidly. Examples of such instances are for example `rect100sharp_med` or

[2]Recall that the values $b_{i,j}$ were multiplied by a large integer, these ω values are therefore also actually multiplied by the same value.

lines200_med, for whose the required amount of iterations for various initial values of ϵ is shown in Figures 5.9a and 5.9b. For these instances, the reached function value was the same for all initialization values or at least very similar. This case usually happened in the "easier" instances.

In the other group, which basically contains the "harder" instances, the amount of iterations first increases with increasing initial ϵ but then starts to decrease. Examples of such progress are shown in Figures 5.9c and 5.9d that corresponds to instances rect100round_high and lines100_high. It might seem that having small initial value of ϵ could be also beneficial in this cases but the contrary holds because the low amount of iterations for small initial values is caused by worse results of the algorithm. It means that even though the algorithm terminates faster, it ends with a larger function value. Figures 5.9e and 5.9f show the function value with that the algorithm terminates on these instances based on the initial value of ϵ. The function value for small initializers is higher than for the higher ones.

In all the figures, the maximum value of initial ϵ is

$$\bar{\epsilon} = \max_{i,j} b_{i,j} - \min_{i,j} b_{i,j}, \tag{5.7}$$

which is the initialization proposed in Algorithm 14, where it was also justified why larger values would not change the outcome. From the previous experimental observations, it follows that this initializer lowers the required amount of iteration and also provides lower function values when compared to the other ones.

5.4.2 Typical Progress of Objective Value

Here, we look into a single run of the algorithm – we will show the results for instances lines200_high and rect100sharp_med, for the other instances, the results would be similar[3].

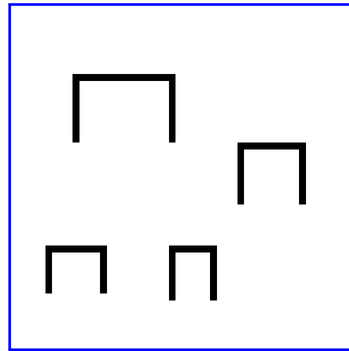
First of all, we show how the current function value decreases during the minimization process. This is shown in Figures 5.10a and 5.11a. These figures contain multiple plots, each for a different initial ϵ value. Since we already know from the previous section that the convergence is basically the fastest with $\epsilon = \bar{\epsilon}$, the result of comparison of the initial ϵ values is not surprising. The shapes of the curves in these figures could be denoted as "typical" results of optimization methods in the sense that it initially decreases steeply but as the value approaches the optimum, the decrease slows down.

The decrease of the function in a single iteration is connected to the current value of ϵ. In Figures 5.10b and 5.11b, we can see by what value the function decreased

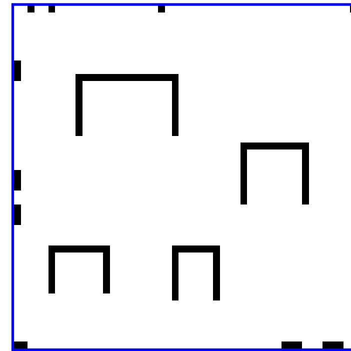
[3]Except for those that are locally consistent already at $\mathbf{x} = 0$. These finish immediately without any iterations and are therefore uninteresting.

in each iteration and what was the current value of ϵ at that time. This is the run, where ϵ was initialized as $\bar{\epsilon}$. From both figures, we can see that larger values of ϵ correspond to a larger decrease in the objective function value. These figures also explain why the convergence with significantly lower ϵ is slower – because the initial value is small, it is not possible to make the large steps in the beginning as with the larger initializations.

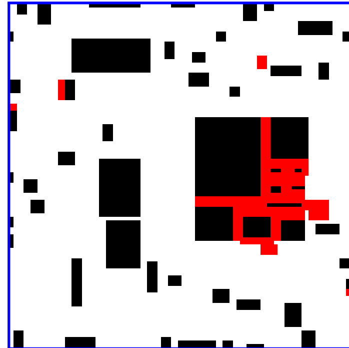
Note that in the case of Figure 5.10b, the final value was reached already with $\epsilon \approx 2 \cdot 10^6$ and the value of ϵ was then always lowered without changing $\mathbf{x}$. This is why the plot is trimmed for clarity and we do not see how ϵ is lowered to zero.



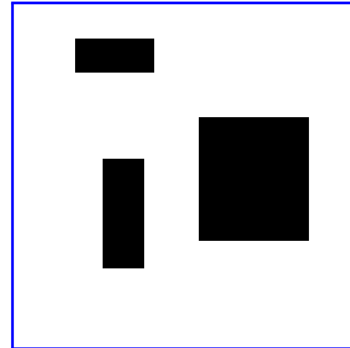
(a) Result of modified `pi50prec_low` instance, the red lines disappear



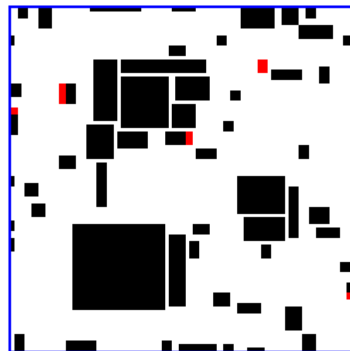
(b) Result of modified `pi50prec_med` instance, the red line disappeared



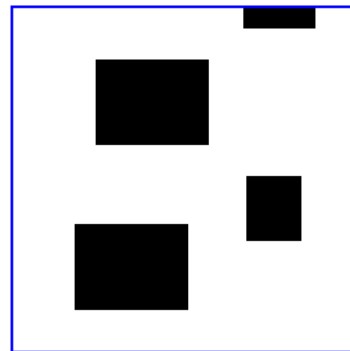
(c) Result of modified `rect100prec_high` instance with $\omega = 1$



(d) Result of modified `rect100prec_high` instance with $\omega = 10$

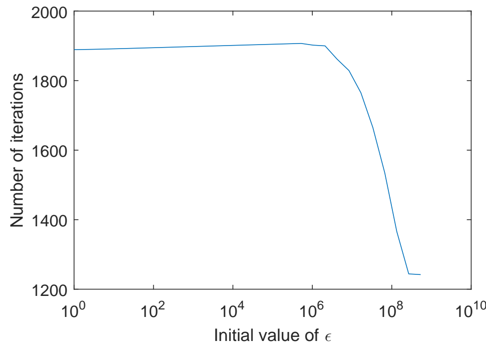


(e) Result of modified `rect100round_high` instance with $\omega = 1$

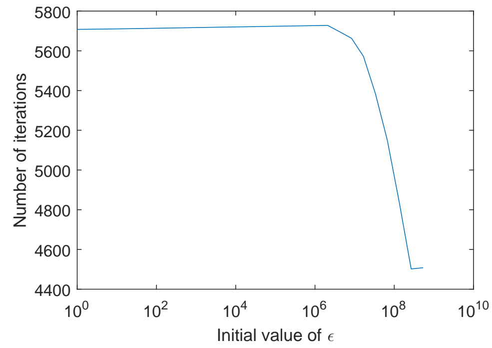


(f) Result of modified `rect100round_high` instance with $\omega = 10$

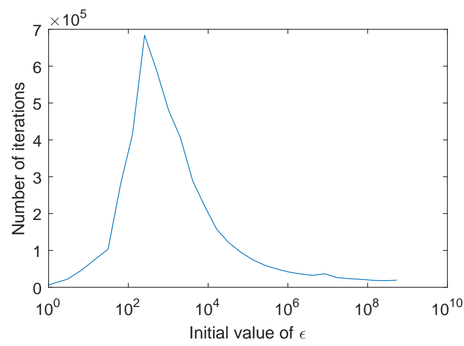
Figure 5.8: Results corresponding to modified instances.



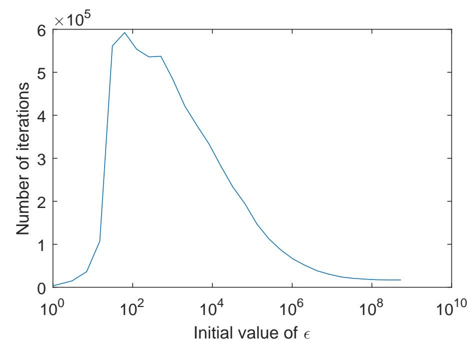
(a) Number of iterations for instance `rect100sharp_med`



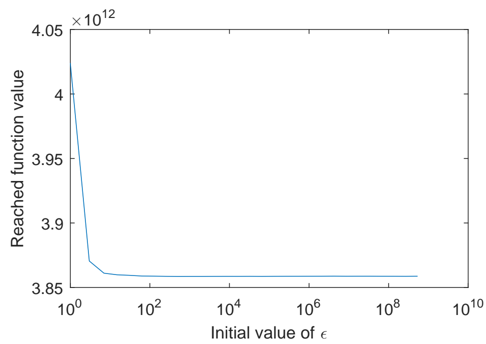
(b) Number of iterations for instance `lines200_med`



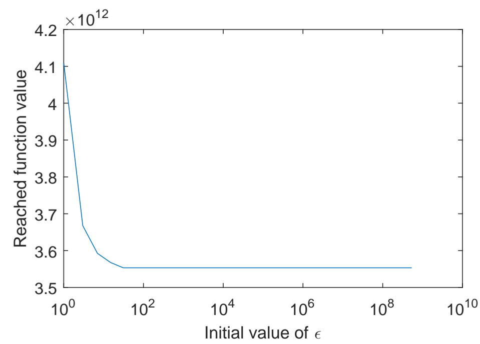
(c) Number of iterations for instance `rect100round_high`



(d) Number of iterations for instance `lines100_high`

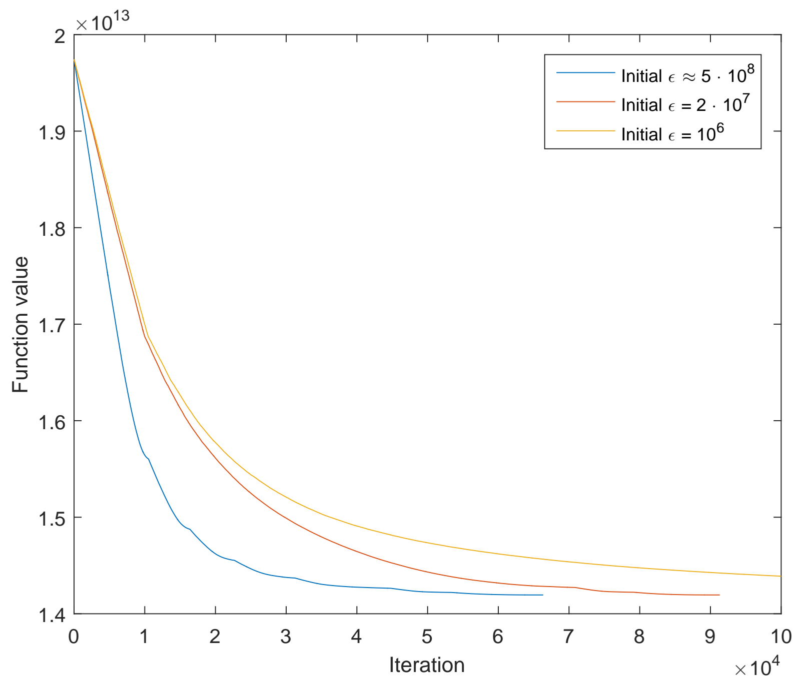


(e) Reached function value for instance `rect100round_high`

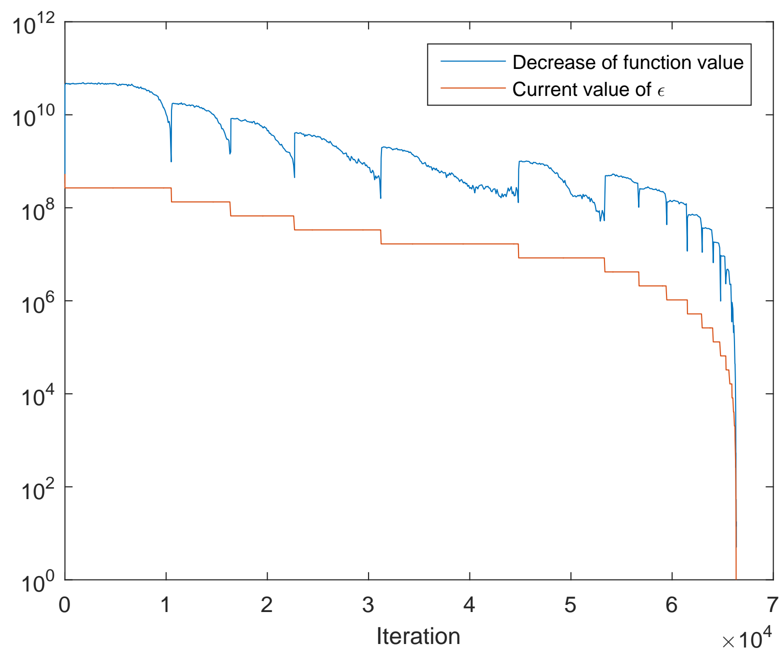


(f) Reached function value for instance `lines100_high`

Figure 5.9: Comparison of results for various initialization values of ϵ.

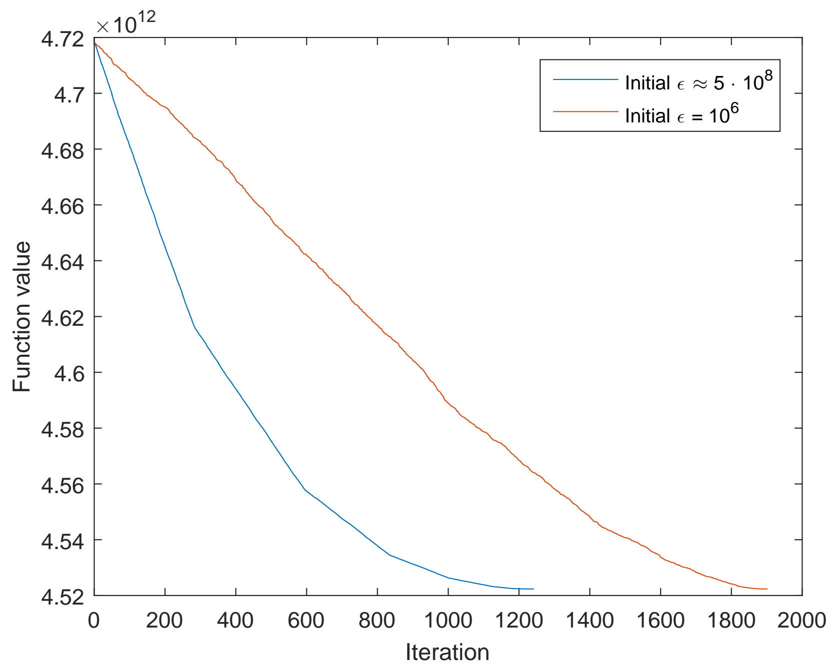


(a) Plot showing the current value of the function with respect to the current iteration and the initial value of ϵ. Notice that the horizontal axis is trimmed and the run corresponding to $\epsilon = 10^6$ terminated after approximately $2.8 \cdot 10^5$ iterations.

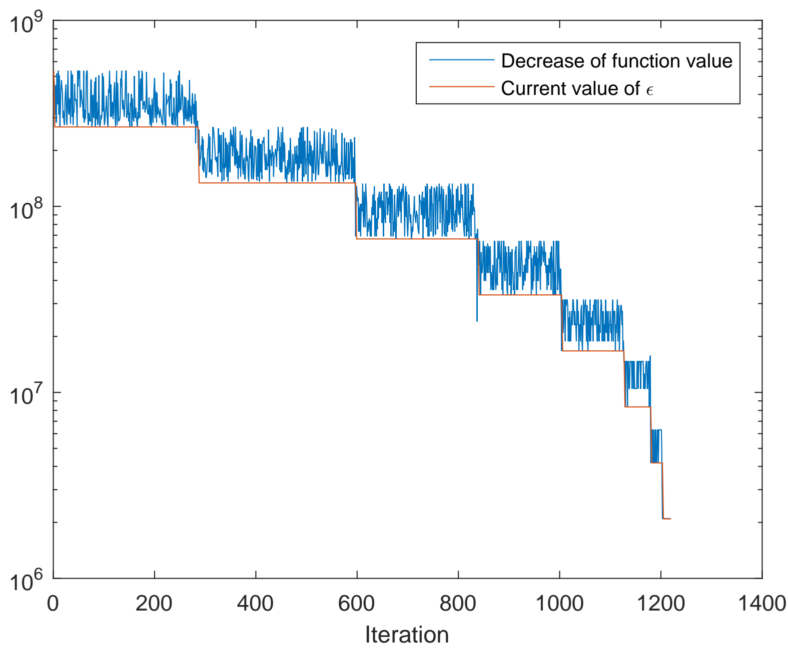


(b) Plot showing the decrease of the function value and the current value of ϵ during a single run of the algorithm, initialized with $\bar{\epsilon}$.

Figure 5.10: Progress of minimization, instance `lines200_high`.



(a) Plot showing the current value of the function with respect to the current iteration and the initial value of ϵ.



(b) Plot showing the decrease of the function value and the current value of ϵ during a single run of the algorithm, initialized with $\bar{\epsilon}$.

Figure 5.11: Progress of minimization, instance `rect100sharp_med`.

Chapter 6

Conclusion and Future Research

In this final chapter, we summarize the contribution of our thesis and also list potential areas for future research that we identified.

6.1 Conclusion

In this thesis, we have overviewed the binary max-sum problem and the related theory, namely Schlesinger's upper bound and its connection to arc consistency. Then, we presented a novel algorithm for convex piecewise-affine function minimization, which is based on the notion of local consistency and works with functions that are given as SMAF or MAF. This algorithm is a generalization of the Augmenting DAG algorithm. We proved the correctness of the algorithm along with the calculation of its asymptotic space complexity and the asymptotic time complexity of one iteration. We also presented its finite-precision version that we implemented in C++ and formulated theorems that in some cases allow to determine the optimality of the given solution.

We tested the algorithm on instances that correspond to Schlesinger's upper bound of a binary max-sum problem, where the max-sum problem was based on two-dimensional grammars and searching the closest image generated by a given grammar. The relaxation that is used by the algorithm showed to be suitable for such problems because it often reached the true optimum. Moreover, our algorithm had significantly shorter runtime on all the instances when compared to an optimal LP solver. On the generated instances, we also tried re-optimizing the solution after a small change in the input instance was done, which could become useful in ill-conditioned or ambiguous problems.

Finally, we also discussed the convergence speed and justified a general parameter initialization that experimentally showed to work suitably for all of the tested instances. It is therefore not necessary for the user of our algorithm to set any parameters or tune them to optimize the result.

6.2 Future Research

During the exploration of the field of local consistency and convex piecewise-affine minimization, we have encountered a number of open questions that might be resolved in the future.

Decrease of Space Complexity

First of all, it might be possible to decrease the space complexity of the algorithm by removal of some additional structures that we currently use but maybe only in the sense of actual space complexity, not asymptotic space complexity. For example, get rid of either the $N(x_k)$ or $N(f_{i,j})$ sets, which take up significant amount of memory. Then, the algorithm might be even more useful for larger instances.

Oracles

Talking about space complexity, in case of the shown instances, it is not necessary to store the whole upper bound function as a SMAF explicitly, but only implicitly using the structure of the problem and the data (i.e. the input image). For example, we could have an oracle that would create a numbering of clusters and a numbering of the subfunctions in them that would not be explicitly stored but only calculated from the size of the input image. Next, to find out the coefficients of the vectors $\mathbf{a}$, it could also be calculated on-demand by the oracle and not explicitly stored. This would be an approach that would reduce the space complexity significantly.

Generalizing the Notion of ϵ-active Subfunctions

Similarly as we defined the notion of ϵ-active subfunctions in Section 3.1.7, we could go further in the generalization. We might not use a single global value of ϵ, but introduce these values for each cluster, i.e. have ϵ_i for each $i \in [l]$. The definition of the active subfunction would then also depend on its cluster's ϵ_i. But, we would also have to design a way of treating the multiple values. And if there were issues with non-existence of the locally consistent point with respect to the vector $\boldsymbol{\epsilon}$, we would have to design an (at least partial) ordering on the $\boldsymbol{\epsilon}$ vectors that would decide which are better in the same sense as we searched for the smallest value in Section 3.3.2.

We could go even further and define an $\epsilon_{i,j}$ value for each individual subfunction. This would basically allow us to mark almost[1] an arbitrary set of subfunctions as active. It might help us to deal with the issue of zero step size, where we could alter

[1]Under the assumption that $\epsilon_{i,j} \geq 0$, the actual maxima in each cluster would be always active. On the other hand, if we allowed $\epsilon_{i,j} \in \mathbb{R}$, then the sets could be completely arbitrary.

the set of active subfunctions and obtain another decreasing direction, in which a non-zero step size would be possible. We could e.g. increase the $\epsilon_{i,j}$ of the subfunction that caused the step size to be zero, so it would become active. However, we would still need to design a complete strategy of treating the $\boldsymbol{\epsilon}$ values.

General Convex Subfunctions

In this thesis, we dealt only with the convex piecewise-affine functions, but we could consider a generalized setting with the minimized function in the form

$$f(\mathbf{x}) = \sum_{i=1}^{l} \max_{j=1}^{m_i} f_{i,j}(\mathbf{x}), \tag{6.1}$$

where the subfunctions $f_{i,j}$ are any convex differentiable functions. Then, we believe that we could almost completely re-use the ideas of both the local consistency algorithm and the algorithm that calculates the decreasing direction and minimize the function f. But, we would have to replace the values $\mathbf{a}_{i,j}$ with the gradient of $f_{i,j}$ at the point $\mathbf{x}$ and also design an efficient method of line search.

Two-label Problems

It was shown in Kolmogorov (2005) and reviewed in Werner (2005) that for binary max-sum problems with 2 labels, it holds that a non-empty AC closure is equivalent to having minimum upper bound. Therefore, it may also be generalized in some sense to the convex piecewise-affine minimization that would provide a guarantee on optimality for a subclass of convex piecewise-affine functions.

Branch and Bound

Nguyen et al. (2014) showed that a type of local consistency can be used to maintain a weighted CSP arc consistent during the search of its state space. Similarly, our algorithm could also be used in such setting, for example in a branch and bound algorithm to provide an upper bound on the quality of the solution. Alternatively, it could also be used in branch and cut methods.

Persistency

Shekhovtsov et al. (2017) claims that in the case of a binary max-sum problem, it is possible to find a part of an optimal solution even in polynomial time and also determine which labels will never be in any optimal solution. It remains as an open question for future research whether this notion could be generalized for SMAF minimization.

Other Problems

Last but not least, it is important to test the developed algorithm on other instances than the ones arising from max-sum problems. However, due to the large extent of the thesis, we consider this out of its scope and leave it for future work.

Viewing the minimization of the upper bound as a minimization of a convex piecewise-affine function is beneficial because it is a more general problem that could also be applied in other cases, where such function arises. Additionally, we can also view the transformation from the upper bound minimization to the function as an intermediate layer, which provides a simpler interface when compared to accessing all the values of the binary max-sum problem.

Bibliography

R. K. Ahuja and J. B. Orlin. A capacity scaling algorithm for the constrained maximum flow problem. *Networks*, 25(2):89–98, 1995.

K. R. Apt. The rough guide to constraint propagation. In J. Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99*, pages 1–23, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48085-3.

C. Bessiere. Constraint propagation. In *Foundations of Artificial Intelligence*, volume 2, pages 29–83. Elsevier, 2006.

S. Boyd, L. Xiao, and A. Mutapcic. Subgradient methods. 2003.

M. C. Cooper, S. De Givry, M. Sánchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.

J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

L. R. Ford Jr and D. R. Fulkerson. A simple algorithm for finding maximal network flows and an application to the hitchcock problem. Technical report, RAND CORP SANTA MONICA CA, 1955.

I. Gurobi Optimization. Gurobi optimizer reference manual, 2016. URL `http://www.gurobi.com`.

J. K. Johnson, D. M. Malioutov, and A. S. Willsky. Lagrangian relaxation for map estimation in graphical models. *arXiv preprint arXiv:0710.0013*, 2007.

O. Juan and Y. Boykov. Capacity scaling for graph cuts in vision. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE, 2007.

J. H. Kappes, B. Andres, F. A. Hamprecht, C. Schnörr, S. Nowozin, D. Batra, S. Kim, B. X. Kausler, J. Lellmann, N. Komodakis, and C. Rother. A comparative study of modern inference techniques for discrete energy minimization problem. In *Conf. Computer Vision and Pattern Recognition*, 2013.

V. Kolmogorov. Primal-dual algorithm for convex markov random fields. *Microsoft Research MSR-TR-2005-117*, 2005.

V. Kolmogorov. Convergent tree-reweighted message passing for energy minimization. *IEEE transactions on pattern analysis and machine intelligence*, 28(10): 1568–1583, 2006.

N. Komodakis, N. Paragios, and G. Tziritas. MRF optimization via dual decomposition: Message-passing revisited. In *Intl. Conf. on Computer Vision*, 2007.

Y. Koval' and M. Schlesinger. Two-dimensional programming in image-analysis problems. *Autom. Remote Control*, 37:1269–1285, 1976. ISSN 0005-1179; 1608-3032/e.

V. A. Kovalevsky and V. K. Koval. A diffusion algorithm for decreasing the energy of the max-sum labeling problem. Glushkov Institute of Cybernetics, Kiev, USSR. Unpublished, approx. 1975.

H. Nguyen, S. de Givry, T. Schiex, and C. Bessiere. Maintaining virtual arc consistency dynamically during search. In *Intl. Conf. on Tools with Artificial Intelligence (ICTAI)*, pages 8–15. IEEE Computer Society, 2014.

D. Průša. Two-dimensional languages. *Faculty of Mathematics and Physics, Charles University in Prague*, 2004.

D. Průša and T. Werner. Universality of the local marginal polytope. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 37(4):898–904, April 2015.

D. Průša and T. Werner. Lp relaxations of some np-hard problems are as hard as any lp. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1372–1382. SIAM, 2017.

P. Ravikumar, A. Agarwal, and M. J. Wainwright. Message-passing for graph-structured linear programs: Proximal projections, convergence and rounding schemes. In *Proceedings of the 25th international conference on Machine learning*, pages 800–807. ACM, 2008.

M. Schlesinger. Syntactic analysis of two-dimensional visual signals in the presence of noise. *Cybernetics*, 12(4):612–628, 1976.

M. I. Schlesinger and V. Giginjak. Solving (max,+) problems of structural pattern recognition using equivalent transformations. *Upravlyayushchie Sistemy i Mashiny (Control Systems and Machines), Kiev, Naukova Dumka*, 1, 2007.

A. Shekhovtsov, P. Swoboda, and B. Savchynskyy. Maximum persistency via iterative relaxed inference with graphical models. *PAMI: Transactions on Pattern Analysis and Machine Intelligence*, PP(99), July 2017. ISSN 0162-8828. doi: 10.1109/TPAMI.2017.2730884.

Y. Weiss, C. Yanover, and T. Meltzer. Map estimation, linear programming and belief propagation with convex free energies. *arXiv preprint arXiv:1206.5286*, 2012.

T. Werner. A linear programming approach to max-sum problem: A review. *Research Reports of CMP*, December 2005.

T. Werner. A linear programming approach to max-sum problem: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(7):1165–1179, July 2007. ISSN 0162-8828. doi: 10.1109/TPAMI.2007.1036.

T. Werner. On Coordinate Minimization of Convex Piecewise-Affine Functions. *ArXiv e-prints*, Sept. 2017.

Appendix A

Attached Files

Attached files contain the generated instances, source codes of the algorithm, source files of this file and the master thesis in PDF format. The structure of the directories is described in the following table.

Directory/File	Description/Content
`/instances`	generated instances of SMAF
`/latex`	LaTeX source files of this text
`/src`	source codes of the implementation in C++
`thesis.pdf`	master thesis