A Self-Tuning Filtering Implementation for the Number Field Sieve

Jason Papadopoulos

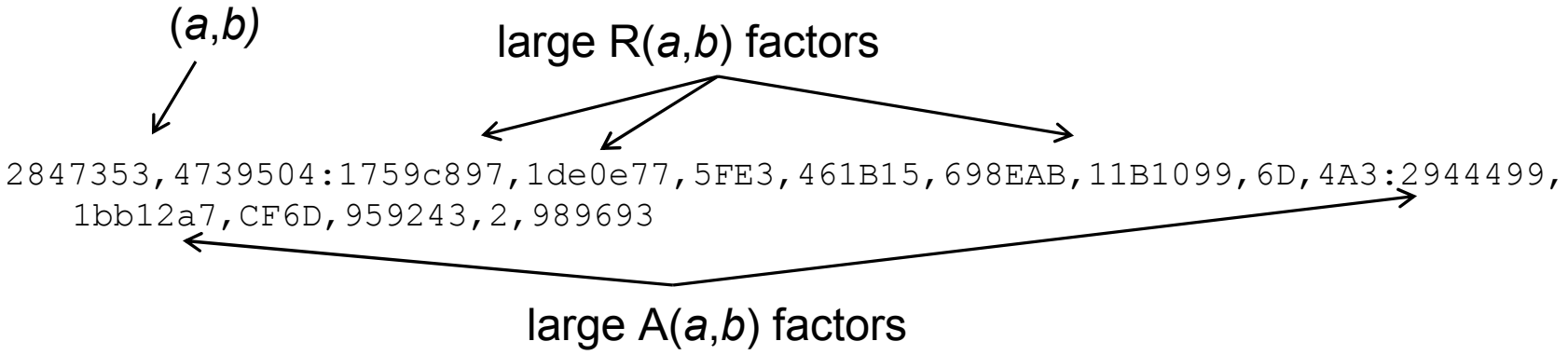
Introduction

- Msieve is a library for integer factorization that includes a complete implementation of the number field sieve (NFS) algorithm. This library has been used to complete some of the largest known NFS factorizations

- NFS code status as of version 1.38:
  - Polynomial selection: work in progress (adapting code of Kleinjung)
  - Sieving: line sieve only
  - Filtering: complete, contains the majority of development work
  - Linear algebra: complete and fast, needs performance analysis
  - Square root: complete, but fairly basic (brute force algorithm)

NFS Filtering

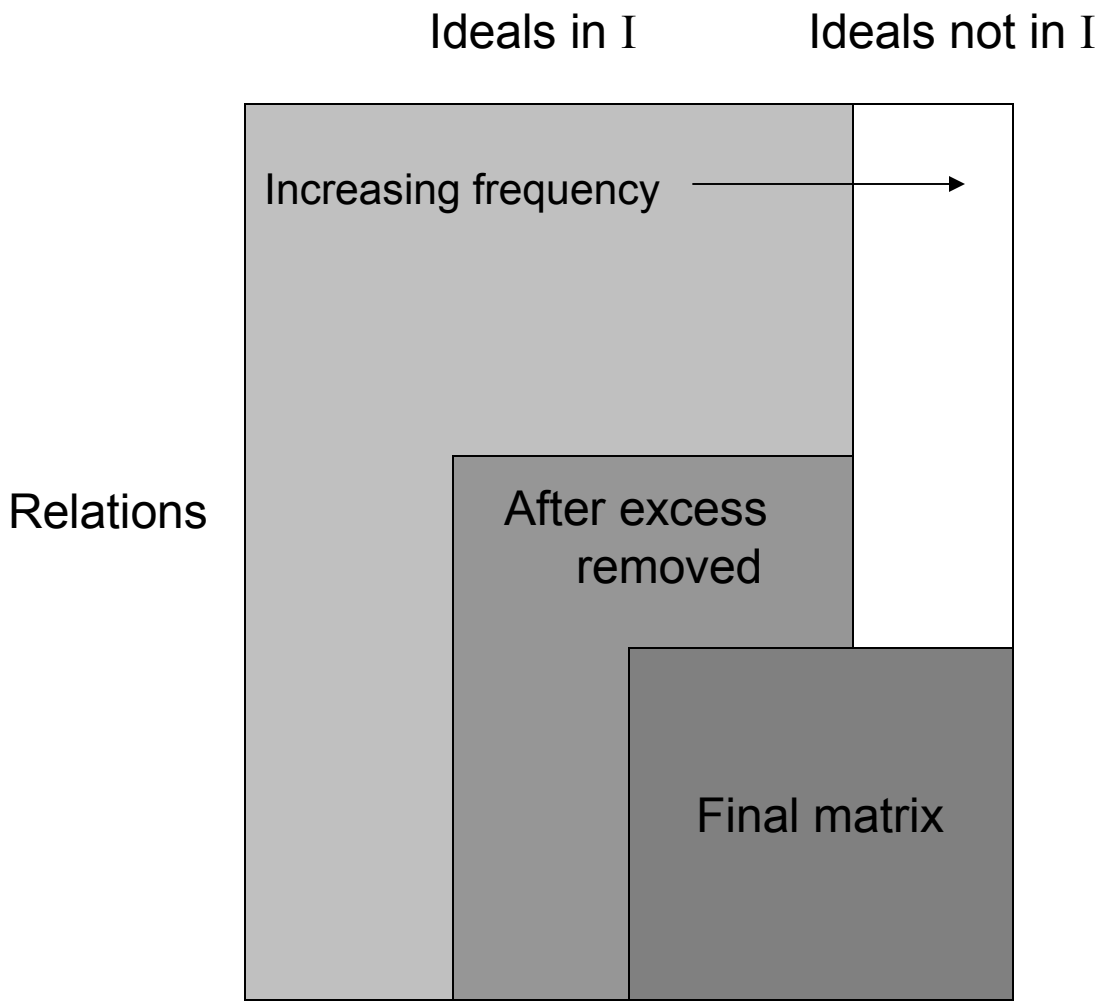
- Filtering is the process of converting a dataset of (millions to billions of) sieving relations into a much smaller collection of GF(2) vectors
- Vectors become matrix columns used by NFS linear algebra
- Each relation contains the factors of two polynomial values, the rational polynomial R(*a*,*b*) and the algebraic polynomial A(*a*,*b*). For the factorization of $6^{256}+5^{256}$ one relation could be



NFS Filtering

- Factors of these polynomials are called *ideals*, and filtering attempts to merge relations into *relation-sets* containing repeated ideals.
- In GF(2), ideals that appear an even number of times do not appear in the corresponding matrix column, so if an ideal is forced to appear an even number of times in *all* relation-sets, the matrix becomes one row smaller
- Filtering succeeds if most ideals are eliminated this way, and there is one relation-set for each ideal remaining
- If the set of relations $\mathbf{R}$ contains ideals from a set $\mathbf{I}$, then filtering can form at least ($\#\mathbf{R} - \#\mathbf{I}$) relation-sets that contain no ideals from $\mathbf{I}$ (these are cycles in a graph). This is the *excess* in the dataset
- The minimum excess required to form a matrix is the number of ideals in $\mathbf{R}$ that are not in $\mathbf{I}$

NFS Filtering Process



Linear algebra runtime ~ (matrix dimension) x (# matrix nonzeros)

Some Actual (GNFS) Numbers

Number	$6^{383}+1$ (C165)	$5^{421}-1$ (C180)
Relations	225M (~2 π(2^{31}))	383M (~1.5 π(2^{32}))
After duplicates removed	186M	302M
After singletons removed	113M relations, 82M ideals	209M relations, 146M ideals
Ideals not in $\mathbf{I}$	5.3M	13M
After most excess removed	22.6M relations, 16.5M ideals	47.7M relations, 32.6M ideals
Matrix size	7656343	16983811
Avg. column weight	77.8	75.3

NFS Filtering Challenges

- The input dataset can be very large, filtering must touch all of it
- Most relations have to be thrown away (which ones to keep?)
- The dividing line between stopping the filtering and starting the linear algebra is very fuzzy
- Poor parameter choice can make filtering and / or linear algebra impossible
- User choices are expected to adapt to the data (good filtering job sometimes requires trial-and-error by a skilled user)

Msieve Filtering Subsystem

- 4600 lines of C code
- Designed to be completely automatic:
  - No externally visible parameters
  - No user option to save intermediate results or change control flow
  - Automatically chooses filtering bound, desired matrix excess, level of merging effort, cutoff point for starting linear algebra
- Optimized to conserve memory / disk as much as possible, initial passes are always out-of-core and used fixed-size data structures (memory use hopefully bounded by that of the linear algebra)
- Making the filtering tune itself dramatically increases the potential userbase size and volume of bug reports, and reduces the turnaround time for fixes and tweaks

Internal Filtering Steps

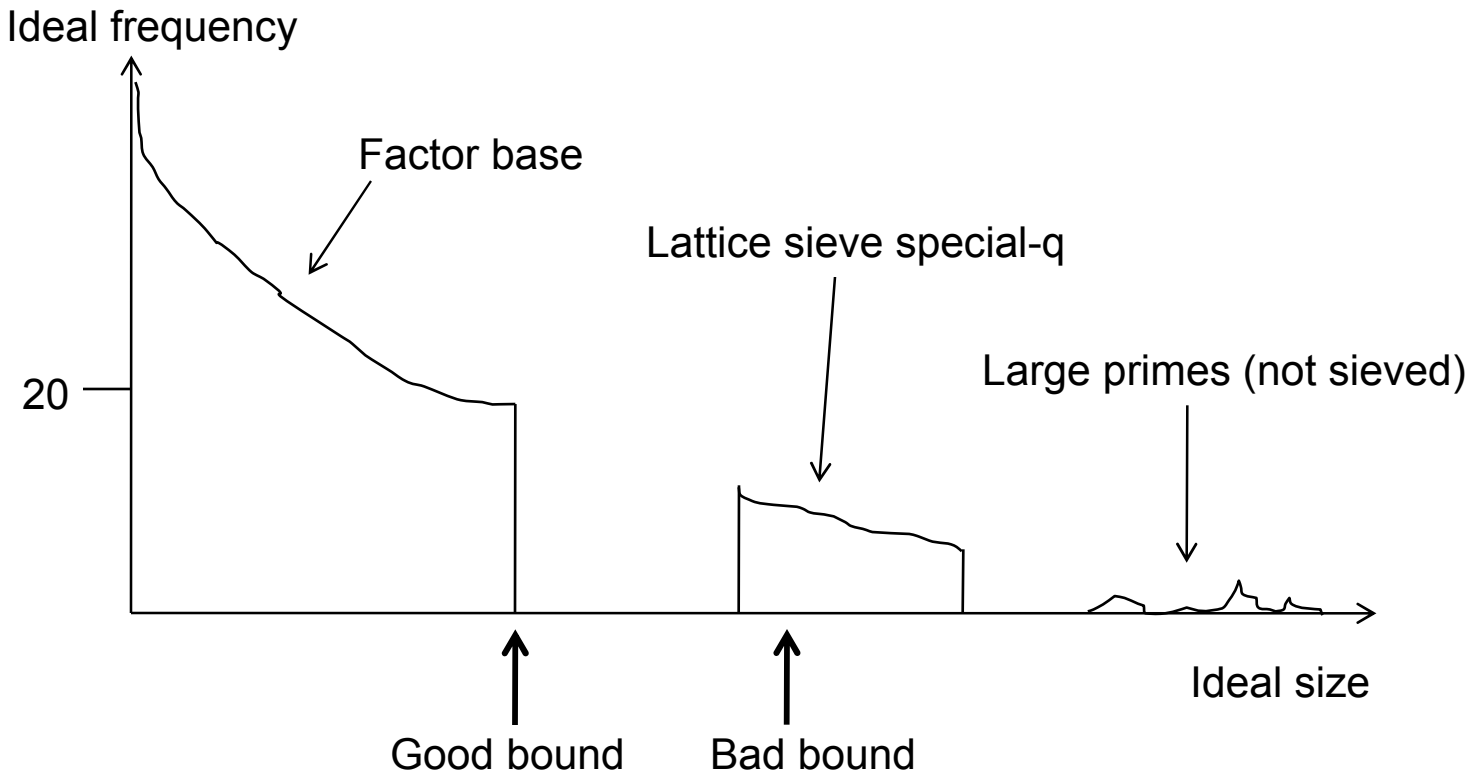
- Delete duplicates
- Decide on a filtering bound
- Choose $\mathbf{I}$, a subset of large ideals above the filtering bound
- Delete singletons
- Perform 2-way merges and delete relation-sets that are heavy, or contain many low-frequency ideals (“clique removal” from Cavallar(2000))
- Complete the merge phase
- Optimize the final collection of relation-sets

Choosing a Filtering Bound

- We want a bound that lets the filtering see only the sparse large ideals, without necessarily enumerating them; ideals that occur too often cannot be removed by the filtering, so it is wasteful to track them
- We cannot just use the bound from sieving; it may not be the right size for the filtering
- The choice of bound is important and subtle

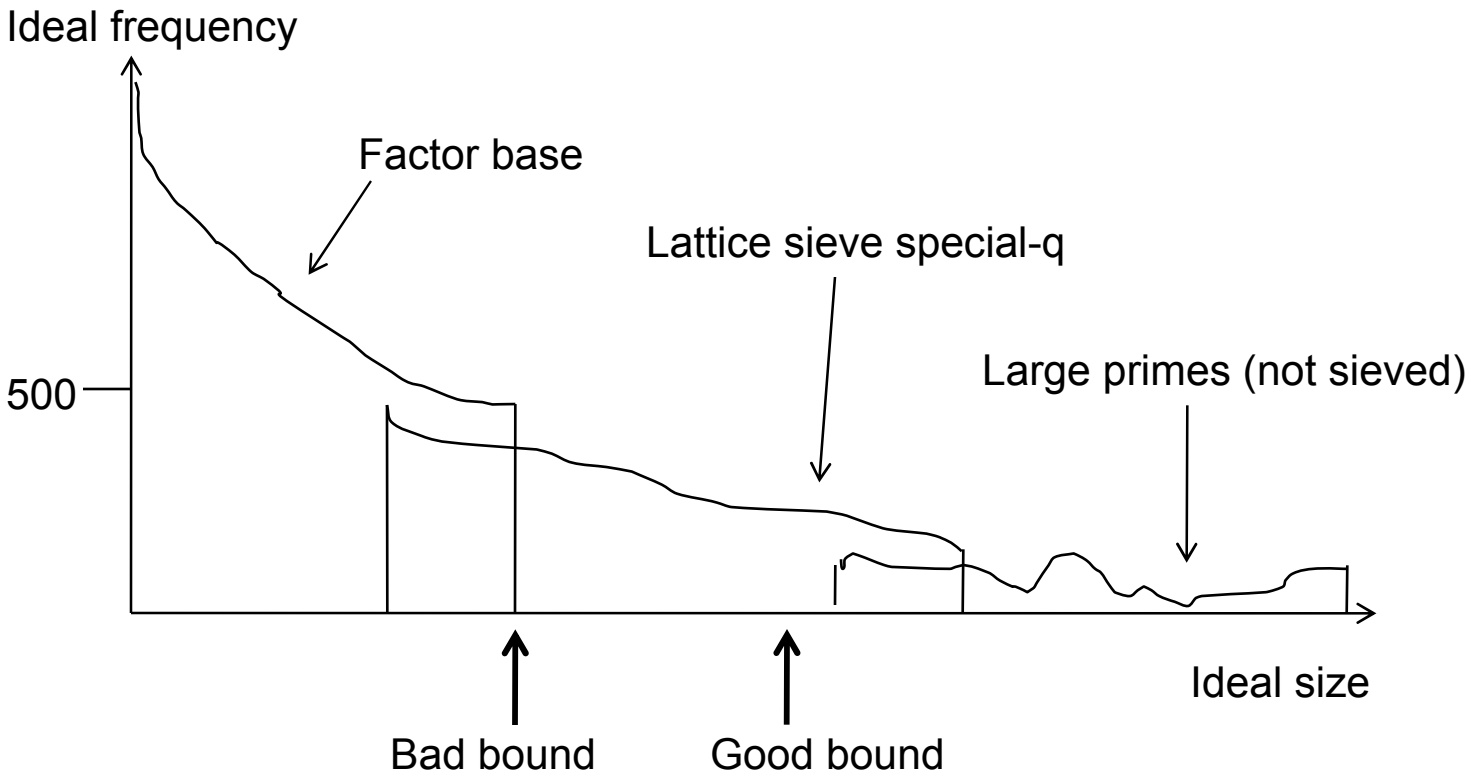
Choosing a Filtering Bound – Typical Case

Without much oversieving, making the filtering bound too large overestimates the number of dense ideals not in $\mathbf{I}$, which forces filtering to finish early, which makes the matrix too sparse for block Lanczos to be likely to succeed



Choosing a Filtering Bound – Oversieved Case

With significant oversieving, even ideals above the factor base bound have a high frequency, so the filtering bound should be large to keep the memory use manageable. Once singletons are removed, the bound may no longer be good!



The Engineering Solution: Choose Two Bounds

- Add each ideal of each relation to one 128K bucket in range $[0, 2^{32}]$; bound 1 is the middle of the last bucket for which

$$\text{(ideals in bucket)} / \text{(primes in bucket)} \geq 40.0$$

- Read in all the ideals larger than this first bound, and delete the singletons. Bound 1 may be very large but most singletons get pruned in this step

- Bound 2 is extremely small (< 1M); to avoid an explosion of memory use, $\mathbf{I}$ consists of ideals (exceeding bound 2) with frequency ≤ 20. Each relation also gets a count of ideals not in $\mathbf{I}$

- Perform the rest of the filtering; if the resulting matrix is too sparse, repeat with smaller bound 2 and larger max frequency

The Merge Problem

Given:

- A collection of ideals **I**, numbered uniquely
- A collection of relations; each relation has
  - A unique number
  - A list of large ideals from **I**
  - A count of ideals not in **I**
- *min_cycles*, the number of ideals not in **I**

Find:

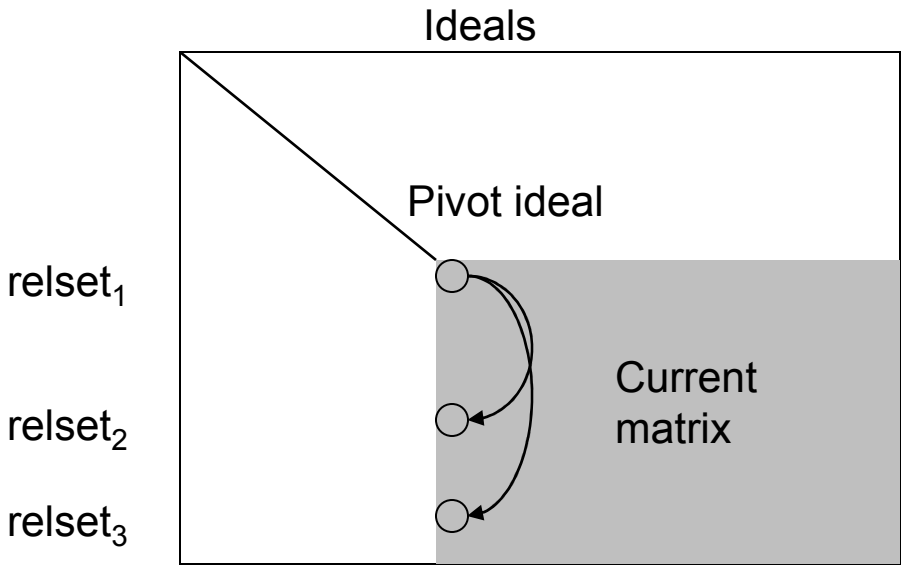
- A collection of relation-sets, each with a list of relation numbers

Such that:

- There are *min_cycles* + *S* relation-sets, where *S* is the number of ideals from **I** that appear when each relation-set has its ideal lists merged in GF(2)
- (number of relation-sets) x (number of nonzeros in relation-sets) is minimal

The Merge Problem

- Each merge operation removes one ideal and one relation-set, adding entries (“fill-in”) to the remaining relation-sets
- Removing ideals with weight > 2 looks like sparse Gauss elimination



What Ideal Merge Order Produces Minimum Fill?

- This is *the central question* of NFS filtering, and also of sparse direct solvers for linear systems
- LaMacchia/Odlyzko (1990) use ad hoc methods; Cavallar (2000) assumes the ordering is arbitrary, if the size of pivot relation-sets obeys a weight bound
- Modern machines have enough memory to store the entire current matrix as seen by the filtering, allowing a global view of the effects of a given merge (may not be true!)
- This problem is hugely important for math and industry, and has been studied for decades; compared to floating-point matrices,
  - NFS matrices are extremely unsymmetric, extremely dense (compared to matrices from engineering problems) and *extremely* large
  - Numerical stability is not an issue in GF(2)
  - Accidental cancellation rarely happens in floating point problems but will happen often for NFS matrices

Markowitz Pivoting (1957)

- Given c relation-sets containing a given large ideal, if relation-set i contains r_i nonzeros then (an upper bound on) the fill-in created by merging these relation-sets is the *Markowitz weight*

$$(\min_i \ r_i - 1)(c - 1)$$

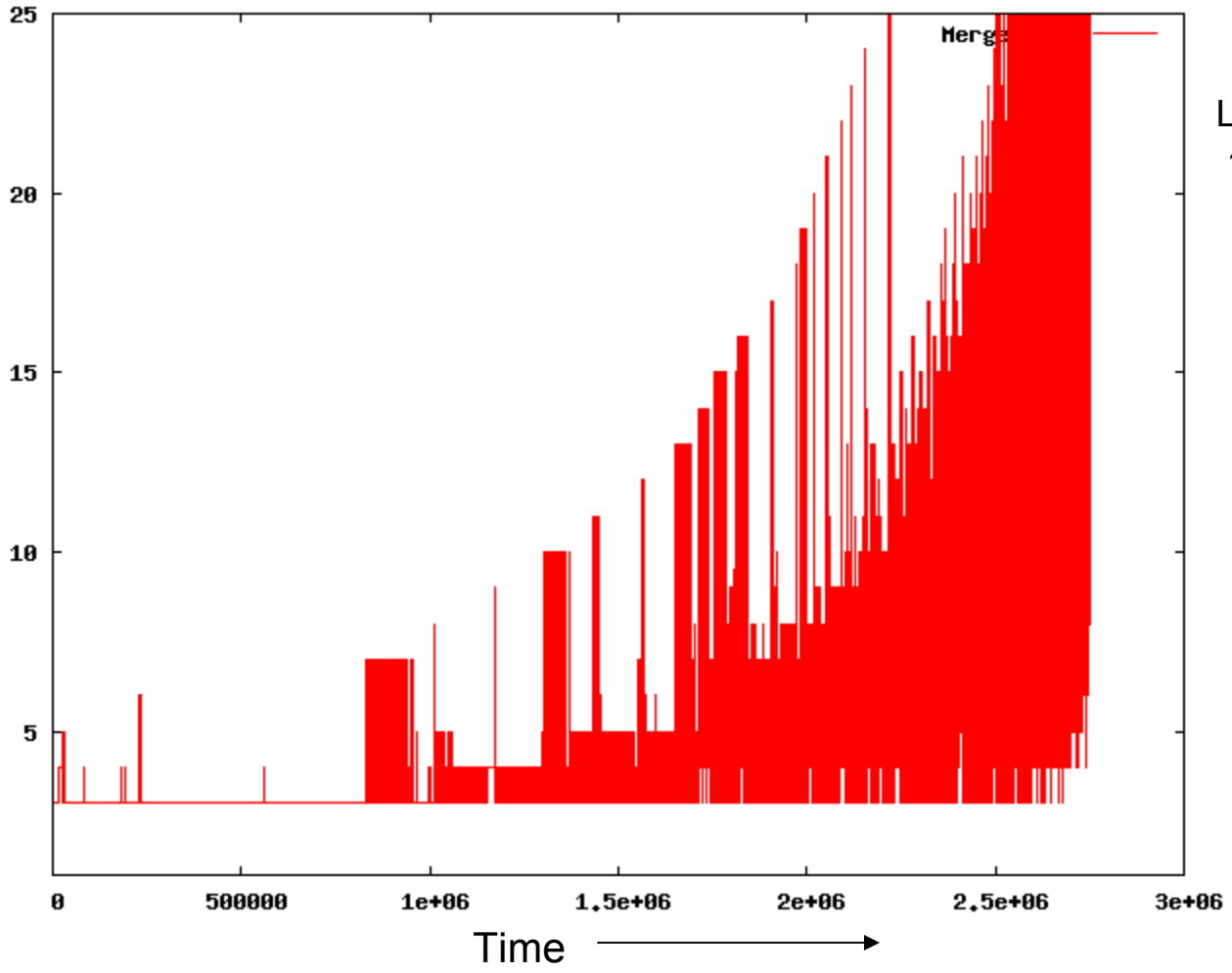
- Markowitz pivoting chooses ideals to merge in order of increasing Markowitz weight, with the weights of all ideals recalculated after every merge; we greedily choose the ideal that perturbs the current matrix the least

- A discrete priority queue data structure, keyed by Markowitz weight, allows finding the next ideal and updating all the others in O(1) time

- The merge itself uses Cavallar's spanning tree method to maximize incidental cancellations

A Markowitz Merge Movie

Merging for $6^{256} + 5^{256}$ (2.75 million merges)



Largest merge
~ 140 relsets

When Should Merging Stop?

- Merging until the size-density product stops decreasing often yields a matrix that is too large, too sparse, or both
- Instead, choose a matrix density where block Lanczos is known to converge most of the time, perform the minimum amount of merging to get a usable matrix, and finally continue merging until the estimated weight of the lightest relation-sets exceeds the safe bound
- This scheme builds a workable matrix quickly, then trades off additional matrix density for smaller matrix size in the case of significant excess relations
- Current code targets an average of 70 nonzeros per column

When Should Merging Stop?

- Ideals belong to one of three sets:
  - *Dense* ideals are not tracked by the filtering; only the count matters (there are at least *min_cycles* such ideals)
  - *Inactive* ideals are allowed to exist in the final matrix (need not be merged)
  - *Active* ideals must be merged

- A relation-set is considered complete if it has no active ideals; the matrix is considered complete if it has more than (dense ideals + inactive ideals) complete relation-sets

- Ideals can switch between the active and inactive sets; this dynamically changes the number of complete relation-sets and the target number for a complete matrix

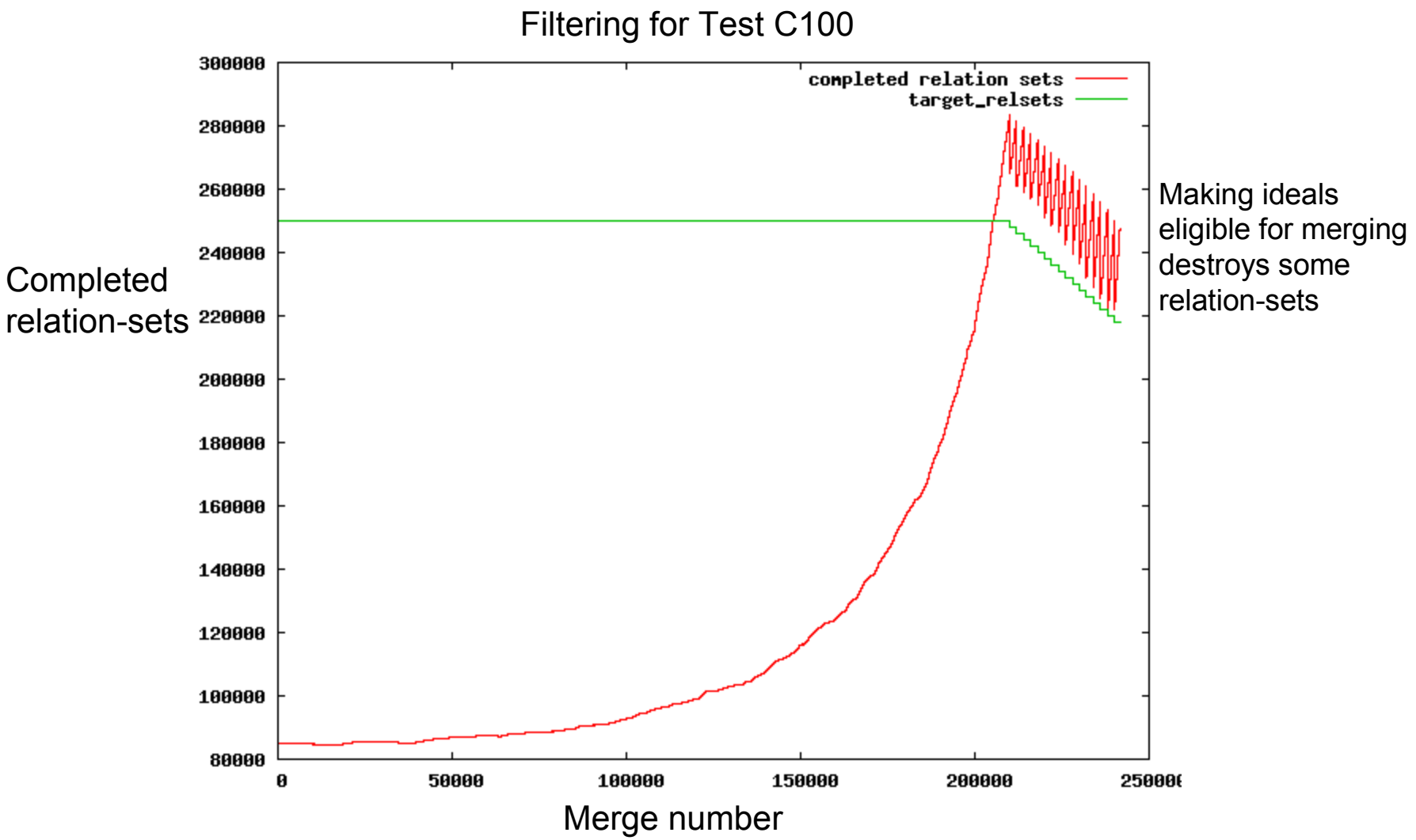
- An ideal becomes dense (i.e. is *buried*) by removing it from the ideal lists of all relation-sets

Main Merge Algorithm

Add all ideals to the *inactive heap*, keyed by Markowitz weight
Move *min_cycles* ideals with lowest Markowitz weight to the *active heap*
target_relsets = *min_cycles* + (ideals in inactive heap)

```
While (1) {
    If (active heap is empty) {
        If (average weight of lightest target_relsets completed
            relation-sets exceeds 70.0) {
                break;
        }
        bury up to 500 of the heaviest inactive ideals, update min_cycles
        move up to 2000 of the lightest inactive ideals to the active heap
        recalculate target_relsets
    }
    Merge the (one) lightest active ideal
    Update the count of completed relation-sets
    Swap ideals between the active and inactive heaps until all the
                ideals with lowest weight are active
}
```

Main Merge Algorithm



Relation-set Postprocessing

- Denny and Muller (1995) developed a method based on reducing “cycle bases” for improving the sparsity of a collection of relation-sets
- When applied as a postprocessing pass to a finished collection of relation-sets, this method reduces the number of relations appearing in the matrix by 2-10%, in about the same time needed for merging
- The method is more effective in the case of less oversieving, because it seeks to lighten relation-sets containing relations in common with other relation-sets

Reducing the Cycle Basis

Two relation-sets with relations

A – B – C A – B – D – E

have the same *cycle basis* as relation-sets

A – B – C C – D – E

except the latter is one relation smaller (and thus has fewer nonzeros, assuming no incidental cancellation).

Hence, whenever relation-sets have many relations in common we can potentially rearrange the cycle basis to one containing fewer relations. The postprocessing pass makes this systematic

Reducing the Cycle Basis

The algorithm as implemented:

- Assign a unique number to each relation
- Pretend the relation list in each relation-set is actually an ideal list
- Heapify the “ideals” based on Markowitz weight
- Perform a “merge phase”, proceeding relation by relation in order of increasing Markowitz weight. Each merge uses Cavallar’s spanning tree algorithm to turn *c* relation-sets into *c* (and not *c* - 1) possibly different relation-sets
- Stop when all relations have been processed

Why does this improve the matrix?

- There are usually fewer relations than large ideals in those relations; merging at the relation level is not fooled by untracked entries or ties in Markowitz weight at the ideal level
- The last few merges of large ideals have huge numbers of relation-sets, often too many for spanning tree methods to be used; thus the heaviest merges are often suboptimal

Comparing Implementations

Number	Code	Relations	Matrix size	Avg. Nonzeros / Column
C136	CADO	22.6M	2406529	135.6
C136	Msieve 1.36	22.6M	2489829	98.0
$2^{1826} + 1$ (C164)	Aoki et. al.	459M	7500801	167.0
$6^{383} + 1$ (C165)	Msieve 1.38	226M	7676537	99.7

(Msieve numbers include dense and quadratic character rows)

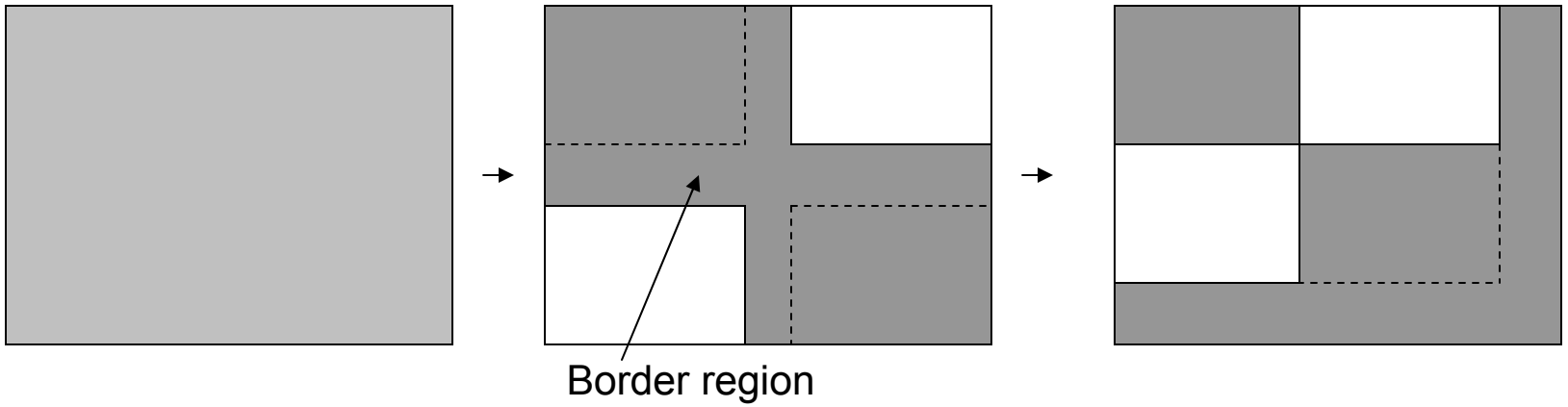
Future Work

- Scale to even larger size problems
  - Use JUDY library instead of ordinary hashtables
  - Tune the handling of data on disk
  - Handle primes > 32 bits, > 4G relations
- Try to find why generated matrix doesn’t solve sometimes
- Add more math to produce better matrices
  - How much room is there for improvement?

More Math

- Markowitz pivoting is a *greedy local* algorithm; it assumes that minimizing the current fill-in will optimize the total fill-in later
- *Global* methods rearrange the ordering of pivots so that ideals that have low weight *and low connectivity* are merged first
- There is a huge literature on matrix column ordering methods
  - Most methods are graph-based and rely on the matrix being symmetric
  - For unsymmetric matrices, global methods are pretty much limited to variants of recursive graph partitioning, with the rows and columns of the matrix represented as a bipartite graph

Graph Partitioning



- Use a graph partitioning algorithm to form two disconnected groups of nonzero matrix entries, with a (hopefully small) border region
- Renumber the rows / columns to put the disconnected regions first
- Repeat recursively for all three regions, down to some minimum size
- Sort the regions in order of increasing connectivity, then perform the regular merge phase one region at a time

Graph Partitioning

- A good partition of the matrix limits the non-local fill-in caused by merging, so local methods like Markowitz pivoting are more likely to be optimal; fill-in is deferred to neighborhoods that will be pruned anyway

- Graph partitioning is hugely important, many algorithms are available:
  - Kernighan-Lin (improvements by Fiduccia / Mattheyses)
  - Multilevel algorithms (spectral methods, recursive KLFM)

- May or may not help
  - An NFS filtering dataset makes an *extremely* large partitioning problem
  - The advanced algorithms are highly memory-intensive
  - Initial results not very encouraging; matrices are dense enough that border regions are fairly large

Conclusion

- Converting relations from NFS factoring into a matrix need not be a labor-intensive and parameter-driven process
- The amount of memory in modern machines allows treating the NFS filtering problem as a sparse Gauss elimination problem
- Standard techniques from sparse linear algebra, along with domain-specific tweaks, allow *automatically* producing much better matrices compared to the current state of the art
- What does this mean for the largest problems?
- Current release: http://www.boo.net/~jasonp/qs.html