Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Erik Kähkönen

Implementing Delegable Inference for Graphical Models

Master's Thesis
Espoo, December 30, 2022

Supervisor: Professor Petteri Kaski
Advisor: Professor Petteri Kaski

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Erik Kähkönen		
Title:	Implementing Delegable Inference for Graphical Models		
Date:	December 30, 2022	**Pages:**	61
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Petteri Kaski		
Advisor:	Professor Petteri Kaski		

Factor graphs are a versatile tool with a wide range of applications from multilinear algorithm design to the study quantum many-body systems. However, exact inference in factor graphs is #P hard in general. This motivates the development of inference algorithms which can be robustly delegated to extensive and possibly unreliable computational infrastructure. Building on non-interactive proof systems and error-correcting polynomial codes, Karimi *et al.* [18] recently presented such a framework for delegable exact inference for factor graphs. The framework enables a delegator to recover and probabilistically verify the results of delegated inference, with tolerance for errors in the delegated computations. The main contribution of this thesis is a library implementing Karimi *et al.*'s framework. We review some of the theoretical foundations upon which Karimi *et al.*'s framework is built, and outline the key aspects of our implementation. The full source code of the library is available online [17]. We evaluate our library on the tasks of matrix multiplication and computing matrix permanents, by reducing them to the task of inference in factor graphs. We show empirically that the desirable theoretical properties of Karimi *et al.*'s framework hold; in particular, the complexity of verification was found to be negligible relative to the complexity of performing inference. However, the library's overall performance was found to fall short of being competitive. Fortunately, the inference framework admits massive parallelism beyond what our implementation exploited, implying that large speedups are possible. We identify performance bottlenecks and provide suggestions for how to extend the library to improve its performance.

Keywords:	graphical models, exact inference, noninteractive probabilistic proof systems, algorithm engineering
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Master's Programme in Computer, Communication and Information Sciences

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Erik Kähkönen		
Työn nimi:	Delegoitavan päättelyn toteutus graafisille malleille		
Päiväys:	30. joulukuuta 2022	**Sivumäärä:**	61
Pääaine:	Computer Science	**Koodi:**	SCI3042
Valvoja:	Professori Petteri Kaski		
Ohjaaja:	Professori Petteri Kaski		

Faktorigraafit ovat monipuolinen työkalu, jolla on laajasti sovelluksia monilineaarisesta algoritmisuunnittelusta kvanttijärjestelmien tutkimiseen. Kuitenkin tarkka päättely faktorigraafeissa on yleisesti #P-vaikeaa. Tämä motivoi kehittämään päättelyalgoritmeja, jotka voidaan delegoida laajamittaisen ja mahdollisesti epäluotettavan laskentainfrastruktuurin suoritettavaksi. Pohjautuen ei-vuorovaikutteisiin todistusjärjestelmiin ja virheenkorjauspolynomikoodeihin, Karimi *et al.* [18] hiljattain esittivät kyseisenlaisen algoritmin delegoitaville tarkalle päättelylle faktorigraafeille. Algoritmi mahdollistaa päättelytehtävän delegoinnin ja päättelyn tuloksen probabilistisen tarkistamisen, tarjoten myös toleranssia virheille delegoidussa päättelyssä. Tämän opinnäytetyön pääkontribuutio on Karimi *et al.*:n algoritmin toteuttava kirjasto. Käymme läpi joitain teoreettisia perusteita, joille Karimi *et al.*:n algoritmi on rakennettu, ja annamme yhteenvedon toteutuksen keskeisistä kohdista. Kirjaston lähdekoodi on saatavilla verkossa [17]. Arvioimme kirjastomme toimivuutta kahdella laskennallisella tehtävällä—matriisikertolaskulla sekä matriisipermanentin laskemisella—redusoimalla ne päättelyksi faktorigraafeissa. Todennamme Karimi *et al.*:n algoritmin toivottavat teoreettiset ominaisuudet empiirisesti; erityisesti delegoidun tuloksen *tarkistamisen* kompleksisuuden havaittiin olevan mitätön verrattuna itse päätelyn *suorittamisen* kompleksisuuteen. Kirjaston yleisen suorituskyvyn todettiin kuitenkin olevan vähemmän kuin kilpailukykyinen. Karimi *et al.*:n algoritmi kuitenkin mahdollistaisi korkea-asteisen rinnakkaisuuden, jota toteutuksemme ei hyödyntänyt; toteutuksesta voisi siis tehdä paljon nopeamman. Identifioimme suorituskyvyn pullonkaulat ja annamme ehdotuksia, kuinka kirjastoa voisi kehittää sen suorituskyvyn parantamiseksi.

Asiasanat:	graafiset mallit, tarkka päättely, ei-vuorovaikutteiset probabilistiset todistusjärjestelmät, algoritmisuunnittelu
Kieli:	Englanti

Acknowledgements

First and foremost, I am grateful to my supervisor, Professor Petteri Kaski, for providing me with the opportunity to work on this project, and for his continuous support throughout it. His prodigious knowledge and acumen were often instrumental in guiding the project, and I gained many insights from his feedback and advice; without them this thesis would be of notably poorer quality. I would also like to thank my family for their support. Finally, but not unimportantly, I would like to thank the taxpayers of Finland, particularly for the educational system which they fund.

Espoo, December 30, 2022

Erik Kähkönen

Contents

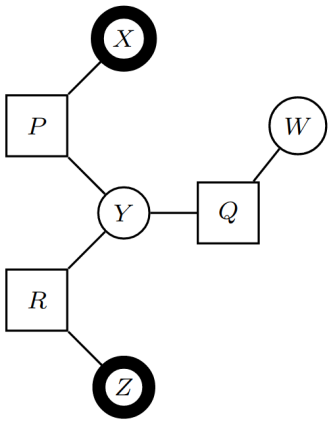
Chapter 1

Introduction

Graphical models are mathematical tools that describe how complex structures factorize into simpler structures. A *factor graph* is a type of graphical model that consists of a bipartite graph of factor-nodes and variable-nodes. While conceptually simple, factor graphs are very general, in that they can represent diverse things of interest and they have a very wide variety of applications: Many important algorithms such as the discrete Fourier transform comprise multilinear maps, and are expressible as exact inference in factor graphs [2]; factor graphs have been used in error-correcting codes at least since Gallager's paper on low-density parity-check codes [9]; under the guise of tensor networks, factor graphs have many applications in theoretical physics [26]; and, perhaps most famously, factor graphs have diverse applications as probabilistic graphical models [22]. This thesis is concerned with those applications of factor graphs where *exact* inference[1] is desirable.

A key operation for performing exact inference is that of *factor contraction*. To illustrate how factor contraction works, consider the factor graph G in Figure 1.1: The circular vertices—X, Y, Z, and W—are binary variables, and the square vertices are factors. Each factor is an array with one dimension per adjacent variable. The variables with bold peripheries form the *boundary* of G. Having set the boundary to be $B = \{X, Z\}$, contracting all factors in G amounts to computing the marginal of variables X and Z. To contract a pair of factors, say P and Q, we replace them by a new factor $PQ(x, y) = \sum_w P(x, y)Q(y, w)$, obtained by taking the product of the elements of P and Q along corresponding dimensions and summing over all non-boundary variables that are adjacent only to P or Q. Similarly, to contract the resulting factor PQ with R, we replace them by PQR, given by

[1]Note: The distinction here is between *exact* and *approximate* inference. *Probabilistic* inference—for example computing conditional marginal distributions—can be either exact or approximate.



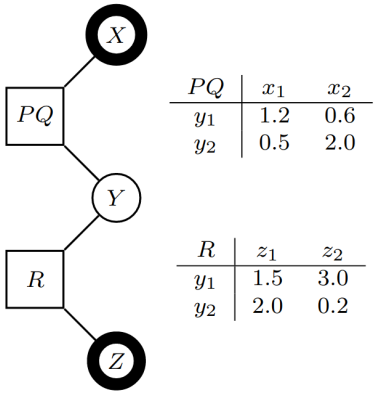
P	x_1	x_2
y_1	1.0	0.5
y_2	0.5	2.0

Q	w_1	w_2
y_1	0.5	0.7
y_2	0.6	0.4

R	z_1	z_2
y_1	1.5	3.0
y_2	2.0	0.2

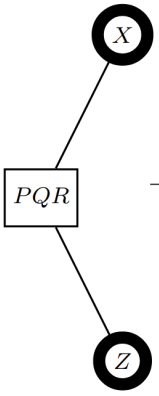
Figure 1.1: Simple factor graph G with three factors and four binary variables. The boundary comprises variables X and Z.

$PQR(x, z) = \sum_y PQ(x, y)R(y, z)$.



PQ	x_1	x_2
y_1	1.2	0.6
y_2	0.5	2.0

R	z_1	z_2
y_1	1.5	3.0
y_2	2.0	0.2



PQR	x_1	x_2
y_1	2.8	3.7
y_2	4.9	2.2

Figure 1.2: Left: the factor graph obtained by contracting P and Q to PQ in G. Right: the final graph obtained by contracting PQ and R to PQR.

In general, the computational complexity of contracting factor graphs depends strongly on the order in which factors are contracted. A given factor graph may admit a contraction order with low-order polynomial total cost, and yet contracting the same graph may be exponentially expensive with an inefficient contraction order. And in general, as we will see in Chapter 2, factor graph contraction is #P-hard[2]. Consequently, realistic applications of exact inference in factor graphs are liable to require large-scale and massively parallel computation. While modern compute hardware is in general highly

[2]The complexity class #P consists of the set of *counting* problems corresponding to *decision* problems in NP.

reliable, hardware errors do occur, and become more likely as the scale of a computation increases; for a survey of hardware reliability in the context of large-scale compute infrastructure, see *e.g.* [37]. Thus the requirement for large-scale infrastructure motivates the development of methods for reliably *delegating* computations; that is, outsourcing the computations to large-scale hardware—possibly rented from unreliable counterparties—in a way that enables efficient verification of the result and correction of possible errors.

Appositely, Karimi, Kaski, and Koivisto [18] developed a framework for highly parallelizable exact inference in factor graphs which provides error-correction and fast probabilistic verification of the result. The framework constructs a polynomial extension of the contracted factor graph—called a *proof polynomial*—and an algorithm for evaluating that proof polynomial without access to its coefficients. Using a toolbox of near-linear-time algorithms for working with polynomials [36], the proof's coefficients can be decoded [10] from the evaluations, even in the presence of errors. The proof's coefficients can then be used both to quickly recover the results of inference, and to perform polynomial identity testing, providing strong probabilistic verification of the result. To our knowledge, there does not exist any other framework or algorithm for error-correcting and verifiable exact inference in graphical models.

The goal of this thesis was to take Karimi *et al.*'s delegable inference framework from theory to practical reality, by implementing it such that

(i) the implementation enables empirical testing of the framework's theoretical properties,

(ii) and the implementation is efficient, or at least easily extensible to be efficient.

To that end, we implemented the framework as a C++ library, and made the source code available online [17]. The library provides facilities for efficiently working with factor graphs over finite fields, and—making use of a library written by Kaski [19]—implements the algorithms for evaluating, decoding, and verifying proof polynomials, which together constitute Karimi *et al.*' framework.

Scope and limitations. Due mainly to a shortage of time, we unfortunately did not write an implementation that exploits parallelism; instead our implementation only runs on CPU and is single-threaded. It is also important to note that the algorithm implemented in this thesis performs *exact* inference in factor graphs. In principle, exact inference could be used in almost any application of factor graphs; however, in many applications, exact

inference is intractable in practice, and approximate inference—for example using message-passing algorithms—may be preferred instead.

Structure of this thesis. We begin in Chapter 2 with a brief overview of graphical models, focusing on factor graphs and the inference problem in factor graphs. Chapter 3 reviews the theory underlying the delegable inference framework. Chapter 4 summarizes key elements of a toolbox of fast algorithms for working with polynomials, which are indispensable for implementing the framework. We summarize some salient aspects of our implementation in Chapter 5, and present the results of experiments done with said implementation in Chapter 6. Conclusions and a brief discussion of possible future work are given in Chapter 7.

Chapter 2

Graphical models

Graphical models are versatile tools that find application in many different domains. Consequently there exist many different formulations of graphical models, and the literature contains a diversity terminology for them. The delegable inference algorithm studied in this thesis builds on the *factor graph* formulation of graphical models. In this chapter, we give a very brief overview of graphical models in general, then review factor graphs in particular, and define and analyse the *inference problem* for factor graphs.

Typically, a graphical model consists of a graph, a set of variables associated with the vertices of that graph, and a set of functions over those variables. The high-level motivation for graphical models is that they provide a concise yet very general way to describe how a function factorizes into simpler functions. This enables the design of efficient and general algorithms for computing various properties of a wide variety of functions. The separation of description and algorithm also provides modularity between the two, enabling the development of general algorithms independently of the specifics of any single model. As a concrete example, in the context of *probabilistic* graphical models (PGMs), a graph represents a set of probability distributions; the graph captures how the distributions factorize based on independences between sets of variables.

To illustrate how graphical models can be useful, consider the example in Figure 2.1 of a factor graph representing a simple probability distribution. The joint distribution is of form $p(X_1, X_2, \ldots, X_n) = p(X_1) \prod_{j=2}^{n} p(X_j|X_{j-1})$,

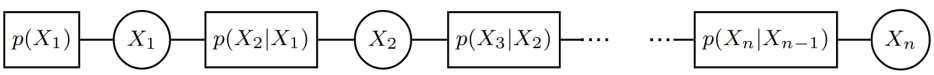


Figure 2.1: A factor graph representation of a probability distribution that factorizes into a chain.

and thus describes a causal chain. Suppose we wanted to infer the marginal $p(X_i)$ of variable X_i. In the absence of information about how the joint factorizes, we would have to compute $p(X_i)$ as[1]

$$p(X_i) \;=\; \sum_{x_1} \dots \sum_{x_{i-1}} \sum_{x_{i+1}} \dots \sum_{x_n} p(x_1, x_2, \dots, X_i, \dots, x_n) \;,$$

which would induce cost exponential in the number of variables. However, as the graph in Figure 2.1 makes evident, there is a cheaper alternative, namely computing

$$p(X_i) \;=\; \sum_{x_{i-1}} p(X_i|x_{i-1}) \sum_{x_{i-2}} p(x_{i-1}|x_{i-2}) \dots \sum_{x_1} p(x_2|x_1)p(x_1)$$

starting from the rightmost terms. This is equivalent to contracting the factor graph in Figure 2.1 from left to right. This way, the cost is *linear* rather than *exponential* in the number of variables. Additionally, intermediate results—*i.e.* factors resulting from contraction—can be memoized and reused to compute other marginals, obviating unnecessary repeated computations. And representing the joint distribution as a graph data structure enables the application of generic algorithms for finding and executing efficient contraction orders.

Commonly used and extensively studied formulations of graphical models include Bayesian networks (BNs) [27], Markov random fields (MRFs) [21], tensor (hyper)networks, and factor graphs. As shown by Frey [8], Factor graphs are strictly more expressive than MRFs and BNs: Any MRF or BN can be converted into a factor graph and back without loss of information, such that the original MRF or BN is recovered; but the converse does not hold, as there exist ways to factorize a function such that that factorization can be expressed precisely with a factor graph, but not with a BN or MRF. Furthermore, Robeva and Seigal [30] show that there is a one-to-one correspondence between factor graphs[2] and tensor hypernetworks, which in turn are a mild generalization of tensor networks as often used in physics. Thus, while this thesis focuses on factor graphs, in principle the results presented in this thesis apply also to tensor networks, BNs, and MRFs, with the caveat that converting an MRF into a factor graph may be computationally intractable, as it requires identifying all maximal cliques in the MRF.

[1] Here lowercase x_j denotes one possible concrete instantiation of variable X_j.

[2] Robeva and Seigal use the term "undirected graphical model", which is often used to refer to MRFs, but their Definition 1.2 in [30] is essentially equivalent to a factor graph, up till normalizing constant.

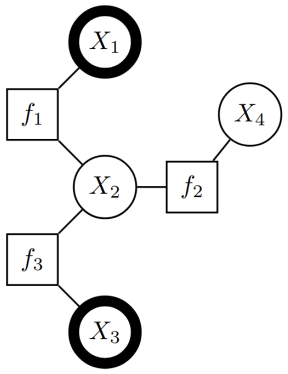
2.1 Factor graphs

There are many somewhat different possible definitions for factor graphs. For example, a general definition of factor graphs would admit directed edges and continuous factor-functions over uncountable domains. However, in the remainder of this thesis, we will follow the definitions and notation of Karimi *et al.* [18]. Thus we will be considering undirected factor graphs with discrete factor-functions; in what follows, $\mathbb{F}$ will denote a finite field.

Definition 1 (Factor graph)**.** *A* factor graph G *is a connected bipartite graph, bipartitioned into a set* $\mathcal{X} = \{X_1, X_2, ..., X_n\}$ *of* variables *and a set* $\mathcal{F} = \{f_1, f_2, ..., f_m\}$ *of* factors*. Each variable* X_i *is associated with a discrete nonempty domain* $\mathcal{D}_i$*. Denote the set of indices of variables adjacent to factor* f_k *by* $S_k \subseteq U := \{1, 2, ..., n\}$*. For any* $S \subseteq U$*, let* $\mathcal{D}_S$ *denote the Cartesian product* $\prod_{i \in S} \mathcal{D}_i$*. Each factor* f_k *is associated with a map* $\mathcal{D}_{S_k} \to \mathbb{F}$*.*

For the purpose of inference—discussed in the next section—we select a subset of variables to be the *boundary*, and denote their indices by $B \subseteq U$. For any $k \in \{1, 2, ..., m\}$ and point $v \in \mathcal{D}_{S_k}$, we denote the value of factor f_k's map at v by $f_k(X_i = v_i : i \in S_k)$ or more briefly by $f_k(v)$. If for all variables X_i in G it holds that $|\mathcal{D}_i| \geq 2$, and also all non-boundary variables in G are connected to at least two factors, we say that G is *non-degenerate*.

As an illustrative example of the above definitions, Figure 2.2 shows the factor graph from Chapter 1, expressed in the notation of Definition (1).



$\mathcal{X} = \{X_1, X_2, X_3, X_4\}$
$\mathcal{F} = \{f_1, f_2, f_3\}$

$U = \{1, 2, 3, 4\}$
$B = \{1, 3\}$

$S_1 = \{1, 2\}$
$S_2 = \{2, 4\}$
$S_3 = \{2, 3\}$

Figure 2.2: A (degenerate) factor graph with three factors and four variables.

2.2 Inference in factor graphs

Given a factor graph G with boundary B, we are interested in the problem of computing the map $g : \mathcal{D}_B \to \mathbb{F}$, where

$$g(v) \;=\; \sum_{w \in \mathcal{D}_{U \setminus B}} \prod_{k=1}^{m} f_k(v, w) \;. \tag{2.1}$$

We will call the problem of computing $g(v)$ for all v in $\mathcal{D}_B$ the *inference problem*. Although this name is perhaps most suggestive of *probabilistic* inference, it is instructive to note that the problem of computing g is quite general—it is relevant in many contexts. As examples, consider the following:

(i) Let $A_1, A_2, ..., A_m$ be matrices over $\mathbb{F}$. Matrix A_i can be seen as a factor over two variables r_i and c_i with domain-sizes corresponding to the dimensions of A_i. The product $A_1 A_2 \cdots A_m$ can then be seen as a chain-like factor graph G. Taking the boundary B to be $\{r_1, c_m\}$, computing the map $g : \mathcal{D}_{r_1} \times \mathcal{D}_{c_m} \to \mathbb{F}$ then corresponds exactly to evaluating the product $A_1 A_2 \cdots A_m$. In other words, chained matrix multiplication is reducible to the inference problem on factor graphs.

(ii) When G represents a probability distribution, the inference problem corresponds to computing the marginal distribution of the variables indicated by B.

(iii) In the context of quantum computing [24], quantum circuits are a form of tensor network [3], and evaluating such a circuit amounts to contracting the corresponding tensor network. And as mentioned earlier, tensor networks are essentially equivalent to factor graphs.

Factor contraction. To compute g, we repeatedly use an operation called *factor contraction*. Let $f_a, f_b \in \mathcal{F}$, with $a \neq b$. Denote by T_i the set $\{1 \leq k \leq m \;:\; i \in S_k\}$ of indices of factors adjacent to variable X_i. We say that i is *internal* to the contraction of f_a and f_b if $i \notin B$ and $T_i \subseteq \{a, b\}$; *i.e.*, if X_i is not in the boundary and X_i is not adjacent to any factors other than f_a or f_b. Let $I_{ab} \subseteq U$ consist of exactly those i that are internal to the contraction of f_a and f_b. Since G is connected, we have $I_{ab} \subseteq S_a \cup S_b$. Define the operation of *contracting* the factors f_a and f_b as follows:

1. Delete f_a and f_b from G.

2. Add to G a new factor f_{ab} which is adjacent to $(S_a \cup S_b) \setminus I_{ab}$, and defined for all v in $\mathcal{D}_{(S_a \cup S_b) \setminus I_{ab}}$ as follows:

$$f_{ab}(v) \;=\; \begin{cases} \sum_{w \in \mathcal{D}_{I_{ab}}} f_a(v,w)\, f_b(v,w) & \text{if } I_{ab} \neq \emptyset \\ f_a(v)\, f_b(v) & \text{if } I_{ab} = \emptyset \end{cases} \tag{2.2}$$

3. Delete from G the variables indexed by I_{ab}.

The map g can be obtained from G by contracting factors until only one factor remains. If G is non-degenerate, then the end result is independent of the order of contractions—the final factor always has g as its map. For this to hold also when G is degenerate in the sense of containing singly-connected non-boundary variables, some preprocessing is required in some of the contractions; see Section 5.3.

Complexity of inference by contraction. Assuming that the cost of evaluating a factor at a given point is $\mathcal{O}(1)$, the cost of contracting a pair of factors f_a and f_b is proportional to $\left|\mathcal{D}_{(S_a \cup S_b) \setminus I_{ab}}\right| \cdot |\mathcal{D}_{I_{ab}}|$, which is the same as $\left|\mathcal{D}_{(S_a \cup S_b)}\right|$. Given a factor graph G with m factors, to obtain g we need a sequence of $m-1$ contractions. The total cost of such a sequence depends strongly on how factors are ordered within the sequence. For example, consider a tree-shaped factor graph with maximum degree k, variable domain sizes in range $[D_{\min}, D_{\max}]$, and n variables: If one orders the contractions so as to proceed from the leaves inward, and always contracts factors that share a variable, then the total cost will be $\mathcal{O}(mD_{\max}^{k+1})$. If instead one were to order the contractions so as to first contract together for each level of the tree all factors at that level, then contract together all factors at even levels, and then contract together all factors at odd levels, then the final contraction would have *exponential* cost $\omega(D_{\min}^{n})$. Figure 2.3 illustrates this inefficient contraction order on a binary tree.

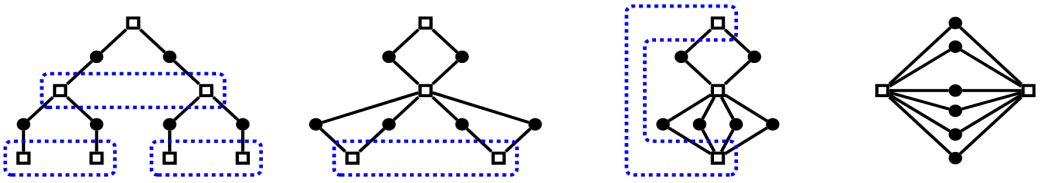


Figure 2.3: A very inefficient contraction order of a small tree-like factor graph. White squares are factors, black nodes are variables. Left: Original factor graph. Dashed blue regions indicate contractions.

From the above example we see that finding efficient contraction orders is an important problem. Using a dynamic programming approach, deter-

ministically finding an optimal contraction order by exhaustive search can be done in time $\tilde{\mathcal{O}}(2^m)$, where m is the number of factors. This is still intractable, and various heuristic algorithms have been developed for finding efficient contraction orders [32].

In general, the problem of factor graph contraction (FGC) is in fact #P-hard. This can be shown by a straightforward reduction of #SAT to FGC. To see this, first recall that #SAT and #3SAT are counting-equivalent; *i.e.*, it is possible to transform in linear time any CNF formula ϕ into a 3CNF formula ψ with the same number of satisfying assignments. Now, given a 3CNF formula ψ, construct an integer-valued factor graph G as follows:

1. For each Boolean variable $x_1, ..., x_n$ in ψ, let G have a corresponding binary variable.

2. For each clause C in ψ, construct a factor f_C, such that f_C is adjacent to exactly those variables which are in C. Define the map of f_C such that for all v in $\{0,1\}^{|C|}$, we have

$$f_C(v) \;=\; \begin{cases} 1 & \text{if } C(v) \text{ is true,} \\ 0 & \text{if } C(v) \text{ is false.} \end{cases}$$

   Since each clause C in ψ contains at most 3 variables, constructing a factor f_C takes constant time.

3. Let the boundary of G be empty.

Now, for a given assignment $v \in \{0,1\}^n$, the map associated with G evaluates to

$$\prod_{C \in \psi} f_C(v) \;=\; \begin{cases} 1 & \text{if } C(v) \text{ is true for all clauses } C \text{ in } \psi, \\ 0 & \text{otherwise,} \end{cases}$$

which is the same as $\psi(v)$. And thus the (degenerate, scalar-valued) map associated with G, which is the same as the result of contracting G, is

$$\sum_{v \in \{0,1\}^n} \prod_{C \in \psi} f_C(v) \;=\; \sum_{v \in \{0,1\}^n} \psi(v) \;=\; \sum_{v \in \{0,1\}^n} \phi(v),$$

which is the number of satisfying truth-assignments to the original CNF formula ϕ, as desired. This completes the reduction of #SAT to FGC, showing that FGC is #P-hard.

Chapter 3

Delegable inference

Williams [39] presented a noninteractive proof system—known as a "Merlin-Arthur" protocol—for the problem of multipoint arithmetic circuit evaluation. In Williams' system, the proof is a univariate polynomial $Q(x)$, such that the desired circuit-evaluations can be efficiently recovered from $Q(x)$, and $Q(x)$ enables fast verification of the correctness of those evaluations. However, in Williams' system, the prover, called "Merlin", is assumed to provide the proof instantaneously, at zero computational cost. More generally, in addition to providing a method for efficient probabilistic verifiability via polynomial identity testing, polynomials also provide methods for error-correction: It is possible to transform between the dual and primal representations of a polynomial—namely a set of evaluations and a vector of coefficients—in near-linear time, even if some of the evaluations were erroneous. This property of polynomials can be used to construct error-correcting codes [29].

Building upon the expedient properties of polynomials, and upon systems like Williams', Björklund and Kaski [5] presented a template for delegable computation which does *not* require the prover to be a wizard: instead, in Björklund and Kaski's framework, the proof is computed in a highly parallelizable manner, and in a way that provides robustness to errors. Algorithms based on the template are called *Camelot algorithms*.

In this chapter, we first briefly outline the Björklund-Kaski template, and then review a Camelot algorithm for solving the inference problem as defined in Section 2.2. The algorithm was developed by Karimi, Kaski, and Koivisto [18]. This chapter merely recalls their work and explicates a few points that had been left implicit in their work. We also follow their notation, with a few exceptions. Henceforth, we will refer to this algorithm as the *delegable (factor graph) contraction algorithm*, or DCA for brevity.

3.1 Outline of the Björklund-Kaski template

The high-level idea of the Björklund-Kaski template is as follows. Suppose we are given a computational problem $\mathcal{P}$, such as for example counting subgraphs in host graphs, or counting the number of solutions to CNF formulas, or computing matrix products. The goal is to be able to delegate and verify the solution of any instance P of $\mathcal{P}$. To this end, first a polynomial $\hat{h} \in \mathbb{F}[x]$, of degree d, is mathematically constructed such that

(i) the solution to P can be efficiently obtained via evaluations of $\hat{h}$,

(ii) and the coefficients of $\hat{h}$ can be used as a proof of correctness of the solution.

Next, an algorithm E for evaluating $\hat{h}$ without access to $\hat{h}$'s coefficients is defined. With the polynomial $\hat{h}$ and algorithm E in hand, delegation of any P can proceed as follows:

1. The delegate computes $e \geq d+1$ evaluations of $\hat{h}$ using E, at distinct points $\xi_1, \ldots, \xi_e$. These evaluations are independent and can be done in parallel.

2. The results $\left\{(\xi_i, \hat{h}(\xi_i))\right\}_{i=1}^{e}$ of the delegated evaluations effectively form a Reed-Solomon code [29]. Thus, even if at most $(e-d-1)/2$ evaluations are erroneous, the delegator can both recover $\hat{h}$'s coefficients $\lambda_0, ..., \lambda_d$ and also identify *which* evaluations were in error. This can be done for example using Gao's near-linear time decoding algorithm [10].

3. The solution to P can then be recovered via evaluations of $\hat{h}$, which can be computed efficiently using the coefficients $\lambda_0, ..., \lambda_d$. The exact way in which the solution is recovered depends on the problem P; for example, in the cases of [18] and [20], the solution is recovered by summing over a certain set of evaluations of $\hat{h}$.

Given coefficients $\lambda'_0, ..., \lambda'_d$ of a polynomial $\hat{h}'$, the delegator can verify that the solution is correct by checking that $\hat{h}' = \hat{h}$. And the equality of $\hat{h}'$ and $\hat{h}$ can be tested using randomized polynomial identity testing; for details, see Section 3.4.

Since the coefficients produced by the delegate need to be verified, we require that one evaluation using E, plus recovering the solution using $\hat{h}$, must be a lot more efficient than computing the whole result without delegation.

3.2 The proof polynomial

Recalling the inference problem from Section 2.2, we have a factor graph G with variables $X_1, X_2, ..., X_n$, factors $f_1, f_2, ..., f_m$, and with boundary B which is a subset of the cutset $C \subseteq U = \{1, 2, ..., n\}$. We wish to compute the map $g : \mathcal{D}_B \to \mathbb{F}$, where $\mathbb{F}$ is a field, and $g(v) = \sum_{w \in \mathcal{D}_{U \setminus B}} \prod_{k=1}^{m} f_k(v, w)$ for all v in $\mathcal{D}_B$. As outlined in Section 3.1, the goal here is to construct a proof polynomial $\hat{h} \in \mathbb{F}[Z]$ from which the desired result—the map g—can be efficiently recovered.

The first core idea in constructing $\hat{h}$ is to extend the map g into a multivariate polynomial $\hat{g} \in \mathbb{F}[Y_i : i \in C]$, with one polynomial indeterminate Y_i for each cutset variable i in C. To do this, first define for each i in C an arbitrary injective map $\eta_i : \mathcal{D}_i \to \mathbb{F}$. Then, for each factor f_k and each w in $\mathcal{D}_{S_k \setminus C}$, construct a polynomial $\hat{f}_{k,w}$ in $\mathbb{F}[Y_i : i \in C]$ that satisfies for all v in $\mathcal{D}_C$ the identity

$$\hat{f}_{k,w}\big(Y_i = \eta_i(v_i) : i \in S_k \cap C\big) \;=\; f_k\big(w, X_i = v_i : i \in S_k \cap C\big) \tag{3.1}$$

and

$$\deg_{Y_i} \hat{f}_{k,w} \leq \begin{cases} |\mathcal{D}_i| - 1 & \text{if } i \in S_k \cap C, \\ 0 & \text{if } i \in S_k \setminus C \ . \end{cases} \tag{3.2}$$

Since the maps η_i are injective, and the degrees of the indeterminates are as in Equation (3.2), the polynomials $\hat{f}_{k,w}$ exist and are unique. They can be constructed by Lagrange interpolation over Equation (3.1). Define the polynomial $\hat{g}$ as follows:

$$\hat{g}(Y_i : i \in C) \;=\; \sum_{w \in \mathcal{D}_{U \setminus C}} \prod_{k=1}^{m} \hat{f}_{k,w}(Y_i : i \in C) \ . \tag{3.3}$$

This brings us close to having the desired kind of proof polynomial; however, we still need to get from $\hat{g}$ to a *univariate* polynomial $\hat{h} \in \mathbb{F}[Z]$. To that end, first define an arbitrary injection $\tau : \mathcal{D}_C \to \mathbb{F}$. Then, for each $i \in C$, define a polynomial $\hat{\ell}_i \in \mathbb{F}[Z]$ that satisfies for all $v \in \mathcal{D}_C$

$$\hat{\ell}_i\big(Z = \tau(v)\big) \;=\; \eta_i(v_i) \tag{3.4}$$

and for all i in C

$$\deg_Z \hat{\ell}_i \;\leq\; |\mathcal{D}_C| - 1 \ . \tag{3.5}$$

Once again, the polynomials $\hat{\ell}_i$ exist and are unique. Finally define the proof polynomial

$$\hat{h}\big(Z\big) \;=\; \hat{g}\big(Y_i = \hat{\ell}_i(Z) : i \in C\big) \ . \tag{3.6}$$

With the proof polynomial $\hat{h}$ constructed as above, $g(s)$ can be computed via $\left|\mathcal{D}_{C\setminus B}\right|$ evaluations of $\hat{h}$. To see this, observe that for any s in $\mathcal{D}_B$, we have the following:

$$\begin{aligned} g(s) &= \sum_{u\in\mathcal{D}_{U\setminus B}} \prod_{k=1}^{m} f_k(u,s) \\ &\qquad\qquad \text{[Split } \mathcal{D}_{U\setminus B} \text{ into components } \mathcal{D}_{C\setminus B} \text{ and } \mathcal{D}_{U\setminus C}\text{]} \\ &= \sum_{t\in\mathcal{D}_{C\setminus B}} \sum_{w\in\mathcal{D}_{U\setminus C}} \prod_{k=1}^{m} f_k(w,t,s) \\ &\qquad\qquad \text{[Apply Equation (3.1)]} \\ &= \sum_{t\in\mathcal{D}_{C\setminus B}} \sum_{w\in\mathcal{D}_{U\setminus C}} \prod_{k=1}^{m} \hat{f}_{k,w}\Big(Y_i = \eta_i(t_i),\, Y_j = \eta_j(s_j) \,:\, i\in C\setminus B,\, j\in B\Big) \\ &\qquad\qquad \text{[Apply Equation (3.3)]} \\ &= \sum_{t\in\mathcal{D}_{C\setminus B}} \hat{g}\Big(Y_i = \eta_i(t_i),\, Y_j = \eta_j(s_j) \,:\, i\in C\setminus B,\, j\in B\Big) \\ &\qquad\qquad \text{[Apply Equation (3.4)]} \\ &= \sum_{t\in\mathcal{D}_{C\setminus B}} \hat{g}\Big(Y_i = \hat{\ell}_i\big(\tau(t,s)\big) \,:\, i\in C\Big) \\ &\qquad\qquad \text{[Finally, by Equation (3.6)]} \\ &= \sum_{t\in\mathcal{D}_{C\setminus B}} \hat{h}\big(Z = \tau(t,s)\big)\ . \end{aligned} \tag{3.7}$$

In other words, the inference problem on G can be solved via $\left|\mathcal{D}_{C\setminus B}\right|$ evaluations of $\hat{h}$. Using Horner's rule, each evaluation takes $\mathcal{O}(\deg_Z \hat{h})$ operations in $\mathbb{F}$; and from Equations (3.6), (3.5), (3.3), and (3.2) it follows that

$$\deg_Z \hat{h} \;\le\; \big(|\mathcal{D}_C| - 1\big) \sum_{k=1}^{m} \sum_{i\in S_k\cap C} \big(|\mathcal{D}_i| - 1\big)\ . \tag{3.8}$$

3.3 The evaluation algorithm

In this section, we review Karimi *et al.*'s [18] algorithm for evaluating the proof polynomial defined in Section 3.2. However, we eschew the scheme

based on explicitly constructing Vandermonde matrices, and instead explicate the more efficient scheme based on repeated univariate interpolation, which Karimi *et al.* mentioned but left implicit.

The setting is as follows. We are given a factor graph G with boundary B, a cutset $C \supseteq B$, injective maps $\eta_i : \mathcal{D}_i \to \mathbb{F}$, the injective map $\tau : \mathcal{D}_C \to \mathbb{F}$, and a point $\zeta \in \mathbb{F}$. We wish to compute the evaluation $\hat{h}(\zeta)$ of the proof polynomial $\hat{h}$ defined in Section 3.2. We also assume that the domain of every variable admits a total order, and thus for all $j \in U$ there exists a bijection between $[\mathcal{D}_j] = \{0, 1, ..., |\mathcal{D}_j| - 1\}$ and $\mathcal{D}_j$. We denote the n^{th} element of $\mathcal{D}_j$ by $\mathcal{D}_j(n)$.

As a first step, interpolate the polynomials $\hat{\ell}_i$. This needs to be done only once for the given G, C, τ, and $\{\eta_i\}_{i \in C}$; subsequent evaluations with different ζ may use the same set of interpolated polynomials $\{\hat{\ell}_i\}_{i \in C}$.

Next, every factor f_k in G that is adjacent to at least one cutset variable is iteratively transformed into a factor $\bar{f}_k : \mathcal{D}_{S_k \setminus C} \to \mathbb{F}$ consisting of evaluations of the polynomials $\hat{f}_{k,w}$, defined as

$$\bar{f}_k(w) \;=\; \hat{f}_{k,w}\big(Y_i = \hat{\ell}_i\big(\zeta\big) : i \in S_k \cap C\big)\,.$$

We now explain in more detail how this is accomplished. For every i in C and every k such that $i \in S_k$, the following is done:

1. For each $w \in \mathcal{D}_{S_k \setminus C}$ and $v \in \mathcal{D}_{S_k \cap C \setminus \{i\}}$, the partially evaluated polynomial $\hat{f}_{k,w}\left(Y_j = \eta_j(v_j) : j \in S_k \cap C \setminus \{i\}\right)$ is a polynomial in $\mathbb{F}[Y_i]$. Interpolate the coefficients $\pi_{w,v} \in \mathbb{F}^{|\mathcal{D}_i|}$ of that polynomial according to Equation (3.1).

2. Replace the factor f_k with a factor f'_k consisting of all the interpolated coefficients, defined for each $n \in [\mathcal{D}_i]$ as $f'_k\big(w, v, \mathcal{D}_i(n)\big) = \pi_{w,v}(n)$.

3. Create a variable $P_{i,k}$ with domain $[\mathcal{D}_i]$. Detach X_i from f'_k, and set $P_{i,k}$ and f'_k to be adjacent; *i.e.*, replace X_i with $P_{i,k}$ among the neighbors of f'_k.

4. Create factor $e_{i,k}$, adjacent to $P_{i,k}$, defined as $e_{i,k}(c) = \hat{\ell}_i(\zeta)^c$.

5. Contract f'_k and $e_{i,k}$ over their single shared variable $P_{i,k}$. One can verify from the above definitions and Equation (2.2) that, for each $w \in \mathcal{D}_{S_k \setminus C}$ and $v \in \mathcal{D}_{S_k \cap C \setminus \{i\}}$, the resulting factor takes value

$$\hat{f}_{k,w}\left(Y_i = \hat{\ell}_i(\zeta),\, Y_j = \eta_j(v_j) \;:\; j \in S_k \cap C \setminus \{i\}\right).$$

Once all $i \in C$ have been processed as above, all cutset variables are fully detached; remove them from the factor graph.

Denote by $G_{Z=\zeta}$ the resulting factor graph, where all cutset variables have been removed, and each factor f_k has been replaced by $\bar{f}_k$. The graph $G_{Z=\zeta}$ can be thought of as the evaluation of a polynomial extension of G at ζ. Indeed, if $\zeta = \tau(v)$ for some $v \in \mathcal{D}_C$, then $G_{Z=\zeta}$ is the same as G with all factors partially evaluated at v. Finally, contract the factors in $G_{Z=\zeta}$, until a single scalar-valued factor remains. This scalar value is the desired $\hat{h}(\zeta)$.

3.4 Preparing and verifying the proof

As outlined in Section 3.1, the coefficients of the proof polynomial are computed by evaluating the proof polynomial at multiple distinct points using the evaluation algorithm (see Section 3.3) and then decoding the coefficients from the obtained point-evaluation pairs. A degree-d polynomial is uniquely determined by any $e \geq d+1$ unique point-evaluation pairs, so long as no more than $\lfloor (e-d-1)/2 \rfloor$ of those evaluations are erroneous. And as shown by Gao [10], given e such point-evaluation pairs, we can both decode the coefficients and also identify the erroneous evaluations in $\mathcal{O}\big(M(e)\log(e)\big)$ operations in $\mathbb{F}$, where $M(e)$ is the cost of multiplying degree-e polynomials. Recall that for the proof polynomial, an upper bound for the degree d is given in (3.8). Thus a proof with tolerance for x errors can be prepared by delegating the computation of

$$2x + 1 + \big(|\mathcal{D}_C| - 1\big) \sum_{k=1}^{m} \sum_{i \in S_k \cap C} \big(|\mathcal{D}_i| - 1\big)$$

unique evaluations of $\hat{h}$ via the evaluation algorithm. Note that these evaluations are independent and can be performed in parallel.

Verifying the proof. The proof can be verified by randomized polynomial identity testing. Assume the following setting: We are given a factor graph G with boundary B, a cutset $C \supseteq B$, injective maps $\eta_i : \mathcal{D}_i \to \mathbb{F}$, the injective map $\tau : \mathcal{D}_C \to \mathbb{F}$, and a polynomial $\hat{h}'(Z)$ in coefficient form

$$\hat{h}'(Z) \;=\; \lambda'_0 + \lambda'_1 Z + \lambda'_2 Z^2 + \ldots + \lambda'_d Z^d \,,$$

and we want to determine whether $\hat{h}'$ equals the true proof polynomial $\hat{h}$ corresponding to G, C, $\{\eta_i\}_{i \in C}$ and τ. We can assume that the degree d of $\hat{h}'$ is bounded by (3.8). (If it is not, then we can immediately reject $\hat{h}'$.) To test whether $\hat{h}'$ equals $\hat{h}$, we draw a value ζ from $\mathbb{F}$ uniformly at random,

and compare $\hat{h}'(\zeta)$ to $\hat{h}(\zeta)$, where $\hat{h}'(\zeta)$ can be evaluated in $\mathcal{O}(d)$ operations using Horner's rule, and $\hat{h}(\zeta)$ is computed using the evaluation algorithm.

Since a degree-d polynomial is uniquely determined by any $d+1$ unique point-value pairs, two *distinct* degree-d polynomials can agree at at most d unique points. Thus if $\hat{h}'$ is incorrect, the probability that $\hat{h}'(\zeta)$ equals $\hat{h}(\zeta)$ is at most $d/|\mathbb{F}|$. Conversely, if $\hat{h}'$ is correct then $\hat{h}'(\zeta)$ always equals $\hat{h}(\zeta)$ for any ζ. Thus this test has no false negatives, and a false positive rate of at most $d/|\mathbb{F}|$. And greater certainty can be attained simply by repeating the test with another uniform random draw. With t independent tests, the probability of an incorrect proof being accepted is at most $(d/|\mathbb{F}|)^t$.

3.5 Complexity of the DC algorithm

Given a factor graph G and a cutset C, denote by $G[C]$ the factor graph obtained by conditioning on the variables in C taking some specific values. The graph $G[C]$ is structurally equivalent to G with all variables in C removed. A slightly modified version of Theorem 1 in [18] is as follows:

Theorem 1. *Let G be a factor graph with m factors, and variables with domains of size at most D. Given a cutset C of size k, and a contraction ordering that allows for contracting $G[C]$ in time $T_{G[C]}$, the DC algorithm solves the inference problem using*

(i) $\tilde{\mathcal{O}}(D^{k+1}m)$ *independent, parallelizable, evaluations of a proof polynomial, each requiring* $\tilde{\mathcal{O}}\big(D^k + k\,\|G\| + T_{G[C]}\big)$ *operations in* $\mathbb{F}$,

(ii) and one interpolation that requires $\tilde{\mathcal{O}}(D^{k+1}m)$ *operations,*

(iii) and one multipoint evaluation of the proof polynomial, also requiring $\tilde{\mathcal{O}}(D^{k+1}m)$ *operations.*

The correctness of the DC algorithm has been established in Sections 3.2 and 3.3. What remains to be shown is that the complexity is as in Theorem 1. To that end, in what follows we will go through the phases of the algorithm as described in 3.3 and analyse the cost of each phase.

The first step in the DC algorithm is to interpolate the polynomials $\{\hat{\ell}_i\}_{i\in C}$, as defined by Equation (3.4). As will be discussed in Section 4.3, the complexity of interpolating a polynomial from d points is $\mathcal{O}\big(M(d)\log(d)\big)$. From (3.5) we then have that the complexity of interpolating each of the k polynomials $\hat{\ell}_i$ is $\mathcal{O}\big(M(|D_C|)\log(|D_C|)\big) \in \mathcal{O}\big(M(D^k)\log(D^k)\big)$. Recalling that with Schönhage-Strassen multiplication, we have $M(n) = n\log(n)\log\log(n)$,

we obtain the first $\mathcal{O}\big(kD^k \log^2(D^k) \log\log(D^k)\big) = \tilde{\mathcal{O}}(D^k)$ term in the evaluation cost claimed in Theorem 1.

After interpolating the polynomials $\{\hat{\ell}_i\}_{i\in C}$, the DCA performs one $\tilde{\mathcal{O}}\big(\,|\mathcal{D}_i|\,\big)$ interpolation for each element in $\mathcal{D}_{S_j\setminus\{i\}}$, for each factor f_j adjacent to a cutset variable i, for each cutset variable i. In other words, the number of operations in this phase is at most

$$\sum_{i\in C}\sum_{j\,:\,i\in S_j} \big|\mathcal{D}_{S_j\setminus\{i\}}\big| \cdot \tilde{\mathcal{O}}\big(\,|\mathcal{D}_i|\,\big) \;=\; \tilde{\mathcal{O}}\left(\sum_{i\in C}\sum_{j=1}^{m} \big|\mathcal{D}_{S_j}\big|\right),$$

which is indeed the same as the second term in the evaluation cost, namely $\tilde{\mathcal{O}}\big(k\,\|G\|\,\big)$.

The last step in processing cutset variable i and factor f_j is to contract f'_j with $e_{i,j}$. Denoting by S_a and S_b the sets of variables adjacent to f'_j and $e_{i,j}$ respectively, we have from Equation (2.2) and the definitions of f'_j and $e_{i,j}$ that the cost of this operation is $|\mathcal{D}_{S_a\cup S_b}| \le \big|\mathcal{D}_{S_j}\big|$. Thus the total cost of these contractions is also $\tilde{\mathcal{O}}\big(k\,\|G\|\,\big)$.

After the factor graph G has been transformed—by the above interpolations and contractions—into the graph $G_{Z=\zeta}$, what remains to be done is to contract $G_{Z=\zeta}$. Since $G_{Z=\zeta}$ is structurally equivalent to $G[C]$, the cost of this contraction is by assumption $T_{G[C]}$.

In order to decode the coefficients of the proof polynomial $\hat{h}$, we need $d+1$ evaluations, where d is the degree of the proof. Recalling Equation (3.8) we can bound d:

$$\begin{aligned}
\deg_Z \hat{h} \;&\le\; \big(\,|\mathcal{D}_C| - 1\big)\sum_{j=1}^{m}\sum_{i\in S_j\cap C}\big(\,|\mathcal{D}_i| - 1\big) \\
&\le\; |\mathcal{D}_C|\, m\, |C|\, D \\
&\le\; D^{k+1} m k\ .
\end{aligned}$$

This establishes the claimed $\tilde{\mathcal{O}}\big(D^{k+1}m\big)$ bound on the number of evaluations in Theorem 1.

Once the evaluations have been computed, decoding the coefficients of $\hat{h}$ requires one call to Gao's [10] decoding algorithm, with input size d. As seen in Section 4.4, the cost for this is $\tilde{\mathcal{O}}(d) = \tilde{\mathcal{O}}\big(D^{k+1}m\big)$. This is the interpolation referred to in point (ii) of Theorem 1.

Finally, the result of inference, namely the map $g : \mathcal{D}_B \to \mathbb{F}$, can be recovered according to Equation (3.7). This requires first evaluating a degree-d polynomial at $\big|\mathcal{D}_{C\setminus B}\big| \cdot |\mathcal{D}_B| = |\mathcal{D}_C|$ points, and then performing $\mathcal{O}(|\mathcal{D}_C|)$ additions in $\mathbb{F}$. As will be shown in Section 4.3, the complexity of multipoint

evaluation of degree d polynomials is $\mathcal{O}\big(M(d)\log(d)\big)$; and noting that $d \geq |\mathcal{D}_C|$, we see that the cost of this final step is dominated by the multipoint evaluation, with complexity $\tilde{\mathcal{O}}\,(d) \leq \tilde{\mathcal{O}}\big(D^{k+1}m\big)$. We finish the informal proof of Theorem 1 by noting that this final cost is accounted for in point (iii).

Relation to standard inference algorithms. We now analyse the cost $T_{G[C]}$ of contracting $G[C]$. As seen in Section 2.2, contracting a factor graph is #P-hard in general. However, more fine-grained complexity bounds can be obtained by considering the structure of the graph. In particular, graphs with low *tree-width*[1] have contraction orders that allow for relatively efficient inference [21, Theorem 9.12]. And in fact Chandrasekaran *et al.* [6] show that—under certain plausible assumptions—low treewidth is the *only* structural property that ensures the possibility of tractable inference. This makes the choice of cutset C important, as it affects the treewidth w of $G[C]$. Unfortunately, finding a cutset C giving minimal w is NP-hard; various heuristic algorithms for finding good cutsets exist [4]. Furthermore, as noted in Chapter 2, even if $G[C]$ admits an efficient contraction order, *finding* such an order is still intractable (NP-hard) in general. Although note that an efficient contraction order needs to be found only once for $G[C]$, and can be employed for all configurations of C.

Theorem 9.12 in [21] gives a bound on the complexity of exact inference in terms of the maximum clique size in a graph induced by a variable elimination algorithm. To translate that bound to factor graphs, first note that variable elimination can be replaced by contraction of all factors adjacent to the target variable; albeit that in general this might eliminate more than just the target variable. And from Koller *et al.*'s definition of the variable elimination algorithm (Algorithm 9.1) and of induced graphs (Definition 9.5), one can deduce that the induced graph for any valid factor graph is chordal. Since the treewidth of a chordal graph is the maximum clique size minus one, we have that a factor graph with m factors, largest variable domain size D, empty cutset, and treewidth w can be contracted to a scalar in time $\mathcal{O}(mD^{w+1})$.

Thus, to summarize: Under the assumption that we can efficiently find an optimal contraction order for $G[C]$, standard algorithms for exact inference bound $T_{G[C]}$ by $\mathcal{O}(mD^{w+1})$.

If we now consider the full problem of inference on G with cutset C of size k, assuming we have an optimal contraction order for $G[C]$, the cost of inference using the conditioning algorithm [21, Algorithm 9.5] is $\mathcal{O}(mD^kD^{w+1})$,

[1]Here the treewidth of a factor graph means the treewidth of the graph with variables as vertices, with two variables adjacent iff they share a factor.

where w is the treewidth of $G[C]$. In cases when inference is hard—*i.e.* when there are no cutsets giving small w—we note that $T_{G[C]}$ dominates the other terms of Theorem 1. In that setting, the DC algorithm costs roughly Dm times as much as the cutset conditioning algorithm; but in contrast to cutset conditioning, DCA also enables fast verification of the result.

3.6 Chinese remaindering

In previous sections, we often assumed that $\mathbb{F}$ is a finite field. However, in practical applications, factor graphs would in general comprise floating-point numbers, represented in some finite-precision radix-point number system. In this section, we outline a reduction from contracting such rational-valued factor graphs to contracting factor graphs comprising only integers from prime fields.

Let G be a factor graph comprising rational values, each represented with at most d digits in radix R; that is, for each value ρ we have

$$\rho \;=\; \sigma R^u \left(s_1 R^{-1} + s2R^{-2} + \ldots + s_d R^{-d}\right)$$

for some integer u, $\sigma \in \{0, 1\}$, and $s_1, s_2, \ldots, s_d \in \{0, 1, \ldots, R\}$. The reduction proceeds as follows:

1. Convert each factor to f_k to an integer-valued factor by multiplying all of its values by the largest denominator. More precisely, for each factor f_k, find the smallest exponent c_k such that for all $v \in \mathcal{D}_{S_k}$ we have $R^{c_k} f_k(v) \in \mathbb{Z}$, and then multiply all values of f_k by R^{c_k}. Denote the resulting factor graph $G_{\mathbb{Z}}$.

2. For $k \in \{1, 2, ..., m\}$, let u_k be the largest exponent of any value of f_k in G. Then for all values ρ of f_k we have $|\rho| \leq R^{u_k}$. Thus, recalling (2.1), we can upper-bound the absolute values of $g_{\mathbb{Z}}(v)$ for all $v \in \mathcal{D}_B$

$$\begin{aligned} |g_{\mathbb{Z}}(v)| \;&\leq\; \sum_{w \in \mathcal{D}_{U \setminus B}} \prod_{k=1}^{m} R^{c_k} f_k(v, w) \\ &\leq\; \sum_{w \in \mathcal{D}_{U \setminus B}} \prod_{k=1}^{m} R^{c_k} R^{u_k} \\ &\leq\; \left|\mathcal{D}_{U \setminus B}\right| \cdot R^{\sum_{k=1}^{m} c_k} R^{\sum_{k=1}^{m} u_k} \;=\; M\,. \end{aligned}$$

3. Find distinct primes $p_1, p_2, \ldots, p_l$ such that $\prod_j p_j > 2M$. For each prime p_j, let $G_{\mathbb{Z}_{p_j}}$ be the factor graph obtained by taking all values in $G_{\mathbb{Z}}$ modulo p_j.

4. For $j = 1, 2, \ldots, l$, performing all arithmetic operations modulo p_j, contract $G_{\mathbb{Z}_{p_j}}$ to obtain $g_{\mathbb{Z}_{p_j}}$.

Once the maps $g_{\mathbb{Z}_{p_j}}$ have been computed for $j = 1, 2, \ldots, l$, the desired map g can be recovered. To see this, recall the Chinese Remainder Theorem:

Theorem 2 (Chinese Remainder Theorem for $\mathbb{Z}$)**.**
Let $p_1, ..., p_l \in \mathbb{Z}$ be coprime and greater than 1, and let $P = \prod_{i=1}^{l} p_i$. Let $a_i \in \mathbb{Z}$ be such that $0 \leq a_i < p_i$, for $i = 1, ..., l$. Then, for the system of equations

$$x \equiv_{p_i} a_i, \;\; i = 1, \ldots, l$$

where $\equiv_p$ denotes congruence modulo p, there exists a solution, and that solution is unique modulo P.

A solution to the system of congruences in Theorem 2 is

$$x \;:=\; \sum_{i=1}^{l} a_i \left(\prod_{j \neq i} (p_j)_{p_i}^{-1} \right) \prod_{j \neq i} p_j \tag{3.9}$$

where $(p_j)_{p_i}^{-1}$ denotes the inverse of p_j modulo p_i. Note that for all $v \in \mathcal{D}_B$ and $j \in \{1, 2, \ldots, l\}$ we have $g(v) \equiv_{p_j} g_{\mathbb{Z}_{p_j}}$, and the other conditions of Theorem 2 are also satisfied. Thus we can apply Theorem 2 and recover the values of g via evaluations of the r.h.s. of Equation (3.9).

Chapter 4

The fast polynomial toolbox

In previous chapters, we have seen how inference in graphical models can be made delegable using a framework built on computing certain kinds of polynomials. In order to implement that framework in practice, we need efficient algorithmic tools for working with univariate polynomials. In particular, we need fast algorithms for multiplication, computing quotients and remainders, multi-point evaluation, and interpolation from possibly corrupted data. This chapter gives an overview of a toolbox of such algorithms. With the exception of Gao's [10] decoding algorithm—which is used for recovering polynomial coefficients from partially corrupted data—the exposition in this chapter is largely a summary of key chapters in [36].

4.1 Fast polynomial multiplication

All of the other basic operations on polynomials—all of the other tools in the algorithmic toolbox—build upon multiplication. Consequently, having a fast algorithm for multiplication is crucial. Let f and g be polynomials in $\mathbb{F}[x]$, of degree at most d. The straightforward, naïve method of multiplying fg requires $\mathcal{O}(d^2)$ operations in $\mathbb{F}$. Much work has been done in developing faster algorithms for polynomial—and integer—multiplication. Two notable algorithms with better asymptotic complexity include a well-known $\mathcal{O}(n^{1.59})$ algorithm by Karatsuba and Othman, and an $\mathcal{O}\big(n(\log n)(\log\log n)\big)$ algorithm by Schönhage and Strassen [33]. Algorithms with even better asymptotic complexities have also been developed, but they do not provide advantages in practice, as they begin outperforming for example Schönhage and Strassen's algorithm only for impractically large inputs; for an example of such a "galactic algorithm", see *e.g.* [12].

As shown in Chapter 8 of [36], the complexity of Schönhage-Strassen

multiplication for polynomials in commutative rings in general is roughly $63n\log(n)\big(\log\log n\big) + \mathcal{O}(n\log n)$. In the context of delegable inference on factor graphs, the degrees of proof polynomials can easily exceed 2^{15}; see for example Table 6.1 in Section 6.2. While the crossover point between the Karatsuba and Schönhage-Strassen algorithms depends on many factors, from the algorithms' implementation details to the hardware on which they are executed, it seems reasonable to surmise that the degrees of proof polynomials in realistic applications would often exceed that crossover point. This makes Schönhage-Strassen a reasonable choice of multiplication algorithm for the purposes of this thesis. In what follows, we will give a very high-level overview of the algorithm. This overview is based largely on Chapter 8 of [36].

Let $f = \sum_{i=0}^{d} \phi_i x^i \in \mathbb{F}[x]$. A natural way to represent f is as a vector of coefficients, $[\phi_0, \phi_1, ..., \phi_d]$. However, multiplying polynomials in this coefficient form is not very efficient: Karatsuba's algorithm may be the most efficient algorithm operating on polynomials in coefficient form, and it is $\mathcal{O}(n^{1.59})$. But f is also fully determined by any set of $d+1$ evaluations at unique points $\xi_0, ..., \xi_d$; that is, for any set $\{\xi_i\}_{i=0}^{d}$ of unique points, f can be represented by the set $\big\{\big(\xi_i, f(\xi_i)\big)\big\}_{i=0}^{d}$. Multiplication of polynomials f and g in this dual form is of only linear complexity: One need only take $d+1$ point-wise products $f(\xi_i)\, g(\xi_i)$.

The high-level idea in Schönhage-Strassen multiplication is to transform the operands f and g from coefficient form to the dual (evaluation form), perform point-wise multiplications to obtain the dual representation of fg, and then perform an inverse transformation to obtain fg in coefficient form. The transformation between primal (coefficient form) and dual (evaluation form) is carried out as a discrete Fourier transform (DFT). When the ring $\mathbb{F}$ contains a primitive root of unity of order $n = 2^k$ for some $k \in \mathbb{N}$, the n-point DFT can be performed in time $\mathcal{O}(n\log n)$, and is called a fast Fourier transform (FFT) [36, Chapter 8.2]. The availability of such an FFT then makes it possible to multiply polynomials $f, g \in \mathbb{F}[x]$ in time $\mathcal{O}(n\log n)$, conditional on $\deg(fg)$ being less than n.

However, when the ring $\mathbb{F}$ does *not* contain the required primitive roots of unity, the abovementioned FFT is also not directly available. Schönhage-Strassen multiplication overcomes this obstacle by reducing the original problem of computing fg into a problem of computing $\hat{f}\hat{g}$, where $\hat{f}$ and $\hat{g}$ are elements of a cleverly constructed quotient ring R where the *coefficients* are polynomials, such that R is guaranteed to have a suitable root of unity (which conveniently turns out to be a simple monomial). The polynomials $\hat{f}$ and $\hat{g}$ can then be multiplied using the above-mentioned scheme consisting of FFT,

pointwise multiplication in dual form, and finally inverse FFT. Since the coefficients are polynomials of degree at most $\sqrt{n}$, performing the pointwise multiplications in the dual leads to at most $\sqrt{n}+1$ further multiplications of polynomials, but now of degree at most $\sqrt{n}$; these multiplications are handled with recursive calls. Assuming only that 2 is a unit in $\mathbb{F}$, the above approach yields an $\mathcal{O}(n\log(n)\log\log(n))$ algorithm for multiplying polynomials $f, g \in \mathbb{F}[x]$ such that $\deg(fg) < n = 2^k$ for some $k \in \mathbb{N}$.

4.2 Fast polynomial quotient and remainder

For most polynomial rings of interest, computing the quotient and remainder of two polynomials can be reduced to iterated multiplication. The complexity of the resulting division algorithm is equivalent to that of multiplication, up till constant factors. In what follows, we summarize the main points in the design of such an algorithm.

Suppose we are given $a, b \in R[x]$, where R is a commutative ring with 1, and $\deg(a) = n \geq m = \deg(b)$. We want to find $q, r \in R[x]$ such that $a = qb + r$ and $\deg(r) < m$. If the leading coefficient β of b is not a unit in R, then no such q and r exist; thus we assume that β is a unit. We can further assume without loss of generality that β is 1; *i.e.* that b is monic. (For unit β, we can reduce division of a by b to division of $\beta^{-1}a$ by $\beta^{-1}b$; the quotient q will be the same, and the desired r can be recovered with one multiplication by β, and $\beta^{-1}b$ is monic.)

With foresight, define the *k-reversal* of a polynomial a as $\mathrm{rev}_k(a) = x^k a(\frac{1}{x})$. Note that when $k = \deg(a)$, $\mathrm{rev}_k(a)$ is a polynomial with the same coefficients as a, but in reverse order. We now note that

$$x^n a\left(\frac{1}{x}\right) \;=\; x^{n-m} q\left(\frac{1}{x}\right) x^m b\left(\frac{1}{x}\right) \;+\; x^{n-(m-1)} x^{m-1} r\left(\frac{1}{x}\right)$$

or, equivalently,

$$\mathrm{rev}_n(a) \;=\; \mathrm{rev}_{n-m}(q)\ \mathrm{rev}_m(b) \;+\; x^{n-m+1}\,\mathrm{rev}_{m-1}(r)\ .$$

Thus we have that

$$\mathrm{rev}_{n-m}(q) \;\equiv\; \mathrm{rev}_n(a)\ \mathrm{rev}_m(b)^{-1} \pmod{x^{n-m+1}}\ .$$

The foresight now pays off: finding q reduces to finding the inverse of $\mathrm{rev}_m(b)$ modulo x^{n-m+1}, multiplying by $\mathrm{rev}_n(a)$, and $(n-m)$-reversing the result. The remainder is then also easy to obtain as $r = a - qb$.

Computing the inverse of $\mathrm{rev}_m(b)$ modulo x^{n-m+1} is reduced to iterative polynomial multiplications via Newton's method. In more general terms, we are given $f \in R[x]$ and $k \in \mathbb{N}$, such that $f(0) = 1$, and we wish to find $g \in R[x]$ such that $fg \equiv 1 \pmod{x^k}$. Applying Newton's method to find the root of $1/g - f = 0$, we have

$$g_{i+1} \;=\; g_i - \frac{1/g_i - f}{-1/g_i^2} \;=\; 2g_i - g_i^2 f \;.$$

Choosing $g_0 = 1$, it is easy to show inductively that for all i, we have $fg_i \equiv 1 \pmod{x^{2^i}}$. Indeed, the base case $i = 0$ holds since $f(0) = 1$; and for $i > 0$ the induction hypothesis gives us that $fg_{i-1} = 1 + hx^{2^{i-1}}$ for some $h \in R[x]$, and so

$$\begin{aligned} fg_i \;&=\; f\left(2g_{i-1} - g_{i-1}^2 f\right) \\ &=\; 2 + 2hx^{2^{i-1}} - \left(1 + 2hx^{2^{i-1}} + h^2 x^{2^i}\right) \\ &=\; 1 - h^2 x^{2^i} \;\equiv\; 1 \pmod{x^{2^i}} \;. \end{aligned}$$

From the above, we see that the inverse of $\mathrm{rev}_m(b)$ modulo x^{n-m+1} indeed exists, and can be obtained with $\lceil \log(n-m+1) \rceil$ iterations of Newton's method. Each iteration requires a constant number of polynomial multiplications. In total, the complexity is $\mathcal{O}\left(M\big(\deg(a)\big)\right)$, where M is the cost of polynomial multiplication; for details, see Chapter 9 of [36].

4.3 Multipoint evaluation and interpolation

As seen in Chapter 3, fast polynomial interpolation is required for the delegable inference algorithm. Here we summarize an algorithm for interpolation with $\mathcal{O}\big(M(n)\log(n)\big)$ complexity, where n is the number of interpolation points—and thus also an upper bound on the degree of the resulting polynomial—and M is the complexity of polynomial multiplication. This interpolation algorithm in turn builds upon an algorithm for fast multi-point polynomial evaluation. We begin by outlining the multi-point evaluation algorithm.

Multipoint evaluation. Let $n = 2^k$ for some $k \in \mathbb{N}$, let $f \in R[x]$ be a polynomial of degree at most n, and denote by $\upsilon_0, \upsilon_1, ..., \upsilon_{n-1}$ the points in R at which we wish to evaluate f. For any $f \in R[x]$, evaluating f at a point υ is the same as taking the remainder of f modulo $(x - \upsilon)$. Thus multi-point evaluation can be reformulated as computing the

remainders $\text{rem}(f, x-\upsilon_0), \text{rem}(f, x-\upsilon_1), \ldots, \text{rem}(f, x-\upsilon_{n-1})$. Now note that for any monic $g \in R[x]$ and any $h \in R[x]$ that divides g, it holds that $\text{rem}\big(\text{rem}(f,g), h\big) = \text{rem}(f,h)$.[1] This naturally leads to the following algorithm:

1. Construct the *subproduct tree* T: A full binary tree with the monomials $p_{0,i} = (x - \upsilon_i)$ as the leaves, and the j^{th} vertex of level l (counting upward from leaves) containing the product $p_{l,j} = p_{l-1,2j} \cdot p_{l-1,2j+1}$ of its children. The root of T will thus be $p_{k,0} = \prod_{i=0}^{n-1}(x - \upsilon_i)$.

2. Starting at the root of T, take the remainder $r_{k,0} = \text{rem}(f, p_{k,0})$. Traverse down T, taking at vertex j of level l the remainder $\text{rem}(r_{l+1,\lfloor j/2 \rfloor},\, p_{l,j})$. Note that the product-polynomials $p_{l,j}$ are all monic and divide the product-polynomials in their parent vertices. Thus, when the leaves are reached, we will have obtained the desired remainders $\text{rem}\big(f, (x-\upsilon_i)\big)$.

To construct T, at level $l = 1 \ldots k$ we need to perform 2^{k-l} multiplications of polynomials of degree 2^l, and thus perform $2^{k-l}\mathcal{O}\big(M(2^l)\big) = \mathcal{O}\big(M(2^k)\big) = \mathcal{O}\big(M(n)\big)$ operations at each level. If we use fast division when computing the remainders while traversing down T, at each level l we once again do $\mathcal{O}\big(M(n)\big)$ work. Since T has $k = \log(n)$ levels, the above algorithm solves n-point evaluation of polynomials of degree less than n with a time complexity of $\mathcal{O}\big(M(n)\log(n)\big)$.

Interpolation. Suppose we are given distinct points $\upsilon_0, \upsilon_1, ..., \upsilon_{n-1}$ and arbitrary points $\gamma_0, \gamma_1, ..., \gamma_{n-1}$ in R. We wish to compute the unique polynomial $f \in R[x]$ of degree less than n that satisfies $f(\upsilon_i) = \gamma_i$ for all i. It is easy to check that f is given by the Lagrange interpolation formula:

$$f(x) \;=\; \sum_{i=0}^{n-1} \gamma_i \prod_{j \neq i} \frac{x - \upsilon_j}{\upsilon_i - \upsilon_j} \;. \tag{4.1}$$

We assume that $(\upsilon_i - \upsilon_j)$ is a unit in R for all $i \neq j$. (Note that since the points υ_i are unique, this holds always when R is a field.) The above expression can be rewritten as

$$f(x) \;=\; \sum_{i=0}^{n-1} \gamma_i \sigma_i \frac{p(x)}{x - \upsilon_i}$$

where $\sigma_i = \prod_{j \neq i} \frac{1}{\upsilon_i - \upsilon_j}$ and $p(x) = \prod_{k=0}^{n-1}(x - \upsilon_k)$.

[1] Indeed, we can assume that $f = qg + \text{rem}(f,g)$, and $g = ph$ for some $q, p \in R[x]$, and thus $\text{rem}\big(\text{rem}(f,g),\, h\big) = \text{rem}\big(f - qg,\, h\big) = \text{rem}(f - qph,\, h) = \text{rem}(f,h)$.

Now note that the derivative of $p(x)$ is $p'(x) = \sum_{j=0}^{n-1} \frac{p(x)}{x-\upsilon_j}$. Evaluating p' at any υ_i leads to all except the i^{th} sum-term vanishing, giving

$$p'(\upsilon_i) \;=\; \frac{p(\upsilon_i)}{x-\upsilon_i} \;=\; \prod_{j\neq i}(\upsilon_i - \upsilon_j) \;=\; \frac{1}{\sigma_i}\;.$$

Thus we can compute the terms $\sigma_0, \sigma_1, \ldots \sigma_{n-1}$ using multi-point evaluation of $p'(x)$, for a computational cost of $\mathcal{O}\big(M(n)\log(n)\big)$. And the derivative p' can be computed in just n arithmetic operations in R, after constructing the subproduct tree to obtain the coefficients of p.

Once the subproduct tree T for p and the terms σ_i have been found, the Lagrange interpolation polynomial can computed by placing the terms $t_{0,i} = \gamma_i\sigma_i$ at the leaves of T, then traversing up T, computing at vertex j of level l the polynomial $t_{l,j} = t_{l-1,2j}\cdot p_{l-1,2j+1} + t_{l-1,2j+1}\cdot p_{l-1,2j}$. The root term $t_{\log(n),0}$ will be the r.h.s. of Equation (4.1). Informally, this can be thought of as propagating upward from each leaf a term $\gamma_i\sigma_i$ in such a way that it ends up getting multiplied by all terms of $p(x)$ except $(x-\upsilon_i)$, and summing the resulting polynomials, thereby finally obtaining the Lagrange polynomial f.

This final traversal along T can be implemented as a simple recursive algorithm, which can be shown to have complexity $\mathcal{O}\big(M(n)\log(n)\big)$; see chapter 10 of [36]. Combining this with the aforementioned cost of multi-point evaluation of p', we have that interpolation of polynomials from n points can be done with total cost $\mathcal{O}\big(M(n)\log(n)\big)$.

4.4 Gao's decoding algorithm

As discussed in Chapter 3, in order to be able to recover the proof polynomial from the results of delegated computations, we need a way to decode the coefficients of a polynomial $\hat{h}$ from possibly erroneous evaluations of $\hat{h}$. More precisely, the problem we are interested in is as follows. Let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be distinct points in R. We are given the values $\beta_1, \beta_2, \ldots, \beta_n$ in R. We want to decode the coefficients $\phi_0, \ldots, \phi_{k-1}$ of a polynomial $f \in R[x]$ of degree $k-1 \leq n$, such that $f(\alpha_i) = \beta_i$ for at least $k-1+\lceil d/2\rceil$ pairs (α_i, β_i), where $d = n-(k-1)$.

An algorithm for solving this problem, due to Gao [10], is as follows:

1. Compute the polynomial $g_0 = \prod_{i=0}^{n}(x-\alpha_i)$ using a subproduct tree.

2. Interpolate the unique polynomial $g_1 \in R[x]$ of degree at most $n-1$ such that $g_1(\alpha_i) = \beta_i$ for $i = 1, \ldots, n$.

3. Apply the extended Euclidean algorithm (EEA) to g_0 and g_1. Stop when the remainder g has degree less than $(n+k)/2$. At that stage, the EEA will have produced polynomials satisfying

$$u(x)g_0(x) + v(x)g_1(x) \;=\; g(x)\ .$$

4. Divide g by v to obtain

$$g(x) \;=\; f_1(x)v(x) \;+\; r(x)$$

where $\deg(r) < \deg(v)$.

5. If $r(x) = 0$ and f_1 has degree less than k, then output the coefficients of f_1. Otherwise, more than $(d-1)/2$ errors occurred; in that case output "Decoding error".

For a proof of the algorithm's correctness see [10].

Recalling the algorithmic tools from previous sections, we see that steps (1), (2), and (4) together have complexity $\mathcal{O}\big(M(n)\log(n)\big)$. That leaves the extended Euclidean algorithm (EEA) of step (3). The details of an efficient implementation of the EEA are somewhat involved, and won't be covered here. However, the EEA also admits an $\mathcal{O}\big(M(n)\log(n)\big)$ implementation; for details see Chapter 11 of [36]. In conclusion, we have that Gao's algorithm—using a fast implementation of the EEA—provides a way to solve the problem of decoding polynomials from n possibly erroneous evaluation-points in time $\mathcal{O}\big(M(n)\log(n)\big)$, so long as the number of errors is at most $\lfloor d/2 \rfloor$. This completes the toolbox of near-linear-time algorithms we need for implementing delegable inference.

Chapter 5

Implementation

5.1 Efficient modular arithmetic

As discussed in Chapter 3, when using the DC algorithm we assume that the factor graph being operated on takes values from a finite field $\mathbb{F}$. In Section 3.6 we saw that we can take $\mathbb{F}$ to be $\mathbb{Z}_p$ for some prime p. Thus, in implementing DCA, an efficient implementation of modular arithmetic is crucial. To that end, we chose to implement Montgomery arithmetic with Montgomery modulus $q = 2^{32}$. In this section, we give an overview of our implementation of Montgomery arithmetic. However, a detailed exposition of Montgomery arithmetic itself would require more space than is reasonable to use here; in this section we recall only some essential ideas. For more details on Montgomery arithmetic, see *e.g.* the original paper by Montgomery [23] or any of the numerous expositions available online.

Modular arithmetic involves division by the modulus, which in general is computationally expensive. However, when the modulus is a power of 2, those divisions reduce to bit-shift operations, which are very efficient. Consider the problem of performing arithmetic modulo p. Let q be a power of 2, greater than p, and coprime to p. The high-level idea in Montgomery arithmetic is to transform operands into something called *Montgomery form*, perform all arithmetic in Montgomery form, and finally transform the result back into conventional form. In Montgomery form, divisions only need to be done modulo q; this way, expensive divisions modulo p are avoided.

Encapsulating Montgomery parameters. In order to perform operations of Montgomery arithmetic, one needs to precompute a certain set $M(q, p)$ of parameters which depends only on q and p. Assuming that the Montgomery modulus q is fixed—for example to 2^{32}—we can denote this set of parameters by $M(p)$. Some subset of those parameters is needed in every

operation. There are multiple possible ways the parameters $M(p)$ could be made available in every operation. A very straightforward way would be to define all Montgomery operations as functions taking $M(p)$ as arguments. However, that would make the code more cumbersome, and would lead to all values in $M(p)$ being pushed onto the stack for every single arithmetic operation, which would degrade efficiency.

Instead, for each p, we encapsulated the Montgomery parameters $M(p)$ in a class $\texttt{MP}_p$, with each parameter a `const static` member of that calss. Montgomery operations were defined as template functions taking $\texttt{MP}_p$ as template argument, making any given parameter $\texttt{param} \in M(p)$ available as $\texttt{MP}_p\texttt{::param}$. This way the Montgomery parameters $M(p)$ are known at compile-time, and the compiler can produce for each p a function with $M(p)$ hard-coded into it; thereby removing both the need to pass arguments around in code and the need to push all parameters $M(p)$ onto the stack at every function-call. Finally, Montgomery-form scalars were encapsulated in the template class `Zp`, taking $\texttt{MP}_p$ as template argument. Operations on $\texttt{Zp<MP}_p\texttt{>}$ were defined in terms of the template functions instantiated with $\texttt{MP}_p$ as template parameter.

The above approach has the benefits of making code more concise, and of improving efficiency by obviating needless copying of parameters onto the stack. However, it comes at the cost of needing to define a separate class $\texttt{MP}_p$ for every modulus p. For details on how this was managed, see Section 5.5.

Selecting the Montgomery modulus. The modulus p determines how many unique elements there are in $\mathbb{F} = \mathbb{Z}_p$. Thus, for it to be possible to encode the proof polynomial (see Section 3.4), p must be greater than d, where d is the degree of the proof polynomial. And in order to make proof verification effective with few trials, we should have $p > kd$ for some $k \geq 2$. On the other hand, if p does not fit into a standard 32-bit or 64-bit integer datatype, performance would deteriorate greatly. Also, assuming each of the $d + 1$ coefficients of the proof polynomial takes at least 4 bytes to represent, having $p \approx 2^{31}$ and $d \approx 2^{30}$ would already give us a polynomial that takes 4 GiB of memory, which is within an order of magnitude of what fits into a typical computer's RAM. In the interest of efficiency, we chose the Montgomery modulus to be $q = 2^{32}$, and the primes p defining the fields $\mathbb{F}$ to be the primes slightly less than 2^{31}; with the understanding that it would be straightforward to extend the implementation to $q = 2^{64}$ and $p \approx 2^{63}$.

5.2 External library for polynomials

An efficient implementation of the delegable inference framework requires efficient algorithms for working with polynomials. Chapter 4 outlined such an algorithmic toolbox at a theoretical level. Six-subgraph [19] is an implementation of the Björklund-Kaski template applied to the problem of counting subgraphs in a host graph. As such, Six-subgraph also contains an implementation of the fast polynomial toolbox. In this project, rather than reimplement the wheel, we essentially extracted and copied those parts of Six-subgraph needed to implement the fast polynomial toolbox. In some more detail, the main things copied from Six-subgraph were the following:

(i) A class for polynomials.

(ii) Schönhage-Strassen multiplication, including an implementation of the underlying FFT.

(iii) Functions for polynomial quotient and remainder.

(iv) Functions for polynomial multipoint evaluation.

(v) Functions for polynomial interpolation.

(vi) Gao's decoding algorithm, and a fast implementation of the extended Euclidean algorithm upon which the decoding algorithm relies.

In addition, our implementation of modular arithmetic followed the corresponding implementation in Six-subgraph very closely.

It is worth noting that Six-subgraph included parallelized implementations of those parts of the polynomial toolbox that can be parallelized; including in the form of GPU kernels for critical subroutines such as FFT butterflies. The parallelized versions are likely many times faster than the single-threaded CPU-based ones, but would also have taken more effort to get working. Due to time limitations, only the simpler and less efficient single-threaded versions were used in this project.

5.3 Factor contraction

Recall the problem of contracting two factors f_a and f_b, as defined in Section 2.2, particularly Equation (2.2). For a software implementation of factor contraction, we set the following criteria:

(i) It must support modular arithmetic; for example by supporting tensors whose elements are instances of the class `Zp` discussed in Section 5.1.

(ii) It must be written in C or C++, or easy to integrate into C/C++ code with minimal overhead.

(iii) It should be open-source.

(iv) It should either provide GPU acceleration, or be easy to extend to provide such.

(v) It should either be actively developed, or simple enough to be easy for us to maintain ourselves.

There exist many libraries for working with factor graphs and contracting factors [28]. However, we were unable to find any libraries that satisfied all of the criteria mentioned above, and so we decided to write our own simple implementation. In this section, we outline the main features of our implementation of factor contraction.

Reducing factor contraction to matrix multiplication. To begin, it is useful to distinguish different subsets of variables involved in the contraction. Denote by E the set $(S_a \cup S_b) \setminus I_{ab}$ of indices of variables adjacent to f_a or f_b but external to the contraction. Let $E_a = E \setminus S_b$ and $E_b = E \setminus S_a$ denote the sets of external variables adjacent only to one of the factors, and let $E_{ab} = E \cap S_a \cap S_b$ denote the set of external variables adjacent to both factors. Let $J_a = I_{ab} \setminus S_b$ and $J_b = I_{ab} \setminus S_a$ denote the sets of contraction-*internal* variables adjacent only to one of the factors, and let $J_{ab} = I_{ab} \cap S_a \cap S_b$ denote the set of internal variables adjacent to both factors. Note that if the factor graph G is non-degenerate, then $J_{ab} = I_{ab}$ and $J_a = J_b = \emptyset$.

With these definitions, we can write a more detailed version of Equation (2.2). We set aside the relatively trivial case where $I_{ab} = \emptyset$ and consider only the case $I_{ab} \neq \emptyset$. For all $v_{ab} \in \mathcal{D}_{E_{ab}}$ and $v_a \in \mathcal{D}_{E_a}$ and $v_b \in \mathcal{D}_{E_b}$, we have

$$\begin{aligned} f_{ab}(v_{ab}, v_a, v_b) &= \sum_{w_{ab} \in \mathcal{D}_{J_{ab}}} \sum_{w_a \in \mathcal{D}_{J_a}} \sum_{w_b \in \mathcal{D}_{J_b}} f_a(v_{ab}, v_a, w_{ab}, w_a)\, f_b(v_{ab}, v_b, w_{ab}, w_b) \\ &= \sum_{w_{ab} \in \mathcal{D}_{J_{ab}}} \left(\sum_{w_a \in \mathcal{D}_{J_a}} f_a(v_{ab}, v_a, w_{ab}, w_a) \right) \left(\sum_{w_b \in \mathcal{D}_{J_b}} f_b(v_{ab}, v_b, w_{ab}, w_b) \right) \\ &= \sum_{w_{ab} \in \mathcal{D}_{J_{ab}}} f'_a(v_{ab}, v_a, w_{ab})\, f'_b(v_{ab}, w_{ab}, v_b)\,. \end{aligned} \tag{5.1}$$

From Equation (5.1) we observe that after summing over singly-connected internal variables to obtain f'_a and f'_b, we have for each v_{ab} what is essentially a matrix multiplication. The operand matrices $f'_a(v_{ab}, \cdot)$ and $f'_b(v_{ab}, \cdot)$ have dimensions $|\mathcal{D}_{E_a}| \times |\mathcal{D}_{J_{ab}}|$ and $|\mathcal{D}_{J_{ab}}| \times |\mathcal{D}_{E_b}|$ respectively. The resulting matrix $f_{ab}(v_{ab}, \cdot)$ forms a size $|\mathcal{D}_{E_a}| \times |\mathcal{D}_{E_b}|$ sub-array of the factor f_{ab}. In this way, factor contraction can be reduced to a repeated combination of matrix multiplications and permutations of multi-dimensional arrays. We next discuss those permutations.

Improving efficiency by permuting factor data in memory. The efficiency of any computational task—such as matrix multiplication—depends strongly on how effectively the processor's registers and caches are used. For example, on the Intel Skylake microarchitecture, latencies for L1, L2, and L3 caches are 4, 12, and 44 clock cycles, respectively [15]. And locating and fetching data from main memory may well take an order of magnitude more cycles than accessing L3 cache. Hence, unless fetches from distant caches or main memory are infrequent, most processor cycles will be wasted due to IO latencies. Each fetch operation retrieves a contiguous block of data from around the target address; these blocks are called *cache lines*, and are typically on the order of 64 bytes in size [15]. The main way to avoid frequent fetches from distant caches is to arrange data in memory such that each fetched cache line contains as many as possible of the data soon to be needed by the processor.

In the abstract, it is convenient to think of the factors in a factor graph as multi-dimensional arrays. But in reality, the data constituting factors are stored linearly in memory. With the factors f'_a and f'_b, we have two arrays of sizes $|\mathcal{D}_{J_{ab}}| \cdot |\mathcal{D}_{E_a}| \cdot |\mathcal{D}_{E_{ab}}|$ and $|\mathcal{D}_{J_{ab}}| \cdot |\mathcal{D}_{E_b}| \cdot |\mathcal{D}_{E_{ab}}|$. However, the order of their dimensions may be suboptimal. To make the matrix multiplications in Equation (5.1) efficient, we permute the factor-data so as to have

(i) f'_a arranged in $|\mathcal{D}_{E_{ab}}|$ contiguous blocks of $|\mathcal{D}_{E_a}|$ contiguous blocks of size $|\mathcal{D}_{J_{ab}}|$,

(ii) f'_b arranged in $|\mathcal{D}_{E_{ab}}|$ contiguous blocks of $|\mathcal{D}_{E_b}|$ contiguous blocks of size $|\mathcal{D}_{J_{ab}}|$,

(iii) the shared dimensions in E_{ab} and J_{ab} ordered identically in f'_a and f'_b.

This way, for every $v_{ab} \in \mathcal{D}_{E_{ab}}$, all data needed for that matrix multiplication is available in contiguous blocks in both f'_a and f'_b: Each of the $|\mathcal{D}_{E_a}|$ rows of $f'_a(v_{ab}, \cdot)$ is in a block of size $|\mathcal{D}_{J_{ab}}|$, and each of the $|\mathcal{D}_{E_b}|$ columns of

$f'_b(v_{ab}, \cdot)$ is in a block of size $|\mathcal{D}_{J_{ab}}|$. Also, the row- and column-blocks are ordered compatibly, such that computing each element of the matrix $f_{ab}(v_{ab}, \cdot)$ reduces a direct dot product of two such blocks.

Summary of the implementations. As outlined above, the computationally expensive parts of factor contraction reduce to generalized transposition (permutation) and matrix multiplication. These were both given very straightforward but also inefficient implementations.

Matrix multiplication was implemented as the standard algorithm, with complexity $\mathcal{O}(nmk)$ for matrices of sizes $n \times m$ and $m \times k$. For anything close to an optimal implementation of matrix multiplication, one would need to consider techniques beyond just improving memory layout in RAM. However, the design of hardware-aware algorithms for efficient matrix multiplication is a broad topic in its own right, and beyond the scope of this thesis. Many existing libraries provide matrix multiplication routines with various performance characteristics; see *e.g.* [16] or [38]. The implementation of matrix multiplication in this project was left as a simple placeholder, with the understanding that it would be straightforward to replace with a more performant library implementation.

General transposition of factor-arrays was implemented as a simple linear-time algorithm. Given a permutation $\pi : \mathbb{N}^k \to \mathbb{N}^k$ and an array A with dimensions $(d_1, d_2, ..., d_k)$ and total volume $n = d_1 d_2 \cdots d_k$, the problem of transposing A to obtain an array B with dimensions $\pi(d_1, d_2, ..., d_k)$ can be reduced to permuting the data of A according to a certain bijection $\beta : \{0, 1, ..., n-1\} \to \{0, 1, ..., n-1\}$. Given an index $i \in \{0, 1, ..., n-1\}$, the value $j = \beta(i)$ is obtained by

1. computing the representation $(x_1, x_2, ..., x_k)$ of i in the positional number system with variable base $(d_1, d_2, ..., d_k)$,
2. computing the permutation $(y_1, y_2, ..., y_k) = \pi(x_1, x_2, ..., x_k)$,
3. computing the value $j \in \{0, 1, ..., n-1\}$ represented by $(y_1, y_2, ..., y_k)$ in the positional number system with variable base $\pi(d_1, d_2, ..., d_k)$.

A straightforward way to transpose A is then to map each element $A[i]$ to $B[\beta(i)]$. The cost of transposing A to B using this simple method is $\mathcal{O}(kn)$. As with matrix multiplication, more sophisticated implementations exist [16, 34]; replacing the above placeholder implementation of transposition with a more performant one was left for future work.

5.4 Reduction and reconstruction via Chinese remaindering

In order to apply the DC algorithm to a factor graph G with *rational* values, we need to reduce the problem of contracting G to a problem of contracting factor graphs $G_{\mathbb{Z}_{p_j}}$ with *integer* values modulo a prime p_j. Section 3.6 gave a theoretical description of that reduction; this section outlines the main aspects of how the reduction was implemented.

In the implementation, and in what follows, we assume that rational values are represented as 32-bit IEEE-754 floating-point numbers [13]. Extending the implementation to 64-bit IEEE-754 floating-point numbers would be straightforward.

Transforming from floating-point numbers to integers. The first step in the reduction is to transform the floating-point factor graph G to an integral factor graph $G_{\mathbb{Z}}$. While doing so, we also record information needed for converting back to the floating-point representation; we also record information needed to compute the upper bound M on the absolute values of the map $g_{\mathbb{Z}}$ of $G_{\mathbb{Z}}$. More precisely, for each factor f_i in G, we do the following:

1. Find the smallest exponent c_i such that $f_i(v) \cdot 2^{c_i}$ is an integer, for all $v \in \mathcal{D}_{S_i}$. This corresponds to finding for each value of f_i the exponent of its least significant bit, and then taking the negative of the smallest of those exponents.

2. Find the smallest exponent b_i such that $f_i(v) \leq 2^{b_i}$ for all $v \in \mathcal{D}_{S_i}$.

3. Multiply all values of f_i by 2^{c_i}.

Once each factor has been processed as above, the conversion exponent $\sum_i c_i$ and upper-bound exponent $\sum_i b_i$ are saved for later use. For further details on how float factor graphs are converted to multiprecision integer factor graphs, the reader may refer to the source code [17].

The factor graph $G_{\mathbb{Z}}$ thus produced may have values larger than what can be represented by standard 32- or 64-bit integer datatypes. To represent $G_{\mathbb{Z}}$, the GNU multiple precision arithmetic library (gmplib) [11] was used. The multiprecision numeric types provided by gmplib are implemented as arrays; this makes it impossible to pass them to—or return them from—functions *by value*. To overcome such inconveniences, a simple wrapper class `MPInt` was written to encapsulate gmplib's `mpz_t` type. Factor graphs such as $G_{\mathbb{Z}}$ can then be conveniently constructed using the class `MPInt` for their values.

Splitting the `MPInt`-valued factor graph $G_{\mathbb{Z}}$. As discussed in Section 5.1, the moduli p_j defining the finite fields $\mathbb{Z}_{p_j}$ were chosen to be 31-bit primes. Thus we need to reduce the problem of contracting $G_{\mathbb{Z}}$ to a problem of contracting multiple factor graphs $G_{\mathbb{Z}_{p_j}}$, for 31-bit moduli p_j. To that end, we compute the upper bound $M = 2^{\sum_{i=1}^{m} c_i} 2^{\sum_{i=1}^{m} b_i} |\mathcal{D}_U|$, and then we choose the moduli $p_1, p_2, ..., p_k$ by taking as many distinct primes just less than 2^{31} as needed so that $\prod_{j=1}^{k} p_j > 2M$. (Note: we must use $2M$ rather than M, in order to be able to recover negative values.) For each prime modulus p_j, we then construct the factor graph $G_{\mathbb{Z}_{p_j}}$ by taking all values of $G_{\mathbb{Z}}$ modulo p_j and transforming them to Montgomery form. The Montgomery-form values of $G_{\mathbb{Z}_{p_j}}$ are implemented using the class `Zp<MP`$_{p_j}$`>`. Each resulting factor graph $G_{\mathbb{Z}_{p_j}}$ can then be contracted using the DC algorithm.

Obtaining the final result by merging sub-results. Once the factor graphs $G_{\mathbb{Z}_{p_j}}, j = 1, 2, ..., k$ have been contracted, each resulting map $g_{\mathbb{Z}_{p_j}}$ is represented as a factor adjacent to exactly the boundary variables and of type `Zp<MP`$_{p_j}$`>`. The map $g_{\mathbb{Z}}$ is reconstructed by applying the Chinese remainder theorem (CRT) for each $v \in \mathcal{D}_B$. More precisely, for each $v \in \mathcal{D}_B$, the value $g_{\mathbb{Z}}(v)$ is reconstructed by computing the r.h.s. of Equation (3.9), with $a_j = g_{\mathbb{Z}_{p_j}}(v)$.

In order to write the C++ function `crtReconstruct` that performs this reconstruction of $g_{\mathbb{Z}}$, the input factors $g_{\mathbb{Z}_{p_j}}$ need to have types that are known at compile-time. However, for distinct primes $p_i \neq p_j$, the types `Zp<MP`$_{p_i}$`>` and `Zp<MP`$_{p_j}$`>` are distinct; thus the factors $g_{\mathbb{Z}_{p_j}}, j = 1, 2, ..., k$ are also of different types. And the number k of input factors is not known until run-time. To circumvent this problem, each input factor $g_{\mathbb{Z}_{p_j}}$ is separately converted from Montgomery form to `uint_t` type; the `uint_t` factors can then be passed to `crtReconstruct` in a single vector.

Once the map $g_{\mathbb{Z}}$ has been computed, what remains to be done is to recover the signs of the values, and to convert the values from type `MPInt` back to `float`s. To recover signs, for each $v \in \mathcal{D}_B$, the value $g_{\mathbb{Z}}(v)$ is reinterpreted as $g_{\mathbb{Z}}(v) - 2M$ iff $g_{\mathbb{Z}}(v) \in [M, 2M]$, and otherwise left unchanged. Conversion back to floating-point numbers is done simply by multiplying each value $g_{\mathbb{Z}}(v)$ by the inverse $2^{-\sum_{i=1}^{m} c_i}$ of the "conversion factor", with intermediate results stored as values of gmplib's `mpf_t` type.

5.5 Managing execution-dependent types

As explained in Section 5.1, for each prime modulus p, it is necessary to create distinct instantiations of the template classes `MP`$_p$ and `Zp<MP`$_p$`>`. However,

there are unavoidable situations where the moduli p_j, $j = 1, 2, ..., k$ are not know until runtime, which poses problems, since then the types of various functions to be called are also not known until runtime. This section outlines those problems and how they were solved.

Generating definitions for Montgomery parameter classes. Firstly, for each p, a separate definition needs to be written for the class $\texttt{MP}_p$. Doing so manually would be tedious. Instead, we wrote a Python batch generation script in which the user manually defines only the desired list of prime moduli $\{p_j\}_{j=1,2,...,k}$, and the script then, for each p_j, computes the Montgomery parameters $M(p_j)$ and inserts them into a template string containing the source code for the definition of $\texttt{MP}_{p_j}$. The script then simply concatenates the instantiated template strings and writes them to a C++ source file. That file can then be `#include`'d wherever the definitions are needed.

Selecting template function instances at runtime. In situations where the modulus p is determined at runtime—for example when splitting an `MPInt`-valued factor graph (see Section 5.4) or when the modulus p is loaded from a file—we cannot directly write code that proceeds with computations using $\texttt{MP}_p$, since we do not know p at code-writing-time. To circumvent this problem, we use a "type demultiplexing" function `demuxMod`; see Listing 5.1. The `demuxMod` function essentially translates from the numeric modulus p to a call to a template function `f<`$\texttt{MP}_p$`>` instantiated with $\texttt{MP}_p$. The desired template function is passed to `demuxMod` indirectly, via a string identifier (`cmd` in Listing 5.1), and then called via `demuxCmd` (see Listing 5.2).

The exact source code of `demuxMod` cannot be written in advance, since it depends on the list of possible moduli, which should be specifiable by the user. This is once again easily solved with very simple metaprogramming: We wrote a Python script that takes the list of possible moduli and generates the source code file of `demuxMod`.

```
1  std::string demuxMod(
2      std::vector<std::string> workFiles,
3      std::string cmd)
4  {
5      FileHandler fh(workFiles[0]);
6      uint_t modulus;
7      assert (loadParamFromFile<uint_t>(fh, "modulus", modulus));
8      if (modulus == 2147483647) {
9          return demuxCmd<MP2147483647>(cmd, workFiles);
10     } else if (modulus == 2147483629) {
11         return demuxCmd<MP2147483629>(cmd, workFiles);
12     } else if (modulus == 2147483587) {
13         return demuxCmd<MP2147483587>(cmd, workFiles);
```

```
14      } ...
```

Listing 5.1: First 13 lines of the procedurally generated code for the "type demultiplexing" function `demuxMod`.

```
1  template <typename P>
2  std::string demuxCmd(
3      std::string cmd,
4      std::vector<std::string> workFiles)
5  {
6      assert (workFiles.size() >= 1);
7      if (cmd == "contract")
8          return contractMod<Zp<P>>(workFiles[0]);
9      else if (cmd == "reduce")
10         return crtReduce<P>(workFiles[0]);
11     else if (cmd == "decode")
12         return decodeProofCoefficients<Zp<P>>(workFiles[0]);
13     else if (cmd == "verify") {
14         assert (workFiles.size() == 2);  // factorgraph, proof
15         return verifyProof<Zp<P>>(workFiles[0], workFiles[1]);
16     } ...
```

Listing 5.2: First 15 lines of the "command demultiplexing" function `demuxCmd`.

Interfacing through files. The functions to be called via `demuxMod` vary in type signatures. This would make it very inconvenient to pass all possibly-needed arguments through `demuxMod` and `demuxCmd`. Another problem is that some functions—such as `splitMPIntFG`, which reduces one factor graph of type `MPInt` into multiple factor graphs of types `Zp<MP`$_{p_i}$`>`—may need to return a *set* of modular factor graphs, where each factor graph has a different modulus and thus is of different type. However, C++ does not support collections of objects with heterogeneous types defined at runtime (like for example Python lists). To overcome these problems, the interfaces of `demuxMod`, `demuxCmd`, and the functions called through them were changed so as to take *file names* as arguments and return a vector of *file names*. The functions called through `demuxMod` then simply load their parameters from files, and write their results to files.

The results of many of the functions in question are contracted factors or factor graphs, which would need to be written to file anyway; in this sense, this change of interfaces was entirely apposite. On the other hand, abstracting away type incompatibilities by hiding all data in files could be seen as a dirty hack to circumvent the C++ language's type system. Influenced by that perspective, the questionable solution of interfacing through files was restricted to only a necessary handful of relatively simple and high-level

functions, for which losing the benefits of a type system—such as efficiency and guarantees of correctness—did not matter very much. (Being high-level, they account for only a tiny fraction of the computational effort; being simple, their correctness is relatively easy to ensure manually.)

To enable the required file operations, minimalistic tools were implemented for loading and storing parameters to files, and for serializing and deserializing various kinds of scalars, factors, and factor graphs. Another option would have been to integrate an existing library for such file operations; this option was forgone partly in order to maintain full control over the software, and partly to avoid unnecessary complexity in the codebase. In retrospect, even the minimalistic tools ended up being more complex than expected, quite brittle, and still lacked some desirable features such as compatibility with some standard format like JSON, XML, or YAML. Improving or replacing the input-output processing tools with a more refined library was left for future work.

Chapter 6

Experiments and results

In this chapter, we describe how we analysed the performance of our implementation of the DC algorithm and tested its correctness. Unless noted otherwise, in this chapter "DC algorithm" refers to our *implementation* of the algorithm, rather than to the algorithm's theoretical definition. Performance was evaluated on two computational tasks that are amenable to being solved via reduction to factor graph contraction, namely matrix multiplication and computing the permanent of a matrix. The reductions of these computational tasks to factor graph contraction are described in Section 6.1. The algorithm's performance is analysed in Section 6.2.

In addition to extensive unit testing of the software as a whole, the correctness of the evaluation- and decoding subroutines of the DC algorithm were tested by generating random instances of the computational tasks, and comparing the outputs of DC to the outputs of corresponding conventional solution-algorithms. The Chinese remaindering -based reduction and reconstruction were not needed for this, as all task instances were already in modular form. Correctness of the reduction and reconstruction subroutines were tested with unit- and integration tests, which are not covered in this thesis.

6.1 The computational tasks

This section describes the two computational tasks on which performance was evaluated, and which were used to test the software's correctness. We will call the tasks "MatMul" and "MatPerm", for matrix multiplication and matrix permanent. For both tasks, we implemented a simple transformation from problem instances—in the form of one or two matrices—to factor graphs. We begin with the MatMul task.

Matrix multiplication. An instance of the MatMul problem consists simply of a pair of $n \times n$ matrices, with entries in `Zp<MP>` for some set of Montgomery parameters `MP`. We reduce a pair of $n \times n$ matrices to a factor graph G as follows:

1. Let G have three variables, a, b, c, each with domain size n.
2. Define two factors f_1 and f_2, each representing a matrix, such that $S_1 = \{a, b\}$ and $S_2 = \{b, c\}$.
3. For the final factor resulting from contracting f_1 and f_2 to correspond to the product of the matrices, set the boundary and cutset to be $\{a, c\}$.

For more details on reducing matrix multiplication to factor graph contraction, see Section 2.2.

Matrix permanent. An instance of the MatPerm problem is a square matrix with entries in `Zp<MP>` for some set of Montgomery parameters `MP`. The problem of computing matrix permanents is #P-hard [35]. For an $n \times n$ matrix M, Ryser's formula [31] provides an $\mathcal{O}(n^2 2^n)$ method of computing the permanent:

$$\operatorname{perm}(M) \;=\; (-1)^n \sum_{S \subseteq \{1,2,\ldots,n\}} (-1)^{|S|} \prod_{i=1}^{n} \sum_{j \in S} M_{i,j} \;. \tag{6.1}$$

For a given $n \times n$ matrix M, we reduce M to a factor graph G such that contracting G corresponds to evaluating (6.1), as follows:

1. Let G have n binary variables v_i, with $\mathcal{D}_i = \{0, 1\}$ for all $i \in U = \{1, 2, \ldots, n\}$.
2. Let there be n factors, each connected to all variables. Construct each factor f_k by computing, for all w in $\mathcal{D}_U = \{0, 1\}^n$,

$$f_k(w) \;=\; \left(\sum_{j : w_j = 1} M_{k,j} \right) \cdot \begin{cases} -1 & \text{if } w_k = 1 \\ 1 & \text{otherwise.} \end{cases}$$

3. Set the boundary of G to be empty. Assign half the variables to the cutset. Due to symmetry, it does not matter which half of the variables are assigned to the cutset.

We do not know whether this reduction of matrix permanent to factor graph contraction is optimal: Perhaps there exists a reduction which yields factor graphs with lower contraction complexity. We leave that question outside the scope of this thesis, with the understanding that, given a better reduction, it would be straightforward to apply our implementation of DCA to factor graphs produced by that reduction.

6.2 Performance

In this section we summarize the software's performance on the MatMul and MatPerm tasks. For both tasks, we first compared the total execution times of proof construction to proof verification, as functions of the matrix row-count n. By "proof construction" we refer to the combination of computing the required evaluations via the evaluation algorithm (see Section 3.3) and decoding the proof polynomial's coefficients. For MatPerm, we considered both the case where the cutset is left empty and the case where the cutset consists of half the variables. Next, to get a finer-grained understanding of what is going on during execution and of where the resource bottlenecks are, we also tracked times and call counts separately for some of the more important functions: Runtimes for the prover's subroutines are shown in Figures 6.2 and 6.5, and Tables 6.1 and 6.2 list call-counts and times of all main functions, counted separately for different input sizes.

Execution times were measured simply using a global stack of time-points[1], and call-counts were similarly recorded using a global `std::map`. As the software is single-threaded, processor time and wall time are practically identical; we used `std::chrono::steady_clock`. Due to large volumes of calls to low-level routines with small input sizes, we ran into the problem that the instrumentation code itself took up a large fraction of total runtime. To mitigate this, we made the amount of instrumentation code controllable by preprocessor flags, compiled multiple versions of the main executable with different levels of instrumentation enabled, and measured the runtimes of higher-level functions—*i.e.* functions closer to the root of the directed graph defined by the call relation—using executables with lower-level instrumentation disabled. This greatly reduced the instrumentation-induced overhead, but did not completely eliminate it. Notably, for functions that were called many times with small inputs, total times are still at least slightly overestimated. One can see this for example in Table 6.1, where the estimated times

[1]Upon entry into an instrumented function, the start-time is pushed onto the stack, and upon return that time is popped and compared to the end-time to obtain the duration.

spent in the prover's subroutines sum to slightly *more* than the estimated total time of the prover.

All experiments were run on a 2.2 GHz Intel i7-8750H CPU [14]; source code was compiled with g++ version 9.4.0 with optimization set to `-O4`.

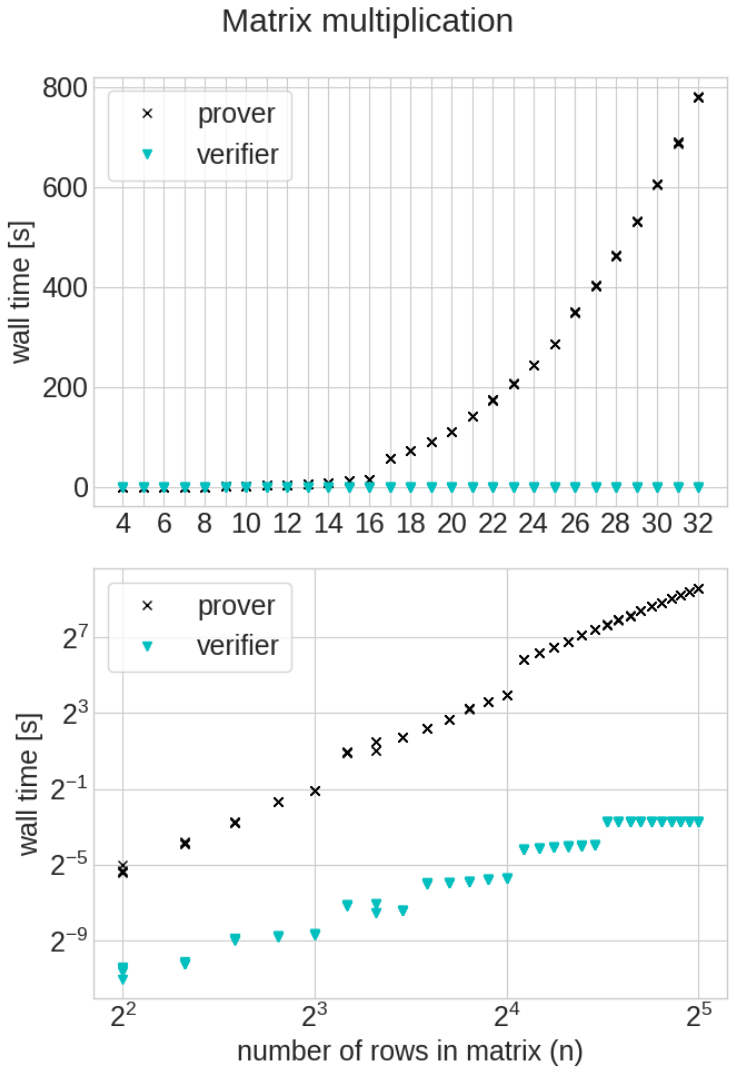


Figure 6.1: Times for proof construction and -verification on the MatMul task. Top: both axes linear. Bottom: both axes logarithmic.

Matrix multiplication. From Figure 6.1 we immediately note that the DC algorithm (as implemented) does not scale beyond very small inputs.

Already at $n \approx 9$ execution time exceeds one second, and grows superlinearly. From the log-log plot in Figure 6.1, one could estimate the growth rate to be very roughly n^5, assuming it is polynomial. From Figure 6.2 we see that the majority of that time is spent on interpolations performed by the prover while computing evaluations of the proof polynomial. Theorem 1 would predict that $\tilde{\mathcal{O}}(D^{k+1}m)$ evaluations are required, each imposing a cost of $\tilde{\mathcal{O}}(k\,\|G\|)$ arithmetic operations due to interpolations, for a total cost of $\tilde{\mathcal{O}}(D^{k+1}mk\,\|G\|)$. In the context of matrix multiplication we have $D = n$, $k = 2$, $m = 2$, and $\|G\| = 2n^2$, and thus a total cost of $\tilde{\mathcal{O}}(n^5)$, which is roughly in agreement with the empirical results.

Figure 6.1 shows that the verification times are negligible compared to proof construction times. This is to be expected: for matrix multiplication, verification requires only one evaluation, whereas proof construction requires $\tilde{\mathcal{O}}(n^3)$ evaluations. It is perhaps interesting to note that verification of matrix multiplication can be done using one $\tilde{\mathcal{O}}(n^2)$ evaluation, while even the asymptotically fastest known matrix multiplication algorithms have complexity greater than $\mathcal{O}(n^{2.37})$ [1, 7].

From Table 6.1 we see that most of the time is spent on interpolation with small inputs. These are the interpolations described in step (1.) of the evaluation algorithm (see Section 3.3). The two calls with input size n^2 are for interpolating the polynomials $\hat{\ell}_1$ and $\hat{\ell}_2$. The single interpolation call with large input is made while decoding the proof polynomial. It is reasonably straightforward to check that the call counts and input sizes are in accordance with theory. We conclude that the algorithm's inefficiency is not due to doing *unnecessary* work, but due to doing *necessary* work *inefficiently*. In particular, all evaluations of the proof polynomial are independent, and could in principle be parallelized. Setting aside possible overhead from provisioning multiple processors and collating results, parallelizing proof-evaluations would speed up execution by a factor roughly equal to the degree of the proof polynomial; in the case of matrix multiplication, a factor of $(n^2-1)\cdot 2\cdot(n-1)$. Furthermore, within each proof-evaluation, the interpolations performed on different factors are independent of one another, and could also be parallelized. Our single-threaded implementation leaves all such low-hanging optimization fruit unpicked.

We note that the execution times jump after $n = 8$ and $n = 16$. These jumps are due to how algorithms in the fast polynomial toolbox (see Chapter 4) pad inputs of size n to be of size $2^k \geq n$, using the smallest possible $k \in \mathbb{N}$. Theory would predict, and we checked empirically for $k \in \{3, 4, 5\}$, that for any $n_1, n_2 \in [2^k + 1, 2^{k+1}]$, calls to these algorithms cost equal time per call for inputs of size n_1 as for inputs of size n_2. Hence for the MatMul problem, where most of the time is spent on calls to algorithms of the polynomial

toolbox with input size n, execution times jump as n goes from being less than or equal to 2^k to being greater than 2^k, for all $k \in \mathbb{N}$. And since the algorithms of the fast polynomial toolbox are near linear time—*i.e.* linear up till logarithmic factors—the jumps will be by a factor not much greater than two.

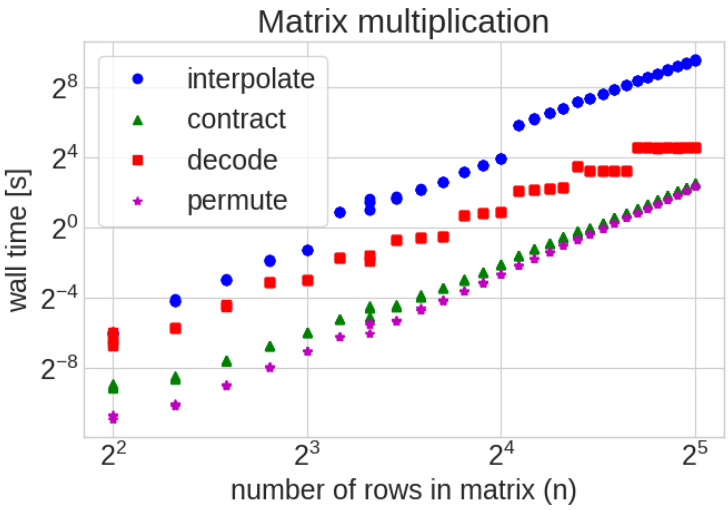


Figure 6.2: Times for main subroutines of proof construction on the MatMul task. Both axes are logarithmic.

Matrix permanent. Total execution times for the prover and verifier, with the cutset $C_{\text{even}} = \{i \in U : \text{rem}(i, 2) = 0\}$ containing half of the variables, are shown in Figure 6.3. Figure 6.4 shows prover and verifier times with an empty cutset. As expected for the problem of computing permanents, the complexity is exponential. From Figure 6.5 and Table 6.2 we see that—as for MatMul—the majority of the time is spent in interpolations. Theorem 1 predicts that the cost of these interpolations—for a MatPerm factor graph as described in Section 6.1—would be $\tilde{\mathcal{O}}\left(D^{k+1}mk\,\|G\|\right) = \tilde{\mathcal{O}}\left(2^{(n/2)+1}n\frac{n}{2}n2^n\right) = \tilde{\mathcal{O}}\left(n^3 2^{1.5n}\right)$. That prediction appears to be supported by the data in Figure 6.3: If one were to fit only an exponential to the data, a reasonable rough estimate would be $c2^{2.0n}$, for some constant c. We note that if one were to fit such an exponential to the prover's times, the constant c would be very large: The prover's execution time reaches one second already at $n \approx 7$. In comparison, a simple unoptimized algorithm for computing permanents, based on Ryser's formula, was found to take less than 80μs for $n = 7$, and reached one second only when $n = 21$.

Since verification requires only one evaluation, and proof construction requires $\tilde{\mathcal{O}}\left(D^{k+1}m\right) = \tilde{\mathcal{O}}(n2^{n/2})$ evaluations, theory predicts that the verifier's times grow slower than the prover's times by a factor of roughly $n2^{0.5n}$. Looking at the verifier's times in Figure 6.3, a reasonable estimate of the slope might be 1.3. And so the empirical results seem to be more or less in accordance with theory: 1.3 is not too far from 0.5 less than the slope of 2.0 estimated for the prover.

When the cutset is left empty both proof construction and verification degenerate to contracting the entire factor graph; in this case the proof polynomial has degree zero, *i.e.* it is a scalar. In this situation, there is no benefit to using the DC algorithm, as proof construction and -verification cost the same amount of resources, and so it is cheaper for the verifier to simply compute the desired result directly. This can readily be seen in Figure 6.4. What is perhaps not obvious *a priori* is the size of the overhead induced by having a non-empty cutset; that is, the overhead due to evaluations of the proof polynomial, decoding its coefficients, and recovering the result by summing over the cutset. By comparing Figures 6.3 and 6.4, we see that that overhead is large: It appears that the asymptotic complexity of proof verification with C_{even} is roughly the same as with an empty cutset, but the multiplicative factor in the time complexity for C_{even} is impractically large.

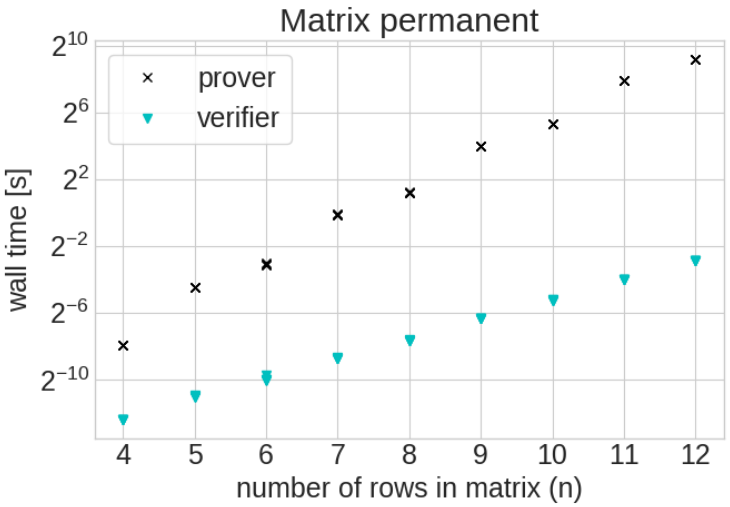


Figure 6.3: Times for proof construction and -verification on the MatPerm task, with cutset containing half of all variables. Time axis is logarithmic.

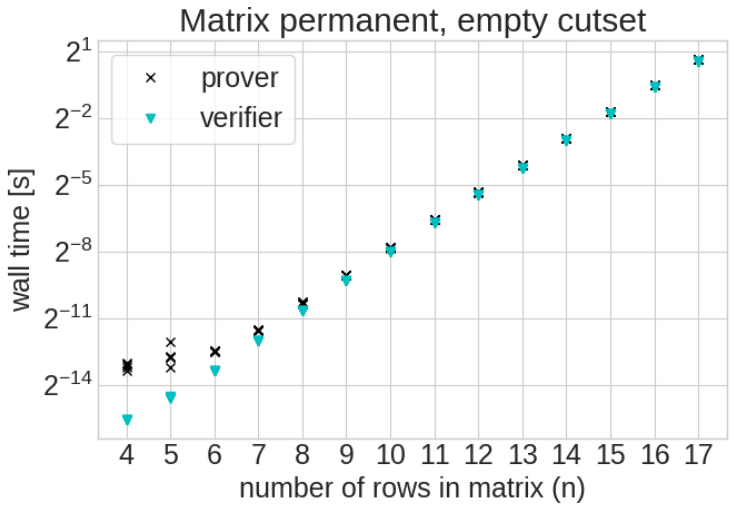


Figure 6.4: Times for proof construction and -verification on the MatPerm task, with empty cutset. Time axis is logarithmic.

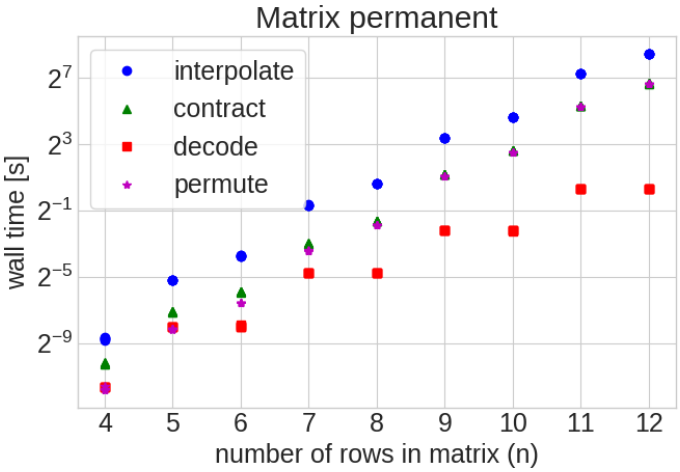


Figure 6.5: Times for main subroutines of proof construction on the MatPerm task, with cutset containing half of all variables. Time axis is logarithmic.

Table 6.1: Statistics for the MatMul task: Call-counts and times for proof construction, for the main subroutines of proof construction, and for verification. Calls to each function are counted separately for different input sizes. For contraction, "Input size" is the cost of the contraction, as in Section 2.2. "Permute" refers to permutations performed during the proof evaluation (to enable efficient interpolation and writing of coefficient data), and does not include permutations performed under factor contraction. For "Permute", "Input size" is the volume of the factor to be permuted.

n	Subroutine	Input size	Count	Time [s]
8	prover		1	0.4696 s
	interpolate	8	14128	0.3171 s
		64	2	0.0022 s
		883	1	0.1073 s
	contract	8	883	0.0030 s
		64	1766	0.0134 s
	decode	882	1	0.1270 s
	permute	64	1766	0.0076 s
	verifier	882	1	0.0025 s
16	prover		1	15.7251 s
	interpolate	16	244832	13.8511 s
		256	2	0.0188 s
		7651	1	1.5373 s
	contract	16	7651	0.0256 s
		256	15302	0.2159 s
	decode	7650	1	1.8907 s
	permute	256	15302	0.1592 s
	verifier	7650	1	0.0187 s
32	prover		1	780.9190 s
	interpolate	32	4059328	746.0560 s
		1024	2	0.1413 s
		63427	1	16.0944 s
	contract	32	63427	0.2161 s
		1024	126854	5.6726 s
	decode	63426	1	23.7310 s
	permute	1024	126854	5.0870 s
	verifier	63426	1	0.1520 s

Table 6.2: Statistics for the MatPerm task, with cutset containing half of all variables. See Table 6.1 for details on what various fields mean.

n	Subroutine	Input size	Count	Time [s]
4	prover		1	0.0040 s
	interpolate	2	1200	0.0022 s
		4	2	0.0000 s
		25	1	0.0002 s
	contract	4	75	0.0002 s
		8	100	0.0003 s
		16	100	0.0004 s
	decode	24	1	0.0003 s
	permute	8	100	0.0001 s
		16	100	0.0002 s
	verifier	12	1	0.0002 s
8	prover		1	2.3237 s
	interpolate	2	923520	1.5031 s
		16	4	0.0002 s
		481	1	0.0286 s
	contract	16	3367	0.0164 s
		32	3848	0.0205 s
		64	3848	0.0377 s
		128	3848	0.0775 s
		256	3848	0.1671 s
	decode	480	1	0.0366 s
	permute	32	3848	0.0138 s
		64	3848	0.0305 s
		128	3848	0.0694 s
		256	3848	0.1576 s
	verifier	120	1	0.0050 s
12	prover		1	590.8700 s
	interpolate	2	219518208	352.0360 s
		64	6	0.0039 s
		4537	1	0.9907 s
	contract	64	49907	0.8613 s
		128	54444	1.1330 s
		256	54444	2.4007 s
		512	54444	5.2226 s
		1024	54444	11.4301 s
		2048	54444	25.0237 s
		4096	54444	54.5900 s
	decode	4536	1	1.2519 s
	permute	128	54444	0.9882 s
		256	54444	2.2322 s
		512	54444	5.0418 s
		1024	54444	11.1727 s
		2048	54444	24.6087 s
		4096	54444	53.9195 s
	verifier	756	1	0.1355 s

Chapter 7

Conclusion

Factor graphs are a versatile tool with a wide range of applications from multilinear algorithm design [2] to the study quantum many-body systems [26]. Exact inference in factor graphs is a computationally challenging problem, which motivates the development of inference algorithms which can be robustly delegated to extensive—and possibly unreliable—compute infrastructure.

In this thesis, we implemented Karimi *et al.*'s delegable inference framework [18] for factor graphs. Our goal was to write an implementation that is correct, efficient, easily extensible, and allows for empirically testing the theoretical properties of the framework. To that end, we implemented the framework as a C++ library, and made it open-source. The library provides facilities for efficiently working with factor graphs over finite fields, and—making use of a library written by Kaski [19]—implements the algorithms for evaluating, decoding, and verifying proof polynomials, which together constitute Karimi *et al.*' framework. Our inital hope was to provide support for GPU acceleration, but we fell short of that goal due to time constraints; the implementation is single-threaded and runs on CPU.

We tested the correctness of our implementation by checking that it's outputs matched those of corresponding conventional algorithms, on procedurally generated random instances of the tasks of computing matrix permanents and -products. The probability of *accidentally* producing the correct result on hundreds of independent computational tasks is very low. We conclude that while the software is very unlikely to be completely free of bugs, at least the remaining bugs probably do not compromise the software's correctness on the tasks of computing matrix products or permanents. (Our suite of unit tests gives us some further confidence in the general correctness of the software.)

We evaluated the implementation's performance on two computational

tasks; namely matrix multiplication and computing the permanent of a matrix. We found that the empirical results were broadly in agreement with theoretical predictions. Notably, the cost of proof verification appeared to scale so as to remain a negligible fraction of the cost of proof construction; with extrapolated asymptotic behavior at least roughly matching the framework's theoretical properties. However, those asymptotic complexities are still high—$\tilde{\mathcal{O}}(n^5)$ for proof construction for matrix multiplication—and we also found that the constant factors in the complexities were quite dire: proof construction times exceeded one second already at $n = 9$ for matrix multiplication and at $n = 7$ for matrix permanent. We conclude that, in its current single-threaded form, our implementation does not begin to scale to practical applications.

We identified the performance bottleneck to be in large numbers of independent calls to polynomial interpolation routines with small inputs. An obvious way to improve performance would be to parallelize those calls. For matrix multiplication, the serial time could in principle be cut down to $\tilde{\mathcal{O}}(n^2)$ by parallelizing all $\tilde{\mathcal{O}}(n^3)$ evaluations of the proof polynomial, with the obvious caveat that, for large n, it may in practice be impossible to provision that many parallel processors. However, even on a single GPU one could get a large constant-factor speedup relative to our single-threaded CPU implementation: Given that a typical NVIDIA GPU can launch 32 concurrent kernels [25] and has a clock frequency of 1.6 GHz, and a typical Intel CPU runs at 2.2 GHz, a naive estimate of that speedup-factor would be $32 \cdot \frac{1.6}{2.2} \approx 23$.

Performance could also be improved by optimizing the evaluation algorithm so as to minimize the number of factor transpositions it performs when interpolating the proof polynomial. And since for most inference tasks it is difficult to determine the optimal contraction order, incorporating procedures for searching for efficient contraction orders would also be worthwhile. But ultimately, in order to reap the benefits of the potential for massive parallelism afforded by Karimi *et al.*'s framework, it would be necessary to extend the software to run on many processors.

A key factor determining whether the implementation is easily extensible is the quality of the source code itself. While our code is mostly legible and more or less reasonably structured, we believe there is always room for improvement in those areas as well. Forks and pull requests on the online repository [17] are welcome.

Bibliography

[1] Alman, J., and Williams, V. V. A refined laser method and faster matrix multiplication. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms* (2021), 522–539.

[2] Austrin, P., Kaski, P., and Kubjas, K. Tensor network complexity of multilinear maps. In *10th Innovations in Theoretical Computer Science Conference, ITCS* (2019), vol. 124 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 7:1–7:21.

[3] Biamonte, J. Lectures on quantum tensor networks. *arXiv.org* (2019).

[4] Bidyuk, B., and Dechter, R. On finding minimal w-cutset. In *ACM International Conference Proceeding Series: Proceedings of the 20th conference on Uncertainty in artificial intelligence* (2004), vol. 70, pp. 43–50.

[5] Björklund, A., and Kaski, P. How proofs are prepared at camelot. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (2016), ACM, pp. 391–400.

[6] Chandrasekaran, V., Srebro, N., and Harsha, P. Complexity of inference in graphical models. *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence, UAI 2008* (2008), 70–78.

[7] Duan, R., Wu, H., and Zhou, R. Faster matrix multiplication via asymmetric hashing. *arXiv.org* (2022).

[8] Frey, B. J. Extending factor graphs so as to unify directed and undirected graphical models. In *UAI '03, Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence* (2003), Morgan Kaufmann, pp. 257–264.

[9] Gallager, R. G. *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1963.

[10] Gao, S. A new algorithm for decoding reed-solomon codes. In *Communications, Information, and Network Security* (2003), pp. 55 – 68.

[11] Granlund, T., and Others. The gnu multiple precision arithmetic library. `https://gmplib.org/`. [Accessed 29-6-2022].

[12] Harvey, D., and Van Der Hoeven, J. Polynomial multiplication over finite fields in time O(n log n). *Journal of the ACM 69*, 2 (2022).

[13] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.

[14] Intel. Intel core i7-8750h processor. `https://ark.intel.com/products/134906/intel-core-i78750h-processor-9m-cache-up-to-4-10-ghz.html`. [Accessed 16-12-2022].

[15] Intel. Intel 64 and ia-32 architectures optimization reference manual. `https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual.html`, 2022. [Accessed 15-12-2022].

[16] Jacob, B., Guennebaud, G., and Others. Eigen C++ template library for linear algebra. `https://gitlab.com/libeigen/eigen`. [Accessed 16-12-2022].

[17] Kähkönen, E. Delegable inference in graphical models. `https://github.com/kahkone/delegable_inference`, 2022.

[18] Karimi, N., Kaski, P., and Koivisto, M. Error-correcting and verifiable parallel inference in graphical models. *AAAI 2020 - 34th AAAI Conference on Artificial Intelligence* (2020), 10194–10201.

[19] Kaski, P. A delegatable and error-tolerant algorithm for counting of six-vertex subgraphs in a graph. `https://github.com/pkaski/six-subgraph`, 2017.

[20] Kaski, P. Engineering a delegatable and error-tolerant algorithm for counting small subgraphs. In *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX* (2018), SIAM, pp. 184–198.

[21] Koller, D., and Friedman, N. *Probabilistic graphical models : principles and techniques*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, 2009.

[22] Kschischang, F., Frey, B., and Loeliger, H.-A. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory 47*, 2 (2001), 498–519.

[23] Montgomery, P. L. Modular multiplication without trial division. *Mathematics of Computation 44*, 170 (1985), 519–521.

[24] Nielsen, M. A. *Quantum computation and quantum information*. Cambridge University Press, Cambridge, 2001.

[25] NVIDIA. Cuda c++ programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications`. [Accessed 12-12-2022].

[26] Orús, R. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics 349* (2014), 117–158.

[27] Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. 1988 Morgan Kaufmann Publishers, Inc., 1988.

[28] Psarras, C., Karlsson, L., and Bientinesi, P. The landscape of software for tensor computations. *ArXiv abs/2103.13756* (2021).

[29] Reed, I. S., and Solomon, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8*, 2 (1960), 300–304.

[30] Robeva, E., and Seigal, A. Duality of graphical models and tensor networks. *Information and Inference: A Journal of the IMA 8*, 2 (06 2018), 273 – 288.

[31] Ryser, H. J. *Combinatorial mathematics*. The Carus mathematical monographs. Mathematical Association of America, Washington, 1963.

[32] Schindler, F., and Jermyn, A. S. Algorithms for tensor network contraction ordering. *Machine Learning: Science and Technology 1*, 3 (2020).

[33] Schönhage, A., and Strassen, V. Schnelle multiplikation grosser zahlen. *Computing 7*, 3-4 (1971), 281–292.

[34] Springer, P., Su, T., and Bientinesi, P. HPTT: A high-performance tensor transposition C++ library. *ARRAY 2017 - Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (2017), 56–62.

[35] Valiant, L. The complexity of computing the permanent. *Theoretical Computer Science 8*, 2 (1979), 189–201.

[36] von zur Gathen, J., and Gerhard, J. *Modern computer algebra*, 3rd ed. Cambridge University Press, Cambridge, England, 2013.

[37] Wadden, J., and Skadron, K. Advances in gpu reliability research. In *Advances in GPU Research and Practice*. 2017, pp. 617–647.

[38] Walther, J., Koch, M., and Others. Boost linear and multilinear algebra library. `https://github.com/boostorg/ublas`. [Accessed 24-6-2022].

[39] Williams, R. Strong ETH breaks with merlin and arthur: Short non-interactive proofs of batch evaluation. *Electron. Colloquium Comput. Complex.* (2016), 2.