Prime pairing in algorithms searching for smooth group order

Pavel Atnashev, George Woltman
patnashev@gmail.com, woltman@alum.mit.edu

September 20, 2021

Abstract Factorization methods like P−1, P+1, ECM have a stage which deals with primes of $a \pm b$ form, where $a + b$ and $a - b$ are processed by a single operation. Selecting a and b such that both $a + b$ and $a - b$ are prime is called 'prime pairing' and can significantly improve performance of the stage. This paper introduces new methods of pairing, which in some cases find pairs for up to 99.9% of primes in a range. A practical algorithm and its implementations are presented.

1 Introduction

The basic idea behind factorization methods like P−1, P+1, ECM is the same, and can be described as hoping that a finite abelian group specific to a factor has smooth order (an order composed of prime powers smaller than a chosen bound B). Using additive notation and $\cdot$ operator as scalar multiplication (repeated group operation), we can describe all methods with the same formulas.

A factorization attempt starts with a generator G, which is multiplied by all prime powers below B. If the group order for some factor is B-smooth, then

$$X(B) = \left(\prod_{p^k < B} p^k \right) \cdot G \equiv 0 \cdot G \pmod{factor}$$

Usually, it is trivial to recover the factor from X by GCD or impossible division. If the order is not B-smooth one can either increase B or attempt the so-called stage 2 in hope that only one prime in the group order is larger than B. Let's define B_1 and B_2 as the bounds of two-stage factorization attempt, $B_1 < B_2$. In stage 1 compute $X = X(B_1)$. In stage 2 compute $p \cdot X$ for each prime in (B_1, B_2) interval and try to recover a factor from each $p \cdot X$ value.

Observe that if $p \cdot X \equiv 0 \cdot G \pmod{factor}$ and $p = a - b$, then

$$a \cdot X = (p + b) \cdot X = p \cdot X + b \cdot X \equiv b \cdot X \pmod{factor}$$

This allows to perform stage 2 very efficiently. Instead of computing $p \cdot X$, only look for the case when it turns to $0 \cdot G$. Select constant D and let $p = iD - r$. Precompute all $r \cdot X$, and iterate through all $iD \cdot X$. For each prime p, $(i-1)D < p < iD$ test if $iD \cdot X \equiv r \cdot X \pmod{factor}$. For the purpose of this paper we call this set of primes a D-section.

In most cases it's possible to define an abs() function such that $\text{abs}(-b \cdot X) = \text{abs}(b \cdot X)$. This is inherent property of P+1 method, one of coordinates of ECM has this property too, and it can be achieved rather inexpensively in P−1 method*. As a result, $\text{abs}(iD \cdot X) \equiv \text{abs}(r \cdot X) \pmod{factor}$ when either $(iD + r) \cdot X \equiv 0 \cdot G \pmod{factor}$ or $(iD - r) \cdot X \equiv 0 \cdot G \pmod{factor}$. Testing for one tests the other too.

*Or P−1 stage 2 can be outright replaced by P+1 stage 2. [1, Section 5]

The choice of D affects the size of precomputed set $\{r \cdot X\}$, which is exactly $\varphi(D)$. If D is composed of small primes, like $D = 2 \cdot 3 \cdot 5 \cdot 7 = 210$, the precomputed set is relatively small, $\varphi(210) = 48$. We consider the application of this paper to be the factoring of big numbers (more than a million decimal digits). For such numbers the size of the precomputed set becomes an issue. Residues modulo a leading edge Mersenne prime candidate can take up to 50MB of system memory when stored in a form suitable for FFT-based multiplication. The general assumption is that due to limitations of available memory, D should be less than 1000.

If more memory is available one should consider alternative approaches to performing stage 2. So-called "FFT continuation" [2] tests all numbers in (B_1, B_2) interval (not only primes), but does it very efficiently. It was implemented in GMP-ECM [3] with great success.

2 Pairing

There are several conditions for primes p and q to be considered a pair (p, q):

$$\frac{p+q}{2} \equiv 0 \pmod{D},\ \frac{p-q}{2} \in \{r\},\ p \equiv -q \pmod{D}$$

The last condition divides primes into $\frac{1}{2}\varphi(D)$ residue classes. Pairs can be made only inside the same class. Consequently, a prime can be paired only with one prime in the previous D-section and one in the next D-section, which seriously limits the probability of successful pairing. One can increase this probability by adding $(r+D)\cdot X$, $(r+2D)\cdot X$, ..., $(r+(L-1)D)\cdot X$ to precomputed set $\{r \cdot X\}$. Doing so provides $2L$ pairing opportunities for each prime. The downside is that the size of precomputed set increases significantly.

Montgomery in [1, Section 4.2] presents an algorithm which builds pairing with D, L parameters. It's a simple greedy algorithm that iterates over an ordered set of primes and selects the smallest possible match for each prime. Despite its simplicity, the algorithm finds the best solution. Because primes are ordered and locked in their residue classes, the smallest match is always the best option.

We take Montgomery's algorithm as a starting point and present methods to improve on it.

2.1 Relocatable primes

Since the goal of stage 2 is to test whether a single prime p in (B_1, B_2) interval is the final divisor of the group order, one does not need to test exactly $p \cdot X$, the same result could be obtained by $cp \cdot X = c(p \cdot X)$. One can multiply a prime by a constant c for convenience of processing. The conditions here are that the product should be relatively prime with D and the product should not exceed B_2. We call such primes 'relocatable primes'. All primes between B_1 and $\frac{B_2}{c_0}$ can be relocated, where c_0 is the smallest constant such that $gcd(D, c_0) = 1$ and $\frac{B_2}{c_0} > B_1$. For example, when $D = 210$, $c_0 = 11$ and primes between B_1 and $\frac{B_2}{11}$ are relocatable.

Relocating primes to the higher ranges of the stage 2 interval increases the density of primes available for pairing. But that is not their most useful feature. If the ratio $\frac{B_2}{B_1}$ is large, there can be several constants satisfying condition $\frac{B_2}{c_0} < cp < B_2$. For example, when $\frac{B_2}{B_1} = 20$, one can use constants $c_0 = 11, c_1 = 13, c_2 = 17, c_3 = 19$. Instead of choosing the constant at initialization phase, add all $\{c\}p$ to the pairing algorithm. Only one of them should be paired, and when that happens the constant becomes fixed and the prime gets relocated to the position where it has a pair. All other relocation options become disabled. This approach significantly increases pairing at virtually no cost.

Another benefit of relocatable primes is that the starting point iD moves from B_1 to $\frac{B_2}{c_0}$. This decreases the amount of D-sections that need to be iterated through.

2.2 Graph matching

Since relocatable primes are not locked in residue classes and have a certain freedom in their final position, a greedy algorithm does not find the best solution anymore. While Montgomery's algorithm can still be used to initialize the pairing, a graph matching algorithm becomes necessary to find the best (or good enough) solution. The set of vertices includes all primes in $(\frac{B_2}{c_0}, B_2)$ interval as well as all relocation positions for primes in $(B_1, \frac{B_2}{c_0})$ interval. Each relocatable prime should be represented by at least one vertex $c_0 p$. The edges of the graph are all pairing opportunities.

Graph matching works by searching for augmenting paths, each of which increases an existing match by 1 pair. Care should be taken when adding a relocation position to the match. It's not a single vertex that gets matched, but an entire subgraph of vertices representing the same relocatable prime. Similarly, when searching for free vertices the entire subgraph is added to the search queue. An augmenting path can start at the smallest prime, then jump to the largest prime, then go back to the middle. This really brings out the benefits of relocatable primes.

The final choice of relocation constant is made in the end. If the subgraph was matched, the prime is relocated to the position of match, otherwise it is relocated to an arbitrary available position.

2.3 Irregular precomputation

The minimal size of precomputed set is $\frac{1}{2}\varphi(D)$, providing one pairing opportunity for each residue class. Let's call the minimal set a unit. The precomputed set in Montgomery's algorithm was composed of several such units in a sequential order: $unit, D - unit, D + unit, 2D - unit, ...$ But there's no requirement that the units should be sequential. In fact, adding irregularity benefits pairing. Consider a precomputed set composed of L units: $unit, unit + D, unit + 3D, unit + 7D, ..., unit + (2^{L-1} - 1)D$. It provides L pairing opportunities at different scales. The density of primes decreases from B_1 to B_2. The irregular precomputed set allows to find pairs at a wide range of distances.

The exact distribution of units is up to an implementation. We have obtained good results with powers of 2 for a wide range of input parameters. It is also easy to precompute. But it's not the optimal distribution in almost all cases. For each particular set of input parameters a better distribution could be found. Further research is needed to determine if there's a general method to obtain the optimal distribution.

Large distances in the precomputed set not only allow to find pairs in sparse areas, but also allow to look for pairs outside the bounds. A prime can be relocated below $\frac{B_2}{c_0}$ or above B_2 as long as its base value (iD) is inside the bounds. This is another optimization that gives some pairing opportunities at no cost.

If there's a shortage of system memory to store the full precomputed set, only a part of the last unit can be precomputed, making L a non-integer number greater than 1.

2.4 Second base

Let $A > 2$ be a prime divisor of D. Consider two representations of a prime: $p = iD \pm r$ and $p = iD + \frac{D}{A} \pm r$. The second representation doubles the amount of pairing opportunities. But more importantly, it lets the prime to break free from its residue class. In graph terms, it adds edges between residue classes in a rather unpredictable way, dramatically boosting the pairing.

For the second representation to be always possible, add values divisible by A to the precomputed set. More precisely, the size of the set becomes $\frac{1}{2}\varphi(\frac{D}{A}) \cdot A \cdot L$. For example, when $D = 210$, $A = 7$, $L = 2$, the size grows from 48 to 56.

There is some freedom in placing the second base. There are $A-1$ possible values which look equivalent in a general sense, but can produce different results in practice, especially if some of the parameters D, A, L, B_1, B_2 are fixed.

If D is not divisible by A^2, one can have $c_0 = A$ and the starting point $iD = \frac{B_2}{A}$. This further decreases the number of D-sections to process. Unfortunately, the amount of work necessary to switch from one D-section to the next doubles, because both iD and $iD + \frac{D}{A}$ need to be incremented by D. This may not be significant for P+1 method, where the increment takes only one multiplication and the boost in pairing can compensate the increased complexity. But for ECM with its complicated arithmetic the second base is never beneficial in D ranges interesting for us. It may become beneficial at large D when the amount of D-sections becomes small and the amortized cost of ECM group addition becomes negligible.

2.5 Pairing sharing

The full implementation of all proposed methods can make the setup phase of a factorization run quite slow. Luckily, the pairing produced is not bound to a specific number being factored. It can be stored and reused with the same parameters. The main parameters are $B_1 < 10^6$ and $B_2 < 10^8$ with only the first 2-3 digits being significant. The other important parameter is the amount of memory available at the host running the factorization. This parameter is also quantized and limited. A pairing suitable for these three parameters can be stored and retrieved from a central repository. Even if it is not there, the host can compute it and upload for all others to use. Pairings in the repository can even be upgraded in the future with new methods and optimizations.

3 Algorithm

The first thing to note about the graph is that it isn't bipartite. The classical graph matching algorithm won't find the best solution. Micali–Vazirani [4] algorithm finds the best solution, but its complexity may not be necessary in practice. In practical applications the pairing reaches 95% with 100 precomputed values, and exceeds 99% with 500 precomputed values. Sacrificing a percent of pairs with a "good-enough" algorithm may be preferrable.

The algorithm we're presenting is based on a simple BFS bipartite matching algorithm. We're using it to demonstrate how the proposed methods of pairing can be implemented. Readers are free to adapt the same ideas to more specialized matching algorithms, or use optimization tweaks that decrease matching but increase performance, if that is their goal.

The algorithm itself is not sensitive to the exact distances between primes. A lot of experiments can be done on the distance list. It is only used to initialize adjacency lists. On the other hand, method-specific algorithms that generate the precomputed set can be extremely sensitive to the form and distribution of distances. A gain in pairing could be lost in increased complexity of precomputation.

A subgraph representing relocatable prime is treated as a single vertex in many cases. It can have only one match, and the exact position that is matched is the one where the prime will be relocated. It also tracks the position from which it was entered in the search, so the augmenting path can be easily back traced. But when it is entered from its match, all of the individual vertices of the subgraph get added to the search queue. And since potential relocation positions can be spread throughout the graph, the scope of the search gets greatly enhanced and the size of the queue can become comparable to the size of the graph.

But first, compute the basic precomputed unit.

Algorithm 1: One unit

Output: A unit of numbers relatively prime to D/A.

```
unit ← ∅;
for i ← 1 to D/2 do
    if gcd(D/A, i) = 1 then
        unit ← unit ∪ {i};
    end
end
```

Now compute all the distances in the precomputed set.

Algorithm 2: All distances

Output: All possible distances in the precomputed set.

```
distribution ← ∅ // We're using powers of two here, which may not be the best.
for i ← 0 to L − 1 do distribution ← distribution ∪ {2^i − 1};
distances ← ∅;
foreach d ∈ distribution do
    foreach u ∈ unit do
        distances ← distances ∪ {u + d · D} ∪ {−u − d · D};
    end
end
```

Each residue can have a pair only at a distance with a base in the middle.

Algorithm 3: Residue distances

Output: Possible distances for each residue.

```
for i ← 0 to D − 1 do
    residue_distance(i) ← ∅;
    if gcd(D/A, i) = 1 then
        foreach d ∈ distances do // the second base here can have a different value
            if i + d ≡ 0 (mod D) or i + d ≡ D/A (mod D) then
                residue_distance(i) ← residue_distance(i) ∪ {2d};
            end
        end
    end
end
```

Note that the distance between any two primes is almost always even.

Now compute relocation constants and initialize relocation subgraphs.

Algorithm 4: Relocations

Output: All possible relocations(), source() for each relocated value and a default relocated() map.

multipliers $\leftarrow \varnothing$;
for $i \leftarrow 3$ **to** B_2/B_1 **do**
    if $gcd(D/A, i) = 1$ **then**
        multipliers $\leftarrow$ multipliers $\cup \{i\}$;
    end
end
foreach $p \in prime\_range(B_1, B_2/multipliers[0])$ **do**
    relocations(p) $\leftarrow \varnothing$;
    foreach $m \in multipliers$ **do**
        if $p \cdot m \geq B_2/multipliers[0]$ *and* $p \cdot m \leq B_2$ **then**
            relocations(p) $\leftarrow$ relocations(p) $\cup \{p \cdot m\}$;
            source($p \cdot m$) $\leftarrow p$;
            relocated(p) $\leftarrow p \cdot m$;
        end
    end
end

Note that there are at most L edges per base for each prime. It allows to store adjacency efficiently.

Algorithm 5: Adjacency

Output: Graph V and its adjacency().

$V \leftarrow$ relocations(prime_range(B_1, B_2/multipliers[0])) $\cup$ prime_range(B_2/multipliers[0], B_2);
foreach $p \in V$ **do**
    adjacency(p) $\leftarrow \varnothing$;
    foreach $d \in residue\_distance(p \pmod D)$ **do**
        $q \leftarrow p + d$;
        if $q \notin V$ **then continue**;
        if $source(q) \neq nil$ *and* $source(p) = source(q)$ **then continue**;
        adjacency(p) $\leftarrow$ adjacency(p) $\cup$ $\{q\}$;
    end
end

The next algorithm is equivalent to Montgomery's algorithm if both V and adjacency() are sorted.

Algorithm 6: Initializing

```
   Output: Some match().
 1 foreach p ∈ V do
 2     if match(p) ≠ nil or (source(p) ≠ nil and match(source(p)) ≠ nil) then continue;
 3     foreach q ∈ adjacency(p) do
 4         if (source(q) = nil and match(q) = nil) or (source(q) ≠ nil and match(source(q)) = nil)
            then
 5             if source(p) ≠ nil then
 6                 relocated(source(p)) ← p;
 7                 match(source(p)) ← q;
 8             else
 9                 match(p) ← q;
10             end
11             if source(q) ≠ nil then
12                 relocated(source(q)) ← q;
13                 match(source(q)) ← p;
14             else
15                 match(q) ← p;
16             end
17             break;
18         end
19     end
20 end
```

After match() is initialized, run graph matching algorithm to find augmenting paths.

Algorithm 7: Matching

Output: Improved match(), a map of relocated() primes.

```
1  restart:
2  begin
3      link ← ∅;
4      foreach p ∈ V do
5          if match(p) ≠ nil or (source(p) ≠ nil and match(source(p)) ≠ nil) then continue;
6          queue ← ∅;
7          enqueue(p);
8          link(p) ← nil;
9          if source(p) ≠ nil then link(source(p)) ← p;
10         while queue ≠ ∅ do
11             front ← dequeue();
12             foreach q ∈ adjacency(front) do
13                 if link(q) ≠ nil or (source(q) ≠ nil and link(source(q)) ≠ nil) then continue;
14                 p ← front;
15                 if (source(q) = nil and match(q) = nil) or
                      (source(q) ≠ nil and match(source(q)) = nil) then
16                     augment(match, link, p, q);
17                     goto restart;
18                 else
19                     link(q) ← p;
20                     if source(q) ≠ nil then
21                         link(source(q)) ← q;
22                         q ← relocated(source(q));
23                         p ← match(source(q));
24                     else
25                         p ← match(q);
26                     end
27                     link(p) ← q;
28                     if source(p) ≠ nil then
29                         link(source(p)) ← p;
30                         foreach r ∈ relocations(source(p)) do enqueue(r);
31                     else
32                         enqueue(p);
33                     end
34                 end
35             end
36         end
37     end
38 end
```

After an augmenting path is found, matching is flipped with extra care for relocated primes.

Algorithm 8: Augmenting

```
1  function augment(match, link, p, q)
2  begin
3      while true do
4          if source(q) ≠ nil then
5              relocated(source(q)) ← q;
6              match(source(q)) ← p;
7          else
8              match(q) ← p;
9          end
10         if source(p) ≠ nil then
11             relocated(source(p)) ← p;
12             match(source(p)) ← q;
13         else
14             match(p) ← q;
15         end
16         if source(p) ≠ nil then p ← link(source(p));
17         if link(p) = nil then
18             return
19         end
20         q ← link(p);
21         if source(q) ≠ nil then q ← link(source(q));
22         p ← link(q);
23     end
24 end
```

3.1 Implementations

The algorithm was implemented in Prefactor (`https://github.com/patnashev/prefactor`) and Prime95 (`https://www.mersenne.org/download/`) programs.

Prime95 implementation uses a "windowed mode", which searches for augmenting paths only in small subsets of the graph to save time. Prime95 mostly runs tests on Mersenne numbers at high B_1, B_2 values, and an additional 1-2% increase in pairing is not worth the extra setup time. The implementation also uses only one base.

Table 1: Pairings for a selected set of input parameters.

B_1	B_2	#primes	D	A	L	#{r}	Pairs		
							Montgomery	Prefactor	Prime95
10 000	1 000 000	77 269	210	1	8	192	32 121 83.1%	38 370 99.3%	38 206 98.9%
10 000	1 000 000	77 269	714	3	5	720	-	38 577 99.9%	-
700 000	23 100 000	1 397 601	84	1	10	120	535 912 76.7%	677 422 96.9%	664 321 95.1%
700 000	23 100 000	1 397 601	210	1	5	120	465 585 66.6%	537 185 76.9%	536 248 76.7%
700 000	23 100 000	1 397 601	210	7	5	140	-	657 563 94.1%	-
700 000	23 100 000	1 397 601	210	1	10	240	566 578 81.1%	680 178 97.3%	671 159 96.0%
700 000	23 100 000	1 397 601	510	3	6	576	-	689 404 98.7%	-

When reading Table 1 take into account that larger D is good, because it decreases the amount of D-sections, but is bad because it increases the size of precomputed set. To determine what parameters are optimal, one needs a cost function specific to the exact factorization method used.

References

[1] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation, 48*, pages 243–264, 1987. https://doi.org/10.1090/S0025-5718-1987-0866113-7.

[2] Peter L. Montgomery and Robert D. Silverman. An FFT extension to the $P - 1$ factoring algorithm. *Mathematics of Computation, 54*, pages 839–854, 1990. https://doi.org/10.1090/S0025-5718-1990-1011444-3.

[3] Paul Zimmermann and Bruce Dodson. 20 years of ECM. *Algorithmic Number Theory*, pages 525–542, 2006. https://doi.org/10.1007/11792086_37.

[4] Silvio Micali and Vijay V. Vazirani. An O($\sqrt{|V|} \cdot |E|$) algoithm for finding maximum matching in general graphs. *21st Annual Symposium on Foundations of Computer Science (SFCS 1980)*, pages 17–27, 1980. https://doi.org/10.1109/SFCS.1980.12.