Motivation

▶ "Quantum factorization of 143"*
▶ "That quantum computation, which used only 4 qubits [...] actually also factored [...] 56153, without the awareness of the authors"†
▶ 291311‡
▶ $1099551473989 = 1048589 \times 1048601$
▶ Variational Quantum Factoring [aoac18]
▶ Adiabatic Factoring
▶ Pretending to factor large numbers on a quantum computer‖

---

*https://doi.org/10.1103/PhysRevLett.108.130501
†https://arxiv.org/abs/1411.6758
‡https://arxiv.org/abs/1706.08061
§https://en.wikipedia.org/wiki/Integer_factorization_records
¶https://bit.ly/3iQDhmT
‖https://arxiv.org/pdf/1301.7007.pdf
Introduction: Factoring

- Factoring is in \( \text{NP} \cap \text{coNP} \).
- Widely believed not to be \( \text{NP} \)-complete.
- Shor’s quantum algorithm: \( \text{polylog}(N) \).
- Many classical sub-exponential algorithms

For real-world security: we care about real-world runtime.

---

Direct approach

**Adiabatic factoring**

\[ N = 143_{10} = 10011111_2 = pq \]

| Modifier | \( p_1 \) | \( p_2 \) | \( p_3 \) |
|----------|----------|----------|----------|

| Carry | \( c_{12} \) | \( c_{23} \) | \( c_{32} \) |
|-------|----------|----------|----------|

| Product | \( p_1 \) | \( p_2 \) | \( p_3 \) |
|---------|----------|----------|----------|

\[ p_1 + q_1 = 1 + 2c_{12} \]
\[ p_1q_1 + q_2 = 1 + 2c_{23} \]
\[ \ldots \]

Reduce to quadratic unconstrained binary optimization (QUBO):

\[ H_0 = (p_1 + q_1 - 2c_{12})^2 \]
\[ H_2 = (p_1q_1 + q_2 - z_{12})^2 \]
\[ \ldots \]

Remove high-order (> 2) terms using additional variables, eg:

\[ p_1q_1q_2 \]

Optimize binary variables to minimize sum of squares:

\[ \min \sum_i H_i = 0 \]

---

Adiabatic factoring (cont.)

Encode each variable in a qubit of a quantum annealer.

Then the objective function describes \( H_P \), the problem Hamiltonian.

Run the adiabatic algorithm.
Adiabatic factoring (cont.)

- QUBO is NP-complete
- "Relative to a permutation oracle [...] the class NP cannot be solved on a quantum Turing machine in time $2^{0.495n - 16.4}$" [bbv96]
- Evidence (no proof!) that the adiabatic algorithm cannot solve QUBO efficiently.
- Maybe an annealer can achieve the full square-root speedup.
- In practice asymptotics may not be meaningful. Look at SAT-solvers: they solve NP-hard problems for sizes relevant in the real world!

SAT factoring

Wires are Boolean variables; gates are clauses.

Negation:

NAND:

Half-adder:

SAT factoring: bias (cont.)

Multiplication algorithm

- Schoolbook multiplication
  - Asymptotically suboptimal
- Karatsuba
  - Asymptotically faster
  - SAT solver is slower in practice
- Others (assumed slower)

Certain semi-primes are easier to solve

- Assume the solver is able to pick up on this
- Also tried "factor any" circuit
  - One multiplication circuit
  - Multiple possible outputs (combined in one large OR gate)
  - Runtime is worse

Results

MapleCOMSPS mean runtime
We could find no patterns in the "easy" primes.

S. R. Verschoor  Factoring with (quantum) SAT solvers  2020–08–19  19/40

Things to look out for in the adiabatic factoring literature:

▶ Not mentioning any asymptotics
▶ Not counting preprocessing in total runtime
▶ Showing only efficiency or effectiveness of preprocessing
▶ Extrapolating small-scale results as evidence of asymptotics

S. R. Verschoor  Factoring with (quantum) SAT solvers  2020–08–19  22/40

Fermat’s factorization method (ignoring trivial factors):

\[ N = a^2 - b^2 = (a + b)(a - b) = pq \]

Congruence of squares is sufficient:

\[ a^2 \equiv b^2 \pmod{N} \]

and \( q = \gcd(a - b, N) \).

Try \( b_i \equiv a_i b_i \pmod{N} \) for many \( a_i \), until you find \( \{b_i\} \subset \{b\} \) such that \( \prod b_i - b_j \) (and \( \prod a^2 - b^2 \)) is a perfect square.

S. R. Verschoor  Factoring with (quantum) SAT solvers  2020–08–19  24/40
Brief overview of the NFS

To find \( (b_i) \), factorize each \( b_i = \prod_k \alpha_k^{e_k} \) into its prime factors:

\[
\prod_j b_j = \prod_j \prod_k \alpha_k^{e_k} = \prod_k \sum_i \alpha_i^{e_i} 
\]

which is a square if \( \sum_i \alpha_i \) is even for all \( k \).

To keep \( k \) small, only consider \( y \)-smooth numbers \( (\alpha_i \leq y) \) for all \( k \), discard non-smooth numbers.

This also allows faster factoring of \( b_i \) using trial division, the elliptic curve method and/or sieving.

Tune the parameters so we can factor with overwhelming probability.

Let

\[
g \left( \sum_{j \neq \alpha} \phi_j \right) = \left( \sum_{j \neq \alpha} \phi_j \right) + \phi_\alpha 
\]

\[L = \text{smooth numbers on the algebraic side using the norm map}
\]

\[
g: \mathbb{Z} \to \mathbb{Z} 
\]

\[
(a, b) = (a + bm)g(a, b) 
\]

Both the rational and algebraic side are smooth iff

\[
F(a, b) = (a + bm)g(a, b) 
\]

\[
U = \{ (a, b) \mid a, b \in \mathbb{Z}, |a| \leq u, 0 < b \leq u \}
\]

Search for \( (a, b) \in U \) such that \( F(a, b) \) is \( y \)-smooth.


Brief overview of the NFS

Find \( y \)-smooth numbers on the algebraic side using the norm map

\[
g(a, b) = -(b) \frac{f'}{f} 
\]

Both the rational and algebraic side are smooth iff

\[
F(a, b) = (a + bm)g(a, b) 
\]

\[
U = \{ (a, b) \mid a, b \in \mathbb{Z}, |a| \leq u, 0 < b \leq u \}
\]

Search for \( (a, b) \in U \) such that \( F(a, b) \) is \( y \)-smooth.


Brief overview of the NFS

Tune the parameters so we can factor with overwhelming probability.

Let

\[
y \in L^{2\epsilon}(\beta) 
\]

\[
u \in L^{\epsilon}(\beta) 
\]

\[
d \in \beta + o(1)(\log N)^{1/3}(\log \log N)^{2/3} 
\]

\[
U \text{ has size } 2^{u+o(1)} \sim L^\epsilon(\beta) 
\]

By the Prime Number Theorem there are

\[
\pi(y) \approx y/\log(y) = \gamma^2(\epsilon)(y) 
\]

primes \( \leq y \). Assume numbers \( F(a, b) \) have the same probability of being prime.

The search range needs to contain \( y^{2+o(1)} = L^{2\epsilon+o(1)} \cdot y \)-smooth \( F(a, b) \).


Brief overview of the NFS

Write \( N \) in base \( m \):

\[
N = m^a + c_{-1}m^{a-1} + \cdots + c_1m + c_0 
\]

Define

\[
f = X^e + c_{-1}X^{e-1} + \cdots + c_1X + c_0 \in \mathbb{Z}[X] 
\]

with root \( \alpha \).

Search for \( S \) such that the following are squares:

\[
\prod_a \left( (a + bm) - X^2 \in \mathbb{Z} \right) 
\]

\[
f'\left( \alpha \right)^2 \prod_f \left( a + bm - \beta^2 \in \alpha \right) 
\]

Let \( \delta \) be the ring homomorphism \( \sum \alpha \delta^1 \mapsto \sum \alpha \cdot m_i \); then we can factor \( N \) as

\[
gcd(\delta(\alpha) - f'(\alpha)X, N) 
\]


Brief overview of the NFS

(Conjectured) complexity

\[
L = \frac{1}{2} \left( \sqrt{\frac{\delta}{\gamma}} + o(1) \right) \exp \left( \left( \sqrt{\frac{\delta}{\gamma}} + o(1) \right) (\log N)^{1/3}(\log \log N)^{2/3} \right) 
\]

This is approximately \( L^{1.925+o(1)} \), where \( L = L_0[1/3, 1] \).

Search \( U \) with Grover's algorithm

\[
L^{1.925+o(1)} \approx L^{1.857} \text{ (for Shor: } \log(2)^2+o(1)) 
\]

\[
\text{Quantumly } \\
\text{Search each with Grover in time } L^{1.925+o(1)} 
\]

\[
\text{Optimize } \delta 
\]

\[
\text{... } \approx L^{1.925+o(1)} 
\]
Finding smooth numbers with Circuit-SAT

Given a Boolean circuit with \( v \) input variables and \( g \) gates: find an assignment to \( v \) such that the circuit outputs TRUE.

Brute-force the solution:

\[ \text{for all } 2^v \text{ possible inputs:} \]

\[ \text{"run the circuit" in time } O(g) \]

Total runtime: \( O(2^v g) \)

Finding smooth numbers with Circuit-SAT (cont.)

Size of \((e_1, \ldots, e_{\pi(y)})\) is lower-bounded by \( \pi(y) \in L^{1.387} \), thus runtime \( L^{1.387} \).

However, \( F(a, b) > N \) for \( N < 2^{140} \)

Finding smooth numbers with Circuit-SAT

Allow non-prime factors \( q \leq y \):

\[ a \longrightarrow (a, b) \longrightarrow \cdots \longrightarrow q_1 q_2 \cdots q_n \]

Require all \( q_i \leq y \) via input encoding.

Note: \( \log F(a, b) = \log(N)^{2/3} \), thus runtime \( L^{2/3} \).

Finding smooth numbers with Circuit-SAT

Direct implementation of smoothness definition:

\[ F(a, b) = \prod_{i=1}^{\pi(y)} p_i^{e_i} \]

with hardcoded primes.

Finding smooth numbers with Circuit-SAT

But maybe it works in practice?

Assume \( o(1) = 0 \) for circuit generation.

Preferable: no input besides \((a, b)\)

Idea: Derandomize Lenstra’s elliptic curve method for factorization (ECM) by fixing randomness at time of circuit generation:

ECM:

1. While \( N < y \):
2. \( p \leftarrow \text{ECM}(N) \)
3. While \( p | N \):
4. \( N \leftarrow N / p \)

Require all \( q_i \leq y \) via input encoding.

Note: \( \log F(a, b) = \log(N)^{2/3} \), thus runtime \( L^{2/3} \).
Finding smooth numbers with Circuit-SAT (cont.)

Step 2 finds a non-trivial factor with probability $\Omega(1 - 1/e)$ over the random choice of the elliptic curve. Runtime is dominated by runtime of a single ECM iteration 

$$K(p) \in L_p[1/2, \sqrt{2} + o(1)]$$

Since $p \leq y \in L_p[1/3, \cdot]$, we get a circuit of size $L_p[1/6, \cdot]$. Repeat this $\text{polylog}(Y)$ times until you factor with probability $1 - o(1)$.

Input-space $2^x \in L_p[1/3, 2x - \beta + o(1)]$

Circuit size $g \in L_p[1/6, \cdot]$

Assuming a quadratic speedup, we have quantum runtime $L_N[1/3, \cdot - 2 - \beta + o(1)]$

Assuming a quadratic speedup, we have quantum runtime $L_N[1/6, \cdot]$

Conclusions/Future work

A quadratic speedup in SAT-solving is (still) insufficient to speed up factoring.

I would like to try SMT solvers, although I expect similar results.

Open question: how to implement this on a quantum annealer?

► A polylog($Y$) sized circuit for smoothness testing?

► Translate the brute-force strategy to QUBO?

Problem

The quadratic speedup over our brute-force Circuit-SAT solver suffices: $\Omega(\sqrt{2})$

However, when we say Circuit-SAT is NP-complete, we measure complexity in $g$

► Best theoretical runtime: $O(2^{2.4058g})$

► The standard translations to SAT/QUBO are polynomial in $g$

► So we expect a solver runtime exponential in $g$: $2^{\alpha g^{1/6}}$

► To beat the NFS this solver requires a superpolynomial speedup.

Thank you

Conclusions/Future work

A quadratic speedup in SAT-solving is (still) insufficient to speed up factoring.

I would like to try SMT solvers, although I expect similar results.

Open question: how to implement this on a quantum annealer?

► A polylog($y$) sized circuit for smoothness testing?

► Translate the brute-force strategy to QUBO?

References

SAT: bits

A bit is either a value (True/False) or a SAT-variable:

```haskell
data Bit = Val {getVal :: Bool} |
          Var {getVar :: Int}
```

This allows:

► Rapid prototyping

► Mixed input (number padding)

► Proving gate correctness (exhaustive testing)

► Randomized testing of arbitrary sized gates
SAT: gates

\[
\text{neg} \colon \text{Bit} \to \text{Bit} \\
\text{neg} \ (\text{Val} \ b) = \text{Val} \ (\text{not} \ b) \\
\text{neg} \ (\text{Var} \ v) = \text{Var} \ (\neg v)
\]

\[
\text{nandGate} \colon \text{Bit} \to \text{Bit} \to \text{SymEval} \text{Bit} \\
\text{nandGate} \ x \ y = \text{do} \\
\quad z \leftarrow \text{nextVar} \\
\quad \text{addClauses} \ [\ [x, z], [y, z], [\text{neg} \ x, \text{neg} \ y, \text{neg} \ z] ] \\
\quad \text{return} \ z
\]

\[
\text{halfAdd} \colon \text{Bit} \to \text{Bit} \to \text{SymEval} \text{(Bit, Bit)} \\
\text{halfAdd} \ x \ y = \text{do} \\
\quad a \leftarrow \text{xorGate} \ x \ y \\
\quad c \leftarrow \text{andGate} \ x \ y \\
\quad \text{return} \ (a, c)
\]

Finding smooth numbers with Circuit-SAT: gates

Optimize gates beyond 3-SAT

\[
\text{fullAdd} \ x \ y \ c i = \text{do} \\
\quad s o \leftarrow \text{nextVar} \\
\quad \text{addClauses} \ [\ [\text{neg} \ x, \text{neg} \ y, \text{neg} \ c i, \text{so}], [\text{neg} \ x, \text{neg} \ y, \text{ci}, \text{neg} \ so], [\text{neg} \ x, \text{neg} \ y, \text{neg} \ ci, \text{so}], [\text{neg} \ x, \text{neg} \ y, \text{ci}, \text{so}], [\text{neg} \ x, \text{neg} \ ci, \text{so}], [\text{neg} \ x, \text{ci}, \text{so}], [\text{neg} \ y, \text{neg} \ ci, \text{so}], [\text{neg} \ y, \text{ci}, \text{so}], [\text{neg} \ ci, \text{so}], [\text{ci}, \text{neg} \ so], [s o, \text{neg} \ ci], [s o, \text{ci}], [\text{so}], [\text{so}], [\text{ci}], [\text{ci}], [\text{ci}], [\text{ci}]] \\
\quad c o \leftarrow \text{nextVar} \\
\quad \text{addClauses} \ [\ [\text{neg} \ x, \text{neg} \ y, \text{co}], [\text{neg} \ x, \text{neg} \ ci, \text{co}], [\text{neg} \ y, \text{neg} \ ci, \text{co}], [\text{neg} \ y, \text{ci}, \text{co}], [\text{neg} \ ci, \text{co}], [\text{ci}, \text{neg} \ co], [\text{ci}], [\text{ci}], [\text{co}], [\text{co}], [\text{co}]] \\
\quad \text{return} \ (s o, c o)
\]