# Performance measurements of a small-space Chromatic Polynomial algorithm

Mats Rydberg

November 22, 2013

**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 et al which 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 et al from 2010. We also present the chromatic polynomial for a certain graph specified by Hillar & Windfeldt.

# 1 Introduction

A *proper $q-$colouring* of a graph $G = (V, E)$ is a mapping $\sigma : V \to [q]$ where $[q] = \{1, 2, \ldots, q\}$ with $\sigma(v) \neq \sigma(w)$ if $vw \in E$. In other words, it 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 $\chi_G(c) > 0$. $\chi_G(t)$ is of high interest in the field of algebraic graph theory, as one of the main artifacts characterizing a graph. 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. [3] [6]

The chromatic polynomial was specified in 1912 by Birkhoff [1], who defined it for planar graphs with the intent on proving the Four Colour Theorem. Whitney extended its definition to general graphs in 1932 [7], and Tutte incorporated it into what is now known as the Tutte polynomial. Haggard, Pearce and Royle [4] 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 $\chi_G(t)$. Björklund, Husfeldt, Kaski and Koivisto 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.

In this report, we present an implementation of the BHKK algorithm and experimental results from running it on selected classes of graphs. In particular, we show instances where BHKK outperforms HPR, and discuss which classes of graphs this generalizes to.

# 2 The algorithm

The algorithm measured in this performance test is described in Björklund et al [2], 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 (ie, 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) $V_2$. [2, sec 5]

The full algorithm in pseudo-code as follows:

Step A. For $q = 0, 1, \ldots, n$, do

1. Partition $V$ into $V_1$ and $V_2$ of sizes $n_1$ and $n_2$.
2. For each $X_1 \subseteq V_1$, do
   a) For each independent $Y_1 \subseteq X_1$, do
   $$h[V_2 \setminus N(Y_1)] \leftarrow h[V_2 \setminus N(Y_1)] + z^{|Y_1|}$$
   b) For each independent $Y_2 \subseteq V_2$, do
   $$l[Y_2] \leftarrow z^{|Y_2|}$$
   c) $h \leftarrow (h\zeta') \cdot l$
   d) $h \leftarrow h\zeta$
   e) For each $X_2 \subseteq V_2$, do
   $$r \leftarrow r + (-1)^{n-|X_1|-|X_2|} \cdot h[X_2]^q$$
3. Return coefficient $c_n$ of $z^n$ in $r$.

Step B. Construct interpolating polynomial $\chi_G(t)$ on points $(q, c_{nq})$.

Step C. Return $\chi_G(t)$.

$N(Y)$ is the set of all vertices in $G$ adjacent to at least one vertex in $Y$, $x\zeta'$ denotes the fast up-zeta transform and $x\zeta$ the fast down-zeta transform (of $x$) (see [2, sec 2]. $h$ and $l$ are arrays of size $2^{n_2}$ of polynomials (initialized to zeroes), $r$ is a polynomial. For a more detailed description, see [2, p 9].

## 2.1 Optimizations

Here are presented some improvements to the algorithm that are either natural, mentioned in [2] 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, but in practice it is even faster, as we will encounter an edge with high probability before we've scanned the whole graph.

[1] Multiple edges and self-edges are two graph invariants 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.

- 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.

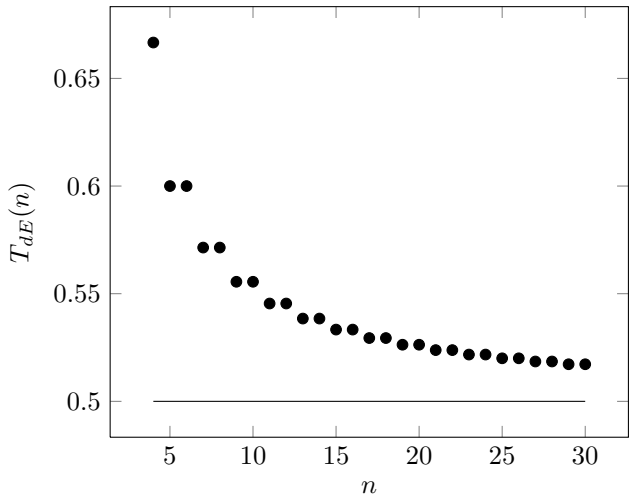
Here we define the *density* of a graph $G$ as $dE = m/m_{max} = \frac{2m}{n(n-1)}$. This immediately tells us the *smallest possible* $\omega(G)$. Let us call it $\omega_{min}(G)$. In fact, the following holds:

$$\omega_{min}(G) = \begin{cases} n-a & \text{if } m = m_{max} - a & 0 \leq a < \lfloor \frac{n}{2} \rfloor \\ \lceil \frac{n}{2} \rceil - a & \text{if } \lfloor (\frac{n}{2})^2 \rfloor < m \leq m_{max} - (a+1)\lfloor \frac{n}{2} \rfloor & 0 \leq a \leq \lfloor \frac{n}{2} \rfloor - 2 \\ 2 & \text{if } 0 < m \leq \lfloor (\frac{n}{2})^2 \rfloor & \\ 1 & \text{if } m = 0 & \end{cases} \tag{1}$$

As we can see from the equation, only graphs with $m > \lfloor (\frac{n}{2})^2 \rfloor$ 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

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

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.




This result is quite interesting, because 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. [2]

**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 $d(r) \leq n$. 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 integer 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.

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

## 3.1 NTL 6.0.0

The first is NTL, Number Theoretic Library, written in C++ by Victor Shoup at New York University[10]. It is advertised as one of the fastest implementations of polynomial arithmetics, which is all that we are interested in. Unfortunately, it does not provide any non-trivial way of exponentiating polynomials, and its multiplication algorithms are a bit lackluster after some careful studying. It is very easy to use, provides its own garbage collection and has a rich, high-level interface for library usage.

The functions used are primarily these:

- `ZZX.operator+=()`

  Addition and assignment for polynomials.

- `ZZX.operator*=()`

  Multiplication and assignment for polynomials.

Binaries are called `bhkk-ntl-x.y.z`.

## 3.2 PARI 2.5.5

Experiencing the relative lack of performance boost from the NTL implementation led to finding also the PARI project. It is written in C mainly by a group of French computer scientists at Université Bordeaux [12], together with a calculator-like interface (`gp`) to be used by an end-user (comparable to Maple). The PARI library is provided at a much lower level, requires the user to garbage collect (since it is written in C, after all), has a much steeper learning curve and a very detailed but hard-to-grasp documentation. PARI provides special methods 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.

Binaries are called `bhkk-pari-x.y.z`.

## 3.3 GMP 5.1.2

Both of the libraries used to represent polynomials allow (and encourage) the user to configure them using GMP [8] as the low-level interface for integral arithmetic (actually, for all arithmetic, but we only use integers). Authors of both libraries suggest that using GMP instead of their own, native low-level interface will yield significant performance boosts. For this reason, we have done just that. GMP is well documented, easy-to-use, provides both C and C++ interfaces and even has a well-maintained bug reporting facility (I got an answer the same day!). GMP allows the user a rich variety

of configuration options, and optimization has been done as narrowly as possible to get maximum performance on one machine.

In the implementation, GMP is only directly employed by the graph generating tool. All arithmetic is performed via the interfaces of PARI and NTL, which themselves call underlying GMP functions.

# 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 of the algorithms which was studied 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 zeroes. 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 [8, p 90], which means all libraries used in the programs uses these algorithms *at least* when multiplying integers (ie, coefficients of polynomials).

NTL implements Karatsuba, Schönhage-Strassen and another FFT-based technique for polynomials [11].

I have not found any documentation specifying which algorithms are implemented in the PARI library for polynomial multiplication. From analyzing the source code, it seems as if PARI "converts" the polynomial to an integer and submits it to its integer multiplication function (which would be one of GMPs).

# 5 Experimental results

This section presents selected results in three 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 each development choice. 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. [4] We end by providing the chromatic polynomial for a famous graph.

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 $m_{max} = \frac{n(n-1)}{2}$ 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 $A[i] = 1$.

## 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++) |

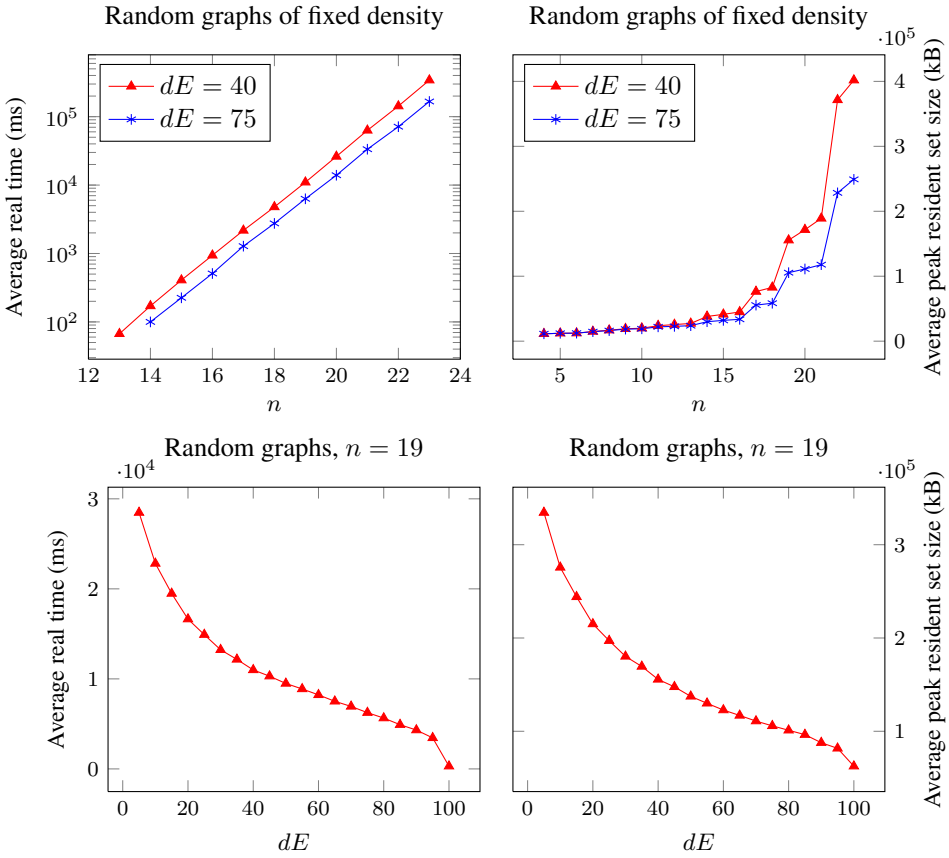
For all time and memory measurements, the GNU `time` 1.7 program is used (see the Linux man pages [9]). The user time, elapsed time (real time) [3] and peak resident set size are the data points recovered as measurements. These measurements are taken by running the specified program on a number [4] of graphs of equal size and the average values are the ones presented.

## 5.2 `bhkk-pari-0.3`

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

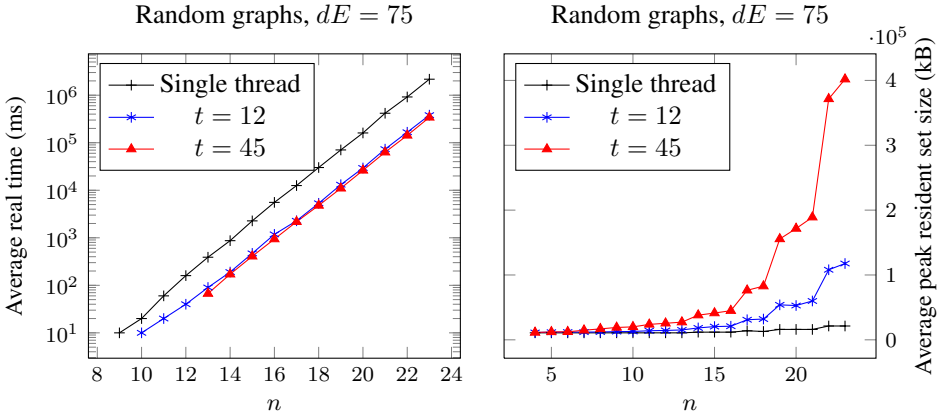
[3] Do note that for single-threaded applications, user time and real time are (more or less) the same. No graphs are therefore provided for the real time measurements. The "less" refers to the fact that there are some units of time scheduled as system time, but these values are too small to be significant in these experiments.

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

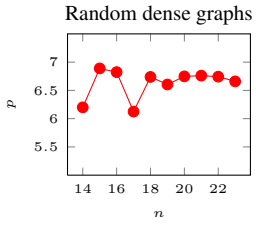



### 5.2.1 Power of parallelization

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.

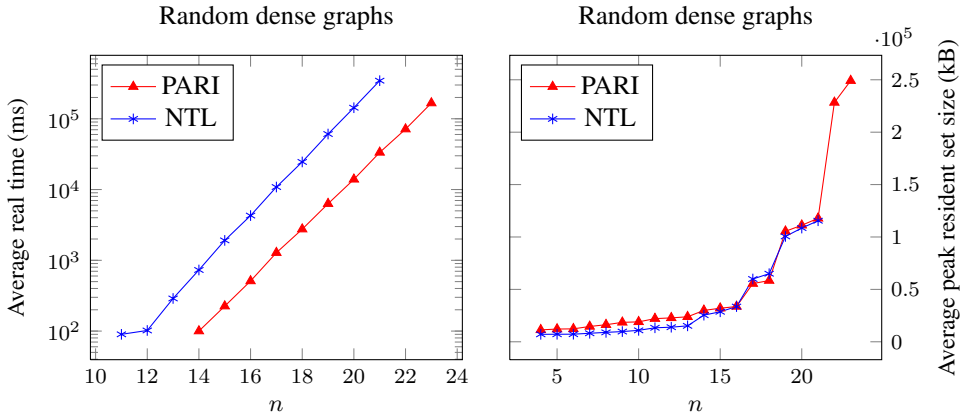


As mentioned above, parallelization (of width 12) allows us to terminate about six times faster. Here we plot the parallelization factor $p$ as the quota of the slowest and fastest plots above.




### 5.2.2 NTL

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.



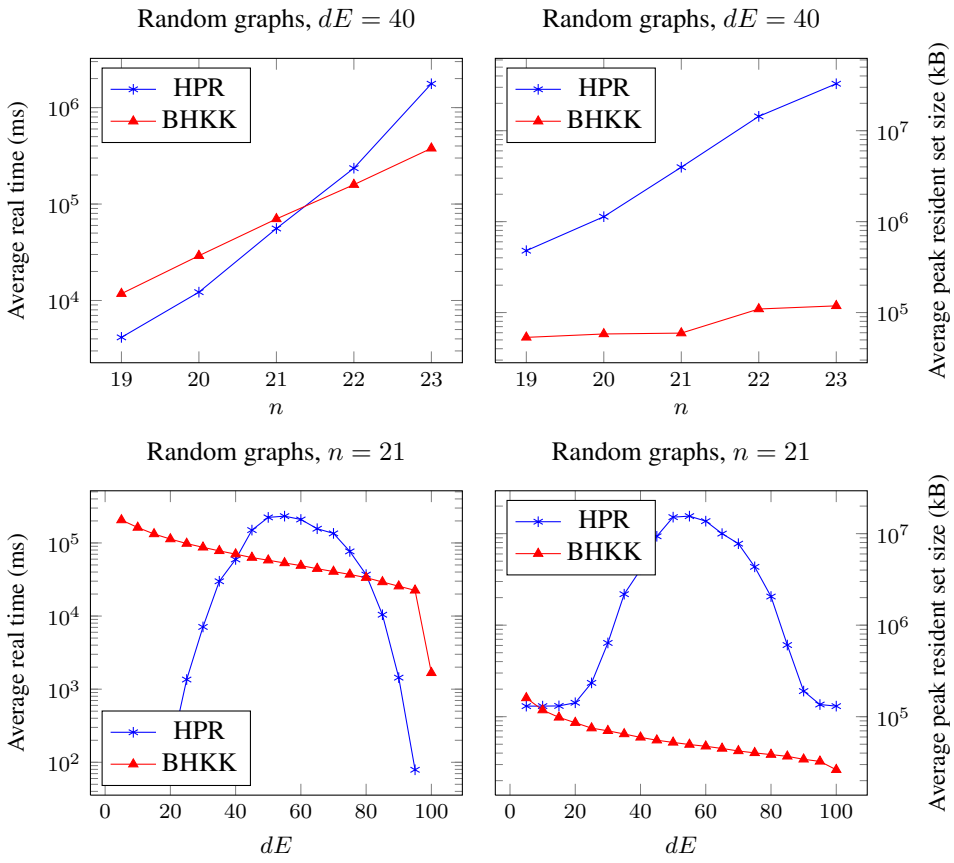

As is made clear, NTL is too slow to compete. This of course tempts the question whether there are even faster libraries to use.

## 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 that 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 [4]. More pressingly, HPR does not give good worst-case performance. Our measurements (see table 1) show a large ($> 95\%$) variance[5]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.

[5]With variance, we mean $v = \frac{M-m}{M}$ where $M$ is the maximum measured value and $m$ the minimum.

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.




These results are somewhat in contrast to those presented in [4]. The data here suggests that the "weak spot" in terms of density would be around $dE = 50$, whereas [4] 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.

| 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 even 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.

| | 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.

Hillar & Windfeldt discussed characterizations of *uniquely* colourable graphs in [5]. That is, a graph $G$ with $\chi_G(q) = q!$ (for some $q$). In their paper, they try to verify their results by testing this fact, but were unable to determine $\chi_G(t)$ using Maple. Using BHKK, we determine that for the uniquely 3-colourable graph in figure 2 in [5], $\chi_G(t) =$

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

$$+ 197006250t^{16} - 767939707t^{15} + 2568812231t^{14} - 7407069283t^{13} + 18445193022t^{12}$$

$$- 39646852659t^{11} + 73339511467t^{10} - 116102230203t^9 + 155931129928t^8$$

$$- 175431211152t^7 + 162362866382t^6 - 120414350156t^5 + 68794778568t^4$$

$$- 28408042814t^3 + 7537920709t^2 - 963326674t$$

and in particular, $\chi_G(3) = 3!$, as expected. This took $1445$ seconds to compute. HPR terminates even faster.

# References


[1] G.D. Birkhoff, "A determinant formula for the number of ways of coloring a map", *Ann. Math.* **14**, (1912), 42-46.

[2] A. Björklund, T. Husfeldt, P. Kaski, M. Koivisto, "Covering and packing in linear space", *Inf. Process. Lett.* **111** (2011), 1033-1036.

[3] G.J. Chaitin "Register allocation & spilling via graph colouring", *Proc. 1982 SIGPLAN Symposium on Compiler Construction*, (1982), 98–105.

[4] G. Haggard, D. Pearce, G. Royle, "Computing Tutte Polynomials", *ACM Transactions on Mathematical Software* **37** (2010), art 24.

[5] C.J. Hillar, T. Windfeldt, "Algebraic characterization of uniquely vertex colorable graphs", *J. Combin. Theory*, **98** (2008), 400-414

[6] D. Marx, "Graph colouring problems and their applications in scheduling", *Periodica Polytechnica, Electrical Engineering* **48** (1–2) (2004), 11–16.

[7] H. Whitney, "The coloring of graphs", *Ann. Math.* **33**, (1932), 688-718 .

[8] The GNU Multiple Precision library, version `5.1.2`
`http://gmplib.org/`

[9] GNU time, version `1.7`
`http://man7.org/linux/man-pages/man1/time.1.html`

[10] V. Shoup, Number Theoretic Library, version `6.0.0`
`http://www.shoup.net/ntl/index.html`

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

[12] PARI/GP, version `2.5.5`, Bordeaux, 2013
`http://pari.math.u-bordeaux.fr/`.

[13] Wikipedia, the Free Encyclopedia, Toom-Cook multiplication
`https://en.wikipedia.org/wiki/Toom%E2%80%93Cook_multiplication`

[14] Wikipedia, the Free Encyclopedia, Shönhage-Strassen algorithm
`https://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen_algorithm`