GPU-Accelerated GOMEA

Solving the max-cut problem by large-scale parallelisation of GOMEA using GPGPU

N. S. Kartoredjo

TU Delft

GPU-Accelerated GOMEA

Solving the max-cut problem by large-scale parallelisation of GOMEA using GPGPU

by

N. S. Kartoredjo

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on June 29, 2023 at 10:00 AM.

Student number:	4271378	
Project duration:	September 1, 2020 – June 29, 2023	
Thesis committee:	Prof. dr. P. A. N. Bosman,	CWI Amsterdam, supervisor
	Anton Bouter, M.Sc.	CWI Amsterdam, daily supervisor
	Prof. dr. ir. C. Vuik,	TU Delft, Committee member

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

With the advances in *General-Purpose computing on Graphics Processing Units* (GPGPU), it is worthwhile to explore whether other areas in the field of *Artificial Intelligence* (AI) can reap the benefits. One such area is *Evolutionary Algorithms* (EAs), which—among other processes—involves the repetitive exchange of genes among individuals. This repetitive nature aligns with our intuition for parallel optimisation, precisely what GPGPU is designed for. Currently, the state-of-the-art approach in EA is known as *Gene-pool Optimal Mixing Evolutionary Algorithm* (GOMEA), which capitalises on the information embedded within the population by computing the *linkage* between genes across the entire population. However, when it comes to parallelising the exchange of complete linkage sets, particularly in the context of our specific problem of interest, the challenge becomes more intricate.

In the case of our problem, known as Max-cut, there are dependencies between genes that must be considered when constructing parallel sets of linkage sets, referred to as *packages*. We propose three solutions: contamination, revision, and association. Contamination fully utilises parallel capabilities but deviates from the concept of linkage sets. Revision constructs the linkage sets as described by GOMEA, but keeps the dependencies between linkage sets within a package untouched. Association on the other hand attempts to resolve the dependencies by generating a dependency graph to create the set of packages.

From our experiments, we can conclude that parallel acceleration using GPGPU is roughly on par with—and sometimes even outperforms—its non-parallelised counterpart. Out of the three solutions, it is evident that association demonstrates the most promising performance profile in terms of approaching the optimal solution. However, the performance falls significantly short of matching the capabilities exhibited by GOMEA. Furthermore, all of the solutions face a significant burden when evaluating the fitness for each exchanged linkage set. An option to consider as an extension to the current setup is known as *partial evaluation*, although the performance exhibited by contamination implies that simplicity could be the key to success. Further exploration of the acceleration process using widely employed parallel operators—such as those found in linear algebra—has the potential to yield valuable insights for enhancing performance.

Preface

I would like to extend my heartfelt gratitude to those who patiently supported me throughout my Master's Thesis Project. In particular, I would like to express my deep appreciation to Anton Bouter, my daily supervisor, and Peter Bosman, my overall supervisor, for their exceptional tolerance towards my delays and their understanding of my personal commitments. Their unwavering patience played a vital role in enabling me to successfully complete my Master's degree while managing my financial responsibilities.

I am profoundly grateful to my significant other for being my constant companion during these challenging times, aiding me in overcoming my skirmishes, and delightfully demolishing my abominable writing style. Her unwavering support has extended beyond the scope of my thesis, and I am grateful to have her by my side in all aspects of life.

I would also like to express my sincere appreciation to my parents, who have provided me with a loving home where I always feel welcomed. Their willingness to keep their backdoors unlocked has allowed me to pay them unexpected midnight visits, raid their refrigerator, and find solace in their comforting beds.

Lastly, I want to express my gratitude to the lads for their unyielding camaraderie, whether it be within the warm embrace of our beloved local pub or while adhering to the necessary 1.5 metres of physical distance.

N. S. Kartoredjo
Delft, June 2023

Contents

Acronyms

Notation	Description	Page List
AI	Artificial Intelligence	iii, 1, 62
CPU	Central Processing Unit	1, 62
CUDA	Compute Unified Device Architecture	3, 15–18
EA	Evolutionary Algorithm	iii, 1–3, 5, 14, 24, 25, 49, 58
FOS	Family of Subsets	7–10, 14, 15, 29–33, 37–39, 41, 43, 59
GA	Genetic Algorithm	3–7
GOM	Gene-Pool Optimal Mixing	8, 30–32
GOMEA	Gene-pool Optimal Mixing Evolutionary Algorithm	iii, 1–3, 13, 31, 37, 38, 40, 48, 49, 54–62
GPGPU	General-Purpose computing on Graphics Processing Units	iii, 1, 3, 15, 32, 41
GPU	Graphics Processing Unit	1–3, 15, 18, 23, 31–33, 38, 57–59, 61, 62
ILG	Independent Linkage Graph	40–42, 61
MP	Marginal Product	7, 15, 29, 30, 32, 43
UPGMA	Unweighted Pair Group Method with Arithmetic mean	3, 22–24, 37, 38, 42, 43, 59
XOR	Exclusive OR	6

Symbols

Notation	Description	Page List
C	Colour vector	25, 27, 28
D	Donor matrix	32, 34, 35
E	Edges	25, 28, 34, 40, 41, 43
G	Graph	25, 28, 40, 41, 46
H	Entropy function	21
I	Individual, or genotype	4–6, 8, 20, 25, 30, 46, 47
N	Frequency matrix	20, 21
O	Offspring, or children	4, 5, 31, 35
P	Population, or parents	4, 5, 9, 20, 31, 33–35
Q	Action set	11, 13
R	Randomised matrix	32, 33
V	Vertices	5, 10, 11, 25, 28, 29, 40, 41
W	Weights	22, 28, 43
Y	Intermediate matrix	34, 35
Φ	Fitness vector	5, 9, 31, 33, 35
Π	Partition	25, 26, 28, 30, 47
Σ	Similarity matrix	20–24
α	Side effect	12
δ	Decoder function	14, 15, 27
ℓ	Length of a genotype	4, 5, 7, 9, 10, 20, 21, 32–34, 36, 38, 40, 41, 48, 49, 59

Notation	Description	Page List
κ	Kernel, or mapping function between states	12, 14, 17–19, 27
$\mathcal{F}$	*Family of Subsets* (FOS)	7–9, 15, 30, 31, 43
$\mathcal{P}$	package	30, 33, 34, 38, 39
$\mathcal{p}$	package item, or linkage set index	30, 32–35, 38, 40, 42
s	Stride	18, 19
μ	Variable	37, 39, 40, 43, 44, 49–54, 57, 60, 61
ν	Frequency	20–22
ϕ	Fitness value	5–9, 27, 31, 33–36, 51
σ	Similarity value	21–24, 38, 42
θ	Task, or an exclusively parallel action	12, 16, 19, 26–28
φ	Fitness function	5, 14, 15, 25, 26
c	Colour	25–29
d	Donor value, or donated gene	32–35
e	Edge	25, 41, 43
f	Linkage set, or FOS element	7, 8, 10, 30, 31, 35, 38, 40–43
g	Gene	4–8, 20, 21, 30, 34, 42
n	Number of individuals inside a population or offspring	4, 5, 16, 18, 19, 21, 22, 28, 31, 33, 51, 56
o	Individual inside the offspring, or child	4–9, 31, 35, 36
p	Individual inside the population, or parent	4–9, 31–36
q	Action	11–14
r	Random value	32

Notation	Description	Page List
r	Round, or step	8–11, 18, 19, 22, 23, 27–29, 38, 41, 43, 44
s	State	11–14, 27
t	Target	48, 51
v	Vertex	10, 11, 25, 26, 28, 29, 41, 43, 44
w	Weight	22–29, 43, 44
y	Gene inside the intermediate matrix	34, 35

1 Introduction

Those still remembering the era of IBMs might be surprised in the changes that have been made in the consumer computer market. Expansions like the sound card have been completely integrated with the motherboard, but perhaps more surprising is that one expansion card survived and stood the test of time; the *Graphics Processing Unit* (GPU) is still present and has become a major component in any desktop configuration. We mostly have the entertainment branch to thank for its popularity. While availability increased and prices dropped, it started to gain the interest of computer scientists. GPUs have been designed with matrix operations in mind, and so they are able to process many—simple, but many—parallel tasks; a modern GPUs is easily capable of processing thousands of parallel tasks at once.

AI has become more and more of a phenomenon within the last decade. Its abbreviation is known to many outside the field, and might even be considered unavoidable. Its multi-purpose nature and competitive performance— which allowed for new engineering frontiers to be opened— has left a mark on both the scientific community and pop culture. The popularity of AI co-insists with the rise of interest in *parallel computing*, especially those in the consumer market. Without going into much detail, training *neural networks* and the like can be translated into a series of matrix operations, which gain a major performance boost if those are processed in parallel. You might recall from your linear algebra classes that operations such as multiplications are beyond tedious to do on paper, but are at its core the same process applied to different values. Throughout the decade, we have seen the rise of *Central Processing Units* (CPUs) containing more and more *threads*, where each thread is able to process a task in parallel. While common commercial processors are equipped with four, eight, or even more threads, those do not provide the parallelism which extensive neural networks demand. Introduce, GPGPU. After the successful introduction of GPU acceleration on neural networks in the late 2000s[12], many successful designs followed, leading up to the state where we are now.

AI is an umbrella term, which besides neural networks branches into many other optimisation algorithms, one of such branches is known as EAs[23]. As the name suggests, EAs take their inspiration from *evolutionary theory* to solve *optimisation* problems. By encoding a solution to a set of *genes*, we can create a *population* for which we can alter the genes through processes like *variation* and *selection*. In contrast to neural networks, which are typically used for supervised and unsupervised learning problems, EAs can be used for a wide range of optimisation problems. In recent years, great progress has been made in regards to the field of EA; the state of the art—known as GOMEA[22]—is capable of extracting mutual information from the population to improve the overall performance.

This thesis will focus on the feasibility of accelerating an EA by translating its processes to the GPU. We will in particular try to accelerate the *performance* of discrete-GOMEA. To measure the performance, we will use *max-cut* as our problem of choice. We would like to ask ourselves whether such an implementation is feasible to start with, and how it stands up against its non-accelerated counterpart.

In Chapter 2 we will start off by introducing the information needed before we can dive into the design for our *GPU accelerated GOMEA*. We will visit trivial topics such as the implementation of discrete-GOMEA and the integration of the GPU, and some less trivial topics like the introduction of graph colouring and its relevance to parallel dependencies. Two chapters are dedicated to the implementation of the GPU accelerated GOMEA; this is what this thesis is all about. In Section 2.1 we will discuss

the GPU acceleration of the complete variation process within discrete-GOMEA. In Chapter 4 we will finalise our design by introducing our set of flavours for GPU accelerated GOMEA, known as *packagers*. Next, we will measure the performance of our design by running a set of experiments in Chapter 5. Finally, we will discuss our findings in Chapter 6 and answer our question: Is it feasible to design a GPU accelerated EA, and how does it perform against discrete-GOMEA?This chapter ends with some recommendations for future work.

2

Background

This thesis will focus on accelerating discrete-GOMEA through large-scale parallelisation by using GPGPU. GOMEA is an obvious choice as it is considered the state of the art within the field.[22] Before we are able to explore the design behind *GPU accelerated GOMEA*—or GPU-GOMEA for short—we first need to provide ourselves with the required background information. In Section 2.1 we explore the design behind GOMEA, which includes to general idea behind EAs and the concept it took from *Genetic Algorithms* (GAs).

Next, we will introduce the concept behind large-scale parallel acceleration, which we will split up into three sections. First in Section 2.2 we will explore the definition behind a parallel process. Second, in Section 2.3 we will look into the parallel dependencies, which will become a major topic when we will design GPU-GOMEA. And at last, we will introduce our methodology for large scale-parallelisation in Section 2.4, which involves the use of GPGPU power by NVIDIA's *Compute Unified Device Architecture* (CUDA) technology.

Following this, we will spend two sections introducing the construction of the linkage tree. In Section 2.5 we will discuss the construction of the similarity matrix by making use of mutual information. In Section 2.6 we will discuss the clustering technique used to create the linkage tree, which is known by *Unweighted Pair Group Method with Arithmetic mean* (UPGMA). Additionally, each section will have a closer look at the design behind acceleration the processes through paralellisation.

At last, we will introduce three problems which are interesting to us. In Section 2.7 we will introduce the max-cut problem, which—as the title suggests— contains instances we attempt to solve with GPU-GOMEA. In Section 2.8 we introduce graph colouring, which provides us with a tool to find the independent partition of vertices. In Section 2.9 we introduce the concept behind *packages* which will find its use later at the design of GPU-GOMEA.

2.1. Optimal Mixing Evolutionary Algorithms

This section will provide all the background information one should need to understand GOMEA. This might feel extensive, but we should be completely comfortable before we move on to the implementation of GPU-GOMEA in Chapter 3. In Section 2.1.1, we start off with the basics of an EA, introducing topics like genes, populations, variation, selection, etc. Second, in Section 2.1.2 we will continue by exploring the basics behind genetic algorithms. Third, we move on to some of the core concepts of GOMEA, which we divide into two sections: the linkage model in Section 2.1.3, and optimal mixing in Section 2.1.4. And finally in Section 2.1.5 we will focus on the construction of the linkage tree by making use of mutual information and clustering.

2.1.1. Evolutionary Algorithms

Evolutionary Algorithms (EAs)[23] are a set of *optimisation* algorithms which are inspired by processes described in *evolutionary theory*. Evolution by natural *selection* is described as the process in which *genetic* information from the *population* is passed on to their *offspring*. Each *individual* inside the population is represented by a set of *genes*, known as the *genotype*. During the process of reproduction, the genes from the *parents* are tossed, possibly mutated, and eventually recombined into a new set of

genes which represents a new individual: their *child*. It is these genetics which contain the *traits* of an *individual*; changing the information also means changing the traits. A child is able to inherit the traits from one of their parents, or can have completely new ones caused by *variation* of their genes as a result of either the *mutation* or *recombination*. Each individual might face the possibility to survive into adulthood, and to produce children on their own. It is the traits that determine the success of their survival; the combination of traits describes the *fitness* of the individual, and thus their success to survive. Individuals with a low fitness risk a higher chance of not making it to adulthood. Those who do, will be able to reproduce and pass their traits on to their offspring.

Within the description for natural selection we can observe some characteristics of an optimisation algorithm. It is up to us to encode the terms *fitness* and *genes* to a problem more suitable within a mathematical context.[23] Let us first focus on our definition for genes. The population P is a set of individuals I which are described by their genotype, which is presented as a set of genes g. In nature we are bound by a set of pairs formed by molecular structures. However, from a mathematical perspective we have the liberty to translate our genes to any object we desire; numbers, symbols, sets, or whatever else suits our needs. For the purpose of our research, we focus primarily on *binary* genes, i.e. values which are either 0 or 1. This is a relatively popular approach for *Genetic Algorithms* (GAs), which we discuss later in Section 2.1.2.[23] It is up to us to encode the problem domain such that it is represented by the set of genes; the length ℓ and the encoding of the genotype to a set of binary values are consequentially bound to the problem description. Because we use binary values, instead of visualising the individual as a list, we might also favour a binary notation in which case we annotate our gene by its base value 2, i.e.

$$\begin{aligned} g &\triangleq \text{gene} \\ I &\triangleq \text{individual (or genotype)} \\ I &= \begin{bmatrix} g_0 & g_1 & \cdots & g_{(\ell-1)} \end{bmatrix} : g_i \in \{0,1\} \\ I &= g_0 g_1 \ldots g_{(\ell-1)2} : g_i \in \{0,1\} \text{ (base-2 notation)} \end{aligned} \tag{2.1}$$

Instead of using sets for our individuals and sets-of-sets for our population, we will use vectors and matrices respectively. Especially the latter representation will become important later when we will have a look at *GPU acceleration* in Section 2.4. To conclude, we will represent our individuals by a row vector containing ℓ genes, and our population by an $n \times \ell$ matrix such that each row represents an individual, i.e.

$$\begin{aligned} P &\triangleq \text{population (or parents)} \\ O &\triangleq \text{offspring (or children)} \\ P &= \begin{bmatrix} p_0 & p_1 & \cdots & p_{n-1} \end{bmatrix}^\mathsf{T} : p_i \in \{0,1\}^\ell \\ O &= \begin{bmatrix} o_0 & o_1 & \cdots & o_{n-1} \end{bmatrix}^\mathsf{T} : o_i \in \{0,1\}^\ell \end{aligned} \tag{2.2}$$

Example 2.1.1. Let us quickly go through an example related to the visual representation of individuals. Let us take the individual $I = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$. Instead of presenting it as a row vector, we can use the base-2 notation, which results in $I = 010110_2$. The subscript 2 is a common way to annotate a string of numbers with its corresponding base, in our case base 2. Similarly we can simplify our notation for the population P. Normally we would approach the population as a $n \times \ell$ matrix, i.e.

$$P = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{2.3}$$

However, we could visualise the same by using the alternative representation of a vector, i.e.

$$P = \begin{bmatrix} 010110_2 & 110011_2 & 101101_2 \end{bmatrix}^\mathsf{T} \tag{2.4}$$

Be aware that we still approach the population as a $n \times \ell$ matrix, and our individual representation is only for visual purposes.[1] In some occasions, we will only be interested in a single individual, which allows us to use the notations I and g_j for said individual and its respective gene. In others, when we are concerned about a specific individual inside a population, we will prefer the notations p_i and $p_{i,j}$ respectively.

Each individual p_i within the population P can be ranked by a measurement known as the *fitness value* ϕ_i. To calculate the fitness you need to apply a *fitness function* φ to said individual.[23] Within a mathematical context, this fitness function is usually given as part of a problem description, and includes a *decoder* to map a genotype to the problems domain. As a result, the fitness represents the solution for the given problem instance; the closer we get to the optimal solution, the larger the fitness value will be, and thus the better the ranking should be. Those individuals which are part of the lower ranked, might not be able to survive and thus will be eliminated from the offspring. Who survives—and thus where the cut lies between high and low ranked individuals—is determined by the *selection* process of the EA itself. For our research we only consider fitness values which are represented by *signed integers*. As we have similarly done for the population—and for similar reasons—we will use a vector to represent the set of fitness values, i.e.

$$\begin{aligned} \phi &\triangleq \text{fitness value} \\ \varphi(I) &\triangleq \text{fitness function} \\ \phi_i &= \varphi(p_i) : \phi \in \mathbb{Z}^+ \\ \Phi &= \begin{bmatrix} \phi_0 & \phi_1 & \cdots & \phi_{n-1} \end{bmatrix}^\mathsf{T} \end{aligned} \tag{2.5}$$

In general, EAs provide a set of processes which applies *variation* and *selection* to a set of individuals. There are no limits within our definition of *variation* nor *selection*; how we will mix the genes is up to us. Similarly, we are not bound to a fixed number of children produced, nor to the number of parents needed. Additionally—and in contrast to natural selection—we are able to improve our variation by evaluating the population and thus *learning* about its progress. To solve our problem instance, we will have multiple *generational steps* which generates a new population each step until one of the individuals has a fitness which meets the criteria that we are looking for. The whole process can be found in Algorithm 1.

Algorithm 1 The common EA Outline using a randomly generated population.

```
for each o_i ∈ O do                          ▷ Initialise random population
    o_i ← Random(ℓ, 2)
    φ_i ← Fitness(p_i)
end for
while not Terminate(Φ) do
    O ← Variation(P)
    {P, φ} ← Selection(O, Fitness)
end while
```

2.1.2. Genetic Algorithms

Now that we are knowledgeable of the basic building blocks which describe EAs, we are still left with some elements of which the implementation has yet to be resolved: How many parents do we use, how many children are created, how do we recombine the genes, and how do we provide variation? As you might expect, there have been many flavours of EAs. Out of the bunch the most popular one —and the one we are interested in for this thesis— is known under *Genetic Algorithms* (GAs)[23].

First off—to answer our question regarding the population and offspring—we could provide two parents which create two children, thus keeping the population consistent. GAs are one of the many EA

[1] To allocate binary values we will use the *Boolean* primitive type, which in C/C++ occupies one byte. Therefore we will keep seven bits unused for each element inside the population. Translating the binary values to numbers—which for our example translates to $I = 101101_2 = 45_{10}$—might be an interesting approach to optimise memory usage, however we will not consider it in this thesis.

flavours which use this setup. Second, GAs use a process known as *crossover* to apply variation. Crossover is a technique which generates children by exchanging genes between parents at certain random recombination points within the length of a genotype.[23] The simplest crossover *model* contains only one recombination point and is known as the 1-point crossover.

Example 2.1.2. Let us have a quick look at what it means to apply variation using crossover. First let us introduce the population containing two individuals. To add some visual aid, the genes of the second individual is highlighted in **red**, i.e.

$$\begin{aligned} p_0 &= 00000_2 \\ p_1 &= \mathbf{11111}_2 \end{aligned} \tag{2.6}$$

Now, let us consider a 1-point crossover. Normally this point is taken at random, but considering this is an example we are a bit cheeky and already know that our recombination point will take place after the third gene. By using the previously described population, we can generate two children which will represent our offspring, i.e.

$$\begin{aligned} o_0 &= 000\mathbf{11}_2 \\ o_1 &= \mathbf{111}00_2 \end{aligned} \tag{2.7}$$

Notice how all genes which used to be present within the population are still present within the offspring; no information will get lost during crossover. We can repeat this process using 2-point crossover which—as you might expect—uses two recombination points instead. Again, these points will randomly be determined, but with the power of examples we have observed that these point will take place after the first and fourth gene. By using the same parents as stated earlier, we get a new set of children, i.e.

$$\begin{aligned} o_0 &= 0\mathbf{111}0_2 \\ o_1 &= \mathbf{1}000\mathbf{1}_2 \end{aligned} \tag{2.8}$$

2.1.3. Linkage Model

If we try to solve a problem instance using a standard GA, it becomes clear that not all recombination points are equally favourable. Sometimes it is simply better to pick one over another; a favourable recombination point might improve the average fitness faster than the alternatives, or it simply reveals the optimal solution while the alternatives might not. If we dive a little bit deeper and try to create a better understanding on which recombination points are more favourable than others, a structure seems to appear. This structure reveals that some genes like to be inherited together during crossover; these genes are somehow *linked* to each other. It might not surprise you that both crossover and *genetic linkage* are concepts which find their origin in genetics.[21] Though—in contrast to what you would find in nature—our understanding of genetic linkage does not require linked genes to be close to one another; linked genes can be located everywhere inside the genotype.

Example 2.1.3. To further explore the definition of linkage, let us take a look at a small example. Herein we consider a problem instance which uses four variables for which the fitness function is defined as the sum of *Exclusive OR* (XOR) pairs, i.e.

$$\phi(I) = g_0 \oplus g_1 + g_2 \oplus g_3 \tag{2.9}$$

What is interesting about the XOR operator is that in isolation you can not determine whether a gene should be either a 1 or a 0. For our example, let us take a population of four individuals so the optimal solution is within the gene space. To add some visual aid to the process, the second and fourth individual will be highlighted in **red**, i.e.

$$\begin{aligned} p_0 &= 0100_2, \quad \phi_0 = 1 + 0 = 1 \\ p_1 &= \mathbf{1011}_2, \quad \phi_1 = 1 + 0 = 1 \\ p_2 &= 1110_2, \quad \phi_2 = 0 + 1 = 1 \\ p_3 &= \mathbf{1011}_2, \quad \phi_3 = 1 + 0 = 1 \end{aligned} \tag{2.10}$$

At random, it is determined that a recombination point will be located after the first gene. Additionally, it is determined that the first pair of children are the offspring of the parents p_0 and p_1, and similarly the second pair is the offspring of p_2 and p_3, i.e.

$$\begin{aligned} o_0 &= 00\,11_2, & \phi_0 &= 0+0=0 \\ o_1 &= 1111_2, & \phi_1 &= 0+0=0 \\ o_2 &= 1\,011_2, & \phi_2 &= 1+0=1 \\ o_3 &= 1110_2, & \phi_3 &= 0+1=1 \end{aligned} \tag{2.11}$$

What we observe, is that not only can an unfavourable recombination point stagnate the average fitness of a population, it can potentially decrease it. In contrast, if by random it is determined that the recombination point is allocated after the second gene, we observe a more favourable outcome, i.e.

$$\begin{aligned} o_0 &= 0100_2, & \phi_0 &= 1+0=1 \\ o_1 &= 1011_2, & \phi_1 &= 1+0=1 \\ o_2 &= 1111_2, & \phi_2 &= 0+0=0 \\ o_3 &= 1010_2, & \phi_3 &= 1+1=2 \end{aligned} \tag{2.12}$$

Choosing the right recombination point—rather than one by random—has shown to be relevant in our example. Whether a recombination point is favourable depends on the linkage between the genes. For our example, allowing the genes $\{g_0, g_1\}$ to be preserved *together* becomes heavily favourable, but just the gene g_0—or any individual gene for that matter—not per se.

Our hypothesis is that many problems out there have some form of underlying structure. Instead of using randomly chosen recombination points, we could optimise our GA by *learning* the underlying structure, to ultimately reveal the set of linked genes. What we basically achieve by learning the structure, is that we create a *model* based on our set of linked genes, a *linkage model* if you will.

A *linkage set* containing multiple linked genes is described by the indices of said genes. To describe the collection of all linkage sets, we will use a *Family of Subsets* (FOS), which describes a set of subsets over a common powerset[22], i.e.

$$\begin{aligned} &\mathcal{F} \triangleq \textit{Family of Subsets}\ \text{(FOS)} \\ &f \triangleq \text{linkage set} \\ &\mathcal{F} = \{f_0, f_1, ..., f_n \mid f_i \subseteq \{0, 1, ...\ell - 1\}\} \end{aligned} \tag{2.13}$$

Additionally, we make sure that each gene is represented in at least one of the linkage sets in our FOS. This way, no gene is left out when we will perform our generational step using the linkage sets.[22] Notice that it is completely valid to have only one gene inside a linkage set, which communicates that there is no linkage between said gene and any other, i.e.

$$\begin{aligned} &\forall i \in \{0, 1, ...\ell - 1\} \text{ such that} \\ &\quad \exists j \in \{f_0, f_1, ..., f_n\} : i \in f_j \end{aligned} \tag{2.14}$$

By using a FOS we allow ourselves to have multiple sets which intersect with one another. If none of the linkage sets do so, our FOS is better known as an *Marginal Product* (MP) FOS[22], i.e.

$$\begin{aligned} &\mathcal{F}_{mp} \triangleq \textit{Marginal Product}\ \text{(MP) FOS} \\ &\mathcal{F}_{mp} = \mathcal{F} \text{ such that} \\ &\quad f_i \cap f_j : i, j \in \{0, 1, ..., \ell - 1\} \end{aligned} \tag{2.15}$$

In such a case, even if the genes within a linkage set are linked to each other, there is no linkage between the sets themselves.[22] If we have an MP FOS such that each linkage set only contains one gene, then the structure is also known as a *univariate* FOS[22], i.e.

$$\begin{aligned} &\mathcal{F}_u \triangleq \textit{univariate FOS} \\ &\mathcal{F}_u = \mathcal{F} \text{ such that} \\ &\quad f_i = \{i\} : i \in \{0, 1, ..., \ell - 1\} \end{aligned} \tag{2.16}$$

We will continue to shed light on this notation of linkage in Section 2.3 when we have a closer look at dependencies.

2.1.4. Optimal Mixing

Gene-Pool Optimal Mixing (GOM)[22] is the process in which we use the linkage set to generate a new set of offspring. Using GOM will ensure that the offspring has a greater or equal fitness in comparison to its original parent. This constraint is known as *elitism*. By keeping individuals with equal or higher fitness, we ensure that the fittest individuals are allowed to pass down their traits to the next generation.

Instead of using crossover between two parents, GOM uses one parent and multiple *donors*, one for each linkage set described by our FOS. A donor is nothing more than another individual inside the population that donates the genes described by one of the linkage sets; the number of donors—-and the genes they donate—thus depends on the FOS structure. Before each donation is confirmed, it is checked whether the fitness value has improved or not. If the fitness value has declined, the donation is cancelled. If all donations have been processed, this individual that used to be part of the original population now represents the offspring; the population size is kept equal. Each donation is processed *sequentially* in *random* order. The complete process can be found in Algorithm 2.

Example 2.1.4. This example is used to illustrate how optimal mixing is performed. For the sake of convenience, we use a population with five individuals, each containing eight genes is depicted. We assume a FOS containing four linkage sets. For our example we will perform optimal mixing on first individual p_0 which by the end will represent our offspring o_0. At random, we have assigned each of the linkage sets to the remaining individuals, i.e.

$$\begin{aligned} p_0 &= 00001111_2 \\ p_1 &= 11001100_2, \quad f_0 = \{0, 1\} \\ p_2 &= 00111100_2, \quad f_1 = \{3, 6\} \\ p_3 &= 11110000_2, \quad f_2 = \{2, 4, 5\} \\ p_4 &= 01010101_2, \quad f_3 = \{7\} \end{aligned} \tag{2.17}$$

Let us not concern ourselves with how we got the linkage set at the moment, we will focus on that topic later in Section 2.6. Now, let us come up with a fitness function. For our example, we will use the *one-max* fitness function, which is equal to the sum of all genes, i.e.

$$\phi(I) = \sum_{i=0}^{\ell-1} g_i \tag{2.18}$$

Let us attempt to represent the process for optimal mixing as compact as possible. For each round r_i we take the donor p_i and linkage set f_0 and apply optimal mixing to it. If the fitness is larger or equal to the previous one, we keep it and provide it to round r_{i+1}. We will highlight the donated genes in **red**, i.e.

$$
\begin{array}{rlll}
r_0: & o_0 = 00001111_2 & \phi = 4 & \\
\text{donor}: & p_1 = \mathbf{11}001100_2 & & \\
 & o_0 = \mathbf{11}001111_2 & \phi = 6 & \text{better fitness} \\
r_1: & o_0 = 11001111_2 & \phi = 6 & \\
\text{donor}: & p_2 = 001\mathbf{1}11\mathbf{0}0_2 & & \\
 & o_0 = 110\mathbf{1}11\mathbf{0}1_2 & \phi = 6 & \text{same fitness} \\
r_2: & o_0 = 11011101_2 & \phi = 6 & \\
\text{donor}: & p_3 = 111\mathbf{1}0\mathbf{00}0_2 & & \\
 & o_0 = 111\mathbf{1}00\mathbf{0}1_2 & \phi = 5 & \textbf{worse fitness!} \\
r_3: & o_0 = 11011101_2 & \phi = 6 & \\
\text{donor}: & p_4 = 0101010\mathbf{1}_2 & & \\
 & o_0 = 1101110\mathbf{1}_2 & \phi = 6 & \text{same fitness} \\
\text{result}: & o_0 = 11011101_2 & \phi = 6 &
\end{array}
\tag{2.19}
$$

Algorithm 2 GOM($p_i, P, \mathcal{F}, \Phi$) variation for a FOS model

$o \leftarrow p_i$
$\phi \leftarrow \phi_i$
for each $f \in$ Shuffle($\mathcal{F}$) **do**
  $p' \leftarrow$ RandomParent(P) ▷ Such that $p' \neq p_i$
  $o' \leftarrow o$
  for each $j \in f$ **do** ▷ Donate genes described by linkage set f
    $o'_j \leftarrow p'_j$
  end for
  $\phi' \leftarrow$ fitness(o')
  if $\phi \leq \phi'$ **then** ▷ rate new fitness value
    $\phi \leftarrow \phi'$
    $o \leftarrow o'$
  end if
end for
return $\{o, \phi\}$

2.1.5. Linkage Modeling

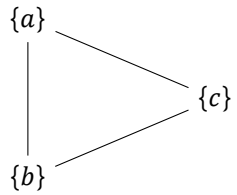
Sometimes it is relatively easy to determine the underlying structure by having a closer look at the given problem description. However, in other cases it is simply not possible to determine the structure a priori; we would like to be able to learn the linkage model instead. To construct a linkage set, we use the *mutual information* as our *similarity* metric to determine the linkage between two genes. If we want to construct our model, we are tempted to calculate the linkage for each and every subset of genes. However, this will explode to ℓ^2 calculations, where larger subsets will consequentially take more variables and thus require more time to be processed. This whole process unsurprisingly will take quite a large computational complexity to be resolved.[22] Alternatively, we could use an iterative approach and *merge* two linkage sets if their linkage is greater than any other in the current set of linkage sets, we can continue this process until we are left with the complete set.

Instead of calculating all ℓ^2 linkages, we will start off with the univariate FOS. For each pair we calculate the mutual information and determine the strongest linkage. This will require $1/2 \cdot \ell(\ell - 1)$ calculations. If we have found our pair, we merge them together as a new set; now we have $\ell - 1$ linkage sets left. For the next *round* we don't need to re-calculate the linkage between the non-merged linkage sets, but we do need it for each pair of non-merged and merged linkage sets. This will require an additional $\ell - 1$ calculations.

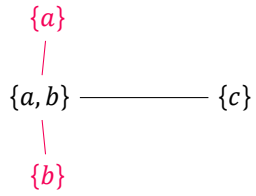
For each round r_i we have to re-calculate the linkage between $\ell - r_i$ pairs. To determine the linkage between pairs at round r_i, we now have $i + 1$ genes to consider; this will increase our computational complexity, as we have discussed before. Instead, we could leverage on the previously determined linkages. If two linkage sets f_i and f_j both have a linkage with another set f_k, then for the set $f_i \cup f_j$ we can take the *average* of those linkages to determine the new linkage between it and linkage set f_k. By taking the average we can determine the linkage in $O(1)$ time.

In general, for each round r_i it will take $\ell - r_i$ calculations to determine the linkage between the non-merged and merged subset pairs. In the end what is left is the linkage set which contains all genes: the complete set. By using our previously described process, it will require $\ell - 1$ rounds to get from the univariate FOS to the complete set. What we see is that, for each update we have less calculations to consider; we are able merge all linkage sets in $O(\ell)$ time.

Example 2.1.5. Let us have a look what this merging technique implies. In this example, rather than using the mutual information as a similarity metric, we will use the Euclidean distance; if the distance between two vertices are short, then the linkage is strong. Let us take a complete graph $G = (V, E)$ which contains three vertices described in Figure 2.1a.



(a) A graph containing three vertices. Notice that the edge between $\{a\}$ and $\{b\}$ contains the shortest distance.



(b) The graph after the first round. Notice how the new vertex $\{a, b\}$ can be found roughly between the old ones, which are coloured red.

Figure 2.1: A graph containing three vertices; each vertex represents a linkage set.

What we see is that the shortest distance can be found between the vertices $\{a\}$ and $\{b\}$. So for our first step we will merge those to create the new vertex $\{a, b\}$. Here we will use the average distances between $\{a\} - \{c\}$ and $\{b\} - \{c\}$ to represent the distance $\{a, b\} - \{c\}$. In Figure 2.1b we have visualised how our new graph will look after the first round. Notice how the new vertex can be found roughly between the old vertices; it does not seem to be unreasonable to consider the average distance as an useful approximation.

This whole process can be translated into a directed graph $G = (E, V)$. We will create our graph such that each linkage set is represented by a vertex $v \in V$ inside our graph. Each edge $\{v_i, v_j\} \in E$ meanwhile represents a merge; a linkage set v_i has been merged during some round r_k to create v_j. By following those conditions we have created a graph which is also known as a *binary tree* where each vertex represents a linkage set, a *linkage tree* if you will.

Example 2.1.6. We shall draw a linkage tree to inspire the imagination. Each Linkage tree starts with a set of linkage sets, such that each gene can be found in one linkage set. For our example, we will create a linkage tree using five genes, which result in a start of five linkage sets, i.e.

$$F = \{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}\} \tag{2.20}$$

Let us assume that we already know the order of linkage sets formed by merging two sets f_i and f_j. Each new linkage set will be generated every round r_i, resulting into a list of linkage sets, i.e.

$$\begin{aligned} r_1 &: \quad \{0\} \cup \{1\} \quad = \{0, 1\} \\ r_2 &: \quad \{2\} \cup \{3\} \quad = \{2, 3\} \\ r_3 &: \{2, 3\} \cup \{4\} \quad = \{2, 3, 4\} \\ r_4 &: \{0, 1\} \cup \{2, 3, 4\} \quad = \{0, 1, 2, 3, 4\} \end{aligned} \tag{2.21}$$

If we now use each linkage set as a vertex $v \in V$, we can represent a tree by creating an edge between vertices v_i and v_j if v_i has been merged during some round r_k to create v_j. For round r_1 this implies that there should be an edge between $\{0\}$ and $\{0,1\}$, but also between $\{1\}$ and $\{0,1\}$. The complete tree can be found in Figure 2.2.

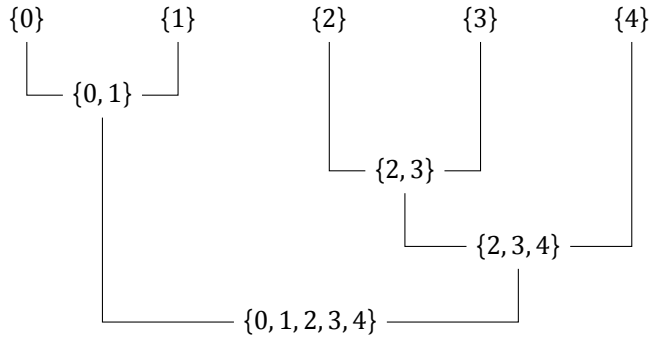


Figure 2.2: An arbitrarily linkage tree initialised by using five genes.

2.2. Parallel Processes

To be able to talk about processes, we will represent them as a transition systems rather than an instruction set. In such a representation, we define a graph $T = (V, E, Q)$ where vertices and edges represent *states* and *transitions* respectively such that multiple transitions can share the same *action* $q \in Q$. Actions represent atomic events which produce only one single outcome; the atomic event can neither branch of, nor can it be interfered by any other event [6]. This makes sense if we illustrate it into a graph.

Example 2.2.1. Let us take for example the alarm in Figure 2.3. We see the three actions $\{set, reset, beep\}$ which sets the time, resets the time, and which makes horrendous beeping sounds respectively. Notice how, by definition, an action can only happen before or after another; an action can not spawn from another, and an action can not interact with another.

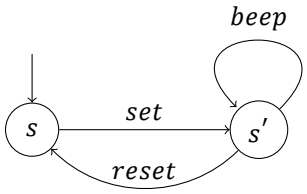


Figure 2.3: An example transition system describing an alarm containing three actions.

There are special cases where multiple actions are able to be performed at the same time. We say that those actions are performed in *parallel*. A parallel process is the set of parallel actions which can be translated to one single transition. For those actions, the order of execution should not matter; each parallel action can perform before, after, or simultaneously with any other parallel action. This view on how parallel actions are performed is called *interleaving*.[6] If interleaving is described, we say that those actions are *independent* of one another. This leads to a diamond shaped transition system for any set of actions, as shown in Figure 2.4.

We define that two actions p and q are performed in parallel, i.e. $p \parallel q$, only if both actions are independent of one another. The parallel process $p \parallel q$ terminates if both parallel actions p and q terminate. If we have more actions, for instance three, we write them down as $p \parallel q \parallel r$.[8] The parallel operator is commutative and associative, i.e.

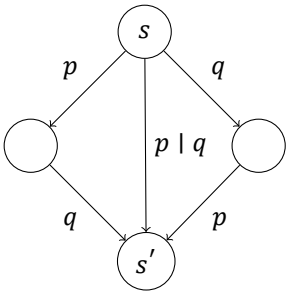


Figure 2.4: The parallel process $p \parallel q$ represented as a transition system. Here, the symbol | is used to represent if two processes happen *simultaneously*.

$$\begin{aligned} p \parallel q &= p \parallel q \\ (p \parallel q) \parallel r &= p \parallel (q \parallel r) \end{aligned} \tag{2.22}$$

We describe a parallel action as a *task* θ to emphasise that it has been designated to be performed in parallel exclusively. Similarly, if we want to emphasise that a state transition $s \to s'$ is performed in parallel we will use the alternative arrow notation $s \rightsquigarrow s'$.

Tasks—like serial processes—can be described as functions. For reasons that we will discuss later in Section 2.4.1, it is quite common for tasks to utilise the same mapping function to achieve parallelisation over a given set. This mapping function is also known as a *kernel function* κ. To be explicit about which elements are mapped, we will make use of the index notation. For similar reasons, we would like to express the variable for which we apply the kernel function, i.e.

$$\begin{aligned} \theta &\triangleq \text{task (parallel action)} \\ \theta : s \rightsquigarrow s' &\triangleq \text{parallel transition} \\ \kappa_i &\triangleq \text{kernel (mapping function)} \\ \text{kernel}\langle i \rangle &\triangleq \text{named kernel} \\ \kappa_i : \alpha_i \mapsto \alpha_i' &= f(\alpha_i), \text{ given} \\ \theta_i : s \rightsquigarrow s',\ \alpha_i \subseteq s,\ \alpha_i' &\subseteq s' \end{aligned} \tag{2.23}$$

Example 2.2.2. Let us have a look at an example task and how we could represent this in a meaningful manner. Let us take a mapping function κ which takes an element inside the set x and multiply it by two, i.e. $\kappa_i : x_i \mapsto 2x_i$. A quick observation should confirm that we can perform this process in parallel for any i. Let us have a look at the set $x = \{1, 2, 4\}$, and focus solely on the task θ_2. We can visualise the change in state by expressing our task as a transition, i.e.

$$\theta_2 : \{1, 2, \mathbf{4}\} \to \{1, 2, \mathbf{8}\} \tag{2.24}$$

If we wish to perform both tasks θ_0 and θ_2 in parallel, we could be explicit about the process by making use of the parallel transition notation, i.e.

$$\theta_0 \parallel \theta_2 : \{\mathbf{1}, 2, \mathbf{4}\} \rightsquigarrow \{\mathbf{2}, 2, \mathbf{8}\} \tag{2.25}$$

Additionally, if we want to apply our kernel function to a different set, i.e. $X \rightsquigarrow Y$, we can be explicit and embed the variable y_i into our kernel notation, i.e.

$$\kappa_i : x_i \mapsto y_i = 2x_i \tag{2.26}$$

2.3. Parallel Dependencies

We say that some actions are *dependent* on one another if they cannot be performed in parallel. We can confirm their (in)dependency by evaluating if they violate our definition of a parallel process as we described in Section 2.2. To recall, two or more actions are independent of one another if they can be translated to one single transition. There are two ways in which we can violate our description of a parallel process:

1. There is at least one action that does not start in the same state, i.e. $\nexists q : s \to s_q$, given $\forall q \in Q$
2. There is at least on action that does not end in the same state. $\nexists q : s'_q \to s'$, given $\forall q \in Q$

The first violation is what we know as a regular sequential process; if indeed some actions do not start at the same state, than there can be a path found between those actions, which forms a sequence. The second violation is a common cause of dependencies in kernel design, and is known as a *data dependency*[10], which we will discuss in more detail in Section 2.3.1. Additionally, we should have a closer look at the dependencies at play when it comes to the problem instance.

We should recall that GOMEA is partially reliant on the problem description; if there are somehow dependencies embedded in the problem description, we might have invalidated the variation process if those dependencies are not encapsulated inside a shared linkage set. Beyond the construction of a linkage tree, we might want to have a closer look at the problem description itself. This becomes even more relevant if we want to accelerate optimal mixing using parallelisation on multiple linkage sets. We will have a closer look at the *fitness dependency* in Section 2.3.2.

2.3.1. Data Dependency

A dependency caused by the second violation is also known as a data dependency. Derived from its name, we might not be surprised that the second violation revolves around resource management. data-dependencies arise when some actions require both read and write processes on the same resource. There is a dependency associated to each read and write order:[10]

- flow dependency: read after write, or RAW
- anti dependency: write after read, or WAR
- output dependency: write after write, or WAW
- input dependency: read after read, or RAR

Although we still considered the transition of two actions performed simultaneously as part of the parallel process, in reality this is hardly plausible. To synchronise two actions such that both will be performed at the same time, is in practice unfeasible. What we get in response is thus not three but two possible states. Which of the two states is preferred depends on which action is performed first. In practice, the final state is completely non-deterministic. This phenomenon is known as a *race condition*.

Example 2.3.1. We shall demonstrate the second violation through the following example. If we have the actions $p : \{x, y\} \to \{x + y, y\}$ and $q : \{x, y\} \to \{x, x + y\}$. Let us have the start state $s_0 = \{1, 3\}$. From those actions, we can draw our transition diagram and see if we get our diamond structure. We can see our transition system in Figure 2.5.

From our transition system, we see that we will end up in three different states, and thus violate our definition. What we see is that a parallel process heavily depends on the resource management.

2.3.2. Fitness Dependency

There are many interpretations you can take on the meaning of *dependency*, one of them we found in the sequence of actions. Another approach is by understanding it as a *linkage*; if an action changes one variable, it automatically changes the variable which depends on it. This is known as a *variable dependency*. An independent variable is deemed as such if it cannot be expressed by another variable, and vice versa. There is something to say in regards to the perspective of an expression, for instance, we could easily have rewritten the expression and considered another variable dependent; the perspective is always context dependent.

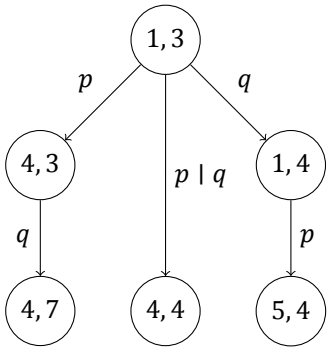


Figure 2.5: The actions p and q if they are performed before, after, and simultaneously of one another.

Example 2.3.2. For example, if we take a look at the expression $y = a + b$. Here, we have been able to express y by using two other variables. We say that y is the dependent variable, and both a and b are the independent ones. Let us try to visualise the linkage as a state transition. We create a set s which contains all of our three variables, and try to alter the variable a through the kernel κ. To not break our definition of actions, we should state that reading the variables from s to reaveal its values, is an action on its own. Let us observe the action q for which we applied the kernel κ, where we will highlight the altered variables in **red**, i.e.

$$\begin{aligned} s &= \{y, a, b\} = \{6, 2, 4\} \\ \kappa &: a \mapsto 2a \\ q_\kappa &: \{6, 2, 4\} \to \{\mathbf{8}, \mathbf{4}, 4\} \end{aligned} \tag{2.27}$$

We performed a kernel on a and we expected a change on only a, however, we additionally observe a change in y; our expectations did not match our observations. These two variables are somehow linked with each other; a change in a implies a change in y. In our deterministic environment, the only conclusion we can come to, is that y is not an independent variable. Finding such a linkage is a simple proof to state whether a problem is independent or not.

This notion of dependency as a linkage is unsurprisingly applicable to our linkage sets.[22] The variables of a problem instance are those found inside the fitness function, and thus correlates to the genes through the encoding of said problem. Because there is encoding involved in EAs to translate the problem variables to the set of genes, there is always a variable dependency between both. However, what we are more interested in is whether there is a dependency among the problem variables, and therefore among the genes themselves; these are known as *fitness dependencies*.

If our problem instance is fully fitness independent—i.e. all genes are independent of one another—we expect each gene to have a clear and independent *contribution* to the fitness value; each contribution should have only one variable dependency. If a contribution which has more than one variable dependency exists, we say that those variables are *fitness dependent* on each other. This is equal to questioning ourselves if we can create an univariate FOS from our problem instance. We say that our problem instance contains exclusively independent genes if we can express the fitness function as a sum of decoding contribution functions which take only one gene each, i.e.

$$\varphi_U(I) = \sum_{g_i \in I} \delta_i(g) \tag{2.28}$$

Recall that ideally, a linkage set is constructed in such a way that within a linkage set the variables are dependent on one another, while the variables between the linkage sets are independent of each other. However we should keep in mind that this is not a constraint for the construction of a FOS, as we have experienced with the linkage tree. We can loosen our constraint on the sum of contributions by

using the MP FOS: a problem instance contains exclusively independent subsets of dependent genes, if we can express the function as a sum of decoding functions each containing a set inside the MP FOS, i.e.

$$\varphi_{MP}(\mathcal{F}) = \sum_{f_i \in \mathcal{F}_{MP}} \delta_i(f) \tag{2.29}$$

2.4. General Purpose GPU Acceleration

Graphics Processing Units (GPUs) are dedicated processor units which are highly specialised in generating visuals on a digital displays. They are designed to accommodate the *shader model*, in which each task is dedicated to drawing one vertex, or shading one pixel fragment. Since their early depiction, GPUs have evolved into high performance units excelling in parallel workloads. This came to the notion of researches and developers which desired the need for other, more *general purpose* algorithms to be accelerated using these highly specialised, but also easily accessible processor units[16].

First in Section 2.4.1, we will have a look at the definitions of threads, blocks, grids and kernels. Second, we will dive into the memory hierarchy in Section 2.4.2. Understanding the hierarchy is an important part of the design of the algorithm. Bad memory management could result in a hefty performance hit, more so than we might expect in a non-accelerated algorithm. In Section 2.4.3, we will finish off with an in-depth look into atomic functions and the reduction algorithm. Reduction is an algorithm to accelerate commutative operations, even if those variables are dependent by definition. Reduction will become an important part in our design, as we will encounter one very common sequential operation; the summation of weighted edges is simply unavoidable.

2.4.1. Threads, Blocks, Grids and Kernels

Compute Unified Device Architecture (CUDA) is NVIDIA's answer to the desire for GPGPU. CUDA is an extension of the C/C++ language. Developers are tasked to design *kernels*, which will be executed across parallel *threads*. Kernels are nothing more than C functions which can contain any form of procedure, with some limits. A kernel is declared using the `__global__` specifier. For example. a kernel function called `Kernel` which requires no parameters will translate to the kernel declaration `__global__ void Kernel() {...}`.[18]

Threads are organised inside *blocks*, and blocks inside *grids*. Each block represents a three-dimensional matrix of threads, and similarly a grid represents a three-dimensional matrix of blocks. The dimensions of each block depends on the configuration defined by the developer. Each block can contain 1024 threads[2], meaning that the maximum sized block is undefined. On the hardware, each thread is additionally organised inside a *warp*. Each warp consist out of 32 threads, such that all threads will execute the same instruction. While most of the warp processes happen behind the scenes of the developer—which in contrast to threads and blocks are not specified at a kernel call—we should still take them into account such that we can achieve some improvements by making explicit use of warp programming.[13]

Similarly, the dimensions of the grid are configurable. However—in contrast the the definition of the block—the maximum sized grid is predefined; it describes a maximum of $2^{31} - 1 \times 65535 \times 65535$ blocks.[19] A visualisation is available in Figure 2.6. To declare the dimensions for both the grids and the blocks, we call the kernel with the extended function-call syntax `<<<dimGrid, dimBlock>>>`. Both the `dimGrid` and `dimBlock` can be either an integer or a three dimensional vector. The integer is a shortcut for using only one dimension, which is sometimes enough for the developer.

> **Example 2.4.1.** Let us take an example. We desire to call an example kernel named `Kernel` which requires the use of 6 blocks, each containing 32 threads. This translates to the function call `Kernel<<<6, 32>>>(...)`.

Each thread—and similarly each block—is identifiable by the coordinates within the matrix. These coordinates are accessible as a vector representing the dimensions `x`, `y`, and `z`. These vectors for both the block and grid coordinates are provided by the kernel as the vectors `threadIdx` and `blockIdx`

[2]The maximum threads per block, and the grid dimensionality depends on the architecture of the GPU.

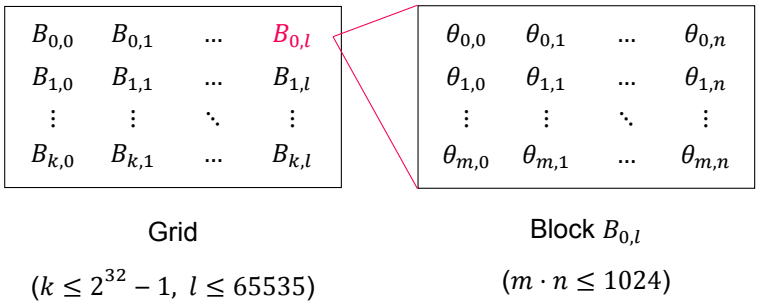


Figure 2.6: A visual representation of threads θ organised inside blocks B, which are similarly organised inside the grid. Illustrated are only the two dimensions for both the block and the grid.

respectively. Because the dimensions of the blocks might vary depending on the configuration, the kernel also provides the vector `blockDim`. Together they form a unique identifier for each thread.

Example 2.4.2. Let us have a look at an example. We want to parallelise a procedure which updates the vector Y by applying a scalar α to another vector X, and adds it to the original vector Y: $\{x, y\} \mapsto y = \alpha x + y$. We can parallise this procedure by realising that there is an independent process associated to each element of the vector Y. Given $|Y| = |X| = n$, we are required to launch at least n threads. If we keep the number of threads inside a block fixed at the maximum allowed threads, we only need to take care of the number of blocks. Because we now might declare more threads than needed, we must check if a thread will not access memory which happens to be out of bounds. We can parallise our example using CUDA, which kernel is described in Listing 2.1.

```
1 __global__ void Kernel(size_t n, float alpha, float *x,
      float *y) {
2   auto i(blockIdx.x * blockDim.x + threadIdx.x);
3   if( i >= n ) { return; }
4   y[i] = alpha * x[i] + y[i];
5 }
6
7 unsigned blocks((n + kMaxThreads - 1) / kMaxThreads);
8 Kernel<<<blocks, kMaxThreads>>>(n, alpha, x, y);
```

Listing 2.1: A CUDA kernel example for the kernel $\{x, y\} \mapsto y = \alpha x + y$

Now that we are familiar with the notion of dimensionality of threads, it might also be useful to view our tasks as such. Take for example the task $\theta_{i,j} : x_{i,j} \rightsquigarrow y_{i,j}$. Even if it might be clear from the definition that we desire a $n \times m$ matrix, we can also show this visually, i.e.

$$\theta_{i,j} : x_{i,j} \mapsto y_{i,j}$$
$$\theta = \begin{bmatrix} \theta_{0,0} & \theta_{0,1} & \cdots & \theta_{0,n-1} \\ \theta_{1,0} & \theta_{1,1} & \cdots & \theta_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{m-1,0} & \theta_{m-1,1} & \cdots & \theta_{m-1,n-1} \end{bmatrix} \tag{2.30}$$

2.4.2. Memory Hierarchy and Allocation

CUDA provides the developer with multiple memory spaces. Depending on the needs, each memory space provides its own scope of accessibility. Because one provides a higher level of accessibility, we talk about a memory hierarchy. Each thread is assigned its own *local memory* and *registers*. This memory space is only accessible for the thread that it is using it. Next, we have *shared memory*, which is memory accessible from a block that it is assigned to; each of the threads within the block is able to access the shared memory space. At last we have *global memory*, which is accessible by each

grid, and thus by each thread. Global memory is the data we assign at the host, and which is used as a parameter to call the kernel. Accessing memory comes with a performance penalty. However, it depends which memory space you will use. Registers are considered the fastest, after shared memory, and at last local and global memory. It is thus sometimes beneficial to copy data from the global memory to the shared memory.

To allocate registers and local memory, we simply use the syntax provided by C. Each thread has a limited amount of registers and can be allocated through the declaration of a variable. Local memory on the other hand is used for spilled over registers, large structures which consume too much space as a register, or arrays with a non-constant size specifier. To allocate local memory, we could make use of the latter, although we should be aware that it is up to the compiler to make this decision. To allocate shared memory, we need to do two things. First, we are required to define the size by declaring it at the kernel call as the third parameter inside the call-function syntax, i.e. `<<<dimGrid, dimBlock, dimMem>>>`. Second, to declare the shared memory inside the kernel, we must use `__shared__` specifier prior to the variable declaration. Because we declared our memory at the host, we must also include the `extern` keyword[3]. Global memory must be allocated at the host by using the specially created function `cudaMallocManaged(data, length)`. This function allows us to allocate the memory on the device while also automatically managing it on the host by the *Unified Memory* system. What this means is that we can use `data` as any normal array at the host, while also providing it to any kernel call. To synchronise the data between both the host and the device we can call `cudaDeviceSynchronize()`.

Example 2.4.3. Let us take an example which uses everything we know about memory hierarchy. What we want is to design a kernel which calculates its own value scaled by π and incremented by its neighbour, i.e. $\kappa_i : x_i \mapsto y_i = \pi \cdot x_i + x_{i+1}$. To keep things simple, we require that x is not larger than the maximum number of threads allowed in one block. The complete process can be found in Listing 2.2.

```
__global__ void Kernel(size_t n, float *x) {
  auto i(blockIdx.x * blockDim.x + threadIdx.x);
  if( i >= n ) { return; }

  auto pi(3.14159f);          \\ Declare Local memory
  extern __shared__ d[];      \\ Declare shared memory
  d[i] = x_i[i];

  __syncthreads();            \\ Synchronise threads
  y[i] = pi * d[i] + d[i+1];
}

constexpr unsigned kMaxThreads(1024);
constexpr unsigned n(4);      \\ Arbitrary size < 1024

float x[n]{1, 2, 3, 4};
cudaMallocManaged(x, n);      \\ Allocate global memory

Kernel<<<1, kMaxThreads, n>>>(n, x);

cudaDeviceSynchronize();      \\ Synchronise with device
cout << x[0] << endl;         \\ prints 26.436..
```

Listing 2.2: A CUDA kernel example for the kernel $x_i \mapsto y_i = \pi \cdot x_i + x_{i+1}$ given $|x| < 1024$. Notice how there are no race conditions as we store the data x temporarily to shared memory.

2.4.3. Atomic Functions and Reduction

As we have discussed earlier in Section 2.3, not every process is independent of one another, and thus cannot be accelerated by performing them in parallel. *Commutative operators* such as the *summation*,

[3] It is possible to declare the shared memory at the device, however, the main benefit of declaring it at the host is that the size can be dynamically allocated.

product, *maximum*, and *minimum* are common operators in mathematics and computer science which can be found in all kinds of processes. The result of a commutative operation is variable dependent of each and every variable inside the operator. Despite these dependencies, we can still improve these kind of operators by using parallel acceleration. First let us have a look at a build in solution in CUDA to resolve data dependencies. Second we will have a closer look at the actual kernel design to improve our commutative operators.

Atomic Functions
If an action assigned to a variable inside a sequence of commutative operations can be meaningfully accelerated by parallelisation, we still might want to take advantage of that. One option is to write the result for each variable in the process to a designated address in global memory, similarly as we would do for any other mapping function. Back on the host we can retrieve the memory from the device, and calculate the sequential operation by iterating over each item in the newly created sequence. Alternatively, we can use what CUDA calls *atomic functions*. Atomic functions allow a read-modify-write operation to an address in global or shared memory, guaranteeing for the process to be performed without the interference of other threads, meaning that the memory address cannot be accessed by any other process[17]. Examples of common atomic functions are `atomicAdd()`, and `atomicMax()`. Any atomic operation can be constructed based on the compare-and-swap `atomiCAS()` function, allowing for extensions beyond what has already been provided by CUDA. The support for atomic functions depends on the architecture of the GPU, and is still in development. For instance, integer atomic functions have been around since *compute capability* version 1.1—commonly associated with a subset of the Tesla architecture, released in 2006—and at version 8.0—commonly associated with the Ampere architecture, released in 2020—the atomic addition operation was expended to support 64bit floating points values[19]. Note that, if we use n atomic functions calls to perform some sequential operation on n elements within a sequence, we still lose quite some performance in comparison to a simple iterator on the host.

Reduction
There is a solution that makes use of parallelisation to accelerate commutative operations. *Reduction* is a tree-based approach used within each thread block[9]. Instead of iterating over all variables s_i within the sequence S, we use a divide-and-conquer approach by only using the operation on a pair of variables s_i and s_j. i.e.

$$\begin{gathered} O(\cdot) \triangleq \text{Operator} \\ \kappa_{i,j} : s_i \mapsto O(s_i, s_j) \end{gathered} \tag{2.31}$$

We can use this kernel to perform a task for each pair, as long as those pairs do not intersect with one another. If we started with n variables in our sequence, we created $n/2$ pairs, which results in $n/2$ solutions. We can continue this process until there is only one variable left. Notice how we only need one task for two variables, meaning that we only need half the amount of tasks that has been used in the previous round. This process of calling the kernel for multiple pairs in multiple rounds will create a binary tree, with at the root the final solution. To store all in between between solutions, reduction heavily relies on shared memory, which is expected to be much faster than local memory, as discussed in Section 2.4.2. The time complexity of reduction is equal to the depth of the binary tree. If we assume that each parallel process can be performed in $O(1)$ time, then the whole reduction process takes up $O(\log(n))$ time.

Reduction can be realised in multitude of ways, but in this case, we focus on *sequential addressing*. First, to create a binary tree of operations, we must start with a sequence in which the number of variables is a power of two. If this is not the case, we must reduce our sequence to match the closest power of two, i.e.

$$n' = 2^m : n' < n,\ m = \lfloor \log_2(|s|) \rfloor \tag{2.32}$$

Second, we will not be choosing our pairs by random. Instead we will use a specified distance between each pair, which is known as the *stride* s. Sequential addressing uses a stride starting at half of the total number variables. For each round r_x, we will reduce this distance by half until our stride is equal to zero, i.e.

$$\kappa^{\mathcal{s}}_{i,x} : s_i \mapsto O(s_i, s_{i+\mathcal{s}}) : \text{such that}$$
$$\mathcal{s} = 2^{m-x},\ r_x,\ n' = 2^m \tag{2.33}$$

To reduce the number of variables inside our sequence, and thus to include all variables beyond n', we can yet again use our reduction technique, except now we use a stride equal to n'. What we do however need to take into consideration is that we do not access variables outside our specified sequence. Notice that we can take any arbitrary value for n'; the closest value to n seems to be an obvious first choice. However due to hardware limitation this is not always achievable. Such a case requires us to have multiple rounds to reduce the size of our sequence. The algorithm can be found in Algorithm 3.

Algorithm 3 Reduction$\langle i \rangle(S, r, n)$

```
x ← 1
m ← ⌊log2(|S|)⌋
n' ← 2^m
shared s'_i ← s_i
if i + n' ≤ n then
    s'_i ← O(s'_i, s_{i+n})
end if
sync threads
while x < m do
    s ← 2^{m−x}
    if i > s then
        return
    end if
    s'_i ← O(s'_i, s_{i+σ})
    x ← x + 1
    sync threads
end while
r ← s_0
```

Example 2.4.4. Let us take an example. We have a sequence $S = \{4, 2, 2, 1\}$ where we want to calculate the summation i.e. $O(i, j) = i + j$. We start with $n = |s| = 2^2 = 4$ tasks, each representing a leaf in our binary tree. For the first round, we use a stride of $\mathcal{s} = 2^{2-1} = 2$. Notice how we only continue those tasks θ_i such that $i \leq \mathcal{s}$. If we look at θ_1, we see that our value becomes $s_1 = 2 + 1 = 3$. We do the same for θ_0, which gives us $s_0 = 6$. If we take another round, we see that our stride changes to $\sigma = 2^{2-2} = 1$, meaning that only θ_0 will continue. If we calculate this value, we end up at one vertex; our root. This root contains the eventual solution of our summation. The example is visualised in Figure 2.7.

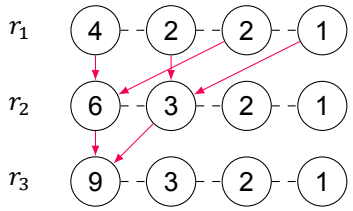


Figure 2.7: Reduction using sequential addressing to calculate the summation over the sequence $S = \{4, 2, 2, 1\}$.

2.5. Mutual Information

In order to create the linkage tree, we have chosen to use mutual information for our similarity metric. To determine the mutual information between two genes—which all together represents the similarity

matrix Σ—-we need to calculate their probability, which we can achieve by creating a $(\ell - 1) \times (\ell - 1)$ frequency matrix N. In Section 2.5.1 we will introduce the process behind the construction of the similarity matrix. In Section 2.5.2 we will have a closer look at the parallelisation of said construction, which we will use later in Section 2.1.

2.5.1. Construction

The frequency $\nu_{i,j} \in N$ is represented by tuple containing the combinations available for the genes g_i and g_j. In our thesis we only consider binary values for our genes, which results in four combinations. Because each pair can be mirrored, we only need to use a *triangular matrix* , i.e.

$$\begin{aligned} N &\triangleq \text{Frequency matrix} \\ \nu &\triangleq \text{Frequency tuple} \\ \nu &= (b_0, b_1, b_2, b_3) \end{aligned} \tag{2.34}$$

$$N = \begin{bmatrix} - & - & \cdots & - & - \\ \nu_{1,0} & - & \cdots & - & - \\ \nu_{2,0} & \nu_{2,1} & \cdots & - & - \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \nu_{(\ell-1),0} & \nu_{\ell-1,1} & \cdots & \nu_{(\ell-1),(\ell-2)} & - \end{bmatrix} \tag{2.35}$$

To create the frequency tuple, we look into the population and count the number of combinations we can find. To determine which combination should contribute to which value inside the tuple, we translate it to a binary representation given g_i is the most significant bit, i.e.

$$\nu_{i,j} = \sum_{I \in P} \nu' : \nu' = \begin{cases} (1,0,0,0), & g_i g_{j2} = 11_2 \\ (0,1,0,0), & \cdots = 10_2 \\ (0,0,1,0), & \cdots = 01_2 \\ (0,0,0,1), & \cdots = 00_2 \end{cases} \tag{2.36}$$

Example 2.5.1. Next, we have a small example to showcase how the frequency is calculated. First let us have a look at the following population:

$$P = \begin{bmatrix} 110110_2 \\ 101010_2 \\ 100100_2 \\ 111110_2 \end{bmatrix} \tag{2.37}$$

Now, let us try to calculate the frequency for $\nu_{1,3}$. For our example, we only have to sum up four tuples, i.e.

$$\begin{array}{cccccc} g_1 & g_3 & & & & \\ 1 & 1 & \Longrightarrow & \nu_0' & = & (1,0,0,0) \\ 0 & 0 & \Longrightarrow & \nu_1' & = & (0,0,0,1) \\ 0 & 1 & \Longrightarrow & \nu_2' & = & (0,0,1,0) \\ 1 & 1 & \Longrightarrow & \nu_3' & = & (1,0,0,0) \\ \hline \multicolumn{6}{l}{\nu_{1,3} = \nu_0' + \nu_1' + \nu_2' + \nu_3' \quad = \quad (2,0,1,1)} \end{array} \tag{2.38}$$

What we see is that we get all kinds of values inside our tuple; they are almost equally distributed. We can thus observe that the population did not settle into a distinct combination for those two genes. We can conclude that these pairs of genes are weakly linked. An example of the strongest linkage possible for our population would be if one of the combinations was equal to four and consequentially the others were zero; for example, the genes g_0 and g_5 are strongly linked if $\nu_{0,5} = (0,4,0,0)$.

By using the frequency matrix N we are able to construct the similarity matrix Σ. Translating one to the other is as simple as applying a mapping function which uses the tuple as input. To calculate the mutual information, we use the *entropy function* H[22]. To visualise which linkage set is associated with which row and column we will add them as labels, i.e.

$$\begin{aligned} \Sigma &\triangleq \text{ similarity matrix} \\ H(\cdot) &\triangleq \text{ entropy function} \\ \sigma_{i,j} = H_(\nu_{i,j}) &= \sum_{b \in \nu_{i,j}} \frac{b}{n} \log\left(\frac{b}{n}\right) \end{aligned} \tag{2.39}$$

$$\Sigma = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ \vdots \\ \ell-1 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & \cdots & \ell-2 & \ell-1 \end{array} \\ \begin{bmatrix} - & - & \cdots & - & - \\ \sigma_{1,0} & - & \cdots & - & - \\ \sigma_{2,0} & \sigma_{2,1} & \cdots & - & - \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \sigma_{(\ell-1),0} & \sigma_{(\ell-1),1} & \cdots & \sigma_{(\ell-1),(\ell-2)} & - \end{bmatrix} \end{array} \tag{2.40}$$

In information theory, entropy has the property to reach its maximum value if all variables are equally distributed, i.e. all frequencies are equal to one other. If all our four combinations result in the same frequency—which results in the maximum entropy—we deduce that the linkage is at its weakest. If we want to merge the linkage set pair with the strongest linkage, we thus must look at the smallest entropy inside the similarity matrix.

Example 2.5.2. We shall now plot out an example. Let us assume that we want to determine the entropy of two outcomes. With only two different outcomes, their probabilities are constructed such that one is equal to x and the other to $x - 1$. If we take a coin flip for instance, we assume both sides equally occur, and thus $x = 0.5$. If we calculate the entropy for the coin flip, we see that it results in an entropy $H(0.5) = 1$. We can plot this for every value x as visualised in Figure 2.8. What we see is that the entropy reaches its maximum if both probabilities are equal to one another. This also applies when there are more than two outcomes.

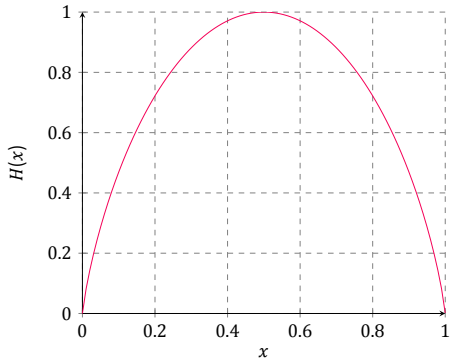


Figure 2.8: The entropy for two outcomes given their probability is equal to x and $x - 1$ respectively, i.e. $H(x) = -xlog_2(x) - (1 - x)log_2(1 - x)$

2.5.2. Parallel Acceleration

To determine the mutual information, we have to perform two sub-processes: first, we need to create the frequency matrix, and then the second process entails calculating the similarity matrix from it. To accelerate the first, we see one problem. We need to sum the results over the set of pairs to get our frequency. Therefore we can conclude that this process is variable dependent, and thus cannot be accelerated. If we recall back in Section 2.4.3, we see that we could make use of the atomic addition

to prevent non-deterministic behaviour and still perform the process on the device. Due to the use of buckets, optimistically we improve the number of evaluations from n to $n/4$; making use of parallel acceleration to calculate the frequency matrix did not improve the time complexity. There are pros and cons to this approach. One benefit is that we can keep the data on the device, preventing unnecessary and rather slow communication between the host and device. On the other side, atomic functions in practice turn out to be less optimal than simple iterators on the host.

Now, let us have a look at accelerating the construction of the similarity matrix using parallelisation. To get the matrix, we calculate the similarity described in each 4-tuple by using *entropy* as our similarity measurement. The entropy can be calculated as its own kernel simply because its information is isolated to a single tuple, i.e.

$$\text{entropy}\langle i,j\rangle : \nu_{i,j} \mapsto \sigma_{i,j}, \text{such that}$$
$$\sigma_{i,j} = \sum_{\nu'_k \in glsnu[i,j]} \frac{\nu'_k}{n} \log \nu'_k \tag{2.41}$$

2.6. UPGMA

To construct the linkage tree, we will apply the average clustering of *Unweighted Pair Group Method with Arithmetic mean* (UPGMA).[20] UPGMA uses an averaging technique to cluster two subsets together until only the complete set is left. If we start off with n subsets, then it will take $n-1$ rounds to reach the complete set. UPGMA constructs a tree structure called the *dendrogram*. Each of the vertices inside a dendrogram represents a cluster, and the edges represent which subsets have been clustered to form the new subset. Unique to the dendrogram is that it records the *height* of each cluster on an additional axis. By squinting our eyes it should become clear that the dendrogram is similar to our linkage tree, with the only difference being the lack of height representation for the linkage tree.

In Section 2.6.1 we will have a closer look at the construction of the dendogram, which includes the process behind merging two subsets together. In Section 2.6.2 we will have a close look at accelerating the process by making use of parallelisation.

2.6.1. Construction

To cluster two subsets, we first need to find the smallest value inside our similarity matrix $\sigma_{i,j}$. To update the similarity matrix, we remove the rows and columns which represent the subsets i and j, and create new ones for the subset $i \cup j$; the $m \times n$ similarity matrix shrinks to a $m-1 \times n-1$ matrix. To create the new similarity values for the subset $i \cup j$ we will make use of our previous similarity matrix. For a subset x, to create the similarity between it and the new subset, we will make use of the weighted average for both the $\sigma_{x,i}$ and $\sigma_{x,j}$. This means that we keep track of the weights for each subset inside the similarity matrix; each subset $\sigma_i \in S$ has an associated weight $w_i \in W$. The weighted average thus uses the weights w_i and w_j as scalars for both their similarity values, and averages them out using the same weights. The weight associated with our new subset will be equal to the sum of the previous weights, i.e.

$$\begin{aligned} \sigma_{x,i\cup j} &= \frac{w_i\sigma_{x,i} + w_j\sigma_{x,j}}{w_i + w_j} \\ w_{x,i\cup j} &= w_i + w_j \end{aligned} \tag{2.42}$$

Example 2.6.1. Let us have an example to illustrate the clustering technique for merging two linkage sets. First, let us take the following 5×5 similarity matrix Σ such that we have five linkage sets, each containing only one gene, i.e.

$$r_0 : \quad \Sigma = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} - & - & - & - & - \\ 90 & - & - & - & - \\ 80 & 70 & - & - & - \\ 60 & \mathbf{40} & 50 & - & - \\ 65 & 75 & 85 & 95 & - \end{bmatrix} \end{array}, \quad w = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{2.43}$$

Notice how we used the triangular matrix instead of the complete matrix. Also, to keep the visualisation clear, we do not write the linkage set with only one gene as a set. First, we find out that **40** is the smallest entropy registered, which we have highlighted **red**; the linkage sets d and e have the strongest linkage among all the other linkage set pairs. Second, we must update the matrix such that it contains the row and column for our new linkage set $\{d, e\}$ and delete the rows and columns for d and e respectively. Let us focus on the creation of $\sigma_{a,\{b,d\}}$. Notice how we use a triangular matrix, and thus if $\sigma_{a,d}$ does not exist, there will be a value for $\sigma_{d,a}$. We first check the weights for both d and e which are both equal to 1. Now, we simply follow the mapping function, i.e.

$$\begin{aligned} \sigma_{a,\{b,d\}} &= \frac{w_b\sigma_{a,b} + w_d\sigma_{a,d}}{w_b + w_d} \\ &= \tfrac{1}{2}(60 + 90) \\ &= 75 \\ w_{\{b,d\}} &= 1 + 1 \\ &= 2 \end{aligned} \tag{2.44}$$

We can simply repeat this for all the other linkage sets, which results in our new 4×4 similarity matrix. We also need to make sure to update the weight for our newly created linkage set; instead of keeping the old weights, we remove it and align the new weight to the corresponding row, i.e.

$$r_1: \quad \Sigma = \begin{array}{c} \\ a \\ \{b,d\} \\ c \\ e \end{array} \begin{array}{c} \begin{array}{cccc} a & \{b,d\} & c & e \end{array} \\ \begin{bmatrix} - & - & - & - \\ \mathbf{75} & - & - & - \\ 80 & \mathbf{45} & - & - \\ 65 & \mathbf{85} & 85 & - \end{bmatrix} \end{array}, \quad w = \begin{bmatrix} 1 \\ \mathbf{2} \\ 1 \\ 1 \end{bmatrix} \tag{2.45}$$

2.6.2. Parallel Acceleration

Let us recall that UPGMA that each round contains three steps: retrieve the largest similarity—i.e. the smallest value—inside our matrix, create a new subset and update the matrix accordingly, and at last update the dendogram. For now we are not that interested in updating the dendogram, which leaves us with two sub-processes we desire to accelerate using parallelisation.

To find the smallest value from a set of values we can be short of, and should realise that the *less than* operator is commutative; to find the smallest value, we can make use of reduction.[3]

If we take a closer look at the updating process, we can observe the same repetitive calculation. It might thus not be surprising that we are able to translate the merging process into a kernel. A quick glance should reveal that the kernel is data independent for each pair i and j; reading data simultaneously from the similarity matrix will not impact the calculated similarity for the newly created subset. In regards to the weighted vector, we do not really need to accelerate the process to update it. However, due to the involvement of actual hardware, we might want to reduce the communication between host and device by performing the weight update on the device, and thus to translate the update process to a kernel.

Even though the processes within a round do not contain any dependencies, the same cannot be said about the processes among the rounds. Without parallel acceleration, it will take a time complexity of $O(n^3)$ to complete. By accelerating UPGMA using parallel processes, we can improve the complexity to $O(nlog(n))$ time.

In Section 2.4.1 we have encountered a constraint that must be addressed. To make use of GPU acceleration, we must take into account the requirement of fixed size vectors. As we recall, UPGMA requires us to remove two rows and two columns which represent the to-be-merged subsets $\{\sigma_i, \sigma_j\}$, and additionally introduce a new row and column, which represent the merged subset $\sigma_{i\cup j}$. The rather simple solution to this issue is to re-use the row and column of one of the subsets, let us say subset σ_i. For our kernel, we introduce a special condition: if either the column or the row belongs to the subset σ_i, reuse it to calculate the new values for the subset $\sigma_{i\cup j}$, i.e.

$$\text{update-row}\langle x\rangle : \{\sigma_{x,i}, \sigma_{x,j}\} \mapsto \sigma_{x,i}, \text{ such that}$$
$$\sigma_{x,i} = \frac{w_i\sigma_{x,i} + w_j\sigma_{x,j}}{w_i + w_j},\ i < j \tag{2.46}$$

Notice that we could also create a similar kernel to update the column, except now i and j are flipped. As a consequence of re-using a row and a column, the weight w_x that used to represent the weight for σ_x now represents the weight for the new subset. It should come to our attention that this does not matter in the grand scheme of the complete UPGMA process. We should not forget to still remove one row and column. To do so, we assign a value that is larger than the largest value inside our similarity matrix to make sure that these are never chosen again. In practice we assign the value -1 to them and add a condition to our *less than* operator, i.e.

$$\text{delete-row}\langle x\rangle : \{\sigma_{x,i}, \sigma_{x,j}\} \mapsto \sigma_{x,j}, \text{ such that } \sigma_{x,i} = -1,\ i < j \tag{2.47}$$

Example 2.6.2. Let us take the following 5×5 similarity matrix Σ such that we have five linkage sets, each containing only one gene, i.e.

$$R^0: \quad D = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} - & - & - & - & - \\ 90 & - & - & - & - \\ 80 & 70 & - & - & - \\ 60 & \mathbf{40} & 50 & - & - \\ 65 & 75 & 85 & 95 & - \end{bmatrix} \end{array}, \quad w = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{2.48}$$

In our example we find the pair $\{b, d\}$ with the largest similarity, which we highlighted **red**. We actually do exactly the same as with its sequential counterpart. The only difference lies in the fact that we do not resize our matrix. We have decided that the subset with the smallest index will be used to store our result. Now in practice we do not need to alter the elements outside our triangular matrix, however for visual clarity we mark them with $\times$. Let us have a look at how we update the similarity between our pair and the element a. For this example, we will reuse the element $\sigma_{a,b}$. We now only need to retrieve the similarity for $\sigma_{a,d}$. However, we see that it lies outside the triangular matrix. No worries, we simply use $\sigma_{d,a}$. Now, we use our common UPGMA process to calculate the new similarity. We do this for all row and column values associated to $\{b, d\}$, which we will highlight. To exclude the row and column values assigned to the removed ones represented by the subset d, we need to make sure to assign the -1. At the end we update the weights, i.e.

$$R^1: \quad D = \begin{array}{c} \\ a \\ \{b,d\} \\ c \\ \times \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & \{b,d\} & c & \times & e \end{array} \\ \begin{bmatrix} - & - & - & \times & - \\ \mathbf{75} & - & - & \times & - \\ 80 & \mathbf{45} & - & \times & - \\ -1 & -1 & -1 & \times & \times \\ 65 & \mathbf{85} & 85 & -1 & - \end{bmatrix} \end{array}, \quad w = \begin{bmatrix} 1 \\ \mathbf{2} \\ 1 \\ \times \\ 1 \end{bmatrix} \tag{2.49}$$

2.7. Max-Cut

Max-cut is our problem of choice. It is a well known NP-complete problem which is known to be interpretable by EAs. We will first have a look at the formal description in Section 2.7.1. After that, we will have a special look at the dependencies it contains when interpreted by an EA in Section 2.3.

2.7.1. Problem Description

Max-cut is a well known NP-complete problem which describes the partition of a weighted graph into two subsets such that the edges between those subsets is maximised, finding its recurrence in fields such as as network design, statistical physics and VLSI design.[5][1] Formally, the problem is stated as follows:

Given an instance of a weighted graph $G = (V, E)$—such that each edge $e \in E$ has been decorated with a weight $w \in \mathbb{Z}+$, i.e. $e = \{v_i, v_j, w\}$—is there a partition of vertices Π into a disjoint set Π^0 and Π^1 such that the sum of the weights of each edge from E that have one endpoint in Π^0 and the other in Π^1 have been maximised?[5]

The name max-cut is derived by the act of *cutting* the edges between those two partitions, and rating the problem by taking the sum of the weights for those cut edges. Alternatively we can determine the fitness by iterating over all edges and contributing their weight if their endpoints can be found in different partitions, i.e.

$$\varphi : \sum_{\{v_i, v_j, w\} \in E} x \text{ such that } x = \begin{cases} 0, & \text{if } v_i, v_j \in \Pi^0 \\ 0, & \text{if } v_i, v_j \in \Pi^1 \\ w, & \text{otherwise} \end{cases} \tag{2.50}$$

Example 2.7.1. In the following example, we use a graph with five vertices as described in Figure 2.9a.

(a) A cut is registered through the edges $\{b, c\}$ and $\{c, e\}$ visualised by one red line, i.e. the *cut*.

(b) A cut is registered by colouring the vertex c differently from all other vertices.

Figure 2.9: An problem instance graph with five vertices. Two visualisation are realised to display a registered cut realised by the partitions.

Here, we have given each edge its own weight, and we have registered a cut at the edges with weights 9 and 5. Notice that we could visualise the same cut if we colour vertex c differently than any other vertex, which is visualised in Figure 2.9b. This will create two partitions, i.e.

$$\begin{aligned} \Pi^0 &= \{a, b, d, e\} \\ \Pi^1 &= \{c\} \end{aligned} \tag{2.51}$$

The total weight of the registered cut is equal to $9 + 5 = 14$. If we spend enough time, we can see that the optimal solution will have a cut which registers a weight equal to 22, i.e.

$$\begin{aligned} \Pi^0 &= \{a, c\} \\ \Pi^1 &= \{b, d, e\} \end{aligned} \tag{2.52}$$

These partitions we mentioned earlier can be distinguished by colouring the vertices either of two colours ○ or ●, or more practically, 0 or 1. To make life a little bit easier, we create an additional vector C which will keep track of any colour c_i contributed to a vertex v_i, i.e.

$$\begin{aligned} &C \triangleq \text{ Colours} \\ &C = \begin{bmatrix} c_0 & c_1 & \cdots & c_{|I|-1} \end{bmatrix}^\top \text{such that} \\ &c_i = x : v_i \in \Pi^x \end{aligned} \tag{2.53}$$

We can simplify our fitness function by taking into account the colour vector. This makes it relatively easy to translate the max-cut problem to an EA problem instance. By looking from a bird's-eye view, we can see that a colour vertex can be interpreted as a genotype; we can use EAs to mix the genes and to try out different coloured vertices and thus different solutions, i.e.

$$\varphi_{MC} : \sum_{\{v_i, v_j, w\} \in E} (c_i \oplus c_j)\, w_i \tag{2.54}$$

2.7.2. Parallel Dependencies

If we want to obtain the optimal solution using parallel acceleration, we should look into the dependencies which exist within the problem description. Max-cut is a problem which can be described by a single kernel: for each gene represent a vertex, colour it either 0 or 1. It should feel intuitive that colouring vertices is data independent, as long as we do not colour the same vertex.

Example 2.7.2. Let us use the graph again in Figure 2.9b. Now, let us have a look at the dependency of colouring two vertices at the same time. We will try to colour both vertices a and b in parallel such that both switch from partition, i.e.

$$\begin{aligned} \Pi^0 &= \{a, b\} \\ \Pi^1 &= \varnothing \\ \theta_a &\rightsquigarrow \{\Pi^1 = \{a\}\} \\ \theta_b &\rightsquigarrow \{\Pi^1 = \{b\}\} \end{aligned} \tag{2.55}$$

Similarly as what we have been doing in Section 2.3, we can visualise the parallel process $\theta_a \parallel \theta_b$ as a transition diagram, which you can find in Figure 2.10. Unsurprisingly, we should be able to colour two or more vertices in parallel.

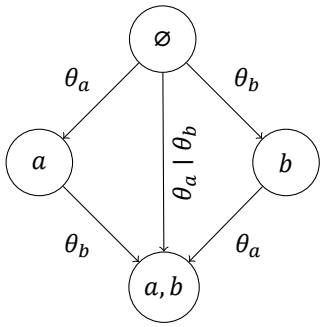


Figure 2.10: The tasks θ_a and θ_b if they're performed before, after, and simultaneously of one another. Each vertex represents the state of the partition Π^1.

On the other hand, on a fast glance we should be able to notice that max-cut contains a multitude of fitness dependencies. By having a look at the fitness function, we see that each contribution is dependent on the colour of two vertices; through the colouring of either endpoints, an edge might be cut or not. In the worst case—which takes the *complete graph*—a change in colour would impact $|E|-1$ fitness contributions. For any connected graph containing more than one edge, we cannot create a marginal product of independent linkage sets.

If we want to avoid any fitness dependencies whatsoever, we have to use a subset of the problem instance. If we decide to alter the partition for a vertex, we cannot alter its *neighbouring* vertices in parallel, as they will change the same fitness contribution. We can however alter a non-neighbouring vertex in parallel. If we take the set of non-neighbouring vertices—which can be more than two—then each and every vertex will be independent of one another. In a sense, we alter the fitness function for each subset such that they only take one variable per contribution. The complete graph is the only problem instance which does not have a set of non-neighbouring vertices.

Example 2.7.3. Let us illustrate the fitness dependencies in respect to max-cut. Let us start off with a line graph containing three vertices, which can be found in Figure 2.11.

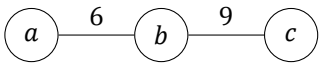


Figure 2.11: A problem instance graph with three vertices.

For our example, there are 8 possible combinations imaginable, however, we should already be able to observe the variable dependency by changing each vertex to the other partition. For our example, we will use two sets: one represents the list of colours, the other represents the list of fitness values as described by Equation (2.54), i.e.

$$\begin{aligned} C &= \{c_a, c_b, c_c\} \\ \delta &= \{\delta_{a,b}, \delta_{b,c}\} \\ \theta_i : C &\rightsquigarrow (C \setminus c_i) \cup \{\neg c_i\} \\ \kappa_{x,y} : \phi_{x,y} &\mapsto (c_x \oplus c_y)\ w \\ \theta' : \delta &\rightsquigarrow \kappa(\delta) \end{aligned} \tag{2.56}$$

Let us start with base state s, and change each of them individually in comparison to this base while we observe their side effects, which are coloured **red**, i.e.

$$s = \{C = \{\mathbf{1}, 0, 0\},\ \delta = \{0, 0\}\} \tag{2.57}$$

$$\begin{aligned} r_1 :&\quad \theta_0 \rightsquigarrow \{C = \{\mathbf{1}, 0, 0\}\} \\ r_2 :&\quad \theta' \rightsquigarrow \{\delta = \{\mathbf{6}, 0\}\} \\ \hline r_1 :&\quad \theta_1 \rightsquigarrow \{C = \{0, \mathbf{1}, 0\}\} \\ r_2 :&\quad \theta' \rightsquigarrow \{\delta = \{\mathbf{6}, \mathbf{9}\}\} \\ \hline r_1 :&\quad \theta_2 \rightsquigarrow \{C = \{0, 0, \mathbf{1}\}\} \\ r_2 :&\quad \theta' \rightsquigarrow \{\delta = \{0, \mathbf{9}\}\} \end{aligned} \tag{2.58}$$

If we observe the fitness contribution $\delta_{a,b}$, we see that it both depends on c_a and c_a, as expected; we say that both are fitness dependent on one another. The same can be said about c_b and c_c. Additionally, We can observe that changing c_b will affect both fitness contributions $\delta_{a,b}$ and $\delta_{b,c}$; both contributions are dependent on c_b.

However, because both contributions are dependent on c_b, that does not mean that c_a and c_c are dependent of each other through c_b. One way to tackle this is by considering c_b to be a constant. Staying constant is similar to saying that we do not insist to run c_b in parallel with the others. If we look at our action θ', we see that there are no variable dependencies in δ anymore. Take for example the constant $c_b = 1$, i.e.

$$\begin{aligned} C &= \{c_a, c_c\} : c_i \in \{0, 1\} \\ \delta &= \{\delta_a, \delta_c\} \\ \theta' : \delta &\rightarrow \{c_a \oplus 1, 1 \oplus c_c\} \end{aligned} \tag{2.59}$$

2.8. Graph Colouring

Graph colouring draws our interest because it allows us to create a subset of non-neighbouring vertices. As we have learned in Section 2.7.2, neighbouring vertices are dependent of one another if we want to perform max-cut. By creating a subset of non-neighbouring vertices we have succeeded in isolating the vertices which are able to be altered in parallel. We should be aware that graph colouring is an NP-hard optimisation problem. However, if we do not desire an optimal solution, we actually are able to get a coloured graph within a doable time using parallelisation.

We will first have a look at the problem description in Section 2.8.1, and finish with a design for parallel accelerated graph colouring in Section 2.8.2.

2.8.1. Problem Description

Graph colouring is a well known NP-hard optimisation problem.[2]. For a instance graph G, it desires the minimum number of required colours such that two neighbouring vertices do not share the same colour. Formally we can describe the problem as follows:

Given an instance graph $G = (V, E)$, what is the minimum set of partitions $\{\Pi^0, \Pi^1, \ldots \Pi^n\}$ given that there are no two adjacent vertices within the same partition?

Graph colouring is interesting to us, as it describes exactly what we need to resolve our max-cut dependency problem; if we are able to colour or graph into multiple partitions, than each partition will consequentially be without any neighbouring vertices and thus will not contain any dependencies. By colouring the graph we have been able to create set of subsets which are dependent of each other, but which elements are not, i.e.

$$\begin{aligned} &\theta(\Pi^0) \nparallel \theta(\Pi^1) \nparallel \ldots \nparallel \theta(\Pi^n) \\ &\theta(\pi_0) \parallel \theta(\pi_1) \parallel \ldots \parallel \theta(\pi_m), \text{ given } p \in \Pi \end{aligned} \tag{2.60}$$

Minimising the number of colours needed to colour a graph is consequentially similar to minimising the number of non-neighbouring subsets. As each subset must be processed in sequential order, the lower the number of colours, the less sequences are needed.

2.8.2. Parallel Acceleration

Ideally we would reduce the number of partitions—and thus the number of sequences—to its minimum requirement. As we have stated earlier, graph colouring is an NP-hard problem. Realistically, we would like to colour the graph adequately fast. Luckily, graph colouring can be accelerated by using parallel processes.[14] If we favour just an approximation instead of an optimisation.

Similar to max-cut, we will use a colour set C such that the colour represents the partition which the vertex is an element of. To realise parallel colouring, we need to understand a simple trick. If we decorate each vector with an uniquely assigned random value, by comparing each vertex with their neighbours random value, we can eventually colour the complete graph, i.e.

$$\begin{aligned} &W \triangleq \text{ Assigned random value} \\ &W = \begin{bmatrix} w_0 & w_1 & \cdots & w_{|V|-1} \end{bmatrix}^\mathsf{T} \text{such that} \\ &\quad w_i \neq w_j : \forall w_j \in W,\ v_i \in V \end{aligned} \tag{2.61}$$

We therefore localised the problem to a single vertex. The idea is as follows. For each vertex, determine whether a vertex contains the largest assigned value among its neighbours. If so, we assign the colour c_0 to it, and remove the vertex from the graph. We repeat this process for multiple rounds r_i where for each round we assign a colour c_i, until we are left with the empty graph. The kernel that we described can be performed in parallel, as it does not contain any dependencies. The requirement to search for the maximum value eliminates any condition which would cause a data dependency. By removing the vertices which have already been coloured there is always another vertex containing the largest value among its neighbours.

There is a downside to this approach. Each kernel contains a different number of vertices to process, this number also changes for each round. In the worst case, a complete graph is provided, which uses a kernel requiring $|V|$ variables. A kernel design which requires an irregular number of variables will become a problem if we implement it on actual hardware, which we will encounter later in Section 2.4.

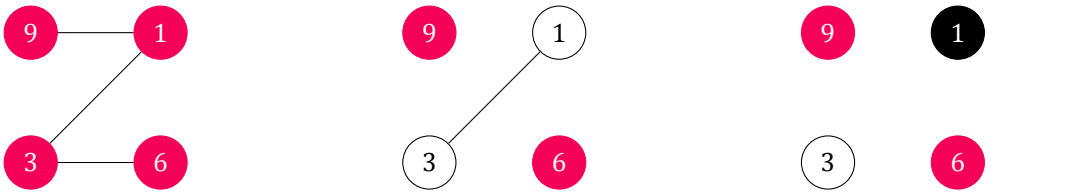
Instead, we could create a kernel which is localised to a single edge. Instead of comparing all neighbours of a vertex, we simply compare the endpoints of an edge. We start each round r_i by assigning the colour c_i to all vertices inside the graph. Instead of colouring the vertex with the maximum assigned value, we remove the colour from a vertex if it is the minimum value of two endpoints. We remove the coloured vertex from our graph and repeat the process until there are no edges left. The last remaining vertices will be assigned by one shared colour. By removing the colour of vertices which are

assigned the lower value of two endpoints, we are left with a set of vectors which contain the maximum value among its neighbours. Notice that we could remove the colour of the same vertex if it is by two or more edges. However, because there is no ability to add a colour, we end up in the same expected state; race conditions do not interfere with our outcome. The kernel can be found in Algorithm 4.[15] In the worst case, we encounter yet another complete graph. In such a case, the optimal solution takes $|V|$ colours. Which for our kernel requires $|V|$ rounds.

Algorithm 4 ParallelColouring$\langle i \rangle(E, W, C, c)$

```
{v_x, v_y} ← e_i ∈ E
if c_x ≠ −1 or c_y ≠ −1 then                ▷ Check if a vertex is disconnected
    return
end if
c_x, c_y ← c                                ▷ Assign the colour at the start
sync all threads
if w_x < w_y then
    c_x ← −1                                ▷ Remove the colour
else
    c_y ← −1
end if
```

Example 2.8.1. Let us have a look at an example to illustrate the graph colouring kernel described in Algorithm 4. We start with a problem instance of four vertices, each connected such that the graph form one line, which can be found in Figure 2.12.



(a) Round r_1: assign the red colour ● to all vertices.

(b) Round r_2: assign the white colour ○ to the remaining vertices.

(c) Round r_3: assign the black colour ● to the last vertex.

Figure 2.12: An example of the graph colouring process using the kernel described in Algorithm 4. The graph contains four vertices, each connected such that it forms one single line. For each round the newly coloured vertex is disconnected from the graph to visualise the removal.

At round r_1 we start off by colouring all vertices the same colour, **red** in our case. This is visualised in Figure 2.12a. For each edge—which there are three of—we determine the smallest value. For our three edges, these values are 1, 1, and 3. Notice that we can remove a colour multiple times, this does not affect the outcome. We are left with the vertices which have the values 6 and 9 assigned to them. We can verify that these are indeed no neighbours of each other. Next, we remove these vertices from our graph. In our example this means disconnecting them from the graph. Our new rounds start of with now just two vertices left. This is visualised in Figure 2.12b. After just two rounds we end up with no edges left; the graph could be coloured using just three colours. The end result can be found in Figure 2.12c.

2.9. Set Packing

Previously in Section 2.1.3, we have discussed two specific cases for the Family of Subsets: the marginal product and the univariate FOS. We would like to introduce another case, which will finds its usage later in Chapter 3. If we want to accelerate the donation of linkage sets by making use of parallisation, we should ensure that each and every linkage set is independent of one another. Luckily, we can recall that the construction of a MP FOS implies the independency between the linkage sets.

Surely, if there there is no overlap between a pair of linkage sets, then we can be assured that we will not encounter any data dependencies.

Now let us revisit the construction of a FOS via GOM, which revolves around the creation of the linkage tree. By definition, each neighbouring vertex overlaps each other; as a result, we cannot construct an MP FOS containing all linkage sets inside our tree. The best we can do, is to collect the linkage sets into their own, disjoint partitions. As a result, each partition is a set of independent linkage sets, while the partitions among themselves are not. Notice that each of these partitions does not have to contain all genes, but ideally we would like them to.

Let us introduce the *set packing* problem.[11] Set packing is an NP-complete problem described by the requirement to find a partition Π which contains the maximum number of disjoint subsets among an Family of Subsets. Here, the partition is also known as a *package*, i.e.

> Given a base set I, and the Family of Subsets $\mathcal{F}$, find a partition $\Pi \subseteq \mathcal{F}$ of disjoint sets such that the cardinality is maximised.

To find an approximation, set packing requires a time complexity equal to $O(\sqrt{|I|})$.[11] Set packing and the *maximum independent set* are shown to be mutually reducible. Therefore, if we want to find the least amount of packagesneeded for a FOS, we could alternatively solve the graph colouring problem.

In previous sections we already added the emphasis of using a vector instead of a list; we would like to do the same for packages. Instead of having sets of linkage sets, we would like to contribute a linkage set inside our package to each gene. For practical reasons which we later discuss in Chapter 3, we would like to keep the vector to the same size as the individual. Notice that this is a valid upper bound, as we cannot have a better package than the MP FOS. This vector representation however implies that we should also have to assign a value for when a gene is not included to the partition, i.e.

$$\begin{aligned} &\mathcal{P} \triangleq \text{ Package} \\ &\mathcal{P} = \begin{bmatrix} \mathcal{p}_0 & \mathcal{p}_1 & \cdots & \mathcal{p}_{|I|-1} \end{bmatrix}, \text{ such that} \\ &\mathcal{p}_i = \begin{cases} j, & \text{if } g_i \in f_j \\ -1, & \text{otherwise} \end{cases} : g_i \in \Pi \end{aligned} \tag{2.62}$$

Example 2.9.1. Lets explore this definition by using an example. Let us take an FOS with intersecting linkage sets, i.e.

$$\mathcal{F} = \{\{0,1\},\{0,1,2\},\{1,2,3\},\{2\}\} \tag{2.63}$$

By definition we can conclude that this FOS is not an MP, and if we want to create a partition of independent linkage sets, we have no other option than to use a package. For our example, we can create a package from the linkage sets f_0 and f_3. Notice that, to express all linkage sets packages, we are required to construct at least three of them.

To translate the package as a vector, we will use the index of the linkage sets inside the FOS as the value inside the vector; the gene g_2 inside f_3 will map the index 3 to the element $\mathcal{p}_2$. For all genes which are not present inside the package, we assign the value -1 to it, i.e.

$$\begin{aligned} &\mathcal{P} = \{f_0, f_3\} \\ &\mathcal{P} = \{\{0,1\},\{2\}\} \\ &\mathcal{P} = \begin{bmatrix} 0 & 0 & 3 & -1 \end{bmatrix} \end{aligned} \tag{2.64}$$

3

GPU Accelerated GOMEA

Prepared with the required background knowledge, we are now able to have a first look at the core design choices behind the design of GPU accelerated GOMEA for discrete values, which we will simply shorten to GPU-GOMEA. The core principle behind designing GPU acceleration revolves around the isolation of kernels. The use of an iterative loop is a major indication for possible acceleration; if the process within the body of the loop is data independent of one another, the process could be translated to a kernel.

If we have a look at GOM—as described in Algorithm 2—we observe the design of the main iterative loop; to perform optimal mixing, we need to iterate over all linkage sets inside the FOS. If we expand the scope to the complete variation process, then we know that we have to perform an additional iterative loop, this time over all individuals. The complete variation process can be found in Algorithm 5. Do note here that we have explicitly stated the sub-processes for *donations* and *fitness evaluations*, which will prove to be useful later. Deceivingly, we could conclude that our variation process requires one kernel—which we translate to a $n \times |\mathcal{F}|$ matrix—such that each task will process GOM in respect to a single individual and linkage set, i.e.

$$\text{gom}\langle i, j\rangle : \{p_i, \phi_i, f_j\} \mapsto \{o_i, \phi_i\} \tag{3.1}$$

Algorithm 5 Variation($P, \mathcal{F}, \Phi$)

```
O ← ∅
for each p ∈ Shuffle(P) do
    o ← p
    for each f ∈ Shuffle(F) do
        o ← Donation(o, f, P)
        {o, φ_i} ← FitnessEval(o, p, φ_i)
    end for
    o_i ← o
end for
return O
```

There are two problems related to this approach. GPU acceleration mainly benefits if we can perform simple arithmetic per thread, which only requires a few intermediate variables. Recall from Section 2.4.2 that the moment we need more than a few variables—like a set—we are doomed to use global memory, which will diminish every form of acceleration we might want to achieve. This poses as the first problem. What this means, is that we should not create a kernel which takes a set of genes and uses another set of donations to create yet another set of genes to represent the offspring.

Following this, our approach makes a significant assumption, which states that GOM will not encounter any dependencies. Previously in Section 2.1 however, we have discussed that by definition our linkage tree will not be data independent. This would prove to be our second problem. The best we can do to prevent this from happening, is to construct sets of packages. Additionally, in Section 2.7.2

we have discovered that max-cut contains fitness dependencies. Luckily in Section 2.8, we have found that we are able to construct sets of genes in such a way that we could avoid these dependencies.

Informed with the deceiving nature of our assignment, we will have a closer look at the restrictions at play and how we are able to resolve them. The complete implementation will be split into two chapters: the current chapter will discuss the core design choices revolved around the complete variation process, while the latter chapter will elaborate on the construction of the packages. Recall from our description for the complete variation process—found in Algorithm 5—that we are tasked to implement two sub-processes. To start off, in Section 3.1 we will discuss how we would implement the donation process using GPU acceleration. Later, in Section 3.2 we will discuss the intermediate fitness evaluation process.

Before we continue, due to the dependencies present at GOM, we would like to introduce an additional process which enforces the elitism of an individual; we need to make sure that our child is at least equal or better than the parent. Max-cut is one of those problems in which dependencies could result into a child being less fit than the parent. If we perform GPU acceleration to generate the offspring, at the end of it all, we compare each individual with the original population, only keeping those with the largest fitness. We will close off this chapter by having a closer look at the implementation for the elitist evaluation process in Section 3.3.

3.1. Parallel Donation

Instead of creating a kernel for each individual and linkage set, we will take another approach, one which favours the acceleration of the donations itself. What we know is that we should be able to donate each gene independently, as long as the linkage sets we want to use, are disjoint of each other. Thus, we should be able to perform parallel donations for a given package. Ideally, we only need one package, i.e. we have found an MP FOS for our problem instance. In such a case, we can reduce the donation process from $O(\ell^2)$ time complexity to a stunning *optimistic* $O(1)$ time. We say optimistic, because we do not consider any overhead caused by the use of GPGPU such as the communication between the device and the host among other things.

Now, let us focus on the kernel itself. Recall from Algorithm 2 that we are tasked to randomly pick a donor and to apply the respective gene. For explanatory reasons, we will store the donated genes into a $n \times \ell$ matrix D. In practice, to reduce the required memory, will use the offspring—which shares the same dimensions—for this purpose. To randomly assign a new donor to each gene, we generate a $n \times \ell$ matrix R such that each gene $p_{i,j}$ has a corresponding donor $r_{i,j}$. As condition, each random variable must be an index for genes inside our population, i.e.

$$\begin{aligned} R &\triangleq \text{Randomised matrix} \\ D &\triangleq \text{Donation matrix} \end{aligned} \tag{3.2}$$

$$\begin{aligned} D &= \begin{bmatrix} d_{0,0} & d_{0,1} & \cdots & d_{0,\ell-1} \\ d_{1,0} & d_{1,1} & \cdots & d_{1,\ell-1} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n-1,0} & d_{n-1,1} & \cdots & d_{n-1,\ell-1} \end{bmatrix} : d \in \{0, 1\} \\ R &= \begin{bmatrix} r_{0,0} & r_{0,1} & \cdots & r_{0,\ell-1} \\ r_{1,0} & r_{1,1} & \cdots & r_{1,\ell-1} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n-1,0} & r_{n-1,1} & \cdots & r_{n-1,\ell-1} \end{bmatrix} : r \in \{0, 1, ..., \ell\} \end{aligned} \tag{3.3}$$

By constructing the random matrix, we have taken care of the scenario where each and every gene requires their own donor, i.e. if we have a univariate FOS. But what about other FOS constructions? To make sure that the genes of the same linkage set also use the same donor, we use the corresponding linkage set. This is described in our package as a column index inside our random matrix. This way, if genes are described to be in the same linkage set, by definition, they share the same value inside the package, and thus would point to the same value inside the random matrix, i.e.

$$\text{opti-mix}\langle i, j\rangle : p_{i,j} \mapsto d_{i,j}, \text{ such that } d_{i,j} = r_{i,k},\ k = \mathscr{p}_j \tag{3.4}$$

Example 3.1.1. Using an example, we shall next explore the kernel. Let us use a population with three individuals, a predetermined package, and random matrix. For demonstrative purposes, we use a genotype that solely uses symbols a, b, and c, i.e.

$$\begin{aligned} P &= \begin{bmatrix} a & a & a & a \\ b & b & b & b \\ c & c & c & c \end{bmatrix} \\ R &= \begin{bmatrix} 1 & 2 & 0 & 1 \\ 0 & 0 & 2 & 2 \\ 2 & 1 & 1 & 0 \end{bmatrix} \\ \mathcal{P} &= \begin{bmatrix} 0 & 1 & 0 & 2 \end{bmatrix} \end{aligned} \tag{3.5}$$

Now, let us try to determine the donated gene $d_{1,2}$. Simply put, for the individual p_1 we want to know what the gene p_2 will receive as a donation. We first have to look at the corresponding linkage set described by the package. This reveals that our gene is part of the linkage set $\mathcal{p}_2 = 0$. Second, we will use this value as the column index for our random matrix R, which will reveal our donor $r_{1,0} = 0$. At last, we now are able to determine the donated gene $d_{1,2} = p_{0,2} = a$. In red we have highlighted the involved values for our example. To generate the rest of the donation matrix, we simply use the same process for all genes, i.e.

$$D = \begin{bmatrix} b & c & b & a \\ a & a & a & c \\ c & b & c & b \end{bmatrix} \tag{3.6}$$

Notice that, if we do not perform any fintess evaluation—that is, if we want to keep all the donated genes—we can interpret our donation matrix as our offspring.

3.2. Parallel Fitness Evaluation

With just having accelerated the donation process, we have not quite finished yet. If we recall Algorithm 2, we know that a major part of optimal mixing involves the intermediate fitness evaluation of the donated genes. For ease of discussion, let us focus on one individual. Ideally, we would like to evaluate each and every linkage set in parallel. However, due the construction of our linkage tree, we cannot donate genes in parallel for all linkage sets, and thus we are left with using packages. For each package, it is unclear how many linkage set it contains, and how many genes each linkage set describes. GPU acceleration favours the use of fixed vectors, and thus, we are left with a non-ideal situation. Instead of evaluating each linkage set, we could alternatively evaluate each gene instead. What this means is that each gene $p_{i,j}$ will have a fitness value $\phi'_{i,j}$ which takes into account all donations described by the same linkage set, i.e.

$$\begin{aligned} \Phi' &= \text{ Intermediate fitness matrix} \\ \Phi' &= \begin{bmatrix} \phi'_{0,0} & \phi'_{0,1} & \cdots & \phi'_{0,\ell-1} \\ \phi'_{1,0} & \phi'_{1,1} & \cdots & \phi'_{1,\ell-1} \\ \vdots & \vdots & \ddots & \vdots \\ \phi'_{n-1,0} & \phi'_{n-1,1} & \cdots & \phi'_{n-1,\ell-1} \end{bmatrix} \end{aligned} \tag{3.7}$$

In the best case scenario, our package describes a univariate FOS, and thus, we already intended to have ℓ different fitness values for each individual. On the other hand, in the worst case scenario, our package is described by the complete set, and thus, we calculate ℓ fitness values while we could have complied by just calculating it once. Notice however, if we could accelerate our intermediate fitness evaluation for each gene, then optimistically the time complexity is equal to the time complexity of calculating the fitness value only once.

Even if we evaluate each gene separately, we still desire our linkage set perspective; if we reject a donor gene for a given linkage, we should expect that all donor genes within the same linkage set are rejected. This can only happen if we make sure that each gene within the same linkage set represents the same *intermediate individual*.

To construct this intermediate individual, we use both the information from the population P and the donor D matrices. Because all information is already available, we do not have to construct the actual individuals as a three-dimensional $n \times \ell \times \ell$ matrix Y. Instead we can interpret it while we calculate the fitness value provided by our problem instance. This way, we are able to save some memory on the host. For illustrative purposes however, we keep our assumption that the intermediate individuals exist as their own matrix Y i.e.

$$\begin{aligned} Y &\triangleq \text{ Intermediate matrix} \\ y_{i,j} &\triangleq \text{ Intermediate Individual} \\ y_{i,j} &= \begin{bmatrix} y_{i,j,0} & y_{i,j,1} & \cdots & y_{i,j,\ell-1} \end{bmatrix}^\top : y \in \{0,1\} \end{aligned} \tag{3.8}$$

For each gene within the intermediate individual $y_{i,j}$, we determine whether we use the donor or the original individual by looking at the package. As each fitness value $\phi_{i,j}$ corresponds to the gene g_j inside an individual p_i, so does each intermediate individual. Therefore, each intermediate individual is mapped to the linkage set $\mathcal{p}_j$ by the gene it represents. To construct the intermediate individual, if the linkage set $\mathcal{p}_k$ for a gene $y_{i,j,k}$ matches that of $\mathcal{p}_j$, then the gene is equal to the donor $d_{i,k}$, otherwise we will use the gene from the parent, i.e.

$$\begin{aligned} &\text{inter-ind}\langle i,j,k\rangle : \{d_{i,k}, p_{i,k}\} \mapsto y_{i,j,k}, \text{ such that} \\ &y_{i,j,k} = \begin{cases} d_{i,k}, & \text{if } \mathcal{p}_j = \mathcal{p}_k \\ p_{i,k}, & \text{otherwise} \end{cases} \end{aligned} \tag{3.9}$$

By using the kernel, we can optimistically construct the intermediate matrix in $O(1)$ time. With it, we can calculate the fitness for each and every gene inside our donor matrix D. Calculating the fitness however depends on the problem description. Recall in Equation (2.50) that for our problem of interest, the fitness function is described by a summation over a list of weighted edges. It so happens to be that, by making use of reduction—which we discussed in Section 2.4.3—we can optimistically accelerate our fitness calculation to a time complexity of $O(\log(|E|)$.

Example 3.2.1. Let us have a look at an example to illustrate the intermediate fitness calculation process. First, we are in need of a fitness function. In our example, we will establish a pattern T which must be matched. For each correct symbol, we increase the fitness by 1; if all characters are matched, we will have antimal fitness equal to $\phi = \ell$, i.e.

$$\begin{aligned} T &\triangleq \text{ target pattern} \\ T &= \begin{bmatrix} T_0 & T_1 & \cdots & T_\ell \end{bmatrix}^\top \\ \phi &: \sum_{I_i \in I} x, \text{ such that } x = \begin{cases} 1, & \text{if } T_i = I_i \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{3.10}$$

With our fitness function defined, we consider the parameters we need. For our population, we will use one parent containing four symbols. As a result, we will have a similarly sized child, but more importantly, we will have a $\ell \times \ell$ matrix for our intermediate individuals, which should be a bit more visually comprehensible than a three-dimensional matrix. For our example, we will disregard the process behind the creation of the individual, donor vector, and the package; we simply assume that we have them, i.e.

$$\begin{aligned} T &= \begin{bmatrix} b & a & b & c \end{bmatrix}^\top \\ \mathcal{P} &= \begin{bmatrix} 0 & 1 & 2 & 0 \end{bmatrix}^\top \\ P &= \begin{bmatrix} a & a & a & a \end{bmatrix}^\top \\ D &= \begin{bmatrix} b & b & c & c \end{bmatrix}^\top \end{aligned} \tag{3.11}$$

Let us focus on the first intermediate individual y_0 which represents the gene p_0. By taking a look at the package, we see that our linkage set is equal to $\mathcal{p}_0 = 0$. Notice that this

linkage set contains two genes, i.e. $f_0 = \{0, 3\}$. We thus should expect the same intermediate individual between the genes p_0 and p_3. We next generate our intermediate individual y_0. For the first gene $y_{0,0}$ inside our intermediate individual, it should be obvious that we use the same linkage set, i.e. $y_{0,0} = b$. Now, for $y_{0,1}$, we will look at the corresponding linkage set for the gene p_0 and see if it matches to that of our gene p_1. We quickly conclude that $\mathcal{p}_0 \neq \mathcal{p}_1 \Leftrightarrow 0 \neq 1$, and thus we take the gene from the individual instead. We will highlight each gene inside the first intermediate individual in red, and we will repeat the same process for all other intermediates. With all our intermediates in place, we could calculate the fitness for each of them, i.e.

$$\theta_{\text{inter-ind}}: \quad P = \begin{bmatrix} a \\ a \\ a \\ a \end{bmatrix}, \; D = \begin{bmatrix} b \\ b \\ c \\ c \end{bmatrix} \; \rightsquigarrow \; Y = \begin{bmatrix} b & a & a & c \\ a & b & a & a \\ a & a & c & a \\ b & a & a & c \end{bmatrix}, \; \phi' = \begin{bmatrix} 2 \\ 0 \\ 1 \\ 2 \end{bmatrix} \tag{3.12}$$

Notice how we get the same individuals at y_0 and $glsy[_3]$, as we did expect. At last, we finalise by accelerating the actual evaluation part. With the fitness for each donated gene determined, we now make a decision: if the fitness is better than the parent, we keep the donor, otherwise, we repeat the process. Optimistically, this process takes $O(1)$ time to complete, i.e.

$$\begin{gathered} \text{inter-eval}\langle i, j \rangle : \{d_{i,j}, p_{i,j}\} \mapsto o_{i,j}, \text{ such that} \\ o_{i,j} = \begin{cases} d_{i,j}, & \text{if } \phi'_{i,j} \geq \phi_i \\ p_{i,j}, & \text{otherwise} \end{cases} \end{gathered} \tag{3.13}$$

Example 3.2.2. Let us round off with yet another example in continuation of our previous example. Bring back to mind that we have found the fitness for each gene inside our donor matrix. We should also remind ourselves to calculate the fitness ϕ, which corresponds to our individual, i.e.

$$\begin{aligned} \phi &= 1 \\ \Phi' &= \begin{bmatrix} 2 & 1 & 0 & 2 \end{bmatrix}^\top \\ P &= \begin{bmatrix} a & a & a & a \end{bmatrix}^\top \\ D &= \begin{bmatrix} b & b & c & c \end{bmatrix}^\top \end{aligned} \tag{3.14}$$

For each of the genes inside our donor matrix, we will evaluate if we wish to keep it or not. First, let us have a look at d_0, we see that the fitness related to this gene is larger than that of the original individual, i.e. $\phi'_0 > \phi \Leftrightarrow 2 > 1$, thus, we map our donor gene to the corresponding gene o_0 inside our offspring. We can continue this for each gene until we have mapped the complete offspring. What we observe is that for our newly created offspring, the fitness is greater than the previous individual, i.e.

$$O = \begin{bmatrix} b & a & a & c \end{bmatrix}^\top, \; \phi = 2 \tag{3.15}$$

3.3. Elitist Evaluation

At the start of this chapter, we have made the announcement that optimal mixing could contain fitness dependencies which—as the name suggests—are fully the result of the fitness function provided by the problem during the fitness evaluation. As a result, it is possible that—at the end of the complete variation process—a child has a smaller fitness than the original parent. To combat this, we perform an additional evaluation, which we call *elitist evaluation*. This process is nothing more than the re-evaluation of the offspring: if an individual inside the population has a larger fitness in comparison to its child, the complete individual will be recovered, i.e.

$$\text{elite-eval}\langle i,j\rangle : p_{i,j} \mapsto o_{i,j}, \text{ such that}$$
$$o_{i,j} = \begin{cases} p_{i,j}, & \text{if } \phi_i^{glsp} > \phi_i^{glsp} \\ o_{i,j}, & \text{otherwise} \end{cases} \tag{3.16}$$

By executing all kernels in sequence, we have been able to accelerate the complete variation process. This whole process will be bounded by the complexity of the fitness function. In our case with the max-cut problem we have seen that the fitness function will have an optimistic time complexity equal to $O(\log(\ell))$.

4

Packagers

To construct a package, we need a *packager*. A packager represents an entity which creates and delivers the packages. A packager is therefore responsible for the learning process at the start of a generational step, and for the delivery of new packages at the start of optimal mixing. Because the packager keeps track of the number of created and delivered packages, it also is responsible for determining the end of a generational step. By far the simplest packager is one that creates the univariate FOS package; there is no need to learn a new package based on the provided population, and by providing only one package, each generational step only performs optimal mixing once.

In regards to discrete-GOMEA, we intend to create a packager such that the linkage tree is packed into multiple different packages. In Section 4.1 we will introduce our first attempt of such a packager. We will mainly focus on the translation from a linkage tree to a set of packages. In Section 4.2 we will elaborate on our findings by incorporating our understanding of fitness dependency within max-cut. We will combine our background knowledge to construct a new type of graph such that it represents the dependencies among the linkage set. At last in Section 4.3, we will discuss an alternative approach which similarly will exploit our understanding of fitness dependencies, however this time without the use of a linkage tree.

Each section is constructed such that it contains at least one subsection to discuss the packing of the packages themselves, and one subsection to introduce a parameter for each of the packages known as the μ-variable.

4.1. Revision

Revision is our first attempt to accelerate discrete-GOMEA in its fullest, beyond the scope of the variation process. Revision encapsulates the sub-processes to create a package by learning the mutual information from the population to create a linkage tree, which should match with the one created by discrete-GOMEA. This will include a GPU accelerated process to construct the similarity matrix and to generate the linkage sets using UPGMA. Back in Section 2.5 and Section 2.6 we have discussed both the solutions for designing a parallel process to construct a similarity matrix and to create a linkage tree respectively. With that out of the way, we are able to move on and have a look at the construction of the package in Section 4.1.1, which will complete our design for revision by taking a closer look at the design of the packing process. In Section 4.1.2 we will introduce the μ-variable relevant for revision, and introduce three different variations: *square-root*, *exponential*, and *logarithmic*.

4.1.1. Packing

There are many ways to encode a linkage tree into a set of packages. One simple way is to create a snapshot each time a vertex is created, i.e. each time a linkage set is merged. These snapshots describe the subsets which are still available to be merged for that round. We will call these snapshots *revisions*, hence the name of the packager. If we lay out the revisions in the proper historical order, we should realise that from the information alone we should be able to construct a linkage tree, and vice versa. Notice that we do not care about the height information described by the dendogram, as this information is not present in the linkage tree. We could however store it alongside our revision if

we wanted to. We can represent this revision list as a $(n-1) \times \ell$ revision matrix, in which each row i represents a revision at round r_i, i.e.

$$\begin{aligned} \mathcal{P}^r &\triangleq \text{ Revision list} \\ \mathcal{P}^r &= \begin{bmatrix} \mathcal{p}_0^r & \mathcal{p}_1^r & \cdots & \mathcal{p}_{n-1}^r \end{bmatrix}^\mathsf{T} \end{aligned} \tag{4.1}$$

To generate the revision list, we expand our UPGMA process such that we are able to create a new revision at the end of each round. We initialise our revision matrix by generating the first revision $\mathcal{p}_0^r$, which will be equal to the univariate FOS. Afterwards, at the end of each round r_x—in which we have merged the two linkage sets f_i and f_j—we want the merge to be reflected in our revision list. Way back in Section 2.6.2 we made the decision to reuse the row and column occupied by i if $i < j$; we would like to reflect this in our revision. To create the revision $\mathcal{p}_{x+1}^r$ we copy the data from the previous round $\mathcal{p}_x^r$ unless the linkage set at $\mathcal{p}_{x,y}^r$ matches the linkage set at $\mathcal{p}_{x,j}^r$. In such a case we will use the linkage set described at $\mathcal{p}_{x,i}^r$, i.e.

$$\begin{aligned} &\text{revision}\langle y \rangle : \mathcal{p}_{x,y}^r \mapsto \mathcal{p}_{x+1,y}^r, \text{ such that} \\ &\mathcal{p}_{x+1}^r = \begin{cases} \mathcal{p}_{x,i}^r, & \text{if } \mathcal{p}_{x,y}^r = \mathcal{p}_{x,j}^r \\ \mathcal{p}_{x,y}^r, & \text{otherwise} \end{cases} \end{aligned} \tag{4.2}$$

Notice that we can easily accelerate this process by parallelisation for each index y as there are no dependencies involved. With the revision list generated, we are able to pick a packing from the list and use it at the generational step in our GPU-GOMEA. Similarly to what we do for discrete-GOMEA, we shuffle our list, iterate over all packages until we have used them all once.

Example 4.1.1. Let us start our example by using a population containing individuals with 5 genes. By an unspecified process we can generate a dendrogram as seen in Figure 2.7.

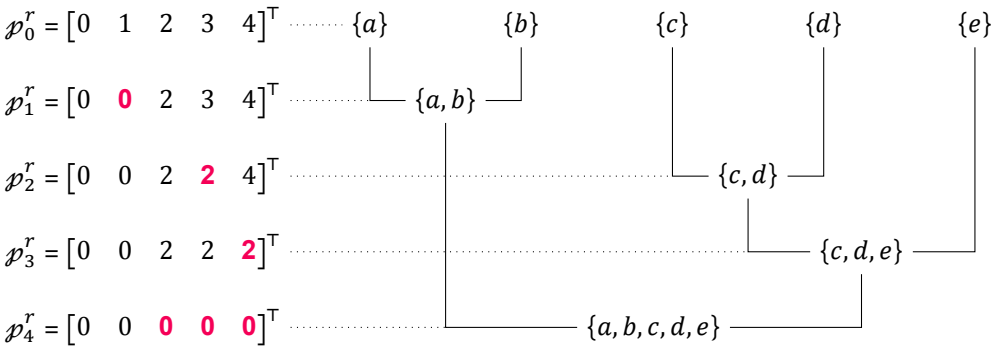


Figure 4.1: A dendrogram generated by some unknown process. For each merge there is a package associated with it. The combination of all packages creates the revision list.

Let us have a look at the first merge. This first merge will represent our second revision $\mathcal{p}_1^r$, as our first revision is simply the initial univariate FOS. For our first round we have merged the linkage sets $\{a\}$ and $\{b\}$ together. We could assign a new value to our linkage set—for instance f_5—but instead we will reuse the index for the smallest linkage set, in our case f_0.

First, we will have a look at the linkage set index associated with the subset $\{b\}$, which is equal to $f_1 = 1$. Second, we will copy all linkage set indices from the previous round, except if they match the index 1. At last, for those matched cases we replace the value with $f_0 = 0$ i.e. $\mathcal{p}_{1,1}^r = 0$. From a visual perspective it might seem less trivial to find which subset belongs to which linkage set index, but with the similarity matrix in mind we have a clear indicator; the smallest value at $\sigma_{i,j}$ represents the merge of linkage set indices i and j. The revision for each round is visualised in Figure 2.7.

4.1.2. μ-Variable and Compression

With revision there are some obvious variations possible. A common question to pose is whether we would favour the complete revision list, or just a cut-off. The μ-variable represents a scalar expressed as a percentage to the complete length of the revision list, i.e.

$$|\mathcal{P}^r| = (1 - \frac{\mu}{100}) \cdot (\ell - 1) + 1, \text{ given } 0 \leq \mu \leq 100, \mu \in \mathbb{Z}^+ \tag{4.3}$$

The reasoning behind it relies on the impact of the larger linkage sets. A linkage set containing $\ell - 1$ elements will consequentially allow for the donation of $\ell - 1$ genes from just a single donor. As a result, even if the donation has a slightly better fitness, all genes of the parent will be replaced by the donor. This reduces the diversity within the population, and potentially removes values from the gene pool. If such a value might have been part of the optimal solution, it is because it is impossible for the population to reach such solution. Increasing the μ-variable provides a tool to prevent such large linkage sets to occur.

Besides the μ-variable, we also could discard some revisions from our list. The logic behind this is that we actually get quite a lot of duplicate linkage sets inside our list. This is to be expected, however, we might want to reduce the number of repetitions of similar linkage sets, as those might cause problematic issues. First, if a linkage set is considered *weak*—that is, if it does not improve the average fitness by any large margins or at all—we will unnecessarily perform a generational step knowing we will not get any closer to the optimal solution; the fitness is stalling. Second, if a linkage set is considered *strong*, we could lose diversity and in the worst case lose a gene value from the pool. By discarding some elements inside the revision list, we can prevent these issues, however by doing so, we might lose some important information. In the worst case, we provide the maximum value accepted for the μ-variable, which will leave us with only one package: the univariate FOS. We would like to *compress* our revision list in such a way that we can find the balance between keeping most information, while reducing the number of revisions.

We will apply a rather simple approach to compress our revision list: we will shuffle the list and take only the first c number of revisions. The difference between compression and the μ-variable is that the μ-variable takes effect before the construction of the revision list, while compression takes effect after shuffling the revision list. As we might expect, there are multiple flavours to our approach, however we will consider only three of them: experimental, logarithmic and square root, i.e.

$$\begin{aligned} \text{exponential}: &\ c = e^{\ell} \\ \text{logarithmic}: &\ c = \log \ell \\ \text{square-root}: &\ c = \sqrt{\ell} \end{aligned} \tag{4.4}$$

All three of them try to provide a slightly different interpretation of the dendrogram structure. If we assume that the dendrogram will look like a binary-tree, that would move us to expect the depth to be equal to $\log \ell$. In such a case we might only favour one linkage set for each depth layer inside the binary tree. We will find out later in Chapter 5 whether this method proves to be sound or not.

4.2. Association

In Section 4.1 we have discussed the use of revisions to interpret the linkage tree in a way that complies with our requirement for packages. However, this still does not resolve the main issue we have been facing regarding max-cut: How do we prevent fitness dependencies from occurring? If we recall from Section 2.8, there is a way to create a set of partitions of vertices such that each partition does not contain any neighbours. By using graph colouring, we are able to isolate neighbouring vertices in one and the same partition. If we match our linkage sets with these partitions, then each pair of genes which are in different partitions are independent of one another.

We shall first discuss what it means for a linkage set to be independent of one another. Recall that even though the donations are processed in parallel, it is the linkage sets which will be evaluated in parallel, not the donations; within a linkage set, information is shared such that we observe the expected intermediate individuals among our donated genes. This means that if we construct a linkage set, then the fitness dependencies among those donated genes inside the linkage set are deemed irrelevant. If the genes which represent neighbouring vertices are part of one linkage set, we do not have any

desire to break it up into multiple subsets, as we will nullify its whole purpose by doing so; a linkage set describes linkage, and linkage should be preserved, as that is what GOMEA is all about.

Graph colouring can be used to create partitions of non-neighbouring vertices. If we apply graph colouring on the original problem graph, we see that we do get independent vertices, but we potentially break up our linkage sets. What we want is not applying graph colouring to our problem instance, but to our linkage tree.

Example 4.2.1. The following example illustrates a broken linkage set. First, let us use a problem graph using five vertices as visualised in Figure 4.2. What we notice, is that we need at least three colours. Additionally, let us assume that we constructed a linkage tree as visualised in Figure 4.1.

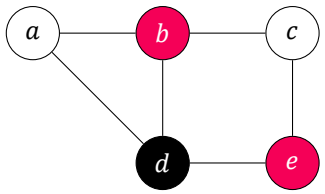


Figure 4.2: A graph with five vertices, each coloured such that they do not share the same colour with one of their neighbours; the minimum of three colours can be achieved for this graph.

Each colour represents a subset of independent non-neighbouring vertices. If we run those subsets in different packages, we will prevent fitness dependencies between genes which represent neighbouring vertices. However, as a result, we break up our linkage sets. For our example, the linkage set $f_7 = \{c, d, e\}$ will be broken up into three packages, i.e.

$$\begin{aligned} \mathcal{P}_\circ &= \begin{bmatrix} - & - & \textcircled{7} & - & - \end{bmatrix}^\mathsf{T} \\ \mathcal{P}_\bullet &= \begin{bmatrix} - & - & - & 7 & - \end{bmatrix}^\mathsf{T} \\ \mathcal{P}_\bullet &= \begin{bmatrix} - & - & - & - & 7 \end{bmatrix}^\mathsf{T} \end{aligned} \tag{4.5}$$

As a result, we got even more packages, where we even have obliterated the linkage between the genes within our linkage sets. Not a good sign all around. We must look at another solution to apply graph colouring while keeping our linkage sets intact.

If we are able to construct a graph such that each edge represents a dependency between two linkage sets, we can make use of a graph colouring which colours represent a package of independent linkage sets. To construct such a *independent linkage graph* we need to extend the design we discussed at the previous section in Section 4.1. The packager responsible for all of this, we will call *Association*. First in Section 4.2.1, we will discuss the creation of the independent linkage graphs. In Section 4.2.2 we will discuss the packing of the linkage sets by making use of graph colouring on the previously mentioned graph. And at last in Section 4.2.3 we will wrap up by introducing the μ-variable for Association.

4.2.1. Independent Linkage Graph

To achieve graph colouring on our linkage tree, we first must create a new graph which we can apply graph colouring upon. The desire is that we construct an edge between linkage sets, solely if the linkage sets are interdependent. We call this new graph the *Independent Linkage Graph* (ILG), i.e.

$$\begin{aligned} G' = (E', V') &\triangleq \text{ ILG} \\ V' &= \{f_0, f_1, ..., f_(2\ell - 1)\} \end{aligned} \tag{4.6}$$

As we construct our linkage tree in rounds, we can expand each round to simultaneously construct our ILG. For each newly created linkage set we will append it to our ILG, find the dependencies, and create the corresponding edges. The linkage set depends on those it intersects with, but also those which contain genes representing neighbouring vertices in regards to their own genes, i.e.

$$
\begin{aligned}
E' &= E^i \cup E^n \\
E^i &= \{e^i \mid \forall e^i = \{f_i, f_j\} : \exists u \in f_i,\ \exists v \in f_j,\ \{u, v\} \in E\} \\
E^n &= \{e^n \mid \forall e^i = \{f_i, f_j\} : \exists u \in f_i, f_j\}
\end{aligned} \tag{4.7}
$$

First, start off with the problem instance, which already should comply with our description for the edges. Note that we will label each vertex with the corresponding gene; we now have initialised our ILG with a set of singletons, each representing a single gene. Second, at the end of each round r_i, we create a new vertex which represents the newly created linkage set $f_{(i+\ell)}$, and add an edge between it and a singleton, which also happens to intersect with our linkage set.

Next, we need to create an edge between it, the remaining intersecting subsets, and the subsets with genes representing neighbouring vertices in respect to their own genes. Instead of iterating over all genes inside all subsets to create the edges, we can apply a simple trick and just look at the neighbours of the intersecting singletons. By definition, another subset is connected to a singleton for the same reasons our newly created linkage set is; it either shares the same gene through the singleton, or it contains a neighbouring vertex in respect to the singleton. At last, we only need to make sure that we do not add duplicates of the same edge. The process can be found in Algorithm 6.

Algorithm 6 AddLinkageSet(G', f_i)

$V' \leftarrow V' \cup \{f_i\}$
for each $j \in f_i$ **do**
  $u \leftarrow v' : v' = \{j\}, v' \in V'$
  $E' \leftarrow E' \cup \{\{u, f_i\}\}$
  $E'_j \leftarrow$ all edges connected to j
  for each $\{j, k\} \in E'_j$ **do**
    $E' \leftarrow E' \cup \{\{k, f_i\}\}$
  end for
end for

Notice that beforehand, we cannot determine the number of vertices needed for our ILG. These kind of flexible lists are easily achieved using sequential processes, but are harder to achieve if we want to accelerate it by making use of parallelisation with GPGPU. In contrast to revision, ILG is partially realised by a sequential process on the host rather than the device.

Example 4.2.2. Let us have a look at how we could construct such an ILG. For our example we will be using the problem instance described in Figure 4.2 and the linkage tree described in Figure 4.1. To start off, we initialise our ILG using the problem instance; instead of describing each vertex as is, we wrap the associated gene inside a set as a singleton, as we can see in Figure 4.3. Notice how our initial state represents the univariate FOS.

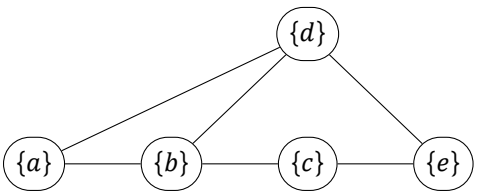
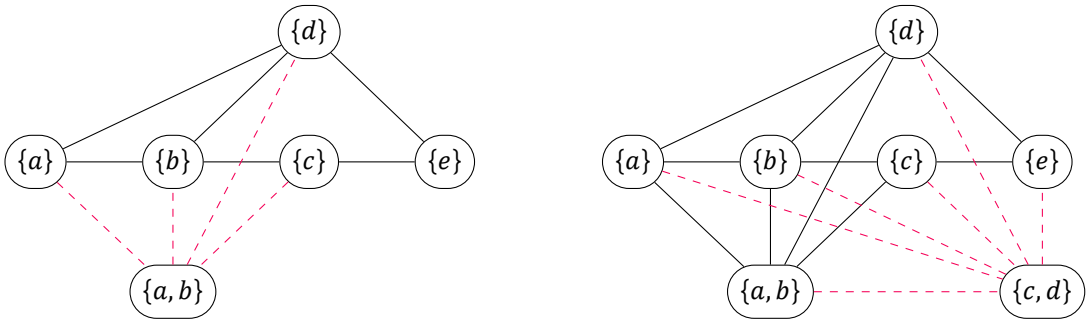


Figure 4.3: The initial ILG representing the initial univariate FOS.

Now, for each newly added linkage set, we will check if they depend on any of the other subsets. Let us start with our first linkage set at f_5: $\{a, b\}$. If we follow the process described in Algorithm 6, we simply look at the vertices $\{a\}$ and $\{b\}$, and observe that they have the neighbouring vertices {{b},{d}} and {{a},{d}} respectively. To add the new edges, we first must create an edge between the newly added linkage set and the intersecting singletons, and then to each of their neighbours $\{c\}$ and $\{d\}$. This process can be found in Figure 4.4a.

Now for $f_6 = \{c, d\}$ which created the linkage set $\{c, d\}$, we repeat the same process. What is interesting for this step is that if we look at the neighbours of vertex $\{c\}$, we see that $\{a, b\}$ is now part of it. For f_6 we thus also created an edge to our previous linkage set at f_6. This process can be found in Figure 4.4b.



(a) The ILG given the new vertex $\{a, b\}$ from the linkage tree has been added.

(b) The ILG given the new vertex $\{c, d\}$ from the linkage tree has been added.

Figure 4.4: The ILG given that a new vertex has been added. The edges are highlighted red to indicate which of them have been recently added.

We can repeat this process for each vertex in the linkage tree. The only ones missing in Figure 4.4 are $f_7 = \{c, d, e\}$ and the root. We do not consider the root to be a useful linkage set, as it contains all the vertices. Drawing F^3 into the graph in Figure 4.4b becomes a mess rather quick; we can observe that f_7 will connect to all vertices.

4.2.2. Packing

Next is creating the packages, which we can achieve by colouring the ILG. Each colour that has been assigned to a linkage set represents a unique package. If two linkage sets have been coloured with the same colour, it means that they do not share neighbouring vertices, nor do they overlap, and thus we can apply parallel acceleration. Similar to revision, because we make use of UPGMA, we receive the largest similarity $\sigma_{i,j}$ which will reuse the index i for the newly created linkage set. Therefore, our linkage sets never exceed an index larger than $\ell - 1$.

Ideally, we would find the least amount of packages needed, however, this would imply that we have solved the set packing problem. Alternatively, we will be satisfied with just any number of packages, which through parallel graph colouring we know that the time complexity is equal to $O(n)$.

Example 4.2.3. Now we can apply graph colouring. We will use the colour to describe a unique package. If we use the same colours in Figure 4.2 and expand it with our new three vertices, we see that we end up with six colours, and thus six packages. i.e.

$$\begin{aligned}
\mathcal{P}_0 &= \begin{bmatrix} 0 & - & 2 & - & - \end{bmatrix}^\mathsf{T} \\
\mathcal{P}_1 &= \begin{bmatrix} - & 1 & - & - & 4 \end{bmatrix}^\mathsf{T} \\
\mathcal{P}_2 &= \begin{bmatrix} - & - & - & 3 & - \end{bmatrix}^\mathsf{T} \\
\mathcal{P}_3 &= \begin{bmatrix} 0 & 0 & - & - & - \end{bmatrix}^\mathsf{T} \\
\mathcal{P}_4 &= \begin{bmatrix} - & - & 2 & 2 & - \end{bmatrix}^\mathsf{T} \\
\mathcal{P}_5 &= \begin{bmatrix} - & - & 2 & 2 & 2 \end{bmatrix}^\mathsf{T}
\end{aligned} \tag{4.8}$$

By making use of the similarity matrix, we automatically get linkage set indices lower than ℓ. One intuitive way of looking at it, is that for a linkage set, it takes the smallest index for all genes inside the linkage set. For our example, the linkage set $f_7 = \{c, d, e\}$ has the associated genes $\{g_2, g_3, g_4\}$ in respective order, and thus will take the index value 2. In the worst case scenario, each linkage set would require its own package. By applying

association, we see that we did improve, from seven to five required packages. Notice how this improvement ratio depends on the problem instance.

4.2.3. μ-Variable

Similar to Revision, the μ-variable provides us a scalar to reduce the number of rounds required to perform UPGMA.

4.3. Contamination

If we look at the dependencies between vertices, we find that they are determined by their related position to each other; if two vertices are neighbours of each other, then they are considered dependent on one another. *Contamination* tries to compensate for those problematic neighbouring vertices, by trying to reduce the number of fitness dependencies. Recall that—even though each gene is processed in parallel—it is the linkage sets which are evaluated as such. If we want to reduce the number of dependencies, we must isolate them within one linkage set. This means that we could reduce the number of dependencies if we make sure that the neighbouring vertices must be within one linkage set. Notice however that on a connected graph, if we want to avoid all dependencies, we end up with a package containing the complete set. That does not sound very useful. On the other side of the spectrum, if we create a linkage set for each vertex there is—which is equal to the univariate FOS—then we encounter the maximum amount dependencies.

In Section 4.3.1 we will dive deeper into the design behind the construction of the package. In Section 4.3.2 we will finalise by introducing the μ-variable associated with contamination.

4.3.1. Packing

Contamination tries to provide us with a balance between both extremes. The package is constructed in rounds. The initial round r_0 starts as the univariate FOS. We append our problem instance graph such that each vertex v_i is assigned a randomly generated weight w_i. Each round, the vertices will compare their own weight with their neighbours': if at least one of the neighbours possesses a larger weight, it will by random choose one of the neighbours and *join* the neighbour by replacing its own random weight by the neighbour's. We say that the vertex has been *contaminated* by the neighbour, hence the name of the solver. We can continue this μ amount of rounds, which we can specify as a variable. What will happen is that every vertex will at some point be contaminated by the largest weight. What started as the univariate FOS ends up as a MP FOS containing the complete set. If we keep track on who contaminates who, then we automatically have found our linkage sets. Notice that contamination will always provide a MP FOS.

Because we will determine the largest weighted neighbour at random, we can use non-deterministic mechanisms to pick one for us. We create a task for each edge, which will determine the largest weight among the adjacent vertices, and overrides the smaller weight with the larger one. If these tasks are performed in parallel, then the non-deterministic mechanisms will make sure one task is slower than the other; if multiple tasks alter the same weight, then the last one will persist. The contamination kernel can be found in Algorithm 7.

Algorithm 7 Contamination$\langle i \rangle(\mathcal{F}, E, W)$

$e = e_i : e = \{v_l, v_r\},\ e \in E$
if $w_l < w_r : \{w_l, w_r\} \in W$ **then**
  $w_l \leftarrow w_r$
  $f_l \leftarrow f_r$
end if

Example 4.3.1. Let us have an example depicting the point-to-point topology. This topology is recognisable by the string of vertices which are characterised by two vertices—the head and the tail— which follows the only path that will visit all vertices. We will use such a topology for our graph, containing five vertices. At the initial round r_0 we will randomly assign weights to each vertex such that we get a weight vector $W = \begin{bmatrix}5 & 3 & 2 & 1 & 4\end{bmatrix}^\intercal$.

Now let us explicitly observe vertex v_3. We observe that the neighbouring vectors v_2 and v_4 are associated with the weights $\{w_2 = 2\}$ and $w_4 = 4\}$ respectively, which are larger than its own weight $w_3 = 1$. The option for v_3 is now to randomly get contaminated by one of both neighbours. Let us for this example always take the smallest weight out of the set. After round r_1 we see that the weight associated to $v_3$has changed to the weight of v_2; v_3 has been contaminated by v_2.

In Figure 4.5 all rounds have been drawn with the μ-variable set equal to 3; there are just two linkage sets left. For this example, you can use the random value as the linkage set index. Notice how, if we picked different values for μ we would have gotten a different package. Also notice how the random nature of the packager provides us with a different weight vector at initialisation, and that even with the same initialisation, we could have ended up with different packages for each round.

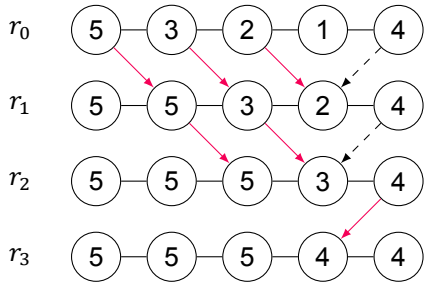


Figure 4.5: An example using the point-to-point topology containing five vertices to illustrate the weights after each contamination round, which weights have been initialised at r_0. Dashed arrows are used to showcase the contamination, while dotted arrows are used to showcase alternative origins.

4.3.2. μ-Variable

The number of rounds μ provides us with a tool to establish the *contamination depth*. By providing a value for the μ-variable, we can assume that the *longest shortest path* within the same linkage set contains at most a depth of μ edges. This should make sense, as the contamination travels one vertex per round, if by pure luck each vertex picks the largest weight exclusively, the contamination depth is equal to the number of vertices it contaminated. The μ-variable provides a tool to optimise for the number of dependencies we would accept to encounter.

5 Experiments

There are many factors that come into play when determining the performance of an evolutionary algorithm. Within computer science, it is not all too uncommon to describe such performance by assigning a tight bound to it. If parallel processes are involved however, it becomes rather difficult to meaningfully describe the processes with such a bound; beyond the complexity of the process itself, we are also bound to hardware of choice, more so than usual. Reading and writing between host and device become real life issues, and similarly the number of blocks and grids. Besides those, it becomes unclear whether a single task shares the same time complexity of an equivalent single sequential process, and whether multiple tasks in parallel have a performance similar to that of only one task. In previous chapters we have discarded these conditions by annotating our time complexity with an optimistic assumption. This of course, does not portray the full process. To determine the performance of a parallel process, we will heavily rely on experiments, crafted in such a way that we could easily analyse it.

We will split this chapter into three sections: the problem description, the setup, and finally the results themselves. First in Section 5.1, we will discuss the problem description, which should assist us to construct conclusions in regards to the performance. Second, in Section 5.2 we will go through the setup, which includes the hardware in use and the problem instances themselves. At last, we will introduce our results in Section 5.3 through the notion of four experiments.

5.1. Problem Description

To measure the performance, we want to construct a simple problem instance for which we can scale its variables such that there is a meaningful correlation between it and that what we want to measure. Simplicity ensures that we should only observe that what we want to measure; there should not be any ambiguity concerning our correlation.

Additionally, for optimisation problems, it would be beneficial if we would be able to determine the optimal solution beforehand; there is a major difference between the complexity for which we want to find the approximation and for which we want to find the optimal solution. A way that provides us with one optimal solution, is to perform the problem instance on an algorithm. Of course, unless this algorithm solved the NP-problem, this approach becomes a little bit cumbersome for larger instances. Alternatively, we construct our instance in such a way that, we can calculate the optimal solution in just $O(n)$ time.

At last, it would be favourable for our problem instance to provide us with only one solution. This way, there is no ambiguity concerning the path our packagers will walk; all packagers need to walk the same road, and will face the same problems. Notice that by definition, there are always two solutions for max-cut; by exchanging the vertices between the partitions we will find an alternative solution.

There are many instances for max-cut we could create, however, we should also confront ourselves with the previous two conditions: How could we create a simple problem instance for which we know the sole optimal solution beforehand? By cleverly constructing a graph, we should be able to do just that. For our experiments, we will favour a fitness which is equal to the total number of edges; every edge has a weight equal to one, of which all will form part of the optimal solution. Back in Section 2.7.2 we discussed that the performance for our parallel acceleration will depend on the adjacent vertices.

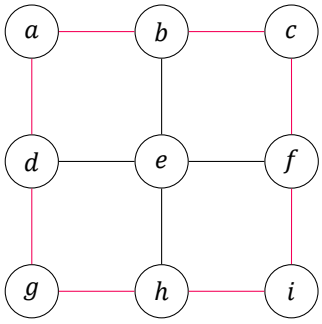
Having all edges weighted equally, simplifies our instance; favouring one cut over the other does not depend on the weights, but the number of edges. By increasing the number of edges, we are able to increase the number of dependencies, which negatively correlates to the performance for our parallel acceleration. To measure the effect of dependencies, we could increase the number of edges per vertex while keeping the number of vertices consistent.

By increasing the number of vertices, we also increase the complexity of our problem instance. The correlation resides in the fact that by increasing the number of vertices, we will also increase the number of genes. First of all, larger genotypes prove to be increasingly harder to randomly be matched with the optimal solution, either by initialisation or by variation. And second, larger linkage sets have the capability to be more destructive and to steer the process to a local minimum instead. In Section 5.1.1 we will introduce the *2D grid* problem description such that we can easily scale up—or down—the number of vertices.

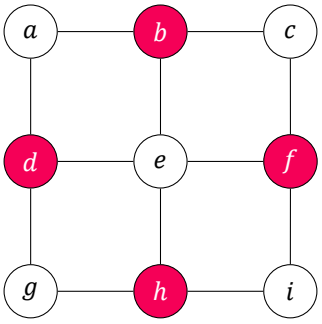
However, increasing the number of vertices is only one of the variables we are able to alter. By changing the number of edges for each vertex, we are able to increase the number of fitness values for the problem instance. In comparison to the 2D grid, we would like to introduce another problem description, this time with more edges per vertex. In Section 5.1.2 we will introduce the *3D grid*, which has just that.

5.1.1. 2D Grid

For our experiments, we will use a *two dimensional grid* of which the vertices—besides those along the border of the edge—will contain four neighbours each. Notice that for large instances, the border in relation to the total number of edges, becomes smallest. Thus, by increasing the number of vertices, we increase the number of edges by a roughly a factor of four. An illustration might be more descriptive; an example of a 3×3 2D grid can be found in Figure 5.1a.



(a) Here the border is visualised in red.



(b) Here an optimal max-cut solution is visualised by colouring some vertices red.

Figure 5.1: an example of a 3×3 two dimensional grid. Notice how the nodes at the border have less than four neighbours.

The benefit of such a grid is that by keeping the rows and columns to the *same odd* number, we are able to determine the optimal solution, which is equal to the sum of the weights of all edges. We are fully able to use non-squared 2D grids, however, in spirit of simplicity we will only consider squared 2D grids with odd numbered dimensions, i.e.

$$G = x \times x \text{ grid, such that } x = 2n + 1, \ , n \in \mathbb{Z}^{+} \tag{5.1}$$

This works because each diagonal forms a set of non-neighbouring vertices. By alternating the colours between neighbouring diagonals, we ensure that they will not contain neighbouring vertices. By making sure that the number of rows and columns are equal to the same odd number, we are able to create a continuous string of alternating colours—- provided we observe them in the same sequence in which we have ordered the letters Figure 5.1a, i.e. $I = 010\ldots10_2$.

Example 5.1.1. We shall view an example of such a construction in Figure 5.1a. We create the set of diagonals by looking at the diagonals from bottom-left to top-right, i.e.

$$
\begin{aligned}
d_0 &= \{a\} \\
d_1 &= \{b, d\} \\
d_2 &= \{c, e, g\} \\
d_3 &= \{f, h\} \\
d_4 &= \{i\}
\end{aligned}
\tag{5.2}
$$

With those diagonals we are able to create two sets—each contain non-neighbouring vertices—by simply alternating the diagonals for each set, i.e.

$$
\begin{aligned}
\Pi^0 &= \{d_0, d_2, d_4\} &&= \{a, c, e, g, i\} \\
\Pi^1 &= \{d_1, d_3\} &&= \{b, d, f, h\}
\end{aligned}
\tag{5.3}
$$

The resulting solution can be found in Figure 5.1b. If we translate it as its binary representation, we get an individual equal to $I = 010101010_2$. Notice that we got a rather easily comprehensible visual representation of our desired solution. In addition, notice that by switching the colours, we get another—yet a different—solution, i.e. 101010101_2. In total there are two solutions available to our 3×3 2D grid.

5.1.2. 3D Grid

With the definition for the 2D grid in the previous section, we are able to scale the number of vertices such that we can measure the performance due to the increase in genotype length. However, we would also like to become aware of the impact of increasing the number of dependencies. In such a case, we still would like to easily deduce our optimal solution. We can do so by adding a dimension to the two dimensional grid, to create a *three dimensional* one. By doing so, we have increased the number of neighbours—for non-border vertices—from four to six vertices per vertex.

To deduce the optimal solution for our 3D grid, we start with a validly coloured 2D grid, and simply alternate these colours for each layer in the third dimension. This provides us with a proper max cut; by altering each layer, we make sure that each neighbour in the third dimension does not match one another. To construct a 3D grid, we now have two variables which we can adjust: the height and the width of the 2D grid, as well as the depth of the third dimension. An example of such a 3D grid can be found in Figure 5.2.

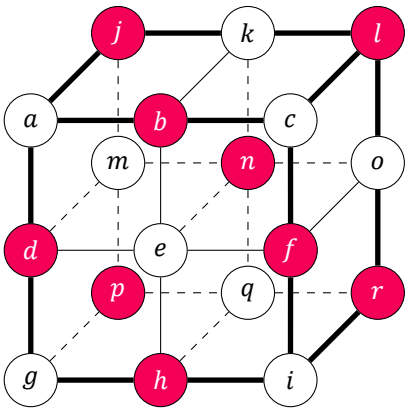


Figure 5.2: an example of a $3 \times 3 \times 2$ three dimensional grid. Thick lines are used to highlight the perimeter of the cube, while the dashed lines are used for vertices hidden by the cube's surface. Here an optimal max-cut solution is visualised by colouring some vertices red.

.

5.2. Setup

Before we move on to the experiments and their results, we should first discuss our setup. In Section 5.2.1 we will discuss the methodology and the hardware used to generate the results. With these,

we should be able to provide a basis in which we are able to construct a conclusion, and to provide the tools to repeat the experiments for future research. In Section 5.2.2 we will discuss our problem instances. With the use of a single list of problem instances, we are able to compare the results between experiments, which similarly will support our claims towards a conclusion.

5.2.1. Methodology

All experiments are run on a *Intel® Xeon®* CPU E5-2630 v4 @ 2.20GHz, using the *NVIDIA® GEFORCE® GTX Titan Xp*. The implementation is written in C++17, and compiled using *GCC* 8.3.0 and the *Cuda compilation tools* release 11.3, V11.3.109. The implementation is fully deprived of any external library besides of the C++ Standard Library. *CMake* 3.18 is used to build the project.

Each experiment is run ten times. The results are recorded if the elite fitness reaches the target fitness—i.e. the optimal solution—and if the target fitness is reached within five minutes. The runtime is recorded by the process itself, which spans the complete runtime from start to finish.

5.2.2. Problem Instances

For all our experiments, we will use both problem descriptions discussed in Section 5.1. To keep the comparison valid, we will try to keep the numbers of vertices equal between problem instances of both problem descriptions. Thus, both instances will only differ in the number of neighbours per vertex; if the number of vertices is equal, a 3D grid instance should have more neighbours per vertex than its 2D instance equivalent. Keep in mind that the number of vertices $|V|$ will consequentially be equal to the genotype length ℓ.

We would favour to plot our problem instances on a log_2 scale. We have chosen for this scale due to the expected performance—which is based on time complexity of max-cut and the performance of discrete-GOMEA—which should be visible on this scale. Recall from our problem description that our 2D grid can be described by one variable, while our 3D grid can be described by two. This leaves us to conclude that we have a restricted number of 2D grid instances to generate, and that there are multiple ways—if possible—to match a 2D instance with a 3D grid instance as closely as we can.

Consequentially, if we desire to measure the performance on the log_2 scale, we would like to reflect this in our variable set. With the restriction in mind, it should be clear that we cannot find a fixed exponential. Instead, we use a set of instances that are *close enough* in comparison with a base equal to 2. For our 2D grid instances and a target t_ℓ this is rather simple to determine, i.e.

$$\begin{aligned} &t_\ell \mapsto x \times x \text{ grid, such that,} \\ &\quad x = 2n + 1,\ n \in \mathbb{Z}^+ \\ &\quad \min(|t_\ell - x^2|) \end{aligned} \tag{5.4}$$

For our 3D grid instances however, *close enough* needs to be specified more thoroughly. Remember that our comparison with an equivalently sized 2D instances is only useful if it contains more neighbours per vertex. With two variables, we are able to create the same instance using different configurations. However, given our requirement, some configurations are more desirable than others. We say a 3D grid instance is *close enough* if the number of vertices in each dimension is equally maximised, i.e.

$$\begin{aligned} &t_\ell \mapsto x \times x \times z \text{ grid, such that,} \\ &\quad z \in \mathbb{Z}^+ \\ &\quad x = 2n + 1,\ n \in \mathbb{Z}^+ \\ &\quad x^3 \leq t_\ell < (x + 1)^3 \\ &\quad \min(|t_\ell - x^2 + z|) \end{aligned} \tag{5.5}$$

For all our experiments, we will use a set of values such that $32 \leq log_2 \leq 4096$. If we follow the definitions in Equation (5.4) and Equation (5.5) to construct our 2D and 3D grid respectively, we will get a list of problem instances expressed in their number of vertices, i.e.

$$\begin{aligned} \ell_{\log} &= \{32, 64, 128, 256, 512, 1024, 2048, 4096\} \\ \ell_{2D} &= \{25, 49, 121, 225, 529, 961, 2025, 3969\} \\ \ell_{3D} &= \{27, 63, 125, 250, 490, 972, 1936, 4050\} \end{aligned} \tag{5.6}$$

5.3. Results

The intent for each experiment is to showcase the capabilities of each packager by visualising its performance, and to enhance our understanding of their behaviour. First, we need to resolve the optimal *configuration* by tweaking the two variables which come for each packager: the μ-variable and the population size. The results for both will be presented in Section 5.3.1 and Section 5.3.2 respectively. Second, we will use two visualisations to showcase the behaviour for each packager: the so called *convergence*, and the profile information. The result for both will be presented in Section 5.3.3 an Section 5.3.4 respectively. Finally, in Section 5.3.5 we will look into the performance of each packager by taking the best results from the previous sections and compare them to discrete-GOMEA by plotting the run time as a function of the problem instance.

5.3.1. μ-Variable

Each packager has a so called μ-variable. This variable can be used a parameter for a given packager; How the μ-variable is interpreted depends on the packager itself. The *contamination* packager will use the μ-variable to determine the number of rounds as described in Section 4.3.2. All other packagers use it as a scalar which represents the cut-off length of the revision list, as described in Section 4.1.2. The μ-variable is thus a variable which can be optimised depending on the problem and packager.

With both the μ-variable and the population size, each packager has two variables to configure. Two variables will create an extensive amount of experiments, resulting into a plane of data-points for each problem instance. Instead, we opt for one variable to be fixed. This causes a chicken-and-egg problem. We start using a fixed value of our problem size, and after that we are able to deduce the μ-variable.

With some intuition we can assume there exists some correlation between the two variables. Determining one, while keeping the other fixed, will not explore the complete variable space; it is just a heuristic guess. Unfortunately, we will not explore beyond this guess, as there are simply pragmatic limits to the scope of a thesis.

Intuitively, we know that a large population will only increase the runtime, but will not break the process, something which we do not know yet for the μ-variable. We fix our population size to 512 individuals. The μ-variable is intended to be plotted on a
log_2 scale. We will pick a range between 1 and 64, i.e.

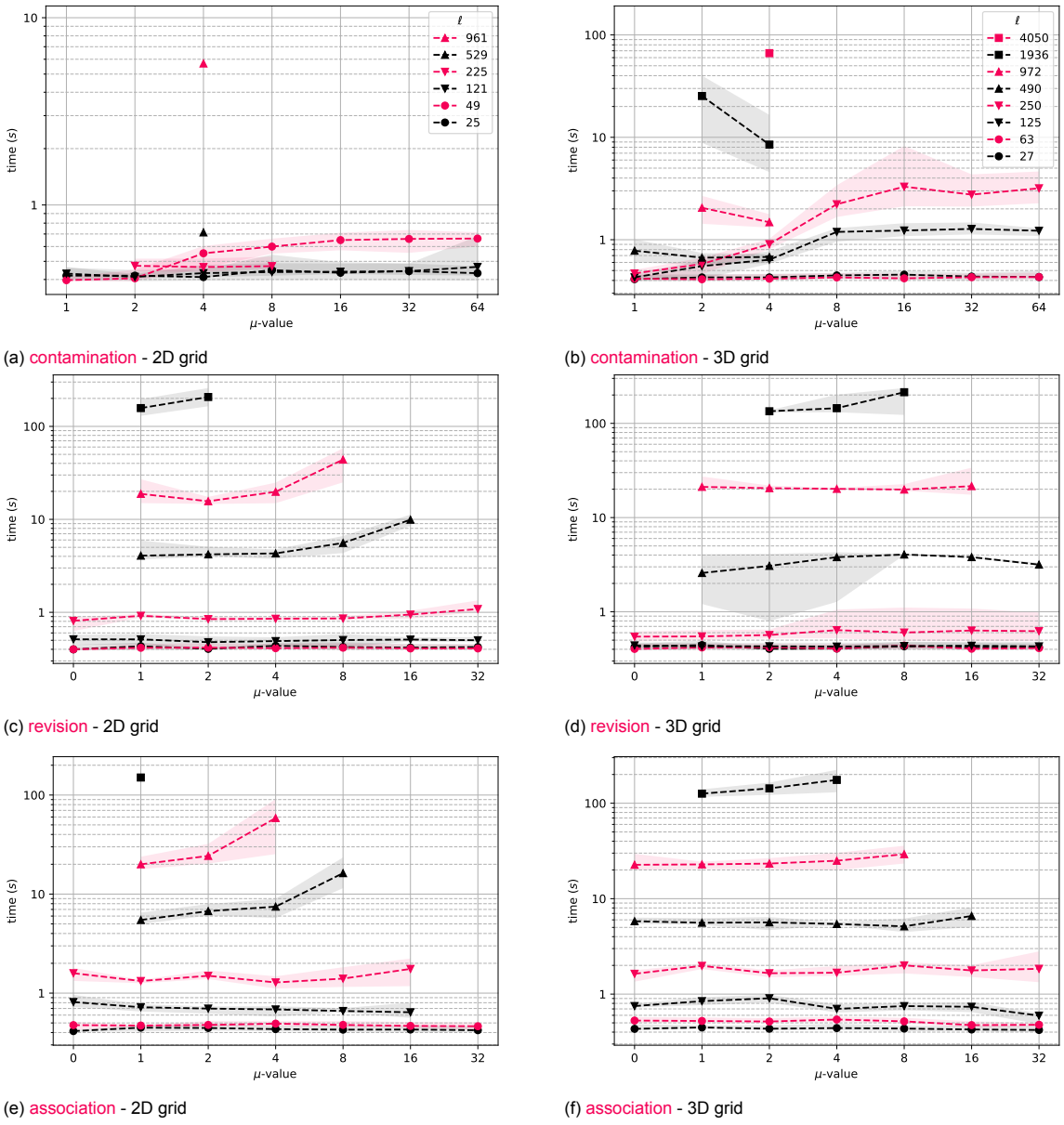
$$\mu = \{1, 2, 4, 8, 16, 32, 64\} \tag{5.7}$$

The results can be found in Figure 5.3 and Figure 5.4 for the packagers {contamination, revision, association} and {square-root, exponential, logarithmic} respectively.

5.3.2. Population

The population size is the common parameter for any EA, seeing that it is also a common performance hit, as each individual must be evaluated. Additionally, the problem size also determines the variation within the population as a whole. If the population size is too small, the EA might get stuck in a local optimum, too large, and it will take a performance hit. The population size is thus a variable which can be optimised depending on the problem and packager.

In Section 5.3.1 we can observe a comfortable μ-variable for each problem instance, which means that we have found a value that is able to work across each problem description. As discussed earlier, with this observation, we are able now able to fix the μ-variable, i.e.



(a) contamination - 2D grid

(b) contamination - 3D grid

(c) revision - 2D grid

(d) revision - 3D grid

(e) association - 2D grid

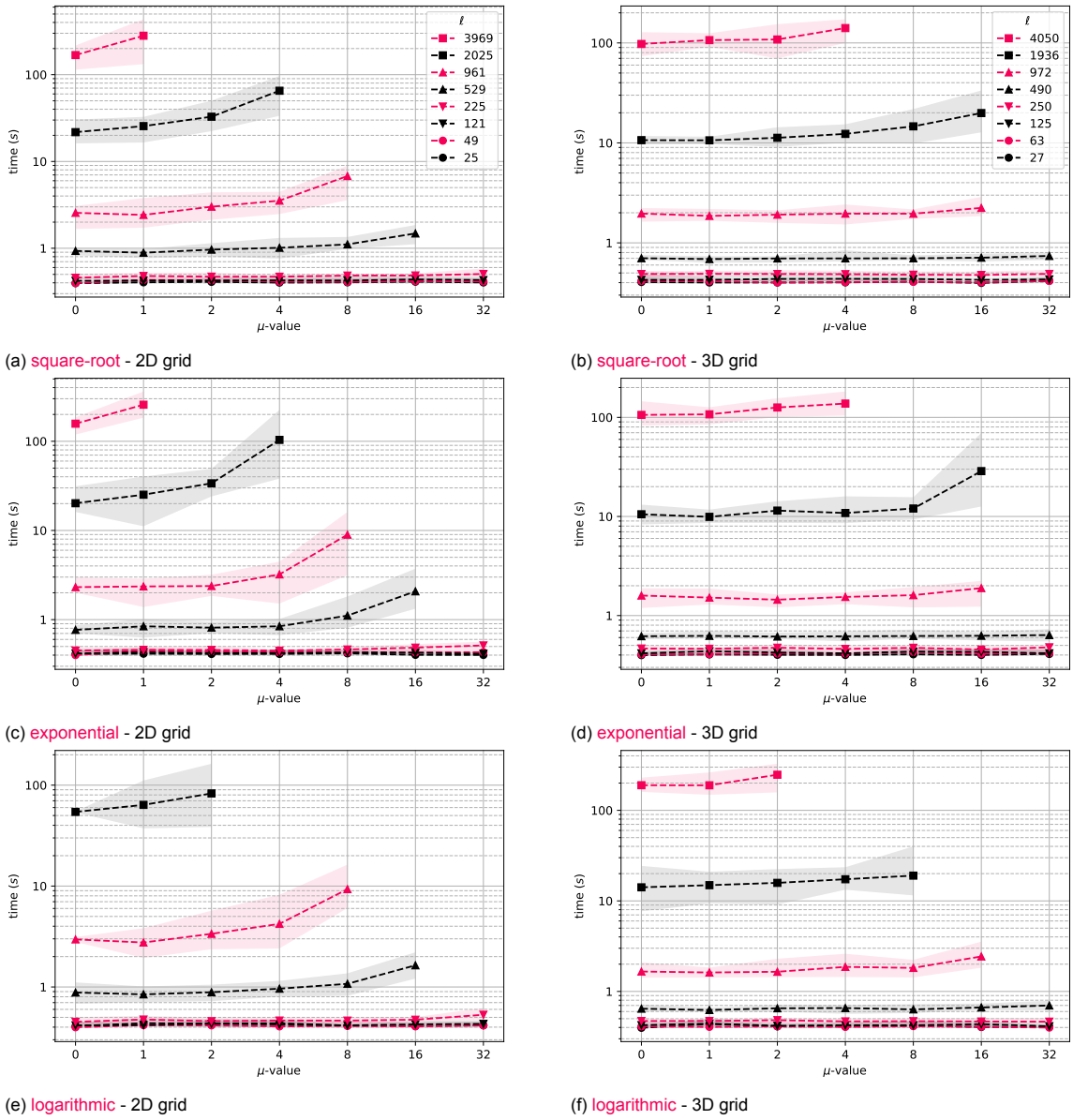
(f) association - 3D grid

Figure 5.3: (Part 1/2) The runtime for a μ-variable, given the population is fixed at 512 individuals. Each sub-figure is a combination of a packager and a problem description, which can be either a 2D grid or a 3D grid.

	μ_{2D}	μ_{3D}
contamination :	4	4
revision :	2	4
association :	1	2
square-root :	1	1
exponential :	1	1
logarithmic :	1	1

(5.8)

The population is intended to be plotted on a log_2 scale, thus, the population increases by a factor of 2. We deem the population smaller than 32 individuals as too small in most cases, and larger than 1024 individuals as uninteresting. We will use a set of log_2 values within the range between 32 and 1024, i.e.



(a) square-root - 2D grid

(b) square-root - 3D grid

(c) exponential - 2D grid

(d) exponential - 3D grid

(e) logarithmic - 2D grid

(f) logarithmic - 3D grid

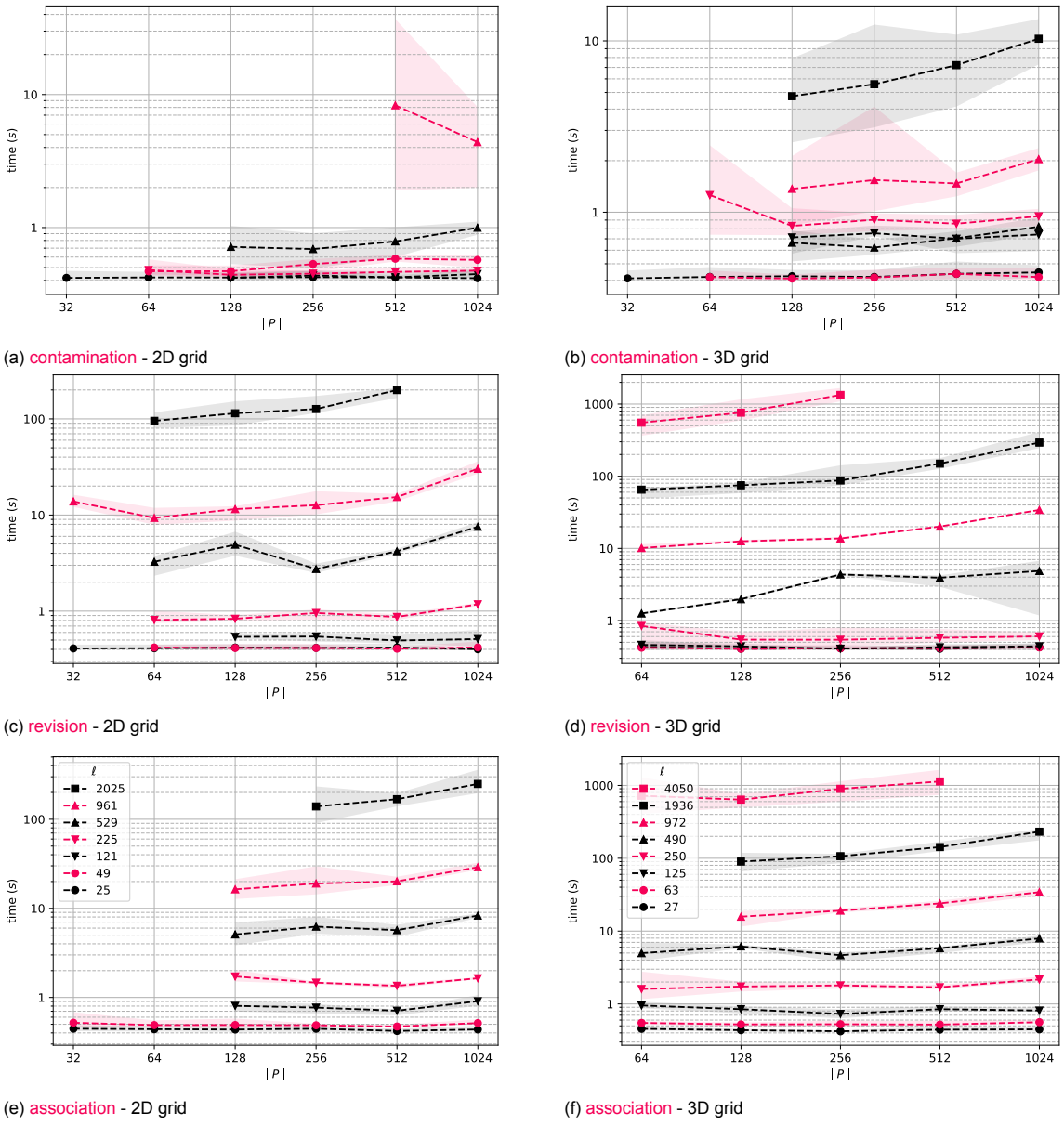
Figure 5.4: (Part 2/2) The runtime for a μ-variable, given the population is fixed at 512 individuals. Each sub-figure is a combination of a revision variant and a problem description, which can be either a 2D grid or a 3D grid.

$$n = \{32, 64, 128, 256, 512, 1024\} \tag{5.9}$$

The results can be found in Figure 5.5 and Figure 5.6 for the packagers {contamination, revision, association} and {square-root, exponential, logarithmic} respectively.

5.3.3. Convergence

The convergence of a packager is described by the scatter plot where each dot represents the best fitness ϕ at some point in time until the packager hits the target fitness t_ϕ. The plot is represented on a $\log_{10}$ scale such that the y-axis represents the distance between best fitness and target fitness, i.e. $t_\phi - \phi + 1$. What we get is a plot unique to each packager. The convergence allows us to take a closer look at the performance in regards to the donations. Favourably, each donation would improve the fitness by a margin better than the previous donation. In the worst case, the fitness after a donation does not improve and therefore is plateauing, which renders the donation itself a waste of runtime. For each case, we would say that the problem experiences a superlinear or a sublinear rate of convergence,



(a) contamination - 2D grid

(b) contamination - 3D grid

(c) revision - 2D grid

(d) revision - 3D grid

(e) association - 2D grid

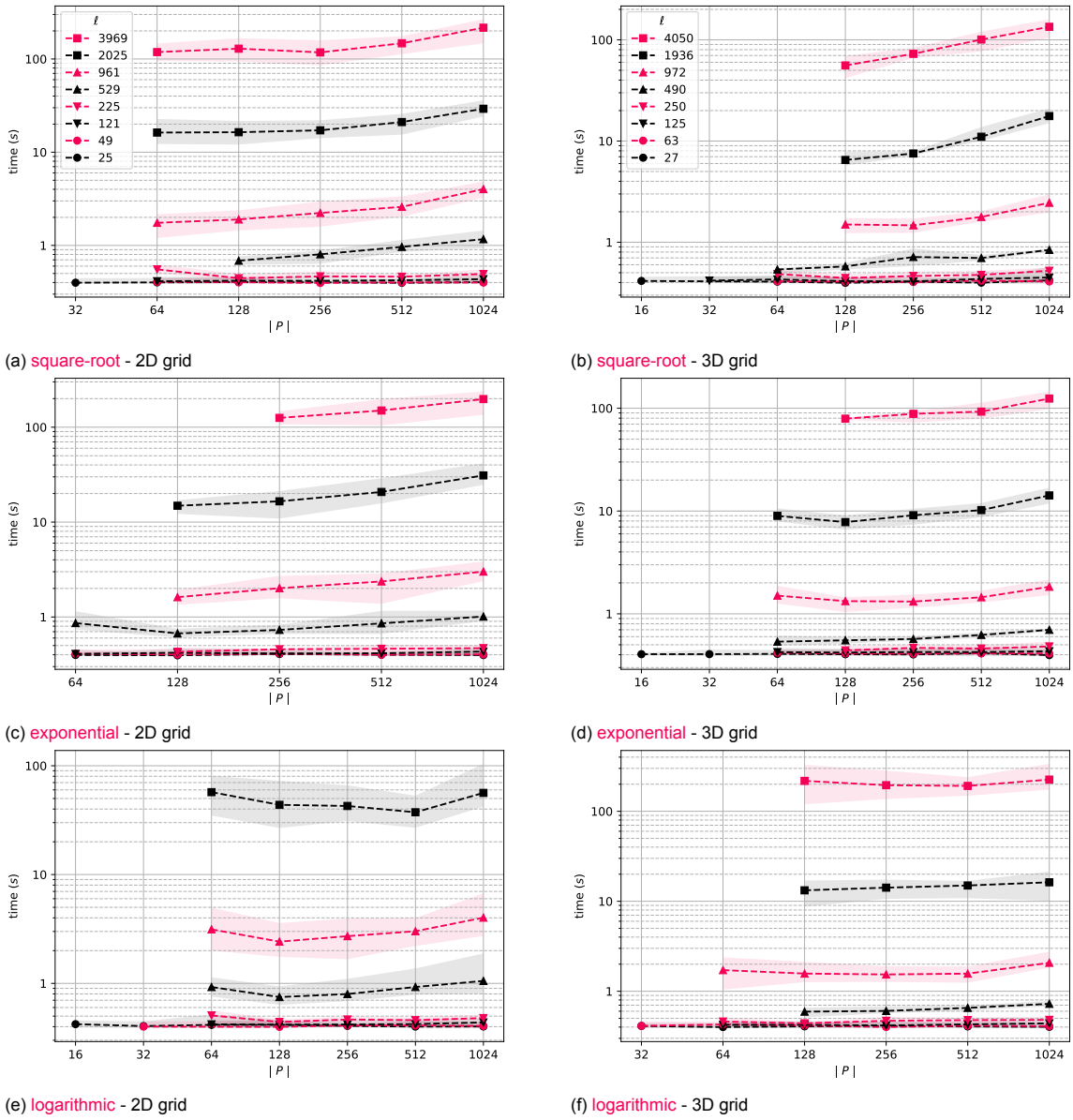
(f) association - 3D grid

Figure 5.5: (Part 1/2) The runtime for a variable population size, given the μ-variable is fixed, see Equation (5.8). Each sub-figure is a combination of a packager and a problem description, which can be either a 2D grid or a 3D grid.

respectively.

For the convergence, we will use an easy-to-run problem instance with a save chose for the population and μ-variable; there is no intend for optimisation, only for observation. We have chosen a population of 512 individuals, and the problem instances 961 and 972 for the 2D and 3D grid respectively. Each packager will have their own μ-variable, i.e.

$$\begin{array}{rcc} & \mu_{2D} & \mu_{3D} \\ \text{contamination}: & 4 & 4 \\ \text{revision}: & 1 & 1 \\ \text{association}: & 1 & 1 \\ \text{square-root}: & 1 & 1 \\ \text{exponential}: & 1 & 1 \\ \text{logarithmic}: & 1 & 1 \end{array} \tag{5.10}$$



(a) square-root - 2D grid

(b) square-root - 3D grid

(c) exponential - 2D grid

(d) exponential - 3D grid

(e) logarithmic - 2D grid

(f) logarithmic - 3D grid

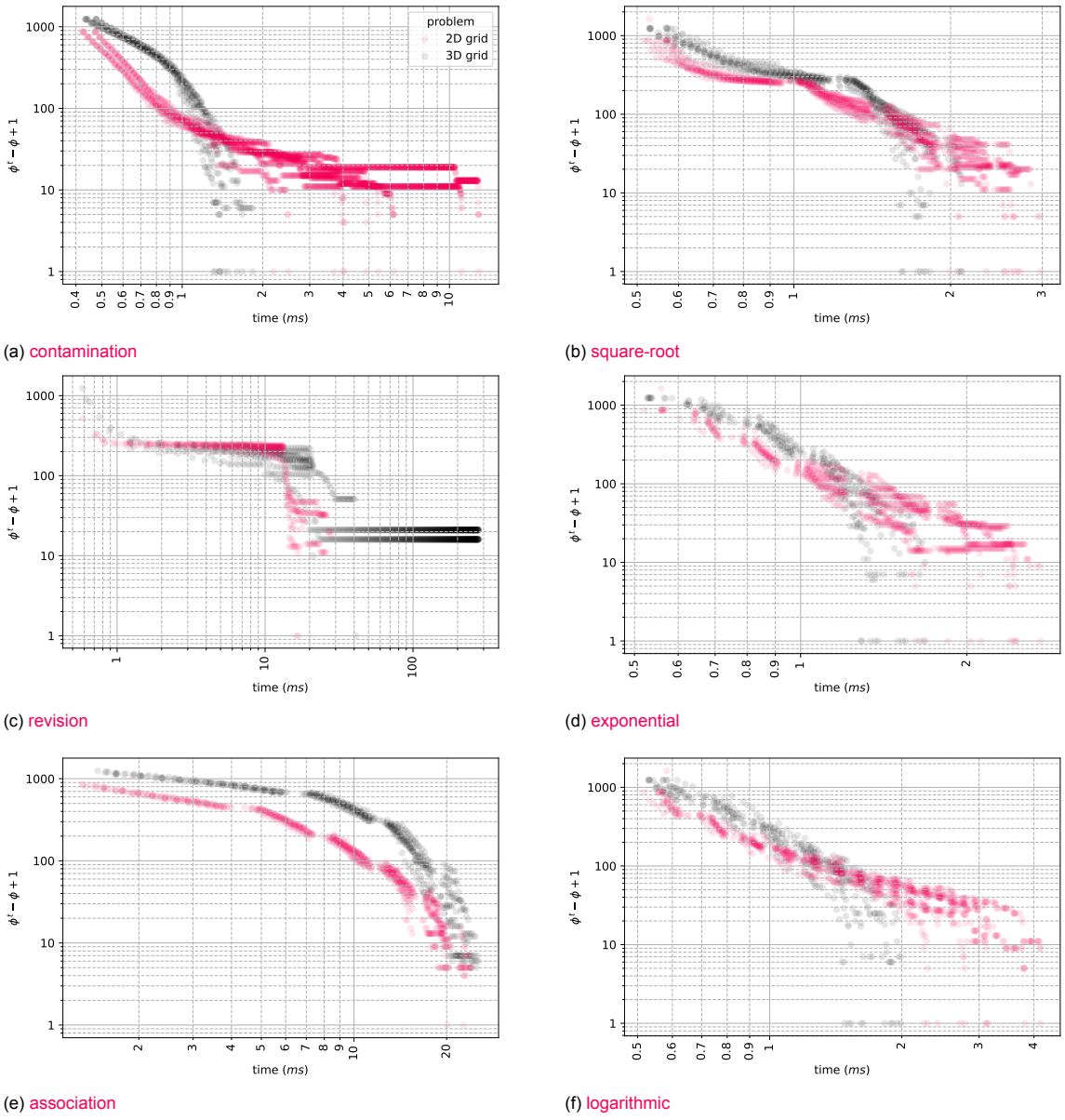
Figure 5.6: (Part 2/2) The runtime for a variable population size, given the μ-variable is fixed, see Equation (5.8). Each sub-figure is a combination of a revision variant and a problem description, which can be either a 2D grid or a 3D grid.

Notice that both the population size and the instances are chosen such that for each problem, the μ-variable matches between the 2D and 3D grid instances. The results can be found in Figure 5.7.

5.3.4. Profile

Nvprof is a tool provided by NVIDIA which visualises profiling data from the command line. This profiling data is presented as a table in which each row represents a kernel process. For each process, profiling data is recorded such as the execution count and the total runtime. By plotting the result into a heat map, we are able to visualise the behaviour for each packager. This heat map is a simple, yet effective way to visualise and compare the behaviour for each packager.

The heat map is constructed by taking the top five kernels for each packager in respect to the measurement. Note that not every packager will share the same top five kernels. If a kernel is in the top five of some packager, it will be added to the heat map, and for all packagers—even if the kernel is not part of the top five—the value will be provided. For this experiment, we are interested in two measurements: one is the total runtime for each kernel as a percentage of the total runtime, and the



(a) contamination (b) square-root

(c) revision (d) exponential

(e) association (f) logarithmic

Figure 5.7: The convergence for each packager on both problem descriptions, given that the μ-variable and population are fixed, see Equation (5.8). Each plot has been scaled such that the data is limit to 2000 entries.

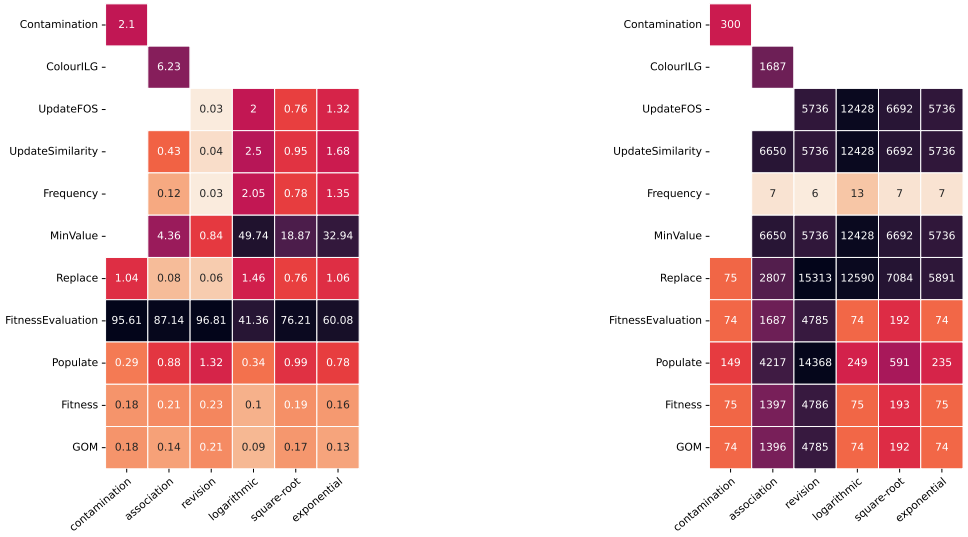
other is the execution count.

For the heat map we will use the same configuration as we used in Section 5.3.3, only this time, we will solely use the 2D grid instances. This because the difference between kernel usage between both problem descriptions are not enough to provide additional insight. The heat map can be found in Figure 5.8.

5.3.5. Competition

Now that we have most of the data, it is interesting to compare the performance with that of the traditional discrete-GOMEA. Now, the focus of this paper might not revolve around this premise, it still is interesting nonetheless. First, let us try to optimise discrete-GOMEA as we have done previously for all packagers. Notice that discrete-GOMEA only has one variable: the population size. This makes it less cumbersome to optimise, as there is no μ-variable available. Similarly we use the same population set as described in Equation (5.9). The results can be found in Figure 5.9.

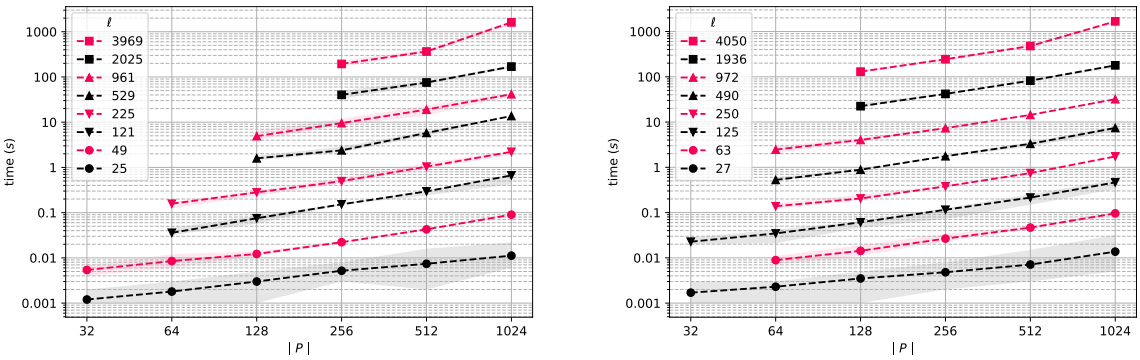
To visualise the results for all packagers and those for discrete-GOMEA, we will use the most



(a) A value represents the percentage of the total runtime.

(b) A value represents the total number of calls to the kernel.

Figure 5.8: A heat map of the most *significant* kernels for each packager.



(a) 2D grid

(b) 3D grid

Figure 5.9: The runtime for a variable population size using discrete-GOMEA on both the 2D and 3D grid instances.

favourable configuration, which we are able to gather from previous results. We are a bit forgiving with the chosen population size for each packager, as the optimal population for each instance does experience some leeway. The configurations for the 2D and 3D grid instances can be found in Equation (5.11) and Equation (5.12) respectively.

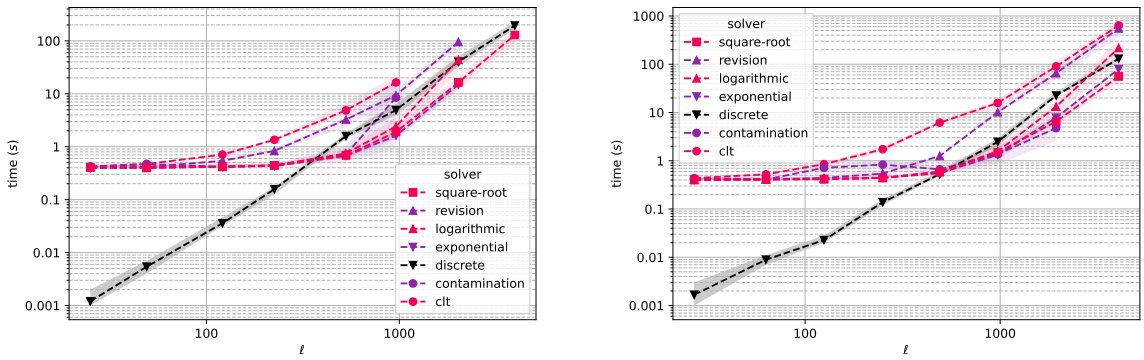
	μ_{2D}	I_{25}	I_{49}	I_{121}	I_{225}	I_{529}	I_{961}	I_{2025}	I_{3969}
contamination :	4	128	128	128	128	128	1024	–	–
revision :	1	128	128	128	128	64	64	64	–
association :	1	512	512	512	512	128	128	256	–
square-root :	1	128	128	128	128	128	128	128	128
exponential :	1	128	128	128	128	128	128	128	256
logarithmic :	1	128	128	128	128	128	128	128	128
discrete :	–	32	32	64	64	128	128	256	256

(5.11)

	μ_{3D}	n_{27}	n_{63}	n_{125}	n_{250}	n_{490}	n_{972}	n_{1936}	n_{4050}
contamination :	4	128	128	128	128	128	128	128	–
revision :	4	128	128	128	128	64	64	64	64
association :	2	128	128	128	128	128	128	128	128
square-root :	1	128	128	128	128	128	128	128	128
exponential :	1	128	128	128	128	128	128	128	128
logarithmic :	1	128	128	128	128	128	128	128	128
discrete :	–	32	32	32	64	64	64	128	128

(5.12)

We will measure the performance by taking the runtime as a function of the problem instance. This should feel intuitive; the larger the problem instance–and thus the larger the genotype—the harder it gets to find the elite fitness. The results can be found in Figure 5.10. Notice that a unique tri-colour scheme is used to improve the visibility.



(a) 2D grid

(b) 3D grid

Figure 5.10: The performance of all packagers, including discrete-GOMEA, visualised for both 2D and 3D problem instances.

6

Conclusions and Discussions

After we have processed the results, the only remaining point on our to-do list, is to summarise it all and to construct some conclusions. There is also some room for discussion, which we will use to eventually provide some suggestions for future work. We will start off with the conclusions in Section 6.2.3, after which we will dive into the discussions in Section 6.1.

6.1. Discussion

Before we move on to the conclusions, let us first have a discussion on the results. The discussions are set up in such a way that we will go through each and every experiment we did state in Chapter 5, except for the comparison. The experiments that we will discuss are the μ-variable in Section 6.1.1, the population size in Section 6.1.2, the convergence in Section 6.1.3, and at last the profile in Section 6.1.4.

6.1.1. μ-Variable

Despite its name, the μ-variable does not share the same definition among all packagers, other than being a parameter; in comparison to the others, contamination takes a different meaning for the μ-variable. If we momentarily take contamination out of the picture, we observe some consistent behaviour among the packagers. This might be expected due to their similarities. If we take a look at the experiments, we are able to establish two generalisations. First, the range of acceptable μ-variables will decrease the larger the problem instance will be, and second, the range will shift to the right if the number of neighbours per vertex increases.

Let us recall that the μ-variable for revision and association represents a scalar to reduce the depth of the linkage tree. This shows us a few things. First, there is indeed some meaning behind *too few* and *too many* larger linkage sets. Second, there seems to be some kind of correlation between the variable and the number of neighbours, which we have established as being equal to the number of fitness dependencies. And last, it shows us that for each problem instance, the variable tends to converge to one single value.

If we look at the compression variants, we observe a different story. Now, there is surely meaning behind too few linkage sets, but this does not apply when there are too many sets. This might not come as a surprise; by compressing the revision list, we randomly might remove some of the larger linkage sets, of which the duplicates are additionally less present.

Unfortunately, there are some limitations to the setup of our experiments, which prevents us from constructing proper conclusions. For starters, we cannot observe whether the convergence will eventually end; we cannot determine whether there is a maximum number of vertices for our problem instance for which GPU-GOMEA will not be able to receive a solution. In addition to this, we did not include the results which took longer than five minutes to run for μ-values. This could still imply that the optimal solution is still feasible, just not within the time we specified.

If we have a special look at contamination, we see some interesting behaviour. For starters, it seems to have a clear favour for $\mu = 4$. Additionally, contamination fails to run on the 2D grid instance with larger number of vertices. At last, it seems to match the same behaviour we observed for revision

and association; in all cases, there seems to be a convergence towards a single value. This might suggest that the convergence is not a property of the packager, but of the problem itself.

6.1.2. Population

If we take a look at the population, we notice some slight differences among the packagers. Let us start with the more general observations. Over the board, the population size contributes little to the overall runtime, however, the consensus is that a smaller population size is in most cases better. For all packagers, a larger population size—with a predictable runtime penalty—will always provide us with an optimal solution, if we have found a minimally allowed population size. Everything smaller than this minimum size exceeded the runtime beyond the five minute mark, and thus was not recorded. This suggests that it might be not feasible to use such small population sizes due to the lack of variety. This is not specific to GPU-GOMEA, as the same pattern can be observed for discrete-GOMEA.

Now, we already have taken a look at discrete-GOMEA, we should also be able to observe that the impact on the run time caused by the population size is much more prevalent than in comparison to our packagers; for our packagers, the impact seems almost nullifiable, but still present. If we recall Section 2.1.4, we made an assumption that we optimistically could establish a time complexity equal to $O(1)$. It shows us why we consider the term *optimistically* in the first place; in practice, there is some overhead to take into account when it comes to kernel design, which we do not consider when speaking of optimistic assumptions.

Let us take a closer look at each and every packager. One thing to point out is contamination, which tends to behave quite differently in comparison to revision and association. It clearly needs a larger population size in comparison to the others. What this hints at, is that contamination needs more variety within its population to find the optimal solution. However, if we take the results discussed in section 6.1.1 into account, then we could also conclude that this is simply the result of contamination failing for larger 2D grid instances. In regards to the 3D grid ones it seems to behave in line with the other packagers.

If we look at the other packagers, we see some differences in regard to the minimum of the allowed population size per problem instance, per number of vertices of said instance. Some, like revision, are hardly affected by the population size, neither are they affected by the difference between the 2D and 3D grid instances. If we look at its variances, then we do see that over all, the minimum required population size is larger for the 3D grid instances in comparison to their respective 2D grid instances. Association—similar to contamination—works the opposite way, and seems to be able to find the optimal solution with a smaller population size for the 3D instances in comparison to the 2D ones.

6.1.3. Convergence

If we take a look at the convergence, we see quite some differences among the packagers, more so than we did in the previous sections. First off, we consider some general observations. Besides contamination, we can see some clearly empty spaces between sets of dots; these spaces represent the complete optimal mixing process. From these we can generally say the optimal mixing process has only a small contribution to the total run time of the EA. Accelerating optimal mixing is just a small—yet still crucial—part of accelerating the EA as a whole. Next, for all packages besides contamination, there are distinct rates visible between the empty spaces; each and every generational step has its own convergence rate. And last, for all packagers we see barely to no dots at the convergence value lower than 10. This highlights the struggle with EAs when it comes reaching the optimal solution.

If we take contamination under the loop, we can observe why it was not capable of reaching the optimal solution for larger problem instances in regards to the 2D grid problem instance; it is plateauing. Interestingly, this is not the case for the 3D grid instance, on the contrary even. For such an instance, the packager experiences a superlinear convergence. This provides us with the hint that contamination could perform better on well-connected graphs.

Upon taking a closer look at revision, we see the worst case of plateauing; the tail for each sublinear rate of convergence takes a majority of the whole body. We also should note that it takes just two generational steps. It is clear that the majority of the useful donations happens at the start of each generation; we could have removed up to ⅔ of the revision list while we still would have been able to reach the optimal solution. This we already expected, thus the revision variations.

If we take a closer look at the variations, we can observe some generalisations. For all packagers,

we observe that it takes more effort to find the optimal solution for the 2D grid instance, as all experience sublinear rates of convergence. This is in line with contamination, which suggests that for the 2D problem instance it is in general harder to find the optimal solution than the 3D one. Also, interestingly, we have found that all variations suddenly switch to a superlinear rate when they are very close to the optimal solution. Additionally for each variation, the runtime between runs seems to diverge from each other when getting closer to the optimal solution. Both being an indication that the variations heavily rely on the diversity of the population. At last, it is interesting to note that the square-root packager causes a clear switch in rates between the first two generational steps, in respect to the 3D problem instance; square-root is on the edge of using too many packages.

At last, we take a closer look at association. Here we see by far the most consistent superlinear rate of convergence, both between the 2D and 3D problem instances, and in the runs per instance. Over all, we should not be surprised by these observations; association takes many efforts to reduce the number of packages while keeping the number of dependencies between linkage set within a package equal to zero.

6.1.4. Profile

Using a heat map makes it relatively easy to spot certain patterns, all that is for us to look at the darker squares to know which kernels are not performing up to par compared to the rest. If we look at the percentage of the total runtime, one observation stands out; for all packages, the fitness evaluation takes the largest percentage. This should however not come to us as a surprise. We already knew that the fitness evaluation for each gene takes an incredible hit on performance, even though it could make use of reduction. Simply that the kernel is required to be performed multiple times within a generational step, makes it such a punishing process. Second place takes—by no one's surprise—a similar kernel which makes use of reduction: the minimum value kernel.

By taking into account the number of calls, we could observe some more interesting aspects. For one, we know that frequency relies on atomic functions, which could have been devastating for the runtime. However, due to the low number of calls, it takes hardly any matter in the total runtime. Second, we see a lot of calls to a specific set of kernels, which include: updating the FOS; updating the similarity matrix; calculating the minimum value; and replacing elements between vectors. We should notice how the minimum value kernel is called more often than the fitness evaluation kernel, but it contributes less to the total runtime. the main difference between the two being that the fitness evaluation performs reduction in parallel for all the donations, while the minimum value is simply performing just one.

6.2. Conclusions

We have shown that it is feasible to translate discrete-GOMEA by making use of GPU acceleration. GPU-glsgomea shows that we capable of accelerating discrete-GOMEA by designing specific kernels to match each process. Discrete-GOMEA can be generalised into two main processes: performing optimal mixing and creating the linkage tree. For the first, we have been able to reduce the runtime complexity for optimal mixing to an optimistic $O(\log(\ell))$ time, therefore removing the relation between the population size and the total runtime. This is also reflected in the results; all packagers show a low correlation between runtime and population size, in contrast to discrete-GOMEA.

For the latter, we had to make some concessions to translate the FOS to be usable within our kernel designs, which took the form of packages. Packages take a big role in GPU-GOMEA, and therefore it is suited to find a packager which we are able to create within a doable time, preferably using parallel acceleration. We have been able to successfully accelerate UPGMA using parallelisation, of which its actual performance makes little contribution to the total runtime. To generate packages using this process, we have proposed two different solutions: revision and association. As an alternative, we have shown that it is feasible to find an alternative packager other than making use of UPGMA in the form of contamination.

There is also an interesting remark applicable for most packagers, which is that the performance improves if the problem instance has more edges for each vertex. It is also interesting because this phenomenon is o so slightly reflected in discrete-GOMEA. With the current results however, we are unable to construct a conclusion out of this single observation.

For each packager that we have constructed, there are some interesting conclusions we could derive from their results specifically. In Section 6.2.1, Section 6.2.2 and Section 6.2.3 we will discuss

those conclusions in regards to the packagers contamination, association and revision respectively.

6.2.1. Contamination

Contamination is an interesting packager among the others. It firstly is not making use of constructing a linkage tree, but instead uses our pre-knowledge about the problem description. And secondly, it is the only packager that is capable of constructing a package in $O(\mu) \approx O(1)$ given $\mu \ll |E|$. Which is indeed the case, as we have observed that $\mu = 4$ is the preferred value for our experiments.

Contamination shows that, even if we have improved our generational process, that does not correlate to the actual runtime. Especially with a small number of edges for each vertex, we see hardship to reach the optimal solution; it is the last few fitness points which are the hardest to reach. What unfortunately also shows is that contamination cannot reach the optimal solution if the problem is too large, while the number of edges for each vertex is small. We see a similar struggle if we increase the number of edges for each vertex. Similarly, or as a result, the μ-value is highly affected by the problem size, and largely affected by the problem description.

To finalise, we should remark the correlation between the problem description and performance. By increasing the number of edges for each gene, we are able increase our performance by a factor of 10. This is huge. Additionally, we see a change in convergence; from plateauing to a proper superlinear rate, we see a major shift. Unfortunately, with the current results, we are not able to construct a proper conclusion out of this observation. Even if our design might be flawed, what contamination mostly illustrates, is the strength of extracting the information from a problem to reach a simple yet powerful design.

6.2.2. Association

By generating an independent linkage graph, we are able to fully implement discrete-GOMEA, while avoiding all dependencies among the linkage sets. By doing so, we are forced to execute some processes on the host, which as a result requires memory to be exchanged between it and the device. This is reflected on the performance; both optimal mixing and the construction of the packages suffer because of it. As a result, we are only able to conclude that the performance penalty is hard to ignore. We also see that the μ-value is largely affected by the problem size and the problem description.

There are interesting remarks to make though. In contrast to the other packagers, we do see a favourable convergence rate; there is no plateauing, and it even sharply decreases the distance to the optimal fitness as it gets closer. Especially the last is remarkable, as we see the other packager struggling with the last few fitness points. Also remarkable, is that the packager is less affected by the change in problem description; even with more edges for each vertex, association is able to finish in roughly the same runtime. This makes it a more flexible solution if the connectives of a problem instance are still unknown. At last, the performance seems to be less affected by the size of the problem, although this could only be confirmed with more excessive experiments. With the convergence rate, there are some hints suggesting that avoiding the dependencies should provide us with some performance gain, but only if we are able to fully transition to parallel acceleration.

One thing to note, is the time complexity to construct the packages. We should be aware of the fact that each linkage tree can only be constructed with a limited amount of packages. To find the least amount of packages, we stumble upon the set packing problem; we need to solve an NP-complete problem. The more linkage sets we can fit inside a package, the more sets we can accelerate using parallelisation, the more performance we gain. However, as long as the optimal solution for the set packing problem is undoable, we can only make use of approximations. Even if we are able to reduce every kernel process to an optimistic $O(1)$ time, we will always be bound to the set packing problem.

6.2.3. Revision and Variants

If we look at revision, there is only one major conclusion we could come to, which is that it is extremely prone to plateauing. Its convergence rate is nothing to write home about. It shows that the majority of the revisions inside the list is nothing more than dead weight. This should not be too much of a surprise, as we already anticipated such a conclusion. Analysing the design of the revision list, we could already predict that the many duplicate linkage sets between revisions would not be favourable regarding the performance.

That is where the compression variants come into play. By reducing the number of duplicates, we prevent plateauing, and thus we are able to increase the performance. Three variants are provided:

square-root, exponential, and logarithmic. Each of them create a differently sized revision list. If a list is too large, plateauing will occur, too small and the effects of constructing the revision list will be noticed. Square-root shows plateauing, logarithmic shows us the effects of reconstructing the revision list, and exponential is somewhere in between. It might thus not be a surprise that exponential is the best performer out of the bunch. It might also not be a surprise that the μ-variable is of no importance anymore, which simplifies the use of packagers.

What is interesting is that, even with the dependencies fully embedded within each revision, we are still able to reach an appreciable performance. The decision to include both revision and association into our experiments should not come as a surprise; by doing so, we are able to ask ourselves: Will the penalty of creating a package of independent linkage tree outweigh the additional generations needed to compensate for breaking those dependencies? We could conclude that, no, it does not. There is still room for debate though, as the convergence for association is more favourable in comparison to the compression variants.

6.3. Future Work

Unfortunately, there will always be some time constraints, as we could not explore more designs nor run more experiments. For those interested in taking the baton, there is always more future work to do. In Section 6.3.1 we discuss the experiments that still could be performed on the current design of GPU-GOMEA and the packagers. In Section 6.3.2 We discuss the potential in regards to ILG. Next in Section 6.3.3 we discuss a potential solution due to lacking performance of the fitness evaluation kernel. And last in Section 6.3.4, we will suggest an alternative approach to design parallel acceleration in regards to GOMEA.

6.3.1. Experiments

The experiments only provide a limited amount of insight, which did hinder us to deduce strong conclusions. One obvious part that is left unexplored, is the full instance space for the μ-variable and the population size. Second, the largest problem instance is still relatively small. We got a hint by looking at the acceptable μ-variable to derive if our packagers were even plausible for larger instances; we can only know for sure if we run more experiments. Third, with the 3D grid instances, we only expanded the number of edges per vertex with one additional problem description. However, to explore the dependencies that resided within those edges, more experiments should be run based on problem instances with increasingly more edges per vertex. At last, we only explored a specific problem description, however, there are many more to consider. Take for instance a random graph, graphs with different weights, etc.

6.3.2. ILG Construction

Association is by far the closest we got in this thesis to accelerating GOMEA using parallelisation. If we recall our conclusion, the convergence rate—backed up with our knowledge about linkage and dependencies—makes it an interesting packager to expand upon. The majority of its performance loss is contributed to by the excessive communication between the host and the device. Being able to create a graph on the device would be one step of many to create a more competitive packager.

6.3.3. Partial Evaluation

Fitness evaluation is the largest contributor to the total run time, and thus takes a heavy hit into the performance. For contamination—which is uniquely is able to construct the package in $O(1)$ time—the process for fitness evaluation takes up 95 percent of the total run time. Because of this, now GPU-GOMEA does not look very interesting in regards to discrete-GOMEA. A performance boost is needed if we want to see GPU-GOMEA become relevant. The question thus arises: Is it possible to optimise evaluation? The short answer being: yes, it is.

First and foremost, we must realise what goes wrong. Fitness evaluation takes such a relatively long time to complete simply because we are required to iterate over all edges; it will take $O(|E|$ to complete the evaluation process. Thus, to reduce the runtime we also must reduce the complexity. The solution revolves around realising that we do not need to evaluate every edge inside the graph. If a vertex switches to another partition, the only edges involved are those connected to the vertex. Therefore, for each donation representing a switch in partition, we should be able to calculate the contribution of that

donation. Instead of calculating the fitness for the complete individual, we simply update the original fitness with the contribution of a donation. This heavily reduces the number of iterations needed. This is known as *partial* evaluation.

To accommodate this, some design changes are necessary. We want to be able to assign a contribution function to each and every donation. This might seem challenging to achieve with the restrictions of kernels. Alternatively, we could construct a single kernel which is able to retrieve the the list of neighbours within an optimistic $O(1)$ time. This is not achievable with an undirected graph represented as a set of edges. By interpreting our instance as a directed graph such that each edge is represented twice, we are able to get the neighbours by scanning the edge list up and down. These hurdles should first need to be overcome if we want to see major performance improvements.

6.3.4. Parallel Linear Algebra

We have designed GPU-GOMEA without the assistance of any libraries, which comes with some pros and cons. One of the benefits is that we are not restricted by the limitation a library could create. By designing each kernel ourselves, we are able to fully express our needs without encountering any overhead due to linking multiple pre-made processes. Second, there is the benefit of portability; without a library, we are only dependent on the CUDA compilation tools. This sounds less problematic than it actually is. Without a proper package manager, it is surprisingly hard to get some of the libraries. Besides availability, versioning and maintenance also seem to be an issue. Additionally, compatibility heavily depends on the hardware in use, which adds the possibility of features getting deprecated.

However, this all does not imply that starting from scratch only comes with benefits. One of the downsides of not using any libraries is the concept of re-inventing the wheel. There are many architectural decisions made that—in hindsight—could have been simplified. One of the decisions made is to not use the *Thrust* library, despite it being bundled with the CUDA compilation tools, which frankly had little backup to do so. In the end GPU-GOMEA makes us of its own vector object which is noticeably similar to its counterpart in Thrust.

If we call back to our introduction, a paragraph is dedicated to the relation between AI and parallel computing. This especially is relevant for neural networks. I would like to have a hot take and state that the popularity could be both contributed to the multi-purpose properties, but also to the simplicity of the design. Because it requires only linear algebra, there is the possibility for optimisation on different fronts. Multiple fields are able to have their own take on it; one could dive into the equations without any knowledge about the low level implementation. Additionally, there is some development in the hardware space to assist general purpose CPUs with dedicated AI accelerators. Beyond making use of silicon, there is some research in regards to embedding neural networks onto analogue computing engines to increase its performance.[4] This highlights the importance of linear algebra in regards to high performance computing and the interest in dedicated hardware.

One question we should ask ourselves is: What do neural network engineers use nowadays? It might not be surprising that they do not write low level CUDA processes, instead they make use of heavily optimised abstractions of said processes. This has the benefit that both low and high level processes can be optimised without requiring knowledge from either. Python is probably the industry standard when it comes to scientific computing. This is partially due to the language itself, but also due to the extensive packages available. Among them are packages such as TensorFlow and NumPy, which are probably one of the most popular packages in regards to machine learning and scientific computing respectively. In response to NumPy, the package CuPy is brought to life, which accelerates some of these processes using GPGPU. With CuPy, we should still be able to construct our custom kernels if needed, providing us a situation where it seems that we can both have the cake and eat it.

If we desire the simplification of our design, instead of relying on CUDA libraries, I would propose an alternative route: to simplify GPU-GOMEA we should look into the transition from kernels to matrix operations. With a set of matrix operations, we are able to benefit from low level optimisation, while we focus on the high level ones. To support my claim, let us have a look at Listing 6.1.

Notice that we are able to express our design within 46 lines of code, which would have taken us 1000+ lines if we have considered writing our own CUDA kernels. If we compare the performance between this and our GPU-GOMEA, we see that both are almost indistinguishable. There are of course some elements missing—such as the construction of the linkage tree—which actually makes GPU-GOMEA hard to implement, however, it should provide us with an interesting alternative.

```
import cupy as np

class UnivariateEA:
  def __init__(self, obj):
    self.obj = obj

  @staticmethod
  def optimal_mixing(problem):
    p_fitness = problem.fitness()
    parents = problem.pop

    x, y = problem.pop.shape
    idx = np.random.choice(x, size=(x, y))
    problem.pop = np.choose(idx, problem.pop)

    o_fitness = problem.fitness()
    revert = np.sign((o_fitness - p_fitness) * 2 + 1)

    idx = revert * np.arange(1, x+1)
    idx = (idx[idx < 0] * - 1) - 1
    problem.pop[idx,:] = parents[idx,:]

  def __call__(self, ind, *args):
    setattr(self.obj, 'step', self.optimal_mixing);
    return self.obj(ind, *args)

@UnivariateEA
class OneMax:
  def __init__(self, i, genes):
    self.ind = i
    self.genes = genes
    self.pop = np.random.choice([0,1], size=(i, genes))

  def fitness(self):
    return np.sum((self.pop + 1) // 2, axis=1)

  def max_fitness(self):
    return self.fitness().max()

  def __str__(self):
    return f'{self.fitness()}'

  def solve(self, solution):
    while self.max_fitness() < solution:
      self.step()
    print(self)
```

Listing 6.1: An implementation for univariate EA to solve one-max problem instances, using only matrix operations.

Bibliography

Articles

[1] Francisco Barahona et al. “An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design”. In: 36.3 (June 1988), pp. 493–513. ISSN: 0030-364X.

[2] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. “Set Partitioning via Inclusion-Exclusion”. In: *SIAM Journal on Computing* 39.2 (2009), pp. 546–563. DOI: 10.1137/070683933.

[3] Yu-Rong Chen et al. “Parallel UPGMA Algorithm on Graphics Processing Units Using CUDA”. In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. 2012, pp. 849–854. DOI: 10.1109/HPCC.2012.120.

[4] Yuan Du et al. “An Analog Neural Network Computing Engine Using CMOS-Compatible Charge-Trap-Transistor (CTT)”. In: 38.10 (Oct. 2019), pp. 1811–1819. ISSN: 0278-0070. DOI: 10.1109/TCAD.2018.2859237.

[11] “Independent sets with domination constraints”. In: *Discrete Applied Mathematics* 99.1 (2000), pp. 39–54. ISSN: 0166-218X. DOI: https://doi.org/10.1016/S0166-218X(99)00124-9.

[12] Yann LeCun et al. “Fast Convolutional Nets With FBFFT: A GPU Performance Evaluation”. In: *Journal of Machine Learning Research* 9.Nov (2008), pp. 1469–1489.

[14] M Luby. “A Simple Parallel Algorithm for the Maximal Independent Set Problem”. In: STOC ’85. Providence, Rhode Island, USA: Association for Computing Machinery, 1985, pp. 1–10. ISBN: 0897911512. DOI: 10.1145/22145.22146.

[16] John Nickolls et al. “Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?” In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730.

[22] Dirk Thierens and Peter A.N. Bosman. “Optimal Mixing Evolutionary Algorithms”. In: *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’11. Dublin, Ireland: Association for Computing Machinery, 2011, pp. 617–624. ISBN: 9781450305570.

[23] Darrell Whitley. “A genetic algorithm tutorial”. In: *Statistics and Computing* 4.2 (June 1994), pp. 65–85. ISSN: 1573-1375. DOI: 10.1007/BF00175354. URL: https://doi.org/10.1007/BF00175354.

Books and Chapters

[5] M. R. Garey and D. S. Johnson. “[ND16] MAX CUT”. In: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. First Edition. W. H. Freeman, 1979. App. A, sec. 2.2, p. 210. ISBN: 0716710455.

[6] Jan Friso Groote and Mohammed Reza Mousavi. “Actions, Behaviour, Equivalence, and Abstraction”. In: *Modeling and Analysis of Communicating Systems*. Cambridge, MA, USA: The MIT Press, 2014. Chap. 2, sec. 1-2, pp. 7–10.

[7] Jan Friso Groote and Mohammed Reza Mousavi. *Modeling and Analysis of Communicating Systems*. Cambridge, MA, USA: The MIT Press, 2014.

[8] Jan Friso Groote and Mohammed Reza Mousavi. “Parallel Processes”. In: *Modeling and Analysis of Communicating Systems*. Cambridge, MA, USA: The MIT Press, 2014. Chap. 5, sec. 1, pp. 69–72.

[10] John L Hennessy and David A. Patterson. “Data Dependences and Hazards”. In: *Computer Architecture: a quantitative approach*. Morgan Kaufmann, 2011. Chap. 3, sec. 1.2, pp. 179–183.

[20] R.R. Sokal and C.D. Michener. *A Statistical Method for Evaluating Systematic Relationships*. Vol. 38. University of Kansas science bulletin. University of Kansas, 1958, pp. 1409–1438.

[21] Alfred H. Sturtevant. “Linkage”. In: *A History of Genetics*. First Edition. Cold Spring Harbor Laboratory Press, 2001. App. 6, p. 39. ISBN: 9780879696078.

Online Sources

[9] Mark Harris. *Optimizing Parallel Reduction in CUDA*. CUDA Webinar 2. 2010. URL: `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf` (visited on 03/15/2022).

[13] Yuan Lin and Vinod Grover. *Using CUDA Warp-Level Primitives*. 2018. URL: `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/` (visited on 03/15/2022).

[15] Maxim Naumov. *A Parallel Algorithm for Graph Coloring*. Graph Coloring: More Parallelism for Incomplete-LU Factorization. 2022. URL: `https://developer.nvidia.com/blog/graph-coloring-more-parallelism-for-incomplete-lu-factorization` (visited on 04/01/2022).

[17] NVIDIA. *Atomic Functions*. CUDA Toolkit Documentation. 2022. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions` (visited on 04/13/2022).

[18] NVIDIA. *Features and Technical Specifications*. CUDA Toolkit Documentation. 2022. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-mode` (visited on 04/13/2022).

[19] NVIDIA. *Features and Technical Specifications*. CUDA Toolkit Documentation. 2022. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications` (visited on 04/13/2022).