Performance measurements of a small-space Chromatic Polynomial algorithm

Mats Rydberg

November 12, 2013

Abstract

The *Chromatic Polynomial* χ_G of a graph G on n vertices is a univariate polynomial of degree n, passing through the points $(k, P_G(k))$ where $P_G(k)$ is the number of k-colourings of G. In this paper, we present an implementation of the BHKK algorithm which computes χ_G 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 another implementation from 2010.

1 The algorithm

The algorithm measured in this performance test is described in Björklund et al [1], and henceforth referred to as the **BHKK** algorithm. It is based on the linear-space Fast Zeta Transform described in the same paper, and is proven to perform in time $O(2^n)$ and space $O(1.2916^n)$.

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 using interpolation.

The 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 [1, 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 [1, p 9].

1.1 Optimizations

Here are presented some improvements to the algorithm that are either natural, mentioned in [1] or invented by the author. This list is by no means exhaustive, nor is every item critical, but the ones we've explored were proved to be efficient.

[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 these are naturally not allowed. In my implementation, I 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.

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

- For $q = 0$, any graph with $n > 0$ can be coloured in 0 ways. For $n = 0$, 0-colouring is undefined. This is a purely semantic question anyway, and empty graphs are not a relevant topic for this paper. This takes $O(1)$ time.
- 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.
- 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). This has not been implemented in my programs.

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

Exploiting $w(G)$ A more sophisticated type of optimization involves exploiting the clique number $\omega(G)$, which is a lower bound of 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 χ_G is known.

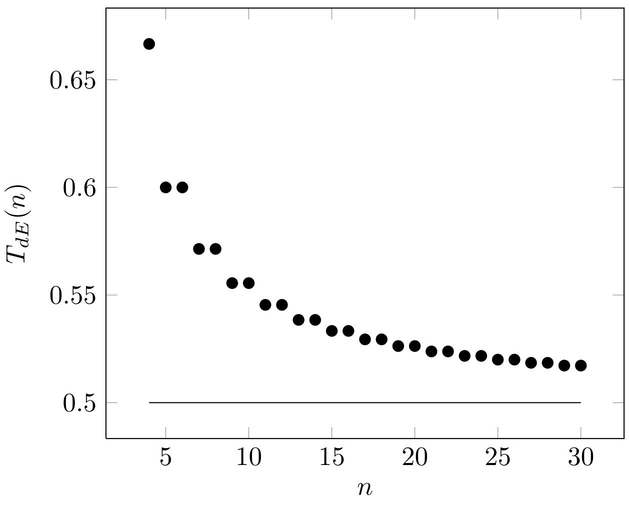
Here we define the *density* of a graph G as $dE = m/m_{max} = \frac{2m}{n(n-1)}$. This immediately tells us the *smallest possible* $\omega(G)$. Let us call it $\omega_{min}(G)$. In fact, the following holds:

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

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 the graphs discussed in 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 would yield an expected decrease in time by about 15%[2].

Parallelization Most importantly, and also mentioned in [1], is to parallelize computations of step A2, as these are independent of eachother. This would yield significant time improvements in theory. With access to $2^{|V_1|}$ processors, we would be able to execute the program in time $O(1.5486^n)$. Typically, we will only have access to a constant number of processors in practice, allowing each of them to execute a range of iterations of step A2. As presented below, we can expect to reduce the time consumption of the program by a factor of around 6.

We may also parallelize the steps A on $O(n)$ parallel CPUs. This would not reduce the exponential factor of the time complexity, but it does reduce the polynomial factor and it is likely to give significant results in practice. I do not investigate this optimization technique, as my resources are already exhausted by parallelizing step A2.

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 $2^{n_1}2^{n_2} = 2^n$. This has not been further investigated.

2 Implementation details

My test implementation only partially supports $n > 64$. In practice, the program does not perform very well for such large problems anyway, so this restriction is not critical. It also allows me to use a quite natural way of encoding sets by simply letting them be

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

a whole integer word, 64 bits long. A one in position i of the word means that vertex i in the graph is present in the set represented by the word.

For polynomial representation I've emplyed the use of two libraries for number theoretic calculations. These also provide interpolation functionality.

2.1 NTL 6.0.0

The first is NTL, Number Theoretic Library, written in C++ by Victor Shoup at New York University[4]. It is advertised as one of the fastest implementations of polynomial arithmetics, which is all that I am 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 I use 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`.

2.2 PARI 2.5.5

Experiencing the relative lack of performance boost from the NTL implementation led me to find also the PARI project. It is written in C mainly by a group of French computer scientists at Université Bordeaux [6], 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 I use 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`.

2.3 GMP 5.1.2

Both the libraries I use to represent polynomials allow (and encourage) the user to configure them using GMP[2] as the low-level interface for integral arithmetic (actually, for all arithmetic, but I 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, I 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 I've tried to optimize as narrowly as possible to get maximum performance on one machine.

In my implementation, I only use GMP to generate randomized graphs. All arithmetic is performed via the interfaces of PARI and NTL, which themselves call underlying GMP functions.

3 Algorithm performance parameters

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

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 of the algorithms which I've studied for graph colouring problems. And this is not only for very dense graphs, but performance 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. This has a direct effect in reducing some additions and assignments, but more importantly has side effects in all subsequent steps.

3.2 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 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 [2, p 90], which means all libraries used in my 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 [5].

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).

4 Experimental results

My tests are performed on randomized graphs, generated for some values of n and dE. The algorithm does not assume anything about the general structure of the graphs, and only exploits the graph invariants mentioned in 1.1.

The following instances have been executed:

- Increasing n in interval $[4, 23]$.

  "Dense" graphs, $dE = 75$.

  "Sparse" graphs, $dE = 40$.
- Increasing density, $n = 19$, $dE \in [5, 100]$.
- "Large" instances, $n = 25$ and $n = 30$, $dE = 75$.

  $n = 30$ is only performed on the fastest versions.

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
Compilator	GCC 4.7.2 (g++)

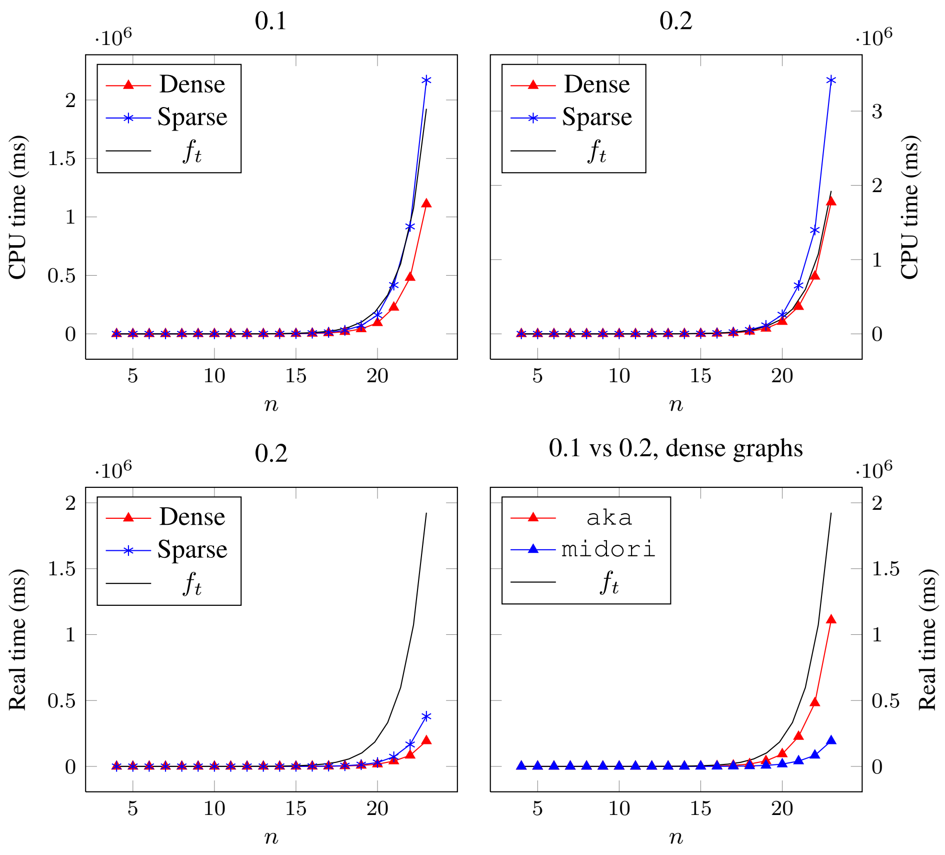
For all time and memory measurements, the GNU time 1.7 program is used [**?**]. The user time, elapsed time (real time)[3]and peak resident set size are the data points recovered as measurements.

4.2 PARI 2.5.5

4.2.1 Increasing n

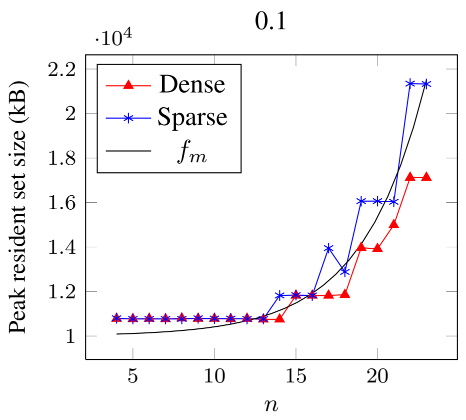
This is time as a function of n. Expected function in theory is $O^*(2^n)$. For comparison, I also plot the function $f_t(n) = \frac{n}{100}2^n$, which is a model of the time usage of the algorithm, where each measured operation takes a hundredth of a millisecond, providing the desired asymptotical behaviour.

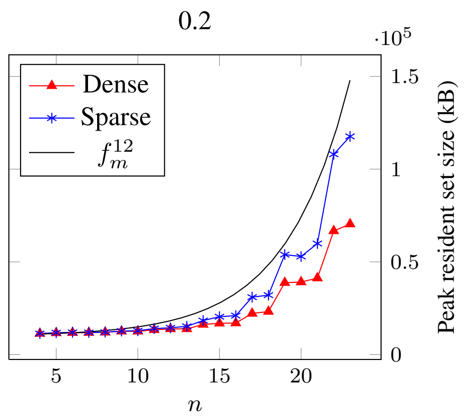
[3]Do note that for single-threaded applications, user time and real time are (more or less) the same. No graphs are therefore provided for the real time measurements. The "less" refers to the fact that there are some units of time scheduled as system time, but these values are too small to be significant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

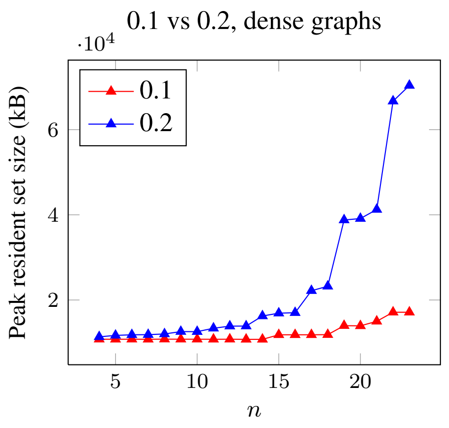


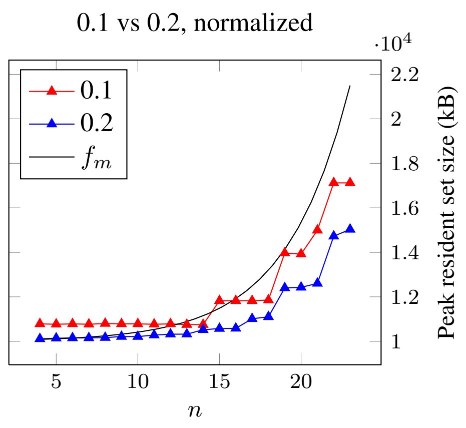
The results are not revealing much that was not expected. We can see that 0.1 actually uses more CPU time than 0.1 (top graphs), some of which could be contributed to the fact that parallelization requires some overhead.

Next we have memory usage as a function of n. Expected function from theory is $O^*(1.2916^n)$. Our modeling function is $f_m(n) = 32 \cdot 1.2916^n + 10000$, which assumes an overhead of 10,000kB and each measured element using up 32kB on average. Since a parallelized program uses also memory in parallel, we also employ the modeling function $f_m^{12}(n) = 12 \cdot 32 \cdot 1.2916^n + 10000$, assuming 12 threads in parallel using maximum memory.





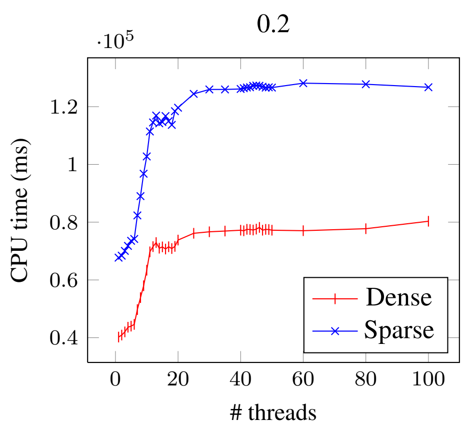


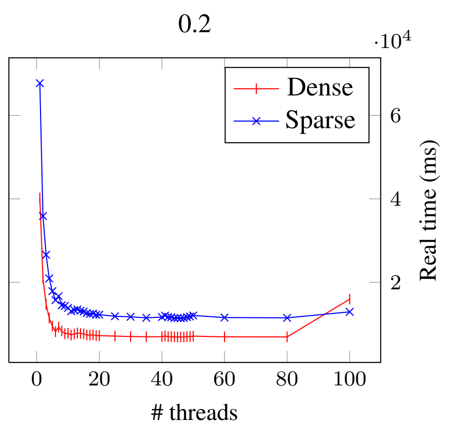


The "normalization" used above follows the model in an attempt to isolate the peak resident set usage by the most memory-expensive thread of 0.2. The measured value mem is divided by $t = 12$ and then we add the number $\frac{t-1}{t} \cdot 10000$ to compensate for our modelled overhead.

4.2.2 Parallelization

Here we investigate how our parallelization scheme acts as we increase the number of threads. We plot the function of user and real time as we increase the number of threads. Recall that our maximum *actual* parallelization width is 12.

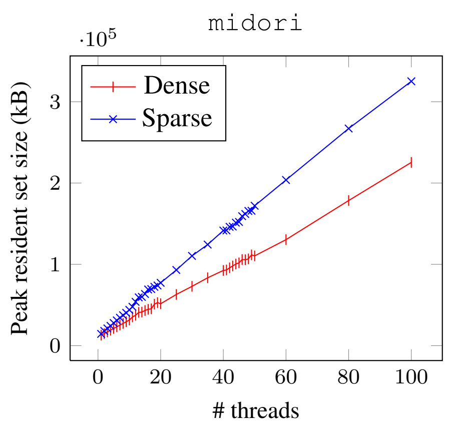


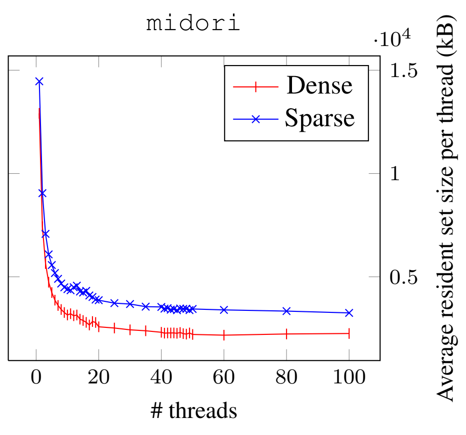


There seem to be no notable difference in how sparse and dense graphs parallelize. As expected, the plots peak at 12 threads in the first figure. What is not expected is that the top performance was measured for around 45 threads (this is also the reason why I have more data points between 40 and 50). It is not hard to realize that some threads will terminate much sooner than others (because they happened to get a range of subsets which were more frequently independent, see section 3.1), and so using $>$ 12 threads could be useful. It is unclear why our best results come around 45, however.

Another interesting note to take is that for 100 threads, processing the dense graph take more time than the sparse graph. This can partly be explained by the fact that the OS is now spending lots of time context switching the 100 threads over the 12 CPUs, but from a more algorithmic point of view, we can make another argument.

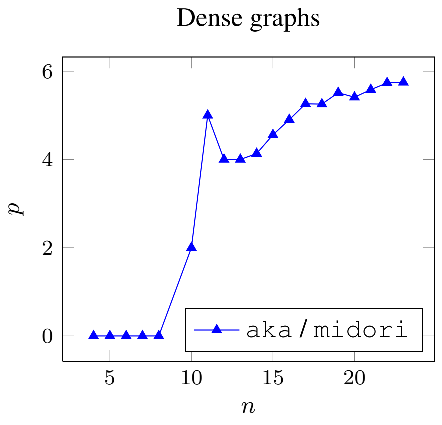
As we let each thread iterate over fewer and fewer subsets from step A2, our real time consumption is more and more dominated by the one thread that takes the longest time to execute. That is, the thread that was allocated with the ”worst” range of subsets (the one with the least dependencies). ”Bad” ranges are more common when the graph is sparse, but as we divide the subsets into smaller ranges, the ”worst” range decreases in ”badness” faster for a sparse graph than for a dense. In fact, the probability that a selected range of subsets from a dense graph includes more independent sets to consider than any selected range from a sparse graph is higher for moderate range sizes (it stays < 0.5, of course), even though it approaches zero as we approach range size one.





The memory usage increases quite linearly as we add threads. This is due to the fact that `midori` stores all the t intermediate return values simultaneously until all threads are ready. The average memory per thread seems to converge to some constant.

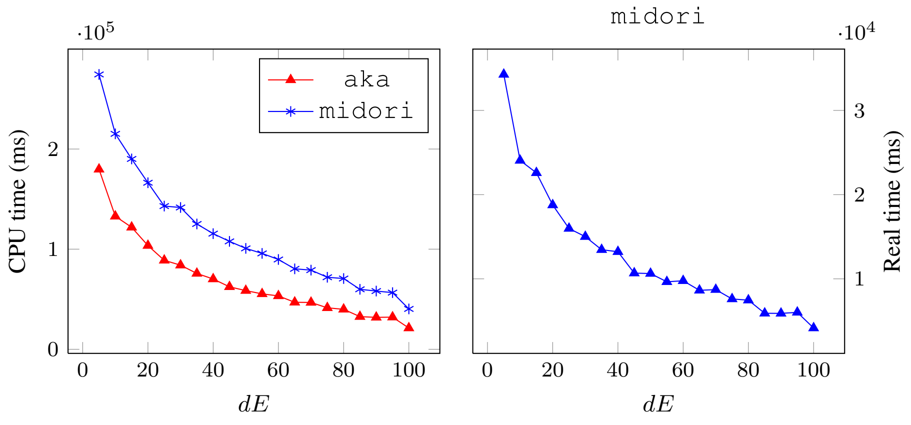
When we use t threads, in theory we should be able to compute the same problem t times faster than if we use 1 thread. This is not the case in practice, as we have seen so far. But how much do we actually gain? Let's define a parallelization factor p as the quotient between the time used by `aka` and the time used by `midori`.



By the measured data, p stays less than half of its theoretical maximum. This is contributed either to inability at the programmers side, or to the fact that not all threads share the time burden equally. The second indicator (while also being more lenient towards the author), gets further support from the above discussion mentioning about 45 threads to be ideal for the given sizes of n and $t_{max} = 12$.

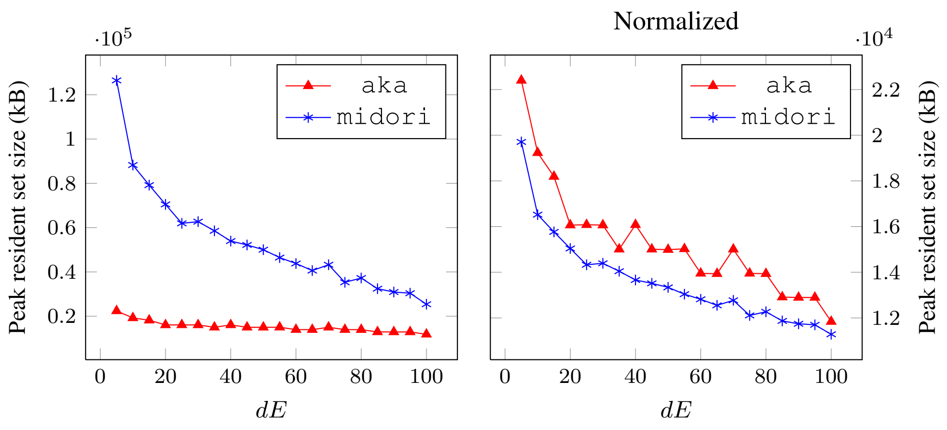
4.2.3 Increasing density

So, we've seen that the density of the graph is important to the performance of the algorithm. But how much falloff can we expect? And are there any certain points of extra interest?



The time fall-off follows the same pattern in user time and in real time. The pattern is also the same if we compare to the single thread version. It is more or less linear.

Memory as a function of density:



As we can see, both time and space consumption falls off the most as we add the first edges. Around 20% density we use about 30% less space and over 40% less time as compared to 5% density. Our best performance is on complete graphs, even though we do not implement any ”special treatment” for graphs of certain density.

4.2.4 ”Large” instance

Single thread: (gets beat) on m_25_40

Program	Time (s)	Peak resident set size (MB)
`chr_pol_pari`	10870	33.93
`tutte`	4906	20129.5

12 threads: (pwns) on m_30_75

Program	CU time (s)	Real time (s)	Peak resident set size (MB)
`chr_pol_pari`	559,841	49,651	2,584
`tutte`	> 94,000	> 95,000	40,135

`tutte` did not finish and was stopped after 95,000 seconds (over 26 hours).

4.2.5 Evaluation step

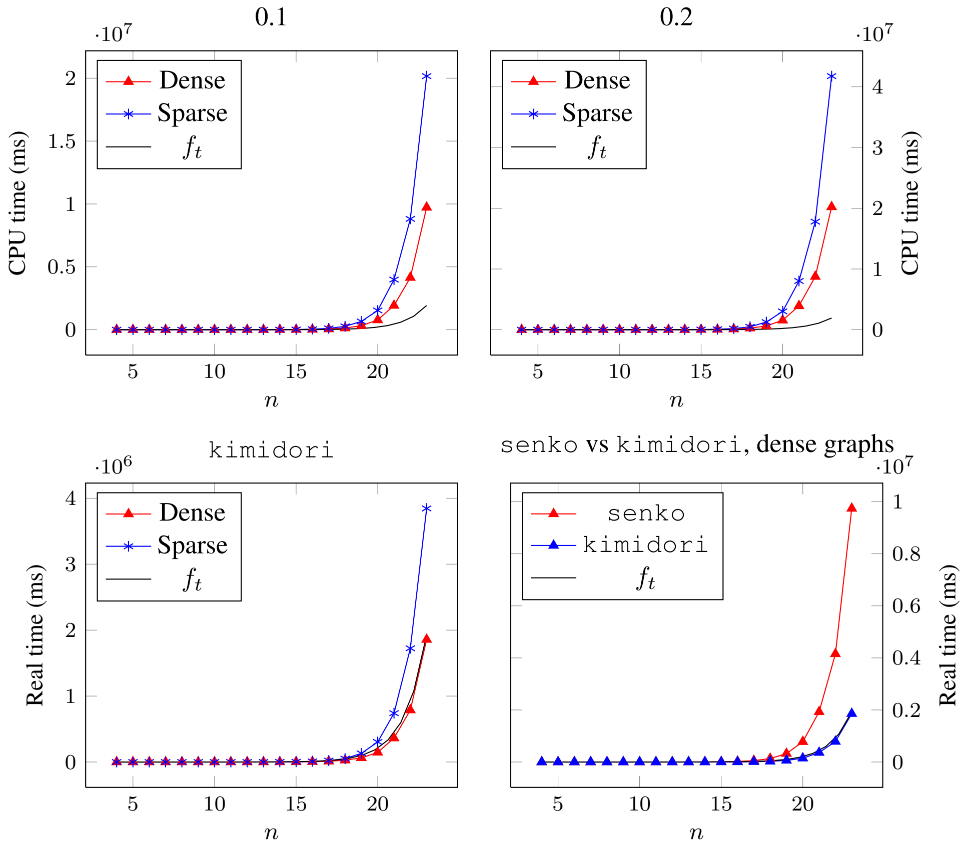
4.3 NTL

The NTL implementation was the first I incorporated, but its performance did not live up to my expectations. For reference, and to clearly show the importance of the actual implementation of polynomial arithmetics, the same simulations have been executed using this library also.

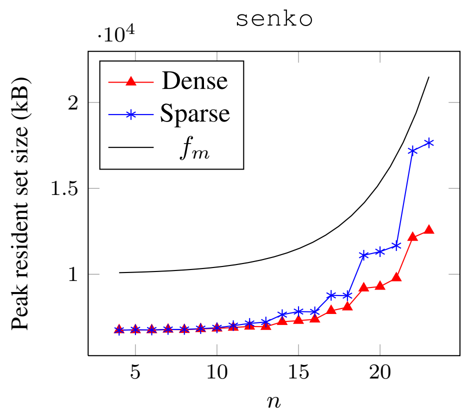
Most of the results in this section are very similar to those in the PARI section. Not much explaining text is thus provided.

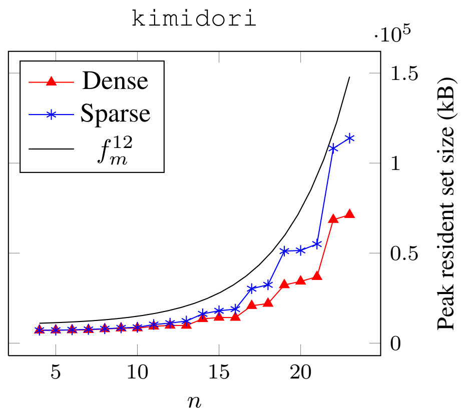
4.3.1 Increasing n

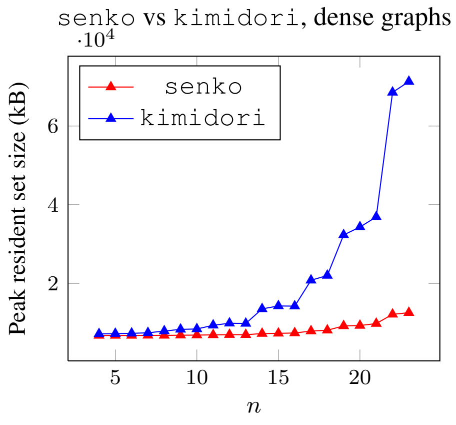
The same model function $f_t(n) = \frac{n}{100}2^n$ is used to reference the results to what is expected. This model does not align as well with the NTL implementations as with the PARI ones.

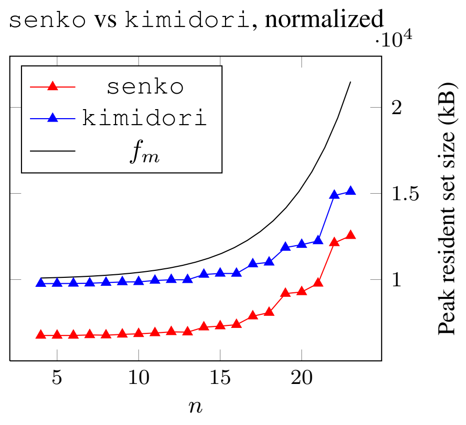


We use the same model for memory usage as above, $f_m(n) = 32 \cdot 1.2916^n + 10000$.



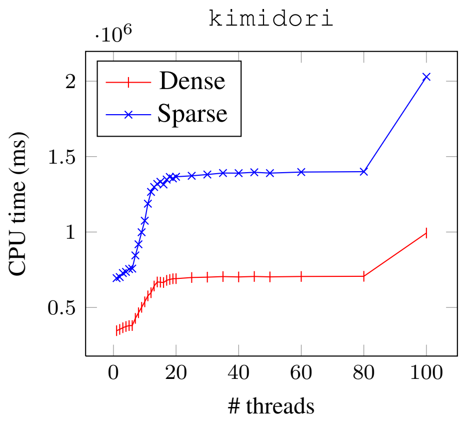


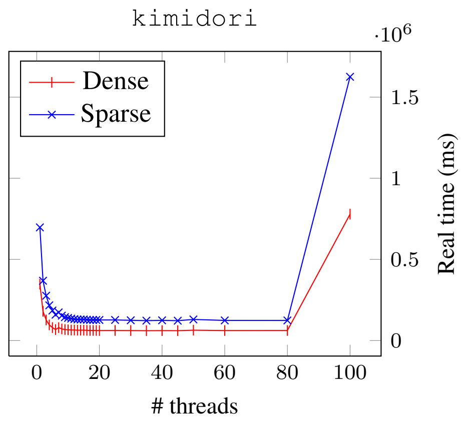


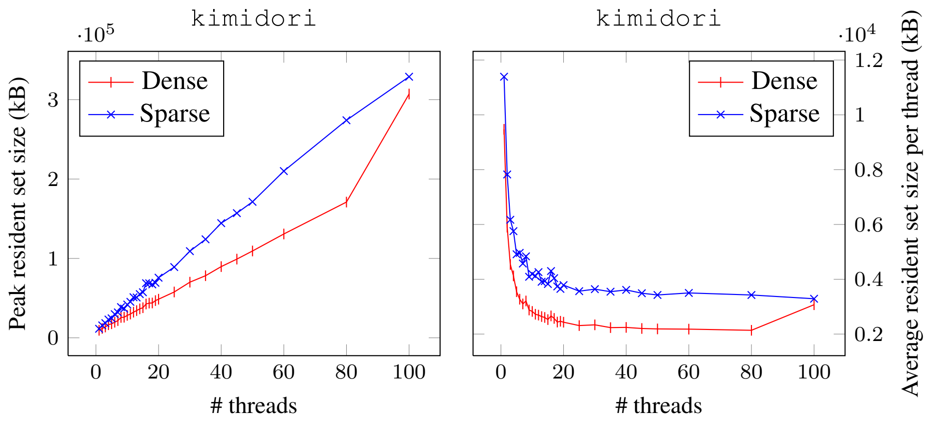


4.3.2 Parallelization

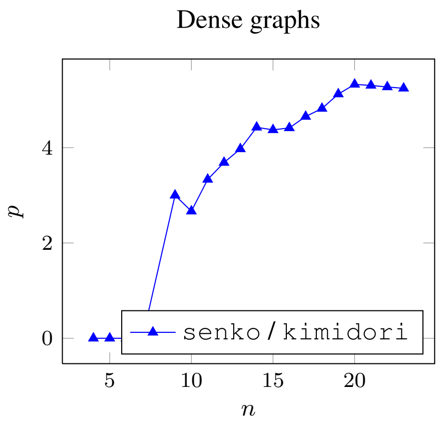
Time and memory usage:







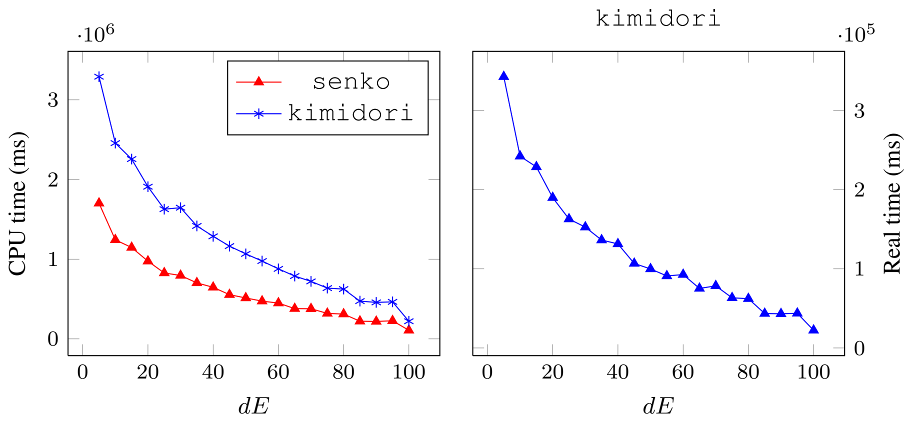
The parallelization factor p:



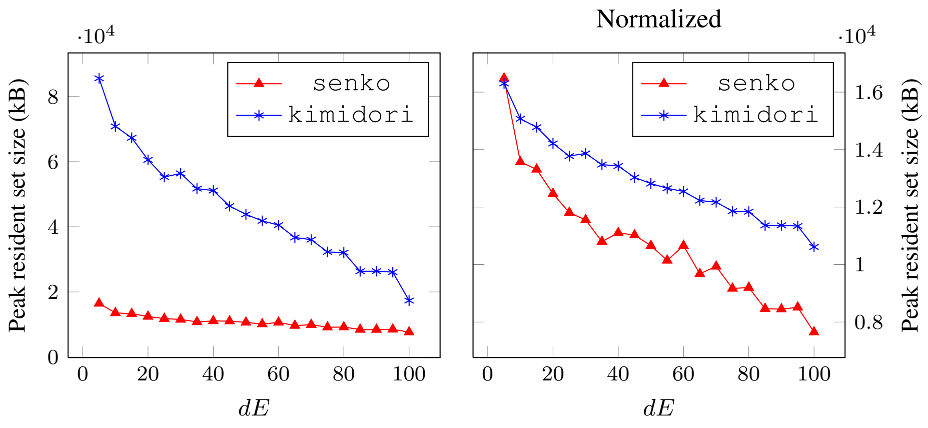
This is much worse than for PARI. NTL does not only perform worse, but it also parallellizes worse.

4.3.3 Increasing density

Time as a function of density:



Memory as a function of density:



4.3.4 Evaluation step

4.4 Naive

Not yet done.

References

[1] http://fileadmin.cs.lth.se/cs/Personal/Thore_Husfeldt/papers/lsfzt.pdf

[2] http://gmplib.org/

[3] http://man7.org/linux/man-pages/man5/proc.5.html

[4] http://www.shoup.net/ntl/index.html

[5] http://www.shoup.net/ntl/doc/ZZX.txt

[6] http://pari.math.u-bordeaux.fr/

[7] https://en.wikipedia.org/wiki/Toom%E2%80%93Cook_multiplication

[8] https://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen_algorithm

[9] Haggard, Pearce, Royle, 2010: *Computing Tutte Polynomials*