Performance measurements of a small-space Chromatic Polynomial algorithm

Mats Rydberg*

January 29, 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 eachother and show our performance against an implementation done by Haggard, Pearce and Royle from 2010. We also present the chromatic polynomials for a small Queen graph and a certain graph specified by Hillar and Windfeldt.

*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], [9], and [12]).

1.1 History

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 [14], and Tutte incorporated it into what is now known as the Tutte polynomial. In 2010, Haggard, Pearce and Royle [7] published a program (referred to here as **HPR**) to compute the Tutte polynomial for graphs using a deletion-contraction algorithm. HPR exploits the isomorphism of induced subgraphs to obtain good performance, and can easily handle many instances of non-trivial sizes. Using the fact that the Tutte polynomial encodes the chromatic polynomial (as well as other graph invariants), HPR is also designed to output $\chi_G(t)$. In 2011, Björklund, Husfeldt, Kaski and Koivisto [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.

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 $|V| > 22$, 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 [8], 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 $q = 0, 1, \ldots n$, 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 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, $f\zeta(U)$, given as

$$f\zeta(U) = \sum_{Y \subseteq U} f(Y)$$

[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.

denotes the fast down-zeta transform of a function f, and $f\zeta'(U)$, given as

$$f\zeta'(U) = \sum_{Y \supseteq U} f(Y)$$

denotes the fast up-zeta transform of f (see [3, sec 2] for the full algorithm). 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 [3, p 9].

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've 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.

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*

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

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.

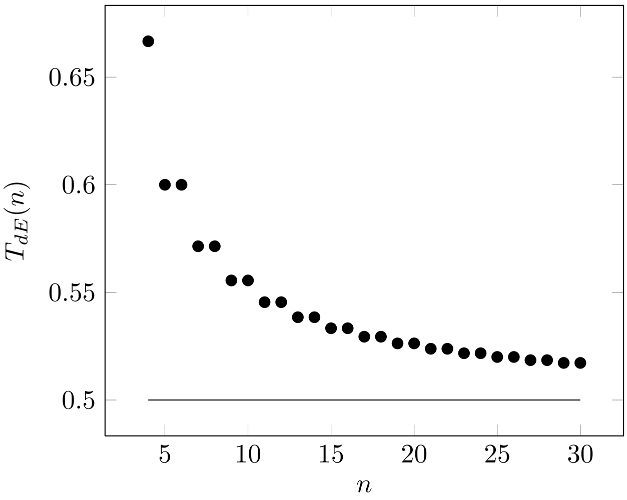
Third, consider the complete graph K_n, with $w(K_n) = n$. Deleting an edge vw will clearly reduce the clique number. Consider the subgraph $K \setminus \{v\}$; it has clique number $n-1$. Removing an edge $uu', u \neq w, u' \neq w$ will lower clique number again by 1. This process may be repeated $\lfloor n/2 \rfloor$ 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 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 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 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 [3].

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

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 below, we can expect to reduce the time consumption of the program by a factor of around 6.

Parallelization 3 The steps A are independent of eachother, 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 $2^{n_1}2^{n_2} = 2^n$.

3 Implementation details

The implementation only partially supports $n > 64$. In practice, the program does not terminate in human time for such large problems anyway, so this restriction is not critical. This allows us to use a natural way of encoding a set of vertices by simply letting it be a whole machine word, 64 bits long. A one in position i of the word means that vertex i (for some ordering) of G is present in the set represented by the word.

For polynomial representation we employ the use of two libraries for number theoretic calculations. These also provide interpolation functionality.

3.1 NTL 6.0.0

The Number Theoretic Library [10] provides a fast implementation of polynomial arithmetics. It 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.

The functions used are primarily these:

- `ZZX.operator+=()`

  Addition and assignment for polynomials.

- `ZZX.operator*=()`

  Multiplication and assignment for polynomials.

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

3.2 PARI 2.5.5

The PARI/GP project [13] is a computer algebra system, primarily designed as a full calculation system comparable to Maple. The back-end, PARI, is also available as a C library, providing polynomial arithmetics among other functionality.

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

3.3 GMP 5.1.2

Both libraries used for polynomial arithmetic allow for the user to configure them using GMP [6] as the low-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.

4 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.

4.1 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 follows directly from steps A(2)a and A(2)b above:

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

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

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

4.2 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 $O(2^n)$.

The GMP library supports Karatsuba, Toom-3, Toom-4, Toom-6.5, Toom-8.5 and Schönhage-Strassen [6, p 90] algorithms, 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 [11].

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.

5 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 [8].

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 size $\frac{1}{2}n(n-1)$ with m ones and rest zeroes, shuffle A using Fisher-Yates shuffle and then add the edge v_iv_{i+1} to the graph if $A[i] = 1$.

5.1 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++)

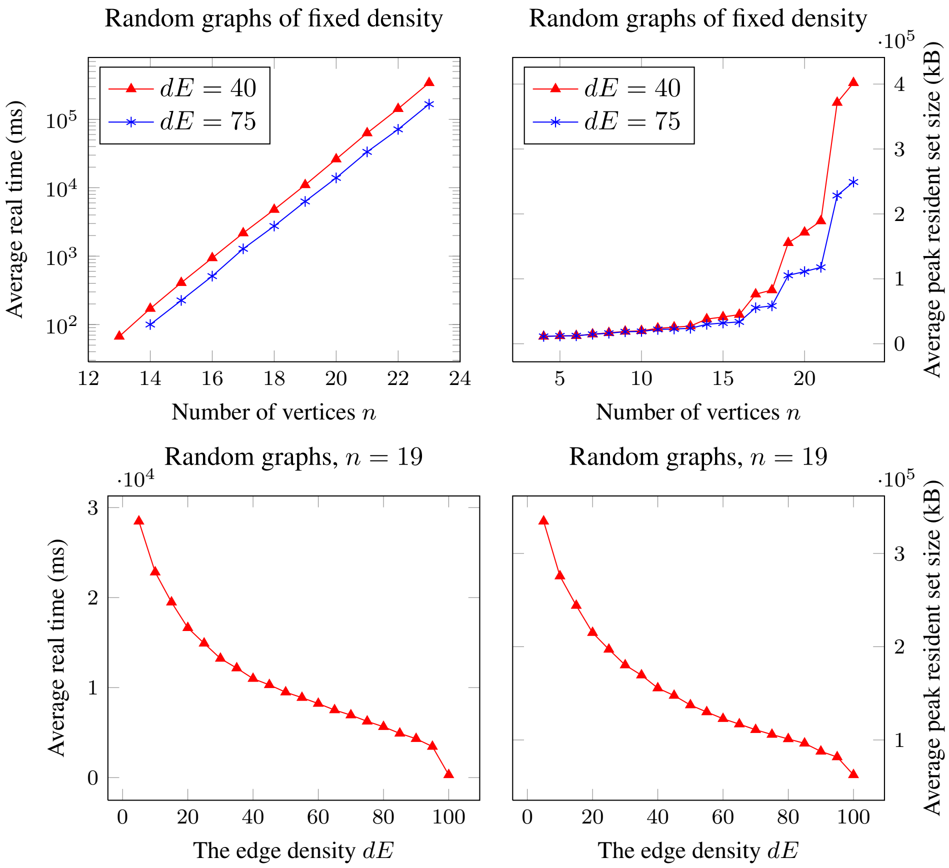
For all time and memory measurements, the GNU `time` 1.7 program is used (see the Linux man pages [5]). The user time, elapsed time (real time) [3] and peak resident set size are the data points recovered as measurements. These measurements are taken

[3] 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.

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.

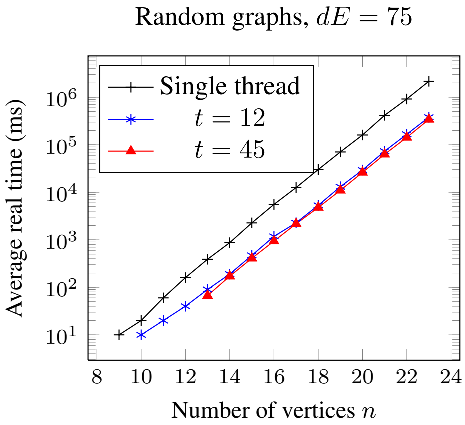
5.2 `bhkk-pari-0.3`

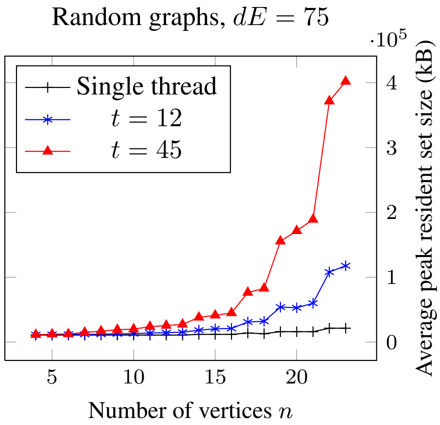
The most powerful implementation is `bhkk-pari-0.3`. It implements these optimizations: $q = \{0, 1\}$, $w_{min}(G)$, parallelization 2, degree pruning. We show here its time and memory consumption in relation to both n and dE.



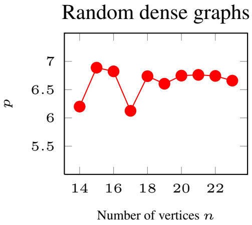
5.2.1 Power of parallelization

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.



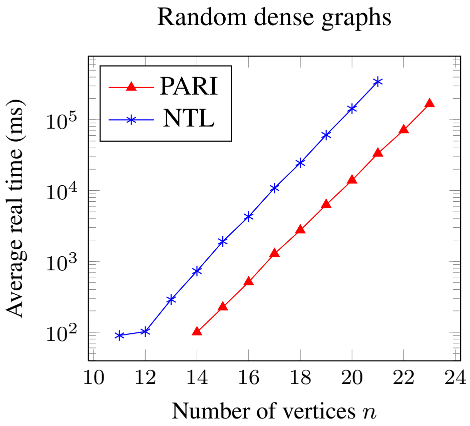


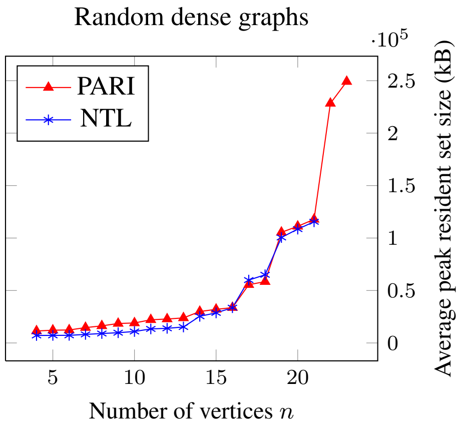
As mentioned above, parallelization (of width 12) allows us to terminate about six times faster. Here we plot the parallelization factor p as the quota of the non-parallelized `bhkk-pari-0.1` and the fastest implementation `bhkk-pari-0.3`.



5.2.2 Polynomial arithmetic library performance

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.



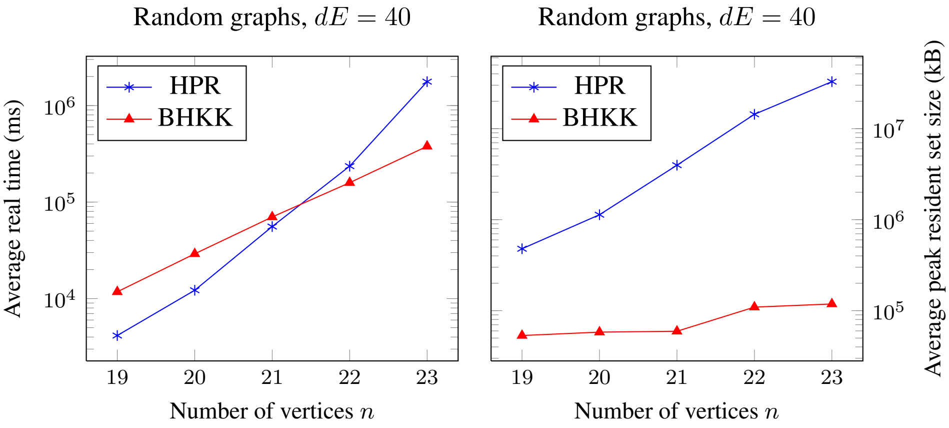


As is made clear, NTL is too slow to compete. This of course tempts the question whether there are even faster libraries to use.

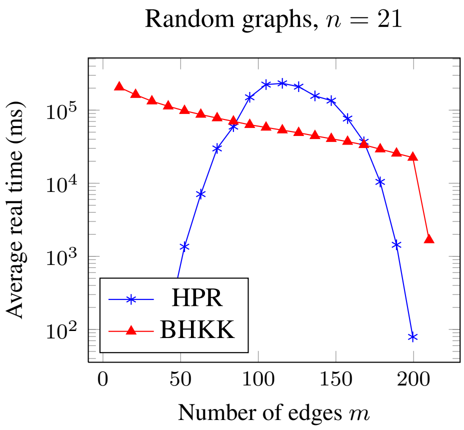
5.3 Relation to other work

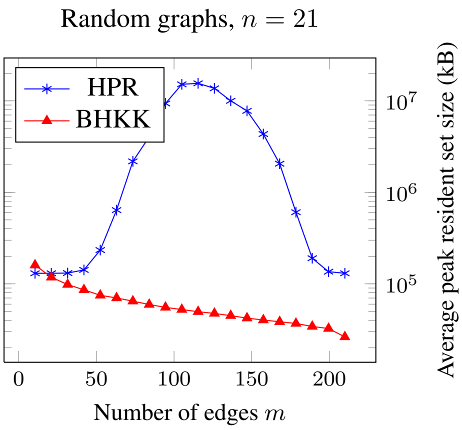
As comparison throughout the development process, we have consistently used HPR. Initially to make sure that the output were the same given same input (and hence, since HPR is a published tool by renowned authors, correct), and eventually, to "race". From performance tests that were made, HPR seems to 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 [7]. More pressingly, HPR does not give good worst-case performance. Our measurements (see table 1) 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've 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.



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





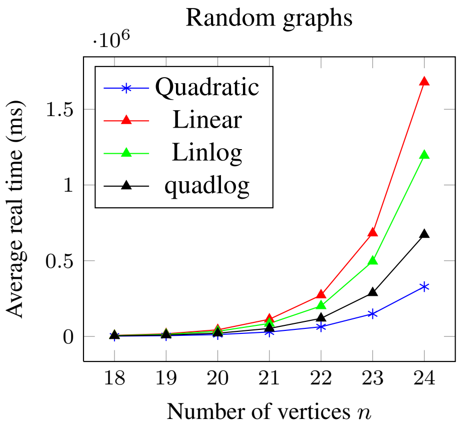
These results are somewhat in contrast to those presented in [7]. The data here suggests that the ”weak spot” in terms of density would be around $dE = 50$, whereas [7] suggests ”weak spot” around $dE = 75$. It’s made clear however, that BHKK shows better asymptotic behaviour, and that the space improvements 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.

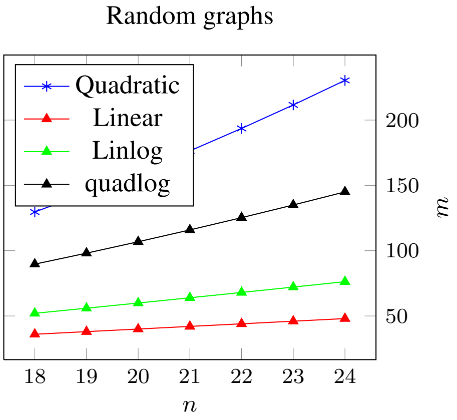
5.3.1 Different edge densities

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

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

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





Size	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 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.

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

Table 2: A first result on a "large" instance, $n = 25$, $dE = 40$. Results are based on a single run on one graph. HPR was allowed 5000MB of cache. `bhkk-pari-0.1` implements $q = \{0, 1\}$ and degree pruning.

	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 3: 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.

6 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.

6.1 Akbari's graph

Hillar and Windfeldt discussed characterizations of *uniquely* colourable graphs in [8]. That is, a graph G with $\chi_G(\chi(G)) = \chi(G)!$, or in other words that there exists only one optimal colouring, unique up to interchangability of the colours. Hillar and Windfeldt makes an attempt to verify their results by determining the chromatic polynomials of two graphs known to be uniquely 3-colourable, in order to test whether $\chi_G(3) = 3!$ for them. However, they were unable to determine the chromatic polynomial of the larger graph (on 24 vertices), Akbari's graph, seen here in figure 1, using Maple, as it uses a very naive algorithm to determine chromatic polynomials. Using BHKK, we successfully determined that for Akbari's graph, $\chi_G(t) =$

$$t^{24} - 45t^{23} + 990t^{22} - 14174t^{21} + 148267t^{20} - 1205738t^{19} + 7917774t^{18} - 43042984t^{17}$$

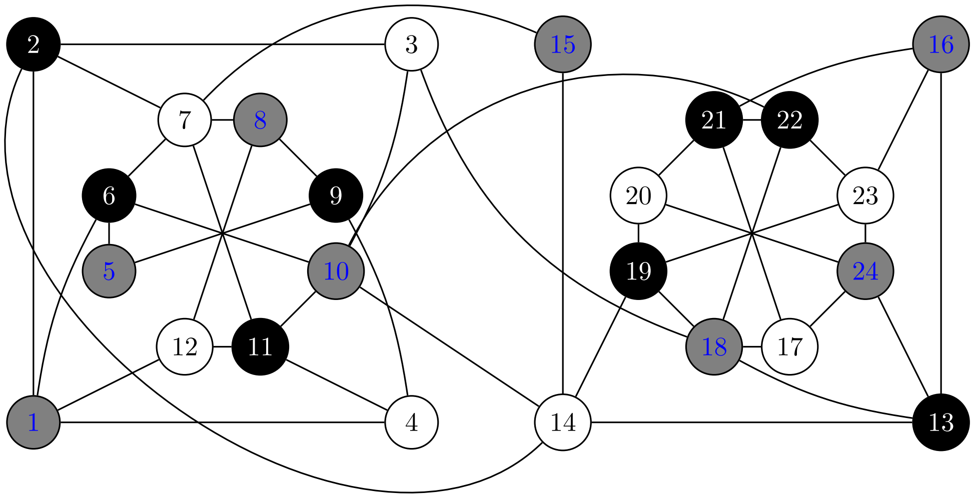


Figure 1: Akbari's graph, coloured in its unique 3-colouring with white, gray and black as colours. Figure copied from figure 2 in [8].

$$+197006250t^{16}-767939707t^{15}+2568812231t^{14}-7407069283t^{13}+18445193022t^{12}$$
$$-39646852659t^{11}+73339511467t^{10}-116102230203t^{9}+155931129928t^{8}$$
$$-175431211152t^{7}+162362866382t^{6}-120414350156t^{5}+68794778568t^{4}$$
$$-28408042814t^{3}+7537920709t^{2}-963326674t$$

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

6.2 Queen graph

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

$$\chi_{Q_5}(t) = t^{25} - 160t^{24} + 12400t^{23} - 619000t^{22} + 22326412t^{21} - 618664244t^{20}$$
$$+13671395276t^{19} - 246865059671t^{18} + 3702615662191t^{17} - 46639724773840t^{16}$$

$$
\begin{aligned}
&+496954920474842t^{15} - 4497756322484864t^{14} + 34633593670260330t^{13}\\
&-226742890673713726t^{12} + 1258486280066672806t^{11} - 5890734492089539317t^{10}\\
&+23071456910844580538t^{9} - 74774310771536397886t^{8} + 197510077615138465516t^{7}\\
&-416375608854898733286t^{6} + 680208675481930270860t^{5} - 824635131668099993614t^{4}\\
&+692768396747228503860t^{3} - 356298290543726707632t^{2} + 83353136564448062208t^{1}
\end{aligned}
$$

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

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

Table 4: Time and memory measurements on computing chromatic polynomials of queen graphs.

References

[1] Akbari, S, Mirrokni, V S, Sadjad, B S.
"Kr-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] GNU Project.
The GNU Multiple Precision library, version `5.1.2`, (2013).
URL `http://gmplib.org/`

[7] 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`

[8] 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`

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

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

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

[12] 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`

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

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