Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Compiler Flag Optimization using Fourier-sparse Poset Functions

Master Thesis

Tierry Hörmann

May 2, 2022

Advisors: Prof. Dr. M. Püschel, Chris Wendler
Department of Computer Science, ETH Zürich

Abstract

Automatic compiler optimizations play an important part in the development process of most programmers concerned about the runtime of their application. While modern compilers ship hundreds of optimization heuristics which can be individually configured during compilation with optimization flags, most users resort to using standard optimization levels such as -O3 for their programs. Such optimization levels provide preconfigured default optimization pipelines handcrafted by compiler experts with the goal of working well for every kind of program. While it has been shown that program specific pipelines can improve the runtime of a program considerably, it is a difficult task to find such a good configuration by hand. In this thesis we will present a supervised learning approach for finding good optimization flag combinations, where we are concerned about both a good selection and a good order of the flags. We tackle this task by using linear regression for finding a Fourier-sparse approximation of a function mapping optimization flag combinations to their resulting runtimes. We show that our approximations fit well on unseen data randomly sampled from the optimization space. Minimization of the Fourier-sparse approximation provides us with new flag combinations which perform comparably well to the highest optimization level of LLVM, showing that our theoretical basis has potential when applied to the problem of compiler flag optimizations. Our methods work well on medium two-figured number of flags and have the potential to scale even higher.

Abstract

Automatische Compiler Optimierungen spielen eine grosse Rolle im tagtäglichen Entwicklungsprozess der meisten Entwicklern die bedacht über die Laufzeit ihrer Programme sind. Während moderne Compiler hunderte Optimierungsheutistiken mitpacken, die individuell mit Optimierungsflags eingestellt werden können, weichen trotzdem die meisten Nutzer auf standard Optimierungslevels wie -O3 aus um ihre Programme zu kompilieren. Diese Optimierungslevels stellen prekonfigurierte Optimierungspipelines zur verfügung, welche von Compilerexperten von Hand zusammengestellt wurden um die Laufzeit beliebiger Programme möglichst effektiv zu reduzieren. Eine eigene effektive programmspezifische Pipeline zu konfigurieren stellt sich als schwierige Aufgabe heraus und wird kaum angewendet, obwohl bereits mehrfach gezeigt wurde, dass damit die Laufzeit teilweise erheblich reduziert werden kann. In dieser Arbeit stellen wir ein Machine-Learning-Modell vor, das mittels überwachtem Lernen gute Kombinationen an Optimierungsflags findet. Dabei betrachten wir sowohl die Auswahl, als auch die Ordnung der Flags. Wir gehen dieses Problem an indem wir mittels linearer Regression eine fourier-sparse Approximation einer Funktion lernen, welche Kombinationen an Optimierungsflags zu ihrer Laufzeit abbildet. Wir zeigen dass wir mit unserem Ansatz gute Approximationen finden, die präzise die Laufzeit von noch nicht beobachteten, zufällig ausgewählten Flagkombinationen voraussagt. Durch Minimierung der Fourier-sparsen Approximation können wir neue Flagkombinationen kreieren, die mit dem höchsten Optimierungslevel von LLVM konkurrieren können, was zeigt dass unsere allgemeine Methodik Potential hat in der Anwendung auf das Compiler-Flag Optimierungsproblem. Unsere Methoden funktionieren gut auf einer mittleren zweistelligen Anzahl an Flags und zeigen Potential, dass sie auf noch grössere Anzahlen hoch skalieren können.

Contents

Contents iii

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Related Work . . . 3
1.3 Background . . . 4
1.3.1 Fourier Analysis on discrete domains . . . 4
1.3.2 Permutations . . . 4

2 A Poset of Optimizations **7**
2.1 The domain . . . 7
2.2 Partial Order . . . 8
2.3 Covering Relation . . . 8

3 Learning Fourier-sparse Poset Functions **13**
3.1 Overview . . . 13
3.2 Linear Regression and Sparse Approximation . . . 13
3.3 Computing maximally correlated columns . . . 16
3.4 Minimizing the Fourier-Sparse Function . . . 22
3.5 Speeding up the Enumeration . . . 24
3.6 Selecting the Training Set . . . 26

4 Approximate Poset Enumeration **31**
4.1 Delta-Approximate Enumeration . . . 31
4.2 Monte Carlo Tree Search . . . 32

5 Implementation and Experimental Setup **35**
5.1 Setup and Usage . . . 35
5.1.1 Example Usage . . . 36
5.2 Runtime Measurement Setup . . . 37
5.3 Machine Learning Algorithms . . . 38

6 Results **41**

7 Conclusion and Future Work **47**
7.1 Limitations and Future Work 47

A Appendix **49**
A.1 Proof of proposition 2.3 (partial order) 49
A.2 Proof of proposition 3.2 (total order) 49
A.3 Selection of Benchmark Programs 50

Bibliography **51**

Chapter 1

Introduction

In this chapter we will give an introduction into the main problem we will tackle. We start by giving some motivation for posing our problem and introduce some basic notion which we will use throughout this thesis. Furthermore, we will present a brief summary over the related work in this chapter.

1.1 Motivation

Compiler provided optimizations are powerful tools integrated in every common modern compiler. They are capable of speeding up a program for a very minor cost in compile time. For many programs this cost is negligible, and therefore most developers rely heavily on the usage of the built-in optimizer to speed up their code. Modern compilers, such as GCC or clang, implement optimizations as multiple independent heuristics, being applied over some abstract representation of the code. For example, both GCC and clang implement an optimization which replaces constant expressions directly with their value if it can be calculated at compile time. To control those optimizations, command-line flags are usually provided. While those flags can have different forms, we focus on boolean flags in this thesis, which can simply turn specific optimizations on or off. With the improvements of the compilers over the years, more and more optimizations have become available over the recent years, with GCC 11 now providing 252 optimization flags and clang 13 a total of 316 analysis and transform passes. [1] Such a number of optimizations makes it infeasible for most developers to keep an overview and be able to effectively use the available optimization options for their use-case. As a countermeasure, compiler developers are now providing default optimization pipelines for different *optimization levels*, which are

[1] Number of optimizations available from the output of `gcc --help=optimizers` for GCC and `opt --print-passes` for clang / LLVM

mainly used by developers nowadays. For both GCC and clang, the highest optimization level in terms of performance can be activated with the `-O3` flag. Unfortunately, this default pipeline is often suboptimal for a specific program.

In this thesis our goal is to find a good custom optimization pipeline for a fixed program with a given input, where an optimization pipeline is an ordered sequence of optimization flags. We approach this task in three steps. First we define a novel poset which mathematically represents our space of optimization flag sequences. We can then mathematically express the effect of an optimization flag sequence to the runtime of a program as a function with our poset as its domain, which we will call the target function. We can then derive a Fourier transform for this domain by using preexisting theory [27]. Based on this Fourier transform, our next step will be to learn an approximation of our function, which will be sparse in the Fourier space. By sparse we mean that most of the Fourier coefficients will be zero. We will see that we can learn such an approximation by directly learning the non-zero Fourier coefficients. Finally, we can minimize our Fourier-sparse approximation and return the minimizer.

Our approach takes both the selection of flags and their order into consideration. Previous research has showed that naively enabling all optimizations is often suboptimal. Due to their heuristic nature, some optimizations might degrade the performance of a program, as previous research shows [30, 12]. The problem of choosing such an optimal subset from the available set of optimization flags has been referred to as the *selection problem* in literature.

But not only finding a good subset is of importance, choosing the right order in which they should be executed is also relevant. The intuition is that, since a compiler optimization transforms the code, executing a flag might enable or disable some subsequent optimizations. We can see this effect with a small experiment. For a given benchmark program let's select 10 optimization flags and use them to form a set of pairs between all those selected flags. We can then measure the runtime of the program for every pair of flags twice by first ordering them as they appear in `-O3` and afterwards in reversed order. Figure 1.1 shows a comparison between those two runtimes, where we plotted the relative improvement of the reversed order compared to the order in `-O3`. We can see that with this simple method, we can improve our runtime by up to roughly 15%. The problem of finding an optimal flag order for a given selection of flags is called the *phase-ordering problem* in literature. It is well known that there is no general optimal order for a selection of optimizations, and there have been multiple efforts to tackle phase-ordering for optimization flags [10].

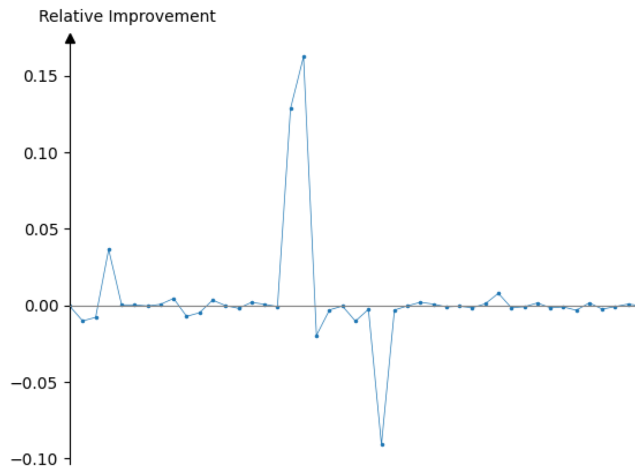

Figure 1.1: The relative runtime improvement between pairs of optimization flags.

1.2 Related Work

The general problem of compiler optimizations, also referred to as compiler autotuning, has been around for several decades [1] and has been tackled with machine learning since the late 90's [16]. The majority of previous work in this field of research uses iterative compilation techniques, where the program is iteratively compiled and evaluated at different times throughout the optimization process. [4] is one of the earliest publication to show that iterative compilation is a viable option for compiler optimization. Iterative compilation is often combined with genetic algorithms [9, 17, 6]. [3] uses Bayesian network learners to outperform previous methods for the selection problem and reaches performance speedups of up to 2.7 compared to `-O3` on an ARM processor with GCC on cBench benchmarks. More recent work uses Bayesian optimizations with random forests for tackling the selection problem [8]. In [2], the authors use 4 flag combinations derived from LLVM's `-O3` with an iterative predictive method for tackling the phase-ordering problem. Our approach is unique in its way of tackling the selection problem and the phase-ordering problem simultaneously without iterative compilation and with its supervised learning approach for analyzing the full optimization space without explicitly considering static or dynamic program features.

1.3 Background

1.3.1 Fourier Analysis on discrete domains

As explained earlier, our approach requires us to derive a Fourier transform for our novel poset. Here we will introduce the mathematical foundation for Fourier analysis on such discrete domains, which can be derived from the general framework of algebraic signal processing (ASP) [24]. ASP introduces an axiomatic approach on defining a signal processing theory for any algebraic domain. [27] proposes a Fourier transform derived from ASP, which we can apply for our poset.

Definition 1.1 (Fourier basis) *The Fourier basis of a poset $\mathcal{P}$ is a matrix Φ indexed by the elements in $\mathcal{P}$ with*

$$\Phi_{y,x} = \iota_{y \leq x}$$

where $\iota_{y \leq x}$ indicates the characteristic (or indicator) function of $x \leq y$

$$\iota_{y \leq x} = \begin{cases} 1, & y \leq x \\ 0, & else \end{cases}$$

The Fourier basis allows us to linearly transform a vector of Fourier coefficients (spectrum) $\widehat{\mathbf{s}}$, indexed by the elements in $\mathcal{P}$, into its corresponding signal $\mathbf{s}$:

$$\mathbf{s} = \Phi \widehat{\mathbf{s}}$$

1.3.2 Permutations

Since we consider the phase-ordering problem, we will naturally encounter permutations when mathematically describing our methods. Here we introduce the relevant background on the mathematics for permutations which we'll be using throughout this thesis.

Definition 1.2 (Permutation) *A* permutation *π over a discrete set S is a bijective mapping from S to itself.*

$$\pi : S \to S$$

At this part it makes sense to introduce some notational conventions which help us with keeping expressions clean and brief. We will generally use lowercase Greek letters for naming permutations and their uppercase counterpart for the domain. Furthermore, we'll use the common notation S_n to indicate the set of all permutations with n elements (over some fixed set). We have

$$|S_n| = n!$$

We will frequently come across the inverse permutation of some permutations π, which we'll denote as π^{-1}. Additionally, we'll write I_n for the identity permutation on N.

Instead of defining a permutation directly over some set Π, we will usually first index the elements from 1 to n and afterwards define the permutation over $N = \{1, \dots, n\}$ (N_n for a concrete n). We'll use the expression $N(x)$ to refer to the index of $x \in \Pi$ and $N^{-1}(i)$ to refer to the original element, whenever we need to be explicit about the original value. In literature, it is very common to define permutations directly over N and ignoring the underlying set. For us however, since we will work a lot with partial permutations, we will need to be more careful.

With the above technique, we can make use of the common one-line notation for expressing permutations. A permutation in one-line notation is simply a tuple where the elements are ordered according to the permutation. For example a permutation π on N_3 with $\pi(3) = 1$, $\pi(2) = 3$ and $\pi(1) = 2$ would be written as $[2, 3, 1]$. Note that $\pi^{-1}(x)$ returns the index of x in the one-line notation.

While our definition of a permutation is sound, it is rather impractical to work with it directly. There is especially no straightforward way of comparing two permutations, which will become crucial as soon as we try to express our target domain as a poset. A better way of describing a permutation for our use-case is with an inversion set. There are two common ways of defining inversion sets: One uses pairs of places (i, j) while the other one uses pairs of elements $(x, y) = (\pi(i), \pi(j))$. We use the element-based definition in this thesis, which will become more convenient for our purpose.

Definition 1.3 (Inversion Set) *The* inversion set *of a permutation π is defined as:*

$$inv(\pi) := \{(\pi(i), \pi(j)) \in \Pi^2.\ i < j \land \pi(i) > \pi(j)\} \tag{1.1}$$

$$= \{(x, y) \in \Pi^2.\ \pi^{-1}(x) < \pi^{-1}(y) \land x > y\} \tag{1.2}$$

An inversion set uniquely identifies a permutation. This mapping from the space of permutations to the space of sets of pairs from N is however injective. In other words, there exist some sets of pairs from N, which cannot identify a permutation.

Example 1.4 *Let $Y = \{(2, 1), (1, 0)\}$. Now let's try to construct a permutation π with Y as an inversion set. Note that $\pi^{-1}(2) < \pi^{-1}(1)$ and $\pi^{-1}(1) < \pi^{-1}(0)$ implies $\pi^{-1}(2) < \pi^{-1}(0)$ and clearly $2 > 0$. But $(2, 0) \notin Y$, which makes it impossible for Y to be an inversion set for any π.*

Another common way of describing a permutation is by decomposing it into transpositions. A transposition is basically a permutation which switches

two elements in the one-line notation and leaves the rest of the sequence in place.

Definition 1.5 (Transposition) *A transposition $\tau = [i, j]$ is a permutation where $\tau(i) = j$ and $\tau(j) = i$ and $\tau(x) = x$ for all $x \neq i$ and $x \neq j$.*

From algebra, we know that every permutation can be expressed as a sequence of transpositions. As an example, consider the permutation $\alpha = [1, 4, 2, 3]$ which can be written as $\alpha = (3, 4) \circ [1, 4, 3, 2] = (3, 4) \circ (2, 4)$.

Unfortunately permutations won't be able to fully describe our domain of interest. If we would only use pure permutations, we would discard the selection problem. Luckily there already exist a mathematical foundation, which can fully describe the elements in our domain.

Definition 1.6 (k-permutation) *A* k-permutation *π over a base domain S is a permutation over $\Pi \subseteq S$ with $|\Pi| = k$.*

We will generally use n to indicate the cardinality of the base set of a k-permutation. The total number of k-permutations for some n is

$$\frac{n!}{(n-k)!}$$

For expressing k-permutations, we can also use the one-line notation. The 3-permutation π on $\{1, 4, 5\}$ with $\pi(1) = 5$, $\pi(4) = 1$ and $\pi(5) = 4$ would then be written as $[5, 1, 4]$. The position of an element $x \in \Pi$ in the resulting tuple can be expressed by $N_k(\pi(x))$ with the translating function N_k upholding the order between two elements.

To describe our target domain, we can now just refrain form fixing k and include k-permutations for some arbitrary k. This more general object is called a *partial permutation*.

Definition 1.7 (Partial Permutation) *A* partial permutation *is a k-permutation for some arbitrary $k \in \{0, \ldots, n\}$.*

Let's denote by $\mathcal{P}_n$ the set of all partial permutations over N. For a fixed n, there are a total of

$$|\mathcal{P}_n| = \sum_{k=0}^{n} \frac{n!}{(n-k)!} = \sum_{k=0}^{n} \frac{n!}{k!}$$

partial permutations. We can already see that with large ns, the discrete domain we are interested in will become huge.[2]

[2] For $n = 60$ we already exceed current estimates of the Eddington number of approximately 10^{80}, which is the number of protons in the observable universe.

Chapter 2

A Poset of Optimizations

In this chapter we will present a novel poset which we will use to mathematically describe the set of optimization flag combinations. We will first introduce the definition of elements in our poset. Afterwards we will define a partial relation and derive the cover relation for it.

2.1 The domain

In chapter 1 we already hinted at the mathematical formalisms we will use throughout this thesis to describe an optimization pipeline, which we'll from now on simply call an *optimization*.

Definition 2.1 (Optimization) *An* optimization *is a partial permutation over a set of compiler optimization flags.*

While we will focus mainly on partial permutations, it is still possible to apply most of the techniques presented in this thesis on other domains as well, as long as we can construct some meaningful partial order. The selection of a domain directly imposes restrictions onto which optimizations are considered in the final optimization problem. For example if we only want to consider the selection problem, we could use a powerset as our domain with the subset relation as the partial order along with a Fourier transform for set functions, such as the one proposed in [25]. For the phase-ordering problem, we could choose the weak order of permutations, which is also a lattice [20]. The weak order of permutations uses the subset relation between the inversion sets of two permutations for its partial relation.

The space of partial permutations can be seen as a combination of a powerset and the space of permutations. We will therefore be able to consider the selection problem and the phase ordering problem in one go. Nevertheless, we still impose some restrictions on our optimization space. In particular, we do not explicitly support an item to occur multiple times, which might

be desirable. Indeed, when running clang with its highest optimization level -O3, it will apply some transformation passes multiple times. With partial permutations we could support this behavior by adding the same optimization flag multiple times into our base set and treating the multiple occurrences as individual flags. This would however make our problem more difficult than necessary. A more proper way might be to work with multiset permutations in our domain. Here we will however stick to the above definition of partial permutations to keep everything simple enough.

2.2 Partial Order

Currently, every element in our domain stands by itself. In other words we do not have any possibility of inferring an element's function value from other elements, which will be our ultimate goal. We can handle this by giving our domain some structure, which imposes a relation on the elements. Since we do not consider the program code and won't have any input other than the elements in our domain, the structure we define on the domain will be integral for constructing the feature space for our ML model. As common for discrete sets, we will give the domain a structure by defining a partial order over it. The partial order is a key ingredient in our approach, since it directly influences the feature space we will use and therefore the success of our final model. Hence, a careful definition of the partial order is important. We will choose an arguably natural definition, which will give us some nice properties and can be evaluated relatively efficiently, as we'll later see.

Definition 2.2 (Optimization Poset) *The* poset of optimizations *is the set of all optimizations over N with a partial order $\preceq$ defined as*

$$\alpha \preceq \beta :\Leftrightarrow A \subseteq B \wedge inv(\alpha) \subseteq inv(\beta) \tag{2.1}$$

for arbitrary optimizations α and β.

Proposition 2.3 *$\preceq$ is a partial order on $\mathcal{P}$.*

Intuitively, $\alpha \preceq \beta$ indicates that β applies at least as many optimization techniques as α, where an optimization technique can be either a new flag or a deviation from the default order. A proof of proposition 2.3 can be found in appendix A.1.

2.3 Covering Relation

When working with posets, we quickly come across a notion called *covering relation*. For example, visualizing the poset with a Hasse diagram makes use of the covering relation. The covering relation will later play an important role.

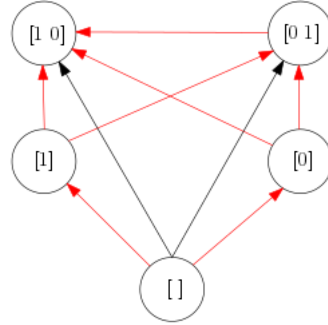

Figure 2.1: An optimization poset with $n = 2$. Red edges represent covering relations.

Definition 2.4 (Covering relation) *Given a poset* $(X, \preceq)$*, an element* $a \in X$ *is* covered *by* $b \in X$ *(*$a \sqsubset b$*) iff* $a \prec b$ *and there exists no* $c \in X$ *such that* $a \prec c \prec b$*.*

As an example, consider figure 2.1, which illustrates an optimization poset with $n = 2$ as a directed acyclic graph, where the edges represent the partial order relation. Red edges indicate covering relations.

Let's try to apply this definition to our poset to derive a direct way for finding covers of an element α. We can approach this by once more considering the powerset clause ($A \subseteq B$) and the permutation clause ($inv(\alpha) \subseteq inv(\beta)$) separately. For the powerset poset it is easy to see that the covers of a set A are all those sets B which contain a single additional element x: $B = A \uplus \{x\}$, where $\uplus$ indicates disjoint union. For the weak order of permutations, we also have an easy way of identifying covers. Given a permutation α, the inversion set of a cover β contains a single additional element $(\alpha(i), \alpha(j))$: $inv(\beta) = inv(\alpha) \uplus \{(\alpha(i), \alpha(j))\}$. This is valid for exactly those permutations which are one transposition $(i, i+1)$ off from α with $\alpha(i) < \alpha(i+1)$. In other words, if we switch two neighboring elements in our one-line notation, which are in increasing order, we get a cover of the original permutation. For example $[1, 4, 2, 3] \sqsubset [1, 4, 3, 2]$.

Let's combine those insights to find the cover relation for our poset. First consider the case $A = B$. With this restriction, both α and β are permutations over the same set A, and we can apply the above transposition method for finding the covers of α. Now if $A \neq B$, we know from the powerset poset that $B = A \uplus \{x\}$. But we cannot simply insert x anywhere into α. We need to ensure that x cannot be switched with a neighbor after insertion such that it removes an inversion. So a cover must have the following form: $\beta = [\alpha_1, \ldots, \alpha_i, x, \alpha_{i+1}, \ldots, \alpha_k]$ with $\alpha_i < x < \alpha_{i+1}$ or $\beta = [x, \alpha_1, \ldots, \alpha_k]$ with $x < \alpha_1$ or $\beta = [\alpha_1, \ldots \alpha_k]$ with $\alpha_k < x$.

Before we continue with the formal conclusion, let's first introduce a new notion, which abbreviates comparisons between optimizations with different domains.

Definition 2.5 (Reduction) *The* reduction π_X *of a partial permutation* π *onto a set* $X \subseteq \Pi$ *is a partial permutation with domain* X *with the same order of elements:*

$$\forall i, j \in X. \pi^{-1}(i) < \pi^{-1}(j) \leftrightarrow \pi_X^{-1}(i) < \pi_X^{-1}(j) \tag{2.2}$$

As an example, let $\pi = [2, 4, 7, 1]$. Then $\pi_{N_2} = [2, 1]$. From the one-line notation we can also see how the inversion set changes from π to π_X. The inversion set of π_X contains all the pairs from $inv(\pi)$ which only have items in X.

Lemma 2.6 *For an arbitrary partial permutation* π *and a subset* $X \subseteq \Pi$*:*

$$inv(\pi_X) = \{(i, j) \in inv(\pi).\ i, j \in X\} \subseteq inv(\pi) \tag{2.3}$$

Proof The proof follows directly from equation (2.2) and definition 1.2.

$$\begin{aligned} inv(\pi_X) &= \{(i, j) \in X^2.\ i < j \wedge \pi_X^{-1}(i) > \pi_X^{-1}(j)\} \\ &= \{(i, j) \in X^2.\ i < j \wedge \pi^{-1}(i) > \pi^{-1}(j)\} \\ &= \{(i, j) \in \Pi^2.\ i, j \notin \Pi \setminus X \wedge i < j \wedge \pi^{-1}(i) > \pi^{-1}(j)\} \\ &= \{(i, j) \in inv(\pi).\ i, j \in X\} \end{aligned}$$

□

Another nice property of the reduction operator is that it allows us to rewrite the subset relation between the inversion sets as used in the definition of our partial order in 2.2. Since the inversion set of α can only contain elements from A, we can reduce β to A and compare the inversion sets there.

Lemma 2.7 *For partial permutations* α, β *with* $A \subseteq B$*, the assertions* $inv(\alpha) \subseteq inv(\beta)$ *and* $inv(\alpha) \subseteq inv(\beta_A)$ *are equivalent.*

Proof We prove the equivalence by proving the implication in both directions.

- $\Rightarrow$: Assume $inv(\alpha) \subseteq inv(\beta)$. Fix $(i, j) \in inv(\alpha) \subseteq A^2$. Clearly $i, j \notin B \setminus A$. Together with lemma 2.6 and $(i, j) \in inv(\beta)$, which we know from our assumption, we conclude $(i, j) \in inv(\beta_A)$.
- $\Leftarrow$: Assume $inv(\alpha) \subseteq inv(\beta_A)$. With lemma 2.6 we directly conclude $inv(\alpha) \subseteq inv(\beta_A) \subseteq inv(B)$ □

Now back to the covering relation. The reduction operator allows us to easily express that we only insert an element into an optimization and do not alter the order of the original elements in any way.

Theorem 2.8 *Let* $\alpha \preceq \beta$ *be two optimizations in* $\mathcal{P}_n$*.* β *covers* α *iff either*

1. $B = A$ *and* $\beta = (i, i + 1) \circ \alpha$ *where* $i \in N_{n-1}$

2. *$B = A \uplus \{x\}$ and $\beta_A = \alpha$ and $\beta(i-1) < x$ if $i > 1$ and $\beta(i+1) > x$ if $i < n$ with $i = \beta^{-1}(x)$*

Proof Without loss of generality, assume that for i as defined in 2.8 case 2, $1 < i < n$. First consider the case where $A = B$. In this case, both α and β are permutations over the same domain A. Additionally, β is defined exactly as the covers in the weak order on permutations. We can therefore focus on the case where $A \subset B$, or even $B = A \uplus \{x\}$, because if $|A| + 1 < |B|$, we can choose γ with $A \subset \Gamma \subset B$ and $\gamma = \beta_\Gamma$. We can also assume that $\beta_A = \alpha$, because otherwise we can choose $\gamma = \beta_A$. Furthermore, if $\beta(i-1) > x$ we choose $\gamma = (i-1, i) \circ \beta$ and if $\beta(i+1)$ we choose $\gamma = (i, i+1) \circ \beta$ with i defined as in 2.8 case 2.

What is left to prove is that β, as defined in case 2, is indeed a cover of α. Let's assume by contradiction that there exists a γ with $\alpha \prec \gamma \prec \beta$. First, we can see from lemma 2.6 on the preceding page that $\gamma_A = \alpha = \beta_A$ and therefore $\Gamma = B$. So γ and β are both permutations on B where $inv(\beta) \setminus inv(\gamma)$ only contains pairs containing x. Let's take a look at a series of permutations over B $\pi_1 \sqsubset \pi_2 \sqsubset \cdots \sqsubset \pi_m$ with $\pi_1 = \gamma$ and $\pi_m = \beta$ where $\sqsubset$ indicates the covering relation for the weak order on permutations. Note that every π_{j+1}, especially $\pi_m = \beta$, can be expressed as $(k, k+1) \circ \pi_j$ where $\pi_j(k) = x$ or $\pi_j(k+1) = x$ and $\pi_j(k) < \pi_j(k+1)$. But since $\beta(i-1) < x$ and $\beta(i+1) > x$, there is no π_{m-1} with the above property and therefore $\gamma = \pi_1 = \pi_m = \beta$ which contradicts our earlier assumption that $\gamma \prec \beta$. $\square$

Chapter 3

Learning Fourier-sparse Poset Functions

3.1 Overview

In chapter 1 we have explained that our goal is to estimate a function over the poset of optimizations, as defined in chapter 2. As we have seen, the cardinality of this domain grows very rapidly by an order of $\mathcal{O}(n!)$. So exhaustively measuring the function quickly gets infeasible. We will rather need to estimate a sparse representation of the function in some feature space from which we can compute function values for individual elements in the domain. As mentioned earlier, we will be using the Fourier basis as proposed in [27] as our feature space and learn a Fourier-sparse estimation of our target function. This chapter introduces our general approach for learning such an estimate and proposes a method for maximizing a Fourier-sparse function on the optimization domain with an algorithm solving the underlying NP-hard problem.

The general framework of learning and minimizing Fourier-sparse poset functions with linear regression and enumeration, as presented in this chapter, was developed by the PhD-advisor of this thesis, Chris Wendler, during his doctoral thesis at ETH Zürich. The contribution of this thesis lies within the application to the poset of optimizations.

3.2 Linear Regression and Sparse Approximation

In chapter 1 we introduced the Fourier basis on DAGs as proposed in [27]. We also discovered that, given a vector of Fourier coefficients $\hat{\mathbf{s}}$, we can reconstruct the original signal $\mathbf{s}$ with a linear transformation. Applied to our poset of optimizations this transformation can be expressed as

$$\mathbf{s} = \mathcal{F}\widehat{\mathbf{s}} \tag{3.1}$$

where the columns of $\mathcal{F}$ are the Fourier basis vectors (frequencies)

$$\mathbf{f}^g = (\iota_{\{g \preceq x\}})_{x \in \mathcal{P}}$$

Unfortunately we cannot assume that we know the function values for every $x \in \mathcal{P}$, as described before. Let $\mathcal{X} \subseteq \mathcal{P}$ indicate a set of training data points (which we can select freely) and y the according function values (which we can measure with our target program and input). We can then use linear regression to find the non-zero Fourier coefficients $w = w^k$. Let's stick to Lasso regression [29], since this motivates a sparse w, and assume that we already have somehow selected a set $\mathcal{G} = \mathcal{G}^k \subseteq \mathcal{P}$ of optimizations, for which we assume non-zero Fourier coefficients. We'll use $m := |\mathcal{X}|$, $k := |\mathcal{G}|$ and $p := |\mathcal{P}|$. Our objective function is

$$\frac{1}{2}\|y - \Phi w\|_2^2 + \lambda \|w\|_1$$

where $\Phi = \Phi^k$ is the Fourier basis $\mathcal{F}$ reduced to the rows in $\mathcal{X}$ and columns in $\mathcal{G}$:

$$\Phi_{x \in \mathcal{X}, g \in \mathcal{G}} = \iota_{\{g \preceq x\}}$$

We can minimize our Lasso objective using coordinate descent [31]. The final predicted w^* can be interpreted as the Fourier coefficients of a Fourier-sparse estimate of the target function on the optimization domain.

While we now have a method of computing Fourier-sparse approximations, we don't know yet how to select $\mathcal{G}$. Since the Lasso objective is a regularized linear least-squares problem, let's consider matching pursuit [19]. Matching pursuit is an algorithm for minimizing the squared L^2-norm of the residual of a linear model such that only a fixed number k of weights are non-zero. More formally, for our case it finds an approximate solution for the following optimization problem

$$\min_{w \in \mathbb{R}^p} \frac{1}{2}\|y - \Phi^p w^p\|_2^2 \text{ s.t. } \|w^p\|_0 \leq k$$

where $\|\cdot\|_0$ indicates the L^0 pseudo norm counting the number of non-zero dimensions. The algorithm iteratively adds new frequencies which, along with their optimal weights (optimal in the sense of minimizing the ordinary least-squares problem), minimize the squared L^2-norm of the residual. Let's step through the mathematical foundation of the algorithm for our case.

Let $h^i(x)$ be our estimate for $x \in \mathcal{X}$ at step i of the matching pursuit algorithm.

$$h^i(x) := \Phi^i_{x,:} w^i = \sum_{g \in \mathcal{G}^i} \Phi^i_{x,g} w^i_g = \sum_{\substack{g \in \mathcal{G}^i \\ g \leq x}} w^i_g$$

The residual r^i at step i is defined as

$$r^i := y - h^i$$

where h^i is the vector representation of $h^i(x)$ indexed by $x \in \mathcal{X}$: $h^i_x = h^i(x)$.

For a new iteration $i+1$, matching pursuit selects a new element $g_{i+1} \in \mathcal{P} \setminus \mathcal{G}^i$ and adds it to the previous set of elements with non-zero weights: $\mathcal{G}^{i+1} = \mathcal{G}^i \uplus \{g_{i+1}\}$. It does so by selecting g_{i+1} such that the squared L^2-norm of the new residual is minimal.

$$g_{i+1} = \underset{g_{i+1} \in \mathcal{P} \setminus \mathcal{G}^i}{\arg\min} \|r^{i+1}\|_2^2 = \underset{g_{i+1} \in \mathcal{P} \setminus \mathcal{G}^i}{\arg\min} \|r^i - w^*_{i+1} \Phi_{:,i+1}\|_2^2$$

where $w^*_{i+1} := w_{g i+1^*}$ and similarly $\Phi_{:,i+1} := \Phi_{:,g_{i+1}}$ indicate the new optimal non-zero weight and associated frequency for g_{i+1} respectively.

For a given g_{i+1}, we can find w^*_{i+1} with an orthogonal projection onto r^i.

$$w^*_{i+1} = \frac{\langle r^i, \Phi_{:,i+1} \rangle}{\|\Phi_{:,i+1}\|_2^2} \tag{3.2}$$

where $\langle \cdot, \cdot \rangle$ indicates the inner product.

Now we can conclude

$$\begin{aligned}
\|r^i\|_2^2 &= \|r^{i+1} + w^*_{i+1} \Phi_{:,i+1}\|_2^2 \\
&= \|r^{i+1}\|_2^2 + \overbrace{2 w^*_{i+1} \langle r^{i+1}, \Phi_{:,i+1} \rangle}^{0} + (w^*_{i+1})^2 \|\Phi_{:,i+1}\|_2^2 \\
&= \|r^{i+1}\|_2^2 + \frac{\langle r^i, \Phi_{:,i+1} \rangle^2}{\|\Phi_{:,i+1}\|_2^2}
\end{aligned}$$

Hence, we select the g_{i+1} which maximizes the expression

$$\frac{\langle r^i, \Phi_{:,i+1} \rangle^2}{\|\Phi_{:,i+1}\|_2^2}$$

If now all available columns $\Phi_{:,i+1}$ had the same squared L^2-norm, we could simply select the g_{i+1} with the maximum absolute correlation between r^i and $\Phi_{:,i+1}$. We'll explain how we find such a g_{i+1} in the next section. That our columns all have the same norm is naturally not true per se, but with a simple linear transformation it suddenly is.

$$\|2\Phi_{:,i+1} - 1\|_2^2 = m$$

since $2\Phi_{:,i+1} - 1 \in \{-1, 1\}^{i+1}$

With the bilinearity of the inner product, we can also see that maximizing the correlation with the transformed column is equivalent to maximizing the correlation with the column directly.

$$\begin{aligned}\underset{\Phi_{:,i+1}}{\arg\max}\langle r^i, 2\Phi_{:,i+1} - 1\rangle &= \underset{\Phi_{:,i+1}}{\arg\max}\, 2\langle r^i, \Phi_{:,i+1}\rangle - r^i \\ &= \underset{\Phi_{:,i+1}}{\arg\max}\langle r^i, \Phi_{:,i+1}\rangle\end{aligned}$$

We are not quite done yet. The matching pursuit algorithm assumes a squared L^2-norm cost function, but earlier on we decided to use Lasso. Fortunately, the Lasso objective also has a closed-form solution when optimizing an individual weight, which we can use instead of 3.2 on the preceding page.

$$w^*_{i+1} = S\left(\frac{\langle r^i, \Phi_{:,i+1}\rangle}{\|\Phi_{:,i+1}\|_2^2}, \lambda\right)$$

with

$$S(g, \lambda) = \begin{cases} g - \lambda & \text{if } g > \lambda \\ g + \lambda & \text{if } g < -\lambda \\ 0 & \text{else} \end{cases}$$

Finally, we can alternately find a new column for Φ with an iteration of matching pursuit and do coordinate descent for some iterations, until we have reached a suitable k, thus concluding our final algorithm.

Let's summarize our above results with an algorithm in pseudocode.

3.3 Computing maximally correlated columns

We are now two steps away from completing our goal. First we need a method for finding the maximum correlation between a residual and a frequency, so we can compute a Fourier-sparse poset function estimate of our target. And finally we need a method for minimizing this Fourier-sparse function. We will see that we can solve both those problems with the same basic approach.

Let's first focus on finding the maximally correlated frequency φ^* of optimization $g^* \in \mathcal{P}$ for a given residual r over $\mathcal{P}$, where both φ^* and r are indexed over a set $\mathcal{X} \subseteq \mathcal{P}$. Let φ indicate the frequency belonging to some arbitrary $g \in \mathcal{P}$, indexed as well over $\mathcal{X}$. We can reformulate the inner product, which gives rise to an upper bound.

Algorithm 1: Algorithm for learning a Fourier-sparse approximation of a poset function.

Input: $\mathcal{X} \subseteq \mathcal{P}$, $y \in \mathbb{R}^{\mathcal{X}}$, $k \in \mathbb{N}$
Output: $\mathcal{G} \subseteq \mathcal{P}$, $w \in \mathbb{R}^{\mathcal{G}}$

```
1  Φ ← ();
2  w ← ();
3  G ← ∅;
4  r ← y;
5  for i ∈ {0, ..., k} do
6  |  g ← arg max_{g∈P} ⟨r, (ι_{g⪯x})_{x∈X}⟩;
7  |  G ← G ⊎ {g};
8  |  φ ← (ι_{g⪯x})_{x∈X};
9  |  w* ← S(⟨r, 2φ − 1⟩/|X|, λ);
10 |  Φ ← (Φ_{:,1}, ..., Φ_{:,i}, φ);
11 |  w ← (w_1, ..., w_i, w*);
12 |  Φ, w ← cd(y, Φ, w) ;                    /* Coordinate descent */
13 |  r ← y − Φw;
14 end
```

$$\begin{aligned}\langle r, \varphi \rangle = \langle r, (\iota_{\{g \preceq x\}})_{x \in \mathcal{X}} \rangle &= \sum_{\substack{x \in \mathcal{X} \\ g \preceq x}} r_x \\ &= \sum_{\substack{x \in \mathcal{X} \\ g \preceq x \\ r_x > 0}} r_x - \sum_{\substack{x \in \mathcal{X} \\ g \preceq x \\ r_x < 0}} -r_x \\ &\leq \sum_{\substack{x \in \mathcal{X} \\ g \preceq x \\ r_x > 0}} r_x \end{aligned} \tag{3.2}$$

With our problem, we are however interested in the absolute correlation, for which we can find the following upper bound.

$$|\langle r, \varphi \rangle| \leq \max \left(\sum_{\substack{x \in \mathcal{X} \\ g \preceq x \\ r_x > 0}} r_x, \sum_{\substack{x \in \mathcal{X} \\ g \preceq x \\ r_x < 0}} -r_x \right) =: \mu(g; r)$$

Note that $\mu(g; r)$ is monotonically decreasing with g. For arbitrary $g_1, g_2 \in \mathcal{P}$ with $g_1 \preceq g_2$ we have $\mu(g_1; r) \geq \mu(g_2; r)$.

Now let's say we have already found a g' whose frequency has a large correlation c with r. During our remaining search for the maximally correlated frequency, we can then discard all $g_2 \succeq g_1$ if $\mu(g_1; r) \leq c$. The idea of our algorithm is now to start at the root of our poset (which is the empty set) and work our way upwards in some manner, keeping track of the largest correlation c we found yet. As soon as our upper bound μ for the correlation drops below c, we no longer need to check any larger elements and can switch directions. When there are no more elements left to check we can be sure we have found the element with the largest correlation.

The problem of finding all elements g from a substructure poset (which is a poset with a unique least element), conforming to an anti-monotonic property, such as $g \geq \mu(g^*; r)$, is called the *enumeration problem*. [21] proposes an algorithm called *reverse search*, which solves the enumeration problem for arbitrary substructure posets. To apply reverse search, we need to define an inverse reduction mapping, which we can construct from a total order and the cover relation of the poset. In chapter 2 we have already derived the cover relation, so we only need to define a total order for our poset. The choice for our total order will not have an impact on our result, since reverse search solves the enumeration problem with any total order. However, our choice might impact the performance of our final algorithm. We choose a rather straight-forward definition based on the lexicographic order of subsets.

Definition 3.1 (Total Order on Optimization Poset) *Let $\alpha, \beta \in \mathcal{P}$. $\alpha \leq \beta$ is defined as*

$$\begin{aligned} \alpha \leq \beta &:\Leftrightarrow A \leq_{lex} B \wedge (A = B \rightarrow inv(\alpha) \leq_{lex} inv(\beta)) \\ &:\Leftrightarrow A <_{lex} B \vee A = B \wedge inv(\alpha) \leq_{lex} inv(\beta) \end{aligned}$$

where $\leq_{lex}$ indicates the lexicographic order of subsets with the order of pairs being induced by the lexicographic order respecting the order of elements.

Proposition 3.2 *$\leq$ is a total order on $\mathcal{P}$.*

A proof of proposition 2.3 can be found in appendix A.1.

Lemma 3 of [21] now provides us with a recipe for constructing the inverse reduction mapping.

$$f^{-1}(\alpha) = \{\beta \in \mathcal{P}. \quad \alpha \sqsubset \beta \wedge \forall \gamma \sqsubset \beta.\, \alpha \leq \gamma\}$$

In words, the inverse reduction mapping of an optimization α is the set of covers β of α for which α is the smallest element when considering all elements that β covers, according to the total order $\leq$. Reverse search now does exactly what we explained earlier. It starts at the root (the empty set) of our poset and recursively runs over the elements returned by the

inverse reduction mapping, whose upper bound μ of the correlation with r is larger than the largest correlation yet found. By using the total order in the inverse reduction mapping, we never encounter an element in $\mathcal{P}$ twice. The inverse reduction mapping therefore implicitly defines a traversal tree over our poset, where we can prune subtrees according to our anti-monotone property μ.

To wrap things up, let's derive a more direct way for computing the inverse reduction mapping for an element of $\mathcal{P}$. First let's note that with our total order, the lexicographic order between the domains A and B is dominant compared to the lexicographic order between the inversion sets. This means that for some $\alpha \in \mathcal{P}$, the inverse reduction mapping can only contain covers β with $B = A$ if β covers no $\gamma \in \mathcal{P}$ with $\Gamma \subset B$. More concrete, β cannot conform to the condition of case 2 in theorem 2.8. We call such a β *element stable*.

Definition 3.3 (Element Stability) *An optimization $\beta \in \mathcal{P}$ is called* element stable *if either $k = |B| = 0$ or $k > 1$, it contains no rising sequence larger than* 2 *and $\beta(1) > \beta(2)$ and $\beta(k-1) > \beta(k)$.*

We are not quite done yet with the case $A = B$. We now also need to make sure that every $\gamma \sqsubset \beta$ with $\Gamma = B$ is larger than α. More precisely, that $inv(\alpha) \leq_{\text{lex}} inv(\gamma)$. Luckily, this is relatively easy to check. The inversion introduced by transposition $(i, i+1)$ must be lexicographically larger than the largest one present in α. We can find the largest inversion in linear time, as we'll show a little later.

For now let's also take a look at the case $B = A \uplus \{x\}$. Here we don't have to consider any γ with $\Gamma = B$, since $\alpha \leq \gamma$ would always be true (except if x is smaller than all elements in A). So we just need to check that $A \leq_{\text{lex}} \Gamma$ which, similar to earlier, is exactly then the case when x is larger than all elements in A.

Theorem 3.4 (Inverse Reduction Mapping) *For a given optimization $\alpha \in \mathcal{P}$ the inverse reduction mapping $f^{-1}(\alpha)$ contains exactly those elements $\beta \sqsupset \alpha$ for which the following conditions hold.*

- *If $B = A \uplus \{x\}$, then $\forall a \in A.\ \beta_{B \setminus \{a\}} \sqsubset \beta \wedge a < x$*
- *If $B = A$ with $\beta = (i, i+1) \circ \alpha$, then β is element stable and*

$$(\beta(j), \beta(j+1)) \leq_{lex} (\beta(i), \beta(i+1))$$

  for all $j \in \mathrm{N}_{|A|-1}$ with $\beta(j) > \beta(j+1)$.

Proof We prove the theorem from the definition of the reduction mapping from [21] for the optimization poset and $\leq$. The reduction mapping of a

non-empty optimization β is defined as the optimization $\alpha \sqsubset \beta$ with $\alpha \leq \gamma$ for all $\gamma \sqsubset \beta$. Let's do a case distinction on the structure of β.

If β is element stable, then for any $\gamma \sqsubset \beta$ (and therefore also for α) we have $\Gamma = B$ and $\beta = (j, j+1) \circ \gamma$ with $i \in N_{|A|-1}$ and $\gamma(j) < \gamma(j+1)$, according to case 1 of theorem 2.8. Note that such an i must exist because the empty set is the unique least element of the optimization poset and therefore every optimization except for the empty set most cover another optimization. Let's assume, without loss of generality, that $j \neq i$. We can derive

$$inv(\gamma) \uplus \{(\beta(j), \beta(j+1))\} = inv(\beta) = inv(\alpha) \uplus \{(\beta(i), \beta(i+1))\}$$

$$inv(\gamma) = inv(\alpha) \uplus \{(\beta(i), \beta(i+1))\} \setminus \{(\beta(j), \beta(j+1))\}$$

From definition 3.1 we can see that $inv(\alpha) \leq_{\text{lex}} inv(\gamma)$. Hence

$$inv(\alpha) \leq_{\text{lex}} inv(\alpha) \uplus \{(\beta(i), \beta(i+1))\} \setminus \{(\beta(j), \beta(j+1))\}$$

$$\{(\beta(j), \beta(j+1))\} \leq_{\text{lex}} \{(\beta(i), \beta(i+1))\}$$

Which is equivalent to the second condition of theorem 3.4.

If β is not element stable, then clearly $B = A \uplus \{x\}$ for some x, since $\alpha \leq \gamma$ if $\Gamma = B$. Now we can make a similar deduction as before, since we can only consider $\gamma \sqsubset \beta$ with $B = \Gamma \uplus \{a\}$. Again assume without loss of generality that $x \neq a$. Then we have $\Gamma = A \uplus \{x\} \setminus \{a\}$ and therefore $\{a\} \leq_{\text{lex}} \{x\}$ which is, along with the assertion *γsqsubsetβ*, equivalent to the first condition of theorem 3.4. □

For computing the inverse reduction mapping for a given optimization α, we can now enlist all covers of α and check whether they conform to the conditions in 3.4. With our current knowledge this is however rather inefficient. We might want to only enlist the valid elements directly. For the first condition of theorem 3.4, we need to make sure that when adding a new element x to α, there doesn't exist a larger element a in the resulting β which we could remove to get an element covered by β. Starting from α, this is rather simple. First find the largest $y \in A$ which we could remove from α to get an element covered by α. Then iterate over all elements $x \in N \setminus A$ with $x > y$ and check whether we can add it anywhere into α to get a cover. If we can, we need to check that either the element to the right is smaller than x or that there is an element two positions to the right that is smaller than x. If x is larger than all elements in A, we can simply append x to the end of α.

The second case is a bit more tricky. Here we actually have no other choice than to enlist all covers β and check the element stable condition and the inversion set superiority. If we can however check those two conditions in

constant time, we still only need linear time to find a possibly linear amount of elements.

Let's first consider the inversion set superiority. We can check this condition by comparing the newly introduced inversion $(\alpha(i+1), \alpha(i))$ lexicographically with the largest inversion of α. Recall that the largest inversion (x, y) of α is defined as the inversion for which all $(i, j) \in inv(\alpha), i < x \vee i = x \wedge j < y$. Given an increasingly sorted list $\bar{\alpha}$ of the elements in A (which is also an optimization), we can find x by comparing $\bar{\alpha}$ with α. x is simply the first element of $\bar{\alpha}$, when staring from the back, where α and $\bar{\alpha}$ deviate from each other. y is now the largest element smaller than x that appears after x in α.

The following algorithm finds the largest inversion in α as described above.

Algorithm 2: Algorithm for finding the lexicographic largest inversion in an optimization

```
   Input: α, ᾱ
   Output: (x, y)
 1 if α = ᾱ then
 2 |  return ∅;
 3 end
 4 k ← |A|;
 5 i ← k;
 6 while α(i) = ᾱ do
 7 |  i ← i − 1;
 8 end
 9 x ← ᾱ(i);
10 y ← 1;
11 for j ∈ N_k do
12 |  if j > i ∧ α(j) < x ∧ α(j) > y then
13 |  |  y ← α(j);
14 |  end
15 end
```

The above algorithm has a complexity of $\mathcal{O}(k)$ and must only be run once when computing the inverse reduction mapping of some optimization.

Now let's take a look at the element stability property. There are basically two cases we need to consider. First, given that the current optimization α is element stable, we can check whether the given transposition $(i, i+1)$ breaks this property. In other words, we need to check that the transposition does not introduce any rising sequences of length 3. Note that we do not need to check the special conditions at the start and the end of α, because

the transposition is not allowed to switch $\alpha(1)$ and $\alpha(2)$ or $\alpha(k-1)$ and $\alpha(k)$ due to the definition of the cover relation. The sequence of length 3 is exactly then introduced when $\alpha(i-2) < \alpha(i-1) < \alpha(i+1)$ or $\alpha(i) < \alpha(i+2) < \alpha(i+3)$, which we can naturally check in constant time.

The second case we need to consider is when α is not element stable, but the transposition makes it element stable. This can only happen if α contains exactly one rising sequence of length 3 or 4. If it contains a rising sequence of length 4, the transposition needs to switch the middle two elements of the sequence. With a rising sequence of length 3, let j be the start index of the sequence. Now the transposition must either satisfy $i = j$ or $i = j+1$, and it cannot introduce a new rising sequence of length 3, for which we can apply the check from above. There are a few more edge-cases we need to consider, because here we cannot assume that the start and end are descending. If there is no rising sequence of length 3 or 4 and either $\alpha(0) < \alpha(1)$ or $\alpha(k-1) < \alpha(k)$ but not both, we can swap $\alpha(0)$ and $\alpha(1)$ or $\alpha(k-1)$ and $\alpha(k)$ respectively. But if $k = 3$, we additionally need to make sure that $\alpha(0) > \alpha(2)$. If the rising sequence of length 3 starts at index 1 we also need to make sure that $\alpha(2) < \alpha(0)$ and if it ends at index $k-1$ we need to check that $\alpha(k) < \alpha(k-2)$.

We can clearly compute the number of rising sequences of length 3 and 4 in linear time for a given optimization, which we only need to do once when computing the inverse reduction mapping.

3.4 Minimizing the Fourier-Sparse Function

We now have an algorithm for finding Fourier-sparse estimates of some function over the optimization domain. This is by itself already quite neat, since we now have a function which we can analyze to better understand the impact of different optimizations on some scalar property, such as the runtime. In our case we want to find the optimization which minimizes the function.

Given a vector of non-zero Fourier coefficients w, belonging to the elements of $\mathcal{G} \subseteq \mathcal{P}$, recall that we can compute the function value $f(x)$ for some $x \in \mathcal{P}$ as follows.

$$f(x) = \Phi_{x,:}w = \sum_{g \in \mathcal{G}} \iota_{\{g \preceq x\}} w_g = \sum_{\substack{g \in \mathcal{G} \\ g \preceq x}} w_g$$

This formulation looks very similar to the formulation in equation 3.2 on page 17. Indeed, if we would replace $g \preceq x$ with $x \preceq g$, we would have the exact same problem to solve and could use the same methods for finding the minimum as we have for finding the maximum correlation. The only

thing holding us currently back is our definition of the partial relation. But we can just use the dual order of $\preceq$ instead.

Definition 3.5 (Dual Order) *The* dual *order $\preceq^{-1}$ of the partial order $\preceq$ is defined as the partial order for which for any $x, y \in \mathcal{P}$*

$$x \preceq^{-1} y :\Leftrightarrow y \preceq x$$

Note that for notational simplicity we have already implicitly used the dual order rather frequently, since $\succeq$ is equivalent to $\preceq^{-1}$.

With the dual order we can now write our minimization objective as a maximization objective similar to 3.2 on page 17.

$$f'(x) = \sum_{\substack{g \in \mathcal{G} \\ x \preceq^{-1} g}} -w_g$$

So we can now derive the covering relation for the dual order and proceed by applying the very same techniques as for computing the maximum correlation. Note that we fulfill the constraint of $\preceq^{-1}$ being a substructure poset, since $\preceq$ has a unique maximum, which is the optimization containing all elements in reverse order.

We will proceed more swiftly with the remaining derivations, since it follows the same basic scheme as for the correlation maximization.

Proposition 3.6 (Dual Covering Relation) *Let $\alpha \preceq^{-1} \beta$ be two optimizations in $\mathcal{P}_n$. β covers α iff either*

1. *$B = A$ and $\beta = (i, i+1) \circ \alpha$ where $i \in N_{n-1}$*
2. *$B = A \setminus \{x\}$ with $x \in A$ and $\alpha_B = \beta$ and $\alpha(i-1) < x$ if $i > 1$ and $\alpha(i+1) > x$ if $i < n$ with $i = \alpha^{-1}(x)$*

We'll use the dual $\leq^{-1}$ of the total order $\leq$ for defining the inverse reduction mapping on the dual poset of optimizations.

Let $C(\alpha)$ be the set of elements which can be added to A to get a cover of α in the optimization poset.

$$C(\alpha) = \{x \in N \setminus \Pi.\ \exists \beta \in \mathcal{P}.\ \alpha \sqsubset \beta \wedge B = A + \{x\}\}$$

Additionally let $inv(\alpha)^C$ be the complement of the inversion set.

$$inv(\alpha)^C = \{(i, j) \in N^2.\ i > j \wedge \alpha^{-1}(i) > \alpha^{-1}(j)\}$$

Theorem 3.7 (Inverse Reduction Mapping on the Dual Poset) *For a given optimization $\alpha \in \mathcal{P}$ the inverse reduction mapping on the dual poset $g^{-1}(\alpha)$ contains exactly those elements $\beta \sqsubset \alpha$ for which the following conditions hold.*

- *If $B = A \setminus \{x\}$ with $x \in A$, then x is larger than all elements in $C(\alpha)$*
- *If $B = A$ with $\beta = (i, i+1) \circ \alpha$, then $C(\beta) = \emptyset$ and for all $(j,k) \in inv(\alpha)^C$, $(j,k) <_{lex} (\alpha(i), \alpha(i+1))$*

Note that we can compute the largest element in $C(\alpha)$ in linear time from the definition of the covering relation on the optimization poset. The lexicographic largest element in $inv(\alpha)^C$ can be computed in linear time as well by slightly modifying algorithm 2 on page 21. For computing whether $C((i,i+1) \circ \alpha) = \emptyset$ for $\alpha(i) > \alpha(i+1)$, the following proposition comes in handy.

Proposition 3.8 *Let $\alpha \in \mathcal{P}$ and $\alpha(i) > \alpha(i+1)$ for some i. Then*

$$C((i,i+1) \circ \alpha) = \emptyset \Leftrightarrow C(\alpha) = \emptyset \wedge \{x \in N.\ \alpha(i+1) < x < \alpha(i)\} \subseteq A$$

3.5 Speeding up the Enumeration

In the reverse search algorithm, we compute the correlation or the function value in every iteration, of which there are possibly exponentially many. Such a computation of the correlation or the function value mainly consists of evaluating the partial order for two optimizations. So clearly the evaluation of $\preceq$ will become a performance bottleneck very quickly. Unfortunately, the naive way of evaluating our partial order includes computing and comparing inversion sets which contain $\mathcal{O}(n^2)$ elements in general. Fortunately we can be smarter and don't need to explicitly compute the inversion sets. Let's say we have two permutations α and β with $inv(\alpha) \subseteq inv(\beta)$. From the weak order of permutations we know that we can reach β from α by a series of transpositions for neighboring elements, each increasing the inversion count (the number of elements in the inversion set) by one. The idea now is to use some distance measure $K(\alpha, \beta)$ which counts the minimum number of transpositions required to reach β from α, and then conclude that $inv(\alpha) \subseteq inv(\beta)$ exactly then when $|\,inv(\alpha)| + K(\alpha,\beta) = |\,inv(\beta)|$.

A commonly used distance measure for permutations, which we can also use for our purpose, is called *Kendall tau distance* [14].

For notational brevity, let i and j in the following be elements of N.

Definition 3.9 (Kendall tau distance) *For two permutations α and β, we define the Kendall tau distance as*

$$\begin{aligned} K(\alpha,\beta) := |\{(i,j) : i > j.\quad & \alpha(i) < \alpha(j) \wedge \beta(i) > \beta(j) \vee \\ & \alpha(i) > \alpha(j) \wedge \beta(i) < \beta(j)\}| \end{aligned}$$

The Kendall tau distance works better together the place-based definition of the inversion set, because it compares the inversions between the two

permutations at fixed places. The place-based definition of the inversion set is however equivalent to the inversion set of the inverse permutation $inv(\pi^{-1})$ for the element-based definition. Note that the inversion count is independent of the choice between the two definitions of the inversion set:

$$|\, inv(\pi)| = |\, inv(\pi^{-1})| \tag{3.3}$$

Theorem 3.10 *Let I be the identity permutation, α and β some arbitrary permutations. Then the following assertions are equivalent:*

$$inv(\alpha^{-1}) \subseteq inv(\beta^{-1}) \tag{3.4}$$

$$K(I,\alpha) + K(\alpha,\beta) = K(I,\beta) \tag{3.5}$$

$$|\, inv(\alpha)| + K(\alpha,\beta) = |\, inv(\beta)| \tag{3.6}$$

Proof First, let's note that

$$inv(\pi^{-1}) = \{(i,j) \in \Pi^2.\ i > j \wedge \pi(i) < \pi(j)\} \tag{3.7}$$

For proving the equivalence between 3.5 and 3.6, note that $I(i) > I(j)$ never holds. It follows that

$$K(I,\pi) = |\, inv(\pi^{-1})| = |\, inv(\pi)|$$

Now let's focus on the proof between 3.4 and 3.6. Since the disjunction in the definition of the Kendall tau distance is exclusive, we can write K as follows.

$$K(\alpha,\beta) = |\, inv(\alpha^{-1}) \setminus inv(\beta^{-1})| + |\, inv(\beta^{-1}) \setminus inv(\alpha^{-1})|$$

We'll now prove the equivalence by proving the implication in both directions separately.

$\Rightarrow$: Assume $inv(\alpha^{-1}) \subseteq inv(\beta^{-1})$. Since $inv(\alpha^{-1}) \setminus inv(\beta^{-1}) = \emptyset$, we have

$$K(\alpha,\beta) = |\, inv(\beta^{-1}) \setminus inv(\alpha^{-1})|$$

Now we can directly conclude

$$\begin{aligned} inv(\alpha^{-1}) \subseteq inv(\beta^{-1}) &\Leftrightarrow inv(\beta^{-1}) = inv(\alpha^{-1}) \uplus (inv(\beta^{-1}) \setminus inv(\alpha^{-1})) \\ &\Rightarrow |\, inv(\beta^{-1})| = |\, inv(\alpha^{-1})| + |\, inv(\beta^{-1}) \setminus inv(\alpha^{-1})| \\ &\Rightarrow |\, inv(\beta^{-1})| = |\, inv(\alpha^{-1})| + K(\alpha^{-1},\beta^{-1}) \end{aligned}$$

$\Leftarrow$: We prove the implication by contrapositive. Assuming $inv(\alpha^{-1}) \not\subseteq inv(\beta^{-1})$ we'll show that $K(\alpha,\beta) \neq |\, inv(\beta^{-1})| - |\, inv(\alpha^{-1})|$.

From our assumption we know that $inv(\alpha^{-1}) \setminus inv(\beta^{-1}) \neq \emptyset$. Hence

$$\begin{aligned} K(\alpha, \beta) &= |\, inv(\beta^{-1}) \setminus inv(\alpha^{-1})| + |\, inv(\alpha^{-1}) \setminus inv(\beta^{-1})| \\ &\overset{!}{>} |\, inv(\beta^{-1}) \setminus inv(\alpha^{-1})| \\ &\geq |\, inv(\beta^{-1})| - |\, inv(\alpha^{-1})| \end{aligned}$$

□

Computing the inversion count of a permutation is a classic algorithmic problem and can be solved with a basic algorithm based on merge sort in $\mathcal{O}(n \log n)$. [7] proposes an algorithm which achieves $\mathcal{O}(n\sqrt{\log n})$ time complexity. Since our ns are however relatively small (up to 61 for our experiments), the introduced overhead of this algorithm due to the added complexity in its implementation dominates the algorithmic runtime improvements.

Now considering the computation of the Kendall tau distance, we can make the following derivation.

$$\begin{aligned} K(\alpha, \beta) &\overset{*}{=} |\{(i,j) : \alpha^{-1}(i) > \alpha^{-1}(j). \quad && i < j \wedge \beta(\alpha^{-1}(i)) > \beta(\alpha^{-1}(j)) \vee \\ & && i > j \wedge \beta(\alpha^{-1}(i)) < \beta(\alpha^{-1}(j))\}| \\ &\overset{**}{=} |\{(i,j) : i > j. && i < j \wedge \beta(\alpha^{-1}(i)) > \beta(\alpha^{-1}(j)) \vee \\ & && i > j \wedge \beta(\alpha^{-1}(i)) < \beta(\alpha^{-1}(j))\}| \\ &= |\, inv(\beta \circ \alpha^{-1})| \end{aligned}$$

The equivalence at $(*)$ works because for the cardinality, the exact content of the set does not matter, and the equivalence at $(**)$ works because the order in which we traverse N does not matter.

To summarize, we were able to reduce the subset comparison between the inversion sets to a comparison between inversion counts, which we can compute in $\mathcal{O}(n \log n)$ time.

3.6 Selecting the Training Set

The selection of the set of optimizations $\mathcal{X}$, for which we measure the runtime of our target program, should be expected to have a great impact on the success of our final optimization. For example, if there is a larger selection of optimization flags, which only impact the performance if they are in some unique order, we can easily miss the impact of this combination if we don't choose $\mathcal{X}$ properly. While a deeper analysis of sampling methods is out of scope for this thesis, we will present two basic random methods for selecting $\mathcal{X}$.

A naive way of sampling a single optimization randomly can be done in a straightforward fashion. We can simply choose some $k \in \{0, \ldots, n\}$ uniformly at random and then sample k individual elements from N. The probability of selecting a specific optimization of length k is then

$$\frac{1}{n+1} * \frac{(n-k)!}{n!}$$

With this method, an individual optimization with a low cardinality has a higher probability of being sampled than an optimization with a high cardinality. This might lead to imbalances during training on a training set sampled with this strategy.

As an alternative, we might want to consider sampling uniformly at random. In this case an individual optimization x would have a probability of being sampled of

$$P(x) = \frac{1}{\sum_{m=0}^{n} \frac{n!}{m!}} \tag{3.8}$$

and a probability of having cardinality k of

$$P(|x| = k) = p_k = \frac{\frac{n!}{(n-k)!}}{\sum_{m=0}^{n} \frac{n!}{m!}} = \frac{1}{(n-k)! \sum_{m=0}^{n} \frac{1}{m!}} = \frac{p_n}{(n-k)!} \tag{3.9}$$

We could now sample k according to p_k and otherwise use the naive strategy as above to sample an optimization uniformly at random. This method would however lead to overflows in most common programming languages when calculating p_k for large n and is therefore not ideal either.

A better strategy might be to iterate over the different k from n to 0 and deciding each step whether to choose the current k or not according to some probability q_k. This way we don't need to calculate p_k directly for sampling k. Instead, by choosing q_k correctly, we can still sample k according to p_k. Let's define

$$q_k = P(|x| = k \mid |x| < k+1) \tag{3.10}$$

Using Bayes' theorem, we get

$$q_k = \frac{\overbrace{P(|x| < k+1 \mid |x| = k)}^{1} * P(|x| = k)}{P(|x| < k+1)} = \frac{p_k}{\sum_{m=0}^{k} p_m} \tag{3.11}$$

The following lemma shows that our definition of q_k is indeed correct.

Lemma 3.11 *For all $k \in \{0, \ldots, n\}$:*

$$p_k = \begin{cases} q_n & \text{if } k = n \\ q_k * \prod_{m=k+1}^{n} (1 - q_m) & \text{if } k < n \end{cases} \tag{3.12}$$

Proof First note that the case $k = n$ in 3.12 on the preceding page follows directly from 3.10 on the previous page, since $|x| < n + 1$ is always true.

For $k < n$, let's rewrite p_k using 3.11 on the preceding page:

$$p_k = q_k * P(|x| < k + 1)$$

What's left to prove is that $P(|x| < k + 1) = \prod_{m=k+1}^{n}(1 - q_m)$ for all $k = \{0, \dots, n - 1\}$. We do this by induction over k.

- *Base case ($k = n - 1$):*

$$P(|x| < n) = 1 - P(|x| = n) = 1 - q_n \tag{3.13}$$

- *Step case:*

$$\begin{aligned} P(|x| < k) &= P(|x| < k \mid |x| < k + 1) * P(|x| < k + 1) \\ &= (1 - P(|x| = k \mid |x| < k + 1)) * P(|x| < k + 1) \\ &= (1 - q_k) * P(|x| < k + 1) \\ &\overset{\text{IH}}{=} (1 - q_k) * \prod_{m=k+1}^{n} 1 - q_m \\ &= \prod_{m=k}^{n} 1 - q_m \end{aligned}$$

□

For our new strategy to be effective, we should have a simple recursive formula for q_k and should be able to calculate q_n even for large n. Let's start with the latter. Remember that $q_n = p_n$ and hence

$$q_n = \frac{1}{\sum_{m=0}^{n} \frac{1}{m!}} \tag{3.14}$$

For small n, we can calculate this value directly using the above formula. For large n, we can approximate q_n with $\frac{1}{e}$, where e is Euler's number, since

$$\sum_{m=0}^{\infty} \frac{1}{m!} = e$$

Note the small error of the denominator of this approximation:

$$e - \sum_{m=0}^{n} \frac{1}{m!} \in \mathcal{O}((n+1)!^{-1})$$

To find a recursive formula for q_k, let's first find a recursive formula for p_k.

$$\begin{aligned} p_{k-1} &\overset{3.9}{=} \frac{p_n}{(n-k+1)!} \\ &= \frac{p_n}{(n-k+1)(n-k)!} \\ &\overset{3.9}{=} \frac{p_k}{n-k+1} \end{aligned}$$

Now we get a nice recursive formula for q_{k-1}^{-1}:

$$\begin{aligned} q_{k-1}^{-1} &= \frac{\sum_{m=0}^{k-1} p_m}{p_{k-1}} \\ &= (n-k+1)\frac{\sum_{m=0}^{k-1} p_m}{p_k} \\ &= (n-k+1)\frac{\sum_{m=0}^{k} p_m - p_k}{p_k} \\ &= (n-k+1)(q_k^{-1} - 1) \end{aligned} \tag{3.15}$$

We summarize the above results within algorithm 3

Algorithm 3: Uniform Sampling of Optimizations

```
   Input: n ≥ 0
1  k ← n;
2  if n > 15 then
3  |  q_k^{-1} ← e;
4  end
5  else
6  |  q_k^{-1} ← Σ_{m=0}^{n} 1/m!;
7  end
8  while k > 0 do
9  |  if rand(1/q_k^{-1}) then
10 |  |  break;
11 |  end
12 |  else
13 |  |  q_k^{-1} ← (n − k + 1)(q_k^{-1} − 1);
14 |  |  k ← k − 1
15 |  end
16 end
17 return randPerm(n, k);
```

In algorithm 3, rand(p) returns true with probability p and randPerm(n,k) returns a random k-permutation.

The recursive expression 3 unfortunately still looks like a factorial when unrolling it. So q_k^{-1} might still explode and might become too large to be represented easily. Luckily it turns out that with our setup, everything works out fine.

Lemma 3.12 *The sequence $q_0^{-1}, \ldots, q_n^{-1}$ is strictly monotonically increasing:*

$$\forall n > 0, k \in \{1, \ldots, n\}. \quad q_{k-1}^{-1} < q_k^{-1}$$

Proof (By induction over *k*) Reformulation of 3.15 on the preceding page gives us

$$q_k^{-1} = \frac{q_{k-1}^{-1}}{n-k+1} + 1$$

- *Base case ($k = 1$):*

$$q_1^{-1} = \frac{q_0^{-1}}{n} + 1 = \frac{1}{n} + 1 > 1 = q_0^{-1}$$

- *Step case ($k \in \{1, \ldots, n-1\}$):*

$$q_k^{-1} = \frac{q_{k-1}^{-1}}{n-k+1} + 1 \overset{\text{IH}}{<} \frac{q_k^{-1}}{n-k+1} + 1 < \frac{q_k^{-1}}{n-k} + 1 = q_{k+1}^{-1} \qquad \square$$

Theorem 3.13 *For all $n > 0$, $k \in \{0, \ldots, n\}$ we have $1 \leq q_k^{-1} \leq e$.*

Proof From lemma 3.12 on the previous page we know that q_k^{-1} is monotonically increasing and therefore $q_0^{-1} = 1$ is the minimum and q_n^{-1} the maximum of the sequence. Furthermore we know from 3.14 on page 28 that

$$q_n^{-1} = \sum_{m=0}^{n} \frac{1}{m!} \leq \sum_{m=0}^{\infty} \frac{1}{m!} = e \qquad \square$$

Chapter 4

Approximate Poset Enumeration

Currently, the enumeration of the poset is the main runtime bottleneck in our algorithm. Solving the enumeration problem with our reverse-search algorithm quickly becomes infeasible, as we have discovered in our experiments. A method for tackling this problem is to abstain from finding the exact maximum. In this chapter we will present two methods for approximate optimization on our poset.

4.1 Delta-Approximate Enumeration

The first method we'll show is a very simple modification of our exact enumeration method, similar to the relaxation of the pruning condition in [26]. Recall that the condition by which we prune subtrees during enumeration is

$$\sum_{\substack{x \in \mathcal{X} \\ g^* \preceq x}} r_x \geq \mu(g; r)$$

for the currently considered optimization g and the best already encountered optimization $g*$.

Remember that μ is an upper bound on the value for g. So intuitively, we might get away well enough by subtracting some small $\delta > 0$ from μ, which would allow us to prune earlier in the tree. The new pruning condition would then become

$$\delta + \sum_{\substack{x \in \mathcal{X} \\ g^* \preceq x}} r_x \geq \mu(g; r)$$

Obviously we then loose the guarantee that we will find the exact maximum. Setting δ properly becomes a question of balancing better runtime with better precision.

4.2 Monte Carlo Tree Search

In recent years Monte Carlo Tree Search (MCTS) has gained a lot of attention due to its successful applications to games such as chess or Go [28]. At its core lies the UCT (upper confidence bounds of trees) algorithm [15], which we'll adapt to our setting for approximately finding the maximum absolute correlation and the maximum function value for our poset. For this chapter we'll primarily consider the maximum absolute correlation problem, but everything can similarly be adapted to the function minimization problem. We'll use [5] as a reference for the following introduction into MCTS, the UCT algorithm and our applications and modifications.

The basic idea of MCTS is to iteratively build a search tree from the current state until some computational budget is reached. Based on the search tree, a decision is made for which action is chosen next. In our setting, the state is an optimization and an action is just some transformation of the current optimization which gives us a new one. The decision tree is built using two policies: the *tree policy* and the *default policy*. The tree policy is responsible for selecting a leaf node from the nodes in the current search tree, from where the search tree should be expanded. It then adds a new child to the retrieved node. The default policy then randomly iterates the underlying decision process starting from the newly created node until an end is reached. The value discovered during this random search is then propagated into the decision-making of the tree policy.

The UCT algorithm uses upper confidence bounds for traversing the tree in the tree policy. A child node v' of the current node v is selected, maximizing the value

$$\mathrm{UCT} = \frac{Q(v')}{N(v')} + 2C_p\sqrt{\frac{2\ln N(v)}{N(v')}} \tag{4.1}$$

where Q indicates the accumulated value per node and N the number of times a node has already been visited. C_p is a constant, typically set to $1/\sqrt{2}$ if $X \in [0,1]$. C_p can be interpreted as a control for the trade-off between exploration and exploitation. The first summand in 4.1 can be understood as the exploitation and the second as the exploration term. The UCT value of unvisited children is understood to be ∞ which forces the tree policy to first fully expand a node before traversing further. This selection is made randomly. The default policy in UCT simply chooses an action uniformly at random and returns the value found at the last node. In our case, a terminal node is an optimization, for which we cannot add any further flags, or the state reached after performing a stop action. This value is then propagated back through the search tree and added to Q for every node encountered earlier in this iteration of the tree policy.

For our application, we use the absolute correlation as the value computed by UCT at the end of the default policy. Since we are interested in finding the maximum absolute correlation, every run of the default policy effectively already works as a query for us. A run of UCT simply returns a modification on the current optimization, for which we expect to find good absolute correlations. For the design of the action space, we are basically free to choose any modifications, as long as we are certain that we can reach every optimization by a feasibly long series of actions from the root element, which we are also free to choose.

If we however assume that we traverse the poset similarly as during enumeration, we could enhance UCT by pruning subtrees with the condition μ, as we did during enumeration. However, for MCTS, we are not bound to follow the cover relation, but can simply traverse the poset, such that $\alpha \preceq \beta$ always holds, where α is the current node and β the next.

For some optimization α, let's choose the action space such that for any next optimization β it holds that $|B| = |A| + 1$ and $\alpha = \beta_A$ and therefore $\alpha \preceq \beta$. In words, our action simply adds a new element at any position in the current optimization, which is easy and efficient to implement. Additionally, we need to add a stop action to every state. The number of actions available for some α is then $(n - |A|)(|A| + 1) + 1$. We'll need to start the algorithm at the empty set, because otherwise we wouldn't be able to encounter every optimization. We can now compute the upper bound μ for the absolute correlation with the residual for every newly added node α in the search tree. If $\mu \leq c$ where c indicates the maximum absolute correlation computed yet, we can prune the subgraph and set $Q(\alpha) = -\infty$.

For the minimization problem, we start at the inverse ordered optimization containing all elements, as we did for enumeration. The action space needs to allow switching and removing elements to be able to reach every optimization. So let's allow two kinds of actions (plus the stop action) for some node α: either a transposition $(i, i + 1)$ where $\alpha(i) > \alpha(i + 1)$ or selecting and removing an element from A. Such a traversal clearly respects the order of $\preceq^{-1}$, and we can again apply pruning similar to the maximum absolute correlation problem.

Let's summarize our contributions from this section. We proposed an adaption of the UCT algorithm for approximately enumerating our optimization poset. By respecting the partial order during traversal, we were able to guide the search by pruning subgraphs using the same technique as for the complete enumeration.

Chapter 5

Implementation and Experimental Setup

The implementation includes two major parts. The first part implements a runtime measurement setup, while the second part implements the ML algorithms. Most of the codebase is written in Python, while the performance critical parts, which are the enumeration and the UCT algorithms, are written in C++ and bridged to Python with pybind11 [13].

5.1 Setup and Usage

We provide the python module `compiler_opt` for interacting with our codebase, which can be found at the project root directory. A readme file can be found in the project root directory containing installation and usage instructions. There are multiple submodules, each providing its own functionalities.

env The `compiler_opt.env` submodule implements the compiler setup and the measurement setup. The compiler setup implements an interface for compiling source files for a given optimization. It is responsible for managing the available flags and different compilation phases. The measurement setup provides an interface for measuring runtimes for a fixed compiler setup.

cbench, polybench The submodule `compiler_opt.cbench` and the submodule `compiler_opt.polybench` implement a measurement setup for the cBench and the polybench benchmark suites respectively.

data The `compiler_opt.data` submodule is responsible for managing (creating storing and loading) datasets for a given measurement setup. It also

provides functions for randomly sampling query points.

estimators The `compiler_opt.estimators` submodule implements an sklearn interface for our Lasso regression algorithm.

experiments The `compiler_opt.experiments` submodule provides multiple functions for fitting and minimizing Fourier-sparse poset functions.

measure The `compiler_opt.measure` submodule is responsible for managing full dataset suites. It provides a simple interface for creating new dataset suites and adding and removing benchmarks and datasets to the suite. A simple implementation of the visitor pattern is provided for analyzing the benchmarks over a complete suite.

evaluate The `compiler_opt.evaluate` submodule provides an interface for collecting solutions and baselines of the compiler optimization problem into result objects. Furthermore it provides function for analyzing those results over a full datasuite.

5.1.1 Example Usage

The following example code creates a new dataset suite for all supported cBench benchmarks and fills it with measurements for $n = 30$ and 1000 datapoints. It then fits and minimizes every dataset with MCTS and stores the results in the suite and finally plots them.

```
import compiler_opt as co

setup = co.cbench.default_setup
setup.compiler = co.env.clang
setup.compiler.load_flags(\
    os.path.join(\
        co.env.proj_root,\
        "flag_lists",\
        "llvm10_no_reps_sorted.txt"))
setup.set_iterations(5)
setup.set_subset(list(range(30)))
setup.tag = "30"
co.measure.fill_dataset_suite(\
    setup,\
    [co.get_random_permutation(1000, setup.n)],\
    co.cbench.benchmarks)
co.experiments.run_evaluation_on_suite(\
    name="30", approx_pattern="30")
```

```
co.evaluate.visualize_suite(names="30")
```

5.2 Runtime Measurement Setup

An implementation of a measurement framework supporting two of commonly used benchmark suites for compiler flag optimizations, polybench [23] and cBench from the collective knowledge framework [11], is provided. The framework has built-in support for both GCC and LLVM / clang [18], while for GCC, only the selection problem is supported, since it does not allow defining a custom order for its optimization flags. Therefore, all of our experiments were performed with LLVM.

Optimization in LLVM happens mainly in the middle-end on LLVM intermediate representation (IR) by a tool called *opt*. We can compile a collection of source files to LLVM IR using clang and llvm-link with the following commands

```
clang -emit-llvm -c \
    -O3 -Xclang -disable-llvm-passes sources.c
llvm-link *.bc -o linked.bc
```

The `-O3` flag enables some (minor) front-end optimizations of clang, which we will not customize, and allows opt to apply its optimizations. With `-Xclang -disable-llvm-passes`, we tell clang to not run opt itself. We can then use opt with our custom optimization flags (e.g. `-loop-unroll`) as follows

```
opt -loop-unroll linked.bc -o optimized.bc
```

For LLVM 13 and above, we additionally need to turn off the new pass manager with `-enable-new-pm=0`, which our implementation does automatically. Since using the legacy pass manager is deprecated since LLVM 13, there is also support for the new pass manager in our implementation. However, since we extract our flags from `-O3` for the legacy pass manager, and since related work mainly uses the legacy pass manager, we used it as well.

After optimization, we can use *llc* to compile to assembly and afterwards clang again to create an executable.

```
llc optimized.bc -o out.s
clang out.s -o a.out
```

LLVM differentiates between transform and analysis passes. Analysis passes analyze the code but do not modify it. The results of analysis passes are then used by transform passes, which, as their name suggests, actively transform

the source code. The pass manager automatically puts required analysis passes before transform passes. However, there might be multiple implementations of the same analysis, which might fit better in some circumstances than the default choice made by the pass manager. So selecting the right analysis passes for a transform pass is a challenge by itself. We mitigate this challenge by extracting some knowledge from -O3. Our implementation can automatically extract the pass pipeline used by -O3 and distinguish between analysis and transform passes. It then creates a list of flag combinations for us to be used, where every flag combination is a transform pass appended to a list of analysis passes. Every transform pass encountered in -O3 is added to this list with all the analysis passes which appear before it in -O3 prepended. This gives us a total of 99 transform passes, some of which appear multiple times. We can reduce the number of flag combinations by removing the duplicate transform passes, which results in 61 individual combinations. For further reducing of the number of combinations we need to consider, we sorted the flag combinations according to their individual runtime improvements for some benchmark. We could then use the best n flag combinations to test our algorithms.

For our experimental setup, we chose a collection of 24 benchmarks from cBench, which ranges over multiple different application domains. The exact selection can be found in appendix A.3. We use the provided scripts of cBench to check the output of a run of the benchmark. If the output differs from the reference during measurements, we impose a large penalty on the given optimization to force our algorithms to avoid similar optimizations during minimization.

To ensure stable measurements, cBench automatically performs multiple iterations for one run of a benchmark. The number of iterations depends on the benchmark, while one run is targeted to take roughly a second. We additionally perform multiple runs per benchmark to further stabilize the measurements and to measure the variance such that we can detect unstable measurement environments.

CBench also provides an infrastructure to check the output after a run. This allows us to check that after applying an optimization, the program still produces correct outputs. We use this infrastructure and impose a large penalty on the runtime for optimizations which fail the check. This motivates our algorithms to refrain from outputting flag combinations which produce faulty programs.

5.3 Machine Learning Algorithms

The implementation of the reverse search algorithm as well as the approximate methods are done in C++. The implementation of UCT and the im-

plementation of reverse search are generic algorithms implemented with template programming. This allows for a more flexible experimental setup, since it enables switching posets easily without a complete reimplementation of the algorithms. While the modular implementation sacrifices some performance, we still didn't neglect performance completely. The algorithms are parallelized, and often we prefer managing memory ourselves instead of using abstract datastructures from the standard library.

For Lasso regression with coordinate descent, we use the ElasticNet implementation provided by sklearn [22]. We split our datasets into a training, a test and a validation set. The training set consists of 80% of the dataset while the test and validation sets both consist of 10%. Those splits are generated randomly for every experiment conducted. We use the validation set for early stopping on the number of non-zero frequencies. Early stopping is implemented so that it records the frequencies and coefficients for the best validation score, which is evaluated every 5 iterations in our experiments. If no new best validation score is found for 20 more evaluations, the algorithm is stopped and the best configuration is returned. The measurements are normalized to have zero mean and unit variance and the hyperparameter λ of Lasso is set to 0.001. For the UCT algorithm, we use a computational budget of $100'000$, where the computational budget indicates the number of runs of the tree policy, before a decision is made.

For the complete powerset enumeration, we use the code from [32].

Chapter 6

Results

For the evaluation, we used datasets with 1000 data points per benchmark. Generating such a dataset for a single benchmark takes on average roughly one hour, which means that generating datasets for a fixed configuration for all benchmarks in our selection takes roughly a day. We generated and evaluated datasets for $n = 61$, 30 and 10 flags. For $n = 30$ we additionally generated datasets with a fixed flag selection for evaluating the phase-ordering problem. We did the same for the selection problem with $n = 61$, where we fixed the order of the LLVM transform passes as they appear in `-O3`.

In our experiments, we soon discovered that the complete enumeration quickly becomes infeasible due to the poset size explosion. So for $n = 30$ and $n = 61$ we completely switched to MCTS for fitting and minimization. Delta-approximate enumeration only allowed us to marginally increase n without imposing a too large δ.

We now present the results of our experiments and provide some interpretation. For every n under consideration, we provide two bar plots. The first plot shows the final runtime for the minimized optimizations along with the runtimes for the different baselines. The second plot shows the coefficient of determination (the R^2 score) for the approximated Fourier-sparse function, evaluated on the test data. We use three baselines to compare the runtime against. The first baseline is the runtime of `-O3`, which we use as our main baseline. All other runtimes are relative to the runtime of `-O3` in the runtime plots. The second baseline is the runtime when using all flag combinations under consideration in reference order (as they appear in `-O3`). The third baseline is the best runtime discovered during measurements when generating the dataset.

Figure 6.1: Runtimes and R^2 scores for $n = 10$.

Figure 6.1 shows the results of our experiments for $n = 10$. Here we were able to evaluate both the complete enumeration ("exact") and MCTS ("approximate"). The score is for both methods rather similar, indicating that, at least for small ns, we don't sacrifice a measurable amount of accuracy when using MCTS. The R^2 scores are high with an average of 0.879042 for the complete enumeration and 0.879067 for the approximate enumeration. The runtime plot shows that we, even with only 10 flags, already reach competitive runtimes to `-O3`, where we beat the runtime of `-O3` with the approximate solution on average by 0.6188% and are 1.6985% worse on average for the exact solution. Remember that the flags were selected based on the individual runtime improvements.

There are two interesting points to note further about figure 6.1. For benchmark 0 (telecom_adpcm_c), the approximate solution performs a lot better than the exact solution. This is most likely not due to the different enumeration methods used. Instead, note that the best measurement is vastly smaller than `-O3`. When looking at the randomly generated dataset splits for both solutions, we can see that the training set for the evaluation of MCTS did include this best optimization, while the training set for the complete enumeration did not. This illustrates our earlier point well that the selection of the query points plays a major role in the success of our algorithms and can be a starting point for future work.

Benchmark 19 (telecom_CRC32) also stands out when considering the score plot. For this benchmark, only very few flag combinations have an impact on the runtime, as we can see from our measurements. Our model is therefore tasked to mostly fit random noise from our measurements, which justifies the R^2 score of about 0.

On average, we find a total of 83 non-zero Fourier coefficients for the approximate enumeration and 72 for the complete enumeration.

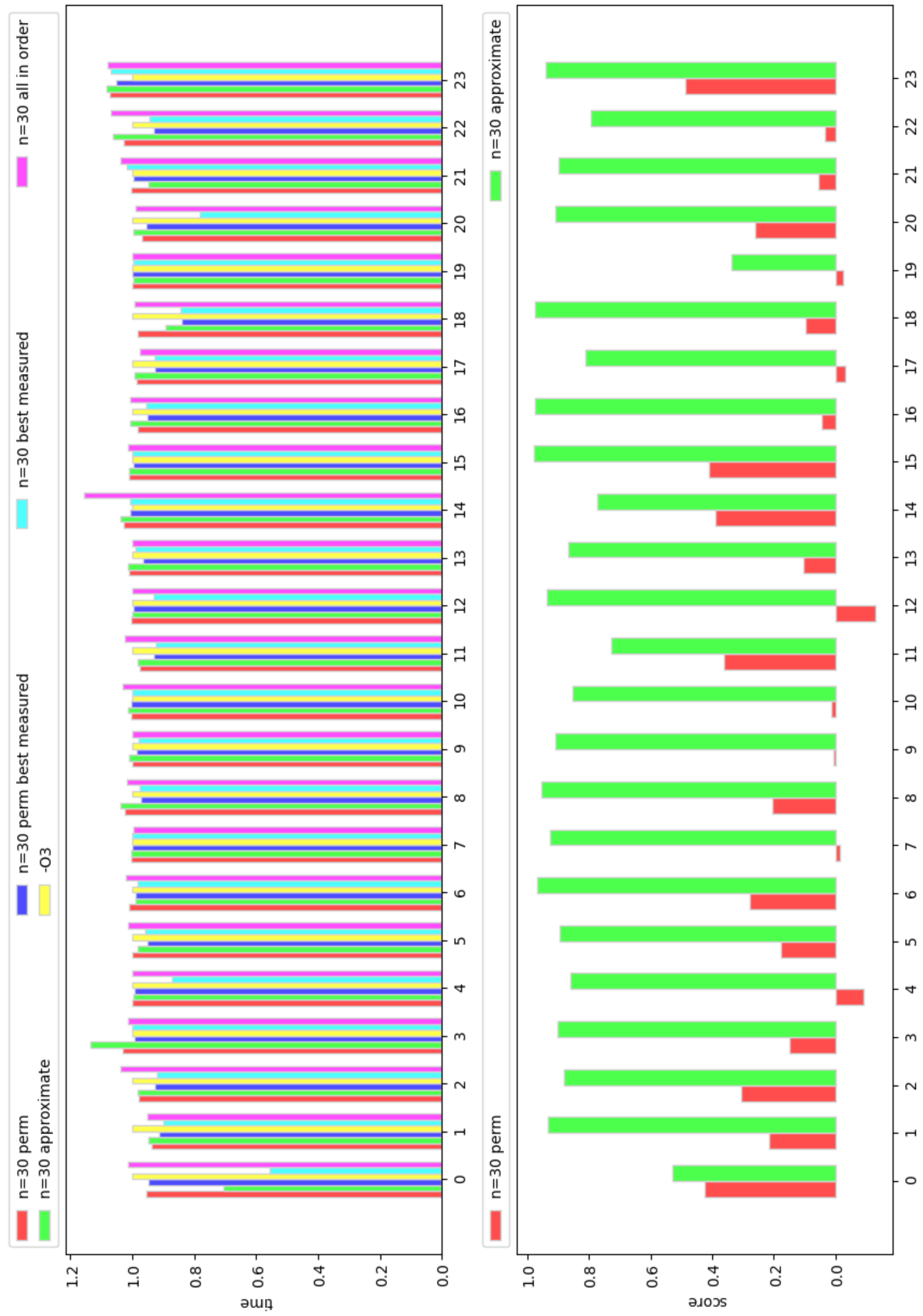

Figure 6.2: Runtimes and R^2 scores for $n = 30$.

Figure 6.2 shows the results for $n = 30$. With this flag selection, we evaluated our approach on the phase-ordering problem as well. In the scores plot we can see that our algorithm has a hard time fitting on the permutation poset. The average R^2 score is only 0.154924 for the permutation poset, while fitting with the full poset gives us an average score of 0.855480. Two explanation come into mind for why fitting on the permutation poset often produces bad scores. First it might simply be more difficult to learn a Fourier-sparse function on the permutation space. Alternatively, the runtime of a program might be rather stable when reordering flags, and isolating the permutation differences which produce runtime differences becomes tricky. In other words, many optimizations are mutually independent. This would again boil down to the problem of selecting a better set of query points. Such a claim can also be supported by our earlier example from figure 1.1, which shows that for this specific benchmark only the order between a few pairs of optimizations is relevant.

For $n = 30$ we already discover a drop in sparsity compared to $n = 10$. For the optimization poset we find an average of 24 non-zero Fourier coefficients and for the permutation poset we even only find 14 non-zero Fourier coefficients. Afterwards early stopping kicks in, showing that we overfit rather early with our training set.

Figure 6.3 shows the results for $n = 61$. Here we also evaluated the enumeration on the powerset. Since the size of the powerset is vastly smaller than the size of our full poset, we are able to fully enumerate instead of using MCTS. However, we still needed to switch to MCTS for minimization.

In this experiment we clearly experience the difficulty of fitting on larger ns. We still reach an average score of 0.682919 for the full poset, but we can also see that the score drops remarkably for some benchmarks. On the other hand, we reach a score of 0.775839 for the powerset and even get better optimizations there. Compared to `-O3` we are 0.3696% faster when sticking to the selection problem with our techniques. When considering the full poset, our runtime is slightly worse than `-O3` by 1.0332% on average. The number of non-zero Fourier coefficients averages at 28 for the full poset and at 24 for the powerset, showing that overfitting becomes even more of a problem. Interestingly enough, we are still able to provide relatively good fits when evaluating on random query points. This might suggest that it is easy to learn a more coarse grained view over the interactions between the different optimization flags, while there are a lot of more hidden interactions which require more care to be discovered.

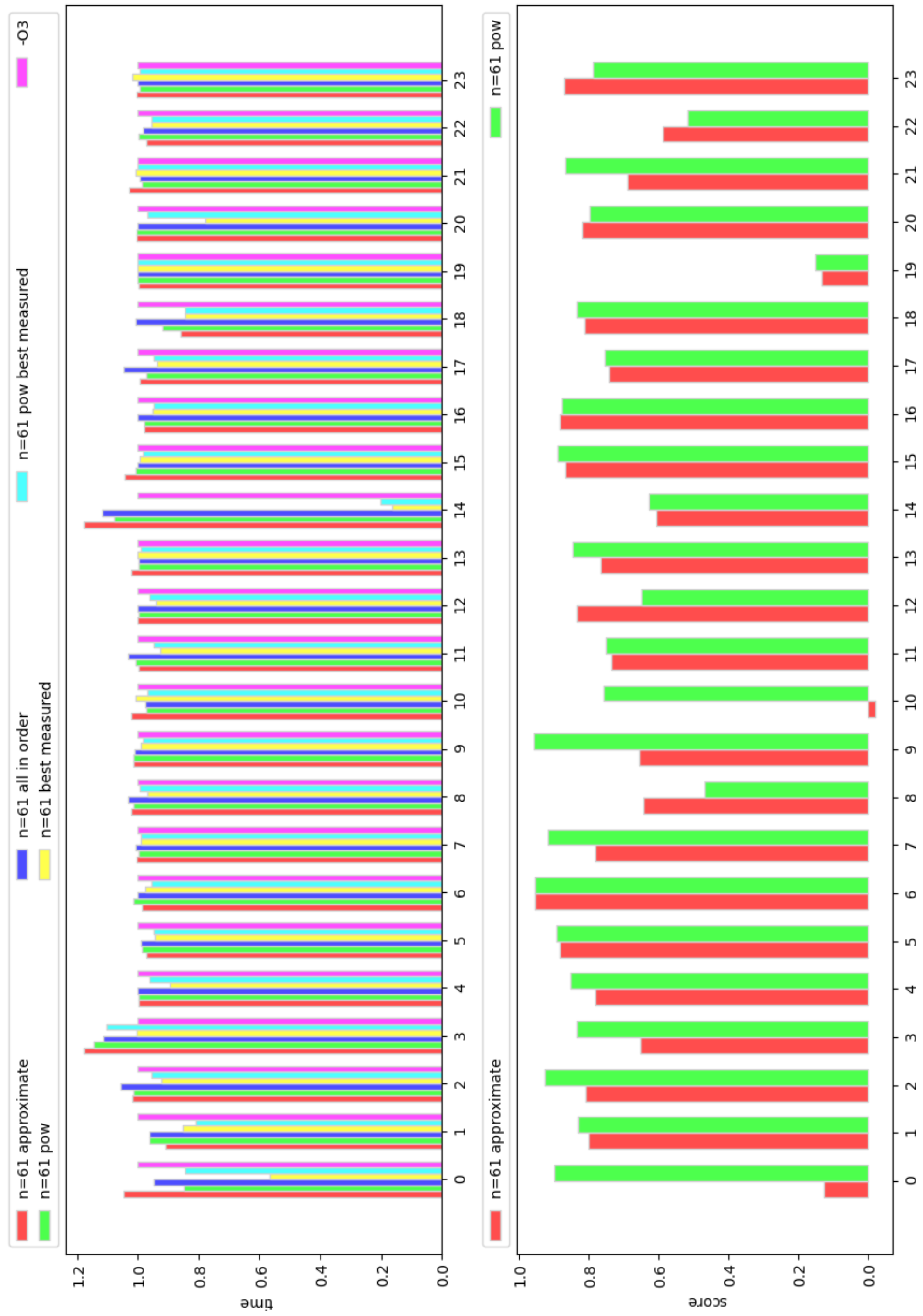

Figure 6.3: Runtimes and R^2 scores for $n = 61$.

Chapter 7

Conclusion and Future Work

We present a novel algorithmic framework for learning and minimizing Fourier-sparse estimations of functions over the domain of partial permutations. Our framework provides a foundation for research on Fourier-sparse approximations applied to the field of compiler flag optimization, where we provide an algorithmic and theoretical basis which can be applied to a very large optimization space. We show that we can apply our algorithms on the problem of finding runtime-optimal optimization flag combinations. Our combinations we generate using this method can compete with the highest optimization level of LLVM even for a small selection of flags. Our algorithms produce good approximations of the underlying real-world poset function when evaluated on randomly sampled unseen query points.

7.1 Limitations and Future Work

In this thesis, we limited our evaluation scope to a fixed dataset with data points randomly sampled from our poset of optimizations. Choosing a better selection of data points might benefit our methodologies, as our results already hint at. One approach might be to employ some active learning techniques, such as Bayesian Optimization, for choosing our query points more carefully.

In practice a low compilation time is integral for any optimization technique to be usable. Most of our related work pays respect to this limiting factor. Since generating our datasets for a program and evaluating them takes relatively long, our methodologies couldn't be applied directly into practice. However, the active learning approach from above might help in this manner, since we would then find ourselves in the domain of iterative compilation methods. Furthermore, our algorithms could be implemented faster by refraining from our generic approach and applying some manual optimization techniques.

Another limiting factor in our design is that we do not directly support flag repetitions in our optimization poset. Furthermore, we do not consider non-boolean flags. We could solve both of those problems by adjusting our function domain to a different poset and otherwise use similar techniques.

Appendix A

Appendix

A.1 Proof of proposition 2.3 (partial order)

Proof Reflexivity:

$$\alpha \preceq \alpha \Rightarrow A \subseteq A \wedge inv(\alpha) \subseteq inv(\alpha) \Rightarrow \top$$

Antisymmetry:

$$\begin{aligned} \alpha \preceq \beta \wedge \beta \preceq \alpha &\Rightarrow A \subseteq B \wedge B \subseteq A \wedge inv(\alpha) \subseteq inv(\beta) \wedge inv(\beta) \subseteq inv(\alpha) \\ &\Rightarrow A = B \wedge inv(\alpha) = inv(\beta) \\ &\Rightarrow \alpha = \beta \end{aligned}$$

Transitivity:

$$\begin{aligned} \alpha \preceq \beta \wedge \beta \preceq \gamma &\Rightarrow A \subseteq B \wedge B \subseteq \Gamma \wedge inv(\alpha) \subseteq inv(\beta) \wedge inv(\beta) \subseteq inv(\gamma) \\ &\Rightarrow A \subseteq \Gamma \wedge inv(\alpha) \subseteq inv(\gamma) \\ &\Rightarrow \alpha \preceq \gamma \end{aligned}$$

□

A.2 Proof of proposition 3.2 (total order)

Proof Reflexivity:

$$\alpha \leq \beta \Rightarrow A \leq_{\text{lex}} A \wedge (A = A \rightarrow inv(\alpha) \leq_{\text{lex}} inv(\alpha)) \Rightarrow \top$$

Antisymmetry:

$$\begin{aligned}
\alpha \leq \beta \wedge \beta \leq \alpha \Rightarrow \quad & A \leq_{\text{lex}} B \wedge (A = B \rightarrow inv(\alpha) \leq_{\text{lex}} inv(\beta)) \wedge \\
& B \leq_{\text{lex}} A \wedge (A = B \rightarrow inv(\beta) \leq_{\text{lex}} inv(\alpha)) \\
\Rightarrow \quad & A = B \wedge (A = B \rightarrow inv(\alpha) \leq_{\text{lex}} inv(\beta)) \wedge \\
& (A = B \rightarrow inv(\beta) \leq_{\text{lex}} inv(\alpha)) \\
\Rightarrow \quad & A = B \wedge inv(\alpha) \leq_{\text{lex}} inv(\beta) \wedge inv(\beta) \leq_{\text{lex}} inv(\alpha) \\
\Rightarrow \quad & A = B \wedge inv(\alpha) = inv(\beta) \\
\Rightarrow \quad & \alpha = \beta
\end{aligned}$$

Transitivity:

$$\begin{aligned}
\alpha \leq \beta \wedge \beta \leq \gamma \Rightarrow \quad & A \leq_{\text{lex}} B \wedge (A = B \rightarrow inv(\alpha) \leq_{\text{lex}} inv(\beta)) \wedge \\
& B \leq_{\text{lex}} \Gamma \wedge (B = \Gamma \rightarrow inv(\beta) \leq_{\text{lex}} inv(\gamma)) \\
\Rightarrow \quad & A \leq_{\text{lex}} \Gamma \wedge (A = B \rightarrow inv(\alpha) \leq_{\text{lex}} inv(\beta)) \wedge \\
& (B = \Gamma \rightarrow inv(\beta) \leq_{\text{lex}} inv(\gamma)) \wedge \\
& (A = \Gamma \rightarrow A = B \wedge B = \Gamma) \\
\Rightarrow \quad & A \leq_{\text{lex}} \Gamma \wedge \\
& (A = \Gamma \rightarrow inv(\alpha) \leq_{\text{lex}} inv(\beta) \wedge inv(\beta) \leq_{\text{lex}} inv(\gamma)) \\
\Rightarrow \quad & A \leq_{\text{lex}} \Gamma \wedge (A = \Gamma \rightarrow inv(\alpha) \leq_{\text{lex}} inv(\gamma)) \\
\Rightarrow \quad & \alpha \leq \gamma
\end{aligned}$$

Strong Connectivity:

$$\begin{aligned}
\alpha \leq \beta \vee \beta \leq \alpha \Rightarrow \quad & A <_{\text{lex}} B \vee A = B \wedge inv(\alpha) \leq_{\text{lex}} inv(\beta) \vee \\
& B <_{\text{lex}} A \vee A = B \wedge inv(\beta) \leq_{\text{lex}} inv(\alpha) \\
\Rightarrow \quad & A <_{\text{lex}} B \vee B <_{\text{lex}} A \vee \\
& A = B \wedge (inv(\alpha) \leq_{\text{lex}} inv(\beta) \vee inv(\beta) \leq_{\text{lex}} inv(\alpha)) \\
\Rightarrow \quad & A <_{\text{lex}} B \vee B <_{\text{lex}} A \vee A = B \wedge \top \\
\Rightarrow \quad & \top
\end{aligned}$$

□

A.3 Selection of Benchmark Programs

0	telecom_adpcm_c	8	consumer_tiffmedian	16	bzip2e
1	automotive_qsort1	9	automotive_susan_s	17	bzip2d
2	consumer_tiff2bw	10	security_rijndael_e	18	automotive_susan_c
3	telecom_gsm	11	consumer_tiff2rgba	19	telecom_CRC32
4	security_blowfish_d	12	security_blowfish_e	20	telecom_adpcm_d
5	consumer_tiffdither	13	consumer_jpeg_d	21	network_dijkstra
6	network_patricia	14	automotive_bitcount	22	consumer_jpeg_c
7	office_rsynth	15	automotive_susan_e	23	consumer_lame

Bibliography

[1] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5), sep 2018.

[2] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. Predictive modeling methodology for compiler phase-ordering. In *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, PARMA-DITAM '16, page 7–12, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, and Cristina Silvano. A bayesian network approach for compiler auto-tuning for embedded processors. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 90–97, 2014.

[4] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space, 1998.

[5] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[6] Agakov Bonilla Cavazos, F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *In Proceedings of the International Symposium on Code Generation and Optimization (CGO*, pages 295–305, 2006.

[7] Timothy M. Chan and Mihai Patrascu. Counting inversions, offline orthogonal range counting, and related problems, 2010.

[8] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209, 2021.

[9] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. *SIGPLAN Not.*, 34(7):1–9, may 1999.

[10] J.W. Davidson, G.S. Tyson, D.B. Whalley, P.A. Kulkarni, J.W. Davidson, G.S. Tyson, D.B. Whalley, and P.A. Kulkarni. Evaluating heuristic optimization phase order search algorithms. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 157–169, 2007.

[11] Grigori Fursin, Anton Lokhmotov, and Ed Plowman. Collective knowledge: Towards r amp;d sustainability. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 864–869, 2016.

[12] Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES '18, page 35–42, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 — seamless operability between c++11 and python, 2016. https://github.com/pybind/pybind11.

[14] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[15] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[16] A. Koseki, H. Komastu, and Y. Fukazawa. A method for estimating optimal unrolling times for nested loops. pages 376–382, 1997. Publisher Copyright: © 1997 IEEE.; 3rd International Symposium on Parallel Architectures, Algorithms, and Networks, I-SPAN 1997 ; Conference date: 18-12-1997 Through 20-12-1997.

[17] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 171–182, New York, NY, USA, 2004. Association for Computing Machinery.

[18] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] S.G. Mallat and Zhifeng Zhang. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.

[20] George Markowsky. Permutation lattices revised. *Mathematical Social Sciences*, 27(1):59–72, 1994.

[21] Sebastian Nowozin. *Learning with structured data : Applications to Computer Vision*. Doctoral thesis, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Berlin, 2009.

[22] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[23] Louis-Noel Pouchet and Tomofumi Yuki. Polybench/c, 2015.

[24] Markus Puschel and JosÉ M. F. Moura. Algebraic signal processing theory: Foundation and 1-d time. *IEEE Transactions on Signal Processing*, 56(8):3572–3585, 2008.

[25] Markus Püschel and Chris Wendler. Discrete signal processing with set functions. *CoRR*, abs/2001.10290, 2020.

[26] Hiroto Saigo, Sebastian Nowozin, Tadashi Kadowaki, Taku Kudo, and Koji Tsuda. Gboost: A mathematical programming approach to graph classification and regression. *Machine Learning*, 75:69–89, 04 2009.

[27] Bastian Seifert, Chris Wendler, and Markus Püschel. Learning fourier-sparse functions on DAGs. In *ICLR2022 Workshop on the Elements of Reasoning: Objects, Structure and Causality*, 2022.

[28] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[29] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society (Series B)*, 58:267–288, 1996.

[30] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215, 2003.

[31] Stephen J. Wright. Coordinate descent algorithms, 2015.

[32] Eliza Wszola. *Machine Learning on Manycore CPUs*. PhD thesis, ETH Zurich, Zurich, 2022.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Compiler Flag Optimization using Fourier-sparse Poset Functions

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Hörmann	Tierry

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
Zürich, 02.05.2022	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.