Algorithmic Engineering Aspects of Fast Zeta Transform-based Graph Colouring Algorithms

Mats Rydberg
rydbergmats@gmail.com

January 22, 2014

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Thore Husfeldt, thore.husfeldt@cs.lth.se

Examiner: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Abstract

The *chromatic polynomial* $\chi_G(t)$ of a graph G on n vertices is a univariate polynomial of degree n, passing through the points $(q, P(G,q))$ where $P(G,q)$ is the number of q-colourings of G. In this paper, we present an implementation of an algorithm by Björklund, Husfeldt, Kaski and Koivisto that computes $\chi_G(t)$ in time $O^*(2^n)$ and space $O^*(1.2916^n)$. We compare the performance of two different core libraries to eachother and show our performance against an implementation done by Haggard, Pearce and Royle from 2010. We also present the chromatic polynomials for a small Queen graph and a certain graph specified by Hillar and Windfeldt.

Keywords: graph colouring, algorithms, chromatic polynomial, master's thesis, parallelization

Acknowledgements

Thank you.

Contents

Chapter 1
Introduction

In this report, we will present experimental results from simulations and tests run on implementations of a few algorithms presented in a report by Björklund, Husfeldt, Kaski and Koivisto [2]. The two main results of that report is an algorithm for performing an algorithm, *the fast zeta transform*, with reduced space requirement from before, along with an application of this result on an algorithm for computing the *chromatic polynomial* of a graph. Our focus will lie on the chromatic polynomial algorithm, but in the process of constructing that program, we also completed implementations of three more direct applications of the fast zeta transform algorithm, solving the familiar *set cover*, *set partition* and *set packing* problems.

1.1 Report structure

We will begin with a theoretical introduction into the area of graph theory, present some previous scientific results, continue with the problem statement that has guided our work during this Master's Thesis, and then shortly mention some of our main results. In the following chapters, we will go through the working process in a more detailed manner, present reflections and in-depth results from the experiments that we have carried through. In the appendix we present selected parts of the code base which make up our implementations, and the chromatic polynomials of a few mentioned graphs.

1.2 Preliminaries

Here we present all necessary definitions and theoretical notation that will be used in this report. In the case of a ill-defined variable used in the following, we refer the reader to this section.

1.2.1 Set problems

In the *set cover* problem, we are given a set U, a family $\mathcal{F}$ of subsets of U and a natural number q, and are tasked with the problem of deciding whether there are q sets in $\mathcal{F}$ whose union equals U. In the *set partition* problem, we must decide whether there are *q pairwise disjoint* sets in $\mathcal{F}$ whose union equals U. In the *set packing* problem, we must decide whether there are q pairwise disjoint subsets of $\mathcal{F}$ whose union is a *subset* of U. Two sets are disjoint if they have no elements in common.

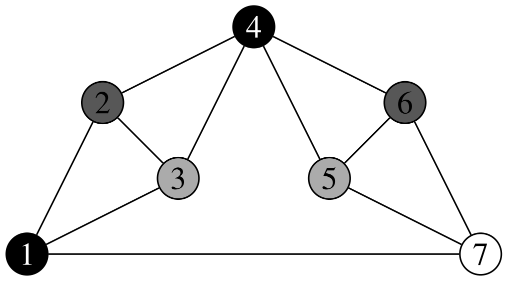
The *counting* versions of these problems asks instead *how many* distinct such selections of sets that exist. In this report, we will only discuss counting versions of these (and other) problems.

1.2.2 Chromatic polynomial

A *graph* is a set of two distinct abstract units called *vertices* and *edges*; an edge being a connection between two vertices. The vertices are contained in a vertex set V, and we say that the graph has *order n* if $|V| = n$. Similarly, we have an edge set E, and say that the graph has *size* $m = |E|$; the maximum size of a graph of order n is $\binom{n}{2} = n(n-1)/2$. We write a graph G as $G = (V, E)$. An edge $vw \in E$ implies $v \in V$ and $w \in V$, where v and w are the vertices connected by vw; v and w are said to be *adjacent*. In this report, edges are undirected, have multiplicity $leq1$ and are never loops. An example graph is found in figure 1.1.

In the *graph colouring* problem, we are given a graph $G = (V, E)$ and a natural number q, and tasked to determine whether there exists a mapping $\sigma : V \to [q]$ where $[q] = \{0, 1, \ldots, q\}$, such that $\sigma(v) \neq \sigma(w)$ for each $vw \in E$. In other words, σ is an assignment of one of *q colours* to each vertex such that no two adjacent vertices get the same colour; we call such an assignment a *proper q-colouring* of G. The counting version of this problem asks *how many* distinct such mappings exist; we call this number $P(G, q)$.

The *chromatic polynomial* $\chi_G(t)$ of the graph G is the polynomial of degree n in one indeterminate that passes through each point $(q, P(G, q))$. To determine $\chi_G(t)$, by for example specifying its coefficients, it suffices to determine the numbers $P(G, q)$ for each $q \leq n$ and then construct $\chi_G(t)$ by interpolation, as $n + 1$ points uniquely identifies an n-degree polynomial. Figure 1.1 presents the chromatic polynomial for a given graph.



$$\begin{aligned}\chi_{MS}(t) = \quad & t^7 - 11t^6 + 51t^5 \\ & -129t^4 + 188t^3 \\ & -148t^2 + 48t \\ \\ \chi_{MS}(4) = \quad & 384\end{aligned}$$

Figure 1.1: To the left, the Moser Spindle graph, MS, on $n = 7$ vertices and $m = 11$ edges, coloured in four different gray scales. To the right, its chromatic polynomial $\chi_{MS}(t)$.

1.2.3 Miscellaneous

The *chromatic number* $\chi(G)$ of the graph G is the smallest q for which $\chi_G(q) \neq 0$; it is the minimum amount of colours needed to produce a proper colouring of G. Graph *density* dE is the size of the graph over its maximum size.

1.3 Background

The chromatic polynomial was specified in 1912 by Birkhoff [1], who defined it for planar graphs with the intent on proving the Four Colour Theorem. Whitney extended its definition to general graphs in 1932 [12], and Tutte incorporated it into what is now known as the Tutte polynomial. See for example Ellis-Monaghan and Merino [3] for more in-depth analysis of the Tutte polynomial. In 2010, Haggard, Pearce and Royle [5] published a program, referred to in the following as **HPR**, to compute the Tutte polynomial for graphs using a deletion-contraction algorithm. HPR exploits the isomorphism of induced subgraphs to obtain good performance, and can easily handle many instances of non-trivial sizes. Using the fact that the Tutte polynomial encodes the chromatic polynomial (as well as other graph invariants), HPR is also designed to output $\chi_G(t)$. In 2011, Björklund, Husfeldt, Kaski and Koivisto [2] presented an algorithm to compute the chromatic polynomial in time $O^*(2^n)$ and space $O^*(1.2916^n)$, referred to here as the **BHKK** algorithm. The notation O^* hides polylogarithmic factors.

1.4 Problem statement

The main question is whether the BHKK algorithm performs well in practice. Irrefutably, it does provide theoretical improvements in the form of a better asymptotic bound on space usage. Intuitively, using less memory means we spend less time handling the memory, and this could reduce also the time consumption. But as the asymptotic bound stays solid, we can not know for sure how impactful such a reduction could be. Björklund *et al* also mentions that BHKK parallelizes well, and provides theoretical bounds suggesting how much of an improvement parallelization provides [2, p.10]. Haggard *et al* does not perform an asymptotic analysis of their algorithm, but simply shows experimental results showing it performs “well”. We will use HPR as a reference in our experiments, making us able to conclude that BHKK does provide improvement if it outperforms HPR. We attempt to answer the following direct questions:

1. Does BHKK outperform HPR as the order of the graph increases?
2. What is the maximum order of a graph that BHKK can compute the chromatic polynomial in human time for?

   With “human time”, we mean less than a month.
3. How does the graph size affect HPR and BHKK, respectively?
4. How much of an improvement can parallelization provide in practice?

1.5 Main results

The answers to the stated questions are, in short:

1. Yes, for $n > 21$.
2. 30.
3. BHKK is faster with increasing size; HPR is slower.
4. About 600%.

We find proof that BHKK does provide better asymptotic behaviour than HPR, for random graphs of quadratic size. We were able to compute the chromatic polynomial of a 30-order graph, but were unable to compute it for a 36-order graph.

Chapter 2

Approach

In this chapter we present an in-depth view of the algorithms studied, including special characteristics that aided us in their implementation. We describe the implementation process in some detail and present the external libraries that were employed. Finally we describe the setup of the experimental environment, including our measurement methods.

2.1 Studied algorithms

While our emphasis lies on the BHKK algorithm for computing chromatic polynomials, we will here briefly describe also a few other algorithms that were studied as a part of our work.

2.1.1 The fast zeta transform

Björklund *et al* [2] base their work on an improvement of the fast zeta transform, FZT. There are two versions of FZT, the down-zeta transform $f\zeta$ and the up-zeta transform $f'\zeta$, and they are defined as

$$f\zeta(X) = \sum_{Y \subseteq X} f(Y), \qquad f'\zeta(X) = \sum_{Y \supseteq X} f(Y)$$

for a function f defined for subsets X of an n-sized universe U. In [2, p.5] is described pseudo-code for an algorithm computing the fast zeta transform in time and space exponential in n.

In the *linear-space* fast zeta transform, we are given additional input in the form of a set family $\mathcal{F}$ that contains all defined inputs for the function f. In other words, $f(X) = 0$ if $X \notin \mathcal{F}$. The main idea is to split U into disjoint parts U_1 and U_2 with sizes n_1 and n_2 respectively, and to iterate over them separately. The algorithm is outlined in [2, sec.3], and uses $O^*(|\mathcal{F}|)$ space:

```
1. For each $X_1 \in U_1$ do:
    a) For each $Y_2 \in U_2$, set $g(Y_2) \leftarrow 0$.
    b) For each $Y \in \mathcal{F}$, if $Y \cap U_1 \subseteq X_1$ then set $g(Y \cap U_2) \leftarrow g(Y \cap U_2) + f(Y)$.
    c) Compute $h \leftarrow g\zeta$.
    d) For each $X_2 \subseteq U_2$, output $h(X_2)$ as the value $f\zeta(X_1 \cup X_2)$.
```

With $n_2 = |\mathcal{F}|$, the linear-space bound is guaranteed.

2.1.2 Coverings, partitionings and packings

The linear-space FZT has direct applications on some set problems, see section 1.2.1. For q-cover, the function f is $f(Y) = [Y \in \mathcal{F}]$, where $[P]$ denotes one if P is true, and zero otherwise, and we find the number of q-covers as

$$c_q(\mathcal{F}) = \sum_{X \in U} (-1)^{|U \setminus X|} f\zeta(X)^q. \tag{2.1}$$

In the algorithm, we in step 1d do not output the values $h(X_2)$, but sum them according to 2.1, outputting the resulting value c_q.

The number of q-partitions is the nth coefficient of the polynomial d_q, derived as in 2.1, but with $f = [Y \in \mathcal{F}]z^{|Y|}$ operating in a ring of polynomials, with indeterminate z, instead. Now we perform all arithmetics using polynomials instead of integers, a fact we will see having a substantial impact on time and space requirements.

Finally, for q-packings,

2.1.3 Chromatic polynomial

By adapting the linear-space FZT, an algorithm for the chromatic polynomial was designed in [2]. The space bound for the resulting algorithm is increased to $O^*(1.29153^n)$.

Our input is an undirected graph G on n vertices with m edges. The main subroutine counts the number of ways to colour G using q colours. This is done for $q = 0, 1, \ldots n$, yielding $n+1$ points (x_i, y_i). Interpolating on these points yields the chromatic polynomial $\chi_G(t)$.

The general idea of the algorithm uses the principle of inclusion-exclusion to count the proper q-colourings of G by actually counting the number of ordered partitions of V into *q independent sets*. The low space bound is obtained by splitting V into two disjoint sets V_1 and V_2 of sizes n_1 and n_2 respectively, where $n_1 = \lceil n\frac{\log 2}{\log 3}\rceil$ and $n_2 = n - n_1$, and then run iterations of subsets of V_1 and store values dependent on (subsets of) V_2 [2, sec. 5].

The full algorithm in pseudo-code as follows:

```
Step A. For $q = 0, 1, \ldots, n$, do
    1. Partition $V$ into $V_1$ and $V_2$ of sizes $n_1$ and $n_2$.
    2. For each $X_1 \subseteq V_1$, do
```

```
        a) For each independent $Y_1 \subseteq X_1$, do
               $h[V_2 \setminus N(Y_1)] \leftarrow h[V_2 \setminus N(Y_1)] + z^{|Y_1|}$
        b) For each independent $Y_2 \subseteq V_2$, do
               $l[Y_2] \leftarrow z^{|Y_2|}$
        c) $h \leftarrow (h\zeta') \cdot l$
        d) $h \leftarrow h\zeta$
        e) For each $X_2 \subseteq V_2$, do
               $r \leftarrow r + (-1)^{n-|X_1|-|X_2|} \cdot h[X_2]^q$
    3. Return coefficient $c_n$ of $z^n$ in $r$.
Step B. Construct interpolating polynomial $\chi_G(t)$ on points $(q, c_{nq})$.
Step C. Return $\chi_G(t)$.
```

Here, $N(Y)$ is the set of all vertices in G adjacent to at least one vertex in Y. The arrays h and l of size 2^{n_2} contain polynomials (initialized to zeroes), r is a polynomial. For a more detailed description, see [2, p 9].

In subsequent sections, we will refer to this algorithm as “the algorithm”, “the BHKK algorithm”, or simply “BHKK”.

2.2 Algorithmic aspects

There are some characteristics of the algorithm that deserve special mention, and that has had some influence on the way we have approached the task of implementing it. Here we present some of these characteristics.

2.2.1 Graph density

The algorithm in itself is designed in a way that allow for a smaller degree of complexity for *dense* graphs, that is, graphs with many edges. This is in contrast to many previously studied algorithms for graph colouring problems. And this is not only for very dense graphs, but the performance of the algorithm is in fact a function that is directly related to graph density, and consistently performs better for every additional edge to a graph. This follows directly from steps A(2)a and A(2)b above:

$$h[V_2 \setminus N(Y_1)] \leftarrow h[V_2 \setminus N(Y_1)] + z^{|Y_1|}$$

$$l[Y_2] \leftarrow z^{|Y_2|}$$

Recall that these lines will only be executed for *independent* sets Y_1 and Y_2. As graph density increases, fewer subsets of the vertex set V will be independent, and fewer of these lines will be executed, leading to the arrays h and l containing more zeros. This has a direct effect in reducing some additions and assignments, but more importantly has side effects in all subsequent steps, as arithmetic with zero-operands is (much) faster. The opposite is of course true for a *sparse* graph, for which the algorithm is significantly slower.

2.2.2 Multiplication algorithms

Much of the complexity of the whole algorithm comes down to how polynomial multiplication is performed. The most common operation is to multiply two polynomials of *small* degree ($\leq n$) but with *large* coefficients. This is because the degree of the polynomials increase as $O(n)$ while their coefficients increase as $O(2^n)$.

Trivially, a polynomial multiplication would be to expand over both operands' coefficients and cross-multiply them in a standard fashion. This is very inefficient, and many techniques have been developed to deal with this problem. In fact, the original issue has always been to multiply two large integers, but the most sophisticated results show methods that make use of polynomials for this purpose. The algorithm with the best asymptotic complexity is the Schönhage-Strassen algorithm, which is based on a Fast Fourier Transform, but it has a large overhead and becomes useful only for huge operands. The most go-to algorithm is the Toom-Cook (aka Toom-k) family, in which Toom-2 (aka Karatsuba) or Toom-3 are the most common.

The technique used in Toom-k for multiplying two large integers is to split them up in parts, introduce a polynomial representation of degree $k - 1$ for these parts (the parts are coefficients), evaluate the polynomials at certain points (the choice of points is critical), perform pointwise multiplication of the evaluated polynomials, interpolate the resulting points into a resulting polynomial, and finally reconstruct the integer from the coefficients of the resulting polynomial. This technique is easily translated for polynomial multiplication as well, where the first and last steps would be skipped.

For an overview and an in-depth description of these algorithms, see for example Wikipedia [13] [14] and Knuth [8, sec. 4.3.3], respectively.

As presented below in section 2.3, we employ a number of external libraries. One of the main reasons for this is to make use of existing implementations of the algorithms mentioned here, and not spend time implementing these ourselves. The external libraries provide the following support:

- The GMP library supports Karatsuba, Toom-3, Toom-4, Toom-6.5, Toom-8.5 and Schönhage-Strassen algorithms [4, p. 90], which means all libraries used in the programs uses these algorithms *at least* when multiplying integers (i.e., coefficients of polynomials).

- NTL implements Karatsuba, Schönhage-Strassen and another FFT-based technique for polynomials [9].

- The external documentation of PARI does not specify which algorithms are implemented, but Karatsuba, some version of Toom-Cook and some FFT-based algorithm seem to exist in the source code.

2.3 Implementation

All development of the programs discussed in this report have been written in C/C++, developed in a Unix (Linux) environment with the GNU compiler collection `gcc` as compiling and optimizing tool. Some of the "helper" programs (for automating tests and similar) have been written in Java.

The first step of the implementation process was to get a working version of the FZT.

2.4 External libraries

In order to focus on the BHKK algorithm and not dwell too much on implementation-specific optimizations, and to reduce the scope of work to produce testable programs, we decided to not implement polynomial arithmetic. Instead we have employed the use of several external libraries that implement polynomial arithmetic, usually with several fast multiplication algorithms in use.

2.4.1 NTL

The Number Theoretic Library by Shoup [10] is a full-fledged C++ code base and provides a rich, high-level interface well suited for library usage. It does lack functionality for non-trivial polynomial exponentiation, and does not implement as many multiplication algorithms as comparable libraries.

NTL is advertised as “one of the fastest implementations of polynomial arithmetics”, which is all that we are interested in. Unfortunately, the results we have produced with it are not competitive. NTL is however very easy to use, provides its own garbage collection and has a rich, high-level interface for library usage.

The functions used are primarily these:

- `ZZX.operator+=()`

  Addition and assignment for polynomials.

- `ZZX.operator*=()`

  Multiplication and assignment for polynomials.

Here, released binaries compiled with NTL are called `bhkk-ntl-x.y.z`.

2.4.2 PARI

The PARI/GP project [11] is a computer algebra system, primarily designed as a full calculation system comparable to Maple. The back-end, PARI, is also available as a C library, providing polynomial arithmetics among other functionality.

The PARI library is provided at a low level, requires the user to garbage collect (since it is written in C, after all), has a much steeper learning curve and a very detailed but hard-to-grasp documentation. PARI provides special functions for exponentiation of polynomials, but it is a bit unclear how these are implemented exactly.

The functions used are primarily these:

- `ZX_add()`

  Addition for polynomials.

- `ZX_mul()`

  Multiplication for polynomials.

- `gpowgs()`

  General exponentiation for PARI types. Used for polynomials.

Here, released binaries compiled with PARI are called `bhkk-pari-x.y.z`.

2.4.3 FLINT

The Fast Library for Number Theory, FLINT, by mainly Hart [6] was also tried out. It does not use GMP, but instead a fork of GMP called MPIR, for low-level arithmetic operations. We did not release any program compiled with FLINT, however, as our initial tests showed no improvements as compared to PARI; it was decided time was best spent on other work.

2.4.4 GMP

NTL and PARI allow for the user to configure them using the GNU Multiple Precision library by mainly Granlund [4], as the lowest-level interface for integral arithmetic. Authors of the libraries suggest using GMP instead of their own native low-level interfaces, as this gives better performance. GMP implements a wide range of multiplication algorithms and provides fast assembler code optimizations for various CPU types.

GMP is well documented, easy-to-use, provides both C and C++ interfaces and even has a well-maintained[1] bug reporting facility. GMP provides a rich variety of configuration options, and we've tried to optimize as narrowly as possible to get maximum performance on one machine. In the implementations, we do not interface with GMP directly, as PARI and NTL hide this behind their own interfaces.

2.5 Experimental setup

Computer specs, graph classes, generators, automated test programs

[1] I got an answer the same day!

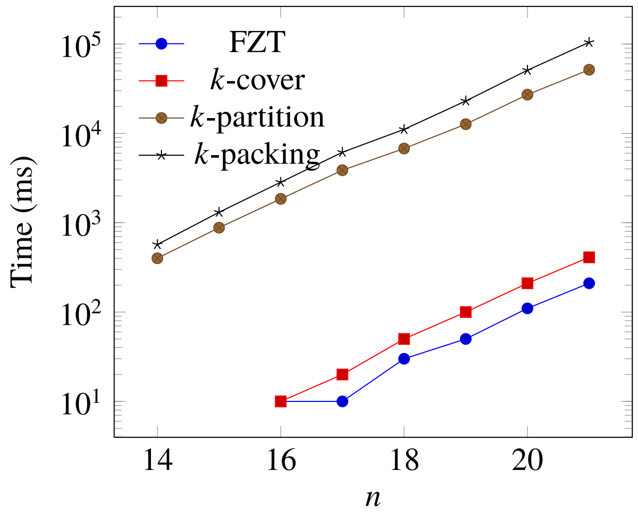
Chapter 3

Experimental results

Here we present graphs and stuff.

3.1 Set problems

While quite a lot of simulation was done for these programs, results were somewhat uninteresting.



What we can see is that there is a clear hierarchy between the problems; k-cover is the most time-consuming one. The "jump" is due to the use of polynomials instead of integers as ring elements. But from this diagram, we can not make any distinction between the asymptotical behaviour of these algorithms.

3.2 Computing chromatic polynomials

3.2.1 Fastest implementation

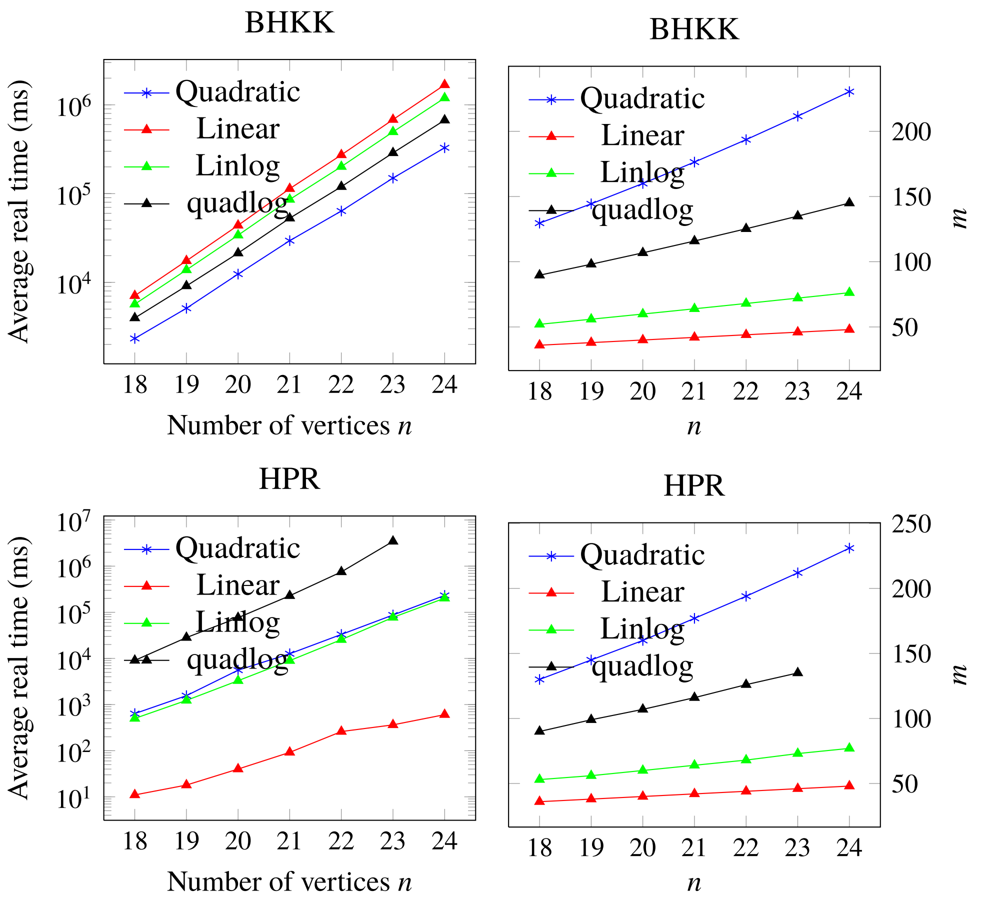
3.2.2 Polynomial arithmetic libraries

3.2.3 Different edge densities

We have four different classes of graphs: linear, log-linear, quadratic, and quadratic-over-log. The four functions are $f_1(\alpha, n) = \alpha n$, $f_2(\alpha, n) = \alpha n^2$, $f_3(\alpha, n) = \alpha n \ln(n)$ and $f_4(\alpha, n) = \alpha n^2 / \ln n$. The graphs generated have n nodes and f_i edges.

The α values are constants that we set to get a good spread of densities, in terms of maximum density. Linear graphs have around 20%, log-linear around 30%, quadratic around 80% and quadratic-over-log around 50%.

$\alpha_1 = 2$, $\alpha_2 = 0.4$, $\alpha_3 = 1$, $\alpha_4 = 0.8$.



3.3 Chromatic polynomials for certain graphs

3.3.1 Akbari's graph

Hillar and Windfeldt discussed characterizations of *uniquely* colourable graphs in [7]. That is, a graph G with $\chi_G(\chi(G)) = \chi(G)!$, or in other words that there exists only one optimal colouring, unique up to interchangability of the colours. Hillar and Windfeldt makes an attempt to verify their results by determining the chromatic polynomials of two graphs known to be uniquely 3-colourable, in order to test whether $\chi_G(3) = 3!$ for them. However, they were unable to determine the chromatic polynomial of the larger graph (on 24 vertices), Akbari's graph, seen here in figure 3.1, using Maple, as it uses a very naive algorithm to determine chromatic polynomials. Using BHKK, we successfully determined $\chi_{Akbari}(t)$:

$$\begin{aligned}\chi_{Akbari}(t) = {} & t^{24} - 45t^{23} + 990t^{22} - 14174t^{21} + 148267t^{20} - 1205738t^{19} + 7917774t^{18} \\ & - 43042984t^{17} + 197006250t^{16} - 767939707t^{15} + 2568812231t^{14} \\ & - 7407069283t^{13} + 18445193022t^{12} - 39646852659t^{11} + 73339511467t^{10} \\ & - 116102230203t^{9} + 155931129928t^{8} - 175431211152t^{7} + 162362866382t^{6} \\ & - 120414350156t^{5} + 68794778568t^{4} - 28408042814t^{3} + 7537920709t^{2} \\ & - 963326674t\end{aligned}$$

and in particular, $\chi_{Akbari}(3) = 3!$, as expected. This took 1445 seconds to compute, using our fastest implementation. HPR however terminated even faster, which was to be expected.

3.3.2 Queen graph

The $n \times n$ *Queen graph* is a graph laid out like a chess board with n squares per side. Each square is a vertex and it has edges to all squares in its column, in its row and in its diagonals. In other words, to each square to which a queen could move, if placed on said square. Here we provide the chromatic polynomials of the 5×5 and 6×6 Queen graphs Q_5 and Q_6, on 25 and 36 vertices, respectively.

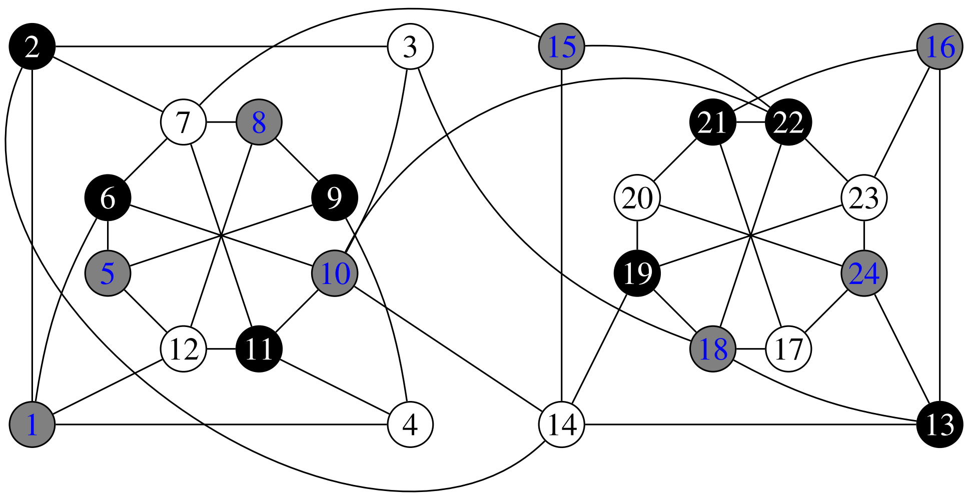


Figure 3.1: Akbari's graph, coloured in its unique 3-colouring with white, gray and black as colours. Figure copied from figure 2 in [7].

$$
\begin{aligned}
\chi_{Q_5}(t) = {} & t^{25} - 160t^{24} + 12400t^{23} - 619000t^{22} + 22326412t^{21} - 618664244t^{20} \\
& + 13671395276t^{19} - 246865059671t^{18} + 3702615662191t^{17} \\
& - 46639724773840t^{16} + 496954920474842t^{15} - 4497756322484864t^{14} \\
& + 34633593670260330t^{13} - 226742890673713726t^{12} \\
& + 1258486280066672806t^{11} - 5890734492089539317t^{10} \\
& + 23071456910844580538t^{9} - 74774310771536397886t^{8} \\
& + 197510077615138465516t^{7} - 416375608854898733286t^{6} \\
& + 680208675481930270860t^{5} - 824635131668099993614t^{4} \\
& + 692768396747228503860t^{3} - 356298290543726707632t^{2} \\
& + 83353136564448062208t
\end{aligned}
$$

$$\chi_{Q_6}(t) =$$

Polynomial	Algorithm	Real time (s)	Peak resident set size (kB)
χ_{Q_5}	BHKK	1453	199216
χ_{Q_5}	HPR	2727	41094832
χ_{Q_6}	BHKK	$\sim 10^7$	-
χ_{Q_6}	HPR	-	-

Table 3.1: Time and memory measurements on computing chromatic polynomials of queen graphs.

Chapter 4
Conclusions

We won vs HPR. Parallelization is key.

Bibliography

[1] Birkhoff, G D.
"A determinant formula for the number of ways of coloring a map"
Annals of Mathematics **14** (1/4) (1912), pp. 42–46, ISSN 0003486X, .
URL `http://www.jstor.org/stable/1967597`

[2] Björklund, A, Husfeldt, T, Kaski, P, Koivisto, M.
"Covering and packing in linear space"
Information Processing Letters **111** (21–22) (2011), 1033 – 1036, ISSN 0020-0190, doi:http://dx.doi.org/10.1016/j.ipl.2011.08.002.
URL `http://www.sciencedirect.com/science/article/pii/S0020019011002237`

[3] Ellis-Monaghan, J, Merino, C.
"Graph polynomials and their applications I: The Tutte polynomial"
in M. Dehmer (ed.), *Structural Analysis of Complex Networks* pp. 219–255 (Birkhäuser Boston, 2011), ISBN 978-0-8176-4788-9, doi: 10.1007/978-0-8176-4789-6_9.
URL `http://dx.doi.org/10.1007/978-0-8176-4789-6_9`

[4] GNU Project.
The GNU Multiple Precision library, version `5.1.2`, (2013).
URL `http://gmplib.org/`

[5] Haggard, G, Pearce, D J, Royle, G.
"Computing Tutte polynomials"
ACM Trans. Math. Softw. **37** (3) (2010), 24:1–24:17, ISSN 0098-3500, doi:10.1145/1824801.1824802.
URL `http://doi.acm.org/10.1145/1824801.1824802`

[6] Hart, W, Johansson, F, Pancratz, S.
Fast Library for Number Theory, version `2.3.0`, (2012).
URL `http://www.flintlib.org/`

[7] Hillar, C J, Windfeldt, T.
"Algebraic characterization of uniquely vertex colorable graphs"
Journal of Combinatorial Theory, Series B **98** (2) (2008), 400 – 414, ISSN 0095-8956, doi:http://dx.doi.org/10.1016/j.jctb.2007.08.004.
URL `http://www.sciencedirect.com/science/article/pii/S009589560700086X`

[8] Knuth, D E.
The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997), ISBN 0-201-89684-2, .

[9] Shoup, V.
Number Theoretic Library, version `6.0.0`, (2013). Source documentation.
URL `http://www.shoup.net/ntl/doc/ZZX.txt`

[10] Shoup, V.
Number Theoretic Library, version `6.0.0` New York University, (2013).
URL `http://www.shoup.net/ntl/index.html`

[11] The PARI Group.
PARI/GP, version `2.5.5` Université Bordeaux, Bordeaux (2013).
URL `http://pari.math.u-bordeaux.fr/`

[12] Whitney, H.
"The coloring of graphs"
Annals of Mathematics **33** (4) (1932), pp. 688–718, ISSN 0003486X, .
URL `http://www.jstor.org/stable/1968214`

[13] Wikipedia.
"Schönhage–strassen algorithm — wikipedia, the free encyclopedia"
(2013). [Online; accessed 17-December-2013].
URL `http://en.wikipedia.org/w/index.php?title=Sch%C3%B6nhage%E2%80%93Strassen_algorithm&oldid=566375677`

[14] Wikipedia.
"Toom–cook multiplication — wikipedia, the free encyclopedia"
(2013). [Online; accessed 17-December-2013].
URL `http://en.wikipedia.org/w/index.php?title=Toom%E2%80%93Cook_multiplication&oldid=551043219`

Appendices

Appendix A
Code examples

Here is presented selected pieces of code from the implementations presented in this report.

A.1 The most executed function

The most important function in the implementation is the `utils::parallel` function. It is the one that is executed on each parallel thread. This is the C++ code for the function, with library-specific code, developer-only comments and debug aid removed.

```
/*
 * Executes the BHKK algorithm for a range of subsets starting
 * from start and ending with end, proceeding in in-order.
 */
void utils::parallel(
      rval_t* r,
      const set_t& start,
      const set_t& end,
      const u_int_t& two_to_the_n1,
      const u_int_t& two_to_the_n2,
      const set_t& v2,
      const u_int_t& n,
      const u_int_t& n2,
      const u_int_t& q,
      bool** matrix) {

   // For each subset X1 of V1
   for (set_t x1 = start; x1 <= end; ++x1) {

      // Arrays of polynomials
```

```
rval_list_t l(two_to_the_n2);
rval_list_t h(two_to_the_n2);

// For each subset Z2 of V2, set h(Z2) <- 0
for (u_int_t i = U_ZERO; i < two_to_the_n2; ++i) {
   init_zero(h[i]);
}

// For each independent subset Y1 of X1,
// set h(V2 \ N(Y1)) <- h(V2 \ N(Y1)) + z^(|Y1|)
for (set_t y1 = EMPTY_SET; y1 <= x1; ++y1) {

   if ((y1 | x1) <= x1) { // <=> Y1 is subset of X1
      if (independent(y1, matrix, n)) {
         rval_t p;
         init_monomial(size_of(y1), p);
         set_t neighbours = EMPTY_SET;
         neighbours_of(y1, matrix, n, neighbours);

         add_assign(
           h[(v2 & (~ neighbours)) / two_to_the_n1], p);
      }
   }
}

// For each independent subset Y2 of V2,
// set l(Y2) <- z^(|Y2|)
for (u_int_t i = U_ZERO; i < two_to_the_n2; ++i) {
   set_t y2 = i * two_to_the_n1;

   if (independent(y2, matrix, n)) {
      init_monomial(size_of(y2), l[i]);
   } else {
      init_zero(l[i]);
   }
}

// Set h <- hS'
fast_up_zeta_transform_exp_space(n2, h);

// Set h <- h*l
for (u_int_t i = U_ZERO; i < two_to_the_n2; ++i) {
   mul_assign(h[i], l[i], n);
}

// Set h <- hS
fast_down_zeta_transform_exp_space(n2, h);

// For each subset X2 of V2,
        // set r <- r + (-1)^(n-|X1|-|X2|) * h(X2)^q
        for (u_int_t i = U_ZERO; i < two_to_the_n2; ++i) {
            set_t x2 = i * two_to_the_n1;
            int exponent = n - size_of(x1) - size_of(x2);
            int sign = exp_neg_one(exponent);

            power(h[i], q, n);
            flip_sign(h[i], sign);

            add_assign(*r, h[i]);
        }
    }
}
```

Appendix B
Additional figures

Some diagrams that aren't very important, but still deserve mention.