Factoring semi-primes with (quantum) SAT solvers

Sebastian Verschoor
Joint work with Michele Mosca and João Marcos Vensi Basso

Institute for Quantum Computing
David R. Cheriton School of Computer Science
University of Waterloo

August 19th, 2020

IQC Institute for Quantum Computing — University of Waterloo

Outline

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 * 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: Quantum optimizers

Basic idea of adiabatic quantum computing (AQC):
- Initialize state according to ground state of H_I (some "easy" Hamiltonian)
- Let H_P be the problem Hamiltonian where the ground-state describes the goal state
- Slowly evolve H_I to H_P (so system stays in ground state)
  - runtime bounded by $T = O(1/g_{min}^2)$
  - g_{min} is the spectral gap of $H(t) = (1 - t/T)H_I + (t/T)H_P$
- Measure solution from endstate

Introduction: Quantum optimizers (cont.)

Interesting because:
- AQC is (polynomially) equivalent to the quantum gate model [**adkllr04**]
- Quantum Annealing (QA): a noisy version of AQC
- Exists now: D-Wave
- We can use it to solve an NP-complete problem:
  - quadratic unconstrained binary optimization (QUBO)
  - thus we can solve any problem in NP
  - but maybe not efficiently

Introduction: SAT

The Boolean satisfiabilty problem (SAT) is the canonical NP-complete problem. Example:

$$(x_1 \vee \neg x_2) \wedge (x_3 \vee (x_4 \wedge x_5))$$

is satisfied by $x_2 \leftarrow$ FALSE and $x_3 \leftarrow$ TRUE.
- NP-complete: polynomially equivalent to all NP-complete problems
- No efficient algorithm for solving it
- Used a lot in practice to solve large NP-hard problems
  - CNF-SAT, eg: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_4) \wedge (x_3 \vee x_5)$

Introduction: Factoring

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

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

Introduction

The (Rivest-Shamir-Adleman) RSA cryptosystem is based on the hardness of factoring large integers.

- Given $N = pq$, it is hard to find p and q

RSA challenge: published March 18, 1991

- RSA-100, 330 bits: factored April 1, 1991
  - $<$ 5 hours on a single core of my laptop
- RSA-250, 829 bits: factored February 28, 2020
- RSA-2048, unbroken

Use this as benchmark, but for much smaller numbers

Direct approach

arXiv.org > cs > arXiv:1902.01448

Computer Science > Cryptography and Security

[Submitted on 4 Feb 2019 (v1), last revised 23 Oct 2019 (this version, v2)]

Factoring semi-primes with (quantum) SAT-solvers

Michele Mosca, Sebastian R. Verschoor

Adiabatic factoring

$N = 143_{10} = 10001111_2 = pq$

Multiplier						1	p_2	p_1	1
					*	1	q_2	q_1	1
Binary-multiplication						1	p_2	p_1	1
					q_1	p_2q_1	p_1q_1	q_1	
				q_2	p_2q_2	p_1q_2	q_2		
			1	p_2	p_1	1			
Carry		z_{67}	z_{56}	z_{45}	z_{34}	z_{23}	z_{12}		
	+	z_{57}	z_{46}	z_{35}	z_{24}				
Product		1	0	0	0	1	1	1	1

- $p_1 + q_1 = 1 + 2z_{12}$
- $p_1q_1 + q_2 + z_{12} = 1 + 2z_{23}$
- ...

Adiabatic factoring (cont.)

Reduce to quadratic unconstrained binary optimization (QUBO):

- $H_1 = (p_1 + q_1 - 1 - 2z_{12})^2$
- $H_2 = (p_1q_1 + q_2 + z_{12} - 1 - 2z_{23})^2$
- ...

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

- $p_1q_1q_2 \xrightarrow{t=p_1q_1} tq_2 + 2(p_1q_1 - 2p_1t - 2q_1t + 3t)$

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 $o(2^{n/2})$" **[bbbv96]**
- 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!

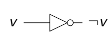
SAT factoring

Main idea:
- Measure performance of the best classical NP-complete algorithms, ie. SAT solvers
- Assume we achieve the square-root speedup over the entire computation (every clockcycle)
- Compare the performance against classical methods
  - asymptotically
  - for real-world instances

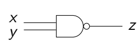
SAT factoring

Wires are Boolean variables; gates are clauses

Negation:

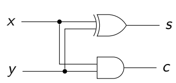


NAND:



$$(x \lor z) \land (y \lor z) \land (\neg x \lor \neg y \lor \neg z)$$

Half-adder:



$$(s \lor x \lor \neg y) \land (s \lor \neg x \lor y) \land (\neg s \lor x \lor y) \land (\neg s \lor \neg x \lor \neg y)$$
$$\land (c \lor \neg x) \land (c \lor \neg y) \land (\neg c \lor x \lor y)$$

SAT factoring: bias

We skewed the methods to maximize classical performance.
Solver: **MapleComSPS [maplecomsps]**:
- DPLL-based SAT solver
  - Backtracking
  - Unit propagation, pure literal elimination, clause learning
- Fastest solver in the 2016 SAT competition

Cryptominisat5
- slower

Local search algorithms like WalkSAT:
- Initialize variables random
- While ∃ UNSAT(clause): flip a variable in UNSAT(clause)
- Structurally closer to annealing
- Performs worse in practice.

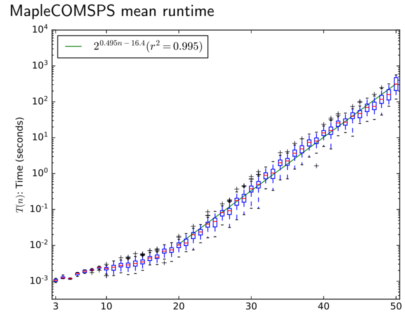
SAT factoring: bias (cont.)

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

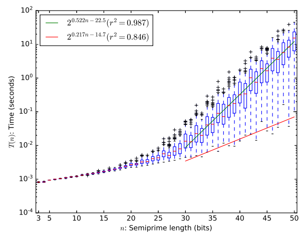
Results

MapleCOMSPS mean runtime



Results

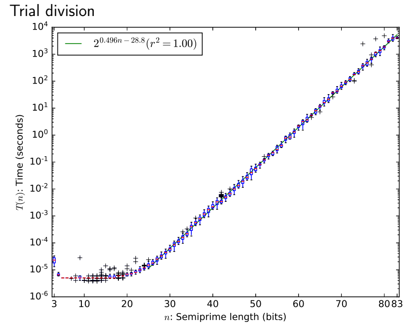
MapleCOMSPS min runtime



We could find no patterns in the "easy" primes

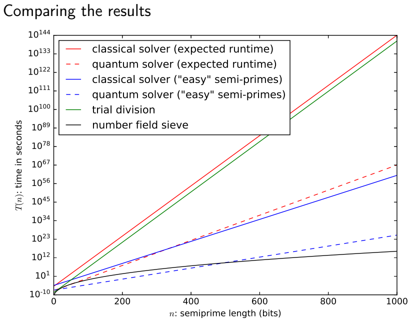
Results

Trial division



Results

Comparing the results



Direct approach

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

Speeding up the number field sieve

arXiv.org > cs > arXiv:1910.09592

Computer Science > Cryptography and Security

[Submitted on 21 Oct 2019]

On speeding up factoring with quantum SAT solvers

Michele Mosca, João Marcos Vensi Basso, Sebastian R. Verschoor

Soon to appear in Nature Scientific Reports

Basics of factoring

Fermat's factorization method (ignoring trivial factors):

$$N = a^2 - b^2 = (a+b)(a-b) = pq$$

Congruence of squares is sufficient:

$$a^2 \equiv b^2 \Rightarrow (a+b)(a-b) \equiv 0 \mod N$$

and $q = \gcd(a-b, N)$.
Try $b_i = a_i^2 \mod N$ for many a_i, until you find $\{b_j\} \subseteq \{b_i\}$ such that $\prod b_j = b^2$ (and $\prod a_j^2 = a^2$) is a perfect square.

Basics of factoring (cont.)

To find $\{b_j\}$, factorize each $b_i = \prod_k p_k^{e_{i,k}}$ into its prime factors:

$$\prod_j b_j = \prod_{j,k} p_k^{e_{j,k}} = \prod_k p_k^{\sum_j e_{j,k}}$$

which is a square if $\sum_j e_{j,k}$ is even for all k.
Store the exponent vectors modulo 2, look for a linear dependency (need $k + o(1)$ vectors).
To keep k small, only consider y-smooth numbers ($p_k \leq y$ 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.

Brief overview of the NFS**

Write N in base m: $N = m^d + c_{d-1}m^{d-1} + \cdots + c_1 m + c_0$. Define

$$f = X^d + c_{d-1}X^{d-1} + \cdots + c_1 X + c_0 \in \mathbb{Z}[X]$$

with root α.
Search for S such that the following are squares:

$$\prod_{(a,b)\in S} (a + bm) = X^2 \in \mathbb{Z}$$

$$f'(\alpha)^2 \prod_{(a,b)\in S} (a + b\alpha) = \beta^2 \in \mathbb{Z}[\alpha]$$

Let ϕ be the ring homomorphism $\sum_i a_i\alpha^i \mapsto \sum_i a_i m^i$, then we can factor N as

$$\gcd(\phi(\beta) - f'(m)X, N)$$

following the description of [bbm17**]

Brief overview of the NFS

Find y-smooth numbers on the algebraic side using the norm map g from $\mathbb{Z}[\alpha]$ to $\mathbb{Z}$:

$$g(a, b) = (-b)^d f(-a/b)$$

Both the rational and algebraic side are smooth iff $F(a, b) = (a + bm)g(a, b)$ is smooth.

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

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

Brief overview of the NFS

(Conjectured) complexity

$$L_N\left[\frac{1}{3}, \sqrt[3]{\frac{64}{9}} + o(1)\right]$$
$$= \exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right)(\log N)^{1/3}(\log\log N)^{2/3}\right)$$

This is approximately $L^{1.923+o(1)}$, where $L = L_N[1/3, 1]$
Search U with Grover's algorithm

- Use Shor's algorithm as Grover oracle (mark smooth numbers)
- Runtime: $L^{\sqrt[3]{8/3}+o(1)} \approx L^{1.387}$ (for Shor: $L^{o(1)}$)
- Qubit requirement: $(\log N)^{2/3+o(1)}$ (for Shor: $(\log N)^{1+o(1)}$)

Brief overview of the NFS

Tune the parameters so we can factor with overwhelming probability. Let

- $y \in L^{\beta+o(1)}$
- $u \in L^{\epsilon+o(1)}$
- $d \in (\delta + o(1))(\log N)^{1/3}(\log\log N)^{2/3}$

U has size $u^{2+o(1)} = L^{2\epsilon+o(1)}$.
By the Prime Number Theorem there are

$$\pi(y) \approx y/\ln(y) = y^{1+o(1)}$$

primes $\leq y$. Assume numbers $F(a, b)$ have the same probability of being prime.
The search range needs to contain $y^{1+o(1)} = L^{\beta+o(1)}$ y-smooth $F(a, b)$.

Brief overview of the NFS

Classically

- Searching $L^{2\epsilon+o(1)}$ integers takes time $L^{2\epsilon+o(1)}$
- Linear algebra takes time $L^{2\beta+o(1)}$
- Balance $2\epsilon = 2\beta$
- Optimize δ
- $\cdots \Rightarrow L^{1.923+o(1)}$

Quantumly

- Partition U in $L^{\beta+o(1)}$ parts of size $L^{2\epsilon-\beta+o(1)}$.
- Search each with Grover in time $L^{\epsilon-\beta/2+o(1)}$.
- Balance $\epsilon + \beta/2 = 2\beta$
- Optimize δ
- $\cdots \Rightarrow L^{1.387+o(1)}$

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.
Bruteforce the solution:

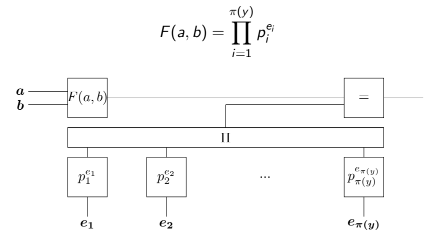
- for all 2^v possible inputs:
- "run the circuit" in time $O(g)$

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

Finding smooth numbers with Circuit-SAT

Direct implementation of smoothness definition:

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



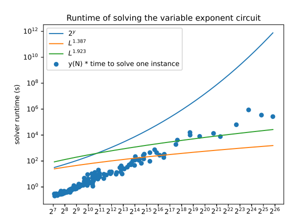
with hardcoded primes.

Finding smooth numbers with Circuit-SAT (cont.)

Size of $(e_1, \ldots e_{\pi(y)})$ is lower-bounded by $\pi(y) \in y^{1+o(1)} = L^{\beta+o(1)}$.
Solver-time will be exponential in $L^{\beta+o(1)}$.

Finding smooth numbers with Circuit-SAT

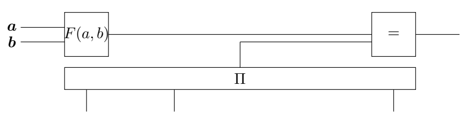
But maybe it works in practice?
Assume $o(1) = 0$ for circuit generation.



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

Finding smooth numbers with Circuit-SAT

Allow non-prime factors $q \leq y$:

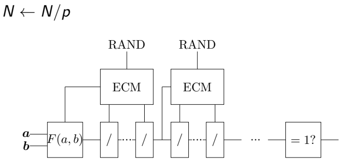


Require all $q_i \leq y$ via input encoding.
Note: $\log F(a, b) = \log(N)^{2/3}$, thus runtime $L_N[2/3, \cdot]$

Finding smooth numbers with Circuit-SAT

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. $p \leftarrow ECM(N)$
3. While $p | N$:
4. $N \leftarrow N/p$



Finding smooth numbers with Circuit-SAT (cont.)

UNIVERSITY OF WATERLOO

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 $p \leq y \in L_N[1/3, \cdot]$, we get a circuit of size $L_N[1/6, \cdot]$.
Repeat this polylog(N) times until you factor with probability $1-o(1)$.
Input-space $2^v \in L_N[1/3, 2\epsilon - \beta + o(1)]$
Circuit size $g \in L_N[1/6, \cdot]$
Assuming a quadratic speedup, we have quantum runtime $L_N[1/3, \epsilon - \beta/2 + o(1)]$

Problem

UNIVERSITY OF WATERLOO

The quadratic speedup over our bruteforce Circuit-SAT solver suffices: $O(\sqrt{2^v}g)$
However, when we say Circuit-SAT is NP-complete, we measure complexity in g

- Best theoretical runtime: $O(2^{0.4058g})$
- The standard translations to SAT/QUBO are polynomial in g
- So we expect a solver runtime exponentional in g: $2^{L_N[1/6,\cdot]}$
- To beat the NFS this solver requires a superpolynomial speedup.

Conclusions/Future work

UNIVERSITY OF WATERLOO

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 bruteforce strategy to QUBO?

Thank you

References

UNIVERSITY OF WATERLOO

SAT: bits

UNIVERSITY OF WATERLOO

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

```
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

```
neg :: Bit -> Bit
neg (Val b) = Val (not b)
neg (Var v) = Var (-v)
```

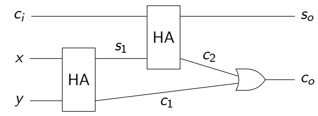
```
nandGate :: Bit -> Bit -> SymEval Bit
nandGate x y = do
  z <- nextVar
  addClauses [[x, z]
             ,[y, z]
             ,[neg x, neg y, neg z]]
  return z
```

```
halfAdd :: Bit -> Bit -> SymEval (Bit, Bit)
halfAdd x y = do
  s <- xorGate x y
  c <- andGate x y
  return (s, c)
```

```
fullAdd x y ci = do
  (s1, c1) <- halfAdd x y
  (so, c2) <- halfAdd s1 ci
  co <- orGate c1 c2
  return (so, co)
```


Finding smooth numbers with Circuit-SAT: gates

Full adder:



Finding smooth numbers with Circuit-SAT: gates

Optimize gates beyond 3-SAT

```
fullAdd x y ci = do
  so <- nextVar
  addClauses [[neg x, neg y, neg ci,     so]
             ,[neg x, neg y,     ci, neg so]
             ,[neg x,     y, neg ci, neg so]
             ,[neg x,     y,     ci,     so]
             ,[    x, neg y, neg ci, neg so]
             ,[    x, neg y,     ci,     so]
             ,[    x,     y, neg ci,     so]
             ,[    x,     y,     ci, neg so]]
  co <- nextVar
  addClauses [[neg x, neg y,             co]
             ,[neg x,        neg ci,     co]
             ,[    x,     y,         neg co]
             ,[    x,            ci, neg co]
             ,[       neg y, neg ci,     co]
             ,[           y,     ci, neg co]]
  return (so, co)
```