Performance measurements of a small-space Chromatic Polynomial algorithm

Mats Rydberg

October 14, 2013

1 Chromatic Polynomial in small space

The algorithm measured in this performance test is described in Björklund et al [1]. 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 $O(1.2916^n)$.

TODO: Define some terms here, like $\chi_G(t)$.

2 Implementation details

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

I've chosen to support adjancency matrices as input structure, representing an arbitrary graph. Another common graph representation is the edge list, which is faster, but since our problem is exponentially hard, another degree of a polynomial term doesn't really affect our performance very much. It is also more straight-forward for me to generate randomized graphs using an adjancency matrix.

QUESTION: Is the chromatic polynomial defined for degrees $> n$? We are exponentiating it to k, thus increasing its degree (also multiplying by some moderate degree polynomials might do this). I have assumed it isn't defined for larger degrees, and so to save time and memory I am simply throwing away all higher-degree terms after exponentiation

For polynomial representation, I've implemented two libraries for number theoretic calculations to be used by my program.

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 arithmetic, 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+=()`, `ZZX.operator*=()`.

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 by a group of mainly French computer scientists at Université Bordeaux [5], together with a calculator-like interface 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. It is a bit unclear to me how exactly PARI implements polynomial exponentiation and multiplication, but from the impression I've gotten and, more importantly, its performance, it has to be non-trivial. I was hoping to find that it implemented some version of Toom-Cook[6], but I do not know for sure.

The functions I use are primarily these: `ZX_add()`, `ZX_mul()`, `gpowgs()`.

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'ven 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. Initially, I also implemented my own, naive polynomial class, but its performance has not been measured compared to the external libraries yet. I expect that it will not have much of a chance.

3 Test structure

In the k-problem testing, it was easier to isolate variables, because there was an exponentially large definition base of one of them ($|\mathcal{F}|$). For the chromatic polynomial, I found it was harder. This is the main reason that I am not setting $|E|$ as constant and varying n, but I set dE constant, which is the *density* of edges in the graph, counted in percent.

The test cases I created were these:

1. Set $k = 3$, $dE = 40$ and vary $n = 4, 6, 7, 8, \ldots, 26, 27$
2. Set $k = 10$, $n = 19$ and vary $dE = 5, 10, 15, \ldots, 95, 100$
3. Set $n = 19$, $dE = 40$ and vary $k = 2, 3, 4, \ldots, 18, 19$

Note that because of an unknown error, $n = 5$ was not tested.

4 Test results

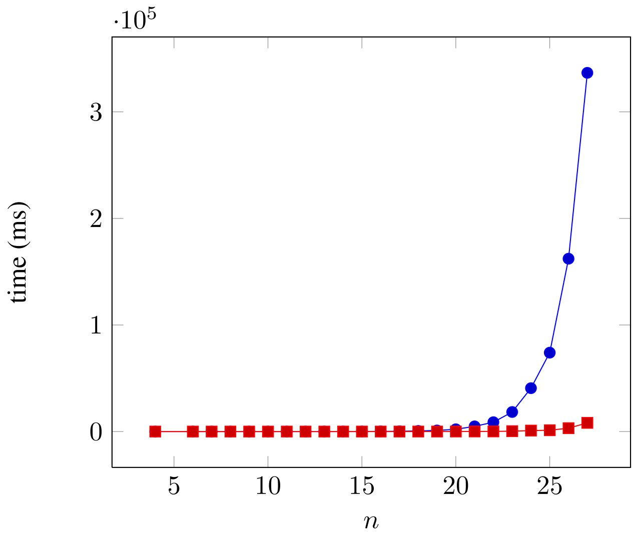
Here follows the performance graphs for all tests. Some interpretational text is also provided. The blue circles show user time in the time graphs and virtual memory in the memory graphs. The red squares show system time in the time graphs and resident set size in the memory graphs.

4.1 PARI 2.5.5

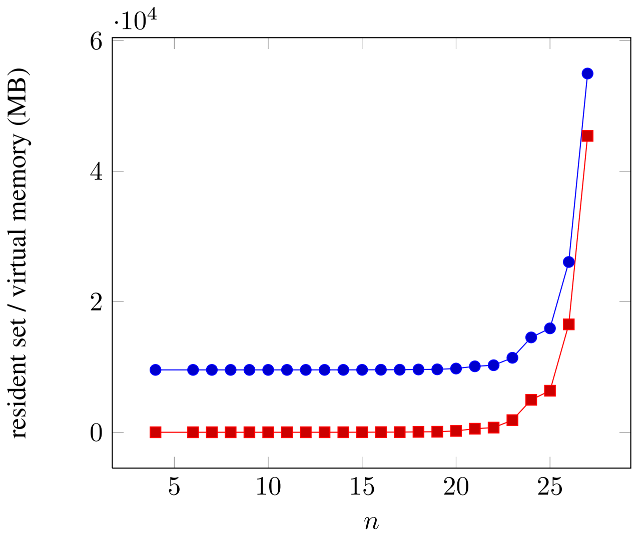
This is the most well-performing program. It computes $\chi_G(t)$ for $n = 27$ in under 350 seconds.

Note that some of the memory diagrams only include one plot (which is the resident set size). This is because PARI is implemented using its own stack, which is also pre-allocated (for speed). I'm pre-allocating it about 10GB of space initially, giving a "constant" virtual memory peak at that value.

4.1.1 Test 1

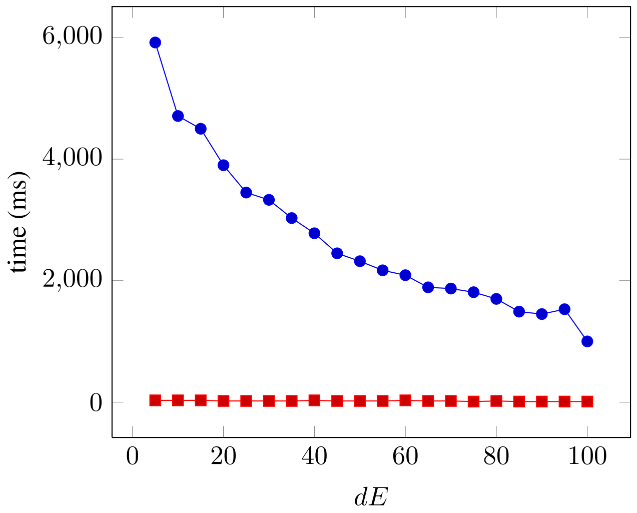


Not surprisingly, time shoots upwards really fast when we get sizeable values of n.



And the same for memory usage. It is hard to see from this graph how much less the actual increase in memory is compared to an asymptotical $O(2^n)$.

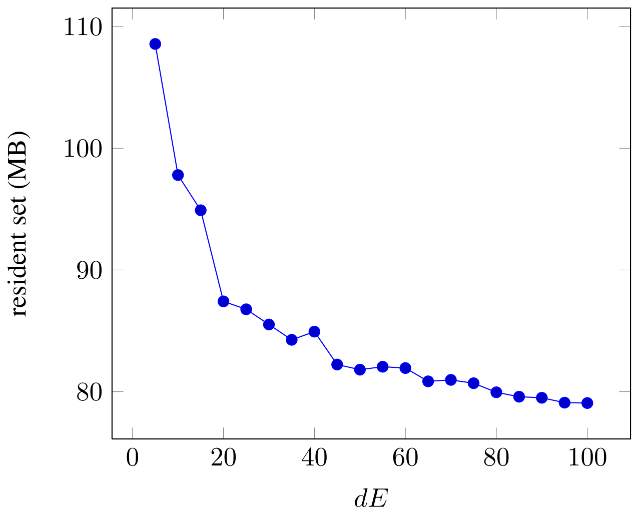
4.1.2 Test 2



This result is the most fascinating one. Here we see a clear systematic *decrease* in time usage as we *increase* the number of edges in the graph. This is however expected after some careful analysis of the detailed parts of the algorithm. In particular, this row:

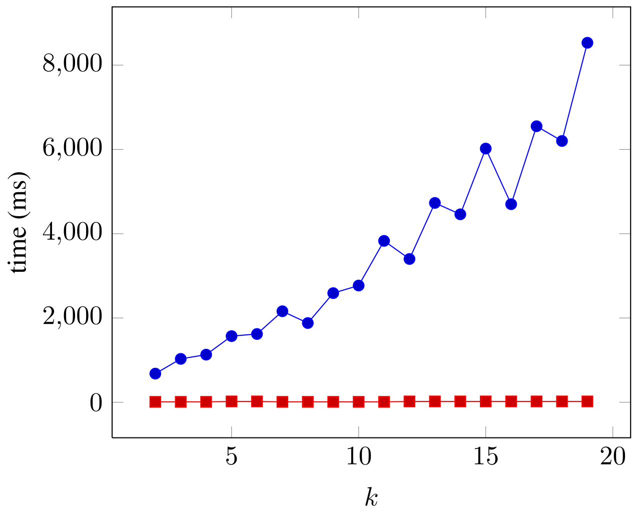
$$h(V_2 \setminus N(Y_1)) \leftarrow h(V_2 \setminus N(Y_1)) + z^{|Y_1|}[Y_1 \text{ is independent in } G]$$

Here we see that if Y_1 is not independent, no calculation is made (addition with zero). With more edges in the graph G, we naturally have smaller likelihood of Y_1 being independent, and less calculations at this step. This also have the consequence that more of the values in the vector h are zero, which in turn give even fewer calculations in the continuation of the algorithm.

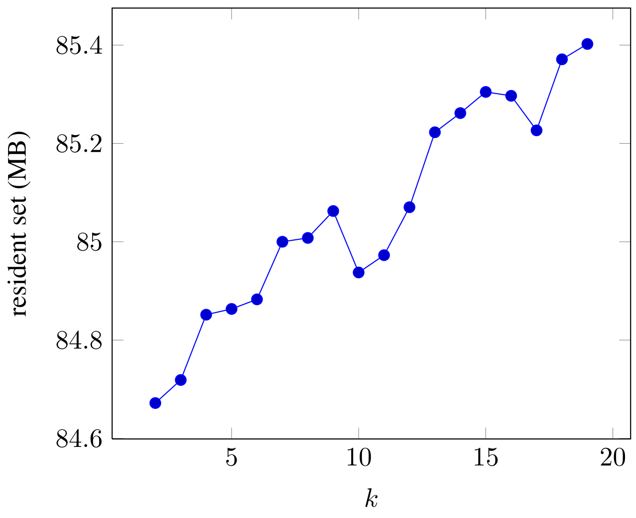


Here we see a similar result as in the time graph. As $|E|$ (or dE) increases, we use less space also.

4.1.3 Test 3



Also an interesting result, although perhaps not very unexpected. Taking larger powers naturally takes more time (more multiplications), but why we have such a variety between odd and even powers remains unclear. Note that the *even* powers are the faster ones.

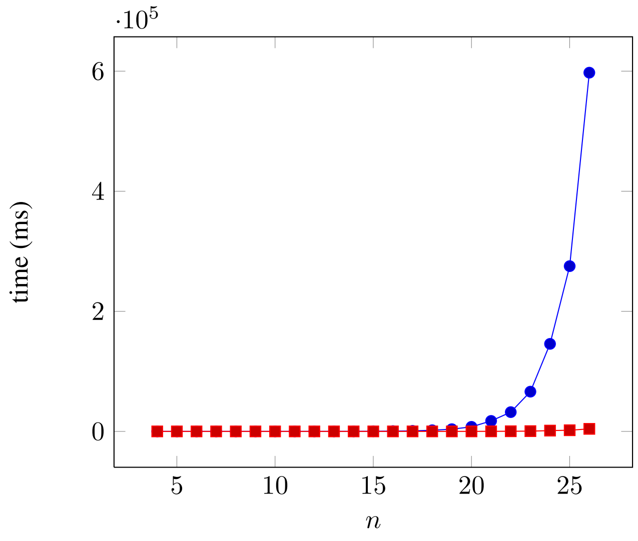


Looking at the values of the y-axis, we see that this is more or less a constant curve. Not much changes as k increases.

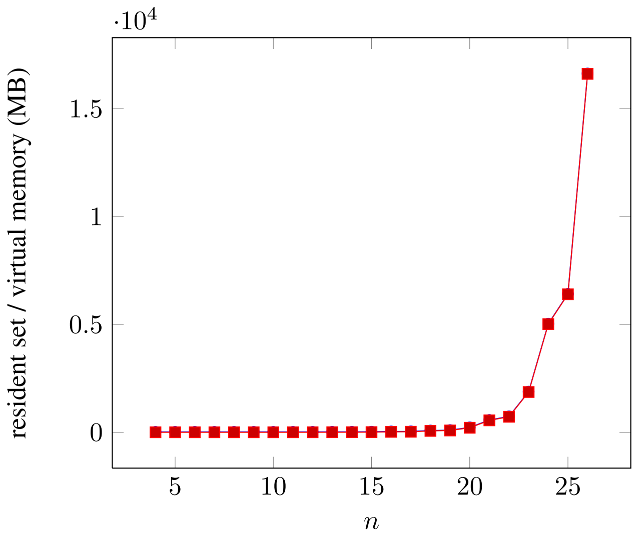
4.2 NTL

Write some text here.

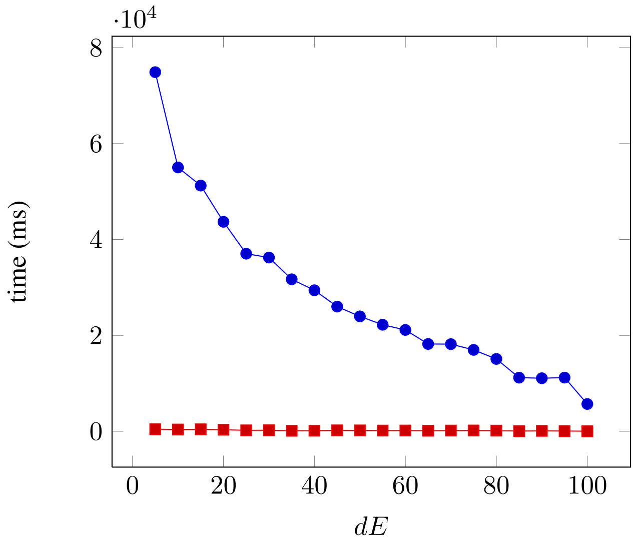
4.2.1 Test 1



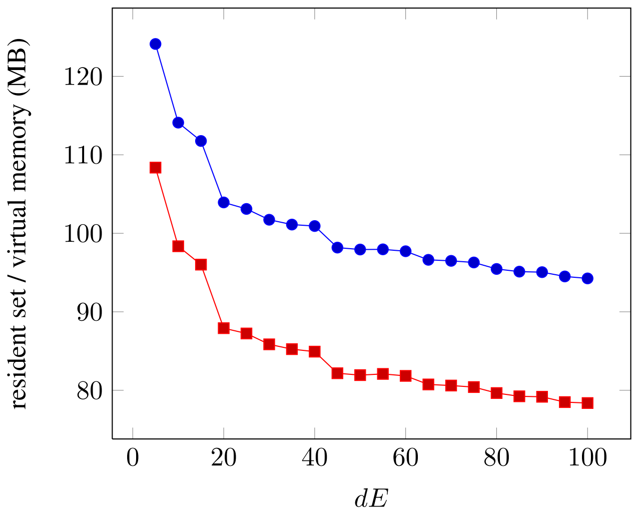
Not surprisingly, time shoots upwards really fast when we get sizeable values of n.



4.2.2 Test 2

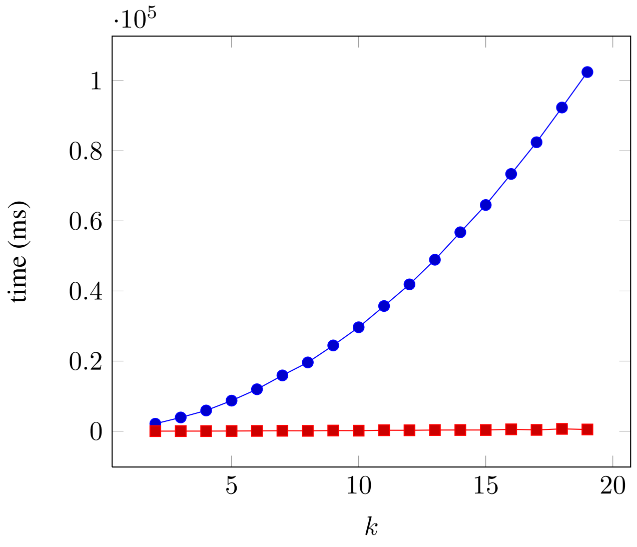


$$h(V_2 \ N(Y_1)) \leftarrow h(V_2 \ N(Y_1)) + z^{|Y_1|}[Y_1 \text{ is independent in } G]$$

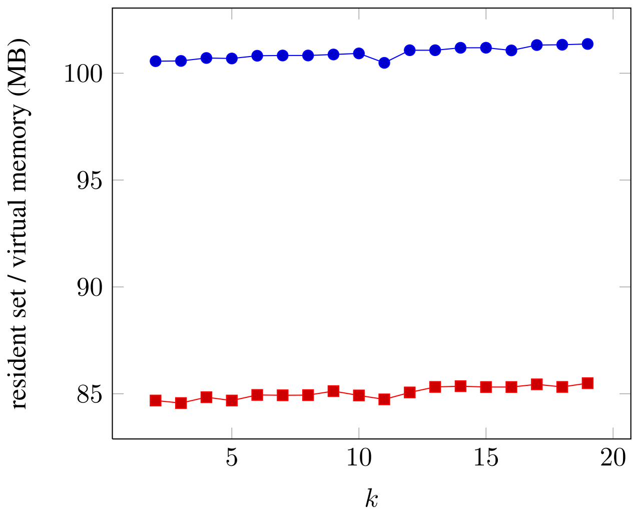


resident set / virtual memory (MB)
120
110
100
90
80
0
20
40
60
80
100
dE

4.2.3 Test 3



$\cdot 10^5$
time (ms)
1
0.8
0.6
0.4
0.2
0
5
10
15
20
k



4.3 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://pari.math.u-bordeaux.fr/

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