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

Mats Rydberg
rydbergmats@gmail.com

October 15, 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 each other 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

To Sasha.

Contents

1 Introduction **7**

1.1 Report structure . . . 7

1.2 Preliminaries . . . 7

1.2.1 Set problems . . . 8

1.2.2 Chromatic polynomial . . . 8

1.2.3 Miscellaneous . . . 8

1.3 Previous work . . . 9

1.4 Problem statement . . . 10

1.5 Main results . . . 10

2 Approach **11**

2.1 Studied algorithms . . . 11

2.1.1 The fast zeta transform . . . 11

2.1.2 Coverings, partitionings and packings . . . 12

2.1.3 Chromatic polynomial . . . 12

2.1.4 Optimizations . . . 13

2.2 Algorithmic aspects . . . 16

2.2.1 Graph size . . . 16

2.2.2 Multiplication algorithms . . . 16

2.3 Implementation . . . 17

2.4 External libraries . . . 18

2.4.1 NTL . . . 18

2.4.2 PARI . . . 18

2.4.3 FLINT . . . 19

2.4.4 GMP . . . 19

2.5 Experimental setup . . . 19

3 Experimental results **21**

3.1 Set problems . . . 21

3.2 Computing chromatic polynomials . . . 22

3.2.1 Fastest implementation 22
3.2.2 Power of parallelization 25
3.2.3 Polynomial arithmetic libraries 27
3.2.4 Different graph size functions 28

4 Chromatic polynomials for certain graphs 31
4.1 Akbari's graph 31
4.2 Queen graph 32

5 Conclusions 35
5.1 Which algorithm should I choose? 35
5.2 Open questions 36

Bibliography 37

Appendix A Code examples 41
A.1 The most executed function 41
A.2 The fast zeta transform 43
A.3 The linear-space fast zeta transform 44

Appendix B Additional figures 47

Appendix C Seminar paper 51

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 [5]. The two main results of that report is an algorithm, *the fast zeta transform*, with reduced space requirement from before, along with an application of this result on another algorithm for computing the *chromatic polynomial* of a graph. Our focus will lie on the chromatic polynomial algorithm, but in the process of implementing that algorithm, 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 describe 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 unfortunate case of an ill-defined variable used in the following chapters, 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 ≤ 1 and are never loops.

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 $\sigma : V \rightarrow [q]$ 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 $P(G, q)$.

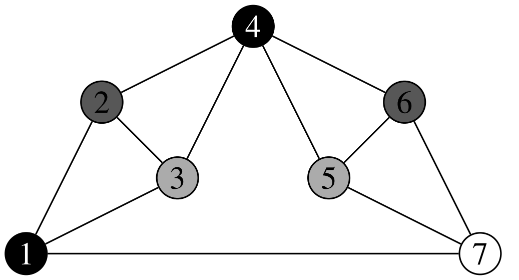
The *chromatic polynomial* $\chi_G(t)$ of the graph G is the polynomial of degree n in one indeterminate that passes through each point $(q, P(G, q))$. To determine $\chi_G(t)$, by for example specifying its coefficients, it suffices to determine the numbers $P(G, q)$ for each $q \leq n$ and then construct $\chi_G(t)$ by interpolation, as $n + 1$ points uniquely identifies an n-degree polynomial. Some characteristics of all chromatic polynomials (proofs can be found in [14, p.13]):

- The coefficient of t^n is always one.
- The coefficient of t^{n-1} is always $-m$.
- The coefficient of t^0 is always 0.
- The signs of the coefficients are alternating positive and negative.

Figure 1.1 presents an example graph and its chromatic polynomial.

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, in percent, $dE = 100 \cdot m/\binom{n}{2}$.



$$\begin{aligned}\chi_{MS}(t) = {} & t^7 - 11t^6 + 51t^5 \\ & -129t^4 + 188t^3 \\ & -148t^2 + 48t \\ \\ \chi_{MS}(3) = {} & 0 \\ \chi_{MS}(4) = {} & 384 \\ dE = {} & 100 \cdot 11/\tbinom{7}{2} = 52\end{aligned}$$

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 $\chi_{MS}(t)$.

1.3 Previous work

The chromatic polynomial was specified in 1912 by Birkhoff [3], who defined it for planar graphs with the intent on proving the Four Colour Theorem. Whitney extended its definition to general graphs in 1932 [20], and Tutte incorporated it into what is now known as the Tutte polynomial. See for example Ellis-Monaghan and Merino [7] for more in-depth analysis of the Tutte polynomial.

The most commonly referenced problem in the field of graph colouring is the one of producing an *optimal* colouring of a given graph, that is, a colouring using $\chi(G)$ colours. Trivially, this can of course be done by exhaustively searching through all possible assignments of the n vertices to q colours, starting from $q = 3$ and moving upwards (since 2-colouring is a problem in P), each step taking time $O^*(q^n)$, bound at the last step by $O^*(n^n)$. The notation O^* hides polylogarithmic factors.

For general colouring, Christofides [6] constructed the first non-trivial algorithm in 1971, running in time $O(n!)$, a result which was first improved by Lawler [16] in 1976, who used dynamic programming and enumeration of maximal independent sets to obtain a running time of $O^*(2.4423^n)$. Björklund, Husfeldt and Koivisto [4] currently hold the fastest known algorithm, exploiting the principle of inclusion-exclusion and the fast zeta transform to obtain a running time of $O^*(2^n)$. That algorithm is also the base of the main algorithm studied in this report.

For small, fixed q, faster algorithms are known; Beigel and Eppstein [2] currently hold the record of 3-colouring since 2005, which takes them $O^*(1.3289^n)$ time; considerably better than $O^*(2^n)$. To achieve this bound, they use a reduction of the colouring into a constraint satisfactory problem, in which they are able to find divisions of the problem into smaller subproblems, and solving these individually. Fomin, Gaspers and Saurabh [8] invented the best known algorithm for 4-colouring in 2007, running in $O^*(1.7272^n)$ time. They base their algorithm on the proof that a graph will either allow efficient branching procedures to reduce the size of the problem, or it will have a small pathwidth, allowing for efficient algorithms to exploit its path decomposition.

In 2010, Haggard, Pearce and Royle [11] published a program, referred to in the following as **HPR**, to compute the Tutte polynomial for graphs using a deletion-contraction algorithm. The basic idea behind HPR goes back to Zykov [23], using cached subgraphs and isomorphism rejection to obtain good performance, and it 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 [5] 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. We discuss this algorithm in detail in section 2.1.3.

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 [5, p.10]. Haggard *et al* does not perform an asymptotic analysis of their algorithm, instead presenting 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 guided us during the implementation process. We describe the implementation process in 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 in this subsection briefly describe a few other algorithms that were studied as a part of our work.

2.1.1 The fast zeta transform

Björklund *et al* [5] 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 [5, p.5] is described pseudo-code for an algorithm computing the fast zeta transform in time and space exponential in n. In the appendix, section A.2, is listed the implementation used for this algorithm.

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 outline, taken from [5, sec.3], uses $O^*(|\mathcal{F}|)$ space:

1. For each $X_1 \subseteq U_1$ do:

    a) For each $Y_2 \subseteq U_2$, 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 $n_2 = \lceil \log_2 |\mathcal{F}| \rceil$ and $n_1 = n - n_2$, 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

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

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, we consider them a $(q+1)$-partition, where q sets are taken from $\mathcal{F}$ and one is an arbitrary subset of U. The result is the nth coefficient of p_q, calculated as

$$p_q(\mathcal{F}) = \sum_{X \subseteq U} (-1)^{|U \setminus X|} (1+z)^{|X|} f\zeta(X)^q.$$

The bulk of the code for the implementations of these algorithms is found listed in the appendix, section A.3.

2.1.3 Chromatic polynomial

By adapting the linear-space FZT, an algorithm for the chromatic polynomial was designed in [5]. 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 $n_1 = \lceil n \log 2 / \log 3 \rceil$ and $n_2 = n - n_1$, and then run iterations of subsets of V_1 and store values dependent on (subsets of) V_2 [5, 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 [5, p 9].

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

2.1.4 Optimizations

Here are presented some improvements to the algorithm that are either natural, mentioned in [5], or invented by the author. This list is by no means exhaustive, nor is every item critical, but the ones we have explored proved to be efficient.

Exploiting q First, we can consider optimizing on the basis of the value of q.

- For $q = 0$, there are 0 colourings, as no graph can be 0-coloured.
- For $q = 1$, there are 0 colourings if and only if $|E| > 0$, otherwise there is exactly 1 colouring. This takes $O(n^2)$ time to check.
- For $q = 2$, it is well-known that the graph can be coloured (or found to be non-colourable) in polynomial time using standard techniques (such as breadth-first search).

These optimizations will reduce the iterations of the loop at step A by three.

Using $\omega_{min}(G)$ A more sophisticated type of optimization involves exploiting the clique number $\omega(G)$, which is a lower bound on the chromatic number $\chi(G)$. Knowing that $\omega(G) \geq a$ for some constant a would allow us to immediately skip all steps A where $q < a$. If $a = n$, we have the complete graph K_n, for which $\chi_G(t)$ is known.

We have found that the *smallest possible* $\omega(G)$, let us call it $\omega_{min}(G)$, is a function of the graph size m. In fact, the following holds:

Lemma 1. *The number $\omega_{min}(G)$ is the lower bound of $\omega(G)$, and its value is*

$$\omega_{min}(G) = \begin{cases} n - \binom{n}{2} + m & \text{if } m \geq \binom{n}{2} - \lfloor n/2 \rfloor \\ \lceil n/2 \rceil - a & \text{if } \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil < m < \binom{n}{2} - a\lfloor n/2 \rfloor, a \in \mathbb{N}_+, a \text{ maximal} \\ 2 & \text{if } 0 < m \leq \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \\ 1 & \text{if } m = 0 \end{cases}$$

Proof. Trivially, $w(N_n) = 1$, where N_n denotes the null graph of order n. This proves the lower-most bound.

Next, consider the complete bipartite graph $K_{\alpha,\beta}$ on n vertices; it is the graph with the most edges that has clique number 2. It is well-known that it has $\alpha\beta$ edges. To maximize this product, we make a half-half partition, setting $\alpha = \lfloor n/2 \rfloor$ and $\beta = \lceil n/2 \rceil$, giving $\alpha\beta = \lfloor n/2 \rfloor \lceil n/2 \rceil$. The point made is that there is no way of assigning more edges than this without yielding a higher clique number.

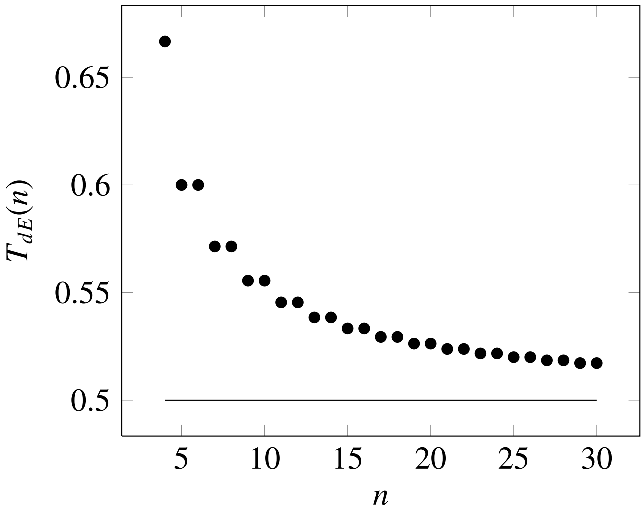
Third, consider the complete graph K_n, where $w(K_n) = n$. Deleting an edge vw will clearly reduce the clique number (by one). Consider the subgraph $K \setminus \{v\}$; it has clique number $n - 1$. Removing an edge $uu', u \neq w, u' \neq w$ will lower clique number again by 1. This process may be repeated $\lfloor n/2 \rfloor$ times. The resulting graph has $\binom{n}{2} - \lfloor n/2 \rfloor$ edges and clique number $n - \lfloor n/2 \rfloor$. This, together with the fact that removing one edge can lower clique number by maximum one, proves the uppermost bound. □

For the final case, for which we do not provide a full proof, we observe that $\binom{n}{2} - \lfloor n/2 \rfloor - \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ is a multiple of $\lfloor n/2 \rfloor$ (this is the variable a in the lemma). It can be shown that this corresponds to the requirement of removing $\lfloor n/2 \rfloor$ edges to reduce clique number by one.

As we can see from Lemma 1, only graphs with $m > \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ provides $\omega_{min}(G) > 2$ and for $q \leq 2$ we already have good optimizations. So how dense is a graph where this bound on m holds? Let us specify the threshold density $T_{dE}(n)$ as

$$T_{dE}(n) = \frac{\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil}{\binom{n}{2}} = \begin{cases} \frac{1}{2}n(n-1)^{-1} & \text{if } n \text{ even} \\ \frac{1}{2}(n+1)n^{-1} = T_{dE}(n+1) & \text{if } n \text{ odd} \end{cases}$$

In conclusion, any graph with $dE > T_{dE}(n)$ can optimize away at least one additional computation of step A above. It also follows that as $n \to \infty$ we will have $T_{dE}(n) \to \frac{1}{2}$. The following plot shows how fast we converge for graphs of orders relevant for this paper.



For higher-order graphs, we have a smaller $T_{dE}(n)$, which gives us a higher probability to be able to optimize (under the assumption that all sizes are equally likely), and it is also for higher-order graphs that we are most interested in optimizing techniques. For a graph with $n = 30$ and $m = 327$ (where $dE = 75$), we would be able to skip evaluating $q \leq 8$, which yields a decrease in execution time by about 14% (see table 3.2).

Parallelization 1 The steps A2 are independent and can be computed in parallel on $2^{|V_1|}$ CPUs. This would yield significant time improvements in theory, reducing the asymptotic time bound to about $O^*(1.5486^n)$. Using a different partition of V with $n_1 = n_2$, we would achieve space and time bounds of $O^*(2^{n/2})$ on as many CPUs [5].

Parallelization 2 Typically, we will only have access to a constant number of CPUs in practice, allowing each of them to not execute one step but a range of iterations of step A2. This allows for heuristics on how to select such ranges so that the overall time bound (set by the range of subsets X that include the *most* independent subsets Y) is minimal. The currently used heuristic is to simply take the subsets in inorder. As presented in section 3.2.2, we can expect to reduce the time consumption of the program by a factor of around 6 in our test environment.

Parallelization 3 The steps A in BHKK are independent of each other, and allows parallelization on $O(n)$ CPUs. This would not reduce the exponential factor of the time complexity, but it will reduce the polynomial factor, and it is likely to give significant results in practice.

Degree pruning In step A(2)e we exponentiate a polynomial of degree $d \leq n$ with q, yielding a polynomial of degree $d \leq nq$. Since $q \leq n$, we could have as much as $d = n^2$. But since we never do any divisions, and never care about coefficients for terms of degree $> n$, we can simply discard these terms, keeping $\deg r \leq n$. This also applies to the multiplications of step A(2)c.

Caching Since in fact all steps of the inner loop of BHKK are independent from q, except the final step A(2)e, we are actually re-computing the same values for the array h as we increase q. If we would cache these values after the first call of step A2, we would be performing only step A(2)e for all the rest of the computations, plus a look-up in our cache table. This would require a cache of size $2^{n_1}2^{n_2} = 2^n$, increasing our space bound, but possibly improving time performance in practice.

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 size

The algorithm in itself is designed in a way that allow for a smaller degree of complexity for graphs of *large size*, 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 large-sized graphs, but the performance of the algorithm is in fact a function that is directly related to graph size, 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 size 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 *small-sized* 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 of these algorithms, see for example Wikipedia [21] [22]. An in-depth description can be found in Knuth [15, sec. 4.3.3].

As presented below in section 2.4, 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 [10, 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 [17].
- 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 (GNU/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. This was then extended with functionality for calculating q-cover. Using native C++ data types, we could only support problems of order $n < 12$, which of course are too small. Thus we employed the use of GMP, in order to represent arbitrarily large integers.

In order to incorporate q-partition and q-packing, an implementation of polynomials was needed. A naive class was made, with slow multiplication algorithms. Basically it is only an array of GMP integers, using cross-multiplication with running time $O(n^2)$. Simulations were then run on the three programs generated, from which the only interesting results can be seen in section 3.1.

That process being mostly an exercise in understanding the main issue, work was started on expanding the FZT implementation to become a BHKK implementation. Once this was done, tests showed that performance was not up to expectations. The problem was of course the polynomial arithmetic; enter the external libraries. First we incorporated NTL, which was easier than expected, but while it was better, it wasn't good enough. Then we turned to PARI, a task that was much the opposite; it was hard to get working properly, but once it did, results were quite satisfactory.

Simulation suites were run many times on the NTL and PARI implementations as they developed, but eventually NTL was abandoned due to being too slow. We gave FLINT a try, but while it was better than NTL, it didn't show any indications on being able to beat PARI. Table 2.1 lists all optimizations that time allowed to be implemented.

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 [18] 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 is very well documented.

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 [19] is a computer algebra system, primarily designed as a full calculation system for end users, 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 (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 [12] 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 mainly by Granlund [10], 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 have 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

All tests were performed on the same machine, with the following specifications.

CPU (cores, threads)	Intel i7-3930K 3.2GHz (6, 12)
OS	GNU/Linux 3.8.13.4-desktop-1.mga3 (Mageia 3) x86_64
Memory	16GB DDR3 1333Mhz
Compiler	GCC 4.7.2 (g++)

[1] I got an answer the same day!

Program ID	Implemented optimizations
`0.1`	degree pruning, $q = \{0, 1\}$
`0.2`	parallelization 2
`0.3`	$\omega_{min}(G)$

Table 2.1: Reference of program versions and characteristics of them. Refer to section 2.1.4 for details of the optimizations.

For all time and memory measurements, the GNU `time` 1.7 program is used (see the Linux man pages [9]). The user time, elapsed time (real time)[2] and peak resident set size are the data points recovered as measurements. These measurements are taken by running the specified program on a number of graphs of equal order and size and the average values are the ones presented. For most tests, we average over 100 graphs, but for larger-order graphs we average over 50 to reduce the time needed for the simulation to finish.

The `time` program encapsulates all the executables, but it is in turn encapsulated in a Java program which collects the output data and calculates averages. The Java program also traverses directory structures in order to find all (several thousands) graph instances to be used as input.

Unless otherwise specified, we perform our tests on random graphs, generated via a simple algorithm: to construct a graph of order n and size m, initialize an $\binom{n}{2}$-sized array of m ones and rest zeros, and then shuffle it using Fisher-Yates shuffle [15, p. 145]. The edge $v_i v_{i+1}$ will be in the resulting graph if the array has a one in position i. The program developed for this purpose outputs an adjacency matrix. Other utility programs are able to convert this to an edge list in the format required for the HPR program. Unless otherwise specified, the graphs are generated with a quadratic size, which is a function of graph order n and a density dE, as

$$m = (dE/100)\binom{n}{2}.$$

There are three "released" versions of the programs that implement the BHKK algorithm. These are linearly developed and implement all the optimizations of the preceding versions. Table 2.1 shows which implements what.

[2] When running single-thread versions, we actually measure "user time", which is nearly equal to real time. "Nearly" refers to the fact that there are some units of time scheduled as system time, but these values are too small to be significant in these experiments.

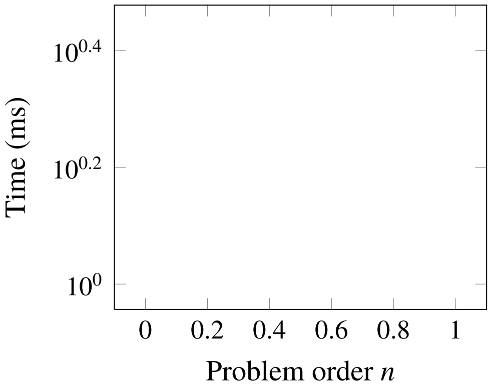
Chapter 3

Experimental results

In this chapter, we focus on the results that we managed to achieve through this Master's Thesis. We have tried to be as picky as possible when selecting what data to present, as there is quite a lot produced. The sections are ordered logically rather than chronologically, and most diagrams are presented as part of the text, and are unnamed.

3.1 Set problems

While quite a lot of simulation was done for these programs, results were somewhat uninteresting. This is due to several reasons. Firstly, the simulation process was not mature enough to be tuned to test the most interesting areas. Secondly, these problems are only an auxiliary part of the scientific worth of the main paper (Björklund *et al* [5]) studied in this Master's Thesis. Thirdly, there was no reference implementation with which to compare performance. Due to these reasons, we only present the hierarchy and absolute values of the running times of the set problems, also including a plot for the "raw" FZT, without any summation in the inner loop.



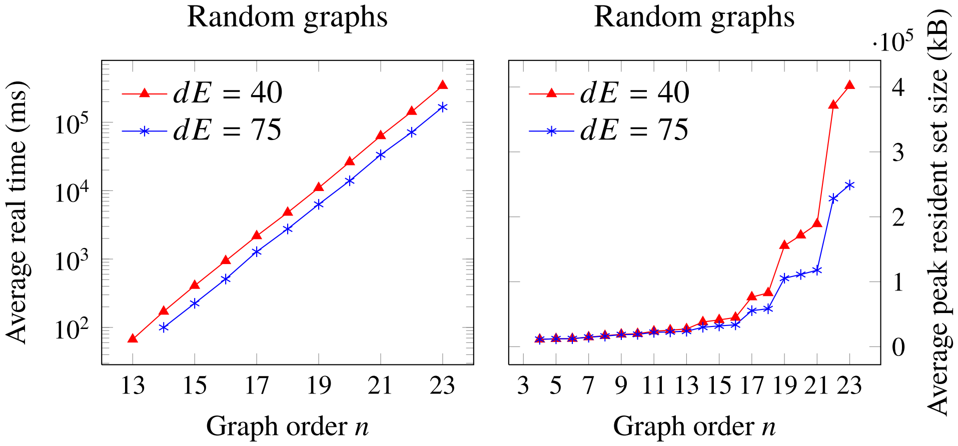
As expected, q-packing is the most expensive algorithm. These programs use a naive polynomial implementation, and it is almost guaranteed that they would perform significantly better if PARI polynomials were used. If it was not obvious already, this plot makes it undoubtedly clear that polynomial arithmetic far outscales integer arithmetic in terms of computational cost.

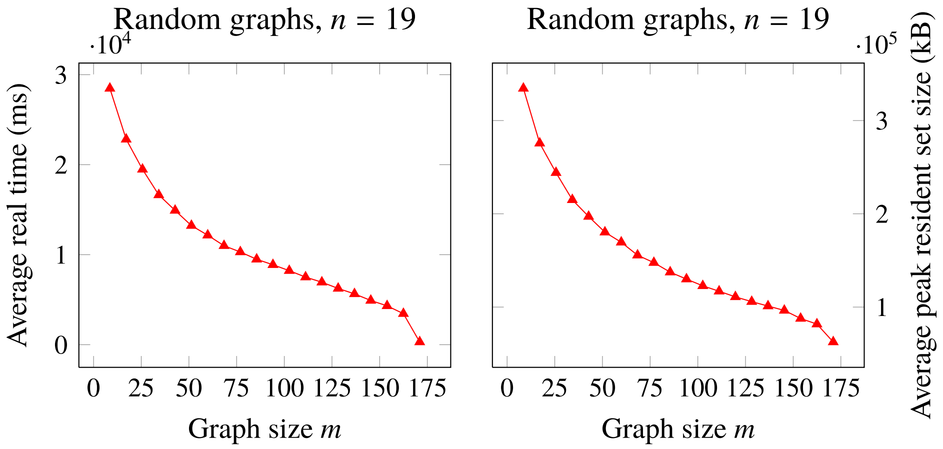
3.2 Computing chromatic polynomials

We turn our focus to the core results of this Master's Thesis, namely the performance of the BHKK algorithm in practice. Since several versions were developed, the potential width of our simulations is huge. We will not present every simulation and its results, but limit ourselves to the most important and interesting ones. In short, it comes down to five cases: first we race the fastest implemented version against itself; second against the reference algorithm HPR; third we race our parallelized version(s) against the non-parallel ones; fourth we race the polynomial arithmetic libraries against each other; and finally we compare the behaviour of BHKK and HPR on a few other classes of random graphs.

3.2.1 Fastest implementation

The fastest program developed is the one we call `bhkk-pari-0.3`. The below plots visualize its performance in relation to increasing order and size, respectively. These runs were executed using 45 virtual threads, a number that was measured to be the lowest-cost in terms of time consumption (see 3.2.2 for details).

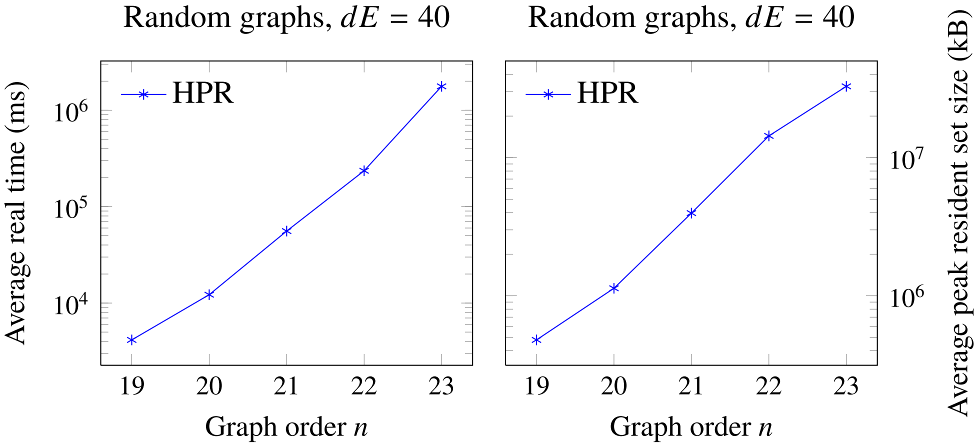




What we can see is that BHKK performs consistently better for graphs of *larger* size, which is in alignment with our discussion in section 2.2.1; we experiment more on this characteristic in section 3.2.4 below. We also see that for graphs of order ≤ 11, the program runs too fast for our measurement accuracy. For order 23, each run takes about 5 minutes. Making 100 such runs is about 10 hours of computing time; we are approaching our exhaust limit for computation time.

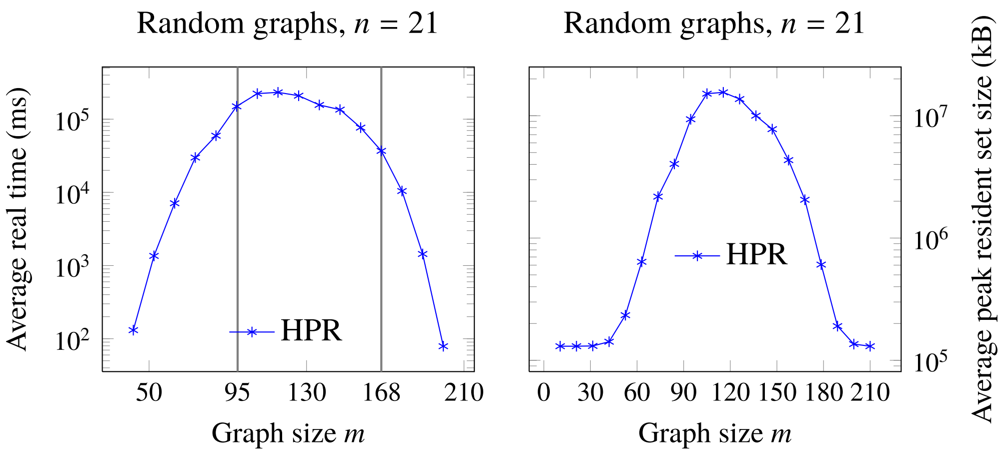
But is this performance any good? To provide an answer we let HPR and BHKK run on the same input graphs. When doing some initial testing, we find that HPR seems to do better if the input has "some kind of structure", rather than being "random". This is manifested by it performing considerably well on any "famous" graph (which usually is famous because such graphs appeal to some "human sense" of pattern-recognition). But in the case of graphs generated according to the randomized process mentioned in section 2.5, HPR does not scale as well as BHKK, and has a "weak spot" for graphs of density around 75 [11, p. 14].

Since the simulations take a lot of time, we narrow our scope of graph orders, concentrating on the point where BHKK and HPR are about equal in performance. These runs use 12 virtual threads for BHKK.



In general, it can be said that for orders below 20, HPR will nearly always outperform BHKK, but on average, we here show that BHKK is faster for graphs of order > 21. In particular, the memory consumption is *considerably* lower; from about 11% at $n = 19$ to 0.4% at $n = 23$.

We now fix our focus on the graphs of the order which seems to be the most competitive for BHKK and HPR, $n = 21$, and vary over the graph size instead.



These results are somewhat in contrast to those presented in [11]. The data here suggests that the "weak" spot in terms of density would be around $dE = 50$, whereas [11] suggests "weak spot" around $dE = 75$. Nevertheless, it is clear that graph size matters, more so for HPR than for BHKK, and that BHKK is a better choice for graphs of certain sizes as indicated by the lines in the diagram, even for orders below 22.

More pressingly, we found that HPR does not give good worst-case performance. Our measurements (see table 3.1) show a large ($> 95\%$) variance in performance on different graphs of equal size. (With variance, we mean the number $1-t/T$, where T is the maximum measured value and t the minimum.) This is not the case with BHKK, which has near-deterministic ($< 15\%$ variance) computation time for a graphs of given dimensions.

Graph metrics	Min. time (s)	Max. time (s)	Min. mem (kB)	Max. mem (kB)
$n = 19, m = 69$	0.30	20.56	154,736	1,903,472
$n = 20, m = 76$	0.43	57.88	164,224	4,789,520
$n = 21, m = 84$	1.45	273.09	224,432	16,424,528
$n = 22, m = 93$	18.97	1335.14	1,375,488	41,091,616

Table 3.1: Variance of HPR over 50 random graphs of equal size. Note that already for $n = 19$, there are over 10^{48} different graphs of this size possible. This implies that the (maximum) variance with high probability is larger than our measurements.

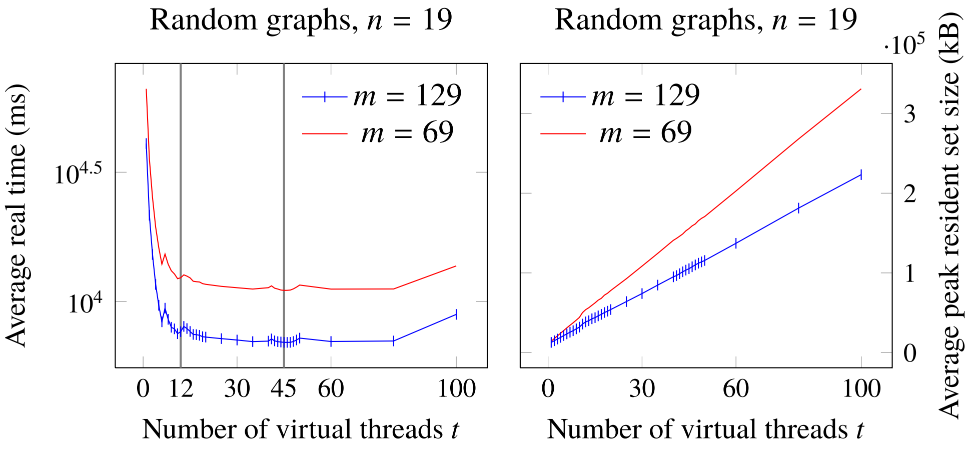
In order to test our limits, we also run HPR and several versions of BHKK on two larger graphs. The results may be unfair to HPR, since some other graph of same order and size might have been easier to compute. Results are shown in table 3.2.

	Real time (s)	Peak resident set size (kB)
$n = 25, m = 225$		
`bhkk-pari-0.2`	941	$1.46 \cdot 10^5$
HPR	$> 260,000$	-
$n = 30, m = 327$		
`bhkk-pari-0.3`	$46,318$	$7.41 \cdot 10^5$
`bhkk-pari-0.2`	$53,834$	$7.41 \cdot 10^5$
`bhkk-ntl-0.2`	$> 2,450,000$	$5.7 \cdot 10^4$
HPR	$> 95,000$	$4.11 \cdot 10^7$

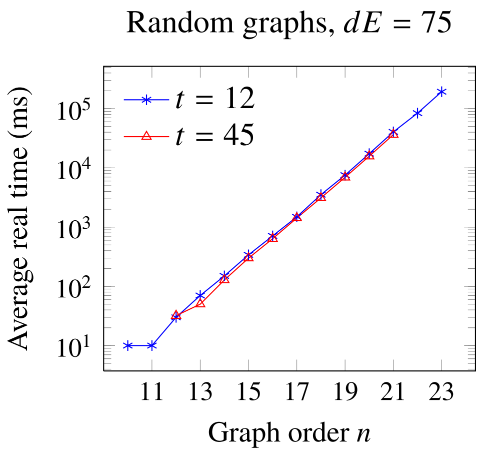
Table 3.2: Larger instances. Results are based on a single run on one graph G^{30} and a smaller graph of order 25. HPR was allowed 10GB cache, but did not terminate within the times specified here. Neither did the program linked with NTL. The chromatic polynomial of G^{30} is found in figure B.2 in the appendix. Note that `bhkk-pari-0.3` is about 14% faster than `bhkk-pari-0.2`, as mentioned in section 2.1.4.

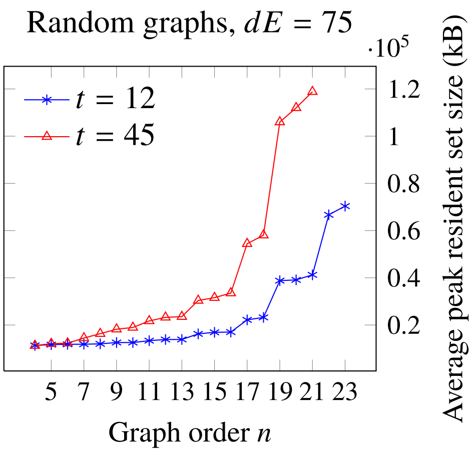
3.2.2 Power of parallelization

As is mentioned in [5], the BHKK algorithm parallelizes well. But to fully make use of its characteristics, we will need an exponential number of processors, as we discussed in section 2.1.4, and we do not have that; we only have access to a constant parallelization width of 12. Typically, some threads will finish before others, and so the use of more than 12 threads could be motivated, to make use of the processing time freed by the "fastest" threads. Specifying this number is tricky, however, and we also pay a large cost in memory. Below we see time and memory consumption for increasing amounts of threads used. The program tested is `bhkk-pari-0.2`.



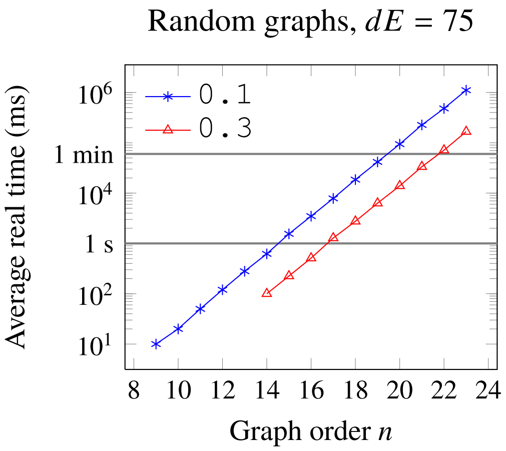
For the first few threads, the impact is significant. We are drastically lowering our execution time, but hit the floor at 12, as expected. The minimum for graphs of this order seems to lie around 45, which seems to be a random number. Interestingly enough, it seems to be consistently better than choosing 12 threads:

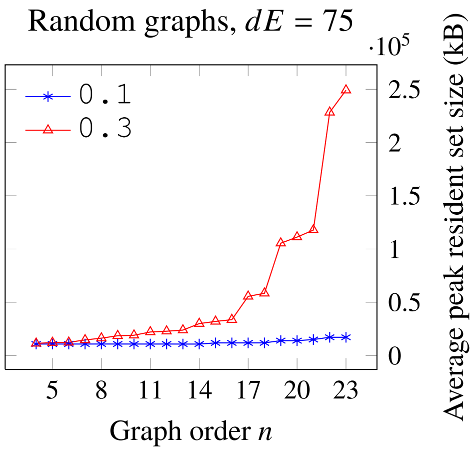


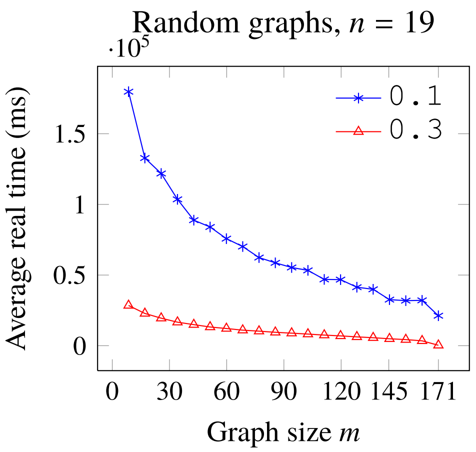


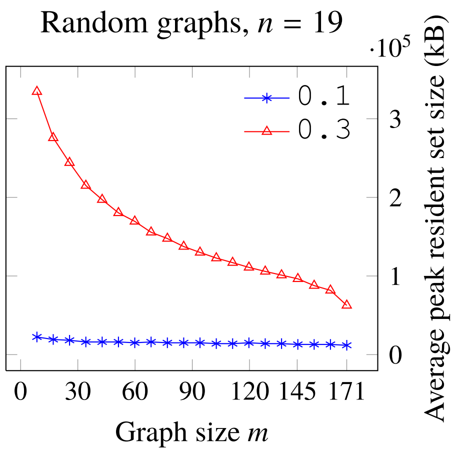
While the time difference is very small, in fact 12 threads were never better than 45. It remains unclear whether graph order, process scheduling techniques or algorithm design is the cause of 45 to be a “magic number”. We do pay for the small time gain with a sizeable memory cost, however.

But how powerful is our parallelization as compared to *no* parallelization? Below we plot the serialized `bhkk-pari-0.1` versus our best implementation `bhkk-pari-0.3`, run with 45 threads.









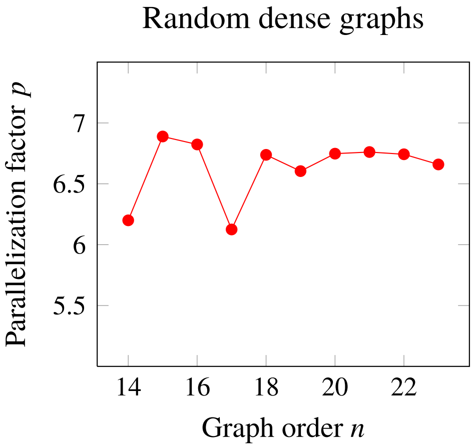
Program	Time (s)	Peak resident set size (kB)
`bhkk-pari-0.1`	10870	$3.48 \cdot 10^4$
HPR	4906	$2.06 \cdot 10^7$
`bhkk-pari-0.3`	1968	$2.68 \cdot 10^5$

Table 3.3: Measurements from a single run on a graph with $n = 25$ and $m = 120$. HPR was allowed a cache of size 5000MB. `bhkk-pari-0.3` was run with 12 threads. See figure B.1 in the appendix for the chromatic polynomial.

Parallelization allows us to increase the order with about 2 and terminate in the same amount of time, on our testing machine. We do pay some memory for this, but as the time gain is so significant, it would almost always be a given to choose the parallelized version.

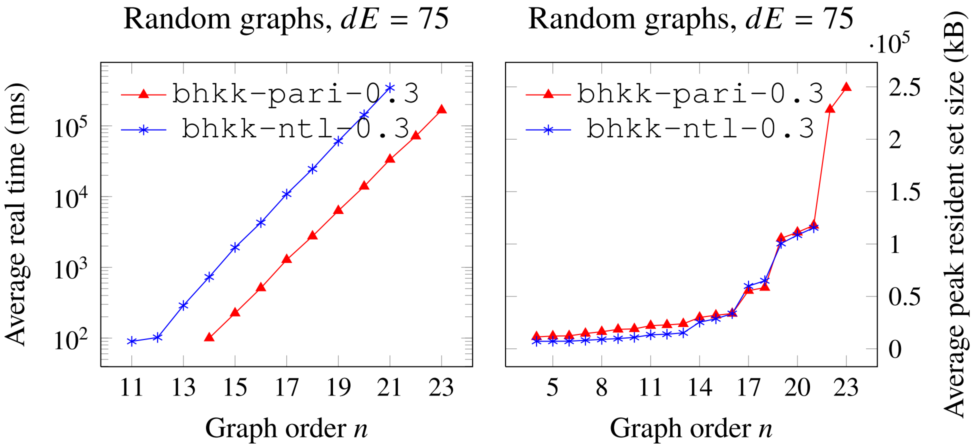
We mentioned in section 2.1.4 that we are able to improve the performance of the algorithm by a factor of around 6 through parallelization. Here we plot the parallelization factor p as the quota of the non-parallelized `bhkk-pari-0.1` and the fastest implementation `bhkk-pari-0.3`. The claim seems to be justified.

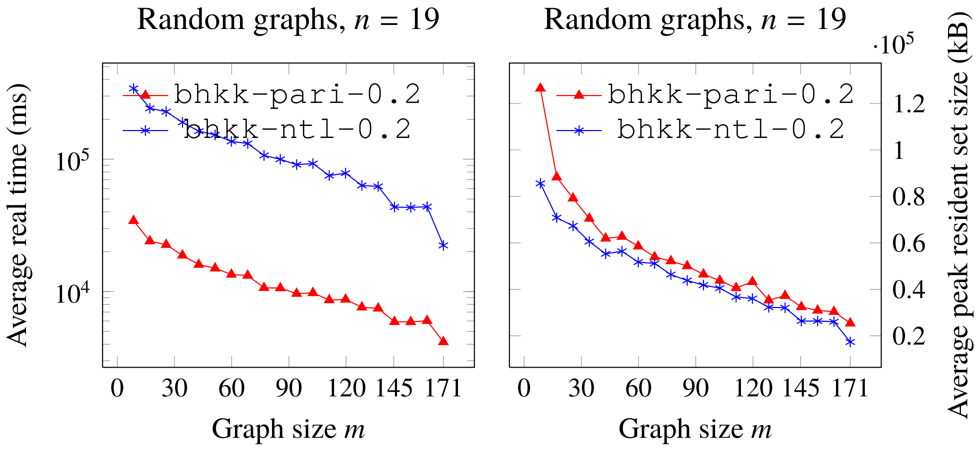
As a final comparison, we refer to table 3.3, where we show our first attempt at defeating HPR, and our eventual success.



3.2.3 Polynomial arithmetic libraries

To provide a little insight in how important the actual implementation of polynomial arithmetic is, we here show a comparison of the implementation linked with PARI and the one linked with NTL.





As is made clear, NTL is too slow to compete. We also did an implementation linked with FLINT, but from initial testing, it did not outperform PARI at any level. For this reason, we did not commence a full simulation suite for that program, as these take quite some time to complete.

3.2.4 Different graph size functions

Since most of our tests have been made on randomized graphs of quadratic size; that is, the size is a function $m = f(n) = O(n^2)$, we are interested if the same behaviour will come up for other size functions. We specify four cases: linear f_{lin}, quadratic f_q, log-linear f_{log}, and quadratic-over-logarithmic f_{qlog}. For reference we include the function for which we have made all other tests, f_{dE}. These are given as follows:

$$\begin{aligned} f_{lin}(n) &= \alpha_{lin} n & \alpha_{lin} &= 2 \\ f_q(n) &= \alpha_q n^2 & \alpha_q &= 0.4 \\ f_{log}(n) &= \alpha_{log} n \ln(n) & \alpha_{log} &= 1 \\ f_{qlog}(n) &= \alpha_{qlog} n^2 / \ln(n) & \alpha_{qlog} &= 0.8 \\ f_{dE}(n) &= \frac{1}{100} dE \cdot \binom{n}{2} = \frac{1}{200} dE \cdot n(n-1) \end{aligned}$$

where the αs are constants, set to provide different values for the functions over the relatively small range of orders that we are able to compute for. Figure 3.1 shows how the sizes grow in our test range.

We run `bhkk-pari-0.3` as BHKK on 50 random graphs of increasing order and with sizes as the specified functions, and compare this to HPR's performance on the same graphs. HPR is here given 10GB cache size.

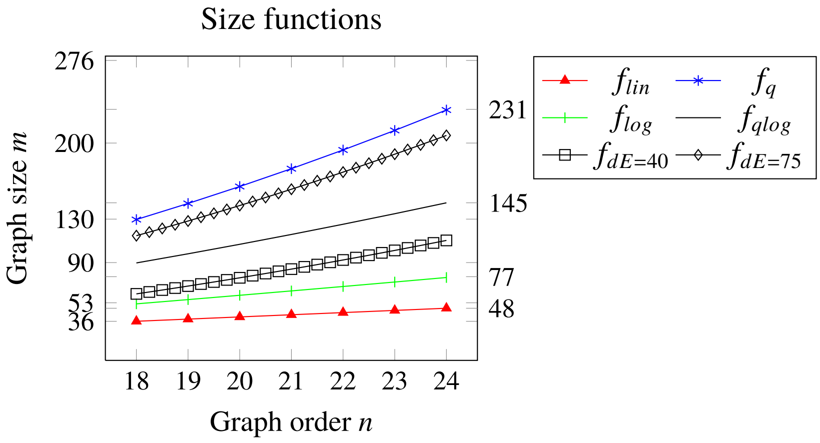
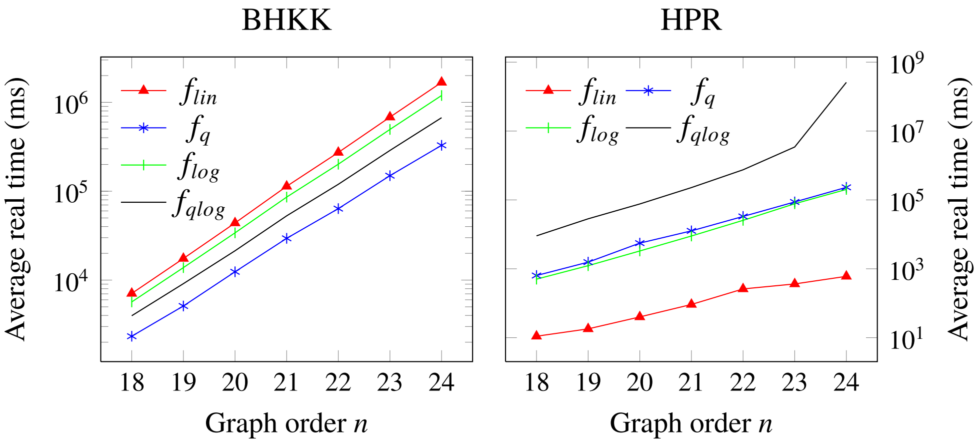


Figure 3.1: The four size functions used in the graph size tests, and the two functions used for most of the other simulations in this report.



For BHKK, the results are somewhat uninteresting, again backing up the conclusion that BHKK simply performs consistently better on larger sized graphs, but otherwise doesn't react on graph size at all. HPR again shows its size dependence, and spikes quite a lot for the f_{qlog} graphs; in fact, the largest ones are only averaged over 4 completed instances since computing these polynomials was hugely expensive.

In figure 3.2 we see BHKK and HPR versus each other on the individual size functions. For both f_q and f_{log}, it is quite clear that HPR increases in time faster than BHKK, but we are unable to decide where exactly BHKK will become the better choice. For f_{lin}, we are unlikely to ever be able to craft a graph where BHKK will be better than HPR, at least in human computing time. The function f_{qlog} actually improves upon the results mentioned in section 3.2.1, showing that already for order 18, BHKK is better than HPR.

The memory plots are included in the appendix, figure B.3. The only thing to note from them is that HPR is so good for the linear-sized graphs that it actually uses less memory than BHKK.

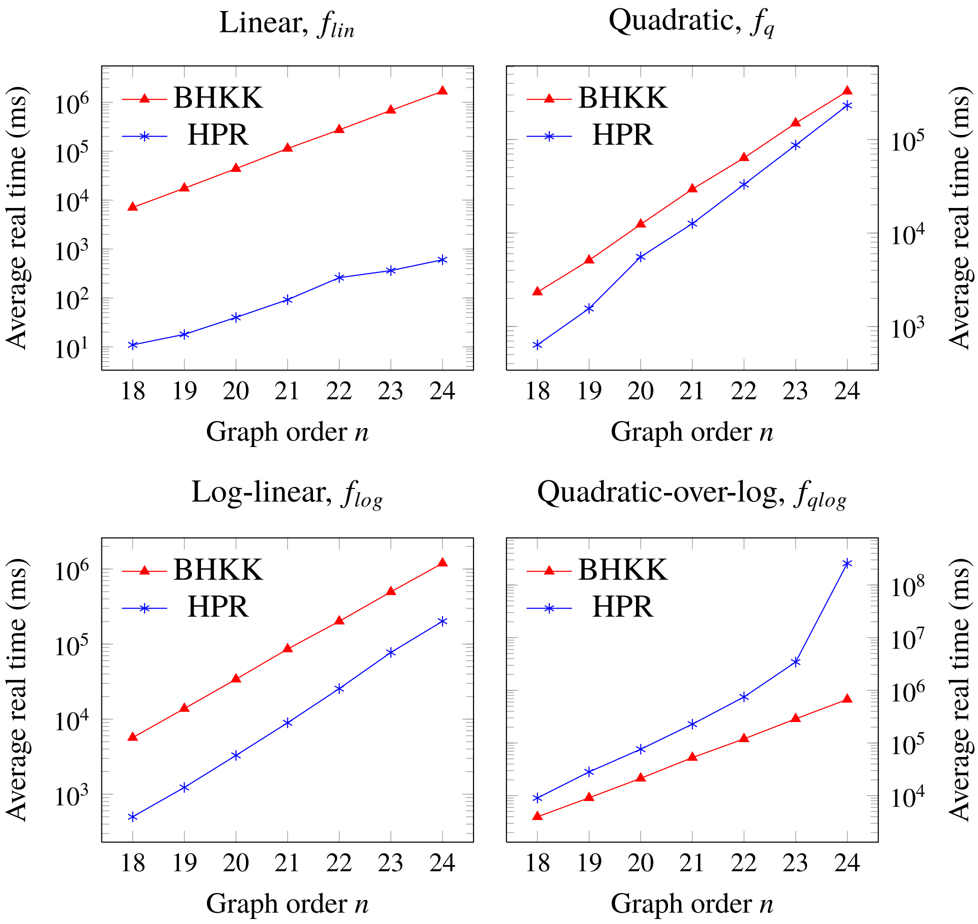


Figure 3.2: Individual size-function tests, time plots.

Chapter 4
Chromatic polynomials for certain graphs

In this chapter we present two well-known graphs and their chromatic polynomials.

4.1 Akbari's graph

Akbari's graph stems from Akbari, Mirrokni and Sadjad [1], who were first to counterprove a certain conjecture of Xu using the graph. Hillar and Windfeldt verified the result using the algorithms they present in [13], where they discuss characterizations of *uniquely* colourable graphs. That is, a graph G with $\chi_G(\chi(G)) = \chi(G)!$, or in other words that there exists only one optimal colouring of G, unique up to interchangability of the colours. Hillar and Windfeldt made an attempt to further 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 4.1, using Maple, as Maple uses a naive implementation of Zykov's [23] deletion-contraction recurrence. 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 (less than half an hour) to compute, using our fastest implementation. HPR however terminated even faster, which was to be expected, given that Akbari's graph has a very small size (density is about 16%).

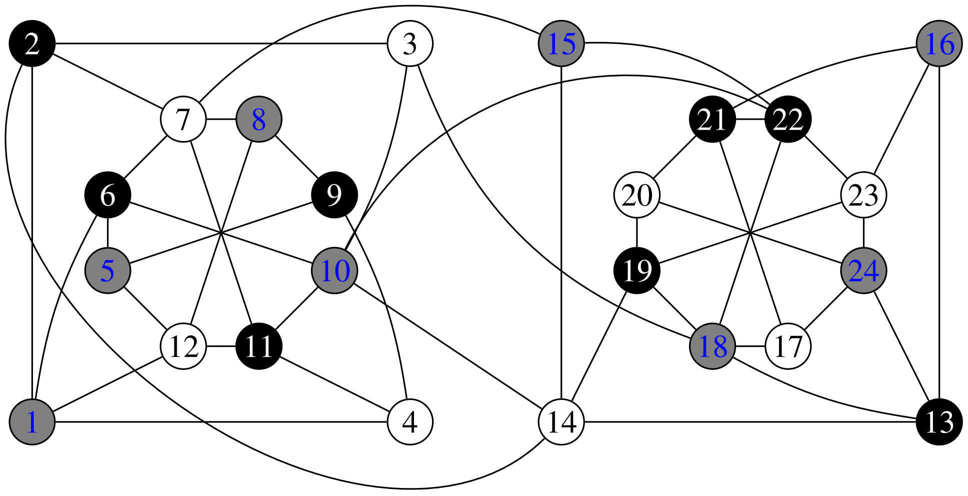


Figure 4.1: Akbari's graph, coloured in its unique 3-colouring with white, gray and black as colours. Figure reconstructed from figure 2 in [13].

4.2 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 chess queen could move, if placed on that square. Here we provide the chromatic polynomial of the 5×5 Queen graph Q_5 on 25 vertices.

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

Polynomial	Algorithm	Real time (s)	Peak resident set size (kB)
χ_{Q_5}	BHKK	1453	199216
χ_{Q_5}	HPR	2727	41094832

Table 4.1: Time and memory measurements on computing the chromatic polynomial of the Queen graph Q_5.

Chapter 5

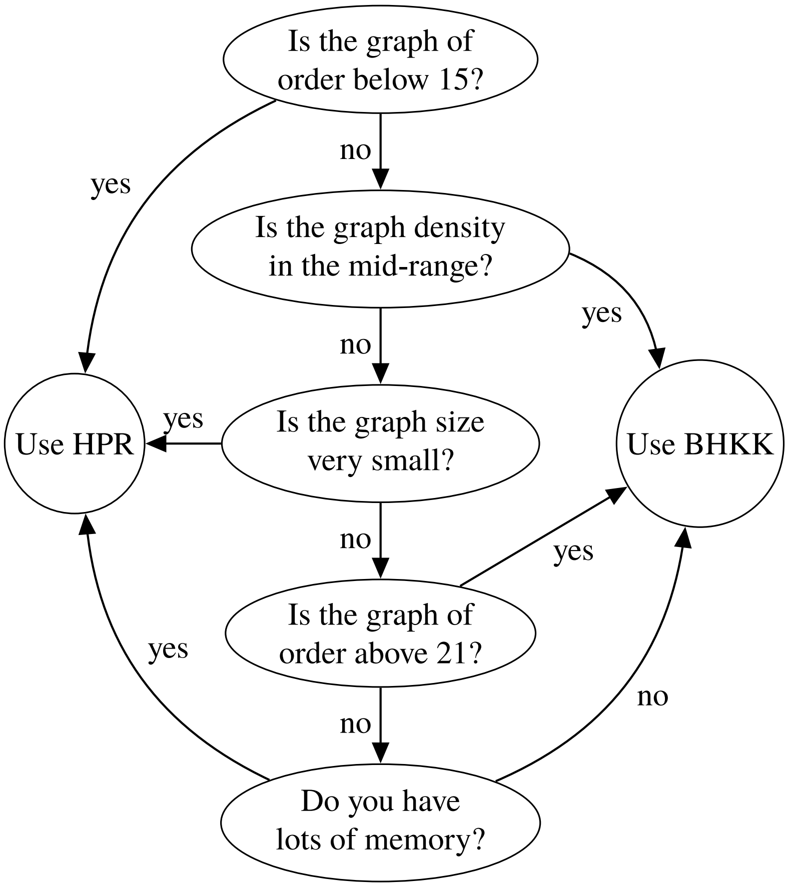
Conclusions

In the first chapter, we presented a summary of the main results that were gathered during the work on this Master's Thesis. In the following chapters, we presented detailed analyses on the results, while not doing any comprehensive reflection on the overall picture painted by their implications. Here we aim to do so.

5.1 Which algorithm should I choose?

Our most important result is of course that which proves the BHKK algorithm outscales the HPR algorithm as graph order increases. But in general, the window where the BHKK algorithm is better than HPR and still not taking forever to terminate is quite small. We were also very generous when deciding the size of HPR's cache, basically because we wished to show that BHKK would still scale better. While no complete tests were performed to back this up, it should be fairly safe to claim that the order bound of 21 could be lowered, if we were to decrease the cache size of HPR.

The size of the graph was of course proven to be very significant for HPR performance, and only somewhat significant for BHKK (relatively speaking). In inprecise terms, we would use this recommendation chart when deciding on which of the two should be chosen to compute the chromatic polynomial:


Is the graph of order below 15?
no
yes
Is the graph density in the mid-range?
yes
no
Use HPR
yes
Is the graph size very small?
Use BHKK
no
yes
Is the graph of order above 21?
yes
no
no
Do you have lots of memory?

5.2 Open questions

There were mainly two questions that we felt should have been answered, but never could be. The first concerns the full utilization of the BHKK algorithm's parallelization capabilities; how fast can we possibly run it, using a network of exponentially many CPUs? We investigated the issue of applying to LUNARC, which is a computing network in Lund for conducting large-scale computations, but unfortunately nothing ever happened with them. Since we were able to speed up the computations by 600% using only twelve CPUs, the thought of speed-up using several hundreds is tickling.

The second question is more implementation-oriented, but also concerns parallelization. As we mention in several places in this report, we select a range of subsets arbitrarily to assign to the virtual threads which then process the algorithm. This range selection is done in the first way we thought of: inorder. The benefit is the ease in which integers encode subsets, and how we are able to simply increment the integers to get to next subset, but there are no guarantees on how spread out the *independent* subsets will be. Indeed, there is good reason to assume this is not an optimal scheme, as low-degree vertices are more likely to be in independent subsets than high-degree vertices, and we don't really spread the vertices. A random distribution of subsets might do better on average. It might even be worthwhile to invest in a more sophisticated algorithm for selecting subset ranges.

Bibliography

[1] S. Akbari, V. S. Mirrokni, and B. S. Sadjad. K_r-free uniquely vertex colorable graphs with minimum possible edges. *J. Comb. Theory Ser. B*, 82(2):316–318, July 2001.

[2] Richard Beigel and David Eppstein. 3-coloring in time $O(1.3289^n)$. *Journal of Algorithms*, 54(2):168 – 204, 2005.

[3] George D. Birkhoff. A determinant formula for the number of ways of coloring a map. *Annals of Mathematics*, 14(1/4):pp. 42–46, 1912.

[4] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, July 2009.

[5] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Covering and packing in linear space. *Information Processing Letters*, 111(21–22):1033 – 1036, 2011.

[6] N. Christofides. An algorithm for the chromatic number of a graph. *The Computer Journal*, 14(1):38–39, 1971.

[7] JoannaA. Ellis-Monaghan and Criel Merino. Graph polynomials and their applications I: The Tutte polynomial. In Matthias Dehmer, editor, *Structural Analysis of Complex Networks*, pages 219–255. Birkhäuser Boston, 2011.

[8] FedorV. Fomin, Serge Gaspers, and Saket Saurabh. Improved exact algorithms for counting 3- and 4-colorings. In Guohui Lin, editor, *Computing and Combinatorics*, volume 4598 of *Lecture Notes in Computer Science*, pages 65–74. Springer Berlin Heidelberg, 2007.

[9] GNU Project. *GNU time, version* `1.7`, 2008.

[10] GNU Project. *The GNU Multiple Precision library, version* `5.1.2`, 2013.

[11] Gary Haggard, David J. Pearce, and Gordon Royle. Computing Tutte polynomials. *ACM Trans. Math. Softw.*, 37(3):24:1–24:17, September 2010.

[12] William Hart, Fredrik Johansson, and Sebastian Pancratz. *Fast Library for Number Theory, version* `2.3.0`, 2012.

[13] Christopher J. Hillar and Troels Windfeldt. Algebraic characterization of uniquely vertex colorable graphs. *Journal of Combinatorial Theory, Series B*, 98(2):400 – 414, 2008.

[14] T. Hubai. The chromatic polynomial. Master's thesis, Eötvös Loránd University, Budapest, Hungary, 2009.

[15] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[16] E.L. Lawler. A note on the complexity of the chromatic number problem. *Information Processing Letters*, 5(3):66 – 67, 1976.

[17] Victor Shoup. *Number Theoretic Library, version* `6.0.0`, 2013. Source documentation.

[18] Victor Shoup. *Number Theoretic Library, version* `6.0.0`. New York University, 2013.

[19] The PARI Group. *PARI/GP, version* `2.5.5`. Université Bordeaux, Bordeaux, 2013.

[20] Hassler Whitney. The coloring of graphs. *Annals of Mathematics*, 33(4):pp. 688–718, 1932.

[21] Wikipedia. Schönhage–strassen algorithm — wikipedia, the free encyclopedia, 2013. [Online; accessed 17-December-2013].

[22] Wikipedia. Toom–cook multiplication — wikipedia, the free encyclopedia, 2013. [Online; accessed 17-December-2013].

[23] A. A. Zykov. On some properties of linear complexes. *Amer. Math. Soc. Translation*, 79, 1952.

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 omitted.

```
/*
 * 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]);
      }
   }
}
```

A.2 The fast zeta transform

The exponential-space fast zeta transform is used in both the linear-space FZT and the BHKK algorithm for chromatic polynomials. It is in a sense the core of both algorithms, and as such we reference its quite straight-forward implementation as a C++ function here.

```
/*
 * This is the 'trivial' Fast Zeta Transform, in exponential time
 * and space. The vector f contains sets in binary code, with
 * each element in f corresponding to the function value of the
 * set whose bitset is the index of the element.
 * Example: the set {1,3,4} is encoded as 11010 (for 5 elements),
 * and its function value is f(11010) = f(26).
 *
 * The function iterates over all subsets of each set and sums
 * the function values, overwriting the existing vector elements
 * with the sums.
 *
 * This function works as a utility function for the Fast Zeta
 * Transform in Linear Space algorithm.
 */
void utils::fast_down_zeta_transform_exp_space(
   const u_int_t& n,
   rval_list_t& f)
{
   for (u_int_t j = U_ONE; j <= n; ++j) {
      u_int_t index = U_ZERO;
      u_int_t step = exp2(j - 1);   // 2^(j-1)
      while (index != f.size) {
         index += step;
         for (u_int_t i = U_ZERO; i < step; ++i) {
            add_assign(f[index], f[index - step]);
            ++index;
         }
      }
   }
}
```

A.3 The linear-space fast zeta transform

Here is listed the direct implementation of the fast zeta transform in linear space. Refer to section 2.1.1 for an algorithm outline. The code here is similar to the code listed in A.1, as that code evolved from this in similar ways that the BHKK algorithm evolved from the linear-space fast zeta transform. In the inner-most loop, the *q*-cover, *q*-partition and *q*-packing implementations differ, as the summing mechanism is the only thing that separate these algorithms.

```
/*
 * This function calculates the Fast Zeta Transform in Linear Space for a given
 * problem type.
 *
 * n is split into n1 and n2, where n = n1 + n2 is assumed to hold.
 * family is a pointer to a list of the sets to be used for covering. Each
 * member of the list is a set represented as a bit vector. A 1-bit in
 * position i means that i is a member of the set.
 * f represents a function with a one-to-one mapping from each member of
 * family to a value in an algebraic ring.
 */
void utils::fast_zeta_transform_linear_space(
      int_t n1,
      int_t n2,
      int_list_t* family,
      rval_list_t* f,
      int_t q,
      rval_t* pk)
{

   int_t two_to_the_n1 = pow(2, n1);      // 2^n1
   int_t two_to_the_n2 = pow(2, n2);      // 2^n2

   int_t u1 = two_to_the_n1 - 1;         // Index of U1
   int_t u2 = pow(2, n1+n2) - two_to_the_n1;   // Index of U2

   // Array g
   rval_list_t g(two_to_the_n2);

   // For each subset X1 of U1
   for (int_t x1 = 0; x1 < two_to_the_n1; ++x1) {

      // For each Y2 in U2, set g(Y2) <- 0
      for (int_t i = 0; i < two_to_the_n2; ++i) {
         g[i].set_degree(n1 + n2);
      }

      // For each Y in F, if YnU1 is a subset of X1,
      // then set g(YnU2) <- g(YnU2) + f(Y)
      for (int_t i = 0; i < family->size; ++i) {
         int_t y = (*family)[i];

         // if YnU1 is a subset of X1
         if (((y & u1) | x1) <= x1) {
            g[(y & u2) / two_to_the_n1] += (*f)[i];
         }
      }

      // Compute h <- gS
      fast_zeta_transform_exp_space(n2, &g);

      // For each subset X2 of U2,
      // calculate sum according to problem
      for (int_t i = 0; i < two_to_the_n2; ++i) {
         int_t x2 = i * two_to_the_n1;
         int_t x = x1 | x2;

         // Sum differently dependent on problem
         // Below is the code for q-packing

         int_t size_of_X = count_1bits(x);

         g[i].raise_to_the(q);
         Polynomial p;
         p.set_degree(n1 + n2);
         mpz_set_ui(p[0], 1);
         mpz_set_ui(p[1], 1);      // p = 1 + z
         p.raise_to_the(size_of_X);
         p *= power_of_minus_one(n1 + n2 - size_of_X);
         g[i] *= p;

         (*pk) += g[i];
      }
   }
}
```

Appendix B
Additional figures

Here are listed diagrams and other material that are referenced in the report, but are not included there because of space and layout considerations. They are included here for completeness.

$$\begin{aligned}\chi_{G^{25}}(t) = {} & t^{25} - 120t^{24} + 7004t^{23} - 264466t^{22} + 7248239t^{21} - 153344983t^{20} \\ & + 2600048319t^{19} - 36209438705t^{18} + 421105990283t^{17} - 4135955829674t^{16} \\ & + 34560766890437t^{15} - 246783380030865t^{14} + 1508631206050028t^{13} \\ & - 7892432249185411t^{12} + 35242762914776770t^{11} - 133668888705602908t^{10} \\ & + 427399153791644727t^{9} - 1139874471086088779t^{8} \\ & + 2498692681846853463t^{7} - 4411354147772665522t^{6} \\ & + 6094947387024502214t^{5} - 6317297993651360713t^{4} \\ & + 4592536463051011507t^{3} - 2072444260206668934t^{2} \\ & + 432421085055458088t\end{aligned}$$

Figure B.1: The chromatic polynomial of a randomized graph G^{25} of order 25 and size 120. The chromatic number is $\chi(G^{25}) = 5$. See table 3.3 for more information.

$$
\begin{aligned}
\chi_{G^{30}}(t) = t^{30} &- 327t^{29} + 51615t^{28} - 5233424t^{27} + 382686517t^{26} - 21479975909t^{25} \\
&+ 961793250522t^{24} - 35248963540057t^{23} + 1076534196021453t^{22} \\
&- 27753653037507059t^{21} + 609633596882748756t^{20} \\
&- 11486014951366426648t^{19} + 186473402834492799447t^{18} \\
&- 2616170388871658868193t^{17} + 31763619843500728892382t^{16} \\
&- 333762149166840764397819t^{15} + 3031469401592279292652750t^{14} \\
&- 23739639439353535507986356t^{13} + 159656766610016051128502971t^{12} \\
&- 917061168312033994004975248t^{11} + 4465924798641656883216311606t^{10} \\
&- 18261928695591046730177964536t^{9} + 61924040048734731870453966178t^{8} \\
&- 171278125425411134681457922876t^{7} + 378017715608264599043675759506t^{6} \\
&- 645775990626321014847788012220t^{5} + 817025553561262732499772944776t^{4} \\
&- 714249827411440616685730920528t^{3} + 380830529862878957731898637120t^{2} \\
&- 91919474710149683725971494400t
\end{aligned}
$$

Figure B.2: The chromatic polynomial of a randomized graph G^{30} of order 30 and size 327. Computing this took 46318.23 seconds using a peak resident set size of 741184 kB. `bhkk-pari-0.3` with 12 threads was used. The chromatic number is $\chi(G^{30}) = 11$.

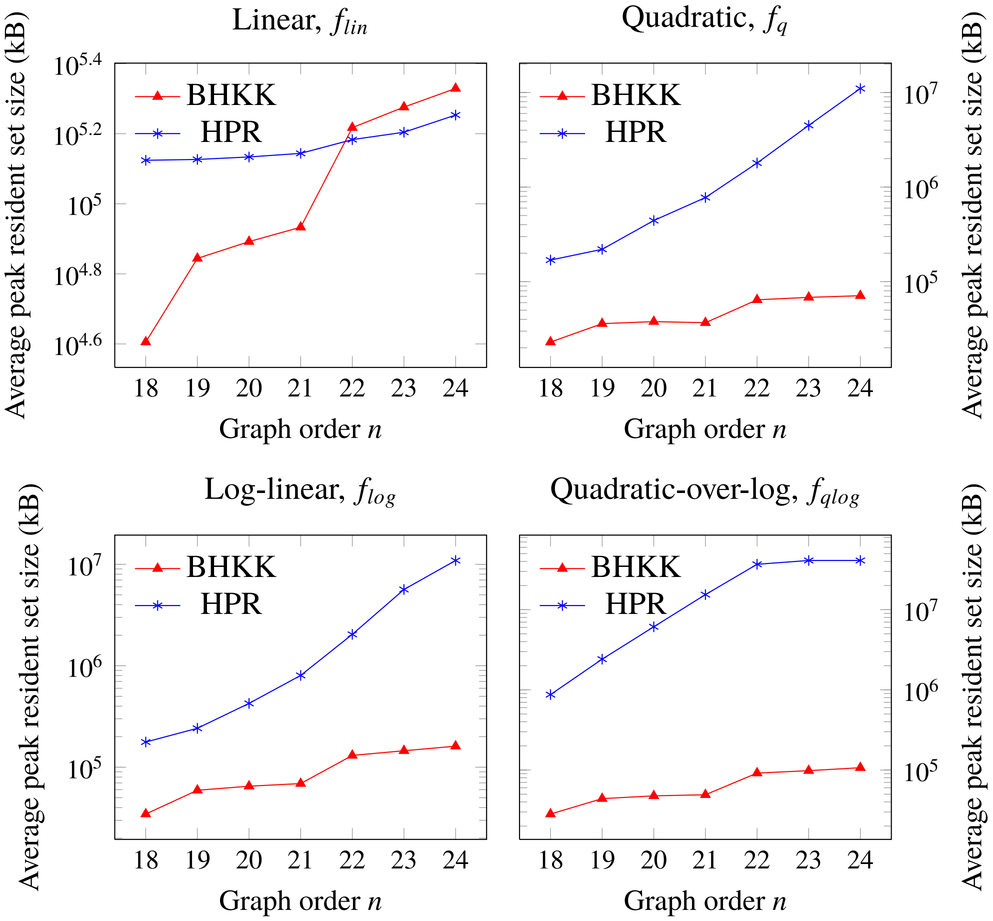


Figure B.3: Individual size-function tests, memory plots. Refer to section 3.2.4 for context and details.

Appendix C
Seminar paper

A short version of the problem statement and results of this Master's Thesis work resulted in a 12-page paper which was handed in to the SEA seminar in Copenhagen. We attach that paper as an appendix to this report for completeness, even though the reader will find much redundancy between it and what has been stated and shown above. (The format and page numbering is independent in this document.)

Chromatic Polynomials in Small Space – Performance in Practice

Mats Rydberg*

October 15, 2014

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 each other 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.

*Master's Thesis student at the Department of Computer Science, Faculty of Engineering (LTH) at Lund University, Lund, Sweden. E-mail: `dt08mr7@student.lth.se`. This work was conducted as a part of the authors Master's Thesis project. Supervisor: Thore Husfeldt, prof. in Comp. Science at LTH.

1 Introduction

A *proper q-colouring* of a graph $G = (V, E)$ is a mapping $\sigma : V \to [q]$ where $[q] = \{1, 2, \ldots, q\}$ such that $\sigma(v) \neq \sigma(w)$ for each $vw \in E$. In other words, σ is an assignment of a *colour* to each vertex v such that no two adjacent vertices get the same colour. The number of ways to q-colour G is $P(G, q)$, and the *chromatic polynomial* $\chi_G(t)$ of a graph G passes through each point $(q, P(G, q))$. In other words, it *counts* the number of ways to colour G for any amount of colours. In particular, the *chromatic number* is the smallest c for which $\chi_G(c) > 0$. The polynomial $\chi_G(t)$ is of high interest in the field of algebraic graph theory, as one of the main graph invariants. Graph colouring is a canonical NP-hard problem, but it also has practical applications for register allocation, scheduling, pattern matching, and in statistical physics, the chromatic polynomial occurs as the zero-temperature limit of the antiferromagnetic Potts model (see for instance [4], [8], and [10]).

1.1 Previous work

The chromatic polynomial was specified in 1912 by Birkhoff [2], 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. In 2010, Haggard, Pearce and Royle [6] published a program (referred to here as **HPR**) to compute the Tutte polynomial for graphs using a deletion-contraction algorithm. The basic idea behind HPR goes back to Zykov [13], using cached subgraphs and isomorphism rejection to obtain good performance, and it 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 [3] 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.

1.2 Results

In this paper, we present an implementation of the BHKK algorithm and experimental results from running it on selected classes of graphs. In particular, we show that for random graphs with $|V| > 21$, the BHKK algorithm performs consistently better than the HPR algorithm, both in time and space consumption. For practically all nonempty graphs BHKK consumes less memory. Our results also show that in practice, the implementation of polynomial arithmetic is crucial, and has a large impact on overall performance.

Furthermore, we give the chromatic polynomial of a graph, here referred to as Akbari's graph from Akbari, Mirrokni and Sadjad [1], discussed in Hillar and Windfeldt [7], which Maple was unable to compute. We also give the chromatic polynomial of the queen graph of size 5×5 (on 25 vertices).

2 The algorithm

The algorithm implemented and measured in this paper is described in Björklund *et al* [3], and is based on a linear-space Fast Zeta Transform described in the same paper.

It is proven to perform in time $O^*(2^n)$ and space $O^*(1.2916^n)$ under certain design parameters. The measured theoretical unit of time is the time it takes to perform an addition or a multiplication of two polynomials of degree n. The measured theoretical unit of space is the memory needed to store such a polynomial. In practice, we can not discern this space and time from the space and time used by other parts of needed data structures, but since these dominate, we can assume that the measured space and time usage will converge asymptotically with the theoretical bounds.

Our input is an undirected graph G on n vertices with m edges[1]. The main subroutine counts the number of ways to colour G using q colours. This is done for $q = 0, 1, \dots n$, yielding $n + 1$ points (x_i, y_i). These are by definition points which the chromatic polynomial $\chi_G(t)$ passes through. $\chi_G(t)$ has exactly degree n, and so we have enough information to recover it explicitly (i.e., specifying its coefficients) using interpolation.

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 [3, sec. 5].

The full algorithm is found specified in [3, p. 9]. In short, we count the q-colourings for each $q \leq n$ independently, and interpolate on the resulting points. When counting colourings, we use an array of polynomials of degree q, constructed by analyzing independent subsets. An important aspect is that the main loop executes for subsets of V_1, independently from each other. This allows us to execute all these loop iterations in parallel.

2.1 Optimizations

Here are presented some improvements to the algorithm that are either natural, mentioned in [3], or invented by the author. This list is by no means exhaustive, nor is every item critical, but the ones we have explored proved to be efficient.

Exploiting q First, we can consider optimizing on the basis of the value of q.

- For $q = 0$, there are 0 colourings, as no graph can be 0-coloured.
- For $q = 1$, there are 0 colourings if and only if $|E| > 0$, otherwise there is exactly 1 colouring. This takes $O(n^2)$ time to check.
- For $q = 2$, it is well-known that the graph can be coloured (or found to be non-colourable) in polynomial time using standard techniques (such as depth-first search).

Using $\omega_{min}(G)$ A more sophisticated type of optimization involves exploiting the clique number $\omega(G)$, which is a lower bound on the chromatic number $\chi(G)$. Knowing

[1] Multiple edges and self-edges are two types of edges that often need special treatment in graph-based algorithms. This is not the case for graph colouring. Any self-edge means the graph is not colourable (a vertex would need to have a different colour from itself), so in this implementation we assume they do not exist. Any multiple edge doesn't affect the problem at all, as we are merely considering the *existence* of an edge between two vertices; if there are more than one that doesn't matter.

that $\omega(G) \geq a$ for some constant a would allow us to omit counting colourings for all $q < a$. If $a = n$, we have the complete graph K_n, for which $\chi_G(t)$ is known.

Here we define the *density* of a graph G as $dE = m/\binom{n}{2}$, where m is the number of edges in G. This immediately tells us the *smallest possible* $\omega(G)$. Let us call it $\omega_{min}(G)$. In fact, the following holds:

Lemma 1. *The number $\omega_{min}(G)$ is the lower bound of $\omega(G)$, and its value is*

$$\omega_{min}(G) = \begin{cases} n - \binom{n}{2} + m & \textit{if } m \geq \binom{n}{2} - \lfloor n/2 \rfloor \\ \lceil n/2 \rceil - a & \textit{if } \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil < m < \binom{n}{2} - a\lfloor n/2 \rfloor, a \in \mathbb{N}_+, a \textit{ maximal} \\ 2 & \textit{if } 0 < m \leq \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \\ 1 & \textit{if } m = 0 \end{cases}$$

Proof. Trivially, $w(N_n) = 1$. This proves the lower-most bound.

Next, consider the complete bipartite graph $K_{\alpha,\beta}$ on n vertices; it is the graph with the most edges that has clique number 2. It is well-known that it has $\alpha\beta$ edges. To maximize this product, we make a half-half partition, setting $\alpha = \lfloor n/2 \rfloor$ and $\beta = \lceil n/2 \rceil$, giving $\alpha\beta = \lfloor n/2 \rfloor \lceil n/2 \rceil$. The point made is that there is no way of assigning more edges than this without yielding a higher clique number.

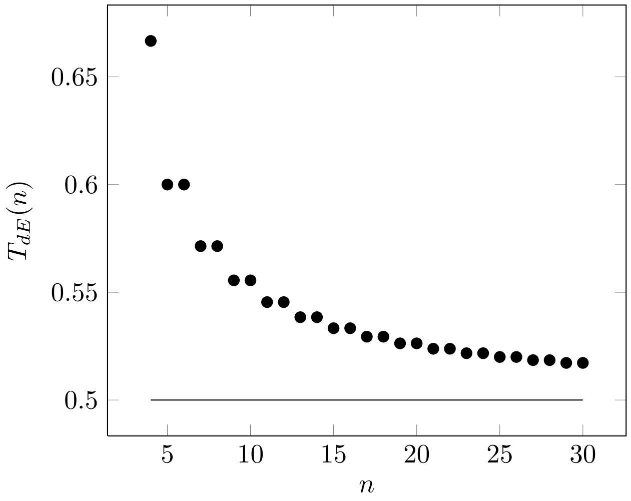
Third, consider the complete graph K_n, with $w(K_n) = n$. Deleting an edge vw will clearly reduce the clique number. Consider the subgraph $K \setminus \{v\}$; it has clique number $n - 1$. Removing an edge $uu', u \neq w, u' \neq w$ will lower clique number again by 1. This process may be repeated $\lfloor n/2 \rfloor$ times. The resulting graph has $\binom{n}{2} - \lfloor n/2 \rfloor$ edges and clique number $n - \lfloor n/2 \rfloor$. This, together with the fact that removing one edge can lower clique number by maximum one, proves the uppermost bound. □

For the final case, for which we do not provide a full proof, we observe that $\binom{n}{2} - \lfloor n/2 \rfloor - \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ is a multiple of $\lfloor n/2 \rfloor$ (this is the variable a in the lemma). It can be shown that this corresponds to the requirement of removing $\lfloor n/2 \rfloor$ edges to reduce clique number by one.

As we can see from Lemma 1, only graphs with $m > \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ provides $\omega_{min}(G) > 2$ and for $q \leq 2$ we already have good optimizations. So how dense is a graph where this bound on m holds? Let us specify the threshold density $T_{dE}(n)$ as

$$T_{dE}(n) = 2\frac{\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil}{n(n-1)} = \begin{cases} \frac{1}{2}n(n-1)^{-1} & \text{if } n \text{ even} \\ \frac{1}{2}(n+1)n^{-1} = T_{dE}(n+1) & \text{if } n \text{ odd} \end{cases}$$

In conclusion, any graph with $dE > T_{dE}(n)$ can omit the computation of the number of q-colourings when $q \leq w_{min}(G)$. It also follows that as $n \to \infty$ we will have $T_{dE}(n) \to \frac{1}{2}$. The following plot shows how fast we converge for graphs of sizes relevant for this paper.



For larger graphs, we have a smaller $T_{dE}(n)$, which gives us a higher probability to be able to optimize, and it is also for larger graphs that we are most interested in optimizing techniques. For a graph with $n = 23$ and $dE = 75$, we would be able to skip evaluating $q \leq 7$, which yields a decrease in execution time by about 15%[2].

Parallelization 1 Potentially, we could parallelize the algorithm using $2^{|V_1|} = O^*(1.5486^n)$ CPUs. This would yield significant time improvements in theory, reducing the asymptotic time bound to about $O^*(1.5486^n)$. Using a different partition of V with $n_1 = n_2$, we would achieve space and time bounds of $O^*(2^{n/2})$ on as many CPUs [3].

Parallelization 2 Typically, we will only have access to a constant number of CPUs in practice, allowing each of them to not execute one loop iteration but a range of iterations for a range of subsets. This allows for heuristics on how to select such ranges so that the overall time bound (set by the range of subsets that include the *most* independent subsets) is minimal. The currently used heuristic is to simply take the subsets in inorder. As presented below, we can expect to reduce the time consumption of the program by a factor of around 6.

Degree pruning When multiplying and exponentiating our polynomials, we increase their degree, potentially as large as qn (we exponentiate with q). But since we never do any divisions, and never care about coefficients for terms of degree $> n$, we can simply discard these terms, keeping degrees $\leq n$.

3 Algorithm performance parameters

The algorithm has some perks that make it perform better or worse for different input. In this section we aim to explore a few of these characteristics.

[2] This number is based on experimental results presented below.

3.1 Sparse and dense graphs

The algorithm in itself is designed in a way that allow for a smaller degree of complexity for *dense* graphs. 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 is a direct consequence of the content of the polynomial arrays, which contain non-zero for each *independent* subset considered (see [3, p. 9, steps 2b and 2c]). As graph density increases, fewer subsets of the vertex set will be independent, and more polynomials will be equal to zero. 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.

4 Experimental results

This section presents selected results in four parts. First, we show the metrics for the best implementation currently available, and discuss how it relates to the theory described above. Second, we compare it to some of the other implementations, to visualize the impact of some development choices. Most importantly, we make a comparison between which library for polynomial arithmetic was used. Thirdly, we compare our results to the Haggard-Pearce-Royle implementation. We end by providing the chromatic polynomials for two small Queen graphs and a graph discussed by Hillar and Windfeldt in [7].

For the first two parts, the tests are performed on randomized graphs, generated for some values of n and dE using a tool developed by the author. The process is basically to fill an array A of length $\frac{1}{2}n(n-1)$ with m ones and rest zeroes, shuffle A using Fisher-Yates shuffle and then add the edge $v_i v_{i+1}$ to the graph if $A[i] = 1$.

4.1 Measurements

All tests are performed on the same machine, with the following specifications.

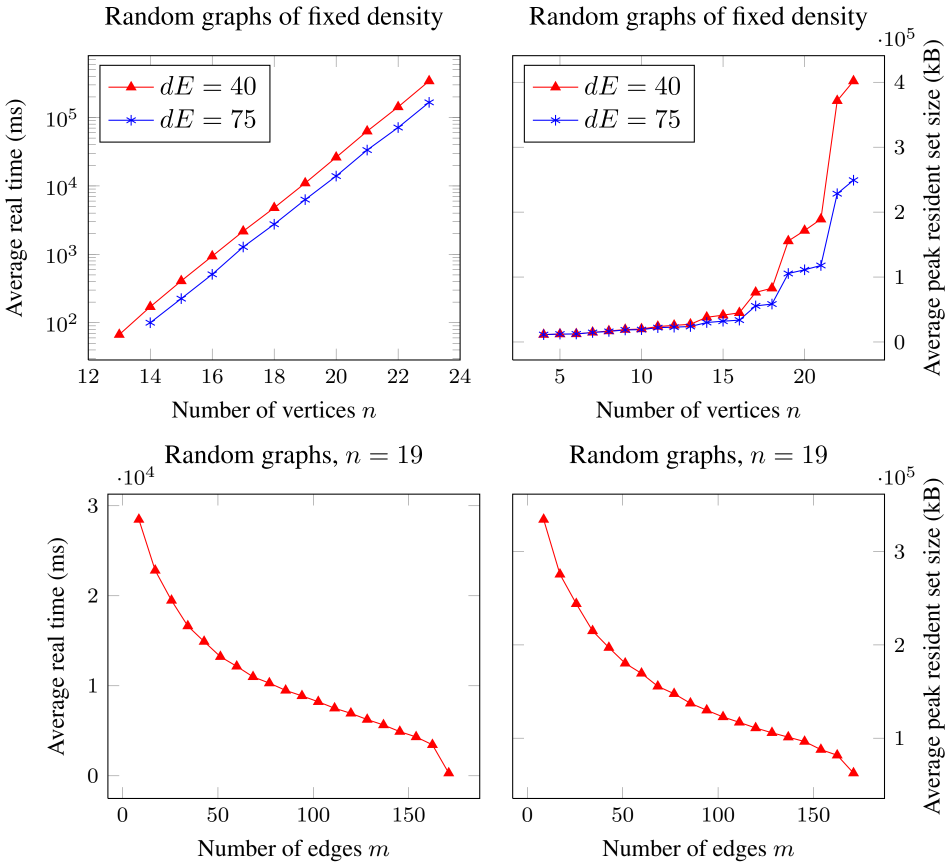
CPU (cores, threads)	Intel i7-3930K 3.2GHz (6, 12)
OS	GNU/Linux 3.8.13.4-desktop-1.mga3 (Mageia 3) x86_64
Memory	16GB DDR3 1333Mhz
Compiler	GCC 4.7.2 (g++)

For all time and memory measurements, the GNU `time` 1.7 program is used (see the Linux man pages [5]). The user time, elapsed time (real time) and peak resident set size (in kilobytes) are the data points recovered as measurements. These measurements are taken by running the specified program on a number[3] of graphs of equal size and the average values are the ones presented.

4.2 `bhkk-pari-0.3`

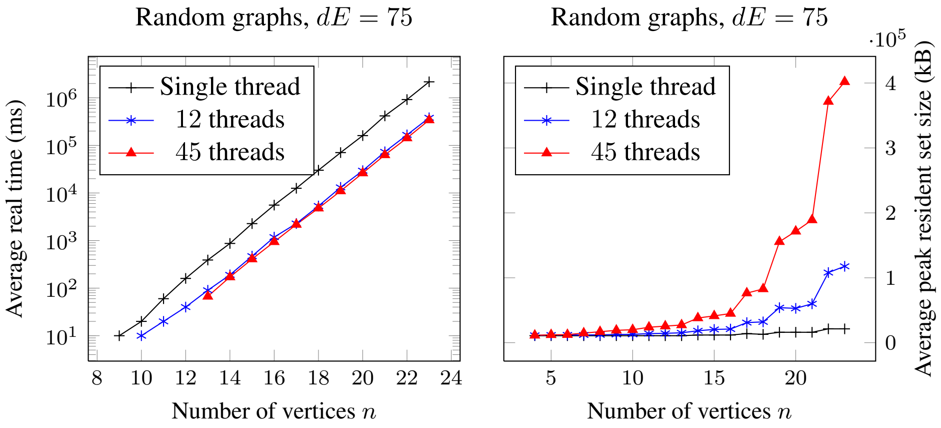
The most powerful implementation is `bhkk-pari-0.3`. It implements these optimizations: $q = \{0, 1\}$, $w_{min}(G)$, parallelization 2, degree pruning. We show here its time and memory consumption in relation to both n and m.

[3] Usually, this is 50 or 100. Smaller for larger graphs because of time restrictions.

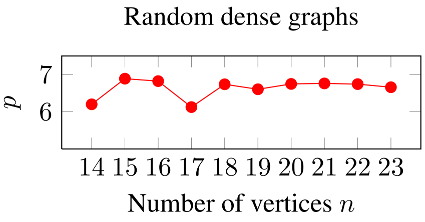


4.2.1 Power of parallelization

On the test machine, the actual parallelization width is 12. But using some waiting threads to take over when the fastest threads have terminated seems to be a smarter approach. Actually, we can show that doing so will lower our time requirements by a few percent, but it comes at a very large cost in memory.

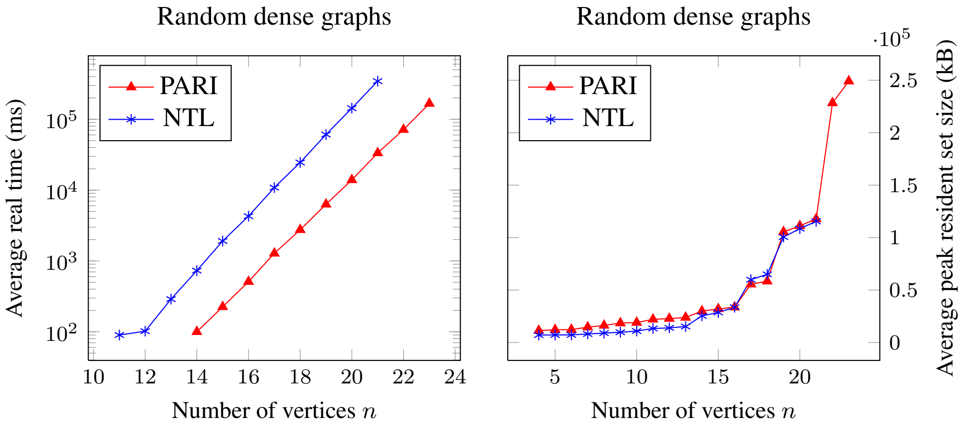


As mentioned above, parallelization (of width 12) allows us to terminate about six times faster. Here we plot the parallelization factor p as the quota of the non-parallelized `bhkk-pari-0.1` and the fastest implementation `bhkk-pari-0.3`.



4.2.2 Polynomial arithmetic library performance

The most common operation in the algorithm is multiplying two polynomials. Thus the implementation of that operation is critical to performance. In this paper, we omit a more detailed description of the implementation techniques used for the BHKK algorithm, but two reference implementations were made, one linked with the PARI library [11], and one with the NTL library [9]. To provide a little insight in how important the actual implementation of polynomial arithmetic is, we also show a comparison of the implementation linked with PARI and the one linked with NTL.



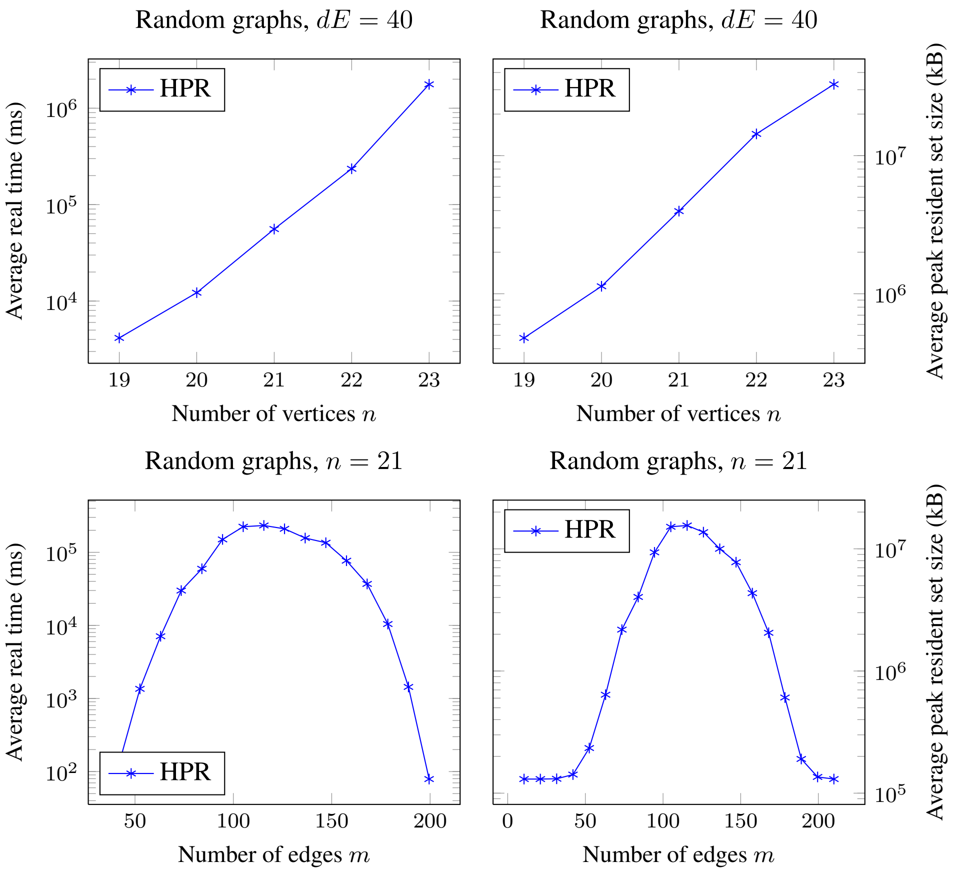
As is made clear, NTL is too slow to compete. This of course tempts the question whether there are even faster libraries to use.

4.3 Relation to other work

As comparison throughout the development process, we have consistently used HPR. Initially to make sure that the output was the same given same input (and hence, since HPR is a published tool by renowned authors, correct), and eventually, to "race". From performance tests that were made, HPR seems to do better if the input has "some kind of structure", rather than being "random". This is manifested by it performing considerably well on any "famous" graph (which usually is famous because such graphs appeal to some "human sense" of pattern-recognition). But in the case of graphs generated according to the randomized process mentioned here, HPR does not scale as well as BHKK, and has a "weak spot" for graphs of density around 75 [6]. More pressingly, HPR does not give good worst-case performance. Our measurements show a large

($> 95\%$) variance[4] in performance on different graphs of equal size. This is not the case with BHKK, which has near-deterministic ($< 15\%$ variance) computation time from given size of input.

The main improvement of BHKK is of course the memory consumption. HPR relies on the use of a cache for recognition of isomorphic graphs in order to perform well. In (most of) these test runs we have allowed HPR to use a lot of cache, about 10GB. This is because we wish to show that BHKK can run faster even when HPR is not directly limited by its cache size.



These results are somewhat in contrast to those presented in [6]. The data here suggests that the ”weak spot” in terms of density would be around 50%, whereas [6] suggests ”weak spot” around 75%. It is made clear however, that BHKK shows better asymptotic behaviour, and that the space improvements are significant, also in practice. Note that the first test (upper graphs) were performed on a sparse graph, meaning BHKK is not working under optimal conditions.

[4] With variance, we mean the number $1 - m/M$, where M is the maximum measured value and m the minimum.

	CPU time (s)	Real time (s)	Peak resident set size (kB)
$n = 25$, $dE = 75$			
`bhkk-pari-0.2`	9,492	941	$1.46 \cdot 10^5$
HPR	-	> 260,000	-
$n = 30$, $dE = 75$			
`bhkk-pari-0.2`	556,895	53,834	$7.41 \cdot 10^5$
HPR	> 94,000	> 95,000	$4.11 \cdot 10^7$

Table 1: Larger instances. Results are based on a single run on one graph. HPR did not terminate within the times specified here. `bhkk-pari-0.2` implements $q = \{0, 1\}$, parallelization 2 and degree pruning.

5 Some chromatic polynomials

Here we provide the actual chromatic polynomials of a few famous graphs, together with data on how expensive these polynomials were to compute.

5.1 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 made 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 1, using Maple, as Maple uses a naive implementation of Zykov's [13] deletion-contraction recurrence. 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, $\chi_G(3) = 3!$, as expected. This took 1445 seconds (less than half an hour) to compute, using our fastest implementation. HPR however terminated even faster, which was to be expected.

5.2 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 that square. Here we provide the chromatic polynomial of the 5×5 Queen graph Q_5, on 25 vertices.

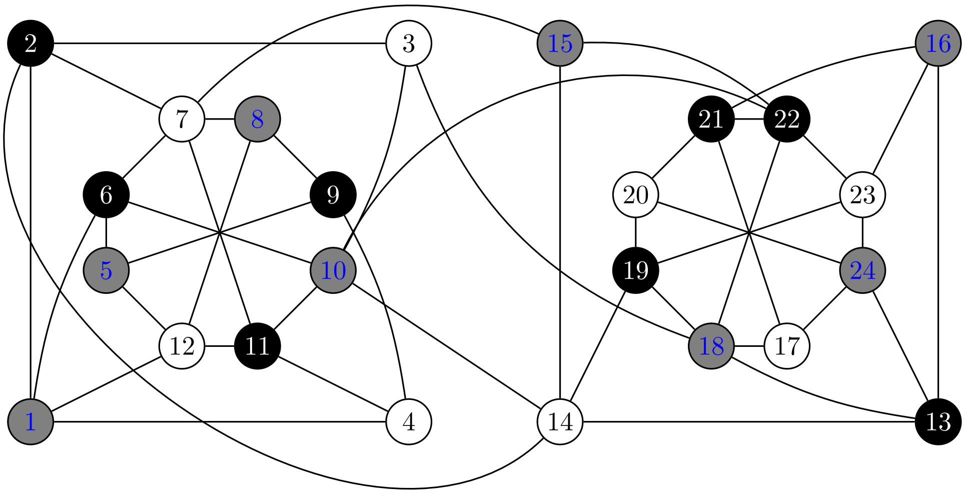


Figure 1: Akbari's graph, coloured in its unique 3-colouring with white, gray and black as colours. Figure copied from figure 2 in [7].

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

Algorithm	Real time (s)	Peak resident set size (kB)
BHKK	1453	199216
HPR	2727	41094832

Table 2: Time and memory measurements on computing the chromatic polynomial of Q_5.

References

[1] Akbari, S, Mirrokni, V S, Sadjad, B S, "K_r-free uniquely vertex colorable graphs with minimum possible edges", *J. Comb. Theory Ser. B* **82** (2) (2001), 316–318, ISSN 0095-8956, doi:10.1006/jctb.2000.2028.
URL `http://dx.doi.org/10.1006/jctb.2000.2028`

[2] 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`

[3] 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`

[4] Chaitin, G, “Register allocation and spilling via graph coloring”, *SIGPLAN Not.* **39** (4) (2004), 66–74, ISSN 0362-1340, doi:10.1145/989393.989403.
URL `http://doi.acm.org/10.1145/989393.989403`

[5] GNU Project, *GNU time, version* `1.7`, (2008).
URL `http://man7.org/linux/man-pages/man1/time.1.html`

[6] 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`

[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] Marx, D, “Graph coloring problems and their applications in scheduling”, in *In proc. John von Neumann PhD students conference* (2004) pp. 1–2.

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

[10] Sokal, A D, “Chromatic polynomials, Potts models and all that”, *Physica A: Statistical Mechanics and its Applications* **279** (1–4) (2000), 324 – 332, ISSN 0378-4371, doi:http://dx.doi.org/10.1016/S0378-4371(99)00519-1.
URL `http://www.sciencedirect.com/science/article/pii/S0378437199005191`

[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] Zykov, A A, “On some properties of linear complexes”, *Amer. Math. Soc. Translation* **79**, ISSN 0065-9290, .