A first test report on simulations of Fast Zeta Transform-based algorithms

Mats Rydberg

January 16, 2014

Contents

1 Introduction

In this text, I will present some results from my work in the form of benchmarks for a few variations of the algorithms examined in this Master's Thesis. The core of all algorithms is the Fast Zeta Transform in Linear Space (FZT), see [1], which makes the programs look very much alike. Four programs are presented in this paper, three of which are solving well-known algorithmical problems. These three are k-cover, k-partition and k-packing, further explained below. The fourth is the "clean" FZT.

One question that this test seeks to answer, or at least to make an indication toward an answer for, is whether the complexity of the arithmetics performed with polynomials in the partition and packing problems is a significant time-consuming part of the algorithm. That is, would it be worthwhile to represent the polynomials in a non-explicit way in order to perform faster multiplication and addition? In this text, we investigate the use of a naive polynomial implementation, to see if it is worthwhile to invest in an alternative implementation.

2 Problem statements

2.1 Fast Zeta Transform

Given a universe $U = \{1, 2, \ldots, n\}$ of integers, a family $\mathcal{F}$ of subsets of U and a function $f : \mathcal{F} \rightarrow R$, where R is an algebraic ring[1], output the Fast Zeta Transform $f\zeta$ of f, given by

[1]In this paper, R is either the set of integers modulo 2 $\mathbb{Z}_2$ or the ring of polynomials $\mathbb{Z}_2[z]$ of degree n with coefficients from $\mathbb{Z}_2$ (for some indeterminant z).

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

For a detailed description of the algorithm, see Björklund et al [1].

2.2 k-cover

Given a universe $U = \{1, 2, \dots, n\}$ of integers, a family $\mathcal{F}$ of subsets of U and an integer k, output $c_k(\mathcal{F})$, which is the number of ways to pick k members of $\mathcal{F}$ such that their union equals U.

Solution: Set f as the characteristic function of $\mathcal{F}$ ($f = 1$ for each $Y \in \mathcal{F}$) and calculate its FZT $f\zeta$. Then get c_k as

$$c_k = \sum_{X \subseteq U} (-1)^{|U \setminus X|} f\zeta(X)^k$$

2.3 k-partition

Given a universe $U = \{1, 2, \dots, n\}$ of integers, a family $\mathcal{F}$ of subsets of U and an integer k, output $d_k^n(\mathcal{F})$, which is the number of ways to pick k pairwise disjoint members of $\mathcal{F}$ such that their union equals U.

Solution: Set $f(Y) = z^{|Y|}$ for each $Y \in \mathcal{F}$. Calculate $f\zeta$ and get d_k^n as the coefficient of z^n in

$$d_k = \sum_{X \subseteq U} (-1)^{|U \setminus X|} f\zeta(X)^k$$

2.4 k-packing

Given a universe $U = \{1, 2, \dots, n\}$ of integers, a family $\mathcal{F}$ of subsets of U and an integer k, output $p_k^n(\mathcal{F})$, which is the number of ways to pick k pairwise disjoint members of $\mathcal{F}$.

Solution: With the insight that a k-packing is indeed a $(k+1)$-partition with the $(k+1)$:th member being an arbitrary subset of U (all the remaining elements in U), we set f as we did with k-partition and get p_k^n as the coefficient of z^n in

$$p_k = \sum_{X \subseteq U} (-1)^{|U \setminus X|} (1+z)^{|X|} f\zeta(X)^k$$

3 First implications and limitations

My initial idea was to test many things. Too many, I soon realized, as some simple calculations showed that I would need over a hundred graphs to present the results. This is of course infeasible due to lack of space and time (no pun intended). So, instead of allowing both `std::vector`- and pointer-based list structures, I will resort to only keep pointer-based lists. Although it would be interesting to see whether `std::vector` provides a lot of overhead for large structures, this will have to be deduced elsewhere.

There are two kinds of integer types used in my program. First there are the index variables for my list structures. They need to be large enough to hold all of the members of $\mathcal{F}$. Since $|\mathcal{F}|$ grows as $O^*(2^n)$, it is the size of n that sets the limitation on our index variables. For convenience, I choose to let my program support $n < 31$, allowing me to use a standard 32-bit sized integer. For the instances provided here, maximum input size is $\sim 10^6$ lines, which means that for $n = 30$ the maximum supported density[2] of subsets is around $1‰$. For $n < 21$, there is no limitation on subset density.

Second, there are the summation variables needed to output the solution for the k-problems. In order to produce correct results, these do naturally need to be arbitrarily large. My programs uses GMP, the GNU Multiple Precision library [2], to support this. An initial idea was to test also the performance of GMP as compared to any standard data type (such as `int`), but as 32 bits is too small even for $n = 12$ (for some k), this was abandoned. The main reason as to why the sums grow so incredibly fast is due to the fact that the algorithm consider ordering of subsets in a k-sized selection to be critical. That is, for k distinct chosen members of $\mathcal{F}$ that solve the problem, there are $k!$ solutions counted.

3.1 Complexity of arithmetics

Since the FZT is running in time $O^*(2^n)$ and space $O^*(|\mathcal{F}|)$, so do the k-problem algorithms as well, assuming that the arithmetics in the ring R take constant time and that each ring element uses constant space. But what about the summation variables of the k-problems? As mentioned, they need to be stored in an arbitrarily large data type, but how large? How fast does it grow and what implications does this have on its use as a term or a factor?

For k-cover, we have an integer that in each step of the inner-most loop (there are $O^*(2^n)$ such steps) take the FZT $f\zeta(X)$ for some $X \in \mathcal{F}$ and raise it to the k. The largest possible value of $f\zeta(X)$ is 2^n. This happens when the density of subsets is at 100% and only for $X = U$ (which has all other sets in $\mathcal{F}$ as subsets). This (and every other) number is taken to the power of k. In k-cover, a set can be picked more than once for a given selection that solves the problem, leaving us with no bound at all on k. This means that the maximum value of the largest term in the sum of c_k is 2^{nk}, where k may approach ∞. In my programs, I have decided to let $k < |\mathcal{F}|$ hold, effectively bounding k by 2^n. Our maximum term is now $(2^n)^{2^n}$.

TODO (or: message to Thore): Not sure where I am going with this... relevance? What's my point? How hard is it exactly to sum with this number?

4 Test results

4.1 Test structure

All problem instances are stored in a file named `*p_n_df` where `*` is `k` for the k-problems and `f` for the FZT, `n` is the size of n and `df` is the density *df* of $\mathcal{F}$ in parts of thousands (‰).

My measures are performed by pausing the process just before normal termination and reading from the process' stat file in `/proc/[pid]/stat` (see the linux man pages for `proc` [3]). The information gathered is the amount of time used (both user and system time), the peak size of the resident set and the peak virtual memory.

The tests are as follows (k is of course not a parameter for the FZT).

[2]In this paper, the *density df* of $\mathcal{F}$ is defined as $|\mathcal{F}| = df \cdot 2^n$.

1. Sets $n = 15$ and $k = 30$ constant and varies *df* over 21 measuring points:

$$1‰, 50‰, 100‰, \dots, 1000‰$$

2. Sets $|\mathcal{F}| \sim 2^{14}$ and $k = 10$ constant and varies n (this means *df* also varies, but that's not relevant) over 8 measuring points:

$$14, 15, \dots, 21$$

3. Sets $n = 15$ and $df = 100$ constant and varies k over 98 measuring points:

$$2, 3, 4, \dots, 49, 50, 55, 60, \dots, 145, 150, 160, 170, \dots, 290, 300,$$

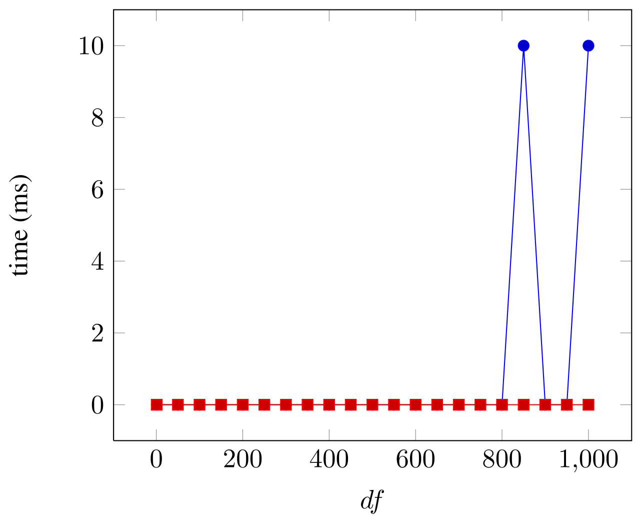
$$350, 400, \dots, 950, 1000$$

All time measurements are made with a precision of 10 milliseconds. For time plots, blue circles show user time and red squares show system time. For memory plots, blue circles show peak resident set size and red squares show peak virtual memory.

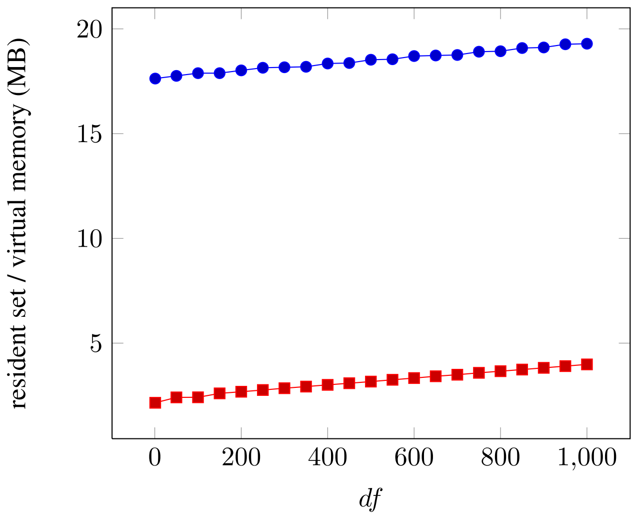
4.2 FZT in Linear Space

Test results for "just" the Fast Zeta Transform are presented here. Note that this program does not perform actual output of the values of $f\zeta$, but just calculates them and then continues the loop. That is of course the very technique that allows the algorithm to use only linear space.

Test 1

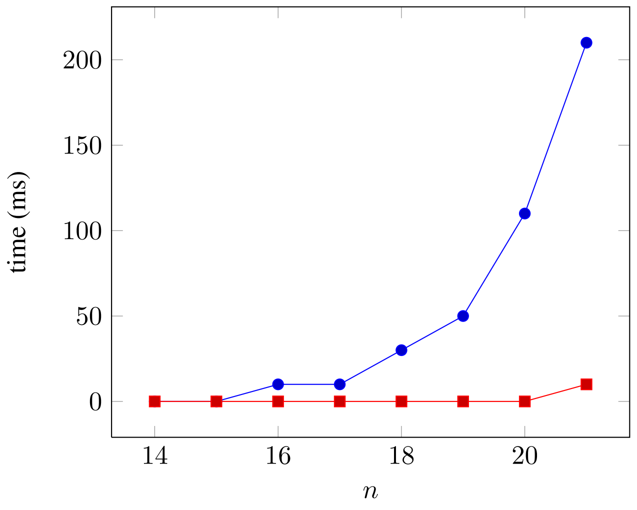


Calculating the FZT is really fast and our precision is too low for any interesting results for this test.

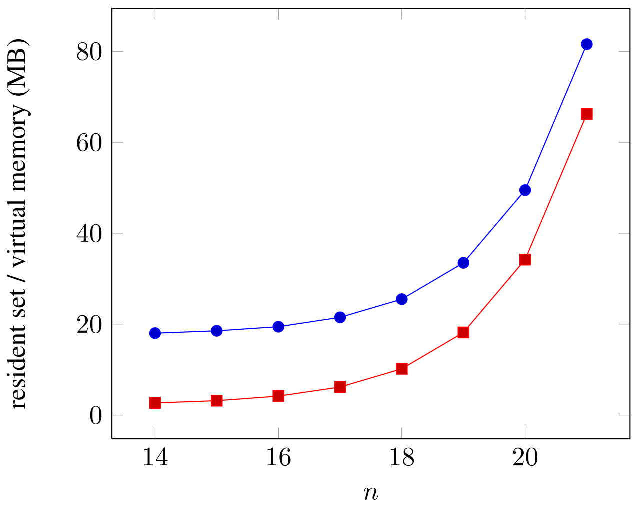


A very shallow linear increase is seen in memory usage for the FZT as we increase the size of the set family $\mathcal{F}$.

Test 2



The amount of system time used is too low to be measured, but we see the expected exponential increase in user time as we increase n.



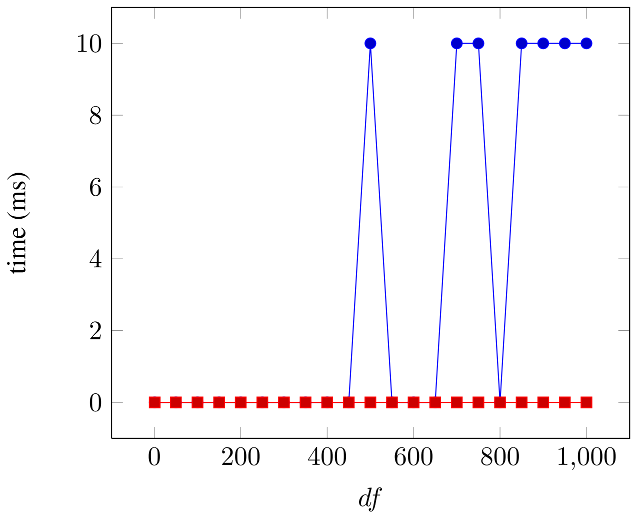
We have a quite clear exponential increae in memory usage as we keep $|\mathcal{F}|$ constant and increase n, which was slightly unexpected.

Test 3

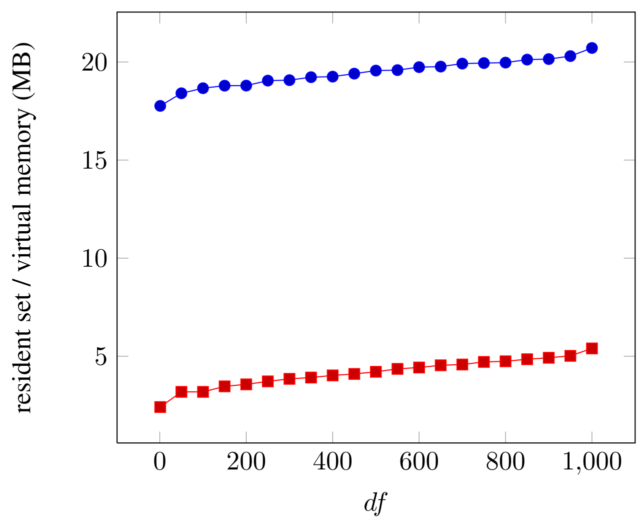
Test 3 does not apply for the FZT.

4.3 k-cover

Test 1

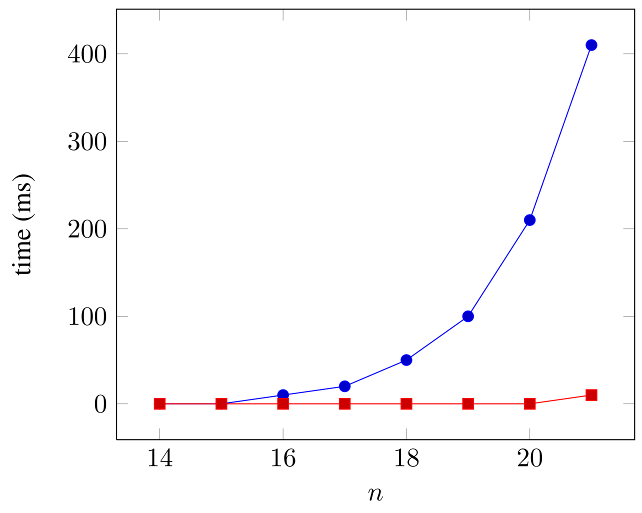


Time measurements were too imprecise to provide any useful information.

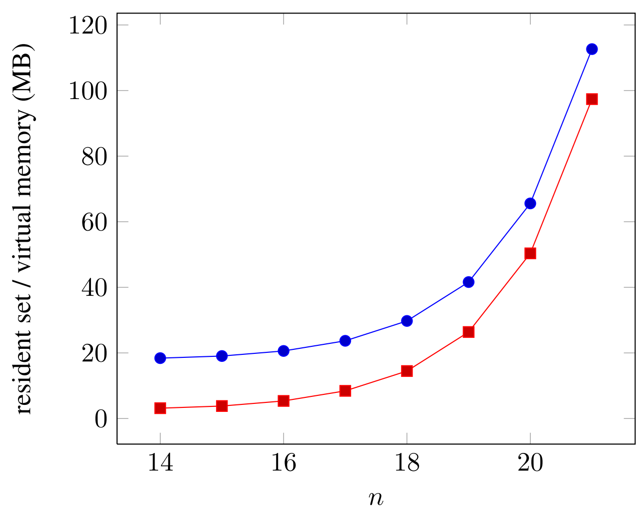


Memory usage increases in a linear fashion for increasing set family size.

Test 2

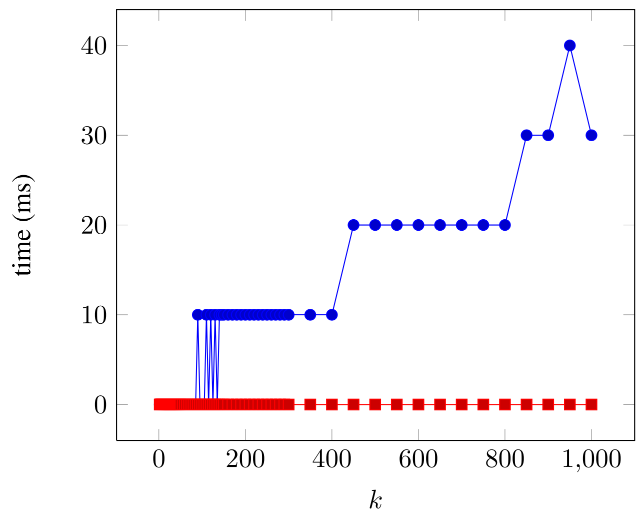


System time measurements are too imprecise, but user time measurements provide the expected exponential increase.

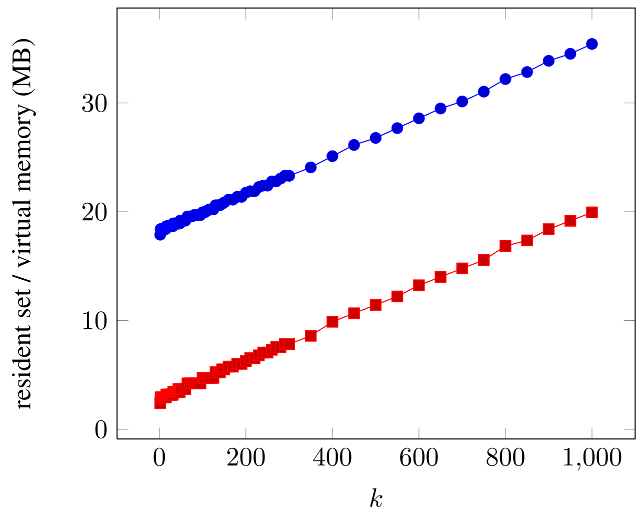


Exponential curves in memory usage even when keeping the set family size constant.

Test 3



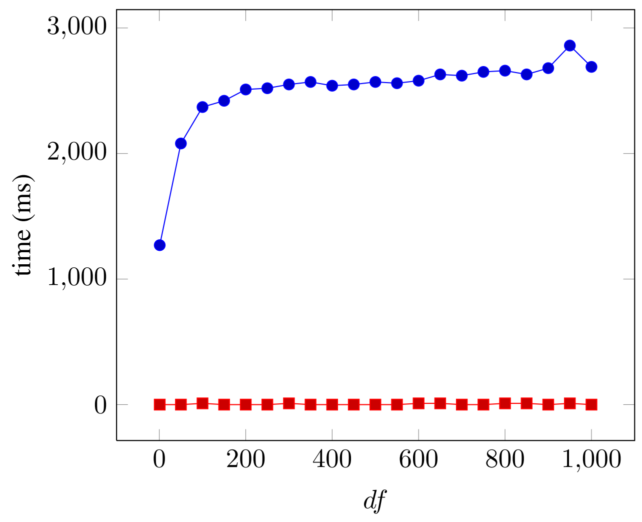
We do see that k does influence time usage, but it doesn't seem to be a very influential parameter. System time is still too small to be measured.



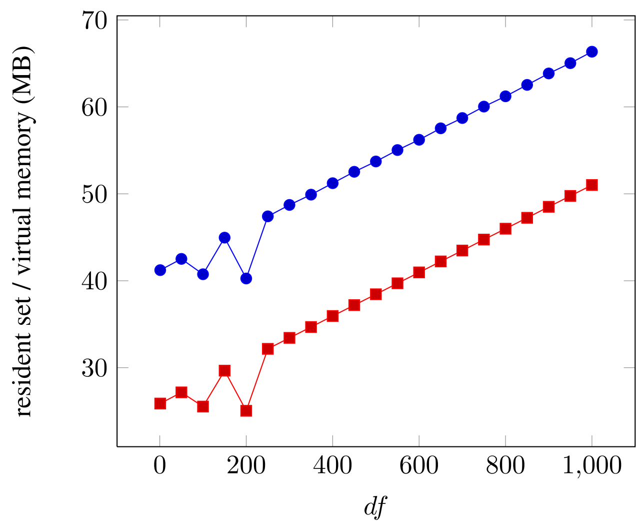
On the memory side of things, k does seem to be an influential parameter, which is to be expected. This is of course mainly in the memory needed for the terms in the output sum.

4.4 k-partition

Test 1

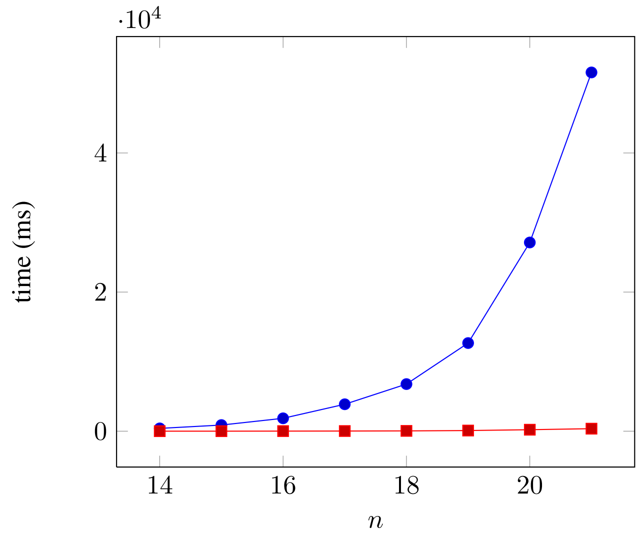


Moving over to arithmetics with polynomials has a huge impact time usage. We can still not measure system time, but user time can now be measured in whole seconds. The increase is however very small, and indicates a logarithmic increase in time.

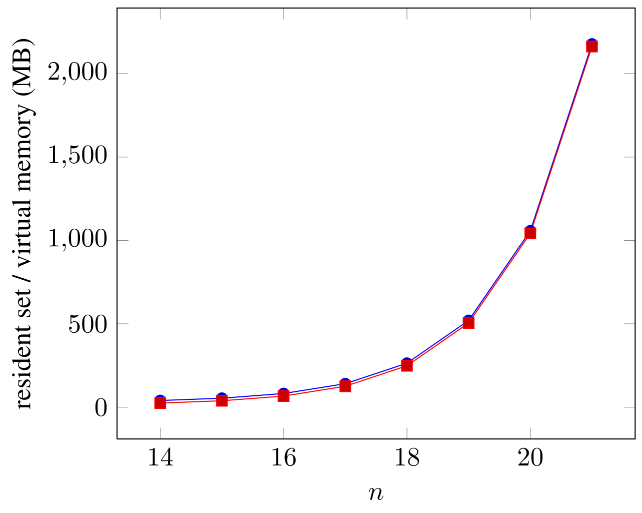


Except for some odd points, this is a very straight linear increase in memory usage.

Test 2

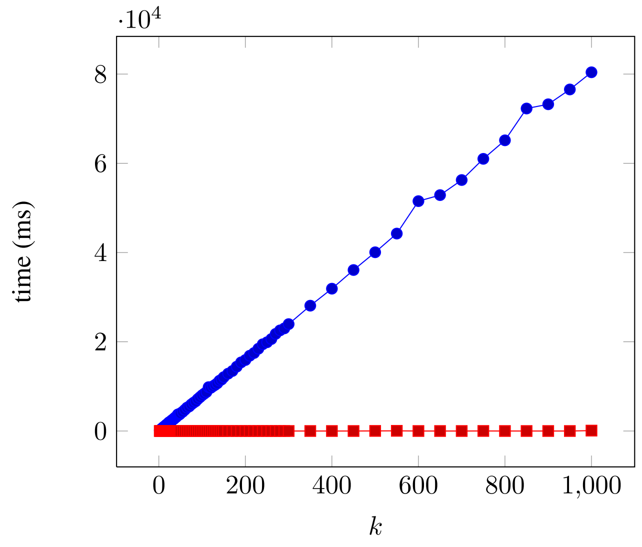


Time usage is still exponential, but polynomial arithmetics take much more time. The large tests are measured in whole minutes.

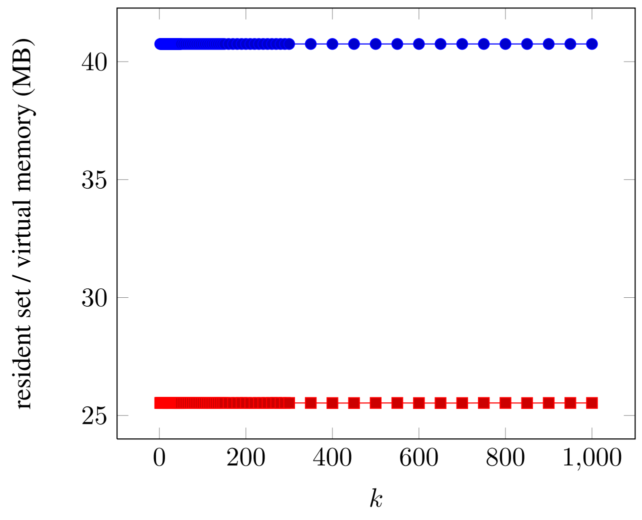


With memory increase being very rapid and with a large base, we see that resident set size and virtual memory usage are asymptotic to eachother. It's still clear that memory usage increases exponentially with increasing n.

Test 3



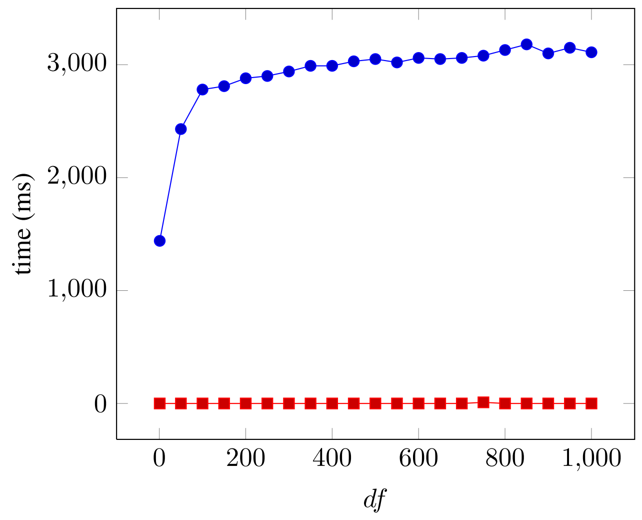
Now our tests take measureable amounts of time, and a clear linear increase is seen as we increase k. For the polynomials, k does seem to be an influential parameter.



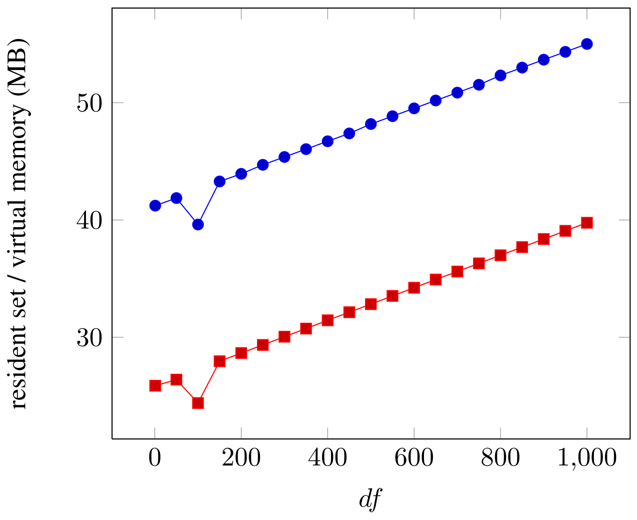
Interestingly enough, the impact of polynomials is that we have traded time for memory; this graph shows us memory is constant through increasing k, but we did pay with linear time increase.

4.5 k-packing

Test 1

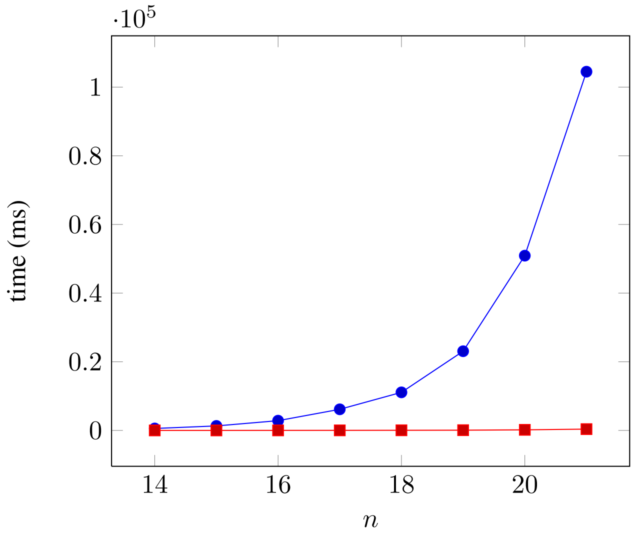


Test results are nearly exactly the same as for k-partition, only with a constant factor more expensive for k-cover. This is expected due to the algorithms design.

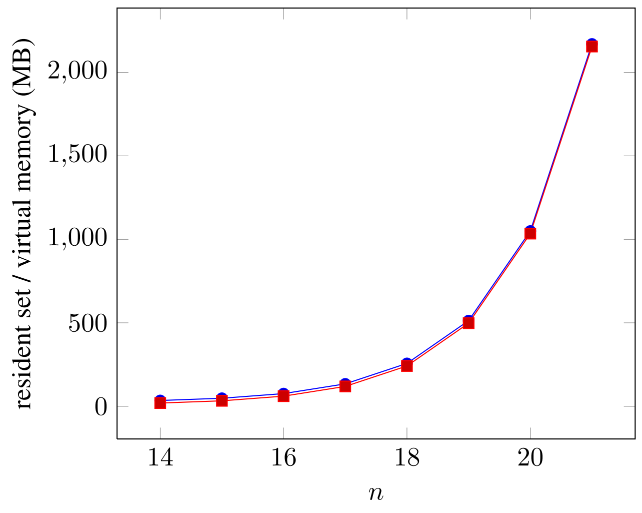


An even straighter linear increase in memory usage than for k-partition.

Test 2

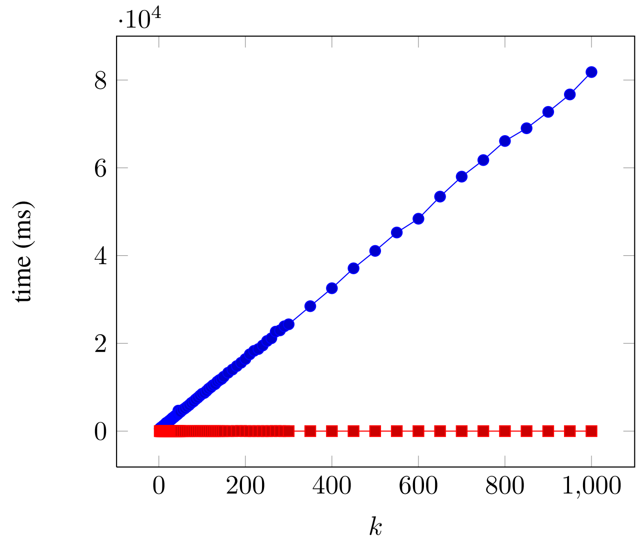


More time used than for any other test, but with the same asymptotic complexity as all others.

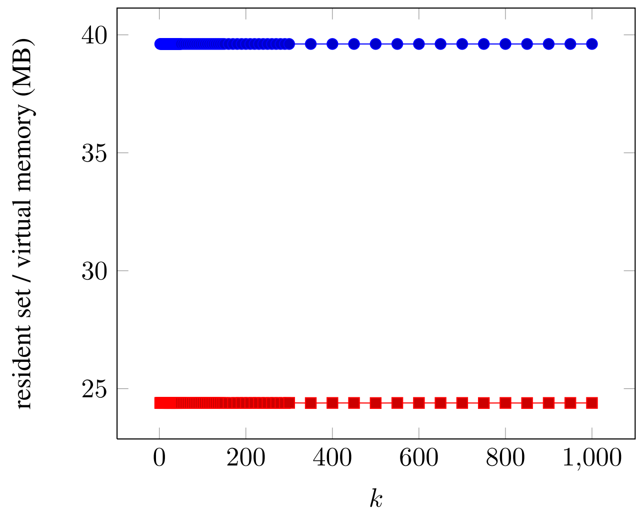


Very similar results as for k-partition.

Test 3



Polynomials give us linear increase in time . . .



…in exchange for constant memory.

5 Conclusions

Test 1 indicates that time dependence on the size of the set family $\mathcal{F}$ is $O(\log |\mathcal{F}|)$ *when we use polynomials* and memory usage is $O(|\mathcal{F}|)$ for both polynomials and integers, both of which are interesting pieces of information. Further testing will have to deduce whether k-cover also provides a logarithmic increase in time as *df* increases, as current tests terminated too quickly for our measurement precision.

Test 2 shows nothing unexpected when it comes to time usage, but the memory usage measurements indicates an exponential increase, even though $|\mathcal{F}|$ is kept constant. This may be expected for the k-problems, as increasing n does mean the sums grow due to the number of terms increasing. But the same pattern present itself even for just running the FZT, despite the claim in [1] that the FZT uses space as $O^*(|\mathcal{F}|)$, which is constant in test 2. Further discussion and simulations will have to investigate why.

Test 3 shows a very interesting phenomenon connected to the use of polynomials: memory usage is asymptotically less than for arithmetics with integers! Again, time is nearly unmeasurable when we use integers, but we do see that there is an increase, perhaps linear. For polynomials, we see a clear linear increase in time, which indicates that for larger instances of problems, choosing to work with polynomials may well be better than integers, as we do need increasing space, and get the same asymptotic increase in time.

6 Code

Here is presented the implementation of the Fast Zeta Transform. It is the same for all four programs, with only two alterations: `r_val` is a polynomial or an integer, depending on problem, and the summation in the inner-most loop is unique for all problems.

```
void utils::fast_zeta_transform_linear_space(
      int_t n1,
      int_t n2,
        int_list_t* family,
        rval_list_t* f,
        int_t k,
        rval_t* pk)
{

   // Variables we only want to calculate once
   int_t two_to_the_n1 = pow(2, n1);      // 2^n1
   int_t two_to_the_n2 = pow(2, n2);      // 2^n2

   // Index of U1 in the range of subsets 2^U
   int_t u1 = two_to_the_n1 - 1;
   // Index of U2
   int_t u2 = pow(2, n1+n2) - two_to_the_n1;

   // Function g
   rval_list_t g(two_to_the_n2);

   // {{ For each subset X1 of U1, do }}
   for (int_t x1 = 0; x1 < two_to_the_n1; ++x1) {
      // x1 is index of X1, but we do not handle X1 explicitly.

      // {{ For each Y2 in U2, set g(Y2) <- 0 }}
      for (int_t i = 0; i < two_to_the_n2; ++i) {
         // We just initialize a 0-vector of size 2^n2.
         // I see no need to map these values to specific
         // indices, but instead we make sure we
         // access the proper value when using g.

         // set polynomial to degree n
         g[i].set_degree(n1 + n2);
      }

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

         // {{ if YnU1 is a subset of X1 }}
         // with index math: since u1 is all-ones for elements in U1,
         // y & u1 will be the index of the set YnU1.
         // y & u1 is the index of a subset of X1 iff
         // y & u1 doesn't contain a 1 in any position where X1
         // doesn't contain a 1.
         // If y & u1 does contain a 1 in a position where X1
         // doesn't, (y & u1) | x1 will be a larger number than x1.
         if (((y & u1) | x1) <= x1) {

            // Since g contains all subsets of U2 in increasing
            // order of index, but indices of subsets of U2 doesn't
            // (generally) come at a distance of 1 from eachother,
            // they come at a distance of 2^n1 (=1 iff n1=0). Since
            // g doesn't contain "holes", we normalize the distances
            // to 1 like this.
            g[(y & u2) / two_to_the_n1] += (*f)[i];
         }
      }

      // I don't need another vector, so I re-use g as h.
      // {{ Compute h <- gS using fast zeta transform on 2^U2 }}
      fast_zeta_transform_exp_space(n2, &g);

      // The vector g contains all subsets of U2, in increasing
      // order of index. But the indices of subsets of U2 doesn't
      // come at a distance of 1 from eachother, unlike with U1.
      // They are separarated by 2^n1 positions.
      // {{ For each subset X2 of U2, output h(X2) as the value fS(X1uX2) }}

      for (int_t i = 0; i < two_to_the_n2; ++i) {
         int_t x2 = i * two_to_the_n1;
         int_t x = x1 | x2;

         ############################################
         ## To the LateX reader
         ##
         ## This is where we sum
         ## Below is the summation code for k-packing
         ##
         ############################################

         // Calculating k-packing

         int_t size_of_X = count_1bits(x);

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

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