On Parallel Computation of Large Smooth-Degree Isogeny

by

Kittiphon Phalakarn

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Mehran Mozaffari Kermani Associate Professor, Dept. of Computer Science and Engineering, University of South Florida
Supervisor:	Anwar Hasan Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Members:	Guang Gong Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
	Mahesh Tripunitara Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal-External Member:	Alfred Menezes Professor, Dept. of Combinatorics and Optimization, University of Waterloo

Author's Declaration

Abstract

The computation of large smooth-degree isogenies is considered to be the most time-consuming task in isogeny-based cryptosystems and, to this end, recently several proposals have been made to speed it up. For implementation in software using a single core, De Feo et al. presented an optimal way to compute such isogenies. The multi-core setting is however far more intricate but offers various ways to reduce the computation time and is an active area of research. This thesis presents a study of speeding-up large smooth-degree isogeny computation with various forms of parallelism and consists of three contributions.

The first contribution of this thesis is two novel theoretical techniques for speeding-up the computation with parallelism. Our proposed technique, called *precedence-constrained scheduling (PCS)*, transforms the isogeny computation into a task scheduling problem with precedence constraints and utilizes several task scheduling algorithms to tackle the problem. Another proposed technique of ours is to formulate the isogeny computation as an integer linear program. Combining both techniques, we are able to reduce the theoretical cost of the isogeny computation by up to 13.02% from the state-of-the-art.

The second contribution of this thesis is two software implementations of the isogeny computation based on our PCS technique. We consider two execution environments for the implementations: one relies only on the parallelism provided by multi-core processors, and the other utilizes multi-core processors supporting the Intel's Advanced Vector eXtensions (AVX) technology. To our best knowledge, we are the first to utilize both parallelization technologies for the isogeny computation. Also, to achieve effective implementations, we modify PCS for each execution environments and equip both implementations with a synchronization handling technique. The implementation results show up to 14.36% speed-up for the first implementation and up to 34.05% speed-up for the second implementation.

The third contribution of this thesis is two applications of using learning-based optimizations to speed-up the parallel isogeny computation. We consider the genetic algorithm and the reinforcement learning algorithm and detail our design rationale when instantiating both algorithms for our problem. From experimental results, the genetic algorithm is able to find a better approach for the isogeny computation. The approach found is nontrivial and is up to 9.95% faster than human's heuristic. On the other hand, the reinforcement learning lags PCS by as small as 2.73%. We use the experimental results of the reinforcement learning to argue that PCS may be nearly or even optimal for the computation.

Acknowledgements

I would like to thank my supervisor, Prof. Anwar Hasan, for his guidance and support throughout the completion of my degree. It would not be possible for me to complete this thesis without his supervision, and I deeply appreciate all his helps—academically and mentally—during difficult times. In addition, I would like to thank Ripple for providing financial support through a prestigious Ripple Graduate Fellowship.

I would also like to thank other members of the thesis examining committee—Prof. Guang Gong, Prof. Mahesh Tripunitara, Prof. Alfred Menezes, and Prof. Mehran Mozaffari Kermani—for their comments and suggestions, which greatly improve the quality of this thesis. Most of them have taught me in various courses, and I am incredibly grateful for all the knowledge I have gained from them. Additionally, I would like to thank Prof. David Jao, Prof. Mark Aagaard, and Prof. Douglas Stebila as well for the knowledge from their courses.

Furthermore, I would like to thank Prof. Vorapong Suppakitpaisarn and Prof. Francisco Rodríguez-Henríquez for their supports as the co-authors of my publications. It was an honor for me to have an opportunity to collaborate with them. Apart from this, I would like to thank Prof. Jason LeGrow, Prof. Ruben Niederhagen, and all anonymous reviewers from ACISP 2022 and TCHES 2023 for their constructive feedback on my work. Regarding the implementations of my proposed techniques, I would like to thank Cervantes-Vázquez et al. [16], Cheng et al. [20], and Longa et al. [68], for sharing their implementations, which I use as a basis for my implementations and comparisons.

Lastly, I would like to thank my family, friends, colleagues, and staffs—in Canada and other different countries around the world—who have provided me with tremendous supports during my time at the University of Waterloo. I am really thankful for their kind words and encouragements.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

c_{mul}, c_{iso} Costs of single point multiplication and degree-ℓ isogeny evaluation.

$\mathcal{C}_1(\mathcal{S})$, $\mathcal{C}_K^{\mathrm{PCP}}(\mathcal{S})$, $\mathcal{C}_K^{\mathrm{CCP}}(\mathcal{S})$, $\mathcal{C}_K^{\mathrm{Hu}}(\mathcal{S})$, $\mathcal{C}_K^{\mathrm{CG}}(\mathcal{S})$, $\mathcal{C}^{\mathrm{MP}}(\mathcal{S})$ Cost functions for strategy.

$\mathcal{C}_1^*(e)$, $\mathcal{C}_K^{\mathrm{PCP}^*}(e,k)$, $\mathcal{C}^{\mathrm{MP}^*}(e,r)$ Costs of optimal strategy.

$\mathcal{D}_{\mathcal{S}} = (\mathcal{V}_{\mathcal{D}_S}, \mathcal{E}_{\mathcal{D}_S})$ Task dependency graph of strategy.

E Elliptic curve.

$\mathcal{E}_{\mathcal{S},\mathrm{mul}}$, $\mathcal{E}_{\mathcal{S},\mathrm{iso}}$, $\mathcal{E}_{\mathcal{S},\mathrm{mul},i}$, $\mathcal{E}_{\mathcal{S},\mathrm{iso},i}$ Sets of edges in strategy of specific operation and column.

$\mathbb{F}_p$, $\mathbb{F}_{p^2}$, $\mathbb{F}_{p^k}$ Finite fields.

K Number of cores.

$[\ell]P$ Point multiplication by $[\ell]$.

$L(\mathcal{S})$ Linear representation of canonical strategy.

$\mathcal{L}(u)$, $\mathcal{L}_{\mathrm{CG}}(u)$ Label functions for vertex.

$\mathbf{M} = (\mathbf{S}, \mathbf{A}, \mathbf{R}, \mathbf{P})$ Markov decision process.

ϕ Isogeny.

$Q(s,a)$, $Q^*(s,a)$ Action-value function

$\mathcal{S} = (\mathcal{V}_{\mathcal{S}}, \mathcal{E}_{\mathcal{S}})$ Strategy.

$\mathcal{S} = \langle S_1, \ldots, S_t \rangle$ Scheduling.

$\mathcal{T}_e = (\mathcal{V}_e, \mathcal{E}_e)$ Graph of all possible operations for computing ℓ^e-isogeny.

$V(s)$, $V^*(s)$ State-value function

Chapter 1

Introduction

Cryptography plays an important role in protecting the communication system and securing information from malicious adversaries. One particular type of cryptosystems called *public-key* cryptosystems allows us to exchange or verify information without the need of shared secrets. Two well-known public-key cryptosystems are the RSA proposed in 1978 by Rivest, Shamir, and Adleman [79], whose security is based on the integer factorization problem, and elliptic curve cryptosystems (ECC) proposed independently in 1985 by Koblitz [52] and Miller [69], whose security is based on the elliptic curve discrete logarithm problem.

However, since the publication of Shor's algorithm [86], both RSA and ECC have been considered to be insecure against future quantum computers. With continual progress in the design of quantum computers, researchers in cryptography have to consider other hard problems as bases for their cryptosystems. Those cryptosystems which are constructed to be resistant against quantum attacks are called *quantum-resistant*, or *post-quantum*, cryptosystems. Currently, research in post-quantum cryptography focuses on six different directions [9, 72]: code-based cryptography, hash-based cryptography, isogeny-based cryptography, lattice-based cryptography, multivariate polynomial cryptography, and secret-key cryptography.

Isogeny-based cryptography is much newer than other post-quantum cryptographic schemes, specifically, those based on codes and lattices. Isogeny-based cryptosystems were first proposed by Couveignes in 1996 [25] and were later independently discovered by Rostovtsev and Stolbunov in 2006 [82, 88]. Recent isogeny-based cryptographic schemes include hash functions [17, 26], the Supersingular Isogeny Diffie-Hellman (SIDH) key exchange [45], the Supersingular Isogeny Key Encapsulation (SIKE) mechanism [44], the

Commutative SIDH (CSIDH) [15], Verifiable Delay Functions (VDFs) [19, 32], and the Short Quaternion and Isogeny Signature (SQISign) scheme [31]. We note that a key-recovery attack exploiting the auxiliary elliptic-curve points of SIDH/SIKE has recently rendered SIDH/SIKE completely insecure [14, 64, 81], but there is no known way to apply similar attacks to the general isogeny problem.

1.1 Motivation

The computation to find the curve and point images of isogenies, required by many cryptosystems, is time-consuming. There have been various proposals (see below) to speed-up the computation to obtain low-latency implementations of those protocols. Also, the speed-up is of interest for VDFs. A VDF is a function that cannot be computed in less time than a prescribed delay. Thus, the function must crucially be as sequential as possible, in the sense that there should not exist any effective parallelization technique that yields a significant acceleration in its computation. In this work, we consider the amount of effective parallelization obtained from isogeny computations, which can be useful for a parameter selection in isogeny-based VDFs. Our focus is on the computation of large smooth-degree isogenies with degree ℓ^e, where ℓ is typically a small prime.

It is known that the best way of performing this task is through the computation of a sequence of degree-ℓ isogenies using Vélu-like formulas and point multiplications by $[\ell]$. The first work which considered this problem is by De Feo et al. [30]. The authors started with an abstraction of the computation called a *strategy* and associated a *cost* with it. The cost of a strategy theoretically represents the execution time of the isogeny computation corresponding to that strategy when it is implemented. In their paper, a dynamic programming equation for mathematically constructing a strategy with the least cost, called an *optimal strategy*, is proposed. These optimal strategies are then utilized in many implementations to reduce execution times [59, 68]. Apart from this, several techniques were introduced to speed-up isogeny computation taking into account various arithmetic aspects of the underlying field [5, 23, 29, 84].

In order to speed-up the computation, one can also adopt a technology especially designed for exploiting paralellism, such as vector instructions of Intel's Advanced Vector eXtensions (AVX). By these special instructions, multiple operations can be performed simultaneously on vectors. For the latest generation of AVX, called AVX-512, each vector (consisting of an array of data) is of length 512 bits and can be operated as eight 64-bit elements, meaning that eight 64-bit operations can be performed within a similar time as a single 64-bit operation. The advantages of AVX-512 have been exploited to speed-up the isogeny computation by a few works [20, 56]. In [20], the authors proposed optimizations in several layers, including base-field arithmetic, extension-field arithmetic, elliptic curve arithmetic, and isogeny computation. Combining all those techniques, the execution time of their implementation is 2.40 times faster compared to that of [68].

To further improve the speed of the isogeny computation, many researchers [16, 57, 58] turned to multi-core platforms, on which multiple operations can be performed simultaneously on different cores. We note that the use of vector instructions is somewhat similar to the multi-core setting, but in the former, the same instruction is performed on all vector elements. As the execution environment changes, strategies and the cost function have to be revised accordingly. The earliest work that analyzes strategies and the cost functions specifically for the multi-core setting is due to Hutchinson and Karabina [42]. Their main contributions are a formalization of a parallel isogeny computation on multi-core platforms called *per-curve parallel (PCP)* and a dynamic programming algorithm constructing optimal strategies under this PCP parallel computation. The experimental results show that, in the multi-core setting, optimal strategies under PCP lead to lower costs compared to original optimal strategies of [30] designed for serial computation. This implies that serial optimal strategies of [30] are not necessarily optimal in the multi-core environment. Looking at the experimental results, the theoretical costs of optimal strategies under PCP are up to 24%, 40%, and 51% cheaper than the costs of optimal strategies of [30] when the number of cores is two, four, and eight, respectively. And when implementing the computation on a three-core platform using the techniques of [42] along with other optimizations, Cervantes-Vázquez et al. [16] could achieve more than 35% speed-up in the execution time compared to the serial implementation. These results clearly show an impact on the speeding-up of isogeny computation for multi-core platforms at the strategy-level.

Apart from PCP, [42] also proposed another parallelization technique called *consecutive-curve parallel (CCP)*, which can be considered as an enhanced version of PCP. From their experiments, costs of strategies under CCP are moderately less than those under PCP. Nonetheless, to the best of our knowledge, no software and hardware implementation utilizes CCP for the computation.

1.2 Research Problem

Although speeding-up the large smooth-degree isogeny computation at the strategy-level is important, there is no work that attempts to do so beyond PCP and CCP. This thesis studies the problem of constructing and parallelizing strategies for the large smooth-degree isogeny computation at the lowest cost/latency possible.

1.3 Contributions

The study of this thesis results in three main contributions:

Precedence-Constrained Scheduling (PCS) Technique. We propose our novel technique of computing the cost of a strategy in the multi-core setting called *precedence-constrained scheduling (PCS)*, which is by transforming a strategy into a precedence-constrained scheduling problem. Then, two scheduling algorithms—Hu's and Coffman-Graham's algorithms—are applied to evaluate the strategy. In addition, we formalize the optimization problem as an integer linear program (ILP). Since the resulting ILP is large and cannot be solved efficiently, we construct strategies by combining the ILP solutions for smaller problems. The experimental results when integrating PCS and ILP show a strategy cost reduction by up to 13.02%, compared to those from [42].

Two Parallel and Vectorized Implementations of PCS. We present two software implementations of the large smooth-degree isogeny computation, based on our proposed PCS parallelization. The first implementation solely considers multi-core parallelism, while the second implementation considers both multi-core parallelism and vectorization technology. To the best of our knowledge, this is the first time that two parallelization technologies are utilized together for the isogeny computation. We provide analyses and modifications on how to effectively apply our PCS to the unique execution environment of each implementation. From our benchmarkings, the execution times of our first implementation are up to 14.36% faster than those from [16] and the execution times of our second implementation are up to 34.05% faster than those from [20, 68].

Two Applications of Learning-Based Optimizations. We provide two applications of learning-based optimizations—genetic algorithms and reinforcement learning—to the problem of constructing and evaluating strategies in the multi-core setting, respectively, in order to achieve less cost. We discuss some possible design options that can be used for the instantiations of these learning-based algorithms and give our design rationale on how we select such options. Via experimental results, the genetic algorithm succeeds in constructing strategies with lower cost compared to the heuristic provided by humans. The cost can be reduced by up to 9.95% and the performance of the genetic algorithm tends to be better when the number of cores is larger. Regarding the reinforcement learning, it lags PCS for strategy evaluation. Nevertheless, the results of the reinforcement learning are as small as 2.73% close to those from PCS, and we argue that PCS may be nearly or even optimal for strategy evaluation.

1.4 Applicability of Our Contributions

We would like to emphasize that our proposals in this thesis are general frameworks which can be applied to other settings involving the computation of large smooth-degree isogeny. We only consider SIKE, specifically SIKEp751, as our case study since SIDH/SIKE were the main focus of the community and a high volume of research work and implementations were presented compared to other recent isogeny-based cryptosystems. Below we briefly discuss some possibilities of applying our contributions to other isogeny-based schemes and implementation settings. More extensive discussion can be found in Section 5.4.

CSIDH. In terms of implementation, the main difference between SIDH and CSIDH is isogeny degrees that need to be computed: SIDH works on degree-ℓ^e isogeny while CSIDH works on degree-$\ell_1 \ell_2 \cdots \ell_n$ isogeny. Despite this difference, the computation of both isogenies can be formulated similarly using the idea of *strategies*. Since this thesis is interested in strategy-level optimization for the computation, we strongly believe that our proposed frameworks have a potential to be adapted for the setting of CSIDH.

SQISign. Recently, the National Institute of Standards and Technology (NIST) has called for additional digital signature proposals to be considered in the post-quantum cryptography standardization process. Among all the submissions, SQISign is the only protocol based on isogeny, and its optimization in various aspects is expected to receive considerable attention from the research community. The source code of the NIST submission of SQISign shows the use of a strategy for its isogeny computation, and hence our proposed techniques can potentially optimize SQISign as well.

Implementations with Large Number of Cores. The implementations presented in this thesis considered processors with up to four cores due to the specification of our machines. Nonetheless, the experimental and implementation results suggest greater speed-up when utilizing processors with more cores. We note that this may result in a decrease in the efficiency of the implementation, leading to a trade-off between the number of additional cores used and the incremental speed-up to be achieved.

Hardware Implementations. We also expect the applicability of our contributions to hardware implementations. Because our techniques are high-level (i.e., strategy-level) optimizations, they are not dependent on the implementation technology. Nevertheless, by the unique characteristics of hardware design, some modifications to our proposed algorithms may be required, offering opportunities for a hardware/software co-design.

1.5 Organization

The rest of the thesis is organized as follows:

- Chapter 2 provides some background regarding supersingular elliptic curves, their arithmetics, and isogenies.
- Chapter 3 explains how large smooth-degree isogenies can be computed. This chapter also reviews the optimal strategy for the single-core setting reported in [30], and the state-of-the-art for the multi-core setting, PCP and CCP, from [42].
- Chapter 4 proposes our first contribution which are PCS and ILP techniques.
- Chapter 5 presents our second contribution which are two parallel and vectorized implementations of PCS.
- Chapter 6 details our third contribution which are two applications of learning-based optimizations to the parallel isogeny computation.
- Chapter 7 concludes the thesis and suggests directions for future works.

Chapter 2

Preliminaries

In this chapter, we review some preliminaries on supersingular elliptic curves and isogenies.

2.1 Elliptic Curves

2.1.1 Curve Equations and Arithmetics

We start with the definitions of elliptic curves and their rational points from [38] as follows.

Definition 2.1 (Elliptic curve and F-rational points)**.** An *elliptic curve E over a field F* is defined by

$$E/F : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1, a_2, a_3, a_4, a_6 \in F$ and $\Delta \neq 0$, where Δ is the *discriminant* of E defined by

$$\begin{aligned}
\Delta &= -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6,\\
d_2 &= a_1^2 + 4a_2,\\
d_4 &= 2a_4 + a_1a_3,\\
d_6 &= a_3^2 + 4a_6,\\
d_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2.
\end{aligned}$$

The set of *F-rational points* on E is

$$E(F) = \{(x, y) \in F \times F : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\} \cup \{\infty\}$$

where ∞ is the *point at infinity.*

Definition 2.2 (Simplified Weierstrass curve)**.** For a field F whose characteristic is not 2 or 3, an elliptic curve over F can be defined by

$$E/F : y^2 = x^3 + ax + b$$

where $a, b \in F$ and $\Delta = -16(4a^3 + 27b^2) \neq 0$.

Although the simplified Weierstrass curve is commonly known, various isogeny-based cryptosystems consider another curve called *Montgomery curve* as it leads to efficient arithmetic implementations [11]. Below is the definition from [24].

Definition 2.3 (Montgomery curve)**.** For a field F whose characteristic is not 2, a *Montgomery curve* over F is an elliptic curve defined by

$$E_{(a,b)}/F : by^2 = x^3 + ax^2 + x$$

where $a, b \in F$ and $b(a^2 - 4) \neq 0$.

The set $E_{(a,b)}(F)$ forms an abelian group under an operation $+$ as described below. Here, ∞ is the identity of the group. When it is clear from the context or the variables a and b are not relevant, we write $E_{(a,b)}$ and E as shorthand notations for $E_{(a,b)}(F)$.

1. *Identity*: For all $P \in E_{(a,b)}$, $P + \infty = \infty + P = P$.
2. *Negatives*: If $P = (x, y) \in E_{(a,b)}$, then $P + (-x, y) = \infty$. The point $(-x, y) \in E_{(a,b)}$ is denoted by $-P$. Note that $-\infty = \infty$.
3. *Point addition*: Let $P = (x_1, y_1) \in E_{(a,b)}$ and $Q = (x_2, y_2) \in E_{(a,b)}$ with $P \neq \pm Q$. Then, $P + Q = (x_3, y_3)$ where
$$x_3 = b\left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 - a \quad \text{and} \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1.$$
4. *Point doubling*: Let $P = (x_1, y_1) \in E_{(a,b)}$ with $P \neq -P$. Then, $[2]P = (x_3, y_3)$ where
$$x_3 = b\left(\frac{3x_1^2 + 2ax_1 + 1}{2by_1}\right)^2 - 2x_1 - a \quad \text{and} \quad y_3 = \left(\frac{3x_1^2 + 2ax_1 + 1}{2by_1}\right)(x_1 - x_3) - y_1.$$

From above, the formulas for point addition and point doubling involve field inversions, which may be considered time-consuming. To avoid such computations, elliptic curves and points can be represented in *projective form*.

Definition 2.4 (Projective form of Montgomery curve)**.** The *projective form* of a Montgomery curve $E_{(a,b)}$ is obtained by replacing (x, y) by $(X/Z, Y/Z)$ and (a, b) by $(A/C, B/C)$. As a result, the projective form is defined by

$$E_{(A:B:C)}/F : BY^2Z = CX^3 + AX^2Z + CXZ^2$$

where $(x, y) \in E_{(a,b)}$ is represented as $(X : Y : Z) \in E_{(A:B:C)}$ and ∞ is represented as $(0 : 1 : 0)$. Here, $(X : Y : Z)$ is called a *projective point*.

Under the projective form, the formulas for point *pseudo-addition* and point *pseudo-doubling* can be expressed, without field inversions, using $(X : Z)$ and $(A : C)$ coordinates. The computations are as follows:

- *Point pseudo-addition*: Let $P = (X_1 : Z_1) \in E_{(A:C)}$, $Q = (X_2 : Z_2) \in E_{(A:C)}$, and $R = P - Q = (X_3 : Z_3) \in E_{(A:C)}$ with $P \neq \pm Q$. Then, $P + Q = (X_4 : Z_4)$ where

$$\begin{aligned} X_4 &= Z_3[(X_1 - Z_1)(X_2 + Z_2) + (X_1 + Z_1)(X_2 - Z_2)]^2, \\ Z_4 &= X_3[(X_1 - Z_1)(X_2 + Z_2) - (X_1 + Z_1)(X_2 - Z_2)]^2. \end{aligned}$$

- *Point pseudo-doubling*: Let $P = (X_1 : Z_1) \in E_{(A:C)}$ with $P \neq -P$. Then, $[2]P = (X_2 : Z_2)$ where

$$\begin{aligned} X_2 &= 4C(X_1^2 - Z_1^2)^2, \\ Z_2 &= 4X_1Z_1(4(A + 2C)X_1Z_1 + 4C(X_1 - Z_1)^2). \end{aligned}$$

2.1.2 Point Multiplication

By the group law, we define the following definition of *point multiplication*.

Definition 2.5 (Point multiplication)**.** Let $P \in E$ and $\ell \in \mathbb{Z}^+$. Then,

$$[\ell]P = \underbrace{P + P + \ldots + P}_{\ell \text{ times}}, \qquad [0]P = \infty, \qquad [-\ell]P = -[\ell]P.$$

A naive algorithm of performing ℓ additions of P to ∞ is not polynomial-time in the size of ℓ, which is $b = \log_2(\ell)$ bits. In this subsection, we describe some polynomial-time algorithms which perform point multiplication. For these algorithms, let the binary representation of ℓ be $\ell_{b-1}\ell_{b-2}\cdots\ell_0$.

The first algorithm is the *double-and-add* algorithm [38], which is similar to the square-and-multiply algorithm for modular exponentiation. Algorithm 2.1 below shows the computation when considering the binary representation of ℓ from left to right (i.e., from ℓ_{b-1} to ℓ_0). The algorithm can be modified to perform from right to left in a similar manner.

Algorithm 2.1: Double-and-add algorithm for point multiplication.

```
   Input  : A point $P$ and a positive integer $\ell = \ell_{b-1}\ell_{b-2}\cdots\ell_0$
   Output: $[\ell]P$

1  $Q \leftarrow \infty$
2  for $i = b-1$ down to $0$ do
3      $Q \leftarrow [2]Q$
4      if $\ell_i = 1$ then $Q \leftarrow Q + P$
5  return $Q$
```

Algorithm 2.1 works correctly but is subjected to *side-channel attacks* since the number of times point addition in Line 4 is performed depends on ℓ. The power consumed during the calculation can be measured and provide information whether Line 4 is performed, resulting in a *power attack* [54]. Likewise, the timing information of the calculation can be utilized, resulting in a *timing attack* [53]. To counter these attacks, we consider the following well-known algorithm called the *Montgomery ladder* algorithm [71].

Algorithm 2.2: Montgomery ladder algorithm for point multiplication.

```
    Input  : A point $P$ and a positive integer $\ell = \ell_{b-1}\ell_{b-2}\cdots\ell_0$
    Output: $[\ell]P$

1   $Q_0 \leftarrow \infty$
2   $Q_1 \leftarrow P$
3   for $i = b-1$ down to $0$ do
4       if $\ell_i = 0$ then
5           $Q_1 \leftarrow Q_0 + Q_1$
6           $Q_0 \leftarrow [2]Q_0$
7       else
8           $Q_0 \leftarrow Q_0 + Q_1$
9           $Q_1 \leftarrow [2]Q_1$
10  return $Q_0$
```

It is not hard to prove by induction that $Q_0 = [\ell_{b-1}\ell_{b-2}\cdots\ell_i]P$ and $Q_1 = Q_0 + P$ after considering ℓ_i in the for loop. The algorithm is also applicable with point pseudo-addition of Montgomery curve, as the difference of Q_0 and Q_1 (which is P) is always available.

In addition to point multiplication $[\ell]P$, we are interested in the computation of $P+[\ell]Q$. The method of firstly computing $[\ell]Q$ and then adding it to P works for normal point addition but does not work for point pseudo-addition, since we need the difference of P and $[\ell]Q$. De Feo et al. in 2014 [30] proposed the following three-point Montgomery ladder, shown in Algorithm 2.3, to perform such calculation.

Algorithm 2.3: Three-point Montgomery ladder algorithm to compute $P+[\ell]Q$.

```
Input  : Three points $P$, $Q$, $R = P - Q$ and a positive integer $\ell = \ell_{b-1}\ell_{b-2}\cdots\ell_0$
Output: $P + [\ell]Q$
1  $A \leftarrow \infty$
2  $B \leftarrow Q$
3  $C \leftarrow P$
4  for $i = b - 1$ down to $0$ do
5      if $\ell_i = 0$ then
6          $C \leftarrow C + A$
7          $B \leftarrow B + A$
8          $A \leftarrow [2]A$
9      else
10         $C \leftarrow C + B$
11         $A \leftarrow A + B$
12         $B \leftarrow [2]B$
13 return $C$
```

Similar to (two-point) Montgomery ladder, one can show that $A = [\ell_{b-1}\ell_{b-2}\cdots\ell_i]Q$, $B = A+Q$, and $C = A+P$ after considering ℓ_i in the for loop. For point pseudo-addition in Lines 6 and 10, one can do so because the difference of A and C (which is P) and the difference of B and C (which is $P-Q=R$) are both available.

Example 2.6. We give an example of a computation of $P+[77]Q$, given three points P, Q, and $R = P - Q$, as shown in Table 2.1. The binary representation of 77 is 1001101_2.

i		**6**	**5**	**4**	**3**	**2**	**1**	**0**
ℓ_i		1	0	0	1	1	0	1
A	∞	Q	$[2]Q$	$[4]Q$	$[9]Q$	$[19]Q$	$[38]Q$	$[77]Q$
B	Q	$[2]Q$	$[3]Q$	$[5]Q$	$[10]Q$	$[20]Q$	$[39]Q$	$[78]Q$
C	P	$P+Q$	$P+[2]Q$	$P+[4]Q$	$P+[9]Q$	$P+[19]Q$	$P+[38]Q$	$P+[77]Q$

Table 2.1: A computation of $P + [77]Q$ using the three-point Montgomery ladder.

We note also that there were attempts to parallelize point multiplication using several techniques. This is beyond the scope of this work, and we refer the interested readers to [74, 78] for more details.

Finally, we define some terminologies related to point multiplication. When the field F is finite (i.e., $F = \mathbb{F}_{p^k}$ for some prime p and $k \in \mathbb{Z}^+$), the set $E(\mathbb{F}_{p^k})$ is also finite. The number of points in $E(\mathbb{F}_{p^k})$, denoted by $\#E(\mathbb{F}_{p^k})$, is called the *order* of E over $\mathbb{F}_{p^k}$. The *order* of a point $P \in E(\mathbb{F}_{p^k})$, denoted by $\mathrm{ord}(P)$, is the smallest positive integer d such that $[d]P = \infty$. We also define an ℓ-torsion subgroup following [87].

Definition 2.7 (ℓ-torsion subgroup)**.** Let E be an elliptic curve and ℓ be a positive integer. The *ℓ-torsion subgroup of E* is defined as $E[\ell] = \{P \in E : [\ell]P = \infty\}$.

2.1.3 Supersingular Elliptic Curves

The supersingularity of an elliptic curve is defined by its order as follows.

Definition 2.8 (Supersingularity)**.** An elliptic curve E over $\mathbb{F}_{p^k}$ is *supersingular* if p divides $p^k + 1 - \#E(\mathbb{F}_{p^k})$. Otherwise, E is *non-supersingular*.

Given a prime p, Bröker [13] presented how to efficiently construct a supersingular elliptic curve E over $\mathbb{F}_{p^2}$ with $\#E(\mathbb{F}_{p^2}) = (p \pm 1)^2$. This is very useful as we are able to guarantee the order of $E(\mathbb{F}_{p^2})$. We note that it is not known in general how to efficiently construct an elliptic curve with a given order.

2.2 Isogenies

The definition of an isogeny and its properties are given in [87].

Definition 2.9 (Isogeny)**.** Let E and E' be elliptic curves over F and their identity elements denoted by ∞ and ∞', respectively. An *isogeny* from E to E' is a morphism $\phi : E \to E'$ satisfying $\phi(\infty) = \infty'$. Two elliptic curves E and E' are *isogenous* if there is a surjective isogeny from E to E'.

To specify an isogeny from an elliptic curve E, one can specify its kernel, which is a finite subgroup of E, as a result of the following proposition.

Proposition 2.10. Let E be an elliptic curve and let Φ be a finite subgroup of E. There exist a unique elliptic curve $E' = E/\Phi$ and a separable isogeny $\phi : E \to E'$ satisfying $\ker \phi = \Phi$. The degree of ϕ, denoted by $\deg \phi$, is equal to the size of $\ker \phi$.

2.2.1 Vélu's Formulas

In [93], Vélu presented an explicit formula describing the equation of the image curve E' and computing $\phi(P)$ for $P \in E$, given E and Φ. In this work, we focus on the case where $\ker \phi$ is generated by a point $R \in E$ of prime order (i.e., $\ker \phi = \langle R \rangle$). We provide below the Vélu's formulas as described in [94].

Theorem 2.11 (Vélu's formulas)**.** Given an elliptic curve $E/F : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ and a point $R \in E(F)$ such that $\mathrm{ord}(R)$ is prime, the equation of the image curve E' and the explicit formula of $\phi : E \to E'$ where $\ker \phi = \langle R \rangle$ can be computed as follows:

1. If $\mathrm{ord}(R) = 2$, let $S = \{R\}$. Otherwise, let $S = \left\{[i]R : 1 \leq i \leq \frac{\mathrm{ord}(R)-1}{2}\right\}$.

2. For $Q = (x_Q, y_Q) \in S$, define the following quantities:

$$\begin{aligned} f_{Q,1} &= 3x_Q^2 + 2a_2x_Q + a_4 - a_1y_Q, \\ f_{Q,2} &= -2y_Q - a_1x_Q - a_3, \\ f_{Q,3} &= \begin{cases} f_{Q,1} & \text{if } [2]Q = \infty, \\ 2f_{Q,1} - a_1f_{Q,2} & \text{otherwise,} \end{cases} \\ f_{Q,4} &= (f_{Q,2})^2. \end{aligned}$$

3. Let $v = \sum_{Q \in S} f_{Q,3}$ and $w = \sum_{Q \in S} (f_{Q,4} + x_Q f_{Q,3})$, the equation of E' is

$$E'/F : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + (a_4 - 5v)x + (a_6 - (a_1^2 + 4a_2)v - 7w).$$

4. For $P = (x, y) \in E(F)$, the formulas for $\phi(P) = (x', y')$ are

$$\begin{aligned} x' &= x + \sum_{Q \in S} \left(\frac{f_{Q,3}}{x - x_Q} + \frac{f_{Q,4}}{(x - x_Q)^2} \right), \\ y' &= y - \sum_{Q \in S} \left(f_{Q,4} \frac{2y + a_1x + a_3}{(x - x_Q)^3} + f_{Q,3} \frac{a_1(x - x_Q) + y - y_Q}{(x - x_Q)^2} + \frac{a_1f_{Q,4} - f_{Q,1}f_{Q,2}}{(x - x_Q)^2} \right). \end{aligned}$$

There is an alternative approach to compute an isogeny due to Kohel [55]. We refer the interested readers to the mentioned work for more details.

Example 2.12. Considering the isogeny ϕ from $E/\mathbb{F}_{11} : y^2 = x^3 + x + 10$ with $\ker\phi = \langle R\rangle$ where $R = (9, 0) \in E(\mathbb{F}_{11})$, we follow Vélu's formulas to compute the equation of E' and $\phi(P)$ where $P = (4, 1)$:

1. Since $\mathrm{ord}(R) = 2$, we let $S = \{(9, 0)\}$.
2. For $Q = (9, 0)$, we have $f_{Q,1} = 2$, $f_{Q,2} = 0$, $f_{Q,3} = 2$, and $f_{Q,4} = 0$, respectively.
3. We have $v = 2$, $w = 7$, and thus E' is defined by $y^2 = x^3 + 2x + 5$.
4. Finally, for $P = (4, 1)$, we obtain $\phi(P) = (8, 4) \in E'/\mathbb{F}_{11}$.

Example 2.13. Considering the isogeny ϕ from $E/\mathbb{F}_{11} : y^2 = x^3 + x + 10$ with $\ker\phi = \langle R\rangle$ where $R = (1, 1) \in E(\mathbb{F}_{11})$, we follow Vélu's formulas to compute the equation of E' and $\phi(P)$ where $P = (4, 1)$:

1. Since $\mathrm{ord}(R) = 5$, we let $S = \{R, [2]R\} = \{(1, 1), (2, 8)\}$.
2. For $Q = (1, 1)$, we have $f_{Q,1} = 4$, $f_{Q,2} = 9$, $f_{Q,3} = 8$, and $f_{Q,4} = 4$, respectively.
   For $Q = (2, 8)$, we have $f_{Q,1} = 2$, $f_{Q,2} = 6$, $f_{Q,3} = 4$, and $f_{Q,4} = 3$, respectively.
3. We have $v = 1$, $w = 1$, and thus E' is defined by $y^2 = x^3 + 7x + 3$.
4. Finally, for $P = (4, 1)$, we obtain $\phi(P) = (1, 0) \in E'/\mathbb{F}_{11}$.

The computation of Vélu's formula takes $\tilde{O}(\ell)$ field operations where $\ell = \deg\phi$. A more recent work of Bernstein et al. [10] gives an improved formula called $\sqrt{}$élu's formula which takes $\tilde{O}(\sqrt{\ell})$ field operations. When considering implementation aspects, Adj et al. [1] applied the Karatsuba algorithm to $\sqrt{}$élu's formula, giving a running time of $O(\sqrt{\ell^{\log_2 3}})$. In this work, we call a task of finding (ϕ, E') from (E, Φ) as *isogeny construction* and a task of computing $\phi(P)$ as *isogeny evaluation*.

2.2.2 Isogeny-Based Cryptosystems

Although it is simple to explicitly describe ϕ given E and Φ, it is conjectured that finding ϕ or $\Phi = \ker\phi$ given E and E' is computationally infeasible, when $\deg\phi$ is sufficiently large. Below we define a computational problem which the security proofs of several isogeny-based cryptosystems are based on.

Definition 2.14 (Isogeny problem)**.** Given two supersingular elliptic curves E and E' which are isogenous, find the description of ϕ or its kernel Φ such that $\phi : E \to E' = E/\Phi$ is the isogeny between two given curves.

As previously mentioned, various cryptosystems can be built based on isogenies. For instance, hash functions, key exchanges, VDFs, and signature schemes. Our work mainly considers the parameter sets of SIKE, which has recently been rendered completely insecure. The descriptions of SIDH and SIKE are provided in Appendix A for reference. In Chapter 5, we explain how to generalize our work to other parameter sets defined by other isogeny-based cryptosystems that are not vulnerable to the recent attacks.

We end this chapter by showing how implementations of isogeny-based cryptosystems can be described by several abstraction layers. The two lowest layers are the prime-field and extension-field arithmetic layers for $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, respectively. Above these layers is the layer for elliptic curve arithmetic, including computations of point multiplication and (small-degree) isogeny. On top of that, we have the layer for large smooth-degree isogeny computation, which is the main focus of this work. Finally, the highest layer is the protocol layer. We note that each layer can be implemented and optimized independently. These abstraction layers can be illustrated as in Figure 2.1.

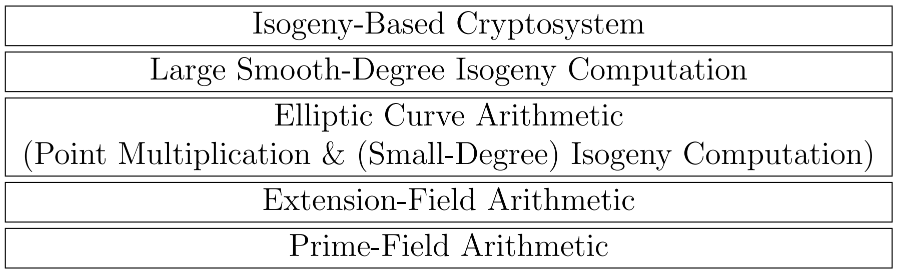

Isogeny-Based Cryptosystem
Large Smooth-Degree Isogeny Computation
Elliptic Curve Arithmetic
(Point Multiplication & (Small-Degree) Isogeny Computation)
Extension-Field Arithmetic
Prime-Field Arithmetic

Figure 2.1: Layers of abstraction for implementation of isogeny-based cryptosystem.

Chapter 3

Computing Large Smooth-Degree Isogenies

One main operation of isogeny-based cryptosystems is to compute $\phi : E \to E/\langle R \rangle$ for a point R of order ℓ^e where ℓ is a small prime and e is a positive integer. This chapter explores existing works of how ϕ can be computed in the single-core and multi-core settings.

3.1 Computation Paradigm

If one naively computes ϕ using Vélu's or $\sqrt{\text{é}}$lu's formulas mentioned in Subsection 2.2, then the computation will take $\tilde{O}(\ell^e)$ or $\tilde{O}(\sqrt{\ell^e})$ field operations, respectively. Since the degree of ϕ is smooth, it is best to decompose ϕ as a chain of degree-ℓ isogenies [30]:

$$\phi : E = E_0 \xrightarrow{\phi_0} E_1 \xrightarrow{\phi_1} E_2 \xrightarrow{\phi_2} \cdots \xrightarrow{\phi_{e-2}} E_{e-1} \xrightarrow{\phi_{e-1}} E_e = E/\langle R \rangle$$

where, for $0 \leq i < e$, $E_{i+1} = E_i/\langle [\ell^{e-i-1}]R_i \rangle$, $R_{i+1} = \phi_i(R_i)$, and $R_0 = R$. We note that $R'_i = [\ell^{e-i-1}]R_i$ is required in order to compute ϕ_i and E_{i+1}. This suggests the following procedure given in Algorithm 3.1 for computing $\phi_0, \ldots, \phi_{e-1}$.

We can describe Algorithm 3.1 using a graph with $\frac{e(e+1)}{2}$ vertices arranged in e columns and e rows as shown in Figure 3.1(a). Each vertex represents a point where points in each column are on the same elliptic curve. The vertex at the upper left corner represents the point R_0 and the leftmost column are points on E_0. The blue top-to-bottom arrows depict point multiplications by $[\ell]$ in Line 4 of the algorithm and the red left-to-right arrows depict isogeny evaluations in Line 6. Here, ϕ_{e-1}, E_e, and R_e are omitted.

Algorithm 3.1: Multiplication-based algorithm for computing degree-ℓ^e isogeny.

```
Input  : A supersingular elliptic curve $E$ and a point $R$ of order $\ell^e$
Output: $\phi_0, \ldots, \phi_{e-1}$ and $E/\langle R \rangle$
1 $E_0 \leftarrow E$; $R_0 \leftarrow R$
2 for $i = 0$ to $e-1$ do
3     $R'_i \leftarrow R_i$
4     for $j = 1$ to $e-i-1$ do $R'_i \leftarrow [\ell]R'_i$
5     Use Vélu's or $\sqrt{}$élu's formulas to compute $\phi_i$ and $E_{i+1}$ from $E_i$ and $\langle R'_i \rangle$
6     $R_{i+1} \leftarrow \phi_i(R_i)$
7 return $\phi_0, \ldots, \phi_{e-1}, E_e$
```

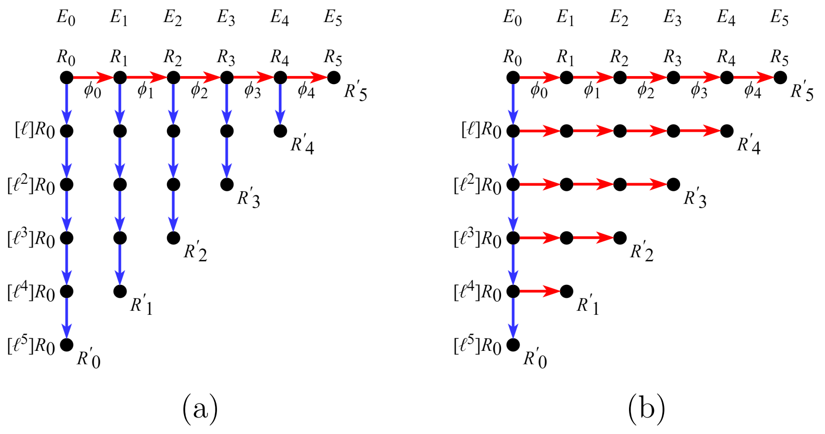

E_0 E_1 E_2 E_3 E_4 E_5
R_0 R_1 R_2 R_3 R_4 R_5
ϕ_0 ϕ_1 ϕ_2 ϕ_3 ϕ_4
R'_5 R'_4 R'_3 R'_2 R'_1 R'_0
$[\ell]R_0$
$[\ell^2]R_0$
$[\ell^3]R_0$
$[\ell^4]R_0$
$[\ell^5]R_0$
(a)
(b)

Figure 3.1: The multiplicative-based and isogeny-based strategies when $e = 6$.

In Figure 3.1(a), one might notice that R'_1 can also be computed by $R'_1 = \phi_0([\ell^{e-2}]R_0)$. This suggests another way of computing $\phi_0, \ldots, \phi_{e-1}$ as shown in Figure 3.1(b). In fact, there are several ways to compute smooth degree isogenies, as we describe next. For the above graphs, Figure 3.1(a) is referred to in [30] as *multiplicative-based* algorithm since it performs as many point multiplications as possible, while Figure 3.1(b) is referred to as *isogeny-based* algorithm as it performs as many isogeny evaluations as possible.

By considering how each point in the graph can be computed from other points, we define the graph $\mathcal{T}_e$ following [30] which shows all possible point multiplications by $[\ell]$ and isogeny evaluations among all vertices. For simplicity, vertices are referred by pairs of their columns and rows, i.e., vertex (i, j) refers to the point $[\ell^j]R_i$ in column i and row j.

Definition 3.1 (Graph of all operations)**.** The graph of all possible operations for computing degree-ℓ^e isogeny is defined as a directed graph $\mathcal{T}_e = (\mathcal{V}_e, \mathcal{E}_e)$ where

- the set of vertices $\mathcal{V}_e = \{(i,j) : 0 \le i,j < e;\ i+j < e\}$,
- the set of directed edges $\mathcal{E}_e = \mathcal{E}_{e,\text{mul}} \cup \mathcal{E}_{e,\text{iso}}$,
- the set of point multiplication $\mathcal{E}_{e,\text{mul}} = \{\langle (i,j),(i,j+1)\rangle : (i,j) \in \mathcal{V}_e;\ i+j < e-1\}$,
- the set of isogeny evaluation $\mathcal{E}_{e,\text{iso}} = \{\langle (i,j),(i+1,j)\rangle : (i,j) \in \mathcal{V}_e;\ i+j < e-1\}$.

A vertex $(i,j) \in \mathcal{V}_e$ is called a *leaf* if $i+j = e-1$.

Example 3.2. Below shows the directed graph $\mathcal{T}_6$. The directed edges in $\mathcal{E}_{6,\text{mul}}$ are in blue and those in $\mathcal{E}_{6,\text{iso}}$ are in red. The set of leaves is $\{(0,5),(1,4),(2,3),(3,2),(4,1),(5,0)\}$.

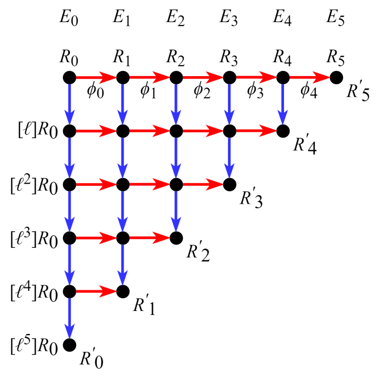

E_0 E_1 E_2 E_3 E_4 E_5
R_0 R_1 R_2 R_3 R_4 R_5
ϕ_0 ϕ_1 ϕ_2 ϕ_3 ϕ_4
R'_5
$[\ell]R_0$
R'_4
$[\ell^2]R_0$
R'_3
$[\ell^3]R_0$
R'_2
$[\ell^4]R_0$
R'_1
$[\ell^5]R_0$
R'_0

Figure 3.2: The directed graph $\mathcal{T}_6$.

For the rest of this thesis, we will omit all annotations in the graph, showing only vertices and edges. By definition, the graph $\mathcal{T}_e$ is a directed acyclic graph (DAG). We say that a vertex u reaches a vertex v if there exists a path from u to v (i.e., a sequence of edges $\langle\langle u,w_1\rangle,\langle w_1,w_2\rangle,\ldots,\langle w_n,v\rangle\rangle$) in a graph. For an instance of $\mathcal{T}_6$, $(0,1)$ reaches $(3,2)$ but $(1,4)$ does not reach $(2,1)$.

Next, we define a *strategy* for computing degree-ℓ^e isogeny as follows.

Definition 3.3 (Strategy)**.** A *strategy* $\mathcal{S}$ for computing degree-ℓ^e isogeny is a subgraph of $\mathcal{T}_e$ containing vertices $(0,0)$ and all leaves where $(0,0)$ reaches all leaves. A strategy $\mathcal{S}$ is *well-formed* if removing any edge from $\mathcal{S}$ results in a graph that is not a strategy.

Example 3.4. Two graphs shown in Figure 3.1 are well-formed strategies. Below shows three more subgraphs of $\mathcal{T}_6$. The graph (a) is not a strategy as $(0,0)$ does not reach $(3,2)$, (b) is a strategy but not well-formed since the edge $\langle(1,3),(1,4)\rangle$ and one in the set $\{\langle(3,1),(4,1)\rangle, \langle(4,0),(4,1)\rangle\}$ can be removed. Removing the edges $\langle(1,3),(1,4)\rangle$ and $\langle(3,1),(4,1)\rangle$ from (b) results in (c) which is a well-formed strategy.

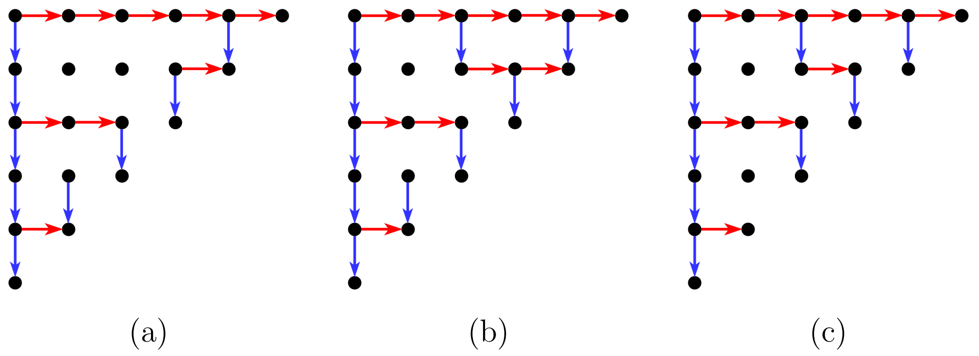


Figure 3.3: Examples of subgraphs of $\mathcal{T}_6$.

Example 3.5. Figure 3.4 shows all well-formed strategies for computing degree-ℓ^4 isogeny.

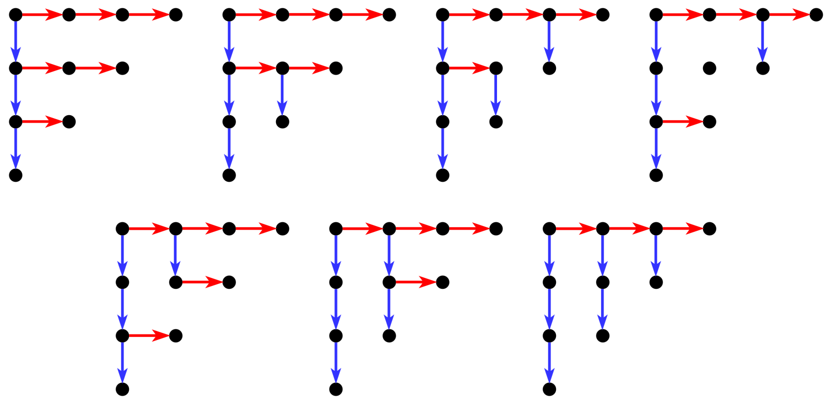


Figure 3.4: All well-formed strategies for computing degree-ℓ^4 isogeny.

Since strategies that are not well-formed have some unnecessary edges, we will consider only well-formed strategies in order to find an efficient strategy. We note that any well-formed strategy gives a valid algorithm to compute degree-ℓ^e isogeny, and in any well-formed strategy, a path from $(0,0)$ to any leaf is unique.

Now we look at how a strategy can be evaluated which defines the cost, i.e., the computation time, of a strategy. In this thesis, we consider two settings: the single-core setting as described in [30] and the multi-core setting as described in [42]. Both works were interested in the cost of a single point multiplication by $[\ell]$ (i.e., $Q \leftarrow [\ell]P$) and the cost of a single degree-ℓ isogeny evaluation (i.e., $Q \leftarrow \phi(P)$). We denote the costs of these operations as c_{mul} and c_{iso}, respectively.

3.2 Single-Core Setting

When only a single core is provided, we have to perform all operations sequentially. For a strategy $\mathcal{S}$, let $\mathcal{S} = (\mathcal{V}_{\mathcal{S}}, \mathcal{E}_{\mathcal{S}})$ with $\mathcal{E}_{\mathcal{S},\text{mul}} = \mathcal{E}_{\mathcal{S}} \cap \mathcal{E}_{e,\text{mul}}$ and $\mathcal{E}_{\mathcal{S},\text{iso}} = \mathcal{E}_{\mathcal{S}} \cap \mathcal{E}_{e,\text{iso}}$. We state the cost in the single-core setting as follows.

Definition 3.6 (Strategy cost in the single-core setting)**.** The cost of a strategy $\mathcal{S}$ in the single-core setting, denoted by $\mathcal{C}_1(\mathcal{S})$, is computed by

$$\mathcal{C}_1(\mathcal{S}) = \#\mathcal{E}_{\mathcal{S},\text{mul}} \cdot c_{\text{mul}} + \#\mathcal{E}_{\mathcal{S},\text{iso}} \cdot c_{\text{iso}}.$$

Example 3.7. Consider three strategies for computing degree-ℓ^6 isogeny. The cost of the multiplicative-based strategy (Figure 3.1(a)) is $15c_{\text{mul}} + 5c_{\text{iso}}$, the cost of the isogeny-based strategy (Figure 3.1(b)) is $5c_{\text{mul}} + 15c_{\text{iso}}$, and the cost of the strategy given in Figure 3.3(c) is $9c_{\text{mul}} + 9c_{\text{iso}}$.

When the cost of a strategy is defined, it is natural to ask for a strategy that gives the least cost. We first state the definition of an optimal strategy.

Definition 3.8 (Optimal strategy)**.** Let c_{mul} and c_{iso} be fixed. A strategy $\mathcal{S}$, computing degree-ℓ^e isogeny, is *optimal* in the single-core setting if for any strategy $\mathcal{S}'$ computing the same degree isogeny, we have $\mathcal{C}_1(\mathcal{S}) \leq \mathcal{C}_1(\mathcal{S}')$.

Example 3.9. Let $c_{\text{mul}} = c_{\text{iso}} = 1$. Figure 3.5 shows two optimal strategies for computing degree-ℓ^6 isogeny with a cost of 16.

The problem of constructing an optimal strategy given e, c_{mul}, and c_{iso} has been extensively studied in [30]. We report some lemmas and an algorithm for constructing the optimal strategy derived from that work. We begin by giving the definition of a *canonical* strategy.

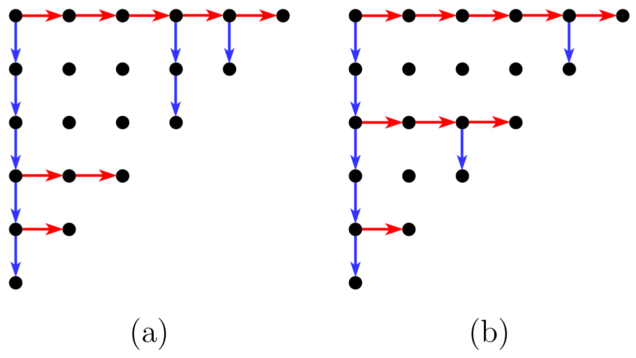


Figure 3.5: Some optimal strategies for computing degree-ℓ^6 isogeny when $c_{\text{mul}} = c_{\text{iso}} = 1$.

Definition 3.10 (Canonical strategy)**.** A *canonical* strategy for computing degree-ℓ^e isogeny is defined recursively as follows:

- If $e = 1$, then $\mathcal{T}_1$ is canonical.
- Otherwise, let $\mathcal{S}_n$, where $1 \leq n < e$, be a canonical strategy for computing degree-ℓ^n isogeny. If $\mathcal{S} = (\mathcal{V}_{\mathcal{S}}, \mathcal{E}_{\mathcal{S}})$ is constructed from $\mathcal{S}_n = (\mathcal{V}_{\mathcal{S}_n}, \mathcal{E}_{\mathcal{S}_n})$ and $\mathcal{S}_{e-n} = (\mathcal{V}_{\mathcal{S}_{e-n}}, \mathcal{E}_{\mathcal{S}_{e-n}})$ by the following steps, then $\mathcal{S}$ is canonical.
  1. Rename all vertices (i, j) in $\mathcal{S}_n$ to $(i, j + (e - n))$.
  2. Rename all vertices (i, j) in $\mathcal{S}_{e-n}$ to $(i + n, j)$.
  3. Construct $\mathcal{V}_{\mathcal{S}} = \mathcal{V}_{\mathcal{S}_n} \cup \mathcal{V}_{\mathcal{S}_{e-n}} \cup \{(0, j) : 0 \leq j < e - n\} \cup \{(i, 0) : 0 \leq i < n\}$ and $\mathcal{E}_{\mathcal{S}} = \mathcal{E}_{\mathcal{S}_n} \cup \mathcal{E}_{\mathcal{S}_{e-n}} \cup \{\langle (0, j), (0, j + 1)\rangle : 0 \leq j < e - n\} \cup \{\langle (i, 0), (i + 1, 0)\rangle : 0 \leq i < n\}$.

In brief, a canonical strategy with e leaves can be split into two canonical strategies with n leaves and $n - e$ leaves. Figure 3.6 depicts the process explained in Definition 3.10. We note that the number of all possible canonical strategies with e leaves is equal to the e-th Catalan number $C_e = \frac{1}{e+1}\binom{2e}{e}$. Next, we state two lemmas from [30] regarding optimal strategies and canonical strategies.

Lemma 3.11. All optimal strategies in the single-core setting are canonical.

Lemma 3.12. Let $\mathcal{S}$ be an optimal strategy, constructed from $\mathcal{S}_n$ and $\mathcal{S}_{e-n}$ as in Definition 3.10. Then, $\mathcal{S}_n$ and $\mathcal{S}_{e-n}$ are optimal strategies.

We note that a canonical strategy is not necessarily optimal.

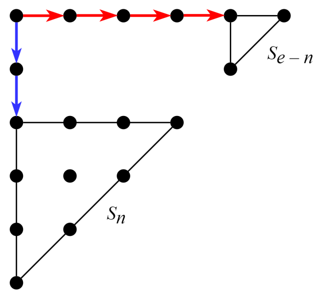


Figure 3.6: A canonical strategy for computing degree-ℓ^e isogeny.

Lemma 3.12 from [30] states the optimal substructure of the problem. As a result, the cost of an optimal strategy for computing degree-ℓ^e isogeny can be calculated by the following recurrence. We abuse the notation $\mathcal{C}_1$ by defining $\mathcal{C}_1^*(e)$ as the cost of an optimal strategy with e leaves. The recurrence for $\mathcal{C}_1^*(e)$ is

$$\mathcal{C}_1^*(e) = \min_{1 \le n < e} \{\mathcal{C}_1^*(n) + \mathcal{C}_1^*(e-n) + (e-n) \cdot c_{\text{mul}} + n \cdot c_{\text{iso}}\}, \quad \mathcal{C}^*(1) = 0$$

which can be transformed into a dynamic-programming algorithm as given in Algorithm 3.2. In addition, the description of a strategy is also required for an implementation. It is mentioned in [44] that a canonical strategy $\mathcal{S}$ constructed from $\mathcal{S}_n$ and $\mathcal{S}_{e-n}$ can be represented by its linear representation $L(\mathcal{S})$, which is the list of $e-1$ integers, where

$$L(\mathcal{S}) = [e-n] \parallel L(\mathcal{S}_n) \parallel L(\mathcal{S}_{e-n}), \quad L(\mathcal{T}_1) = [\,].$$

Algorithm 3.2 also provides the linear representation of an optimal strategy.

Example 3.13. The linear representation for the multiplicative-based strategy in Figure 3.1(a) is $[5, 4, 3, 2, 1]$, that for the isogeny-based strategy in Figure 3.1(b) is $[1, 1, 1, 1, 1]$, that for an optimal strategy in Figure 3.5(a) is $[3, 1, 1, 2, 1]$, and that for another optimal strategy in Figure 3.5(b) is $[2, 2, 1, 1, 1]$.

3.3 Multi-Core Setting

In this setting, we are provided with $K \ge 2$ cores. At first, a K-time speedup from the single-core setting might be expected. However, since we need to compute R_i' in order to continue to the next column, the computation is quite restricted and we are not able to fully utilize all cores at all times during the computation. Nevertheless, having multiple cores helps us reduce the cost as discussed next.

Algorithm 3.2: Computing the cost and linear representation of a single-core optimal strategy for computing degree-ℓ^e isogeny.

```
Input  : $e$, $c_{\text{mul}}$, and $c_{\text{iso}}$
Output: The cost and linear representation of an optimal strategy for computing
        degree-$\ell^e$ isogeny
1 $\mathcal{C}_1^*[1] \leftarrow 0$; $L[1] \leftarrow [\,]$
2 for $e' = 2$ to $e$ do
3     $\mathcal{C}_1^*[e'] \leftarrow \infty$
4     for $n = 1$ to $e' - 1$ do
5         $c \leftarrow \mathcal{C}_1^*[n] + \mathcal{C}_1^*[e' - n] + (e' - n) \cdot c_{\text{mul}} + n \cdot c_{\text{iso}}$
6         if $c < \mathcal{C}_1^*[e']$ then
7             $\mathcal{C}_1^*[e'] \leftarrow c$
8             $L[e'] \leftarrow [e' - n] \parallel L[n] \parallel L[e' - n]$
9 return $\mathcal{C}_1^*[e], L[e]$
```

3.3.1 Computation Model

Before getting into the computation cost, we review the implicit restrictions of the degree-ℓ^e isogeny computation. Unlike the single-core setting, timing plays a crucial role here. Because now we can perform more than one operations at the same time, we have to be careful of which operations are performed first and when they are finished, as they depend closely on each other. This is very important for achieving the minimal cost in this setting. In this thesis, we state two restrictions of how a strategy is evaluated in parallel:

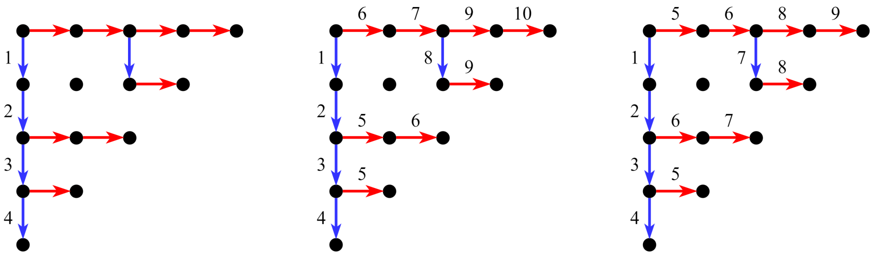
1. To perform a point multiplication by $[\ell]$ corresponding to an edge $\langle(i, j), (i, j + 1)\rangle$, the vertex (i, j) corresponding to the point $[\ell^j]R_i$ must have been computed.

2. To perform an isogeny evaluation corresponding to an edge $\langle(i, j), (i + 1, j)\rangle$, two vertices (i, j) and $(i, e - 1 - i)$ corresponding to the point $[\ell^j]R_i$ and R_i', respectively, must have been computed.

Even though the computation is restricted, there are still several ways of evaluating a strategy in parallel. To the best of our knowledge, even finding the optimal cost of a given strategy in a parallel setting is not an "easy" task, let alone finding an optimal strategy in the set of all well-formed strategies. We discuss these two problems in more detail in Chapter 4.

To have a clearer picture of the problem, we consider the following example of how a strategy is evaluated. In order to specify which operations are performed at which time, each edge is labeled with its finish time. The cost of evaluating a strategy is then labeled on the edge $\langle (e-2,0),(e-1,0)\rangle$, which must be performed as the last operation.

Example 3.14. Suppose we are provided with $K = 2$ cores and let $c_{\mathrm{mul}} = c_{\mathrm{iso}} = 1$. In the strategy below, at time 0, we only have the point R_0 corresponding to the vertex $(0,0)$. Although we have two cores, the only operation we are able to perform is the edge $\langle (0,0),(0,1)\rangle$, hence we can utilize only one core for this operation. This operation is finished at time 1 as it takes time $c_{\mathrm{mul}} = 1$. Again, at time 1, we can only take the edge $\langle (0,1),(0,2)\rangle$. We continue in this fashion until the edge $\langle (0,3),(0,4)\rangle$ is done at time 4 and we obtain R_0'. This first part of the evaluation is illustrated in Figure 3.7(a).



(a) (b) (c)

Figure 3.7: Examples of parallel evaluations of a strategy with $K = 2$.

At time 5, we now have three options: $\langle (0,0),(1,0)\rangle$, $\langle (0,2),(1,2)\rangle$, and $\langle (0,3),(1,3)\rangle$. Because we have two cores, we can choose up to two operations. In Figure 3.7(b), we choose the last two. Here, at time 7, we cannot perform $\langle (1,0),(2,0)\rangle$ and $\langle (2,0),(2,1)\rangle$ in parallel as R_2 is not yet computed. After performing the remaining operations, the last operation is done at time 10. Thus, the cost of the evaluation in Figure 3.7(b) is 10.

On the other hand, at time 5 Figure 3.7(c) chooses $\langle (0,0),(1,0)\rangle$ and $\langle (0,3),(1,3)\rangle$. At time 7, we can then perform two operations $\langle (2,0),(2,1)\rangle$ and $\langle (1,2),(2,2)\rangle$ simultaneously. This is allowed as all required points are already computed. The cost of the evaluation in Figure 3.7(c) is only 9.

We point out that among all well-formed strategies and all ways to evaluate them, an optimal strategy with its optimal evaluation has the cost of 9. This implies that Figure 3.7(c) is one of optimal strategies and evaluations. In Chapter 4, we explain how we (inefficiently) obtain this information. Also, we note that the strategy shown in this example is non-canonical.

The above example demonstrates that the multi-core setting is much more complicated than the single-core setting. In the rest of this subsection, we present the result of Hutchinson and Karabina [42] on constructing low-cost strategies and evaluations under some constraints called *per-curve parallel (PCP)* and *consecutive-curve parallel (CCP)*.

3.3.2 Per-Curve Parallel

For the purpose of analysis, Hutchinson and Karabina started with a simple evaluation of a strategy called *per-curve parallel (PCP)*. Under this evaluation,

(i) only operations of the form $\langle (i,j),(i+1,j)\rangle$ and $\langle (i,j'),(i+1,j')\rangle$ (i.e., isogeny evaluations from the same elliptic curve E_i) can be performed in parallel, and

(ii) point multiplications have to be done as the only operation during one time interval.

When there are n isogeny evaluations from E_i, the cost of performing these isogeny evaluations is $\lceil \frac{n}{K} \rceil \cdot c_{\text{iso}}$. Let $\mathcal{E}_{\mathcal{S},\text{iso},i} = \{\langle (i,j),(i+1,j)\rangle \in \mathcal{E}_{\mathcal{S},\text{iso}}\}$ be the set of isogeny evaluation edges from E_i in a strategy $\mathcal{S}$, the cost of evaluating $\mathcal{S}$ under PCP having K cores is

$$\mathfrak{C}_K^{\text{PCP}}(\mathcal{S}) = \#\mathcal{E}_{\mathcal{S},\text{mul}} \cdot c_{\text{mul}} + \sum_{i=0}^{e-2} \left\lceil \frac{\#\mathcal{E}_{\mathcal{S},\text{iso},i}}{K} \right\rceil \cdot c_{\text{iso}}.$$

Example 3.15. Consider the strategy given in Figure 3.7 with $K = 2$. Under PCP, the isogeny evaluations from E_0 take time $\lceil \frac{3}{2} \rceil \cdot c_{\text{iso}} = 2c_{\text{iso}}$, those from E_1 and E_2 take time $\lceil \frac{2}{2} \rceil \cdot c_{\text{iso}} = c_{\text{iso}}$, and that from E_3 takes time $\lceil \frac{1}{2} \rceil \cdot c_{\text{iso}} = c_{\text{iso}}$. Including five point multiplications performed separately, its cost under PCP is $5c_{\text{mul}} + 5c_{\text{iso}}$. We note that the order in which isogeny evaluations from the same curve are performed does not matter.

Even though PCP does not provide the least cost in the multi-core setting, it allows an extensive analysis to find an optimal strategy with smallest $\mathfrak{C}_K^{\text{PCP}}(\mathcal{S})$. For example, when $K \geq e-1$, the only optimal strategy under PCP is isogeny-based with the cost of $(e-1)c_{\text{mul}} + (e-1)c_{\text{iso}}$. While not stated in [42], it can be proved that there exists an optimal strategy under PCP that is canonical. Hence, we can find an optimal strategy under PCP by finding a least-cost canonical strategy.

Example 3.16. Consider all well-formed strategies for computing degree-ℓ^4 isogeny in Figure 3.4. Under PCP when $c_{\text{mul}} = c_{\text{iso}} = 1$ and $K = 2$, four out of seven strategies shown in Figure 3.8 are optimal with a cost of 7. Notice that there exists an optimal strategy that is canonical, but not all optimal strategies are canonical. When $K = 3 = e-1$, only the isogeny-based strategy is optimal with a cost of 6.

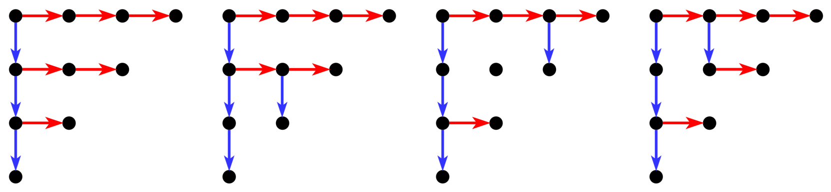


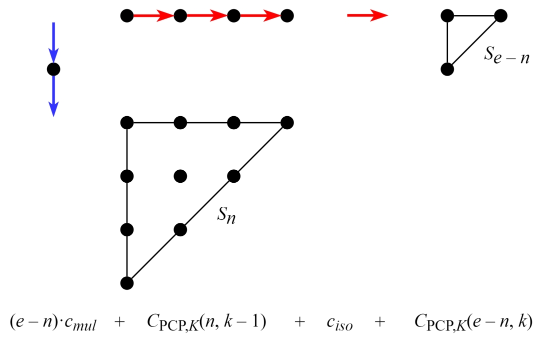
Figure 3.8: All optimal strategies under PCP when $c_{\text{mul}} = c_{\text{iso}} = 1$ and $K = 2$.

Since we can now focus on canonical strategies, we can make use of their substructures. The optimal substructure of the problem was exploited in [42]. This time, however, the subproblem is slightly different from its original problem. Referring to Definition 3.10 and Figure 3.6, some red edges above $\mathcal{S}_n$ can be performed in parallel with operations in $\mathcal{S}_n$. Hutchinson and Karabina took this into account and presented a recurrence describing $\mathcal{C}_K^{\text{PCP}}(\mathcal{S})$ as follows. Again, we abuse the notation and use $\mathcal{C}_K^{\text{PCP}^*}(e, k)$ to denote the cost under PCP for an optimal canonical strategy with e leaves and we can use only k out of K cores in the first evaluation for the edges in all $\mathcal{E}_{\mathcal{S},\text{iso},i}$, $0 \leq i < e - 1$.

Theorem 3.17. Let K, c_{mul}, and c_{iso} be fixed. The cost of an optimal strategy under PCP for computing degree-ℓ^e isogeny is $\mathcal{C}_K^{\text{PCP}^*}(e, K)$, which can be computed recursively using the following recurrence

$$\mathcal{C}_K^{\text{PCP}^*}(e,k) =$$
$$\begin{cases} 0 & \text{if } e = 1, \\ \min\limits_{1 \leq n < e} \{\mathcal{C}_K^{\text{PCP}^*}(n, k-1) + \mathcal{C}_K^{\text{PCP}^*}(e-n, k) + (e-n)\cdot c_{\text{mul}} + c_{\text{iso}}\} & \text{if } e > 1 \text{ and } k > 0, \\ \mathcal{C}_K^{\text{PCP}^*}(e, K) + (e-1)\cdot c_{\text{iso}} & \text{otherwise.} \end{cases}$$

We briefly describe the idea behind this recurrence. For the base case of $e = 1$, the cost is obviously 0 since $\mathcal{T}_1$ has no edges. For the second case, we split a strategy with e leaves in two with n and $e - n$ leaves following Definition 3.10. This time, the cost is computed as a sum of four parts shown in Figure 3.9. For each column of $\mathcal{S}_n$, there is an edge above it which is performed by one core, thus this leaves $k - 1$ cores for the first parallel execution of each column of $\mathcal{S}_n$. On the other hand, $\mathcal{S}_{e-n}$ is not affected and still has k cores for its first parallel execution of each column. The single red edge connecting the front and back parts takes time c_{iso} as we can perform it in the first parallel execution of that column (since $k > 0$) and there is no more edge in that column. The blue edges take time $(e - n) \cdot c_{\text{mul}}$ as usual. For the third case where $e > 1$ and $k = 0$, we have no core left and each column of $\mathcal{S}$ is performed in the second parallel execution. The cost of the first parallel execution is c_{iso} for each of the $e - 1$ columns.



S_{e-n}
S_n
$(e-n)\cdot c_{mul}$ + $C_{\mathrm{PCP},K}(n,k-1)$ + c_{iso} + $C_{\mathrm{PCP},K}(e-n,k)$

Figure 3.9: The cost $\mathcal{C}_K^{\mathrm{PCP}^*}(e,k)$ when $e>1$ and $k>0$.

Theorem 3.17 can be translated into a dynamic-programming algorithm as shown in Algorithm 3.3. It outputs the cost and the linear representation of an optimal strategy under PCP that is canonical. We note that in the case of $K=1$, $\mathcal{C}_K^{\mathrm{PCP}^*}(e,K)$ equals $\mathcal{C}_1^*(e)$. In addition, an algorithm calculating the cost under PCP was also presented in [16], however the results are suboptimal.

Algorithm 3.3: Computing the cost and linear representation of an optimal strategy for computing degree-ℓ^e isogeny with K cores under PCP.

Input : K, e, c_{mul}, and c_{iso}

Output: The cost and linear representation of an optimal strategy for computing degree-ℓ^e isogeny with K cores under PCP

```
1  for $e'=1$ to $e$ do
2      for $k=1$ to $K$ do
3          if $e'=1$ then $\mathcal{C}_K^{\mathrm{PCP}^*}[e'][k] \leftarrow 0$; $L[e'][k] \leftarrow [\,]$; continue
4          $\mathcal{C}_K^{\mathrm{PCP}^*}[e'][k] \leftarrow \infty$
5          for $n=1$ to $e'-1$ do
6              if $k>1$ then $c \leftarrow \mathcal{C}_K^{\mathrm{PCP}^*}[n][k-1] + \mathcal{C}_K^{\mathrm{PCP}^*}[e'-n][k] + (e'-n)\cdot c_{\mathrm{mul}} + c_{\mathrm{iso}}$
7              else $c \leftarrow \mathcal{C}_K^{\mathrm{PCP}^*}[n][K] + \mathcal{C}_K^{\mathrm{PCP}^*}[e'-n][k] + (e'-n)\cdot c_{\mathrm{mul}} + n\cdot c_{\mathrm{iso}}$
8              if $c < \mathcal{C}_K^{\mathrm{PCP}^*}[e'][k]$ then
9                  $\mathcal{C}_K^{\mathrm{PCP}^*}[e'][k] \leftarrow c$
10                 if $k>1$ then $L[e'][k] \leftarrow [e'-n] \parallel L[n][k-1] \parallel L[e'-n][k]$
11                 else $L[e'][k] \leftarrow [e'-n] \parallel L[n][K] \parallel L[e'-n][k]$
12 return $\mathcal{C}_K^{\mathrm{PCP}^*}[e][K], L[e][K]$
```

3.3.3 Consecutive-Curve Parallel

Under PCP, we cannot perform any operation in $\mathcal{E}_{\mathcal{S},\text{iso},i+1}$ while performing operations in $\mathcal{E}_{\mathcal{S},\text{iso},i}$, even though it is allowed to do so and some cores are idle. By this observation, [42] considers another constraint called *consecutive-curve parallel (CCP)*. Let $\mathcal{E}_{\mathcal{S},\text{mul},i} = \{\langle(i,j),(i,j+1)\rangle \in \mathcal{E}_{\mathcal{S},\text{mul}}\}$ be the set of point multiplication by $[\ell]$ edges for points in E_i in a strategy $\mathcal{S}$. Under CCP, while performing operations in $\mathcal{E}_{\mathcal{S},\text{iso},i}$, we are allowed to perform operations in $\mathcal{E}_{\mathcal{S},\text{iso},i+1}$ and $\mathcal{E}_{\mathcal{S},\text{mul},i+1}$ if they are ready to be done.

Because it is more flexible to perform operations in parallel under CCP, it is thus harder to analyze a strategy under this constraint. For this reason, [42] decided to consider only canonical strategies under CCP. As discussed before, operations in $\mathcal{E}_{\mathcal{S},\text{iso},i+1}$ can be performed after R'_{i+1} is computed. In the case that R'_{i+1} is computed by point multiplication edges in $\mathcal{E}_{\mathcal{S},\text{mul},i+1}$, all operations in $\mathcal{E}_{\mathcal{S},\text{mul},i+1}$ must be done first to obtain R'_{i+1}. By this, CCP use a greedy approach to choose which operations will be performed first while considering operations in $\mathcal{E}_{\mathcal{S},\text{iso},i}$ as follows:

(i) Operations in $\mathcal{E}_{\mathcal{S},\text{iso},i}$ are performed from bottom to top.

(ii) If an operation in $\mathcal{E}_{\mathcal{S},\text{mul},i+1}$ is available, then perform one operation in $\mathcal{E}_{\mathcal{S},\text{mul},i+1}$ and $K-1$ operations in $\mathcal{E}_{\mathcal{S},\text{iso},i}$.

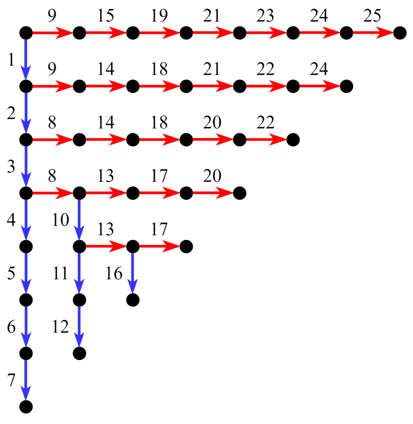
(iii) If operations in $\mathcal{E}_{\mathcal{S},\text{mul},i+1}$ are all done or there is no operation in $\mathcal{E}_{\mathcal{S},\text{mul},i+1}$, start performing operations in $\mathcal{E}_{\mathcal{S},\text{iso},i+1}$ as soon as all in $\mathcal{E}_{\mathcal{S},\text{iso},i}$ is finished.

(iv) If operations in $\mathcal{E}_{\mathcal{S},\text{iso},i}$ are all done before $\mathcal{E}_{\mathcal{S},\text{mul},i+1}$ is exhausted, then perform the remaining operations in $\mathcal{E}_{\mathcal{S},\text{mul},i+1}$ before starting $\mathcal{E}_{\mathcal{S},\text{iso},i+1}$.

We provide an example below for a better understanding of CCP.

Example 3.18. In Figure 3.10, we find the cost of the following strategy under (a) PCP and (b) CCP with $K = 2$ and $c_{\text{mul}} = c_{\text{iso}} = 1$. The cost under PCP is 25. For CCP, at time 8, two operations in $\mathcal{E}_{\mathcal{S},\text{iso},0}$ at the bottom are performed. At time 9 and 10, following (ii), one operation from $\mathcal{E}_{\mathcal{S},\text{iso},0}$ and $\mathcal{E}_{\mathcal{S},\text{mul},1}$ are done. At time 11, following (iv), the last operation in $\mathcal{E}_{\mathcal{S},\text{mul},1}$ is computed. At time 17, following (iii), $\langle(2,0),(3,0)\rangle$ and $\langle(3,3),(4,3)\rangle$ are performed simultaneously. The same happens at time 19. Following the greedy approach above, the cost under CCP is 22, which is less than that under PCP.

Given a canonical strategy $\mathcal{S}$, its cost under CCP, denoted by $\mathfrak{C}_K^{\text{CCP}}(\mathcal{S})$, can be computed by Algorithm 3.4. We define $\#\mathcal{E}_{\mathcal{S},\text{mul},e-1} = 0$ so that our notations work with the algorithm. When $K = 1$, [42] noted that Line 15 needs to be changed to $r \leftarrow r + \#\mathcal{E}_{\mathcal{S},\text{mul},i+1}$.


(a)

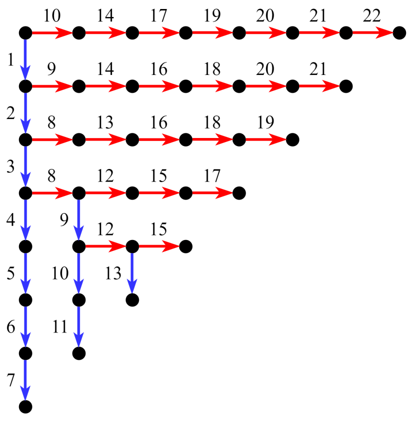

(b)

Figure 3.10: The evaluations of a strategy under PCP and CCP.

	K	2	3	4	5	6	7	8
PCP	Cost	25942.2	22521.6	20373.0	19197.0	17941.2	16978.8	16617.0
	% speedup	24.27	34.26	40.53	43.96	47.63	50.44	51.49
CCP S.O.	Cost	24247.2	21784.8	20941.2	20781.6	20781.6	20781.6	20781.6
	% speedup	29.22	36.41	38.87	39.34	39.34	39.34	39.34
CCP A.C.	Cost	25440.6	22200.6	20880.6	19825.2	19606.2	19218.6	18739.2
	% speedup	25.73	35.19	39.05	42.13	42.77	43.90	45.30
CCP P.O.	Cost	23890.2	20515.2	18252.6	17555.4	16482.0	16021.2	15294.6
	% speedup	30.26	40.11	46.72	48.75	51.89	53.23	55.35

Table 3.1: The cost of best strategies under PCP and CCP from experiments of [42].

While Algorithm 3.4 was given in [42], Hutchinson and Karabina stated that they could find no formula for the cost nor any optimal canonical strategy under CCP. In their experiments, using the parameters $(e, c_{\text{mul}}, c_{\text{iso}}) = (186, 25.8, 22.8)$, the optimal PCP cost and the best CCP cost of strategies sampled from the following sets are computed. The results are shown in Table 3.1 and are compared with $\mathcal{C}_1^*(e) = 34256.4$.

- CCP S.O.: all 1,623,160 optimal strategies under the single-core setting,
- CCP A.C.: 5,000,000 randomly sampled canonical strategies,
- CCP P.O.: 5,000,000 randomly sampled PCP canonical optimal strategies.

Algorithm 3.4: Computing the cost of a given canonical strategy for computing degree-ℓ^e isogeny with K cores under CCP.

```
Input  : $K$, $e$, $c_{\text{mul}}$, $c_{\text{iso}}$, $\#\mathcal{E}_{\mathcal{S},\text{mul},i}$ and $\#\mathcal{E}_{\mathcal{S},\text{iso},i}$ for $0 \le i < e-1$
Output: The cost of a given canonical strategy for computing degree-$\ell^e$ isogeny
        with $K$ cores under CCP

$s, t, \text{leftover} \leftarrow 0$; $r \leftarrow e-1$
for $i = 0$ to $e-2$ do
    $\text{binSize} \leftarrow \#\mathcal{E}_{\mathcal{S},\text{iso},i} - K$
    if $\text{binSize} < 0$ then
        $\text{binSize} \leftarrow K - \text{leftover}$
        $\text{leftover} \leftarrow \#\mathcal{E}_{\mathcal{S},\text{iso},i} - \text{binSize}$
    else
        $\text{binSize} \leftarrow \#\mathcal{E}_{\mathcal{S},\text{iso},i} - \text{leftover}$
    if $\text{leftover} = 0$ then
        $\text{binSize} \leftarrow \text{binSize} - K$
        $s \leftarrow s + 1$
    if $\text{binSize} > 0$ then
        if $\text{binSize} \ge (K-1) \cdot \#\mathcal{E}_{\mathcal{S},\text{mul},i+1}$ then
            $\text{binSize} \leftarrow \text{binSize} - (K-1) \cdot \#\mathcal{E}_{\mathcal{S},\text{mul},i+1}$
            $t \leftarrow t + \#\mathcal{E}_{\mathcal{S},\text{mul},i+1}$
            $s \leftarrow s + \lceil \frac{\text{binSize}}{K} \rceil$
            $\text{leftover} \leftarrow (-\text{binSize}) \bmod K$
        else
            $t \leftarrow t + \lceil \frac{\text{binSize}}{K-1} \rceil$
            $r \leftarrow r + \#\mathcal{E}_{\mathcal{S},\text{mul},i+1} - \lceil \frac{\text{binSize}}{K-1} \rceil$
            $\text{leftover} \leftarrow 0$
    else
        $r \leftarrow r + \#\mathcal{E}_{\mathcal{S},\text{mul},i+1}$
        $\text{leftover} \leftarrow 0$
return $r \cdot c_{\text{mul}} + s \cdot c_{\text{iso}} + t \cdot \max\{c_{\text{mul}}, c_{\text{iso}}\}$
```

As shown in Table 3.1, having multiple cores helps reduce the cost of computing smooth degree isogenies. With a sufficient number of cores and a carefully chosen strategy, the cost can be reduced by more than half. Nonetheless, the cost of computing smooth degree isogenies can be reduced further as we shall see in the next chapter.

3.4 Chapter Summary

In this chapter, we reviewed how a degree-ℓ^e isogeny can be computed by decomposing it into a chain of e degree-ℓ isogenies. We defined a strategy, which is a directed graph showing which operations are performed. Then, we gave the equation for the cost $\mathcal{C}_1(\mathcal{S})$ of a strategy $\mathcal{S}$ in the single-core setting, which is the sum of the cost of each operation in a strategy. As proposed in [30], an optimal cost can be computed by a recurrence and an optimal strategy can be constructed by a dynamic-programming algorithm.

The second half of this chapter dealt with the multi-core setting, where we are allowed to perform up to $K > 1$ operations at one time. The restrictions of how a strategy is evaluated was discussed. After that, we presented PCP and CCP, two ways of evaluating a strategy by [42]. By the simplicity of PCP, we have a recurrence describing the optimal cost under PCP. However, CCP is more complicated to analyze and no formula for the optimal cost under CCP has been found. We ended the chapter by giving experimental results of [42] which show a significant cost reduction from the single-core setting to the multi-core setting.

Chapter 4

Precedence-Constrained Scheduling (PCS) Technique

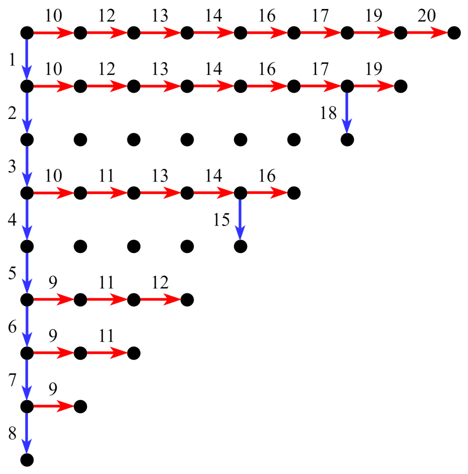
We have seen in the previous chapter that there are various techniques for constructing and evaluating strategies in the multi-core setting. This chapter takes a closer look at the problem and proposes a new approach to evaluate and construct strategies with less costs. The content of this chapter is based on [75].

Here, we first give an example showing that the cost of a canonical strategy under CCP is still not the least cost we can achieve.

Example 4.1. Let $e = 9$, $K = 3$, and $c_{\text{mul}} = c_{\text{iso}} = 1$. Below shows a canonical strategy which is optimal under PCP with the cost of 21. When calculating its cost using Algorithm 3.4, the cost under CCP is 20. The times at which each operation is finished are shown on the corresponding edges in Figure 4.1(a).

Consider another way of evaluating this strategy in Figure 4.1(b). Here, operations that can be performed simultaneously are not limited to ones in the same or consecutive elliptic curves. For instance, three isogeny evaluations ϕ_0, ϕ_1, and ϕ_2 are performed in parallel at time 11. As another example, during time 14, two isogeny evaluations ϕ_2, ϕ_3, and a point multiplication on E_4 are done at the same time. These are not permitted under CCP or PCP. As a result, we achieve a lower cost of 19 for this strategy and evaluation.

We will later prove that for this parameter setting, the least cost that any strategy and any evaluation can achieve is 19. Therefore, in this example, we provide an optimal strategy with its evaluation.


(a)

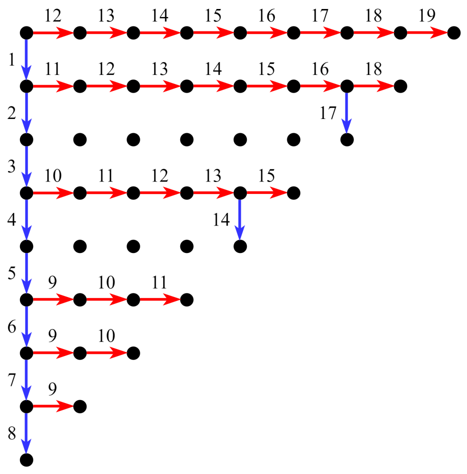

(b)

Figure 4.1: A canonical strategy which does not give the least cost under CCP.

By the above example, one can see that there is still room for improvement regarding this problem of finding optimal strategy and evaluation. It is important to note that, unlike the single-core setting, a strategy in the multi-core setting does not uniquely correspond to how it is evaluated. This does mean that, in order to obtain the least cost possible, we need to search for a strategy and its evaluation that give the least cost as a pair. Evaluating a good strategy in a wrong way might not give us a low cost. On the other hand, starting with a bad strategy will not give us a low cost under any evaluation. This makes it a challenging problem. Moreover, since it is possible that a least-cost strategy may not be canonical, we might not be able to utilize the recursive structure of canonical strategies to solve the problem.

4.1 Precedence-Constrained Scheduling Technique

The only cost measurements for a given strategy $\mathcal{S}$ that we are aware of are $\mathcal{C}_1(\mathcal{S})$ proposed in [30], and $\mathcal{C}_K^{\mathrm{PCP}}(\mathcal{S})$, $\mathcal{C}_K^{\mathrm{CCP}}(\mathcal{S})$ proposed in [42]. In this section, we propose a new technique of computing the cost of a given strategy called *precedence-constrained scheduling (PCS)*. The first part of the technique is to construct the *task dependency graph* of a strategy, and the second part is to evaluate a strategy by using its task dependency graph and precedence-constrained scheduling algorithms.

4.1.1 Task Dependency Graphs of Strategies

Without loss of generality, we assume that for a given strategy $\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S})$, all vertices in $\mathcal{V}_\mathcal{S}$ that are unreachable from $(0,0)$ are removed since they are not related to the cost computation. From Section 3.1 we recall that in any well-formed strategy there is a unique path from $(0,0)$ to any vertices in a strategy. This implies that every vertex in a well-formed strategy that can be reached from $(0,0)$, except for $(0,0)$, must have only one incoming edge. Thus, for a point (i,j) to be available, the operation representing the incoming edge to the point (i,j) must be completed. Therefore, in a strategy, a point and its incoming edge represent the same thing. This concept is important in constructing the task dependency graphs of a strategy.

A task dependency graph is defined as follows.

Definition 4.2 (Task dependency graph)**.** Given a set of tasks $T = \{t_1, ..., t_n\}$, the *task dependency graph* for T is a directed acyclic graph $\mathcal{D}_T = (\mathcal{V}_{\mathcal{D}_T}, \mathcal{E}_{\mathcal{D}_T})$ where $\mathcal{V}_{\mathcal{D}_T} = T$ and $\langle t_i, t_j \rangle \in \mathcal{E}_{\mathcal{D}_T}$ if a task t_i must be performed and finished before a task t_j can begin.

We then give a definition of the task dependency graph of a strategy below.

Definition 4.3 (Task dependency graph of a strategy)**.** The task dependency graph of a strategy $\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S} = \mathcal{E}_{\mathcal{S},\text{mul}} \cup \mathcal{E}_{\mathcal{S},\text{iso}})$ is a directed acyclic graph $\mathcal{D}_\mathcal{S} = (\mathcal{V}_{\mathcal{D}_\mathcal{S}}, \mathcal{E}_{\mathcal{D}_\mathcal{S}})$ where $\mathcal{V}_{\mathcal{D}_\mathcal{S}} = \mathcal{V}_\mathcal{S} \setminus \{(0,0)\}$ and

$$\begin{aligned}\mathcal{E}_{\mathcal{D}_\mathcal{S}} = (&\mathcal{E}_\mathcal{S} \cup \{\langle (i, e-1-i), (i+1, j)\rangle : \langle (i,j), (i+1,j)\rangle \in \mathcal{E}_{\mathcal{S},\text{iso}}\}) \\ &\setminus \{\langle (0,0),(0,1)\rangle, \langle (0,0),(1,0)\rangle\}.\end{aligned}$$

A vertex $(i,j) \in \mathcal{V}_{\mathcal{D}_\mathcal{S}}$ should be thought as a "task" of computing the point (i,j), but it can be thought as the point as well following our discussion earlier. For each isogeny evaluation edge $\langle (i,j), (i+1,j)\rangle$ in $\mathcal{S}$, we add an edge $\langle (i, e-i-1), (i+1,j)\rangle$ to $\mathcal{D}_\mathcal{S}$ to explicitly specify the dependency that we need to have R'_i before we can evaluate ϕ_i. We also remove $(0,0)$, since $(0,0)$ is available from the start and we do not have to perform any task to produce it. The next example depicts this process.

Example 4.4. Consider a strategy from Example 3.14 as shown in Figure 4.2(a). The first step of constructing the task dependency graph of it is to add a green diagonal directed edge for each red isogeny evaluation edge in order to show all implicit dependencies of isogeny evaluation described above. The result of the first step is in Figure 4.2(b). The second step is to remove the point $(0,0)$ and two edges from it. The task dependency graph $\mathcal{D}_\mathcal{S}$ is shown in Figure 4.2(c).

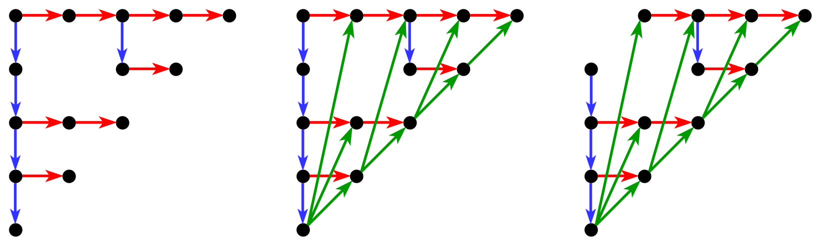


Figure 4.2: Constructing the task dependency graph of a strategy.

Task dependency graphs are useful for computing the costs of strategies as we will see later. In the next subsection, we require that task dependency graphs must not have any *transitive edge*. We give the definition of it and discuss how to remove them from $\mathcal{D}_{\mathcal{S}}$.

Definition 4.5 (Transitive edge)**.** For a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an edge $e = \langle u, v\rangle \in \mathcal{E}$ is *transitive* if there exists a vertex $w \notin \{u, v\}$ in $\mathcal{V}$ such that u reaches w and w reaches v.

Example 4.6. In the task dependency graph in Figure 4.2(c), the edge $\langle(0,2),(1,2)\rangle$ is transitive as $(0,2)$ reaches $(0,4)$ and $(0,4)$ reaches $(1,2)$. The edge $\langle(0,3),(1,3)\rangle$ is also transitive. These two edges are the only transitive edges in the graph.

Aho, Garey, and Ullman [2] presented that, for a general directed graph, the task of removing all transitive edges from a graph, called *transitive reduction*, can be done in $O(|\mathcal{V}|^{\log_2 7})$ steps. For $\mathcal{D}_{\mathcal{S}}$, it can be done in a more efficient way by the following lemma.

Lemma 4.7. All transitive edges in a graph $\mathcal{D}_{\mathcal{S}}$ must be of the form $\langle(i,j),(i+1,j)\rangle$. Also, the edge $\langle(i,j),(i+1,j)\rangle$ is transitive if and only if (i,j) reaches $(i, e-1-i)$.

Proof. For an edge $\langle u, v\rangle$ to be transitive in a directed acyclic graph, the out-degree of u and the in-degree of v must be more than 1. Therefore, all blue point multiplication edges of the form $\langle(i,j),(i,j+1)\rangle$ cannot be transitive.

Next, consider a green diagonal edge of the form $\langle(i, e-1-i),(i+1,j)\rangle$. If there exists another green diagonal edge coming out of $(i, e-1-i)$, its end point must be $(i+1, j')$ with $j' \neq j$. If $j' > j$, it is impossible that $(i+1, j')$ reaches $(i+1, j)$. If $j' < j$, $(i+1, j')$ can reach $(i+1, j)$ by going through a sequence of blue point multiplication edges. However, $(i+1, j)$ is the end point of the diagonal edge implies that it is the end point of the red isogeny evaluation edge $\langle(i,j),(i+1,j)\rangle$. Thus, there is no blue point multiplication edges coming to $(i+1,j)$. By both cases, all green diagonal edges cannot be transitive.

By Definition 4.3, for a red isogeny evaluation edge of the form $\langle (i,j),(i+1,j)\rangle$, there must exist the green diagonal edge $\langle (i,e-1-i),(i+1,j)\rangle$. These are only incoming edges to $(i+1,j)$. Therefore, if this red isogeny evaluation edge is transitive, (i,j) must reach $(i,e-1-i)$. This concludes the proof. $\square$

In order to remove all transitive edges from $\mathcal{D}_\mathcal{S}$, Lemma 4.7 suggests that we only need to go through all red isogeny evaluation edges once and remove $\langle (i,j),(i+1,j)$ if (i,j) reaches $(i,e-1-i)$. Verifying that there is a path from (i,j) to $(i,e-1-i)$ can be simply done by checking if all edges $\langle (i,j),(i,j+1)\rangle, \langle (i,j+1),(i,j+2)\rangle, \dots, \langle (i,e-2-i),(i,e-1-i)\rangle$ exist, since both points are in the same column. When implemented as in Algorithm 4.1, the transitive reduction of $\mathcal{D}_\mathcal{S}$ can be performed in $O(|\mathcal{V}_\mathcal{S}|)$ steps since each vertex (i,j) is visited at most once.

Algorithm 4.1: Transitive reduction algorithm for $\mathcal{D}_\mathcal{S}$.

```
Input  : The task dependency graph $\mathcal{D}_\mathcal{S} = (\mathcal{V}_{\mathcal{D}_\mathcal{S}}, \mathcal{E}_{\mathcal{D}_\mathcal{S}})$ of a strategy $\mathcal{S}$
Output: $\mathcal{D}_\mathcal{S}$ with all transitive edges removed
1 for $i = 0$ to $e-2$ do
2     for $j = e-i-2$ down to $0$ do
3         if $\langle (i,j),(i,j+1)\rangle \notin \mathcal{E}_{\mathcal{D}_\mathcal{S}}$ then break
4         if $\langle (i,j),(i+1,j)\rangle \in \mathcal{E}_{\mathcal{D}_\mathcal{S}}$ then $\mathcal{E}_{\mathcal{D}_\mathcal{S}} \leftarrow \mathcal{E}_{\mathcal{D}_\mathcal{S}} \setminus \{\langle (i,j),(i+1,j)\rangle\}$
5 return $\mathcal{D}_\mathcal{S}$
```

4.1.2 Task Scheduling Algorithms

The problem of scheduling a set of tasks to cores has been studied for a long time and has many applications in various fields such as operating systems and networks. For a given set of tasks, we need to specify which core performs which task and the goal is to minimize the time that the last task is finished. In our setting, we are interested in the problem of task scheduling with dependency: given a set of tasks with its task dependency graph, schedule all tasks to the cores available so that all tasks are done as soon as possible. There are several variants of this problem as itemized below.

- The structure of the task dependency graph: the graph has to be a DAG, but can be restricted to have some structure. For example, in a DAG with *tree-like* structure [41], all vertices can have at most one out-going edge.

- The time required for each task: all tasks can have unit or different length.
- The number of cores: we can have only one core or more than one. In addition, the number of cores can be fixed throughout the scheduling or can vary with time.
- The specification of cores: all cores can be identical, or their performances can be different, e.g., some cores can perform tasks faster than others.
- Preemption: if we are allowed to stop a task before it is finished, perform another task, and then continue the task that was paused (possibly with another core), the scheduling is called *preemptive*. Otherwise, it is *non-preemptive*.

In this thesis, we restrict ourselves to the case of the graphs $\mathcal{D}_{\mathcal{S}}$ with all tasks of unit-length, the number of cores is constant, all cores are identical, and preemption is not allowed. We formally define the problem of task scheduling as follows.

Definition 4.8 (Precedence-constrained scheduling problem)**.** Let $\mathcal{D}_T = (\mathcal{V}_{\mathcal{D}_T}, \mathcal{E}_{\mathcal{D}_T})$ be a task dependency graph, and let K be a positive integer. Suppose that all tasks require one unit of time to complete. A *scheduling of D_T using K cores* is a sequence $\mathcal{S} = \langle S_1, \ldots, S_n \rangle$ of non-empty sets of tasks where S_i is a set of tasks executed at time i such that (i) $S_1, \ldots, S_n$ form a partition of $\mathcal{V}_{\mathcal{D}_T}$, (ii) $\# S_i \leq K$, and (iii) for all $\langle t, t' \rangle \in \mathcal{E}_{\mathcal{D}_T}$, if $t \in S_i$ and $t' \in S_j$ then $i < j$. The finished time of $\mathcal{S}$ is n, the size of $\mathcal{S}$, and is denoted by $\mathcal{T}(\mathcal{S})$.

A scheduling $\mathcal{S}$ is *optimal* if $\mathcal{T}(\mathcal{S}) \leq \mathcal{T}(\mathcal{S}')$ for all possible schedulings $\mathcal{S}'$ of $\mathcal{D}_T$ using K cores. The *(precedence-constrained) scheduling problem* is to find an optimal scheduling for given $\mathcal{D}_T$ and K.

Example 4.9. Consider the task dependency graph in Figure 4.3 with $K = 3$. One possible scheduling for it is $\mathcal{S} = \langle \{t_1, t_4\}, \{t_2, t_3, t_5\}, \{t_6, t_7, t_8\}, \{t_9\} \rangle$ with $\mathcal{T}(\mathcal{S}) = 4$. Obviously, it is not optimal as there exists another scheduling $\mathcal{S}' = \langle \{t_1, t_3, t_5\}, \{t_2, t_4, t_7\}, \{t_6, t_8, t_9\} \rangle$ with $\mathcal{T}(\mathcal{S}') = 3$. Because the lower bound of the finish time is $\lceil \frac{\# \mathcal{V}_{\mathcal{D}_T}}{K} \rceil = \lceil \frac{9}{3} \rceil = 3$, we are certain that $\mathcal{S}'$ is an optimal scheduling. Other optimal schedulings also exist.

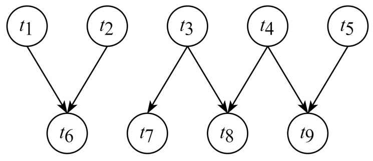


Figure 4.3: A task dependency graph.

Even in this setting, the problem might still be "hard". For general DAGs, Ullman [92] proved that the problem is NP-complete, and Garey and Johnson [35] mentioned that complexity remains open when the number of cores $K \geq 3$ is fixed.

In the rest of this subsection, we take a look at two algorithms. The first algorithm by Hu [41] outputs a scheduling with optimal finished time for $K \geq 1$ cores when the task dependency graph is tree-like. The second algorithm by Coffman and Graham [47] produced a scheduling with optimal finished time when $K = 2$. When $K \geq 3$, no efficient algorithm has been proposed. Nonetheless, there are many approximation algorithms solving this problem with various approximation ratios. Below gives the definition of the approximation ratios of an algorithm.

Definition 4.10 (Approximation ratio)**.** An algorithm $\mathcal{A}$ solving the task scheduling problem has the *approximation ratio* of r if for all instances $(\mathcal{D}_T, K)$ of the problem, the ratio between $\mathcal{T}(\mathcal{S})$ produced by $\mathcal{A}$ and $\mathcal{T}(\mathcal{S}^*)$, where $\mathcal{S}^*$ is an optimal scheduling, is at most r,

$$\frac{\mathcal{T}(\mathcal{S})}{\mathcal{T}(\mathcal{S}^*)} \leq r.$$

Equivalently, we say that $\mathcal{A}$ is an r*-approximation algorithm* of the task scheduling problem. An algorithm which always gives an optimal scheduling has the approximation ratio of 1.

Hu's Algorithm

The first algorithm deals with a task dependency graph which is tree-like, i.e., all vertices has out-degrees of at most one. Hu described that this structure mimics an assembly line, where tasks are making an elementary part or putting several parts together to create a more complex part.

For $u \in \mathcal{V}_{\mathcal{D}_T}$, let $\mathcal{L}(u)$ denote the length of a longest path started at u. In a tree-like graph, the longest path started from each vertex is unique since all vertices have at most one out-going edge. For $u \in \mathcal{V}_{\mathcal{D}_T}$, $\mathcal{L}(u)$ can be computed by walking through the only out-going edge until we are at a vertex with no out-going edge. To find $\mathcal{L}(u)$ for all $u \in \mathcal{V}_{\mathcal{D}_T}$, we can apply a breadth-first search or a depth-first search algorithm to the graph $\operatorname{rev}(\mathcal{D}_T) = (\mathcal{V}_{\mathcal{D}_T}, \operatorname{rev}(\mathcal{E}_{\mathcal{D}_T}))$ where $\operatorname{rev}(\mathcal{E}_{\mathcal{D}_T}) = \{\langle v, u\rangle : \langle u, v\rangle \in \mathcal{E}_{\mathcal{D}_T}\}$ is the set of edges in $\mathcal{E}_{\mathcal{D}_T}$ with their directions reversed.

Hu's algorithm is described in Algorithm 4.2. In short, the algorithm chooses up to K available tasks with largest $\mathcal{L}(\cdot)$ in each iteration until all tasks are performed. The chosen tasks and their edges are then removed from the graph in order to show new available tasks.

Algorithm 4.2: Hu's algorithm.

```
Input  : A tree-like task dependency graph $\mathcal{D}_T = (\mathcal{V}_{\mathcal{D}_T}, \mathcal{E}_{\mathcal{D}_T})$ and the number of
         provided cores $K$
Output: An optimal scheduling $\mathcal{S} = \langle S_1, \ldots, S_t \rangle$
1  Compute $\mathcal{L}(u)$ for all $u \in \mathcal{V}_{\mathcal{D}_T}$
2  $t \leftarrow 0$
3  while $\mathcal{V}_{\mathcal{D}_T} \neq \emptyset$ do
4      $t \leftarrow t + 1$
5      $\mathcal{V}' \leftarrow \{u \in \mathcal{V}_{\mathcal{D}_T} : \text{in-degree of } u = 0\}$
6      Sort $\mathcal{V}'$ by $\mathcal{L}(u)$ in an decreasing order, break ties arbitrarily
7      if $\#\mathcal{V}' \leq K$ then $S_t \leftarrow \mathcal{V}'$
8      else $S_t \leftarrow \{$the first $K$ vertices in $\mathcal{V}'\}$
9      Remove all vertices in $S_t$ and their associated edges from $\mathcal{D}_T$
10 return $\mathcal{S} = \langle S_1, \ldots, S_t \rangle$
```

As mentioned earlier, Hu's algorithm was proved to output an optimal scheduling for input graphs that are tree-like. In addition, [41] also answered another question: what is the smallest number of cores K needed in order to finish all tasks within time T. This question is also interesting but we do not pursue it.

We provide two examples below. The first performs the algorithm with a tree-like graph and the second with a non-tree-like graph. When a task dependency graph is not tree-like, Hu's algorithm might not produce an optimal scheduling. We note that $\mathcal{L}(u)$ for a non-tree-like graph can be computed in the reversed topological order.

Example 4.11. We apply Hu's algorithm to two graphs in Figure 4.4 with $K = 3$. The values $\mathcal{L}(u)$ are written in red above each vertex.

We start with graph (a) which is tree-like. $\mathcal{V}'$ in the first iteration is sorted as $\mathcal{V}' = [t_1, t_3, t_4, t_6, t_8, t_{12}, t_9]$, hence $S_1 = \{t_1, t_3, t_4\}$ and three tasks in S_1 are then removed. The task t_2 are now available as its incoming edge from t_1 is removed. In the second iteration, we have $\mathcal{V}' = [t_2, t_8, t_6, t_{12}, t_9]$ and $S_2 = \{t_2, t_8, t_6\}$. Removing t_2, t_8, t_6 enables t_5 and t_{11} to be included in $\mathcal{V}'$. Continuing in this fashion, one possible output of the algorithm is $\mathcal{S} = \langle \{t_1, t_3, t_4\}, \{t_2, t_8, t_6\}, \{t_5, t_{11}, t_{12}\}, \{t_7, t_9, t_{15}\}, \{t_{10}, t_{13}\}, \{t_{14}\} \rangle$ which is optimal.

Next, consider graph (b) which is not tree-like. In the first round, we can have $\mathcal{V}' = [t_4, t_1, t_3, t_2]$ as ties are broken arbitrarily. If $S_1 = \{t_4, t_1, t_3\}$, the output scheduling cannot be optimal as S_2 must be $\{t_2, t_8\}$ and $\mathcal{T}(\mathcal{S})$ will be at least 5. The optimal scheduling $\mathcal{S}^* = \langle \{t_1, t_2, t_3\}, \{t_4, t_5, t_6\}, \{t_7, t_8, t_9\}, \{t_{10}, t_{11}, t_{12}\} \rangle$ has $\mathcal{T}(\mathcal{S}^*) = 4$.

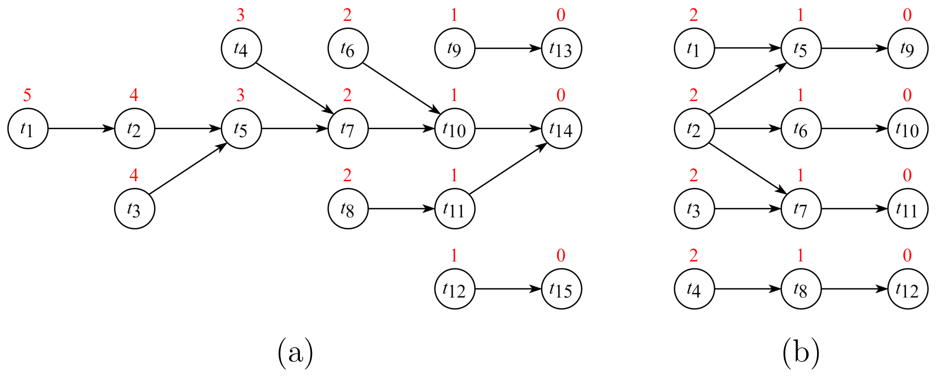


Figure 4.4: Task dependency graph that is tree-like and non-tree-like.

Although Hu's algorithm may not give an optimal scheduling when a task dependency graph is not tree-like, which is the case of our graphs $\mathcal{D}_{\mathcal{S}}$, we have tried applying Hu's algorithm to find the cost of task dependency graphs of some strategies. The experimental results show some interesting outcomes. We explain them in more detail in Section 4.3.

Coffman-Graham's Algorithm

The second algorithm refines how vertices are labeled. In Hu's algorithm, vertices are labeled by the lengths of their longest paths. In [47], Coffman and Graham presented another way to label vertices for DAGs of any structure without transitive edges. After all vertices are labeled, the same technique as in Hu's algorithm is then applied: choose up to K available tasks with largest labels to be performed at each time. By Coffman-Graham's labeling algorithm, some guarantees on the output scheduling can be proved.

The labeling process of Coffman and Graham is described in Algorithm 4.3. We give an example of the function $\mathcal{C}(\cdot)$ in Lines 7–8 as follows: Suppose u has three children v_1, v_2, v_3 and all are labeled with $\mathcal{L}_{\mathrm{CG}}(v_1) = 4$, $\mathcal{L}_{\mathrm{CG}}(v_2) = 3$, and $\mathcal{L}_{\mathrm{CG}}(v_1) = 8$. Then, $\mathcal{C}(u)$ is the list $[8, 4, 3]$ as it is sorted in decreasing order. In Line 8, lists are compared lexicographically, e.g., $[4, 2, 1] < [4, 3]$, $[5, 4, 2] < [5, 4, 2, 1]$, and $[\,] < [3, 2]$.

At first, one vertex with no out-going edge is assigned a label of 1. In each iteration, one vertex is labeled. $\mathcal{V}''$ in Line 5 is the set of unlabeled vertices with all children labeled. By the definition of $\mathcal{V}''$, $\mathcal{C}(\cdot)$ is well-defined for all vertices in $\mathcal{V}''$. The next vertex to be assigned a label is $u \in \mathcal{V}''$ with smallest $\mathcal{C}(u)$. The label is assigned from 1 up to $\#\mathcal{V}_{\mathcal{D}_T}$.

Algorithm 4.3: Coffman-Graham's labeling algorithm.

```
Input  : A task dependency graph $\mathcal{D}_T = (\mathcal{V}_{\mathcal{D}_T}, \mathcal{E}_{\mathcal{D}_T})$
Output: Coffman-Graham's label $\mathcal{L}_{\text{CG}}(u)$ for all $u \in \mathcal{V}_{\mathcal{D}_T}$
1 Choose any vertex $u$ with out-degree of 0 and assign $\mathcal{L}_{\text{CG}}(u) \leftarrow 1$
2 idx $\leftarrow 1$
3 while there is a vertex without a label do
4     idx $\leftarrow$ idx $+ 1$
5     $\mathcal{V}'' \leftarrow \{u \in \mathcal{V}_{\mathcal{D}_T} : u$ is not labeled and all its children are labeled$\}$
6     for $u \in \mathcal{V}''$ do $\mathcal{C}(u) \leftarrow$ the list of all labels of $u$'s children in decreasing order
7     Choose $u \in \mathcal{V}''$ with lexicographically smallest $\mathcal{C}(u)$, break ties arbitrarily
8     $\mathcal{L}_{\text{CG}}(u) \leftarrow$ idx
```

Coffman and Graham proved that, by using $\mathcal{L}_{\text{CG}}(u)$ instead of $\mathcal{L}(u)$ in Algorithm 4.2, the output scheduling is optimal when $K = 2$ for a task dependency graph of any structure. A few years later, Lam and Sethi [61] showed that the algorithm is $(2 - \frac{2}{K})$-approximation for $K \geq 2$. When K is small, the approximation ratio is close to 1.

Example 4.12. We perform Coffman-Graham's algorithm on the graph in Figure 4.4(b). Suppose in Line 1, we choose $\mathcal{L}_{\text{CG}}(t_9) = 1$. In the first iteration, $\mathcal{V}'' = \{t_5, t_{10}, t_{11}, t_{12}\}$ with $\mathcal{C}(t_{10}) = \mathcal{C}(t_{11}) = \mathcal{C}(t_{12}) = [\,]$ and $\mathcal{C}(t_5) = [1]$. By the algorithm, we can choose any one in $\{t_{10}, t_{11}, t_{12}\}$ and assign the label 2. Let us pick $\mathcal{L}_{\text{CG}}(t_{11}) = 2$. We keep track of $\mathcal{L}_{\text{CG}}(u)$ and $\mathcal{C}(u)$ as in Figure 4.5. In the second iteration, only t_9 and t_{11} are labeled, thus $\mathcal{V}'' = \{t_5, t_{10}, t_7, t_{12}\}$. Breaking a tie arbitrarily, we select $\mathcal{L}_{\text{CG}}(t_{12}) = 3$. The next vertex to be assigned the label is t_{10} with label 4.

In the fourth iteration, $\mathcal{V}'' = \{t_5, t_6, t_7, t_8\}$ with $\mathcal{C}(t_5) = [1]$, $\mathcal{C}(t_6) = [4]$, $\mathcal{C}(t_7) = [2]$, and $\mathcal{C}(t_8) = [3]$. Therefore, t_5 receives the label 5. The process continues until all vertices are labeled. To obtain a scheduling, we choose available vertices with highest labels for up to K vertices. Since all labels are unique, there is only one way to construct a scheduling. In our example with $K = 3$, $\mathcal{S} = \langle\{t_2, t_4, t_3\}, \{t_1, t_6, t_8\}, \{t_7, t_5, t_{10}\}, \{t_{12}, t_{11}, t_9\}\rangle$ which is optimal. We note that Coffman-Graham's algorithm does not consider K for labeling.

Example 4.13. This example shows that Coffman-Graham's algorithm might not output an optimal scheduling when $K \geq 3$. Consider the graph in Figure 4.6 from [61] with $K = 3$. If $\mathcal{L}_{\text{CG}}(u)$ are as shown in the figure, we have $\mathcal{S} = \langle\{t_5, t_4, t_3\}, \{t_2, t_1\}, \{t_9, t_8\}, \{t_7, t_6\}\rangle$. However, an optimal scheduling is $\mathcal{S}^* = \langle\{t_1, t_2, t_3\}, \{t_4, t_5, t_6\}, \{t_7, t_8, t_9\}\rangle$. Nonetheless, $\frac{\mathcal{T}(\mathcal{S})}{\mathcal{T}(\mathcal{S}^*)} = \frac{4}{3}$ is no more than $2 - \frac{2}{K} = \frac{4}{3}$. It was proved in [61] that it is always possible to construct a graph such that $\frac{\mathcal{T}(\mathcal{S})}{\mathcal{T}(\mathcal{S}^*)}$ approaches $2 - \frac{2}{K}$.

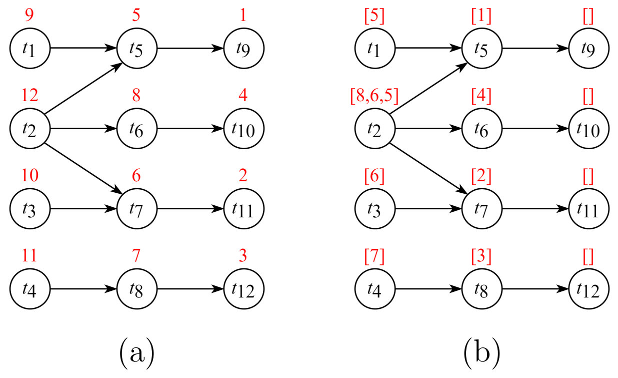


Figure 4.5: The values of $\mathcal{L}_{\text{CG}}(u)$ and $\mathcal{C}(u)$ when Coffman-Graham's algorithm terminates.

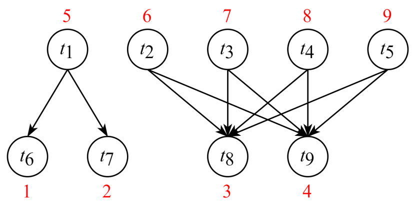


Figure 4.6: A graph not giving an optimal scheduling under Coffman-Graham's algorithm.

We note that approximation algorithms have recently been proposed with smaller ratios, e.g., [34] gave $(2 - \frac{7}{3K+1})$-approximation algorithm and [62] gave a $(1+\epsilon)$-approximation algorithm. We refer the interested readers to the mentioned papers for more details.

We end this subsection by giving a short proof regarding the graph $\text{rev}(\mathcal{D}_T)$ and $\text{rev}(\mathcal{S}) = \langle S_n, \ldots, S_1 \rangle$, where $\mathcal{S} = \langle S_1, \ldots, S_n \rangle$.

Lemma 4.14. If $\mathcal{S}$ is a scheduling for an instance $(\mathcal{D}_T, K)$, then $\text{rev}(\mathcal{S})$ is a scheduling for the instance $(\text{rev}(\mathcal{D}_T), K)$.

Proof. Let $\text{rev}(\mathcal{S}) = \langle S'_1, \ldots, S'_n \rangle = \langle S_n, \ldots, S_1 \rangle$. It is clear that $S'_1, \ldots, S'_n$ form a partition of $\mathcal{V}_{\mathcal{D}_T}$ and $\#S'_i \leq K$. For $\langle t, t' \rangle \in \text{rev}(\mathcal{E}_{\mathcal{D}_T})$, there is $\langle t', t \rangle \in \mathcal{E}_{\mathcal{D}_T}$. Let $t \in S'_i = S_{n+1-i}$ and $t' \in S'_j = S_{n+1-j}$. Because $\mathcal{S}$ is a scheduling, we have $n+1-j < n+1-i$. Therefore, $i < j$ and $\text{rev}(\mathcal{S})$ is a scheduling for $(\text{rev}(\mathcal{D}_T), K)$ as desired. □

Since $\text{rev}(\text{rev}(\mathcal{D}_T)) = \mathcal{D}_T$ and $\text{rev}(\text{rev}(\mathcal{S})) = \mathcal{S}$, Lemma 4.14 suggests that, in order to construct an optimal strategy for $(\mathcal{D}_T, K)$, we can find an optimal strategy for $(\text{rev}(\mathcal{D}_T), K)$ and then reverse it.

4.1.3 Strategy Evaluation with PCS

After we construct the task dependency graph from a strategy and remove all transitive edges, two precedence-constrained scheduling algorithms—Hu's and Coffman-Graham's algorithms—previously described can be applied to obtain a scheduling. Although both algorithms assume that all tasks are of unit-length when scheduling, which is not the case for our setting since $c_{\text{mul}} \neq c_{\text{iso}}$, they can be used as approximation algorithms.

Because both scheduling algorithms are designed for unit-length tasks, we calculate the cost of a strategy evaluation from a scheduling as shown in Algorithm 4.4: for each $1 \leq i \leq \mathcal{T}(\mathcal{S})$, if all tasks in S_i are point multiplications, the cost of S_i is c_{mul}. If all tasks in S_i are isogeny evaluations, its cost is c_{iso}. Otherwise, its cost is $\max\{c_{\text{mul}}, c_{\text{iso}}\}$. The costs of a strategy $\mathcal{S}$ when using Hu's and Coffman-Graham's algorithms with K cores are denoted by $\mathcal{C}_K^{\text{Hu}}(\mathcal{S})$ and $\mathcal{C}_K^{\text{CG}}(\mathcal{S})$, respectively.

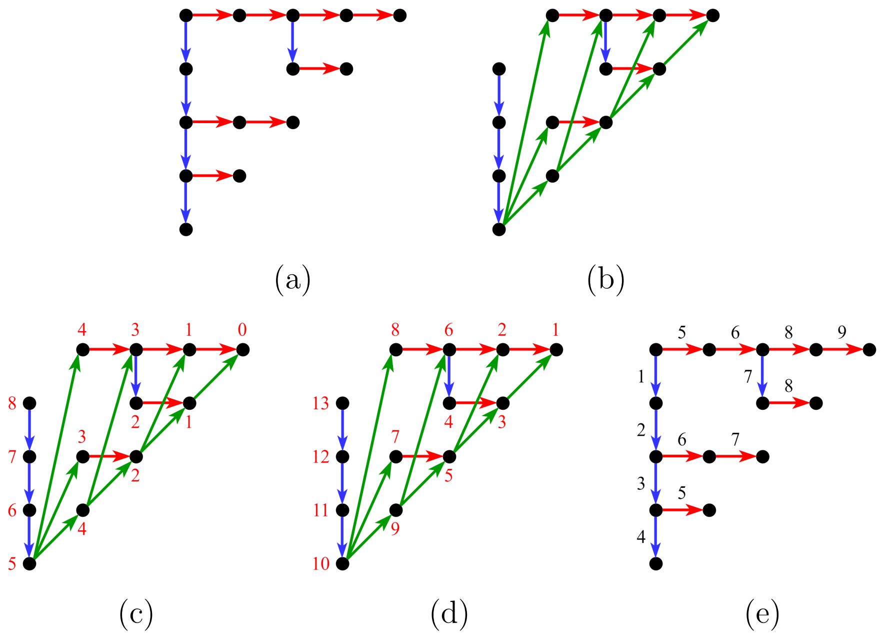


Figure 4.7: Precedence-constrained scheduling (PCS) technique.

Algorithm 4.4: Precedence-Constrained Scheduling (PCS) Technique.

```
Input : A strategy $\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S})$ for computing degree-$\ell^e$ isogeny and the number
        of provided cores $K$
Output: The cost $\mathfrak{C}_K^{\mathrm{Hu}}(\mathcal{S})$ or $\mathfrak{C}_K^{\mathrm{CG}}(\mathcal{S})$ and a scheduling $\mathbb{S}$
1  $\mathcal{E}_\mathcal{S}^* \leftarrow \mathcal{E}_\mathcal{S}$
2  for $\langle (i,j),(i+1,j)\rangle \in \mathcal{S}$ do
3      $\mathcal{E}_\mathcal{S} \leftarrow \mathcal{E}_\mathcal{S} \cup \{\langle (i, e-i-1),(i+1,j)\rangle\}$
4  $\mathcal{E}_\mathcal{S} \leftarrow \mathcal{E}_\mathcal{S} \setminus \{\langle (0,0),(0,1)\rangle, \langle (0,0),(1,0)\rangle\}$
5  $\mathcal{V}_\mathcal{S} \leftarrow \mathcal{V}_\mathcal{S} \setminus \{(0,0)\}$
6  Remove transitive edges from $\mathcal{E}_\mathcal{S}$ (Algorithm 4.1)
7  Label all vertices $v \in \mathcal{S}$ with $\mathcal{L}(v)$ or $\mathcal{L}_{\mathrm{CG}}(v)$ (Algorithm 4.3)
8  $\mathbb{S} \leftarrow \langle\rangle$
9  $t \leftarrow 0$
10 cost $\leftarrow 0$
11 while $\mathcal{V}_\mathcal{S} \neq \emptyset$ do
12     $t \leftarrow t+1$
13     $S_t \leftarrow \{K$ vertices in $\mathcal{S}$ with highest $\mathcal{L}(\cdot)$ and their in-degrees are $0\}$
14     Append $S_t$ to $\mathbb{S}$
15     Remove all vertices in $S_t$ and their out-going edges from $\mathcal{S}$
16     $\mathrm{cost}_t \leftarrow 0$
17     for $(i,j) \in S_t$ do
18         if $\langle (i,j-1),(i,j)\rangle \in \mathcal{E}_\mathcal{S}^*$ then $\mathrm{cost}_t \leftarrow \max\{\mathrm{cost}_t, c_{\mathrm{mul}}\}$
19         else $\mathrm{cost}_t \leftarrow \max\{\mathrm{cost}_t, c_{\mathrm{iso}}\}$
20     cost $\leftarrow$ cost $+ \mathrm{cost}_t$
21 return $(\mathrm{cost}, \mathbb{S})$
```

Example 4.15. We explain how $\mathfrak{C}_K^{\mathrm{Hu}}(\mathcal{S})$ and $\mathfrak{C}_K^{\mathrm{CG}}(\mathcal{S})$ are computed for the strategy shown in Figure 4.7(a). First, its task dependency graph with all transitive edges removed is shown in Figure 4.7(b). Next, all vertices are labeled. The values of $\mathcal{L}(v)$ and $\mathcal{L}_{\mathrm{CG}}(v)$ are provided in Figures 4.7(c) and 4.7(d), respectively. For $K = 2$, Hu's and Coffman-Graham's algorithms give $\mathbb{S} = \langle S_1 = \{(0,1)\}, S_2 = \{(0,2)\}, S_3 = \{(0,3)\}, S_4 = \{(0,4)\}, S_5 = \{(1,3),(1,0)\}, S_6 = \{(1,2),(2,0)\}, S_7 = \{(2,2),(2,1)\}, S_8 = \{(3,1),(3,0)\}, S_9 = \{(4,0)\}\rangle$. In S_5, $(1,3)$ and $(1,0)$ are computed by isogeny evaluations, thus $\mathrm{cost}_5 = c_{\mathrm{iso}}$. In S_7, $(2,2)$ is computed by isogeny evaluation and $(2,1)$ is computed by point multiplication, hence $\mathrm{cost}_7 = \max\{c_{\mathrm{mul}}, c_{\mathrm{iso}}\}$. The costs $\mathfrak{C}_K^{\mathrm{Hu}}(\mathcal{S})$ and $\mathfrak{C}_K^{\mathrm{CG}}(\mathcal{S})$ are therefore $4c_{\mathrm{mul}} + 4c_{\mathrm{iso}} + \max\{c_{\mathrm{mul}}, c_{\mathrm{iso}}\}$. The evaluation when $c_{\mathrm{mul}} = c_{\mathrm{iso}} = 1$ is shown in Figure 4.7(e).

4.2 Strategy Construction with Linear Programming

In addition to an evaluation technique that gives us a low cost from a strategy, we also need efficient strategies that would provide low costs. As discussed earlier, a strategy for the multi-core setting has to be carefully constructed specifically for the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}, K)$. To construct those low-cost strategies, we first formalize the problem mathematically as an integer linear program (ILP) and then use optimal solutions of the ILP to generate strategies.

4.2.1 Optimal Strategies and Evaluations

The problem of constructing a strategy and its evaluation is clearly an optimization problem. We call a pair of a strategy and its evaluation that provides the least cost as *optimal.* In this subsection, we will construct an optimal strategy and evaluation in the simplest case of $c_{\text{mul}} = c_{\text{iso}} = 1$, which can be generalized to the case that $c_{\text{mul}} = c_{\text{iso}}$.

Let $x_{i,j,t} \in \{0,1\}$ be a decision variable such that $x_{i,j,t} = 1$ if the point represented by the vertex (i,j) is computed and is available no later than time t and 0 otherwise. A discrete optimization problem of finding an optimal strategy and its evaluation can be formalized as an integer linear program (ILP) as follows:

$$
\begin{array}{lrll}
\underset{x_{i,j,t}}{\text{minimize}} & & T + 1 - \sum_{t'=0}^{T} x_{e-1,0,t'} & \\
\text{subject to} & & x_{0,0,0} = 1 & \\
& & x_{i,j,0} = 0 & (i,j) \neq (0,0) \\
& & x_{i,j,t} \geq x_{i,j,t-1} & \\
& & x_{i,j,t} \leq x_{i,j-1,t-c_{\text{mul}}} + \dfrac{x_{i-1,j,t-c_{\text{iso}}} + x_{i-1,e-i,t-c_{\text{iso}}}}{2} & \\
& \sum_{i,j} (x_{i,j,t+1} - x_{i,j,t}) \leq K & & \\
& & x_{i,j,t} \in \{0,1\} &
\end{array}
$$

The initial conditions for $x_{i,j,0}$ are $x_{0,0,0} = 1$, since it is available at the start of the isogeny computation, and $x_{i,j,0} = 0$ for $(i,j) \neq (0,0)$. If (i,j) is available no later than time $t-1$, then it is also available no later than time t. Hence, we have the constraint $x_{i,j,t} \geq x_{i,j,t-1}$. Our objective is thus to minimize t' such that $x_{e-1,0,t'} = 1$, the time that $(e-1,0)$ is finished. However, we cannot straightforwardly use this as an objective function because t' is not a decision variable. We instead consider the sum of $x_{e-1,0,t'}$ for $0 \leq t' \leq T$ for some sufficiently large T. The earliest time t' at which $x_{e-1,0,t'}$ is ready can now be expressed by $T + 1 - \sum_{0 \leq t' \leq T} x_{e-1,0,t'}$, which is our objective function.

The fourth constraint comes from two restrictions of the isogeny computation discussed in Subsection 3.3.1: $x_{i,j,t}$ can become 1 by one of these two cases: (i) $(i, j-1)$ is ready at time $t - c_{\text{mul}}$ and (i,j) is computed by a point multiplication, or (ii) $(i-1,j)$ and $(i-1,e-i)$ are available at time $t - c_{\text{iso}}$ and (i,j) is computed by an isogeny evaluation. The first case is possible if $x_{i,j-1,t-c_{\text{mul}}} = 1$. For the second case, both $x_{i-1,j,t-c_{\text{iso}}}$ and $x_{i-1,e-i,t-c_{\text{iso}}}$ must be 1. Hence, we can perform the second case if $\frac{1}{2}(x_{i-1,j,t-c_{\text{iso}}} + x_{i-1,e-i,t-c_{\text{iso}}}) = 1$. Because $x_{i,j,t}$ can become 1 by either of the two cases, the value of $x_{i,j,t}$ is restricted to

$$x_{i,j,t} \leq x_{i,j-1,t-c_{\text{mul}}} + \frac{x_{i-1,j,t-c_{\text{iso}}} + x_{i-1,e-i,t-c_{\text{iso}}}}{2}.$$

The fifth constraint is by the number of cores given. Since we are interested in the case that $c_{\text{mul}} = c_{\text{iso}} = 1$, there can be up to K decision variables that change from 0 to 1 at each time, those represent points computed at that time. Therefore, we have

$$\sum_{i,j} (x_{i,j,t+1} - x_{i,j,t}) \leq K.$$

Given the integer linear program of the problem, we can use a solver to find an optimal strategy and its evaluation in a general setting. Table 4.1 below shows the least possible cost of any strategy in general settings and under PCP using $c_{\text{mul}} = c_{\text{iso}} = 1$. We use shading to express the differences between the cost under PCP and the optimal one.

The results shown in Table 4.1 support our claim that the costs under PCP and CCP are not optimal and we are able to improve them. The differences in the table can be up to 6, and they are expected to grow for larger e and K. However, even in the case of $c_{\text{mul}} = c_{\text{iso}} = 1$ and small $e < 15$, the solver can take several hours to produce a solution. This is expected since integer linear programming is NP-hard [48]. Although it is not practical to construct optimal strategies and evaluations for large e directly using ILP, we will use solutions for small e to construct a low-cost strategy for large e in the next subsection.

e		3	4	5	6	7	8	9	10	11	12	13	14
$K = 1$		5	8	12	16	20	24	29	34	39	44	49	54
$K = 2$	PCP	4	7	10	13	16	19	23	27	31	35	39	43
	Optimal	4	7	9	12	15	18	21	24	27	30	33	37
$K = 3$	PCP	4	6	9	12	15	18	21	24	27	30	33	37
	Optimal	4	6	9	11	14	16	19	21	24	27	29	32
$K = 4$	PCP	4	6	8	11	14	17	20	23	26	29	32	35
	Optimal	4	6	8	11	13	15	18	20	23	25	27	30
$K = 5$	PCP	4	6	8	10	13	16	19	22	25	28	31	34
	Optimal	4	6	8	10	13	15	17	20	22	24	27	29
$K = 6$	PCP	4	6	8	10	12	15	18	21	24	27	30	33
	Optimal	4	6	8	10	12	15	17	19	21	24	26	28
$K = 7$	PCP	4	6	8	10	12	14	17	20	23	26	29	32
	Optimal	4	6	8	10	12	14	17	19	21	23	26	28
$K = 8$	PCP	4	6	8	10	12	14	16	19	22	25	28	31
	Optimal	4	6	8	10	12	14	16	19	21	23	25	27

Table 4.1: The costs of optimal strategies in a general setting and under PCP.

4.2.2 Constructing Low-Cost Strategies with Solutions of ILP

In Subsection 3.3.1, we state Theorem 3.17 from [42] for computing the cost of a least-cost (canonical) strategy under PCP. The theorem implicitly describes how this least-cost canonical strategy is constructed: a strategy with e leaves is divided into two smaller strategies with n and $e-n$ leaves, and the construction performs recursively until the base case $e = 1$ is reached. With the ILP we obtain in the previous subsection, we propose a new way of constructing a strategy which is by precomputing optimal strategies and evaluations for some e and then using them as base cases. We need to slightly modify the ILP in order to find optimal strategies and evaluations corresponding to $\mathcal{C}_K^{\text{PCP}}(e,k)$, but the main idea is the same. Strategies constructed by our proposed technique can then be viewed as a mixture of a canonical part when e is larger than the base case and a possibly non-canonical part when e is one of the base cases.

Similar to PCS in the previous section, we assume that $c_{\text{mul}} = c_{\text{iso}}$ when we formulate the ILP, which is not the case for our setting. Also, we only solve the ILP for up to some value of e and combine them for large e. Hence, strategies resulted from our construction technique are considered as approximations of a least-cost strategy.

4.3 Experiments and Results

For each parameter set $(e, c_{\text{mul}}, c_{\text{iso}}, K)$, we conduct two experiments using our proposed PCS evaluation (Section 4.1) and ILP construction (Section 4.2) techniques as follows:

- Experiment A: We use Theorem 3.17 to construct least-cost canonical strategies under PCP. Since there are many such strategies, we randomly sampled 100,000 of them for evaluation. The cost of strategy $\mathcal{S}$ is then computed as $\min\{\mathcal{C}_K^{\text{Hu}}(\mathcal{S}), \mathcal{C}_K^{\text{CG}}(\mathcal{S})\}$.
- Experiment B: We randomly constructed 100,000 strategies using our proposed strategy construction technique, where we precomputed solutions for ILP for all $e \leq 14$. The cost of strategy $\mathcal{S}$ is also computed as $\min\{\mathcal{C}_K^{\text{Hu}}(\mathcal{S}), \mathcal{C}_K^{\text{CG}}(\mathcal{S})\}$.

We conduct experiments under two sets of parameters from [58], which are also used by [42], for the purpose of comparison. Table 4.2 compares costs obtained by [42] and our experiments under the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}) = (186, 25.8, 22.8)$. Rows 3 and 5 show the smallest $\min\{\mathcal{C}_K^{\text{Hu}}(\mathcal{S}), \mathcal{C}_K^{\text{CG}}(\mathcal{S})\}$ among all randomly sampled strategies in Experiments A and B, respectively. Table 4.3 reports the results under the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}) = (239, 27.8, 17)$. The cost reductions in both tables are compared to the costs under CCP.

The experimental results show the reductions of more than 10% in several cases, which is significant due to the fact that CCP has already improved the cost of PCP and the single-core setting. Our strategy construction technique (Experiment B) works very well when $c_{\text{mul}} \approx c_{\text{iso}}$ as seen in Table 4.2. We expect greater reductions when we precompute solutions of ILP for more values of e.

	K	2	3	4	5	6	7	8
PCP	Cost	25942.2	22521.6	20373.0	19197.0	17941.2	16978.8	16617.0
CCP	Cost	23890.2	20515.2	18252.6	17555.4	16482.0	16021.2	15294.6
Exp. A (PCS)	Cost	22203.0	18622.8	16337.4	15708.6	15091.2	14949.6	14063.4
	% reduction	7.06	9.22	**10.49**	10.52	8.44	**6.69**	**8.05**
Exp. B (PCS+ILP)	Cost	22081.2	18340.2	16400.4	15269.4	14973.6	14999.4	14184.0
	% reduction	**7.57**	**10.60**	10.15	**13.02**	**9.15**	6.38	7.26

Table 4.2: The cost of best strategies under PCP, CCP, and in our experiments under the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}) = (186, 25.8, 22.8)$. The cost $\mathcal{C}_1^*(e)$ for $K = 1$ is 34256.4.

	K	2	3	4	7	8
PCP	Cost	31886.0	27858.0	25328.8	21572.6	20851.2
CCP	Cost	29931.0	25835.0	23390.8	20399.6	19814.2
Exp. A	Cost	28265.0	23625.0	21282.8	19073.6	18641.2
(PCS)	% reduction	**5.57**	**8.55**	**9.01**	**6.50**	**5.92**
Exp. B	Cost	28574.6	23731.0	21337.8	19319.0	18900.4
(PCS+ILP)	% reduction	4.53	8.14	8.78	5.30	4.61

Table 4.3: The cost of best strategies under PCP, CCP, and in our experiments under the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}) = (239, 27.8, 17)$. The cost $\mathcal{C}_1^*(e)$ for $K = 1$ is 41653.8.

In addition, we point out that $\mathcal{C}_K^{\text{Hu}}(\mathcal{S})$ and $\mathcal{C}_K^{\text{CG}}(\mathcal{S})$ of the same strategy $\mathcal{S}$ are equal for all (canonical) strategies sampled in Experiment A, but these costs can be slightly different for some (possibly non-canonical) strategies sampled in Experiment B. When the costs are not equal, $\mathcal{C}_K^{\text{Hu}}(\mathcal{S})$ is smaller for some K and strategies, while $\mathcal{C}_K^{\text{CG}}(\mathcal{S})$ is smaller for some others. This shows that none of the algorithms provides the least cost for strategy evaluation.

Lastly, we present the experimental results for both parameter sets when K is larger than 8. For these values of K, we did not precompute solutions for ILP and thus present the results when applying only PCS (Experiment A). The results are provided in Table 4.4 and Figure 4.8 (including the data for small K).

Once again, PCS outperforms PCP, from 3.28% to 12.34%. The reductions are large when the number of cores K is around 64–128 for both parameter sets. When $K \geq e-1$, it can be seen that PCP and PCS give the same cost since the optimal strategy is the isogeny-based strategy and both PCP and PCS give the same evaluation. Moreover, because the least cost we can have happens when $K = e-1$, we can see that the costs converge to that least cost when K increases. The costs drop drastically for small K and then drop slightly starting at around $K = 16$. This may suggest that using 16 cores for the computation might be the best trade-off between the cost and the computation resources.

K	$(e, c_{\text{mul}}, c_{\text{iso}}) = (186, 25.8, 22.8)$			$(e, c_{\text{mul}}, c_{\text{iso}}) = (239, 27.8, 17)$		
	PCP	PCS	% reduction	PCP	PCS	% reduction
16	13642.8	12844.8	5.85	17678.4	16029.4	9.33
32	12842.4	12135.6	5.50	16043.6	15516.6	3.28
48	12381.6	11310.0	8.65	15426.0	14627.0	5.18
64	11920.8	10484.4	12.05	14808.4	13737.4	7.23
80	11460.0	10046.4	12.34	14190.8	12847.8	9.46
96	11020.2	9948.6	9.72	13573.2	12332.2	9.14
112	10655.4	9561.0	10.27	12955.6	11799.6	8.92
128	10290.5	9287.4	9.75	12532.4	11478.4	8.41
144	9925.8	9127.8	8.04	12260.4	11206.4	8.60
160	9561.0	9059.4	5.25	11988.4	11002.4	8.22
185	8991.0	8991.0	0.00	11563.4	10798.4	6.62
238	8991.0	8991.0	0.00	10662.4	10662.4	0.00

Table 4.4: The cost of best strategies under PCP and PCS in our experiments under two parameter sets when K is large.

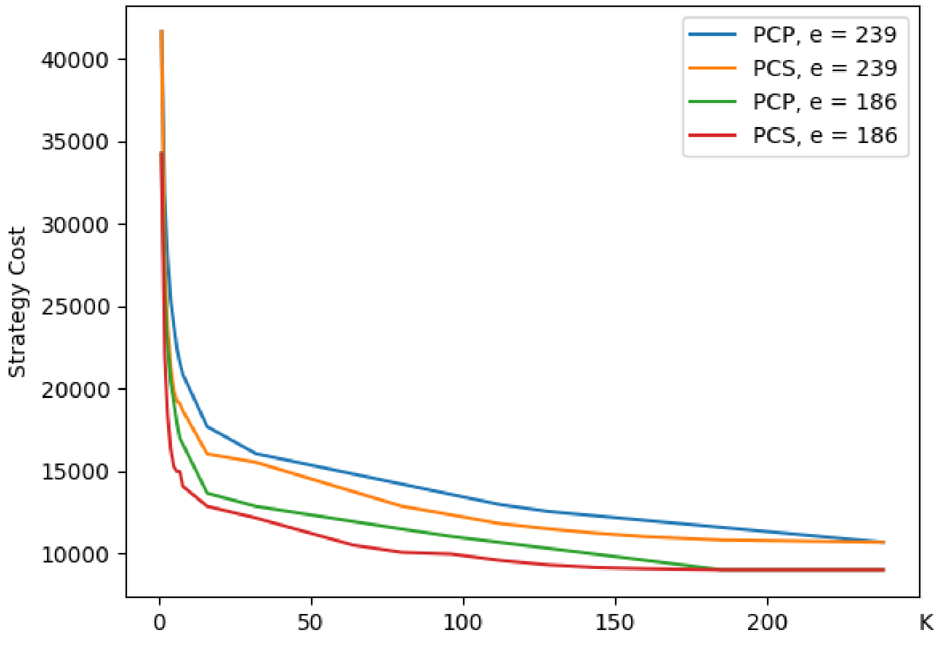


Figure 4.8: A plot for the costs under PCP and PCS in our experiments.

4.4 Chapter Summary

In this chapter, we studied the problem of constructing a strategy for computing degree-ℓ^e isogeny and evaluating it to achieve the least cost possible in the multi-core setting. The first half of this chapter proposed a novel evaluation technique called precedence-constrained scheduling (PCS): we first transformed a strategy into a task dependency graph and then applied task scheduling algorithms to it. We focused on the specific case that all tasks are unit-length. Even then, the problem is NP-hard. Here, we reviewed two task scheduling algorithms: Hu's and Coffman-Graham's algorithms. The first algorithm gives optimal scheduling when a graph is tree-like, but does not give optimal scheduling in general. The second algorithm gives optimal scheduling when $K = 2$ and is proved to be $(2 - \frac{2}{K})$-approximation. Finally, we applied both algorithms to the task dependency graphs constructed from strategies.

Next, we presented a strategy construction technique which utilizes solutions of integer linear programs for small e. We started by formulating the problem of finding an optimal strategy and its evaluation as an integer linear program. The solutions showed that the costs under PCP and CCP are not optimal. Although solving large integer linear program is not practical, we were able to solve small instances and then combined them to obtain low-cost strategies for the case where $c_{\text{mul}} \approx c_{\text{iso}}$.

Via experimental results, we were able to obtain costs that are lower than those under PCP and CCP [42], which already improve the cost of an optimal strategy under the single-core setting [30]. The improvements can be up to 13.02% under some specific parameter sets. For $K \geq 16$, we were also able to improve the cost with various extents.

Chapter 5

Parallel and Vectorized Implementations of PCS

This chapter presents two software implementations of the degree-ℓ^e isogeny computation which exploit the PCS technique and parallelism provided by modern processors. The first implementation is designed for two-to-four-core processors, while the second implementation is designed for two-to-four-core processors which support AVX-512. Since we did not consider any implementation environments in the previous chapter, we provide analyses and modifications on how to effectively apply the PCS technique to the unique execution environment of each implementation. The content of this chapter is based on [76], and both of the implementations are available at https://github.com/kittiphonp/PCS.

We begin by giving a brief introduction of the Intel's Advanced Vecter eXtension AVX-512 technology used in one of our implementations.

5.1 Intel's Advanced Vector eXtension AVX-512

The latest generation of Intel's Advanced Vector eXtensions (AVX), which is AVX-512, provides a way to vectorize and speed-up software by using vectors of length 512 bits and vectorized instructions. One extension of AVX-512 used by [20] and this work is the Integer Fused Multiply-Add extension (IFMA or AVX-512IFMA) which is useful for software libraries requiring large integer arithmetic. As we are mainly interested in the strategy-level optimization, we briefly explain the high-level usage of AVX-512.

In [20] and our implementation, we consider 512-bit vectors as eight elements of 64 bits: $[a, b, c, d, e, f, g, h]$ where each variable is of size 64 bits. The AVX-512 instructions allow us to compute $[a, b, \ldots, h] \oplus [a', b', \ldots, h'] = [a \oplus a', b \oplus b', \ldots, h \oplus h']$ within a similar time as $a \oplus a'$ for certain operations $\oplus$. Thus, we can consider AVX-512 as a form of parallel computation where all cores perform the same operation.

When the operand sizes are larger than 64 bits, they must be divided into 64-bit blocks in order to use AVX-512 instructions: $a = a_{n-1}a_{n-2} \ldots a_1 a_0$ where a_i is of size 64 bits. For a computation, we can use n vectors $V_i = [a_i, b_i, \ldots, h_i]$ for $0 \leq i < n$ to represent eight operands and perform $V_i \oplus V_i'$. One needs to take care of any carry and dependency between blocks to ensure correctness. Nonetheless, there are other usages when we have fewer than eight operands. Other possible options are (i) using $n/2$ vectors $W_i = [a_i, a_{i+n/2}, b_i, b_{i+n/2}, c_i, c_{i+n/2}, d_i, d_{i+n/2}]$ for $0 \leq i < n/2$ to represent four operands, (ii) using $n/4$ vectors $X_i = [a_i, a_{i+n/4}, a_{i+2n/4}, a_{i+3n/4}, b_i, b_{i+n/4}, b_{i+2n/4}, b_{i+3n/4}]$ for $0 \leq i < n/4$ to represent two operands, and (iii) using $n/8$ vectors $Y_i = [a_i, a_{i+n/8}, a_{i+2n/8}, a_{i+3n/8}, a_{i+4n/8}, a_{i+5n/8}, a_{i+6n/8}, a_{i+7n/8}]$ for $0 \leq i < n/8$ to represent only one operand. Following [20], we call these representations and computations as 8-way, 4-way, 2-way, and 1-way, respectively.

5.2 Multi-Core PCS Implementation

Our first software implementation of isogeny computation is designed for multi-core processors without AVX-512IFMA instructions. For this setting, we utilize the equation $\mathcal{C}_K^{\mathrm{PCP}^*}$ (Section 3.3.2) to construct strategies. To achieve better speed-up, we consider the optimization of [16] and modify PCS to accommodate such optimization.

5.2.1 Modifying PCS for Speed-Up

We first describe the speed-up technique of Cervantes-Vázquez et al. [16] for multi-core platforms. Some isogeny-based protocols require computing a point R from other points, e.g., $R \leftarrow P + [m]Q$ for $0 \leq m < \ell^e$, before passing it as an input for isogeny computation. For fixed points P and Q of degree ℓ^e, this computation, typically performed using a three-point ladder algorithm (Algorithm 2.3), takes time roughly $e \cdot c_{\mathrm{mul}}$ when m has bitlength e. However, $[\ell^j]R = [\ell^j]P + [m][\ell^j]Q$ can be computed faster. This is because (i) $[\ell^j]P$ and $[\ell^j]Q$ can be precomputed when P and Q are public and fixed, and (ii) $[m][\ell^j]Q = [m \bmod \ell^{e-j}][\ell^j]Q$ as $\deg([\ell^j]Q) = \ell^{e-j}$. The computation of $[m \bmod \ell^{e-j}]([\ell^j]Q)$ would only take time of $(e - j) \cdot c_{\mathrm{mul}}$. This observation suggests the following implementation.

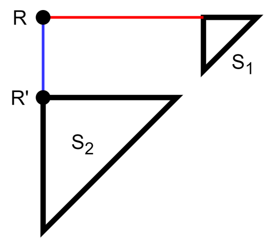


Figure 5.1: Optimization for multi-core platform proposed by [16]: we can compute R' and $\mathcal{S}_2$ without knowing R, thus we can compute R' and $\mathcal{S}_2$ in parallel with R.

Let R denote the result of $P + [m]Q$ and R' denote that of $[\ell^j]P + [m][\ell^j]Q$, where R' is a corner point of a strategy $\mathcal{S}_2$. Since we do not need to know R to compute R' and the computation time of R' is smaller than that of R, Cervantes-Vázquez et al. proposed, for the multi-core platforms, that we devote one core for computing R. While the computation takes place, we use one core to compute R' and then $K - 1$ cores to compute the whole strategy $\mathcal{S}_2$. They would choose j such that the computation time of R is close to the computation time of R' and $\mathcal{S}_2$. After R is computed, several isogeny evaluations are serially performed on R before we start computing $\mathcal{S}_1$ using all K cores.

The optimization of [16] significantly reduces the isogeny computation time. However, it can be further reduced, as we can see that the isogeny evaluations performed on R are done serially and the computation does not utilize all available cores. Here, we apply the PCS technique to fully utilize those cores. The modified PCS that makes use of this optimization is presented in Algorithm 5.1.

The algorithm works in two phases. The first phase is for operations performed after R' is computed but before R. The cost is thus initialized as $(e - j) \cdot c_{\text{mul}}$. In this phase, we can utilize $K' = K - 1$ cores and we cannot compute $\phi(R)$, represented by $(1, 0)$, since the computation of R is not yet finished. The cost computation in Lines 20–24 is done as in the original PCS. In Line 25, we check whether the current cost is at least $e \cdot c_{\text{mul}}$, the cost for computing R. If so, this implies that the computation for R is completed and we can start the next phase. In the second phase, we can utilize all K cores. The algorithm continues until all computational tasks are scheduled. For our algorithm, we do not require that the computation time of R is close to the computation time of R' and $\mathcal{S}_2$.

Algorithm 5.1: Modified PCS for the optimization of [16].

```
    ... (Lines 1–9 are the same as in Algorithm 4.4)
10  cost ← $(e - j) \cdot c_{\text{mul}}$
11  $K' \leftarrow K - 1$
12  Remove vertices $(0,1), \ldots, (0,j)$ and their out-going edges from $\mathcal{S}$
13  **while** $\mathcal{V}_{\mathcal{S}} \neq \emptyset$ **do**
14      $t \leftarrow t + 1$
15      $\mathcal{V}' \leftarrow \{v \in \mathcal{V}_{\mathcal{S}} : \text{in-degree}(v) = 0\}$
16      **if** $K' \neq K$ **then** $\mathcal{V}' \leftarrow \mathcal{V}' \setminus \{(1,0)\}$
17      $S_t \leftarrow \{K'$ vertices in $\mathcal{V}'$ with highest $\mathcal{L}(\cdot)\}$
18      Append $S_t$ to $\mathbb{S}$
19      Remove all vertices in $S_t$ and their out-going edges from $\mathcal{S}$
20      $\text{cost}_t \leftarrow 0$
21      **for** $(i,j) \in S_t$ **do**
22          **if** $\langle (i, j-1), (i,j) \rangle \in \mathcal{E}^*_{\mathcal{S}}$ **then** $\text{cost}_t \leftarrow \max\{\text{cost}_t, c_{\text{mul}}\}$
23          **else** $\text{cost}_t \leftarrow \max\{\text{cost}_t, c_{\text{iso}}\}$
24      $\text{cost} \leftarrow \text{cost} + \text{cost}_t$
25      **if** $\text{cost} \geq e \cdot c_{\text{mul}}$ **then** $K' \leftarrow K$
26  **return** $(\text{cost}, \mathbb{S})$
```

5.2.2 Handling Synchronization

We use the OpenMP API to accommodate multi-threading for our implementation. Although OpenMP is usually used in the single instruction, multiple data (SIMD) paradigm, this tool provides a way to perform different operations on different cores.

The constructs we are using are the OpenMP's `"sections"` and `"section"`. They let us explicitly describe what each core does. Below we show an example of how to use these constructs with three cores. When the program reaches `#pragma omp sections`, it will fork threads and execute them on each core. Each core will execute, in parallel, one of the `#pragma omp section`. When one core is finished, it will wait until all cores are finished. After that, all threads will join together and the program will continue.

```
#pragma omp sections
{
    #pragma omp section
    {
        core1_op();
    }
    #pragma omp section
    {
        core2_op();
    }
    #pragma omp section
    {
        core3_op();
    }
}
```

From our scheduling, it is straightforward to convert $\mathcal{S} = \langle S_1, \ldots, S_n \rangle$ into code: we can have n `"sections"`, the i-th `"sections"` represents S_i, and K `"section"` in each `"sections"`, each represents one core. Nonetheless, the resulting implementation is not effective, as there is overhead when starting and ending `"sections"`. To overcome this issue, it is better to have only one `"sections"` and put all operations of each core across all iterations into each `"section"`, e.g., the first `"section"` includes operations for the first core from all $S_1, \ldots, S_n$.

However, we also need a synchronization mechanism to ensure the order of operations. For instance, it is possible that the second core starts its second operation while the first core is still working on its first operation and the second operation of the second core requires a result of the first operation of the first core. Considering the algorithm that we used to construct $\mathcal{S}$, we should ensure that all operations in S_1 are finished before we start S_2 in our implementation. We solve this issue by implementing our own barriers.

The codes below show an example when working with three cores. The initial values of all variables `Lj_i`, representing a status for j-th core and i-th operation, is 1. For the scheduling, let S_i contain three operations: `core1_opi`, `core2_opi`, and `core3_opi`. We give one operation to each core. After a core performs its task, that core changes the variable to 0, signaling the other cores that its operation is done. Then, that core keeps checking whether other cores finish their operations. As soon as all variables are set to 0, all cores continue to the next operation. This mechanism ensures correct order and is not costly when implemented.

```
#pragma omp sections
{
    #pragma omp section
    {
        core1_op1();
        L1_1 = 0;
        while(L2_1 || L3_1);

        core1_op2();
        L1_2 = 0;
        while(L2_2 || L3_2);

        ...
    }
    #pragma omp section
    {
        core2_op1();
        L2_1 = 0;
        while(L1_1 || L3_1);

        core2_op2();
        L2_2 = 0;
        while(L1_2 || L3_2);

        ...
    }
    #pragma omp section
    {
        core3_op1();
        L3_1 = 0;
        while(L1_1 || L2_1);

        core3_op2();
        L3_2 = 0;
        while(L1_2 || L2_2);

        ...
    }
}
```

We can optimize the implementation further by removing some unnecessary variables. For example, in a case that `corej_op2` depends only on `corej_op1` for each j, the variables `Lj_1` are not necessary and can be removed for efficiency.

5.2.3 Implementation Results

We implement our first isogeny computation, combining all speed-up techniques in previous subsections. The implementations are based on the SIKEp751 parameter set, where the underlying field is $\mathbb{F}_{p^2}$ with $p = 2^{372}3^{239} - 1$. We compute two isogenies of degree 4^{186} and 3^{239}, respectively. After that, we execute them on an Intel(R) Core(TM) i7-8700 processor, benchmarking them with those of [16]. For the reproducibility of the results, the Intel Hyper-Threading and Intel Turbo Boost technologies are disabled. For benchmarking, we employ the same set of parameters, including the prime and the extension field, used in the works that we compare our results with.

Table 5.1 compares execution times of several implementations with one to four cores. The execution times for the single-core setting are shown for reference. For the multi-core setting, we compare four implementations: the implementation of [16] and our implementation which utilizes PCS, each with and without the optimization of [16] described in Section 5.2.1. We note that the optimization is applied only on the implementations of 4^{186}-isogeny computation. For the isogeny computation, there are two rounds for each ℓ^e-isogeny computed: the first round includes the computation of (ϕ, E') from (E, R) and the computation of three image points $\phi(P_1), \phi(P_2), \phi(P_3)$ for some given points P_1, P_2, P_3, while the second round includes only the computation of (ϕ, E') from (E, R).

It is clear from Table 5.1 that there is a reduction in the execution times when there are more cores available. Overall, our implementations have better speed compared to [16], and the reduction percentage increases when there are more cores. For implementations including the optimization of [16], the maximum reduction is up to 14.36% for the second round of 4^{186}-isogeny computation with four cores. For implementations without the optimization, the maximum reduction is up to 16.79% also for the second round of 4^{186}-isogeny computation with four cores. These results show the significance of strategy-level optimization for low-latency parallel isogeny computation.

# Cores	Implementation	4^{186}-isogeny		3^{239}-isogeny	
		Round 1	Round 2	Round 1	Round 2
1	[16]	22.96	18.85	25.98	22.16
2	[16]	20.60	16.47	23.23	19.39
	This work	18.78	14.68	21.24	17.42
	% reduction	8.83	10.87	8.57	10.16
	[16] (∗)	16.30	14.69		
	This work (∗)	14.76	12.85		
	% reduction	9.45	12.53		
3	[16]	19.53	15.42	21.87	18.06
	This work	17.65	13.53	19.60	15.79
	% reduction	9.63	12.26	10.38	12.57
	[16] (∗)	14.95	13.32		
	This work (∗)	13.34	11.71		
	% reduction	10.77	12.09		
4	[16]	19.09	14.89	21.03	17.15
	This work	16.62	12.39	18.68	14.80
	% reduction	12.94	16.79	11.17	13.70
	[16] (∗)	14.08	12.67		
	This work (∗)	12.81	10.85		
	% reduction	9.02	14.36		

Table 5.1: Execution times of various isogeny computation implementations (in million CPU cycles). The first round includes the computation of (ϕ, E') from (E, R) and three image points $\phi(P_1), \phi(P_2), \phi(P_3)$ for some points P_1, P_2, P_3. The second round includes only the computation of (ϕ, E') from (E, R). $(*)$ denotes implementations utilizing the optimization by [16] described in Section 5.2.1. The % reduction shows how much the execution time of our implementation (the row above) is reduced from that of [16] (the row before).

When the aforementioned implementations of isogeny computation is employed to build the complete isogeny-based protocol SIKEp751, we observe superior results with PCS. The description of SIDH and SIKE is provided in Appendix A for reference. We note that the first round of isogeny computation in Table 5.1 refers to the key generation phase of SIDH and the second round refers to the secret agreement phase. In Table 5.2, the data shows similar trends as in Table 5.1. The maximum reduction occurs in the case of encapsulation algorithm when utilizing four cores, with the reduction of up to 11.49%.

# Cores	Implementation	KeyGen	Encaps	Decaps
1	[16]	25.98	42.08	45.18
2	[16] (∗)	23.28	30.76	35.73
	This work (∗)	21.20	27.88	32.18
	% reduction	8.93	9.36	9.94
3	[16] (∗)	21.95	28.32	33.12
	This work (∗)	19.78	25.40	29.59
	% reduction	9.89	10.31	10.66
4	[16] (∗)	21.01	26.72	31.14
	This work (∗)	18.72	23.65	27.74
	% reduction	10.90	11.49	10.92

Table 5.2: Execution times of various SIKE implementations using the SIKEp751 parameter set (in million CPU cycles). (∗) denotes implementations utilizing the optimization by [16] described in Section 5.2.1. The % reduction shows how much the execution time of our implementation (the row above) is reduced from that of [16] (the row before).

5.3 AVX-512 and Multi-Core PCS Implementation

We present in this section our second software implementation of isogeny computation designed for processors supporting AVX-512. We first consider the execution environment for the implementation and propose a modified version of $\mathcal{C}_K^{\mathrm{PCP}^*}$ that is better suited to this setting. We then apply the PCS technique to an implementation of isogeny computation that uses AVX-512 instructions and multi-threading. Lastly, timing results comparing previous implementations with ours are presented.

5.3.1 Constructing Better Strategies using Modified PCP

We consider the implementation of [20] as a starting point. As mentioned in Section 5.1, their implementation sees 512-bit vectors as eight elements. Although we previously described the usage at the low-level operations $a \oplus a'$, the idea can be applied to a higher level of isogeny evaluations. Here, we define n-way isogeny evaluation for $n \in \{8, 4, 2, 1\}$ as representing n elliptic curve points in one vector and evaluating n points simultaneously. They are designed so that we use as many as possible vector elements, although there are fewer than eight points to be evaluated and, as a consequence, fewer points evaluated implies less execution time. As an example, we performed an experiment to obtain execution times for each number of points evaluated concurrently by 4-isogeny. The experiments were performed on an Intel(R) Core(TM) i5-11400 processor. Table 5.3 shows the results.

Points evaluated	1	2	3	4	5	6	7	8
Computation	1-way	2-way	4-way (∗)	4-way	4-way then 1-way	4-way then 2-way	8-way (∗)	8-way
Execution time (CPU cycles)	4900	5800	9300	9400	13600	14500	16400	16500

Table 5.3: Execution times of [20] for different number of points evaluated concurrently by 4-isogeny. (∗) denotes cases where some vector elements are unused.

Based on these results, it is obvious that the execution times differ for different number of points evaluated in the AVX-512 implementation. Therefore, using the equation $\mathcal{C}_K^{\mathrm{PCP}^*}$ with $K = 8$ to construct strategies in this execution environment is not accurate and might not give us the best speed-up, as PCP assumes that K evaluations take time equal to one evaluation. Therefore, before we apply the PCS technique, we should modify PCP first in order to obtain better strategies for the current setting.

For $\mathcal{C}_K^{\mathrm{PCP}^*}(e, k)$, the equation focuses on the number of cores k available to perform operations. We instead focus on the number of operations to be performed. We elaborate on our intuition with Figure 5.2.

Suppose we decompose strategy $\mathcal{S}$ to $\mathcal{S}_1$ and $\mathcal{S}_2$. The costs of point multiplications (shown as a blue vertical thin line) can be determined. One isogeny evaluation connecting $\mathcal{S}_2$ and $\mathcal{S}_1$ (shown as a red horizontal thin line) can also be calculated. For $\mathcal{S}_2$, we see that there is a line representing isogeny evaluations above the triangle of $\mathcal{S}_2$. This line needs to be taken into account when we perform operations of $\mathcal{S}_2$. While PCP says that this line will occupy one core of the processor and we are left with $K - 1$ cores, we note that there

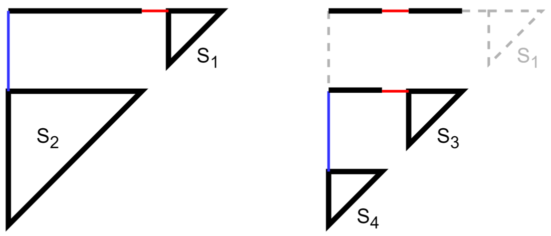


Figure 5.2: Computing the costs of strategies using modified PCP: We need one isogeny evaluation when moving from $\mathcal{S}_2$ to $\mathcal{S}_1$ and two evaluations from $\mathcal{S}_4$ to $\mathcal{S}_3$.

is one line included with $\mathcal{S}_2$. Recursively, $\mathcal{S}_2$ is then decomposed to $\mathcal{S}_3$ and $\mathcal{S}_4$. Since we noted that there is one line above the $\mathcal{S}_2$ triangle, we can infer that there will be one line above the $\mathcal{S}_3$ triangle and two lines above the $\mathcal{S}_4$ triangle. Also, we know that the number of isogeny evaluations between $\mathcal{S}_3$ and $\mathcal{S}_4$ (shown as red horizontal thin lines) is two, which is equal to the number of lines above the $\mathcal{S}_4$ triangle.

To formalize this intuition, let $\mathcal{C}^{\mathrm{MP}^*}(e,r)$ denote the lowest cost of strategies for ℓ^e-isogeny when there are r lines above the strategy triangle. What we would like to determine is $\mathcal{C}^{\mathrm{MP}^*}(e,0)$. The dynamic programming equation for the modified PCP is as follows:

$$\mathcal{C}^{\mathrm{MP}^*}(e,r) = \begin{cases} 0 & \text{if } e=1, \\ \min\limits_{0<i<e}\{\mathcal{C}^{\mathrm{MP}^*}(i,r+1)+\mathcal{C}^{\mathrm{MP}^*}(e-i,r)+f_{\mathrm{mul}}(e-i)+f_{\mathrm{iso}}(r+1)\} & \text{otherwise.} \end{cases}$$

The function $f_{\mathrm{mul}}(n)$ denotes the cost of serially performing n point multiplications by $[\ell]$, and the function $f_{\mathrm{iso}}(n)$ denotes the cost of performing n isogeny evaluations. If we let $f_{\mathrm{mul}}(n) = n\cdot c_{\mathrm{mul}}$ and $f_{\mathrm{iso}}(n) = \lceil n/K\rceil \cdot c_{\mathrm{iso}}$, our $\mathcal{C}^{\mathrm{MP}^*}$ will be $\mathcal{C}^{\mathrm{PCP}^*}$. In other words, our equation is a generalization of PCP.

In the setting of AVX-512, the values of $f_{\mathrm{iso}}(n)$ for $1\le n\le 8$ are taken according to Table 5.3. When $n>8$, it is not straighforward to say that $f_{\mathrm{iso}}(n) = f_{\mathrm{iso}}(n-8)+f_{\mathrm{iso}}(8)$. For an example of $n=10$, we can split the computation to $8+2$, $6+4$, or even $4+4+2$. To obtain the optimal computation, we need another dynamic programming equation:

$$f_{\mathrm{iso}}(n) = \begin{cases} \text{See Table 5.3} & \text{if } 0\le n\le 8, \\ \min\limits_{1\le i\le 8}\{f_{\mathrm{iso}}(n-i)+f_{\mathrm{iso}}(i)\} & \text{otherwise.} \end{cases}$$

Lastly, for the value of $f_{\mathrm{mul}}(n)$, it is almost linear to n. Based on our experiment, [20] gave $f_{\mathrm{mul}}(n) = 5200n+800$ (in CPU cycles) for point multiplications by [4].

5.3.2 Applying PCS to AVX-512 Implementation

After obtaining better strategies, we are ready to apply the PCS technique to them in order to get a more efficient implementation. Since the main advantage of PCS is to perform point multiplications and isogeny evaluations simultaneously but vectorization works well with a single type of operation at a time, we decided to consider multi-threading for our AVX-512 implementation. In particular, we utilize AVX-512 instructions and also two to four cores of the multi-core processor. To the best of our knowledge, this work is the first to combine both vectorization and multi-core processor for isogeny computation.

Revisiting Modified PCP. For $K \in \{2, 3, 4\}$ cores, we can perform up to $8K$ isogeny evaluations at a time. Hence, we revisit our $\mathcal{C}^{\mathrm{MP}^*}$ equation for necessary modifications.

The only thing we need to modify is the function $f_{\mathrm{iso}}(n)$. Now Table 5.3 can be used for n up to $8K$. The same issue happens for $n > 8K$. Nonetheless, the fix is straightforward.

$$f_{\mathrm{iso}}(n) = \begin{cases} \text{See Table 5.3 for } \lceil n/K \rceil & \text{if } 0 \leq n \leq 8K, \\ \min\limits_{1 \leq i \leq 8K} \{ f_{\mathrm{iso}}(n-i) + f_{\mathrm{iso}}(i) \} & \text{otherwise.} \end{cases}$$

Balancing Times of Two Operations. Under PCS, point multiplications by $[\ell]$ and isogeny evaluations are allowed to be performed in parallel on different cores. To effectively perform both, their execution times should not differ much. For example, we see that performing one point multiplication by [4] takes time $f_{\mathrm{mul}}(1) = 6000$ and performing 2-way 4-isogeny evaluations takes time $f_{\mathrm{iso}}(2) = 5800$. Therefore, it is effective to perform one point multiplication by [4] on one core and 2-way 4-isogeny evaluations on another.

Modifying PCS. The PCS technique can be applied to our setting with some changes. To obtain low-latency implementation, we consider two issues. The first one is the balance of operations previously mentioned. By the proof of [42], any two point multiplications by $[\ell]$ in a strategy constructed by $\mathcal{C}^{\mathrm{PCP}^*}$ or $\mathcal{C}^{\mathrm{MP}^*}$ cannot be computed at the same time. This implies that, if one core is used to perform point multiplications, then other remaining cores will be used for isogeny evaluations or left idle. In the former case, the execution times of other cores should be close to the core performing point multiplications. In Algorithm 5.2 which is a modified version of Algorithm 4.4, these are shown in Lines 14–17.

The second issue we consider is the number of isogeny evaluations performed at one time when no point multiplication is available. Under PCP, since K evaluations take time equal to one evaluation, it is best to greedily perform as many evaluations as possible. However, for AVX-512, that is not the case. As an example, suppose $K = 4$ and there are currently 25 isogeny evaluations to be performed. We could perform seven of them on one core and six on three other cores. This will take time $f_{\text{iso}}(7)$. However, it is better to perform only 24 of them and leave one for later, taking time $f_{\text{iso}}(6)$. This has not been considered before because of the design principle of PCP used in all implementations. Thus, our first step is to perform isogeny evaluations as a multiple of K.

We can optimize this further. By looking at Table 5.3 and Section 5.1, it is more effective to perform one, two, four, eight isogeny evaluations at a time on each core. For the example of 25 evaluations, we can perform only 16 of them, rather than 24, with the cost of $f_{\text{iso}}(4)$. By this scheduling, we could effectively utilize AVX-512 instructions. This optimization appears in Lines 18–25 of Algorithm 5.2.

Algorithm 5.2: Modified PCS for K-Core AVX-512 Implementation.

```
   ... (Lines 1–10 are the same as in Algorithm 4.4)
11 while $\mathcal{V}_{\mathcal{S}} \neq \emptyset$ do
12     $t \leftarrow t + 1$
13     $\mathcal{V}' \leftarrow \{v \in \mathcal{V}_{\mathcal{S}} : \text{in-degree}(v) = 0\}$
14     if there exists $(i, j) \in \mathcal{V}'$ such that $\langle (i, j-1), (i, j) \rangle \in \mathcal{E}^*_{\mathcal{S}}$ then
15         $\mathcal{V}' \leftarrow \{2(K-1)$ vertices in $\mathcal{V}' \setminus \{(i, j)\}$ with highest $\mathcal{L}(\cdot)\}$
16         $S_t \leftarrow \{(i, j)\} \cup \mathcal{V}'$
17         $\text{cost}_t \leftarrow \max\{f_{\text{mul}}(1), f_{\text{iso}}(\lceil \#\mathcal{V}'/(K-1) \rceil)\}$
18     else
19         $n \leftarrow \#\mathcal{V}'$
20         if $8K \leq n$          then $n \leftarrow 8K$
21         if $4K \leq n < 8K$ then $n \leftarrow 4K$
22         if $2K \leq n < 4K$ then $n \leftarrow 2K$
23         if $\ K \leq n < 2K$ then $n \leftarrow \ K$
24         $S_t \leftarrow \{n$ vertices in $\mathcal{V}'$ with highest $\mathcal{L}(\cdot)\}$
25         $\text{cost}_t \leftarrow f_{\text{iso}}(\lceil n/K \rceil)$
26     Append $S_t$ to $\mathbb{S}$
27     Remove all vertices in $S_t$ and their out-going edges from $\mathcal{S}$
28     $\text{cost} \leftarrow \text{cost} + \text{cost}_t$
29 return $(\text{cost}, \mathbb{S})$
```

5.3.3 Implementation Results

We implement our second isogeny computation which uses AVX-512 and two-to-four cores, including optimizations in Sections 5.3.2 and 5.2.2. The implementations are based on the SIKEp751 parameter set, where the underlying field is $\mathbb{F}_{p^2}$ with $p = 2^{372}3^{239} - 1$. We compute two isogenies of degree 4^{186} and 3^{239}, respectively. Then, we execute them on an Intel(R) Core(TM) i5-11400 processor, together with other existing works. As usual, we disable the Intel Hyper-Threading and Intel Turbo Boost technologies for reproducibility. For benchmarking, we employ the same set of parameters, including the prime and the extension field, used in the works that we compare our results with.

Table 5.4 shows the results. Two single-core implementations are those proposed by [68] with no use of AVX technologies and [20] with the use of AVX-512. For two to four cores, we present two implementations for each one of them: one is obtained by applying PCP (Section 3.3.2) to strategies constructed from our modified PCP (Section 5.3.1), and the other is obtained by applying our modified PCS (Section 5.3.2) to strategies constructed from our modified PCP (Section 5.3.1). For the isogeny computation, there are two rounds for each ℓ^e-isogeny computed: the first round includes the computation of (ϕ, E') from (E, R) and the computation of three image points $\phi(P_1), \phi(P_2), \phi(P_3)$ for some given points P_1, P_2, P_3, while the second round includes only the computation of (ϕ, E') from (E, R).

# Cores	Implementation	4^{186}-isogeny		3^{239}-isogeny	
		Round 1	Round 2	Round 1	Round 2
1	[68]	20.11	16.49	22.73	19.40
	[20], AVX	8.39	7.71	10.25	9.54
2	This work, AVX, PCP	7.26	6.95	8.63	8.31
	This work, AVX, PCS	6.44	6.34	7.80	7.62
3	This work, AVX, PCP	6.64	6.50	7.94	7.75
	This work, AVX, PCS	5.89	5.96	7.16	7.13
4	This work, AVX, PCP	6.28	6.26	7.51	7.51
	This work, AVX, PCS	5.61	5.75	6.76	6.96
	% reduction	33.13	25.42	34.05	27.04

Table 5.4: Execution times of various isogeny computation implementations (in million CPU cycles). The first round includes the computation of (ϕ, E') from (E, R) and three image points $\phi(P_1), \phi(P_2), \phi(P_3)$ for some points P_1, P_2, P_3. The second round includes only the computation of (ϕ, E') from (E, R). The % reduction shows how much the execution time of our best implementation in the eighth row is reduced from that of [20].

The results clearly show the advantage of using vectorization and multi-core processors for isogeny computation, as all of our multi-core implementations are faster than [68] and [20]. We note that the underlying arithmetic computation of [20] and ours are the same. As indicated in the bottom row of Table 5.4, the reduction is up to 34.05% for Round 1 of 3^{239}-isogeny when utilizing four cores. Once again, the implementation results support the importance of optimizing isogeny computation at the strategy-level. Even though using AVX-512 on a multi-core platform leads to a faster implementation, we may not obtain the best results if we do not consider optimizations for strategy construction and evaluation. By changing the parallelization technique from PCP to PCS, the execution time can be reduced by up to $\frac{6.64-5.89}{6.64} = 11.30\%$ (4^{186}-isogeny, Round 1, three cores).

When the aforementioned implementations of isogeny computation is employed to build the complete isogeny-based protocol SIKEp751, we observe superior results with PCS. The description of SIDH and SIKE is provided in Appendix A for reference. We note that the first round of isogeny computation in Table 5.1 refers to the key generation phase of SIDH and the second round refers to the secret agreement phase. In Table 5.5, the data shows similar trends as in Table 5.4. The maximum speed-up occurs in the case of key generation algorithm when utilizing four cores, with the reduction of up to 34.11% from the single-core implementation of [20].

# Cores	Implementation	KeyGen	Encaps	Decaps
1	[68]	22.88	36.87	44.21
	[20], AVX	10.26	16.12	17.93
	[20], AVX (∗)	10.26	12.80	17.93
2	This work, AVX, PCP	8.61	14.19	15.64
	This work, AVX, PCS	7.77	12.91	14.18
3	This work, AVX, PCP	7.94	13.14	14.43
	This work, AVX, PCS	7.24	11.93	13.11
4	This work, AVX, PCP	7.51	12.56	13.79
	This work, AVX, PCS	6.76	11.41	12.67
	% reduction	34.11	10.86	29.34

Table 5.5: Execution times of various SIKE implementations using the SIKEp751 parameter set (in million CPU cycles). (∗) denotes an implementation where two isogeny computations in Encaps are combined. The % reduction shows how much the execution time of our best implementation in the ninth row is reduced from that of [20] (∗) in the third row.

In addition to benchmarking AVX-512 implementations for cycle counts, we also benchmarked them for wall times (with the Intel Hyper-Threading and Intel Turbo Boost technologies enabled to replicate actual runnings) as in Table 5.6. Based on the results, the cycle counts of all implementations (ref. Table 5.4) correspond to wall times.

# Cores	Implementation	4^{186}-isogeny		3^{239}-isogeny	
		Round 1	Round 2	Round 1	Round 2
1	[68]	4.693	3.787	5.225	4.445
	[20], AVX	1.958	1.801	2.396	2.233
2	This work, AVX, PCP	1.691	1.630	2.017	1.954
	This work, AVX, PCS	1.513	1.506	1.835	1.799
3	This work, AVX, PCP	1.548	1.526	1.872	1.821
	This work, AVX, PCS	1.371	1.397	1.670	1.672
4	This work, AVX, PCP	1.471	1.485	1.762	1.768
	This work, AVX, PCS	1.310	1.363	1.609	1.637

Table 5.6: Execution times of various isogeny computation implementations (in milliseconds). The Intel Hyper-Threading and Intel Turbo Boost technologies are enabled.

5.3.4 Efficiency Analysis

The previous subsection shows 25–34% reduction in the execution time when employing four cores, compared to the single-core implementation of [20]. At first glance, four-time speed-up (or equivalently 75% reduction) might be expected. In this subsection we analyze the theoretical speed-up together with efficiencies of our strategies and implementations. For the analysis, we refer to data under the column of 4^{186}-isogeny, Round 2, in Table 5.4.

In the following, we define the *overall computational cost* for a strategy designed for the multi-core setting as the cost of that strategy when it is evaluated using a single core. The *overall computational cost* reflects the number of operations in a strategy, while the multi-core strategy cost depends both on the number of operations and how much we can parallelize them.

Theoretical Speed-Up. The theoretical single-core strategy cost from [20] (computed using the modified PCP in Section 5.3.1) is 5.53 million CPU cycles, and the four-core strategy cost of that achieved by our four-core implementation (computed by Algorithm 5.2) is 3.39 million CPU cycles. By looking only at the theoretical strategy costs, the expected speed-up is 38.70%. This is 1.5 times higher than what we achieved (25.42%, bottom row of Table 5.4) presumably due to the communication overhead among cores. Next, we analyze the maximum theoretical speed-up. For this, it is common to assume that we have an infinite number of cores. However, [42] and [16] described that the highest level of parallelization is obtained for $e-1$ cores as it is not useful to have more than $e-1$ cores. For the 4^{186}-isogeny, e is 186. In this case, the strategy used for the computation is the *isogeny-based* strategy [30] (i.e., all points except for those in the leftmost column are computed from isogeny evaluations) and its strategy cost is $f_{\mathrm{mul}}(e-1)+(e-1)\cdot f_{\mathrm{iso}}(1)$. For our current setting, this would result in 1.87 million CPU cycles. Hence, even in the case of having plentiful cores, the maximum theoretical speed-up is $\frac{5.53}{1.87}=2.96$ times (or equivalently 66.20% reduction). We note that the actual speed-up from the implementation is expected to be less than this due to the synchronization costs.

Strategy Efficiency. When there are more cores available, the strategies used in our implementations require more computations compared to the one used by [20]. For the strategy used by our four-core implementation, its *overall computational cost* (computed using the modified PCP in Section 5.3.1) is 6.68 million CPU cycles, which is higher than that of [20]. Nonetheless, a higher number of operations allows multiple cores to concurrently perform the computation, resulting in a lower latency. If we apply four cores to the strategy of [20], its strategy cost (computed by Algorithm 5.2) is 4.76 million CPU cycles, which is higher than ours. Therefore, we could say that we increase the *overall computational cost* by $\frac{6.68-5.53}{5.53}=20.80\%$, but reduce the four-core strategy cost by $\frac{4.76-3.39}{4.76}=28.78\%$. This is preferable in order to have low-latency implementations.

Implementation Efficiency. From the aforementioned results, the efficiency of our approach decreases when the number of cores increases. One way of looking at the efficiency of our K-core, $K\in\{2,3,4\}$, implementation is to express it as $\frac{T_1}{K\cdot T_K}$, where T_x is the execution time of the x-core implementation. For our implementations, the efficiencies are $\frac{7.71}{2\times 6.34}=60.80\%$ for two cores, $\frac{7.71}{3\times 5.96}=43.12\%$ for three cores, and $\frac{7.71}{4\times 5.75}=33.52\%$ for four cores. The efficiency is expected to decrease when there are more cores as we trade *overall computational cost* for latency.

In addition, we ran our proposed multi-core implementations utilizing PCS on a single core to have a better understanding of the extent of any overheads in the execution time (e.g., due to communication between cores). The corresponding execution times are shown in Table 5.7, and we compare theoretical costs and actual running times of our multi-core implementations and their serializations corresponding to the computation of 4^{186}-isogeny in Table 5.8. Assuming that the execution times of the serialized versions correspond to their theoretical costs, the overhead percentage increases from $\frac{0.78-0.64}{0.64} = 21.88\%$ for two cores to $\frac{0.57-0.39}{0.39} = 46.15\%$ for four cores. This is another indication that having more cores may not always be beneficial due to an overhead increase.

Implementation	4^{186}-isogeny		3^{239}-isogeny	
	Round 1	Round 2	Round 1	Round 2
Serialization of 2-core PCS	8.86	8.14	10.09	9.57
Serialization of 3-core PCS	9.63	8.87	11.44	10.24
Serialization of 4-core PCS	10.47	10.04	11.62	11.13

Table 5.7: Execution times of the serialized version of our proposed isogeny computation implementations (in million CPU cycles).

Implementation	Theoretical Cost			Execution Time		
	Multi-Core	Serialized	Ratio	Multi-Core	Serialized	Ratio
2-core PCS	4.06	6.39	0.64	6.34	8.14	0.78
3-core PCS	3.66	7.28	0.50	5.96	8.87	0.67
4-core PCS	3.39	8.65	0.39	5.75	10.04	0.57

Table 5.8: Theoretical strategy costs and actual execution times of multi-core and serialized versions of our 4^{186}-isogeny computation implementations, Round 2 (in million CPU cycles). The Ratio columns show the ratio between Multi-Core and Serialized columns.

5.4 Discussion

The implementation results in Sections 5.2 and 5.3 show notable speed-ups when applying PCS with the SIKEp751 parameter set of SIKE as software implementations. In this section, we discuss the applicability of the proposed techniques to a variety of other settings and provide some remarks on the cost functions used in this work.

Applicability to Other Settings. The settings considered below include different vectorization technologies, alternative implementations of arithmetic in the underlying finite field, hardware implementation, and other isogeny-based schemes.

Different vectorization technologies. Although AVX-512 is currently the most powerful extension available, other technologies such as AVX2 may be arguably far more widely used. When using AVX2, vectors are only of size 256 bits and one will need to adjust the implementations of point multiplications, isogeny evaluations, and other primitive arithmetic accordingly. After the implementations of fundamental operations are ready, we require the execution times of those operations (similar to what is discussed in Section 5.3.1) in order to adjust the functions f_{mul}, f_{iso} and apply our proposed techniques. We expect a similar extent of effectiveness when applying our work with AVX2-supported processors. It is also of interest to apply our techniques to ARM's Scalable Vector Extension (SVE).

Alternative arithmetic package. Our techniques are applicable regardless of the implementation of the arithmetic in the underlying field $\mathbb{F}_{p^2}$. The speed of the arithmetic operations has a profound influence on the speed of the implementation. This work relies on the arithmetic implementations of [20, 68] which may no longer represent the state-of-the-art due to the recent result of [63]. A faster arithmetic level implementation will likely result in a higher speed-up. For that, one requires the execution times of those arithmetic operations to adjust with our work.

Hardware implementations. The number of operations that can be performed in parallel can vary based on implementations and the number of logic gates or FPGA slices available. The maximum number of operations performed concurrently and their execution times can be used to customize our work accordingly, similar to earlier discussion. Unlike other works [57, 58], the operations done in each iteration need to be specified explicitly. Thus, the control circuit may become complicated and optimizing it can be challenging.

Other isogeny-based schemes. The proposed implementations have a potential to be adapted for the computation of degree-$\ell_1\ell_2\cdots\ell_n$ isogenies, which are used in some isogeny-based protocols such as CSIDH. The only strategy-level difference is that in this case the isogeny computation is more complex. In particular, the cost of computing an ℓ_i-isogeny differs from that of an ℓ_j-isogeny. This is also true for point multiplication by $[\ell_i]$ and $[\ell_j]$.

To handle these differences, we require some changes to both strategy construction and evaluation. For strategy construction, there are already some works that have studied optimal strategies for the isogeny computation [21, 43]. Although both approaches are designed for serial implementation, they can be extended to vectorized and parallel implementation in the same manner. For strategy evaluation, the scheduling algorithm used by the PCS technique needs to be applicable with tasks of varying length. Example of such algorithms are [22, 37]. Tables 5.9 and 5.10 summarize and compare strategy construction and evaluation techniques for both isogeny computations.

Settings	**Strategy Construction**	**Strategy Evaluation**
Single-Core	Optimal strategies [30]	Sequential
Multi-Core	PCP [16, 42]	Modified PCS (Section 5.2.1)
AVX & Multi-Core	Modified PCP (Section 5.3.1)	Modified PCS (Section 5.3.2)

Table 5.9: Strategy construction and evaluation techniques for computing isogenies of degree ℓ^e in various settings.

Settings	**Strategy Construction**	**Strategy Evaluation**
Single-Core	Optimal strategies [21, 43]	Sequential
Multi-Core	PCP	Modified PCS using [22, 37]
AVX & Multi-Core	Modified PCP	

Table 5.10: Strategy construction and evaluation techniques for computing isogenies of degree $\ell_1\ell_2\cdots\ell_n$ in various settings.

We would like to note that, while extending our work to CSIDH, one also needs to think about the implementation of $\mathbb{F}_p$ arithmetic and the way to handle a case when a sampled point cannot generate an isogeny. In addition, there is a possibility to extend our implementations to SQISign. Since its source code suggests the use of strategies as an improvement, this would be a natural application of our techniques to SQISign. As well, one requires the implementation of $\mathbb{F}_{p^2}$ arithmetic designed specifically for its prime.

The Cost Functions. In this work, the cost of a strategy is based solely on two parameters: the costs of computing point multiplications and isogeny evaluations. This is the approach used in all existing works, to our best knowledge, for both single-core [30] and multi-core platforms [16, 42]. Consequently, the theoretical cost may not be a close approximation of the actual execution time. For better cost computation, one may need to take into account the costs of synchronization, memory access, etc. However, our proposal considers strategy construction and evaluation as separate processes. Thus, we are currently not able to determine the architectural costs during the construction. To handle this, a new computation model needs to be devised, leading to an open problem.

5.5 Chapter Summary

In this chapter, we illustrated how software implementation of large smooth-degree isogeny computation, specifically with vectorization and parallelism, can be further sped-up at the strategy-level. For the first implementation, which considered only the multi-core parallelism, we were able to gain a speed-up when utilizing the modified PCS adapted for the existing optimization. The execution time was reduced by up to 14.36% in our implementation compared to that of [16].

The second implementation, equipped with AVX-512 technology and multi-core processors, combined the use of the modified PCP and PCS in order to provide effective strategies and evaluations crafted for the execution environment. Our benchmarking showed a reduction in execution time of up to 34.05% compared to the single-core implementation of [20] when utilizing up to four cores. Apart from these, our synchronization handling mechanism was also important in achieving a low-latency vectorized and parallel software implementation for the isogeny computation.

At the end of this chapter, we also provided commentaries on the applicability of our work to a variety of other settings including AVX2, hardware implementations, and other isogeny-based schemes such as CSIDH and SQISign. We see that there are promising possibilities to further extend our implementations by considering such executing environments.

Chapter 6

Learning-Based Optimizations

So far, we have proposed *explicit* algorithms to construct and evaluate strategies in order to achieve low cost: construction is mainly based on the recursive structure of canonical strategies, and evaluation is based on precedence-constrained scheduling algorithms. Although we were able to reduce the cost, those approaches are devised based on the perspective and experience of human. This chapter explores some possibilities of using *machine* to provide alternative—and sometimes better—solutions without our guide.

6.1 Using Machine to Solve Optimization Problems

As previously discussed, the problem of constructing a strategy and its evaluation is an optimization problem. In Section 4.2.1, we provided an integer linear program for the case that $c_{\text{mul}} = c_{\text{iso}}$. Here, we define a few problems related to strategies in general.

Definition 6.1 (Strategy problems)**.** Let $\mathcal{C}$ be a function of computing the cost of a strategy $\mathcal{S}$ and $\mathcal{T}$ be a function of computing the cost of a scheduling $\mathbb{S}$. For a given parameter set $(e, c_{\text{mul}}, c_{\text{iso}}, K)$, we define the following problems:

1. *Strategy construction under* $\mathcal{C}$: Construct a strategy $\mathcal{S}$ giving the least cost $\mathcal{C}(\mathcal{S})$ among all possible strategies.

2. *Strategy evaluation for* $\mathcal{S}$: Construct a scheduling $\mathbb{S}$ from $\mathcal{S}$ giving the least cost $\mathcal{T}(\mathbb{S})$ among all possible schedulings of $\mathcal{S}$.

3. *Strategy construction and evaluation*: Construct a pair $(\mathcal{S}, \mathbb{S})$ giving the least cost $\mathcal{T}(\mathbb{S})$ among all possible pairs.

The ultimate goal is to solve the third problem, but we do not see any directions to tackle it at this moment. Up to this point, we principally solve the second problem with PCS and modified PCS, with an attempt to solve the first problem with ILP. We note that, to the best of our knowledge, only the problem of strategy construction under $\mathcal{C}_K^{\mathrm{PCP}}$ is solved for the multi-core setting.

The traditional technique to solve optimization problems is to have domain experts designed some heuristics and construct algorithms from them. However, those can often be suboptimal due to the hard nature of the problems [66]. As alternatives, various techniques let machines to search for good solutions without explicitly mentioning any properties of the problem. In this thesis, we consider two directions of using machines to search for good solutions.

The first direction involves stochastic local search [40] in the space of feasible solutions. Briefly speaking, the search starts with some feasible solutions. Then, more feasible solutions are produced from the current ones and the quality of these newly-produced solutions are used to determine the direction of the search. Some well-known search algorithms include random local search [4], simulated annealing [50], ant colony optimization [27], particle swarm optimization [49], genetic algorithm [39], and other evolutionary algorithms. Several applications of optimizing (cryptographic) computations based on these techniques have been proposed, such as using genetic algorithm to reduce the number of multiplications for matrix multiplication [46], and using random local search to generate fast assembly code for cryptographic primitives [60].

The second direction is based on the emerging trend of *machine learning* algorithms. Machine learning [70] is a study area of computer algorithms that improve automatically through experience or learning from data. According to [12], machine learning algorithms can be classified into three categories:

- *Supervised learning*: Algorithms learn from prepared input-output pairs and try to predict the output for unseen inputs.
- *Unsupervised learning*: Algorithms obtain input data without any corresponding output and try to discover underlying patterns of inputs.
- *Reinforcement learning* [89]: Algorithms, considered as agents, iteratively learn from interactions with their environments and decide which action to perform at each step in order to maximize their rewards.

These techniques are applied to many optimization problems in various research area. For example, reinforcement learning was used to speed-up matrix multiplication algorithms in [28]. We refer the interested readers to [8, 65, 66] for details.

In the next two sections, we apply two above directions to two strategy problems. Specifically, we use genetic algorithm to solve strategy construction problem in searching for low-cost strategies in Section 6.2 and use reinforcement learning algorithm to solve strategy evaluation problem in searching for low-cost schedulings in Section 6.3.

6.2 Strategy Construction with Genetic Algorithm

6.2.1 Genetic Algorithm (GA)

The genetic algorithm (GA) is a search algorithm which mimics the mechanics of natural selection and genetics. It considers an optimization problem as an environment where solutions are individual livings in that environment, and survival and reproduction of individuals are based on the quality or fitness of them (calculated by the objective function).

In general, the genetic algorithm is initialized with a group of randomly generated feasible solutions called *population*. Each individual is represented under some chosen form, usually as a (binary) string or an array. After that, the algorithm performs iteratively. At the end of each iteration, a new population is generated from the current population by some stochastic operations. These operations typically include (i) a process of selecting solutions for reproduction, (ii) a reproduction process to create new solutions (e.g., by crossover or recombination), and (iii) a mutation of newly-generated solutions. Finally, the algorithm stops when the termination condition is met, such as finding a satisfactory solution or completing pre-determined number of iterations. The common framework of the genetic algorithm can be summarized in Algorithm 6.1.

Algorithm 6.1: Genetic algorithm.

```
  Output: A solution of an optimization problem
1 Initialize the population
2 while the termination condition is not met do
3     Select solutions from the population
4     Reproduce new solutions
5     Mutate newly-generated solutions
6     Form new population
7 return the best solution found
```

It is mentioned in [36] that the genetic algorithm is theoretically and empirically proven to provide robust search in complex spaces. It is different from traditional optimization and search in four aspects:

1. GA works with some representation of solutions, not the solutions themselves.
2. GA searches from a group of solutions, not one.
3. GA uses only the objective function without other auxiliary knowledge.
4. GA uses probabilistic rules and not deterministic.

6.2.2 Applying Genetic Algorithm to Our Setting

When applying GA to each optimization problem, the designer of the algorithm still needs to specify some implementation details, including:

- the representation of solutions,
- the size of the population,
- the termination condition, and
- the detail of each stochastic operation.

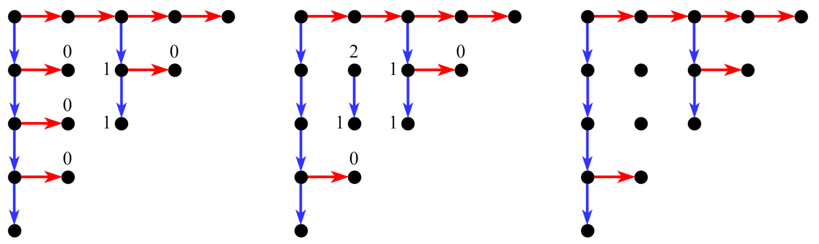
Many of these may not be determined right away and may require some trial-and-error experiments. This section discusses our attempt to utilize GA to construct low-cost strategy $\mathcal{S}$ under a specific cost function $\mathcal{C}(\mathcal{S})$.

The Representation of Solutions. We follow a popular approach and aim to represent strategies using bitstrings. Unlike some optimization problems, it might not be obvious to represent a graph of a strategy as a bitstring. We do so by considering the fact that every vertex in a well-formed strategy must have at most one incoming edge. For vertices $(0, j)$ in the leftmost column and vertices $(i, 0)$ in the top row, their incoming edges are certain. However, for other vertices (i, j) where $i \neq 0$ and $j \neq 0$, there are three options: (i) there is a left-to-right edge from $(i-1, j)$, (ii) there is a top-to-bottom edge from $(i, j-1)$, and (iii) there is no edge. By this, we define two representations for a strategy, one using bitstring and the other using ternary string.

Definition 6.2 (String representations of strategies)**.** For the computation of degree-ℓ^e isogeny, we define two ways of interpreting strings of length $\frac{(e-2)(e-1)}{2}$ as follows. A strategy's vertex (i, j), where $i \neq 0$ and $j \neq 0$, corresponds to the x-th element of the string where $x = \sum_{y=1}^{i-1}(e - y - 1) + j$.

1. *Binary string representation*: Each element of the string can be 0 or 1, where 0 represents the left-to-right edge and 1 represents the top-to-bottom edge.
2. *Ternary string representation*: Each element of the string can be 0, 1, or 2, where 0 represents the left-to-right edge, 1 represents the top-to-bottom edge, and 2 represents no incoming edge.

Example 6.3. For degree-ℓ^5 isogeny, we use strings of length $\frac{3\times4}{2} = 6$ to represent strategies. For binary representation, the string 000110 represents the strategy in Figure 6.1(a). One could obtain the bitstring from the strategy by going through all vertices (i, j) where $i \neq 0$ and $j \neq 0$ from left to right, top to bottom: $(1, 1)$, $(1, 2)$, $(1, 3)$, $(2, 1)$, $(2, 2)$, $(3, 1)$. For ternary representation, the string 210110 represents the strategy in Figure 6.1(b). Both strategies are not well-formed, but we can obtain a well-formed strategy in Figure 6.1(c) by removing some unnecessary edges. Its ternary representation is 220110.



(a) (b) (c)

Figure 6.1: Examples of interpreting binary and ternary strings as strategies.

It is not hard to see that all well-formed strategies can be represented by some strings, thus we cover all solutions in the search space. We note that some ternary strings may not represent valid strategies as some leaves may not be reachable from $(0, 0)$. An obvious example is a ternary string whose all elements are 2. Regarding GA, we extend the definition of our cost functions for strategies to accommodate subgraphs $\mathcal{G}$ of $\mathcal{T}_e$ that are not strategies: $\mathcal{C}(\mathcal{G}) = \infty$.

On the other hand, we prove in the following lemma that all binary strings represent valid strategies (but not necessarily well-formed). This difference between two representations reflects experimental results as we shall see next.

Lemma 6.4. All binary strings in Definition 6.2 represent strategies.

Proof. We can prove the lemma by showing that all leaves $(i, e-i-1)$ are reachable from $(0,0)$ in the corresponding strategies. Instead, we show that all vertices (i,j) are reachable from $(0,0)$. This can be done by an induction on $i+j$.

Basis step ($i+j=0$): $(0,0)$ is reachable from $(0,0)$.

Induction step: Suppose all vertices (i',j') where $i'+j'=n$ are reachable from $(0,0)$, we would like to prove that all vertices (i,j) where $i+j=n+1$ are reachable from $(0,0)$. We consider any vertex (i,j) such that $i+j=n+1$. The case that $i=0$ or $j=0$ is obvious. For the case that $i \neq 0$ and $j \neq 0$, the binary string specifies whether there is an edge from $(i-1,j)$ or $(i,j-1)$. Since $i+j=n+1$, we have $(i-1)+j=i+(j-1)=n$. By induction hypothesis, both vertices are reachable from $(0,0)$. Therefore, by an incoming edge to (i,j), (i,j) is reachable from $(0,0)$. This concludes the proof. $\square$

The Size of Population. We define the representation of strategies for degree-ℓ^e isogeny to be strings of $\frac{(e-2)(e-1)}{2}$ elements. The string length is 17,020 when $e=186$ and 28,203 when $e=239$. For the diversity of the population, we have decided to choose the size of the population to be 10,000. We have tried 100 and 1,000 for the size and the search tends to converge faster but to a strategy with a higher cost. The larger population size is better but the search will take more time. By trial and error, we think 10,000 is a good trade-off.

The Termination Condition. The larger number of iterations is better for the search as it increases an opportunity to find better solutions, but it would result in longer execution time. We obtain satisfactory solutions after 1,000 iterations and use this parameter for the experiments.

The Stochastic Operations. We again follow the common approach and consider three stochastic operations: selection, reproduction (crossover), and mutation. There are various techniques available to choose from, and below are what we used in the experiments.

Selection. For each time that we want to generate new strategies, we randomly select three strategies from the population and keep the best two to be parents.

Reproduction (crossover). For two strings of parents, $s_1 s_2 \cdots s_n$ and $s'_1 s'_2 \cdots s'_n$, we randomly select an index $1 \leq i < n$, and construct two new strings $s_1 \cdots s_i s'_{i+1} \cdots s'_n$ and $s'_1 \cdots s'_i s_{i+1} \cdots s_n$.

Mutation. For newly-reproduced strings, we alter each element to different element (depending whether it is binary or ternary string) with probability 0.001.

New Population Construction. For 10,000 strategies in the population, we reproduce another 10,000 strategies using the above-mentioned approach. Among 20,000 strategies we have, we keep the best 10,000 strategies as the population for the next iteration.

Example 6.5. Consider a case of binary strings and $e = 5$. Suppose two parent strings, shown in Figure 6.2(a), are 110011 and 001101. For the reproduction, let the index be 2. Hence, we construct two new strings 11|1101 and 00|0011, shown in Figure 6.2(b). The mutation process alters the first newly-reproduced string to 111$\underline{0}$01 and alters the second to 0$\underline{1}$0011. The results of all stochastic operations are shown in Figure 6.2(c).

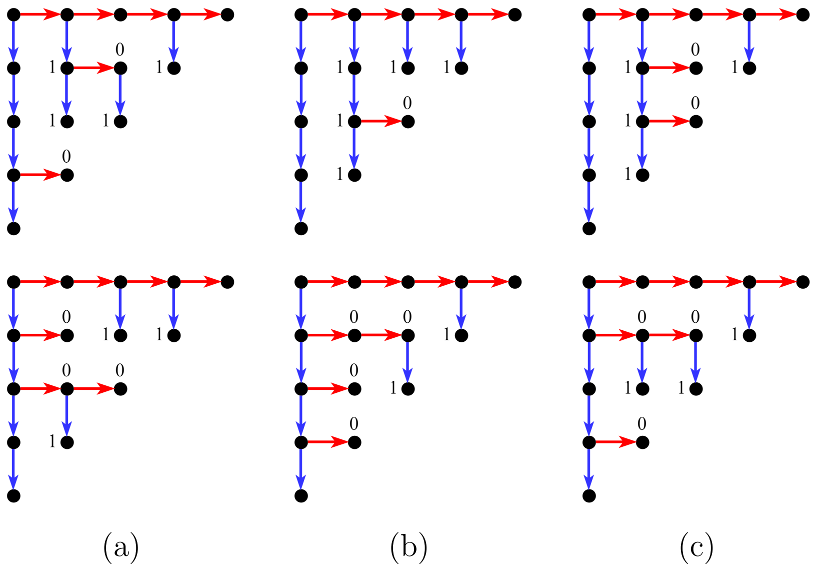


Figure 6.2: Examples of genetic algorithm stochastic operations.

Our genetic algorithm to search for a low-cost strategy for degree-ℓ^e isogeny computation can be summarized as in Algorithm 6.2. We note that each iteration of the for loop in Lines 4–8 is independent and one can speed-up the algorithm by parallelizing this part.

Algorithm 6.2: Genetic algorithm for strategy construction under $\mathcal{C}$.

```
   Output: A low-cost strategy for degree-$\ell^e$ isogeny computation
1  Initialize the population $\mathcal{P}$ with 10,000 random strings of length $\frac{(e-2)(e-1)}{2}$
2  for itr = 1 to 1,000 do
3      $\mathcal{P}' \leftarrow \emptyset$
4      for rep = 1 to 5,000 do
5          Randomly pick three strings from $\mathcal{P}$ and keep two with least $\mathcal{C}(\cdot)$
6          Reproduction with random index $i$
7          Mutation each element of newly-reproduced strings with probability 0.001
8          Add two mutated strings to $\mathcal{P}'$
9      $\mathcal{P} \leftarrow \{$10,000 strategies with least $\mathcal{C}(\cdot)$ in $\mathcal{P} \cup \mathcal{P}'\}$
10 return A strategy with the least cost $\mathcal{C}(\cdot)$ in $\mathcal{P}$
```

We end this subsection by emphasizing our design rationale that our string representations of strategies and the reproduction (crossover) process, as shown in Figure 6.3(a), are designed together such that they reflect the nature of the isogeny computation which is typically performed from left to right in a strategy. Precisely, we think that connecting a good left part of one strategy with a good right part of another will result in a good strategy. Alternatively, one can choose to interpret strings by going through all vertices in top-to-bottom then left-to-right fashion, but when it comes to crossover, strategies will be split and joined horizontally. This is depicted in Figure 6.3(b). Since strategies are evaluated from left to right, we believe that mixing the top part of one strategy with the bottom part of another may not maintain good structures of strategies from one generation to the other, resulting in an ineffective search.

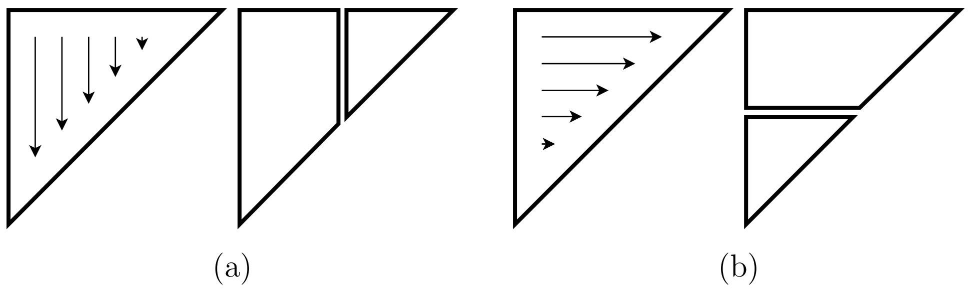

(a) (b)

Figure 6.3: Two approaches for strategy representation and crossover. Arrows show the order of string interpretation.

6.2.3 Experiments and Results

For each parameter set $(e, c_{\text{mul}}, c_{\text{iso}}, K)$, we conduct two searches using GA: the first uses ternary string representations of strategies and the second uses binary string representations. They are denoted in the tables below as GA-3 and GA-2, respectively, and are compared with those from Chapter 4. Again, we consider two parameter sets $(e, c_{\text{mul}}, c_{\text{iso}}) = (186, 25.8, 22.8)$ and $(239, 27, 8, 17)$. The cost function $\mathfrak{C}_K^{\text{Hu}}$ is used as an objective (a.k.a. fitness) function of strategies for GA. The cost reductions in both tables compare the costs of both GA-3 and GA-2 with our previous best results from Chapter 4.

	K	2	3	4	5	6	7	8
Ch. 4	Cost	22081.2	18340.2	16337.4	15269.4	14973.6	14949.6	14063.4
GA-3	Cost	24587.4	18846.6	16455.6	15618.6	14746.8	13931.4	13722.6
	% reduction	-11.35	-2.76	-0.72	-2.29	**1.51**	**6.81**	**2.42**
GA-2	Cost	22848.0	18238.2	16078.2	14889.6	14195.4	13462.2	13120.2
	% reduction	-3.47	**0.56**	**1.59**	**2.49**	**5.20**	**9.95**	**6.71**

Table 6.1: The cost of best strategies from GA under the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}) = (186, 25.8, 22.8)$.

	K	2	3	4	7	8
Ch. 4	Cost	28265.0	23625.0	21282.8	19073.6	18641.2
GA-3	Cost	35513.8	27986.6	24387.2	20572.8	20066.2
	% reduction	-25.65	-18.46	-14.59	-7.86	-7.64
GA-2	Cost	31107.4	25661.2	22820.2	19279.6	18380.6
	% reduction	-10.06	-8.62	-7.22	-1.08	**1.40**

Table 6.2: The cost of best strategies from GA under the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}) = (239, 27.8, 17)$.

The genetic algorithm, specifically GA-2, is able to search for less-cost strategies in several parameter sets. From both tables, GA tends to work better with large K. The reduction can get up to almost 10% for the case of $(e, c_{\text{mul}}, c_{\text{iso}}, K) = (186, 25.8, 22.8, 7)$. As a showcase, we visually present two strategies under this parameter set: one in Figure 6.4 is canonical and recursively constructed in Chapter 4, and the other in Figure 6.5 is constructed by GA-2. We note that the searches terminate after 1,000 iterations, which is approximately three days for each parameter set, and we expect to achieve even less cost when searching longer.

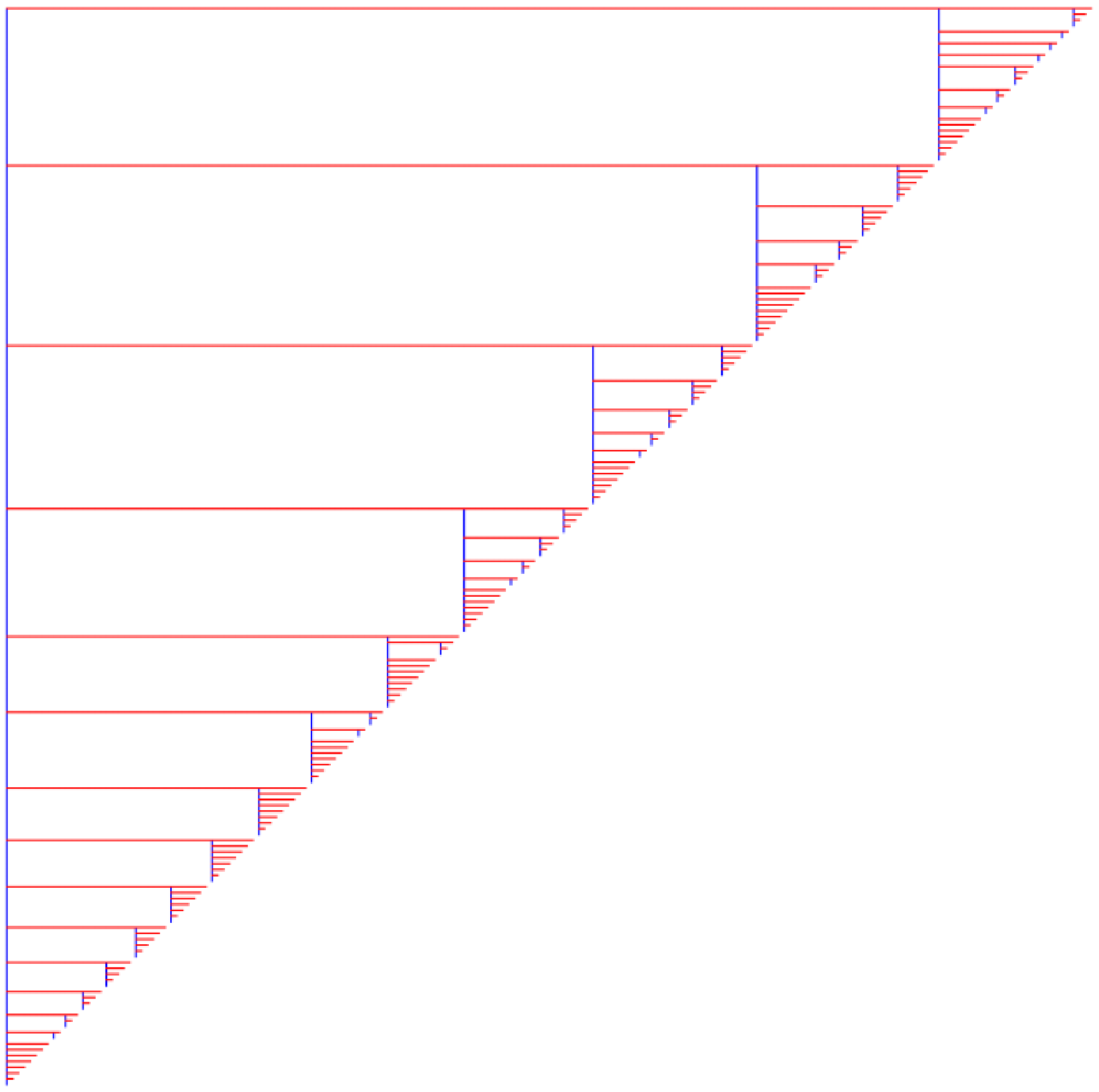


Figure 6.4: Best canonical strategy found in Chapter 4 with the cost of 14949.6 for $(e, c_{\text{mul}}, c_{\text{iso}}, K) = (186, 25.8, 22.8, 7)$.

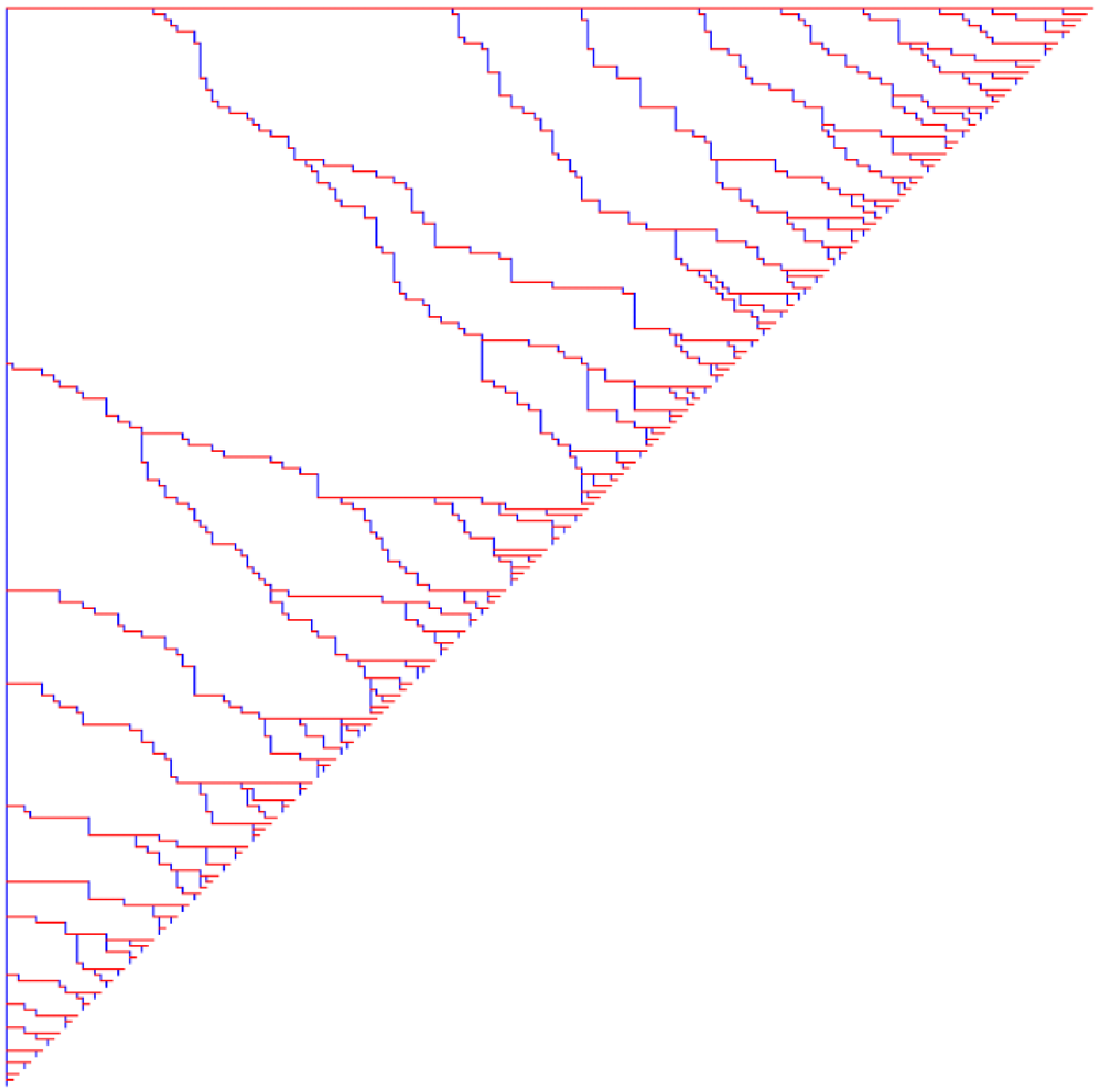


Figure 6.5: Best non-canonical strategy found by GA-2 with the cost of 13462.2 for $(e, c_{\text{mul}}, c_{\text{iso}}, K) = (186, 25.8, 22.8, 7)$.

From the experimental results, one can see the advantage of strategies which do not have nice structures unlike those canonical ones. We believe that canonical strategies limit the use of parallelism to some extent. Below we prove that, apart from the point multiplications in the leftmost column, there can be some operations in a canonical strategy that cannot be parallelized with any other operations. Such operations can be considered as a bottleneck of the computation. Notice that the proof does not apply to non-canonical strategies.

Lemma 6.6. Consider a canonical strategy $\mathcal{S} = (\mathcal{V}_{\mathcal{S}}, \mathcal{E}_{\mathcal{S}})$. If there is an edge $\langle (i,0),(i,1) \rangle$ in $\mathcal{E}_{\mathcal{S}}$, then all edges $\langle (i,j),(i,j+1) \rangle$, $0 \le j < e-i-1$, in $\mathcal{E}_{\mathcal{S}}$ in that column cannot be parallelized with any other operations in $\mathcal{S}$.

Proof. If there is an edge $\langle (i,0),(i,1) \rangle$ in $\mathcal{E}_{\mathcal{S}}$, the point $(i,0)$ must be a corner point of a triangle representing a (sub-)strategy. In the case that $i = 0$, it is obvious that we have to perform all point multiplications $\langle (0,j),(0,j+1) \rangle$, $0 \le j < e-1$, sequentially since we must obtain the kernel generator first before moving to the next column.

In the case that $i \neq 0$, there must be another sub-strategy to the left of the i-th column, following the definition of canonical strategies. That sub-strategy, denoted as $\mathcal{S}'$, can span from the leftmost column (Figure 6.6(a)) or any other column (Figure 6.6(b)). Regardless of the size and structure of $\mathcal{S}'$, in order to obtain $(i,0)$, all computations in $\mathcal{S}'$ must first be completed and then the isogeny evaluation $\langle (i-1,0),(i,0) \rangle$, shown as a red thick line, is performed. Similar to the case of $i = 0$, we must obtain the kernel generator in the i-th column first before moving to the next column. Therefore, all point multiplications in the i-th column, shown as a blue thick line, must be done sequentially. □

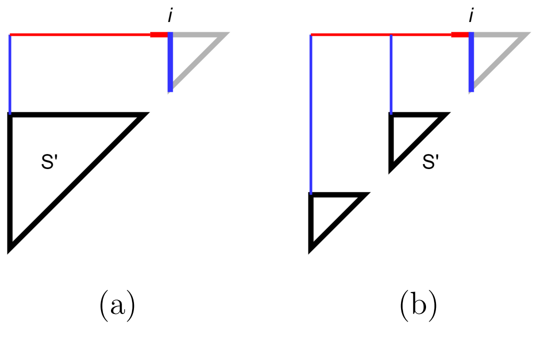


Figure 6.6: Bottleneck operations in canonical strategies.

Although we cannot see any obvious bottleneck in strategies constructed by GA, we are yet to find any patterns of strategies which produce less cost.

Finally, we would like to mention that the string representations of strategies also play an important role in providing effective search. From the results, binary representations (GA-2) give strategies with less costs for all parameter sets. We believe that this is because ternary strings may sometimes give graphs that are not strategies when they are crossed-over and thus considered less useful.

To better see the difference in the effectiveness between GA-3 and GA-2, below we plot the cost of best strategy found from GA-3 and GA-2 during the search when the parameter set used is $(e, c_{\text{mul}}, c_{\text{iso}}, K) = (186, 25.8, 22.8, 7)$. It is evident that GA-2 could construct a strategy with less cost and such strategy can be found faster compared to that of GA-3.

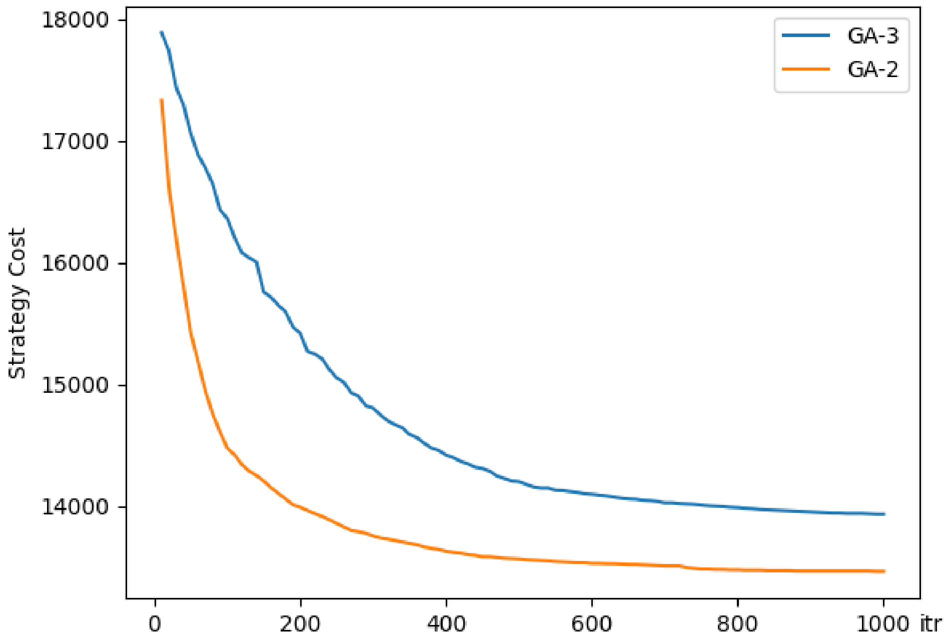


Figure 6.7: A plot for the cost of best strategy found from GA-3 and GA-2.

Even though Algorithm 6.2 with binary representations could construct non-canonical strategies with less cost, we are yet to prove the optimality of its outputs. It may be possible to present another instantiation of the genetic algorithm for strategy construction under $\mathcal{C}$ which is more effective than what we proposed in this section. At this stage, we leave this as an open research question.

6.3 Strategy Evaluation with Reinforcement Learning

6.3.1 Reinforcement Learning (RL)

Reinforcement learning (RL) is an area of machine learning which involves learning which action to take in each situation in order to maximize a reward. The learner, called an *agent*, is not told what to do but must learn by trying those actions. When an agent chooses an action to take, the current situation changes based on the chosen action. The agent then incorporate the new situation to choose its next action, and so on. In many settings, chosen actions not only affect the immediate reward but also the next situation and subsequent rewards. Thus, an intelligent agent has to consider long-term effects when making a decision. The followings are the three most important distinguishing features of reinforcement learning problems [89]:

1. The problem is *close-loop* in a way that agent's actions influence its later inputs.
2. An agent is not directly instructed but learns from its rewards.
3. Actions have long-term consequences in terms of future situations and rewards.

It is common to describe the interaction of the agent to the environment using the following figure. In one situation or *state*, an agent first chooses an action, interacting with the environment. Next, the environment responds by providing a new state where the agent is now in and a reward for performing such action. Treating the new state and reward as inputs, the agent then chooses the next action.

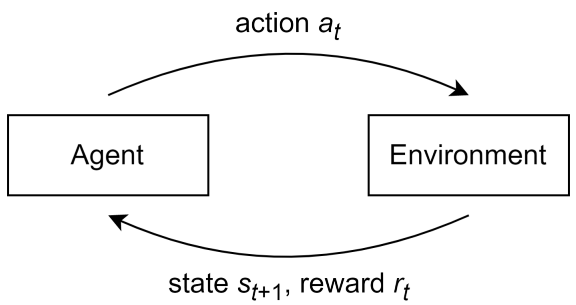


Figure 6.8: A framework of reinforcement learning.

Exploration and Exploitation. One of the challenges which is specific to RL, and not other machine learning categories, is the trade-off between *exploration* and *exploitation*. When an agent decides which action to take, it may choose to perform an action which was already taken during previous tries and gave a high (or even the highest) reward (exploitation). However, by only sticking with what the agent already knows, it loses an opportunity to try other choices which may lead to a higher reward (exploration). The exploration-exploitation dilemma is to balance both operations and still obtain high rewards. The problem has been modeled as the multi-armed bandit problem first described in [80]. In this thesis, we do not go into its detail and briefly consider the dilemma when we discuss the learning algorithm.

Markov Decision Process (MDP). The environment to which an agent interacts is usually described by a (finite) Markov decision process [7]. We provide its definition based on that appeared in [6, 66].

Definition 6.7 (Markov decision process)**.** A *Markov decision process* is a tuple $\mathbf{M} = (\mathbf{S}, \mathbf{A}, \mathbf{R}, \mathbf{P})$ where

- $\mathbf{S}$ is a countable set of states,
- $\mathbf{A}$ is a set of actions,
- $\mathbf{R} : \mathbf{S} \times \mathbf{A} \to \mathbb{R}$ is a reward function, and
- $\mathbf{P} : \mathbf{S} \times \mathbf{A} \times \mathbf{S} \to [0, 1]$ is the transition probability function such that for all states $s \in \mathbf{S}$ and actions $a \in \mathbf{A}$: $\displaystyle\sum_{s' \in \mathbf{S}} \mathbf{P}(s, a, s') \in \{0, 1\}$.

The transition probability function $\mathbf{P}(s, a, s')$ specifies the probability that the current state is changed to s' when taking an action a at s. According to Figure 6.8, an agent, choosing an action a_t at a state s_t, receives the next state s_{t+1} following the distribution $\mathbf{P}(s_t, a_t, \cdot)$ and a reward $r_t = \mathbf{R}(s_t, a_t)$. As this process continues, an agent, starting at a state s_0, will create a sequence $\langle s_0, a_0, s_1, a_1, \ldots \rangle$ which provides a reward of $\sum_t \mathbf{R}(s_t, a_t)$.

The decision of which action a to choose at s can be described by a *policy* $\pi : s \mapsto a$ which can be probabilistic. When following π, we have $a_t = \pi(s_t)$. Thus, the goal of an RL agent is to find an optimal policy π^* that maximizes the expected cumulative sum of rewards: $\pi^* = \text{argmax}_\pi\{\mathbb{E}[\sum_t \mathbf{R}(s_t, \pi(s_t))]\}$. In many settings, short-term rewards are more preferable and the goal of an agent is defined to be finding π^* that maximizes the expected cumulative *discounted* sum of rewards: $\pi^* = \text{argmax}_\pi\{\mathbb{E}[\sum_t \gamma^t \mathbf{R}(s_t, \pi(s_t))]\}$ for a *discount rate* $0 < \gamma \leq 1$.

Value Functions. In order to find π^* for an MDP $\mathbf{M}$, we can apply value-based RL algorithms. The algorithms mainly consider the following two functions:

- The state-value function $V^* : \mathbf{S} \to \mathbb{R}$, where $V^*(s)$ is the maximum expected cumulative discounted sum of rewards when starting at s, and
- The action-value function $Q^* : \mathbf{S} \times \mathbf{A} \to \mathbb{R}$, where $Q^*(s, a)$ is the maximum expected cumulative discounted sum of rewards when performing a at s.

By the above definitions, they satisfy particular recursive (i.e., dynamic programming) relationships called *Bellman equations* as follows:

$$V^*(s) = \max_a \left\{ \mathbf{R}(s,a) + \gamma \left(\sum_{s'} \left(\mathbf{P}(s,a,s') \cdot V^*(s') \right) \right) \right\},$$
$$Q^*(s,a) = \mathbf{R}(s,a) + \gamma \left(\sum_{s'} \left(\mathbf{P}(s,a,s') \cdot \max_{a'} \left\{ Q^*(s',a') \right\} \right) \right).$$

Also, both functions are related to each other:

$$V^*(s) = \max_a \{Q^*(s,a)\},$$
$$Q^*(s,a) = \mathbf{R}(s,a) + \gamma \left(\sum_{s'} \left(\mathbf{P}(s,a,s') \cdot V^*(s') \right) \right).$$

Therefore, if one knows V^* or Q^* for a given MDP $\mathbf{M}$, it is straightforward to determine an optimal action for each state and thus construct π^*. When the number of (s, a) pairs is not too large and all pairs can be iterated, an algorithm called *value iteration* [18, 77] can be used to approximate V^* and Q^*.

Q-Learning. In various settings, the number of (s, a) pairs is very large (or even uncountable) and we cannot go through all possible pairs. Some examples include cases where $\mathbf{A}$ is continuous and cases where $\mathbf{S}$ is very large such as a game of chess which has $\approx 10^{43}$ (or 2^{143}) states [85]. Instead of iterating through all (s, a) pairs, many techniques use random sampling in order to approximate V^* and Q^*. In this thesis, we consider one such algorithm called *Q-learning* [95], but other techniques can be utilized, e.g., Monte Carlo methods [67] and Temporal-Difference (TD) learning [51, 83].

Q-learning estimates Q^* function by building its approximation Q from what an agent has experienced. The value of $Q(s,a)$ is initialized arbitrarily. When an agent explores the environment and obtains s' and $\mathbf{R}(s,a)$ from choosing a at s, the value of $Q(s,a)$ is updated using the following operation, sometimes called *Bellman operation*:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[\mathbf{R}(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)],$$

where $\alpha \in (0,1]$ is called the *step-size parameter*, influencing the rate of learning. In brief, the value of $Q(s,a)$ is updated towards the optimal rewards $\mathbf{R}(s,a) + \gamma \max_{a'} Q(s',a')$ that the agent can obtain, from the agent's current perspective of Q. Under an assumption that all (s,a) pairs are visited infinitely often, it has been proved that $Q(s,a)$ converges to $Q^*(s,a)$ with probability 1 [89].

As previously mentioned, an agent must balance between exploration and exploitation when choosing actions to perform. One solution to this is to use ϵ-*greedy* method to select actions almost-greedily: at a state s, an agent selects the best action $a^* = \text{argmax}_a\{Q(s,a)\}$ with probability $1-\epsilon$, and select a random action in $\mathbf{A}$ with probability ϵ. As an option, it is possible to reduce ϵ over time. Apart from this, other methods for exploration-exploitation lemma include Boltzmann exploration (Softmax method) [89], Pursuit methods [90], Upper Confidence Bounds (UCB) methods [3], and Thompson sampling [91].

Putting all together, the Q-learning algorithm can be expressed as in Algorithm 6.3. Note that the algorithm updates the value of $Q(s,a)$ even before reaching a terminal state.

Algorithm 6.3: Q-learning algorithm.

```
Input  : An MDP $\mathbf{M} = (\mathbf{S}, \mathbf{A}, \mathbf{R}, \mathbf{P})$ and an initial state $s_0$
Output: A policy $\pi$
1  Initialize $Q(s,a)$ arbitrarily, except for $Q(\text{terminal-state}, \cdot) = 0$
2  while the termination condition is not met do
3      $s \leftarrow s_0$
4      while $s$ is not a terminal state do
5          Choose $a \in \mathbf{A}$ using $\epsilon$-greedy method based on $Q(s,a)$
6          Take action $a$, observe $s'$ and $\mathbf{R}(s,a)$
7          $Q(s,a) \leftarrow Q(s,a) + \alpha[\mathbf{R}(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
8          $s \leftarrow s'$
9  Construct $\pi$ from $Q$
10 return $\pi$
```

6.3.2 Applying Reinforcement Learning to Our Setting

From the description of RL, it is suitable to be applied with the optimization problem of evaluating a given strategy $\mathcal{S} = (\mathcal{V}_{\mathcal{S}}, \mathcal{E}_{\mathcal{S}})$. We discuss our attempt in this section. To do so, we model the problem as an MDP and utilize the value functions.

MDP for Our Setting. For evaluating a strategy $\mathcal{S} = (\mathcal{V}_{\mathcal{S}}, \mathcal{E}_{\mathcal{S}})$, we define a state $s \in \mathbf{S}$ to be a subset of $\mathcal{V}_{\mathcal{S}}$ representing points computed so far. An action $a \in \mathbf{A}$ for a given state s is thus a subset of $\mathcal{V}_{\mathcal{S}}$ (of size at most K) representing available points to be computed next. The reward $\mathbf{R}(s, a)$ of an action a is then the cost of simultaneously computing points in a on K cores, in terms of c_{mul} and c_{iso}. For the transition probability function $\mathbf{P}$, our setting does not involve probabilistic transition as the next state can be deterministically determined from the current state and an action. Hence, instead of using $\mathbf{P}$, we define the transition function $\mathbf{N} : \mathbf{S} \times \mathbf{A} \to \mathbf{S}$ which gives the next state $\mathbf{N}(s, a)$ for a pair (s, a).

From an implementation view point, defining a state s as a set of points already computed could be memory-consuming since the size of s is getting larger when more points are computed. Equivalently, a state s can be defined as a set of points that are currently available to be computed. As a result, an action a for a state s is simply a subset of s (of size at most K). We give an example to illustrate two ways of defining a state.

Example 6.8. Consider a strategy in Figure 6.9(a). During an evaluation, two possible configurations we can have are shown in Figures 6.9(b) and 6.9(c). For the configuration in Figure 6.9(b), defining a state using computed points gives $s = \{(0,1), (0,2), (0,3), (0,4), (1,1), (1,3)\}$, while defining a state using available points (shown in yellow) gives $s = \{(1,0), (1,2), (2,1)\}$ which is smaller. For two different configurations in Figures 6.9(b) and 6.9(c), one can see that the set of computed points are different, and the set of available points are also different.

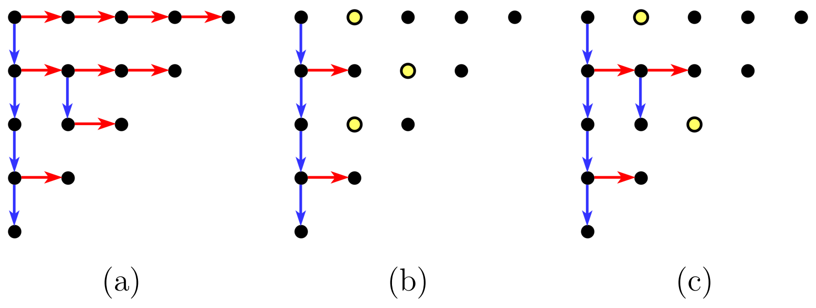


Figure 6.9: Two ways of defining a state.

Below we provide two proofs: the first one indicates the upperbound for the number of points used to define a state for each definition, and the second one shows the correspondence between two definitions of states. Consequently, we decide to define a state as the set of available points in a configuration.

Lemma 6.9. Consider only well-formed strategies, the upperbound for the number of computed points in a configuration is $\frac{e(e+1)}{2} - 1$, and the upperbound for the number of available points in a configuration is e.

Proof. The maximum number of points in a strategy is $\frac{e(e+1)}{2}$ (e.g., in the multiplicative-based and isogeny-based strategies). Since $(0, 0)$ is given as input, the upperbound for the number of computed points in a configuration is $\frac{e(e+1)}{2} - 1$.

Regarding the upper bound for number of available points, we consider a well-formed strategy as consisting of e paths from $(0, 0)$ to each leaf. In a valid configuration, there can be at most one available point on each path. Therefore, the upperbound for the number of available points in a configuration is e. □

Lemma 6.10. Two definitions of states are the same. Formally, the sets of available points are the same for two configurations if and only if the sets of computed points are the same for both configurations.

Proof. If the sets of computed points are the same for two configurations, it is obvious that the sets of available points are the same. For the other direction, we assume, for the sake of contraposition, that the sets of computed points are different between two configurations. We will show that the sets of available points are different between both configurations.

Since the sets of computed points are different, there must be at least one point that is computed in one configuration and not in the other. Also, among those points, there must be the leftmost-and-uppermost point. Let that point be (i, j). Without loss of generality, let (i, j) be not computed in the first configuration and computed in the second configuration. Next, we consider two possible cases.

Case i: (i, j) is computed from $(i, j - 1)$. By how we choose (i, j), $(i, j - 1)$ must be computed in both configurations. Since (i, j) is not computed in the first configuration and computed in the second, this implies that (i, j) is available in the first configuration and not in the second.

Case ii: (i, j) is computed from $(i - 1, j)$. By how we choose (i, j), $(i - 1, j)$ and $(i - 1, e - i)$ must be computed in both configurations. Since (i, j) is not computed in the first configuration and computed in the second, this implies that (i, j) is available in the first configuration and not in the second. This concludes the proof. □

Next, we consider possible actions for a state s. For a state s with n available points, the number of possible actions is $\sum_{k=1}^{\min\{n,K\}} \binom{n}{k}$. However, some actions can be removed from consideration in order to reduce the size of the search space. The following proof describes which actions we need to consider for RL algorithms.

Lemma 6.11. For a state s_i with n_i available points, let $s_i = s_{i,\text{mul}} \cup s_{i,\text{iso}}$ be a union of the sets of available points to be computed by point multiplication and isogeny evaluation, respectively. Let the sizes of $s_{i,\text{mul}}$ and $s_{i,\text{iso}}$ be $n_{i,\text{mul}}$ and $n_{i,\text{iso}}$, respectively. Then, there exists an optimal sequence $\langle s_0, a_0, s_1, a_1, \ldots, s_t\rangle$ providing an optimal reward $\sum_{i=0}^{t-1} \mathbf{R}(s_i, a_i)$ such that each a_i follows one of the following three cases:

1. a_i is a subset of $s_{i,\text{mul}}$ of size $\min\{n_{i,\text{mul}}, K\}$,
2. a_i is a subset of $s_{i,\text{iso}}$ of size $\min\{n_{i,\text{iso}}, K\}$,
3. a_i is a subset of s_i of size $\min\{n_i, K\}$.

Proof. An action a_i can have three possible rewards: c_{mul}, c_{iso}, and $\max\{c_{\text{mul}}, c_{\text{iso}}\}$. In the case of c_{mul}, all points in a_i must be from $s_{i,\text{mul}}$. We show that there exists an optimal a_i of size $\min\{n_{i,\text{mul}}, K\}$. The proof below applies to the cases of c_{iso} and $\max\{c_{\text{mul}}, c_{\text{iso}}\}$.

Suppose there is an optimal sequence $\langle s_0^*, a_0^*, s_1^*, a_1^*, \ldots, s_t^*\rangle$ where there exists $a_i^* \subseteq s_{i,\text{mul}}^*$ of size less than $\min\{n_{i,\text{mul}}^*, K\}$. Thus, there must be a point $(x, y) \in s_i^*$ that is available to be computed but is not included in a_i^*. This point (x, y) must be in some a_j^* for $j > i$ and in all s_k^* for $i \leq k \leq j$. We can construct another optimal sequence

$$\langle s_0^*, a_0^*, \ldots, s_i^*, a_i', s_{i+1}', a_{i+1}^*, \ldots, s_{j-1}', a_{j-1}^*, s_j', a_j', s_{j+1}^*, \ldots, s_t^*\rangle$$

where $a_i' = a_i^* \cup \{(x, y)\}$, $a_j' = a_j^* \setminus \{(x, y)\}$, and $s_k' = s_k^* \setminus \{(x, y)\}$ for $i < k \leq j$. This is a valid sequence since all points that must be computed after (x, y) are still computed after (x, y) and the size of a_i' is no more than K. Also, this sequence is still optimal, as $\mathbf{R}(s_i^*, a_i') = \mathbf{R}(s_i^*, a_i^*) = c_{\text{mul}}$, $\mathbf{R}(s_k', a_k^*) = \mathbf{R}(s_k^*, a_k^*)$ for all $i < k < j$, and $\mathbf{R}(s_j', a_j') \leq \mathbf{R}(s_j^*, a_j^*)$ due to the fact that $a_j' \subseteq a_j^*$. This process can be repeated until a_i' is of size $\min\{n_{i,\text{mul}}^*, K\}$. Therefore, we have proved that there exists an optimal sequence $\langle s_0, a_0, s_1, a_1, \ldots, s_t\rangle$ where each a_i belongs to one of the three cases as desired. $\square$

From the above-mentioned proof, we reduce the number of actions to be considered from $\sum_{k=1}^{\min\{n,K\}} \binom{n}{k}$ to at most $\binom{n_{\text{mul}}}{\min\{n_{\text{mul}},K\}} + \binom{n_{\text{iso}}}{\min\{n_{\text{iso}},K\}} + \binom{n}{\min\{n,K\}}$. As an example, in the case where $n_{\text{mul}} = n_{\text{iso}} = K = 8$ and $n = n_{\text{mul}} + n_{\text{iso}} = 16$, we reduce the number of actions from 39,202 to at most 12,872. Nonetheless, the number of actions to be considered is still large.

Applying Q-Learning. We define $\mathbf{R}(s,a)$ to be a positive number and formulate our problem as a minimization problem. Since our problem do not involve probabilistic transitions and we consider rewards equally throughout the exploration (i.e., $\gamma = 1$), we can simplify two value functions as follows:

$$V^*(s) = \min_a \{\mathbf{R}(s,a) + V^*(\mathbf{N}(s,a))\},$$
$$Q^*(s,a) = \mathbf{R}(s,a) + \min_{a'} \{Q^*(\mathbf{N}(s,a),a')\}.$$

As we can see that the number of actions per state can be large, we will instead estimate V^* function by building its approximation V. Precisely, in our setting $V^*(s)$ is the length of a shortest path from s to the terminal state and $V(s)$ is the upperbound of $V^*(s)$. Using the framework of Q-learning, below is the update (i.e., Bellman) operation for our V function when an agent at a state s performs an action a and moves to the next state $s' = \mathbf{N}(s,a)$:

$$V(s) \leftarrow \min\{V(s), \mathbf{R}(s,a) + V(s')\}.$$

Our RL algorithm for strategy evaluation for $\mathcal{S}$ can be written as Algorithm 6.4. This is similar to a randomized shortest path algorithm with an application of ϵ-greedy method. For the termination condition, we let an agent explore for 500,000 times.

Algorithm 6.4: Q-learning algorithm for strategy evaluation for $\mathcal{S}$.

```
Input  : A strategy $\mathcal{S}$
Output: A scheduling $\mathbb{S}$ and its cost
1  Initialize $V(s) = \infty$, except for $V(\emptyset) = 0$
2  for itr = 1 to 500,000 do
3      $s \leftarrow \{(0,1)\}$
4      while $s \neq \emptyset$ do
5          Choose $a \subseteq s$ using $\epsilon$-greedy method based on $\mathbf{R}(s,a) + V(\mathbf{N}(s,a))$
6          Take action $a$, observe $s' = \mathbf{N}(s,a)$ and $\mathbf{R}(s,a)$
7          $V(s) \leftarrow \min\{V(s), \mathbf{R}(s,a) + V(s')\}$
8          $s \leftarrow s'$
9  $\mathbb{S} \leftarrow \langle\rangle$; $t \leftarrow 0$; cost $\leftarrow 0$; $s \leftarrow \{(0,1)\}$
10 while $s \neq \emptyset$ do
11     $t \leftarrow t + 1$
12     $S_t \leftarrow \text{argmin}_a\{\mathbf{R}(s,a) + V(\mathbf{N}(s,a))\}$
13     Append $S_t$ to $\mathbb{S}$
14     cost $\leftarrow$ cost $+ \mathbf{R}(s,a)$
15     $s \leftarrow \mathbf{N}(s,a)$
16 return (cost, $\mathbb{S}$)
```

Regarding the ϵ-greedy method, with probability ϵ we select an action uniformly at random. For a state s, let $s = s_{\text{mul}} \cup s_{\text{iso}}$ as previously defined and the sizes of s and s_{iso} are n and n_{iso}, respectively. Also, let $m = \binom{n}{\min\{n,K\}}$. For the case that $c_{\text{mul}} > c_{\text{iso}}$, we use the following procedure to uniformly select an action a for s at random:

- If $0 < n_{\text{iso}} < K$, then
  with prob. $\frac{1}{m+1}$, $a \leftarrow s_{\text{iso}}$,
  with prob. $\frac{m}{m+1}$, $a \leftarrow$ a random subset of s of size $\min\{n, K\}$, each with prob. $\frac{1}{m}$.
- Otherwise, $a \leftarrow$ a random subset of s of size $\min\{n, K\}$, each with prob. $\frac{1}{m}$.

Lemma 6.12. Let $c_{\text{mul}} > c_{\text{iso}}$. The above procedure considers all actions (according to Lemma 6.11) with equal probability.

Proof. When $c_{\text{mul}} > c_{\text{iso}}$, $\max\{c_{\text{mul}}, c_{\text{iso}}\} = c_{\text{mul}}$ and we can consider only Cases 2 and 3 of Lemma 6.11 for actions to be considered. When $n_{\text{iso}} = 0$, we uniformly select a subset of s of size $\min\{n, K\}$ as an action (Case 3) and there are m such sets. When $n_{\text{iso}} \geq K$, a subset of s_{iso} of size $\min\{n_{\text{iso}}, K\} = K$ (Case 2) is already considered when we consider a subset of s of size $\min\{n, K\} = K$ (Case 3). Thus, the cases of $n_{\text{iso}} = 0$ or $\geq K$ are correct.

When $0 < n_{\text{iso}} < K$, there are two types of actions to be considered according to Lemma 6.11: (i) a subset of s_{iso} of size $\min\{n_{\text{iso}}, K\} = n_{\text{iso}}$ (Case 2), and (ii) a subset of s of size $\min\{n, K\}$ (Case 3). For (i), there is only one such set (i.e., s_{iso}). For (ii) there are m such sets. If $n > n_{\text{iso}}$, sets in (i) and (ii) are different (because $n_{\text{iso}} \neq \min\{n, K\}$) and hence there are $m + 1$ possible actions in total. The procedure selects each action with probability $\frac{1}{m+1}$. If $n = n_{\text{iso}}$, we have $s = s_{\text{iso}}$ and sets in (i) and (ii) are both s_{iso}. Nonetheless, the procedure selects $a \leftarrow s_{\text{iso}}$ for both (i) and (ii). Therefore, the case of $0 < n_{\text{iso}} < K$ is also correct. □

Online and Offline Updates. One feature of Q-learning is that the value function is updated online before reaching a terminal state. This is useful in various cases. For example, an agent will obtain updated values of states/actions when it visits the same state during the exploration, and sometimes it takes long to reach a terminal state and storing all explored states consumes a large amount of space.

For our setting, it is possible to have an offline update: an agent explores until it reaches the terminal state and then updates V from the terminal state back to the initial state. Formally, for $\langle s_0, a_0, s_1, a_1, \ldots, s_t \rangle$, we update $V(s_i)$ from $i = t - 1$ to 0 using the Bellman operation. By this offline update, the reward of the path propagates back to the initial state faster. For comparison, we try both online and offline updates in our experiments.

6.3.3 Experiments and Results

For each parameter set $(e, c_{\text{mul}}, c_{\text{iso}}, K)$, we conduct two experiments using RL: the first follows Algorithm 6.4 with online updates and the second performs offline updates. They are denoted in the tables below as QL online and QL offline, respectively. The input strategies for the experiments are the least cost ones found in Section 4.3 (by PCS+ILP) and Section 6.2.3 (by GA) under two parameter sets $(e, c_{\text{mul}}, c_{\text{iso}}) = (186, 25.8, 22.8)$ and $(239, 27.8, 17)$. We provide the costs of strategies using $\mathcal{C}_K^{\text{Hu}}$ and $\mathcal{C}_K^{\text{CG}}$ as baselines for comparison. The rows *% diff* show the difference between the costs of RL experiments with $\min\{\mathcal{C}_K^{\text{Hu}}, \mathcal{C}_K^{\text{CG}}\}$.

	K	2	4	8
$\mathcal{C}_K^{\text{Hu}}$	Cost	22081.2	16078.2	13120.2
$\mathcal{C}_K^{\text{CG}}$	Cost	22081.2	16126.2	13153.2
$\min\{\mathcal{C}_K^{\text{Hu}}, \mathcal{C}_K^{\text{CG}}\}$	Cost	22081.2	16078.2	13120.2
QL online	Cost	23560.2	17214.6	13621.2
	% diff	6.70	7.07	3.82
QL offline	Cost	23355.0	16921.2	13509.0
	% diff	5.77	5.24	2.96

Table 6.3: The cost of strategy scheduling from RL under the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}) = (186, 25.8, 22.8)$.

	K	2	4	8
$\mathcal{C}_K^{\text{Hu}}$	Cost	28265.0	21282.8	18380.6
$\mathcal{C}_K^{\text{CG}}$	Cost	28265.0	21282.8	18812.6
$\min\{\mathcal{C}_K^{\text{Hu}}, \mathcal{C}_K^{\text{CG}}\}$	Cost	28265.0	21282.8	18380.6
QL online	Cost	29608.0	23152.8	19055.8
	% diff	4.75	8.79	3.67
QL offline	Cost	29370.0	22914.8	18882.8
	% diff	3.91	7.67	2.73

Table 6.4: The cost of strategy scheduling from RL under the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}) = (239, 27.8, 17)$.

From the results, we see that our reinforcement learning algorithms are not able to outperform Hu's and Coffman-Graham's scheduling algorithms for strategy evaluation. However, the results are within as low as 2.73% from $\min\{\mathcal{C}_K^{\text{Hu}}, \mathcal{C}_K^{\text{CG}}\}$. This suggests that the heuristics used by Hu's and Coffman-Graham's scheduling algorithms are powerful and produce very good results for our setting. Based on an assumption that the results of our RL algorithms are approaching and converging to the optimal, we say that Hu's and Coffman-Graham's algorithms may produce results that is very close to optimal in our setting. Another conclusion from both tables is that RL tends to work better with large K. Again, note that the agent only learns for 500,000 iterations, which can take approximately one day for each parameter set, and we expect to achieve even less cost when learning longer.

We also provide a plot for the cost of best schedulings found during the learning process for the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}, K) = (239, 27.8, 17, 8)$ to illustrate the effectiveness of the updates in our experiments. It is clear that the offline update is more effective for our setting and can lead to results which is closer to optimal. Nonetheless, the difference in the results are around 1–2%.

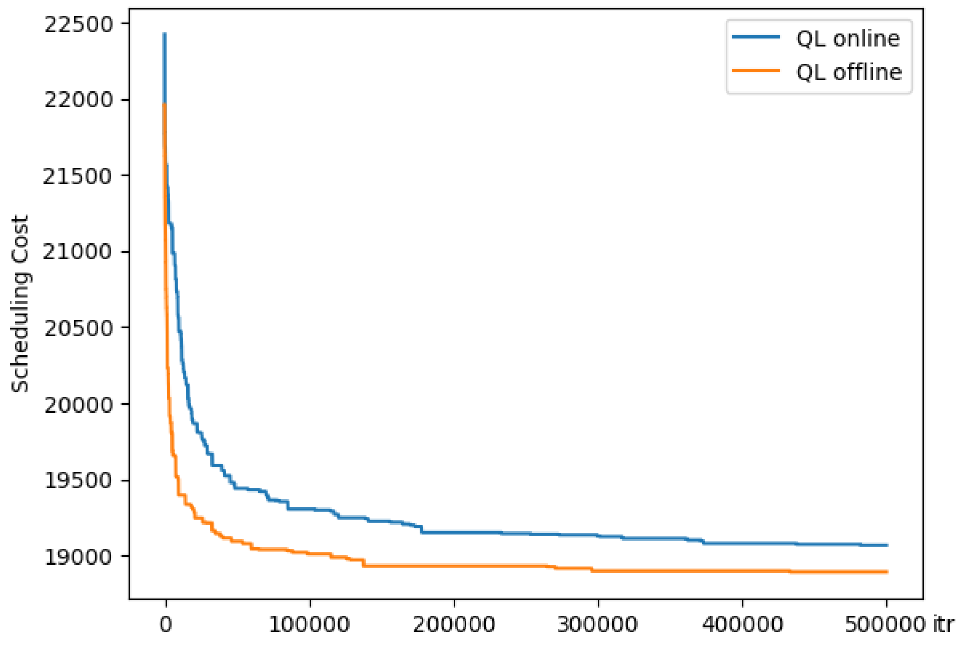


Figure 6.10: A plot for the cost of best scheduling found from RL experiments.

6.3.4 Optimality of PCS

As our reinforcement learning algorithms produce the results that are very close to those from Hu's and Coffman-Graham's algorithm, we conjecture that our PCS technique is nearly optimal, or even optimal, for strategy evaluation for a given strategy $\mathcal{S}$.

To argue about the optimality of PCS, we may have to exhaustively go through all (s, a) pairs in the MDP. Since the number of pairs can be large for a large strategy, we consider a small one when $e = 50$ (instead of 186 and 239 used elsewhere in the thesis). We construct a strategy under the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}, K) = (50, 27.8, 17, 8)$ using GA (shown below in Figure 6.11) and then perform an exhaustive (i.e., depth-first) search on the MDP to obtain an optimal scheduling for this strategy. As a result, an optimal scheduling has the same cost as produced by Hu's algorithm. Hence, PCS is indeed optimal for some strategies. Nevertheless, it is mentioned in Section 4.3 that there exists a strategy $\mathcal{S}'$ where $\mathcal{C}_K^{\text{Hu}}(\mathcal{S}') < \mathcal{C}_K^{\text{CG}}(\mathcal{S}')$ and a strategy $\mathcal{S}''$ where $\mathcal{C}_K^{\text{Hu}}(\mathcal{S}'') > \mathcal{C}_K^{\text{CG}}(\mathcal{S}'')$. Therefore, we believe that RL algorithms could be helpful in suggesting an optimality of PCS, and more works are still required to prove an optimality of PCS.

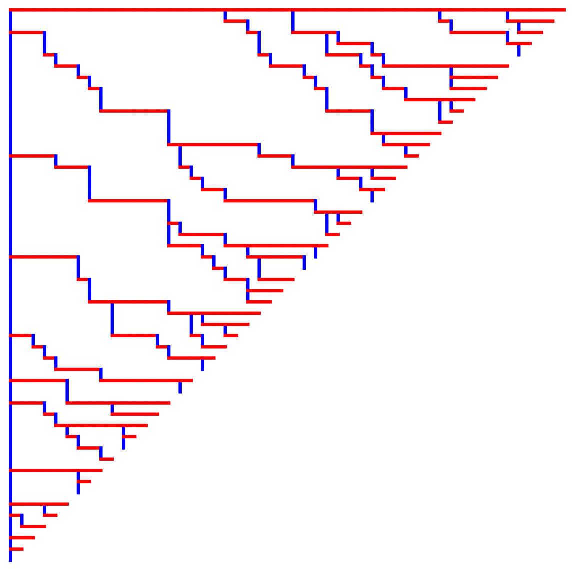


Figure 6.11: A strategy which has optimal scheduling under PCS for $(e, c_{\text{mul}}, c_{\text{iso}}, K) = (50, 27.8, 17, 8)$.

6.4 Discussion

From experimental results in Sections 6.2 and 6.3, we have seen that both learning-based optimizations work better when K gets larger. We think that there may be more solutions that produce less cost for large K and the probability that learning-based algorithms are able to find good solutions increases. In addition, GA works better for the case where $c_{\text{mul}} \approx c_{\text{iso}}$. Thus, it is interesting to look into the parameter set $(e, c_{\text{mul}}, c_{\text{iso}}, K)$ and investigate its implication on the search space and the effectiveness of learning-based algorithms.

Another interesting question is to find better instantiation of learning-based optimizations which is more effective. The algorithms' parameters (e.g., the size of GA population) and heuristics (e.g. RL exploration-exploitation method), sometimes called *hyperparameters* and *metaheuristics*, may be crucial in having effective learning. Hyperparameter optimization [33] is another problem in machine learning that aims to find a set of optimal hyperparameters for a learning algorithm. Using such optimization techniques may enable better results for our setting, but it is beyond the scope of this thesis.

6.5 Chapter Summary

In this chapter, we applied learning-based optimizations to attempt to solve the strategy construction and evaluation problems. We started by constructing low cost strategies using genetic algorithm. GA involves various mechanics, mimicking natural selection such as survival and reproduction, in order to search for solutions with great quality or fitness. To apply GA, we defined the representation of solutions as strings and some GA stochastic operations. As a result, GA was able to find low cost strategies for several parameter sets for up to 9.95%. One thing which influenced the effectiveness of the search was the representation of a solution.

Later, we utilized a reinforcement learning algorithm to construct a scheduling for a given strategy. An agent of RL learned through trial-and-error by interacting with the environment and built up its knowledge from its rewards. We first modeled our setting as a Markov decision process (MDP), providing several proofs regarding states and actions, and then applied Q-learning algorithm on the value function. Although our RL algorithms were not able to outperform Hu's and Coffman-Graham's scheduling algorithms, the experimental results supported that the traditional approximation algorithms for task scheduling are powerful. We concluded by showing that there is a strategy which has an optimal scheduling under PCS and discussing some possible improvements on using learning-based optimization with our setting.

Chapter 7

Conclusions and Future Works

7.1 Conclusions

This thesis is a study of speeding-up (i.e., reducing the latency of) the large smooth-degree isogeny computation at the strategy-level, by exploiting the rich parallelism available in multi-core and vectorized platforms. The study covered both theoretical improvements due to various—traditional and learning-based—algorithms and practical aspects regarding actual implementations such as precomputations and synchronizations.

Our first contribution presented in Chapter 4 is the precedence-constrained scheduling (PCS) technique [75] where strategies are transformed into task dependency graphs which are then utilized to construct schedulings. In previous works, the way a strategy is evaluated is based on a simple set of rules that can be considered suboptimal. In this thesis, we considered Hu's and Coffman-Graham's algorithms for task schedulings. Also, we formalized the problem as an integer linear program and combined optimal solutions for small strategies into larger strategies. As a result, the experiments showed that the cost of strategies can be reduced by up to 13.02%.

The second contribution detailed in Chapter 5 is two actual implementations (in C language) [76] of the large smooth-degree isogeny computation. The first implementation considered two-to-four-core processors. We utilized our PCS and some existing optimizations for the speed-up. To efficiently handle synchronization between cores, we carefully transformed the scheduling from PCS into OpenMP instructions with a barrier mechanism. The benchmarkings denoted a reduction of up to 14.36% in the execution time. The second implementation targeted two-to-four-core processors supporting AVX-512, the vectorization technology from Intel. As vectorization is somewhat different from multi-core, we required some modifications to the strategy construction and evaluation for such setting. We note that, to our best knowledge, previous implementations utilized either multi-core processors or vectorization technology, but not both. Thus, our work is the first work to do so. With various optimizations applied, we were able to achieve a reduction of up to 34.05% in the execution time.

The last contribution is two applications of learning-based optimizations to the problem of the large smooth-degree isogeny computation (Chapter 6). For the first problem of strategy construction under a given cost function, we considered the genetic algorithm to search for better strategies. The algorithm is probabilistic and involves the processes of mixing two strategies to create a new one and removing some strategies with low quality. To instantiate the algorithm, we provided details and gave some discussions regarding our design choices. The improvement from experimental results was as high as 9.95%. For the second problem of strategy evaluation for a given strategy, we considered the reinforcement learning algorithm to learn what action to perform at each step. The agent of the algorithm learns from its environment and explores states using value functions. Even though our reinforcement learning algorithms did not produce better experimental results compared to those of PCS, we used those results to argue the effectiveness of PCS, which is optimal for some strategies.

Table 7.1 below summarizes all techniques, in this thesis and previous works, regarding the computation of large smooth-degree isogeny.

Settings	**Source**	**Strategy Construction**	**Strategy Evaluation**
Single-Core	[30]	Optimal strategies	Sequential
Multi-Core	[42]	Optimal strategies under PCP	Per-Curve Parallel (PCP), Consecutive-Curve Parallel (CCP)
	Chapter 4	Integer Linear Programming (ILP)	Precedence-Constrained Scheduling (PCS)
	Chapter 5	Optimal strategies under modified PCP	Modified PCP, Modified PCS
	Chapter 6	Genetic Algorithm (GA)	Reinforcement Learning (RL)

Table 7.1: Summarization of speeding-up techniques for large smooth-degree isogeny computation.

7.2 Future Works

Although we have significantly improved the results from previous works, we believe that it is still possible to further speed-up the large smooth-degree isogeny computation. Below we list some directions for future work.

Proving optimality and lower bound. The optimality of strategy construction and evaluation techniques proposed in this thesis is yet to be proved. This may be due to the complexity of the scheduling problem and the isogeny computation itself. In addition, we were not able to find any non-trivial lower bound for the cost of the computation. Since the task dependency graphs have some nice structures (e.g., the in-degrees of all vertices are at most two), it might be interesting to analyze strategies and their task dependency graphs to a greater extent using graph theory knowledge in order to obtain insightful information about the optimality and lower bound.

Devising better algorithmic approaches. Apart from the techniques proposed in this thesis, there are various algorithms that can be used as alternatives. Some of those include scheduling algorithms mentioned in Chapter 4, and search and reinforcement learning algorithms mentioned in Chapter 6. The heuristics used by these algorithms may be more suitable to the problem and yield better experimental results. Moreover, the algorithms' hyperparameters and metaheuristics may be of importance in order to have practical algorithms that works well in our setting. The optimization problem for such aspects is another direction one could consider for the future work.

Furthermore, for the problem of constructing an optimal strategy and its evaluation when $c_{\text{mul}} = c_{\text{iso}}$ (Section 4.2), integer linear programming (ILP) is not the only available approach to solve for a solution. The problem can also be formulated as a Boolean satisfiability (SAT) problem [48] and a solution can be found using a solver. An interesting research question to be studied is to compare the sizes of instances of ILP and SAT for our strategy problem and their effectiveness in finding solutions.

Improving the cost model. We mentioned at the end of Chapter 5 that the cost model for the large smooth-degree isogeny computation is based solely on two parameters: the costs of computing point multiplications c_{mul} and isogeny evaluations c_{iso}. This approach is used in all existing works, including ours, but might not provide a close approximation for the actual execution time of implementations. It would be better if one can propose a new cost model that takes into account the costs of synchronization, memory access, etc. At present, we have no insights on how to do so and leave this as an open problem.

Considering other variants of the problem. In Chapter 4, we limited the scope of the task scheduling problem for this thesis to the case where all tasks are of unit-length, the number of cores is constant, all cores are identical, and preemption is not allowed. When some of these restrictions are relaxed, it may reflect some real-world applications (e.g., when the isogeny computation is done in parallel with other procedures of the cryptographic protocol), but the complexity of the problem may be more complicated. Also, another variant of the problem is to find the number of cores that one needs in order to finish the isogeny computation within the given time. Tackling these variants may possibly lead to some insights for the original problem.

Applying to other implementation settings. We considered software implementations which utilizes multi-core parallelism and vectorization for the isogeny computation in this thesis, but one can apply our proposed PCS to other implementation settings as well. As discussed in Chapter 5, there are various technologies available to choose from for the implementation (e.g., AVX2, ARM's SVE, Graphics Processing Unit (GPU), FPGA, etc.). Each has different characteristic and some may be more suitable for the isogeny computation than others. This is still an active area of research and we expect varieties of optimized implementations for the parallel isogeny computation from the research community.

Extending to other isogeny-based schemes. Lastly, we are looking forward to extending our techniques to other isogeny-based schemes such as CSIDH and SQISign. Since these two schemes are much newer than SIDH, there are fewer publications that report the amount of parallelism that can be applied to such schemes. SQISign may be closer to our setting of SIDH/SIKE, while the computation paradigm of CSIDH is different to some extent. Nonetheless, we are hopeful that one would achieve a similar extent of effectiveness when extending our techniques to both schemes. Some discussion and guidelines on this matter are provided in Chapter 5.

References

[1] Gora Adj, Jesús-Javier Chi-Domínguez, and Francisco Rodríguez-Henríquez. Karatsuba-based square-root Vélu's formulas applied to two isogeny-based protocols. *Journal of Cryptographic Engineering*, pages 1–18, 2022.

[2] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[3] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[4] Anne Auger and Benjamin Doerr, editors. *Theory of Randomized Search Heuristics: Foundations and Recent Developments*, volume 1 of *Series on Theoretical Computer Science*. World Scientific, 2011.

[5] Reza Azarderakhsh, Elena Bakos Lang, David Jao, and Brian Koziel. Edsidh: Supersingular isogeny diffie-hellman key exchange on edwards curves. In Anupam Chattopadhyay, Chester Rebeiro, and Yuval Yarom, editors, *Security, Privacy, and Applied Cryptography Engineering - 8th International Conference, SPACE 2018, Kanpur, India, December 15-19, 2018, Proceedings*, volume 11348 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2018.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

[7] Richard Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[8] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *Eur. J. Oper. Res.*, 290(2):405–421, 2021.

[9] Daniel J. Bernstein. *Introduction to post-quantum cryptography*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[10] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. Faster computation of isogenies of large prime degree. *Open Book Series*, 4(1):39–55, 2020.

[11] Daniel J. Bernstein and Tanja Lange. Explicit-formulas database: Genus-1 curves over large-characteristic fields. [Online; accessed 16-April-2023]. https://hyperelliptic.org/EFD/g1p/index.html.

[12] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007.

[13] Reinier Bröker. Constructing supersingular elliptic curves. *J. Comb. Number Theory*, 1(3):269–273, 2009.

[14] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 423–447. Springer, 2023.

[15] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: an efficient post-quantum commutative group action. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 395–427. Springer, 2018.

[16] Daniel Cervantes-Vázquez, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. Parallel strategies for SIDH: toward computing SIDH twice as fast. *IEEE Trans. Computers*, 71(6):1249–1260, 2022.

[17] Denis Xavier Charles, Kristin E. Lauter, and Eyal Z. Goren. Cryptographic hash functions from expander graphs. *J. Cryptol.*, 22(1):93–113, 2009.

[18] Krishnendu Chatterjee and Thomas A. Henzinger. Value iteration. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 107–138. Springer, 2008.

[19] Jorge Chávez-Saab, Francisco Rodríguez-Henríquez, and Mehdi Tibouchi. Verifiable isogeny walks: Towards an isogeny-based postquantum VDF. In Riham AlTawy and Andreas Hülsing, editors, *Selected Areas in Cryptography - 28th International Conference, SAC 2021, Virtual Event, September 29 - October 1, 2021, Revised Selected Papers*, volume 13203 of *Lecture Notes in Computer Science*, pages 441–460. Springer, 2021.

[20] Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Peter Y. A. Ryan. Highly vectorized SIKE for AVX-512. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):41–68, 2022.

[21] Jesús-Javier Chi-Domínguez and Francisco Rodríguez-Henríquez. Optimal strategies for CSIDH. *Adv. Math. Commun.*, 16(2):383–411, 2022.

[22] Fabián A. Chudak and David B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *J. Algorithms*, 30(2):323–343, 1999.

[23] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny diffie-hellman. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 572–601. Springer, 2016.

[24] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic - the case of large characteristic fields. *J. Cryptogr. Eng.*, 8(3):227–240, 2018.

[25] Jean-Marc Couveignes. Hard homogeneous spaces. Cryptology ePrint Archive, Paper 2006/291, 2006. https://eprint.iacr.org/2006/291.

[26] Javad Doliskani, Geovandro C. C. F. Pereira, and Paulo S. L. M. Barreto. Faster cryptographic hash function from supersingular isogeny graphs. Cryptology ePrint Archive, Paper 2017/1202, 2017. https://eprint.iacr.org/2017/1202.

[27] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 1999, Washington, DC, USA July 6-9, 1999*, pages 1470–1477. IEEE, 1999.

[28] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet

Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nat.*, 610(7930):47–53, 2022.

[29] Armando Faz-Hernández, Julio César López-Hernández, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol. *IEEE Trans. Computers*, 67(11):1622–1636, 2018.

[30] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Math. Cryptol.*, 8(3):209–247, 2014.

[31] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 64–93. Springer, 2020.

[32] Luca De Feo, Simon Masson, Christophe Petit, and Antonio Sanso. Verifiable delay functions from supersingular isogenies and pairings. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 248–277. Springer, 2019.

[33] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Automated Machine Learning - Methods, Systems, Challenges*, The Springer Series on Challenges in Machine Learning, pages 3–33. Springer, 2019.

[34] Devdatta Gangal and Abhiram G. Ranade. Precedence constrained scheduling in $(2 - \frac{7}{3p+1})$ optimal. *J. Comput. Syst. Sci.*, 74(7):1139–1146, 2008.

[35] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[36] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning.* Addison-Wesley, 1989.

[37] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell system technical journal*, 45(9):1563–1581, 1966.

[38] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[39] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[40] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[41] Te C. Hu. Parallel sequencing and assembly line problems. *Operations research*, 9(6):841–848, 1961.

[42] Aaron Hutchinson and Koray Karabina. Constructing canonical strategies for parallel implementation of isogeny based cryptography. In Debrup Chakraborty and Tetsu Iwata, editors, *Progress in Cryptology - INDOCRYPT 2018 - 19th International Conference on Cryptology in India, New Delhi, India, December 9-12, 2018, Proceedings*, volume 11356 of *Lecture Notes in Computer Science*, pages 169–189. Springer, 2018.

[43] Aaron Hutchinson, Jason T. LeGrow, Brian Koziel, and Reza Azarderakhsh. Further optimizations of CSIDH: A systematic approach to efficient strategies, permutations, and bound vectors. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security - 18th International Conference, ACNS 2020, Rome, Italy, October 19-22, 2020, Proceedings, Part I*, volume 12146 of *Lecture Notes in Computer Science*, pages 481–501. Springer, 2020.

[44] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchinson, Amir Jalali, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular Isogeny Key Encapsulation, 2020. https://sike.org/files/SIDH-spec.pdf.

[45] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011.

[46] András Joó, Anikó Ekárt, and Juan Pablo Neirotti. Genetic algorithms for discovery of matrix multiplication methods. *IEEE Trans. Evol. Comput.*, 16(5):749–751, 2012.

[47] Edward G. Coffman Jr. and Ronald L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[48] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[49] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of International Conference on Neural Networks (ICNN'95), Perth, WA, Australia, November 27 - December 1, 1995*, pages 1942–1948. IEEE, 1995.

[50] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[51] A Harry Klopf. *Brain function and adaptive systems: a heterostatic theory.* Air Force Cambridge Research Laboratories, Air Force Systems Command, United States Air Force, 1972.

[52] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[53] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[54] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[55] David Russell Kohel. *Endomorphism rings of elliptic curves over finite fields.* PhD thesis, University of California at Berkeley, 1996.

[56] Dusan Kostic and Shay Gueron. Using the new VPMADD instructions for the new post quantum key encapsulation mechanism SIKE. In Naofumi Takagi, Sylvie Boldo, and Martin Langhammer, editors, *26th IEEE Symposium on Computer Arithmetic, ARITH 2019, Kyoto, Japan, June 10-12, 2019*, pages 215–218. IEEE, 2019.

[57] Brian Koziel, A.-Bon E. Ackie, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari Kermani. SIKE'd up: Fast hardware architectures for Supersingular Isogeny Key Encapsulation. *IEEE Trans. Circuits Syst.*, 67-I(12):4842–4854, 2020.

[58] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Fast hardware architectures for Supersingular Isogeny Diffie-Hellman key exchange on FPGA. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *Progress in Cryptology - INDOCRYPT 2016 - 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings*, volume 10095 of *Lecture Notes in Computer Science*, pages 191–206, 2016.

[59] Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari Kermani. NEON-SIDH: efficient implementation of supersingular isogeny diffie-hellman key exchange protocol on ARM. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, pages 88–103, 2016.

[60] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. CryptOpt: Verified compilation with random program search for cryptographic primitives. *CoRR*, abs/2211.10665, 2022.

[61] Shui Lam and Ravi Sethi. Worst case analysis of two scheduling algorithms. *SIAM J. Comput.*, 6(3):518–536, 1977.

[62] Elaine Levey and Thomas Rothvoß. A $(1+\varepsilon)$-approximation for makespan scheduling with precedence constraints using LP hierarchies. *SIAM J. Comput.*, 50(3), 2021.

[63] Patrick Longa. Efficient algorithms for large prime characteristic fields and their application to bilinear pairings. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):445–472, 2023.

[64] Luciano Maino, Chloe Martindale, Lorenz Panny, Giacomo Pope, and Benjamin Wesolowski. A direct key recovery attack on SIDH. In Carmit Hazay and Martijn

Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 448–471. Springer, 2023.

[65] Maryam Karimi Mamaghan, Mehrdad Mohammadi, Patrick Meyer, Amir Mohammad Karimi-Mamaghan, and El-Ghazali Talbi. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *Eur. J. Oper. Res.*, 296(2):393–422, 2022.

[66] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Comput. Oper. Res.*, 134:105400, 2021.

[67] Donald Michie and Roger A Chambers. Boxes: An experiment in adaptive control. *Machine intelligence*, 2(2):137–152, 1968.

[68] Microsoft Research. SIDH library, 2017. https://github.com/microsoft/PQCrypto-SIDH.

[69] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1985.

[70] Tom M. Mitchell. *Machine learning, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[71] Peter L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[72] National Institute of Standards and Technology. Report on post-quantum cryptography. https://nvlpubs.nist.gov/nistpubs/ir/2016/nist.ir.8105.pdf.

[73] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[74] Kittiphon Phalakarn, Kittiphop Phalakarn, and Vorapong Suppakitpaisarn. Optimal representation for right-to-left parallel scalar and multi-scalar point multiplication. *Int. J. Netw. Comput.*, 8(2):166–185, 2018.

[75] Kittiphon Phalakarn, Vorapong Suppakitpaisarn, and M. Anwar Hasan. Speeding-up parallel computation of large smooth-degree isogeny using precedence-constrained scheduling. In Khoa Nguyen, Guomin Yang, Fuchun Guo, and Willy Susilo, editors, *Information Security and Privacy - 27th Australasian Conference, ACISP 2022, Wollongong, NSW, Australia, November 28-30, 2022, Proceedings*, volume 13494 of *Lecture Notes in Computer Science*, pages 309–331. Springer, 2022.

[76] Kittiphon Phalakarn, Vorapong Suppakitpaisarn, Francisco Rodríguez-Henríquez, and M. Anwar Hasan. Vectorized and parallel computation of large smooth-degree isogenies using precedence-constrained scheduling. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):246–269, 2023.

[77] Kittiphon Phalakarn, Toru Takisaka, Thomas Haas, and Ichiro Hasuo. Widest paths and global propagation in bounded value iteration for stochastic games. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 349–371. Springer, 2020.

[78] Kittiphop Phalakarn, Kittiphon Phalakarn, and Vorapong Suppakitpaisarn. Parallelized side-channel attack resisted scalar multiplication using q-based addition-subtraction k-chains. In *Fourth International Symposium on Computing and Networking, CANDAR 2016, Hiroshima, Japan, November 22-25, 2016*, pages 140–146. IEEE Computer Society, 2016.

[79] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[80] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527 – 535, 1952.

[81] Damien Robert. Breaking SIDH in polynomial time. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 472–503. Springer, 2023.

[82] Alexander Rostovtsev and Anton Stolbunov. Public-key cryptosystem based on isogenies. Cryptology ePrint Archive, Paper 2006/145, 2006. https://eprint.iacr.org/2006/145.

[83] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, 1959.

[84] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):1–20, 2018.

[85] Claude E. Shannon. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[86] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[87] Joseph H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate texts in mathematics*. Springer, 1986.

[88] Anton Stolbunov. Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves. *Adv. Math. Commun.*, 4(2):215–235, 2010.

[89] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 1998.

[90] M. A. L. Thathachar and P. Shanti Sastry. A new approach to the design of reinforcement schemes for learning automata. *IEEE Trans. Syst. Man Cybern.*, 15(1):168–175, 1985.

[91] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 1933.

[92] Jeffrey D. Ullman. NP-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, 1975.

[93] Jacques Vélu. Isogénies entre courbes elliptiques. *CR Acad. Sci. Paris Sér. AB*, 273(A238-A241):5, 1971.

[94] Lawrence C. Washington. *Elliptic curves: number theory and cryptography.* CRC press, 2008.

[95] Christopher J. C. H. Watkins and Peter Dayan. Technical note q-learning. *Mach. Learn.*, 8:279–292, 1992.

APPENDICES

Appendix A

SIDH and SIKE

We provide descriptions of the Supersingular Isogeny Diffie-Hellman (SIDH) key exchange [45] and the Supersingular Isogeny Key Encapsulation (SIKE) mechanism [44] as references for our implementations. It should be noted that both cryptosystems are totally insecure and should not be used.

A.1 SIDH

Jao and De Feo [45] in 2011 proposed a key exchange protocol called Supersingular Isogeny Diffie-Hellman (SIDH). We briefly describe the protocol as follows.

Setup: Alice and Bob agree on the following set of public parameters:

- a prime p of the form $\ell_A^{e_A}\ell_B^{e_B} \cdot f \pm 1$ where ℓ_A, ℓ_B are small primes, e_A, e_B are exponents giving $\ell_A^{e_A} \approx \ell_B^{e_B}$, and f is a cofactor,
- a supersingular elliptic curve E_0 over $\mathbb{F}_{p^2}$ with $\#E_0(\mathbb{F}_{p^2}) = (\ell_A^{e_A}\ell_B^{e_B} \cdot f)^2$, and
- bases $\{P_A, Q_A\}$ of $E_0[\ell_A^{e_A}]$ and $\{P_B, Q_B\}$ of $E_0[\ell_B^{e_B}]$.
- For Montgomery's curve, two points $P_A - Q_A$ and $P_B - Q_B$ may be given for point pseudo-addition.

Key Exchange:

1. Alice randomly chooses $m_A \in \mathbb{Z}_{\ell_A^{e_A}}$. She computes an isogeny $\phi_A : E_0 \to E_A$ with kernel $\langle R_A \rangle$ where $R_A = P_A + [m_A]Q_A$, and then sends $E_A, \phi_A(P_B), \phi_A(Q_B)$ to Bob.
   - For Montgomery curve, Alice also sends $\phi_A(P_B - Q_B)$ for point pseudo-addition.
2. Similarly, Bob randomly chooses $m_B \in \mathbb{Z}_{\ell_B^{e_B}}$. He computes an isogeny $\phi_B : E_0 \to E_B$ with kernel $\langle R_B \rangle$ where $R_B = P_B + [m_B]Q_B$, and sends $E_B, \phi_B(P_A), \phi_B(Q_A)$ to Alice.
   - For Montgomery curve, Bob also sends $\phi_B(P_A - Q_A)$ for point pseudo-addition.
3. Upon receiving $E_B, \phi_B(P_A), \phi_B(Q_A)$ from Bob, Alice computes an isogeny $\phi'_A : E_B \to E_{AB}$ with kernel $\langle R'_A \rangle$ where $R'_A = \phi_B(P_A) + [m_A]\phi_B(Q_A)$.
4. Similarly, upon receiving $E_A, \phi_A(P_B), \phi_A(Q_B)$ from Alice, Bob computes an isogeny $\phi'_B : E_A \to E_{BA}$ with kernel $\langle R'_B \rangle$ where $R'_B = \phi_A(P_B) + [m_B]\phi_A(Q_B)$.
5. The shared secret of Alice and Bob is the j-invariant of the resulting elliptic curves: $j(E_{AB}) = j(E_{BA})$.
   - For simplified Weierstrass curve $E : y^2 = x^3 + ax + b$, $j(E) = 1728\frac{4a^3}{4a^3+27b^2}$.
   - For Montgomery curve $E : by^2 = x^3 + ax^2 + x$, $j(E) = 256\frac{(a^2-3)^3}{a^2-4}$.

$$\begin{tikzcd}
& E_A \arrow[dr, "\phi'_B"] & \\
E_0 \arrow[ur, "\phi_A"] \arrow[dr, "\phi_B"'] & & E_{AB} \cong E_{BA} \\
& E_B \arrow[ur, "\phi'_A"'] &
\end{tikzcd}$$

Figure A.1: The commutative diagram of SIDH protocol.

A.2 SIKE

The Supersingular Isogeny Key Encapsulation (SIKE) mechanism is proposed by [44] as an isogeny-based key encapsulation suite. It is based on SIDH and contains two algorithms: CPA-secure public key encryption (PKE) and CCA-secure key encapsulation mechanism (KEM). The protocol specifies some public parameters (depending on the security level) as follows:

- a prime $p = 2^{e_2}3^{e_3} - 1$ for some positive integers e_2, e_3,
- a starting supersingular elliptic curve E_0 over $\mathbb{F}_{p^2}$, where $\#E_0(\mathbb{F}_{p^2}) = (2^{e_2}3^{e_3})^2$,
- bases $\{P_2, Q_2\}$ of $E_0[2^{e_2}]$ and $\{P_3, Q_3\}$ of $E_0[3^{e_3}]$, and
- two points $P_2 - Q_2$ and $P_3 - Q_3$ for Montgomery curve's point pseudo-addition.

SIKE proposes four public parameter sets for different security levels: SIKEp434, SIKEp503, SIKEp610, and SIKEp751, whose names indicate the size of the prime p in bits. These parameter sets are shown in Table A.1, while Table A.2 indicates security strength categories set by NIST [73]. The details of SIKE PKE and KEM can be found in Algorithms A.1 and A.2, respectively, where $(\phi, E') \leftarrow \mathsf{CompIsogeny}(E, R)$ denotes the computation of isogeny $\phi : E \rightarrow E' = E/\langle R\rangle$ and H denotes a secure hash function. We note that, while we use elliptic curve points in the algorithms, the SIKE specification and many implementations use x-coordinates of those points in the computation.

	Target NIST Security Strength	e_2	e_3	p
SIKEp434	1	216	137	$2^{216}3^{137} - 1$
SIKEp503	2	250	159	$2^{250}3^{159} - 1$
SIKEp610	3	305	192	$2^{305}3^{192} - 1$
SIKEp751	5	372	239	$2^{372}3^{239} - 1$

Table A.1: SIKE public parameter sets.

Level	Security Description
1	At least as hard to break as AES128 (exhaustive key search)
2	At least as hard to break as SHA256 (collision search)
3	At least as hard to break as AES192 (exhaustive key search)
4	At least as hard to break as SHA384 (collision search)
5	At least as hard to break as AES256 (exhaustive key search)

Table A.2: NIST security strength categories.

Algorithm A.1: SIKE PKE.

```
Gen() :
    $\mathrm{sk}_3 \leftarrow_\$ \{0, ..., 3^{e_3} - 1\}$
    $R_3 \leftarrow P_3 + [\mathrm{sk}_3]Q_3$
    $(\phi_3, E_3) \leftarrow \mathsf{CompIsogeny}_3(E_0, R_3)$
    $\mathrm{pk}_3 \leftarrow (\phi_3(P_2), \phi_3(Q_2))$
    return $(\mathrm{pk}_3, \mathrm{sk}_3)$
Enc($\mathrm{pk}_3, m, \mathrm{sk}_2$) :
    $R_2 \leftarrow P_2 + [\mathrm{sk}_2]Q_2$
    $(\phi_2, E_2) \leftarrow \mathsf{CompIsogeny}_2(E_0, R_2)$
    $c_0 \leftarrow (\phi_2(P_3), \phi_2(Q_3))$
    $E_3 \leftarrow \mathsf{GetCurve}(\mathrm{pk}_3)$
    $R'_2 \leftarrow \phi_3(P_2) + [\mathrm{sk}_2]\phi_3(Q_2)$
    $(\phi'_2, E_{32}) \leftarrow \mathsf{CompIsogeny}_2(E_3, R'_2)$
    $c_1 \leftarrow H(j(E_{32})) \oplus m$
    return $(c_0, c_1)$
Dec($\mathrm{sk}_3, (c_0, c_1)$) :
    $E_2 \leftarrow \mathsf{GetCurve}(c_0)$
    $R'_3 \leftarrow \phi_2(P_3) + [\mathrm{sk}_3]\phi_2(Q_3)$
    $(\phi'_3, E_{23}) \leftarrow \mathsf{CompIsogeny}_3(E_2, R'_3)$
    $m \leftarrow H(j(E_{23})) \oplus c_1$
    return $m$
```

Algorithm A.2: SIKE KEM.

```
KeyGen() :
    $(\mathrm{pk}_3, \mathrm{sk}_3) \leftarrow \mathsf{Gen}()$
    $s \leftarrow_\$ \{0, 1\}^n$
    return $(s, \mathrm{sk}_3, \mathrm{pk}_3)$
Encaps($\mathrm{pk}_3$) :
    $m \leftarrow_\$ \{0, 1\}^n$
    $\mathrm{sk}_2 \leftarrow H(m \parallel \mathrm{pk}_3)$
    $(c_0, c_1) \leftarrow \mathsf{Enc}(\mathrm{pk}_3, m, \mathrm{sk}_2)$
    $K \leftarrow H(m \parallel (c_0, c_1))$
    return $((c_0, c_1), K)$
Decaps($s, \mathrm{sk}_3, \mathrm{pk}_3, (c_0, c_1)$) :
    $m^* \leftarrow \mathsf{Dec}(\mathrm{sk}_3, (c_0, c_1))$
    $\mathrm{sk}^*_2 \leftarrow H(m^* \parallel \mathrm{pk}_3)$
    $R^*_2 \leftarrow P_2 + [\mathrm{sk}^*_2]Q_2$
    $(\phi^*_2, E^*_2) \leftarrow \mathsf{CompIsogeny}_2(E_0, R^*_2)$
    $c^*_0 \leftarrow (\phi^*_2(P_3), \phi^*_2(Q_3))$
    if $c^*_0 = c_0$ then
        $K \leftarrow H(m^* \parallel (c_0, c_1))$
    else
        $K \leftarrow H(s \parallel (c_0, c_1))$
    return $K$
```