quantum algorithms for sieving short(est) vectors in lattices

Elena Kirshanova, Erik Mårtensson, Eamonn Postlethwaite, Subhayan Roy Moulik

London-ish Lattice Coding & Crypto Meeting
Imperial College
18th September, 2019

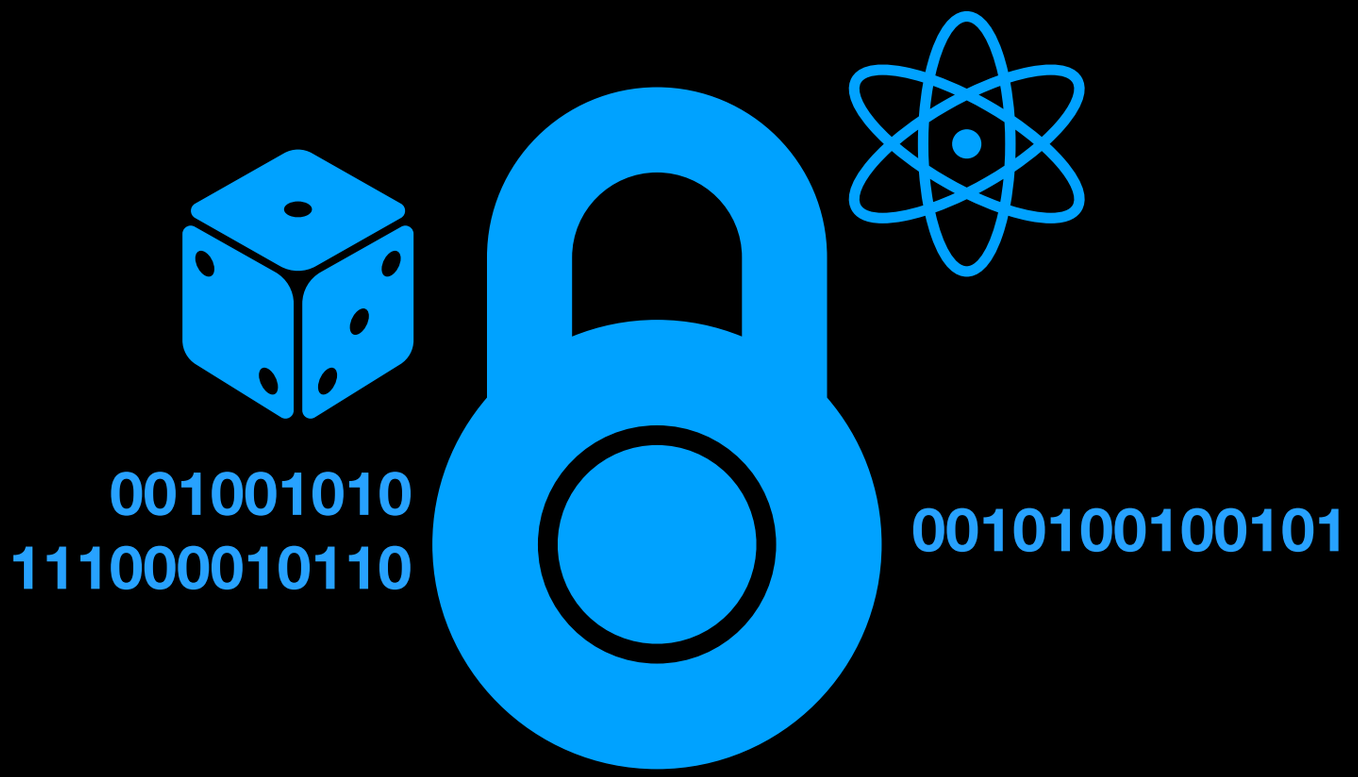
ia.cr/2019/1016
Quantum algorithms for the Approximate k-List Problem and their Application to Lattice Sieving

NIST PQC Standardisation Process - Round 2

Bike	Classic McEllice	CRYSTALS-DILITHIUM	CRYSTALS-KYBER
FALCON	FrodoKEM	GeMSS	HQC
LAC	LEDAcrypt	LUOV	MQDSS
NewHope	NTRU	NTRU Prime	NTS-KEM
Picnic	qTESLA	Rainbow	ROLLO
Round5	RQC	SABER	SIKE
	SPHINCS+	Three Bears	

NIST PQC Standardisation Process - Round 2

Bike	Classic McEllice	CRYSTALS-DILITHIUM	CRYSTALS-KYBER
FALCON	FrodoKEM	GeMSS	HQC
LAC	LEDAcrypt	LUOV	MQDSS
NewHope	NTRU	NTRU Prime	NTS-KEM
Picnic	qTESLA	Rainbow	ROLLO
Round5	RQC	SABER	SIKE
	SPHINCS+	Three Bears	


001001010
111000010110
0010100100101

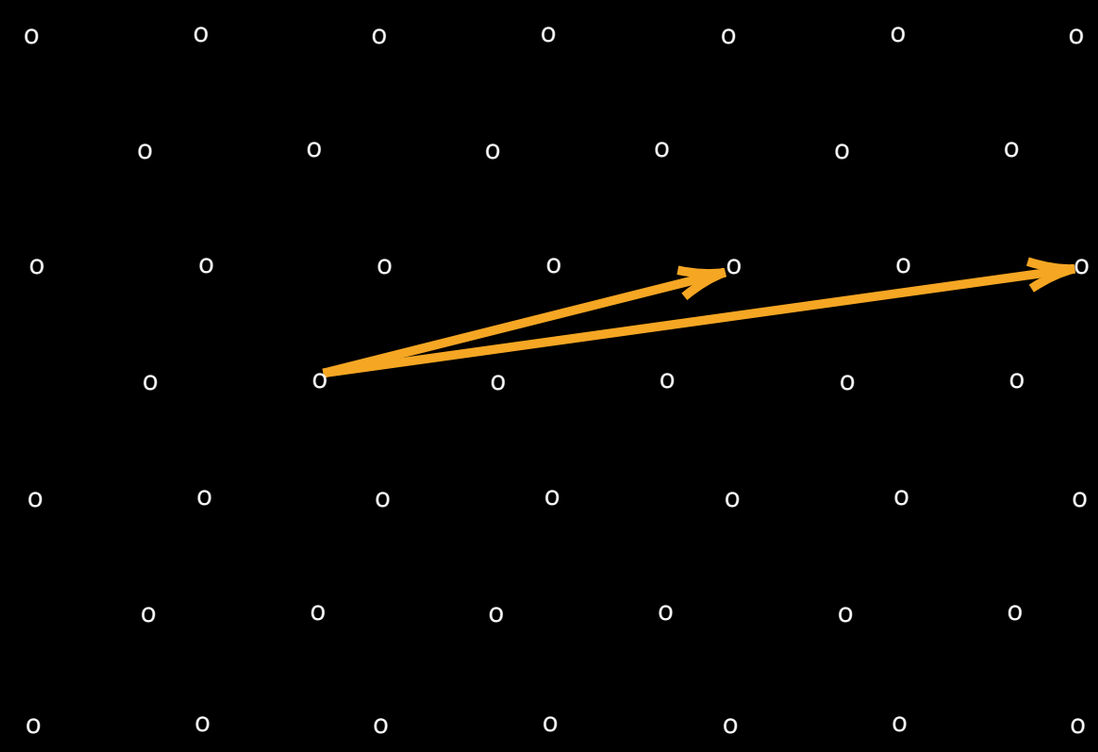
Lattices and SVP

Given m linear independent vectors,

$$B = \{b_1, \dots b_m \ : b_i \in \mathbb{R}^n\}$$

the lattice generated by them is defined as

$$\mathcal{L}(B) = \{\sum_{i=1}^{m} z_i b_i \mid z_i \in \mathbb{Z}\}$$



Lattices and SVP

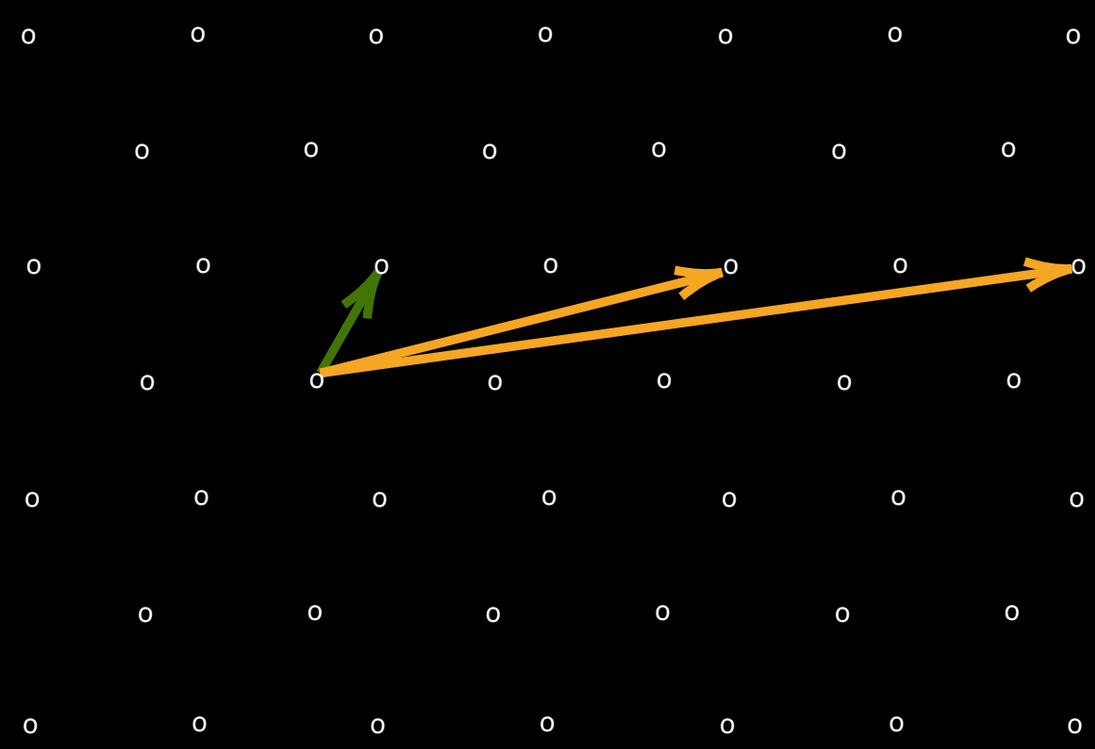
Given m linear independent vectors,

$$B = \{b_1, \dots b_m \ : b_i \in \mathbb{R}^n\}$$

the lattice generated by them is defined as

$$\mathcal{L}(B) = \{\sum_{i=1}^{m} z_i b_i \mid z_i \in \mathbb{Z}\}$$

Shortest Vector Problem (SVP): Given $\mathcal{L}(B)$, find the shortest non-zero vector



Lattices and SVP

Given m linear independent vectors,

$$B = \{b_1, \dots b_m \ : b_i \in \mathbb{R}^n\}$$

the lattice generated by them is defined as

$$\mathcal{L}(B) = \{\sum_{i=1}^{m} z_i b_i \mid z_i \in \mathbb{Z}\}$$

Shortest Vector Problem (SVP): Given $\mathcal{L}(B)$, find the shortest non-zero vector

Best known algorithms are atleast exponential time

provable vs heuristic

SVP hardness

Theory

	Algorithm	$\log_2$(Time)	$\log_2$(Space)
Proven SVP	Enumeration [Poh81, Kan83, ..., MW15, AN17]	$O(n \log n)$	$O(\log n)$
	AKS-sieve [AKS01, NV08, MV10, HPS11]	$3.398n$	$1.985n$
	ListSieve [MV10, MDB14]	$3.199n$	$1.327n$
	Birthday sieves [PS09, HPS11]	$2.465n$	$1.233n$
	Enumeration/DGS hybrid [CCL17]	$2.048n$	$0.500n$
	Voronoi cell algorithm [AEVZ02, MV10b]	$2.000n$	$1.000n$
	Quantum sieve [LMP13, LMP15]	$1.799n$	$1.286n$
	Quantum enum/DGS [CCL17]	$1.256n$	**0.500n**
	Discrete Gaussian sampling [ADRS15, ADS15, AS18]	**1.000n**	$1.000n$
Heuristic SVP	The Nguyen–Vidick sieve [NV08]	$0.415n$	$0.208n$
	The GaussSieve [MV10, ..., IKMT14, BNvdP16, YKYC17]	$0.415n$	$0.208n$
	Triple sieve [BLS16, HK17]	$0.396n$	$0.189n$
	Two-level sieve [WLTB11]	$0.384n$	$0.256n$
	Three-level sieve [ZPH13]	$0.3778n$	$0.283n$
	Overlattice sieve [BGJ14]	$0.3774n$	$0.293n$
	Triple sieve with NNS [HK17, HKL18]	$0.359n$	**0.189n**
	Hyperplane LSH [Cha02, Laa15, ..., LM18, Duc18]	$0.337n$	$0.337n$
	Graph-based NNS [EPY99, DCL11, MPLK14, Laa18]	$0.327n$	$0.282n$
	Hypercube LSH [TT07, Laa17]	$0.322n$	$0.322n$
	Quantum sieve [LMP13, LMP15]	$0.312n$	$0.208n$
	May–Ozerov NNS [MO15, BGJ15]	$0.311n$	$0.311n$
	Spherical LSH [AINR14, LdW15]	$0.298n$	$0.298n$
	Cross-polytope LSH [TT07, AILRS15, BL16, KW17]	$0.298n$	$0.298n$
	Spherical LSF [BDGL16, MLB17, ALRW17, Chr17]	**0.292n**	$0.292n$
	Quantum NNS sieve [LMP15, Laa16]	**0.265n**	$0.265n$

n = dimension

slide stolen from Thijs Laarhoven
http://www.thijs.com/docs/pqc18-slides.pdf

BUT EXPONENTIAL TIME ALGORITHM !!

BUT EXPONENTIAL TIME ALGORITHM !!

Efficiency vs Security: Battle for dimensions

HALL OF FAME

Position	Dimension	Euclidean norm	Seed	Contestant	Solution
1	157	3320	0	L. Ducas, M. Stevens, W. van Woerden	vec
2	155	3165	0	M. Albrecht, L. Ducas, G. Herold, E. Kirshanova, E. Postlethwaite, M. Stevens, P. Karpman	vec
3	153	3192	0	Martin Albrecht, Leo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn Postlethwaite, Marc Stevens	vec
4	152	3217	0	Kenji KASHIWABARA and Tadanori TERUYA	vec
5	151	3233	0	Martin Albrecht, Leo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn Postlethwaite, Marc Stevens	vec

Complete list

*Darmstadt lattice challenge : SVP

SVP hardness

Theory

	Algorithm	$\log_2$(Time)	$\log_2$(Space)
Proven SVP	Enumeration [Poh81, Kan83, ..., MW15, AN17]	$O(n \log n)$	$O(\log n)$
	AKS-sieve [AKS01, NV08, MV10, HPS11]	$3.398n$	$1.985n$
	ListSieve [MV10, MDB14]	$3.199n$	$1.327n$
	Birthday sieves [PS09, HPS11]	$2.465n$	$1.233n$
	Enumeration/DGS hybrid [CCL17]	$2.048n$	$0.500n$
	Voronoi cell algorithm [AEVZ02, MV10b]	$2.000n$	$1.000n$
	Quantum sieve [LMP13, LMP15]	$1.799n$	$1.286n$
	Quantum enum/DGS [CCL17]	$1.256n$	**$0.500n$**
	Discrete Gaussian sampling [ADRS15, ADS15, AS18]	**$1.000n$**	$1.000n$
Heuristic SVP	The Nguyen–Vidick sieve [NV08]	$0.415n$	$0.208n$
	The GaussSieve [MV10, ..., IKMT14, BNvdP16, YKYC17]	$0.415n$	$0.208n$
	Triple sieve [BLS16, HK17]	$0.396n$	$0.189n$
	Two-level sieve [WLTB11]	$0.384n$	$0.256n$
	Three-level sieve [ZPH13]	$0.3778n$	$0.283n$
	Overlattice sieve [BGJ14]	$0.3774n$	$0.293n$
	Triple sieve with NNS [HK17, HKL18]	$0.359n$	**$0.189n$**
	Hyperplane LSH [Cha02, Laa15, ..., LM18, Duc18]	$0.337n$	$0.337n$
	Graph-based NNS [EPY99, DCL11, MPLK14, Laa18]	$0.327n$	$0.282n$
	Hypercube LSH [TT07, Laa17]	$0.322n$	$0.322n$
	Quantum sieve [LMP13, LMP15]	$0.312n$	$0.208n$
	May–Ozerov NNS [MO15, BGJ15]	$0.311n$	$0.311n$
	Spherical LSH [AINR14, LdW15]	$0.298n$	$0.298n$
	Cross-polytope LSH [TT07, AILRS15, BL16, KW17]	$0.298n$	$0.298n$
	Spherical LSF [BDGL16, MLB17, ALRW17, Chr17]	**$0.292n$**	$0.292n$
	Quantum NNS sieve [LMP15, Laa16]	**$0.265n$**	$0.265n$

n = dimension

slide stolen from Thijs Laarhoven
http://www.thijs.com/docs/pqc18-slides.pdf

Heuristic Assumption:
Normalised vectors are uniformly distributed on the unit sphere

Algorithmic idea for sieving

Input : $\{b_1, \ldots, b_n\}$ s.t. $b_i \in \mathbb{R}^n$
Output : $v \in \mathbb{R}^n$

1. Sample L= exp(n) vectors, put them in a list
2. Combine k(=constant) of them ‘cleverly’ to obtain L’ shorter vectors
3. Repeat step 2 with the new list of vectors, poly(n) times
4. Output the shortest vector from the latest list.

[KMPR19] : Can solve SVP, heuristically, in time $2^{0.2989n+o(n)}$ and (classical) memory $2^{0.1385n+o(n)}$, in the QRAM model. This can be implemented on a quantum computer equipped with $poly(n)$ quantum memory.

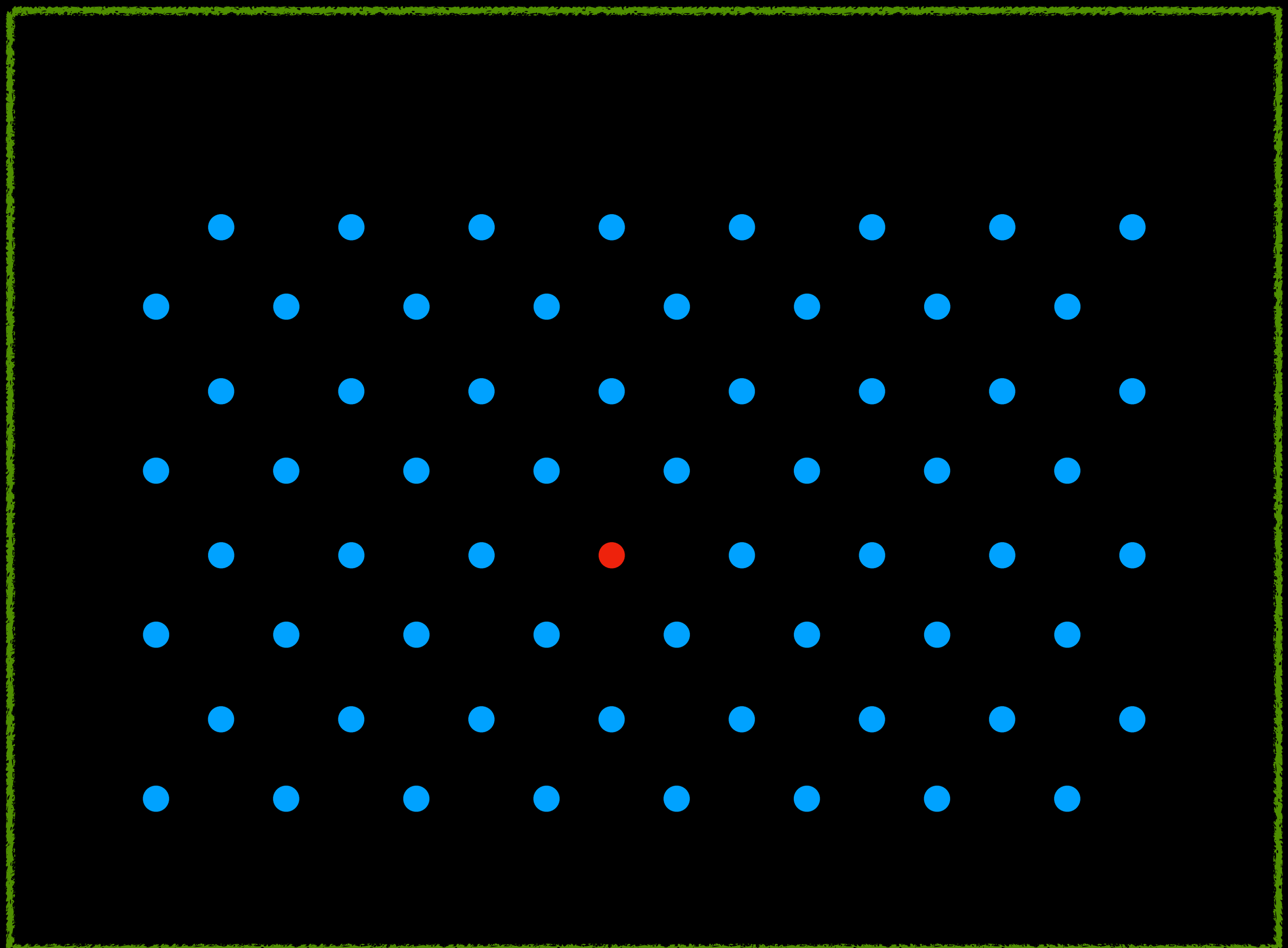
[KMPR19] : Can solve SVP, heuristically, in time $2^{0.1037n+o(n)}$ and (quantum) memory $2^{0.2075n+o(n)}$, in the quantum circuit model.

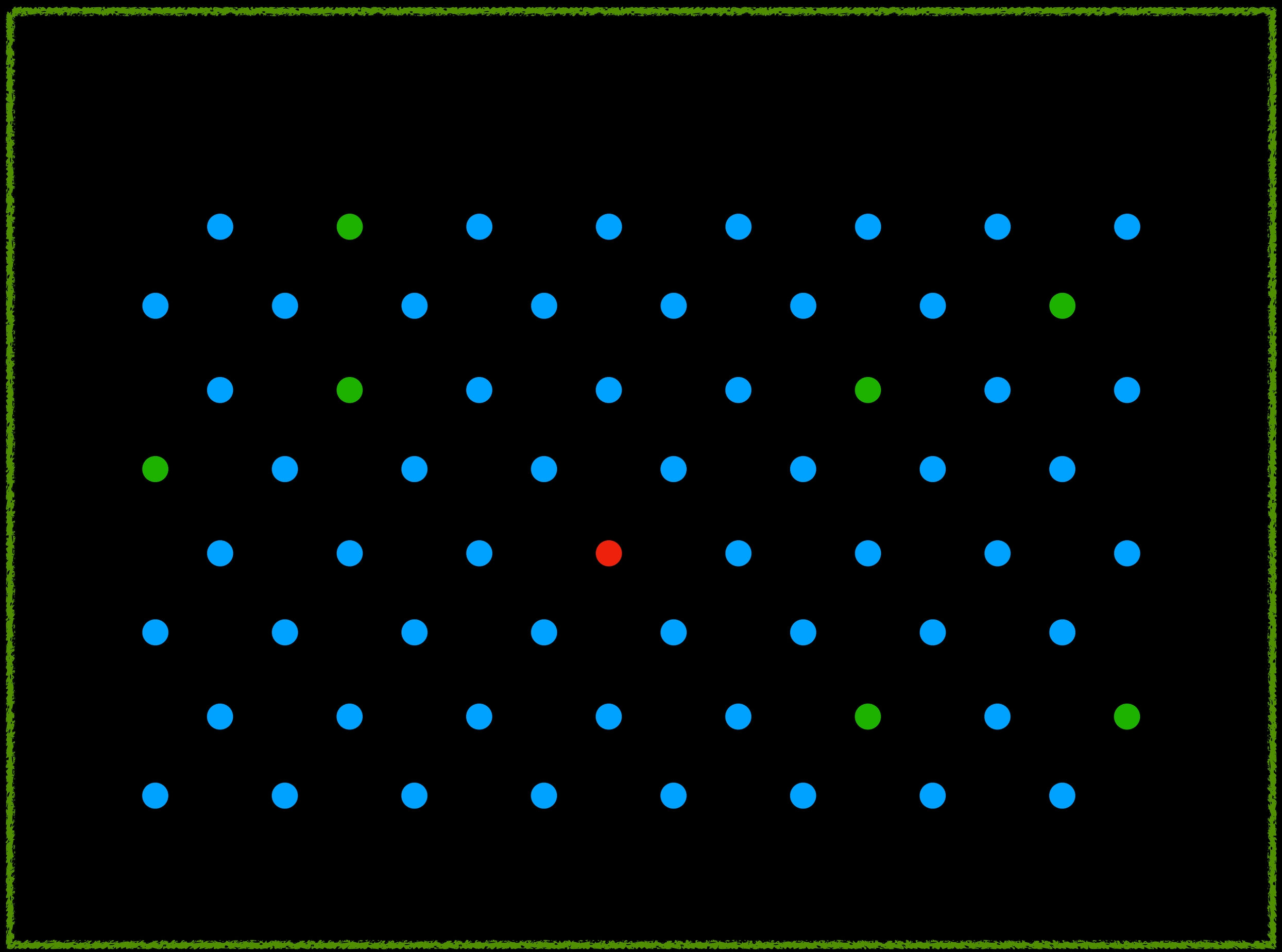
[KMPR19] : Can solve SVP, heuristically, in time $2^{0.2989n+o(n)}$ and (classical) memory $2^{0.1385n+o(n)}$, in the QRAM model. This can be implemented on a quantum computer equipped with $poly(n)$ quantum memory.

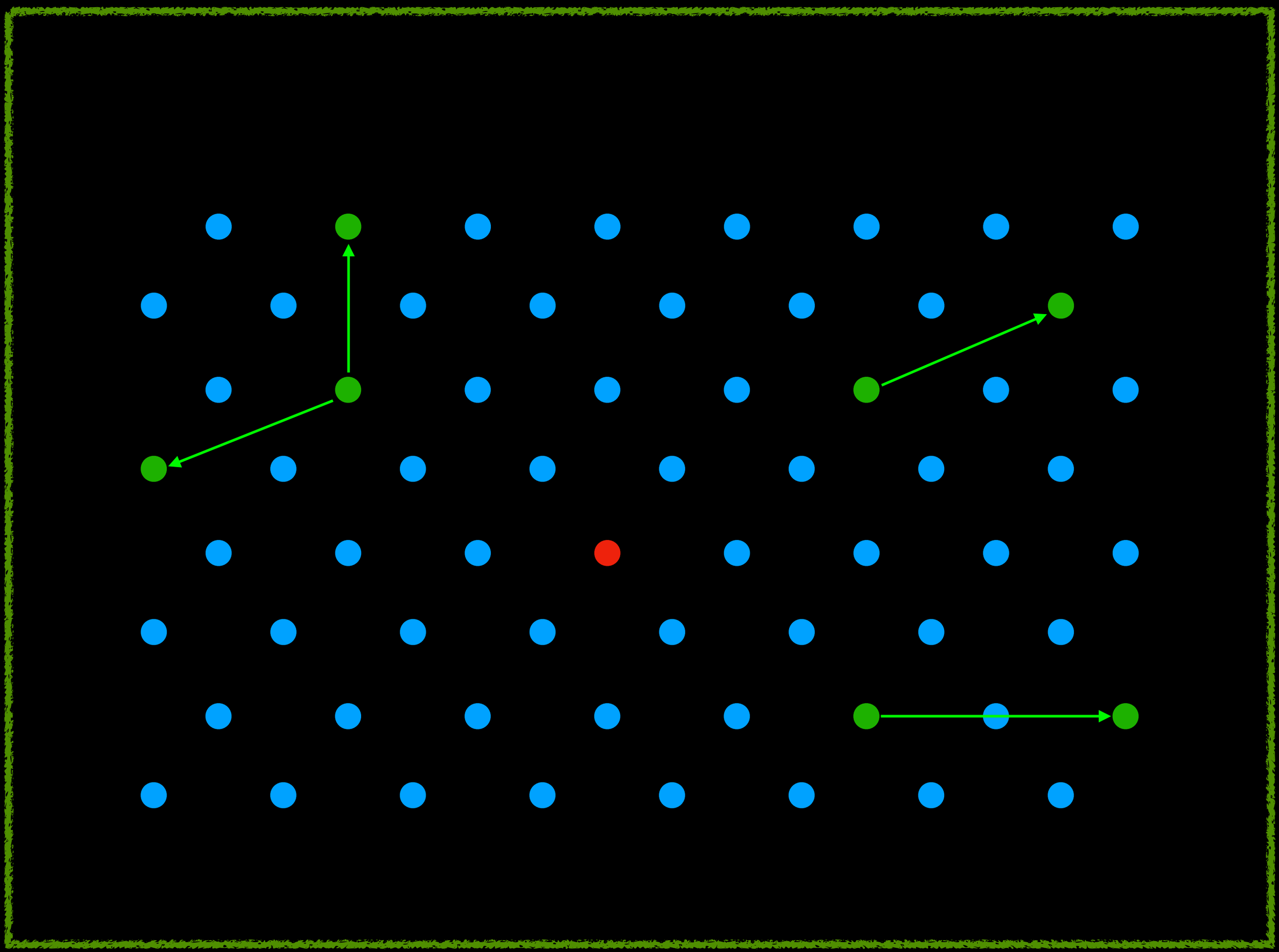
[KMPR19] : Can solve SVP, heuristically, in time $2^{0.1037n+o(n)}$ and (quantum) memory $2^{0.2075n+o(n)}$, in the quantum circuit model.

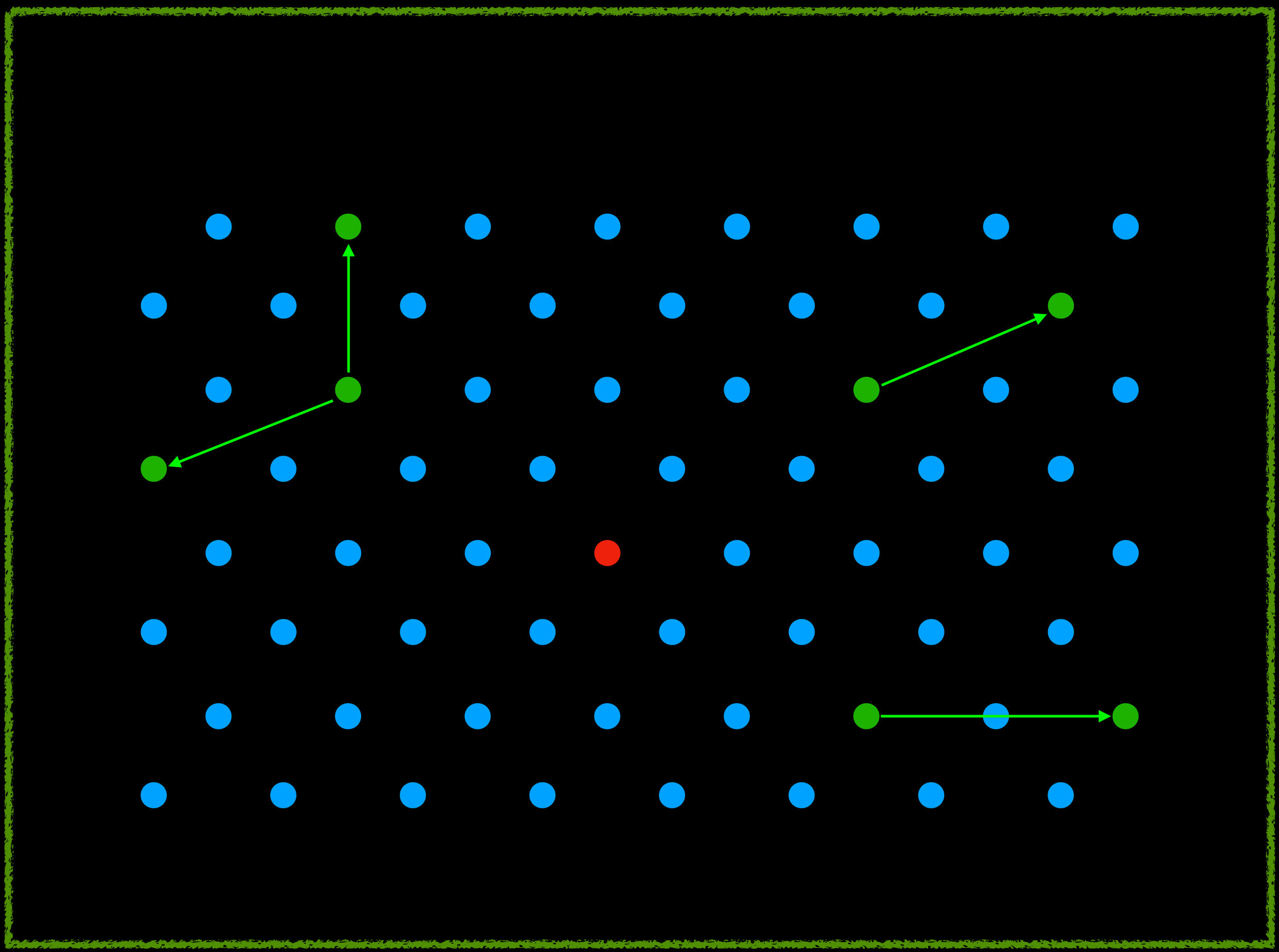
	log2(Time)	log2(Memory)
The Nguyen–Vidick sieve [NV08]	0.415n	0.208n
The GaussSieve [MV10, . . . , IKMT14, BNvdP16, YKYC17]	0.415n	0.208n
Triple sieve [BLS16, HK17]	0.396n	0.189n
Two-level sieve [WLTB11]	0.384n	0.256n
Three-level sieve [ZPH13]	0.3778n	0.283n
Overlattice sieve [BGJ14]	0.3774n	0.293n
Triple sieve with NNS [HK17, HKL18]	0.359n	**0.189n**
Hyperplane LSH [Cha02, Laa15, . . . , LM18, Duc18]	0.337n	0.337n
Graph-based NNS [EPY99, DCL11, MPLK14, Laa18]	0.327n	0.282n
Hypercube LSH [TT07, Laa17]	0.322n	0.322n
Quantum sieve [LMP13, LMP15]	0.312n	0.208n
May–Ozerov NNS [MO15, BGJ15]	0.311n	0.311n
Spherical LSH [AINR14, LdW15]	0.298n	0.298n
Cross-polytope LSH [TT07, AILRS15, BL16, KW17]	0.298n	0.298n
Spherical LSF [BDGL16, MLB17, ALRW17, Chr17]	**0.292n**	0.292n
Quantum NNS sieve [LMP15, Laa16]	**0.265n**	0.265n

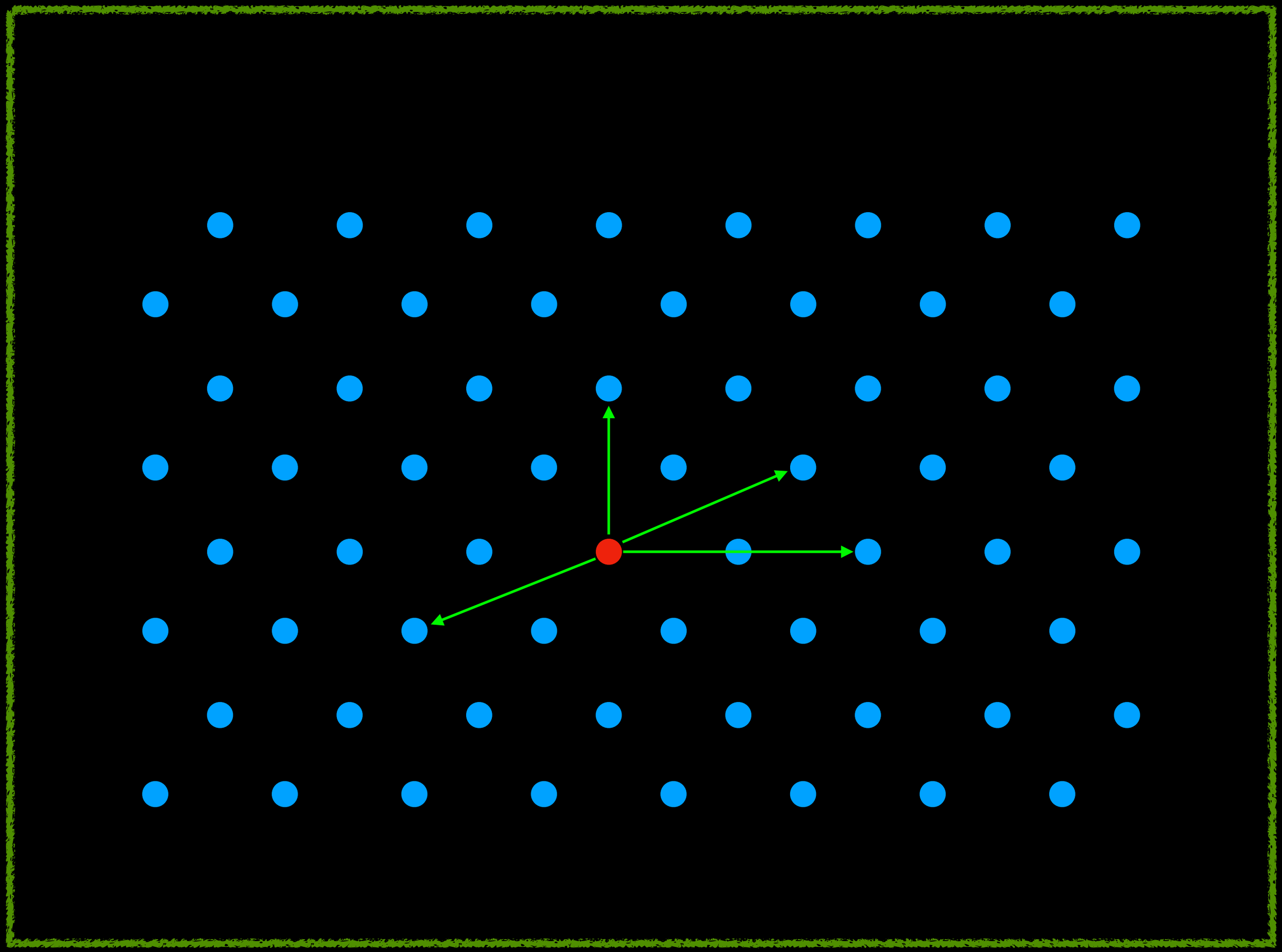
2 - Sieve

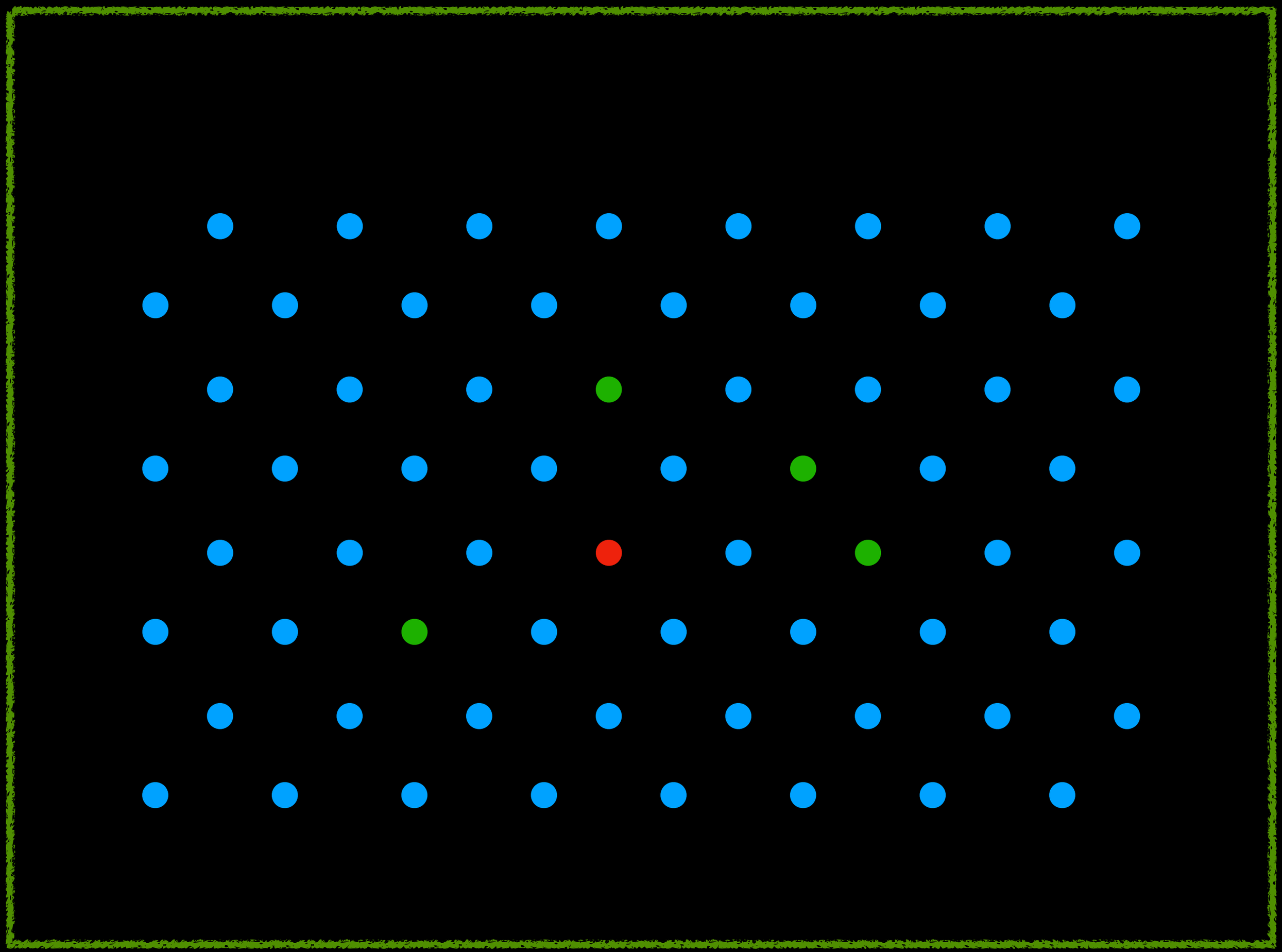


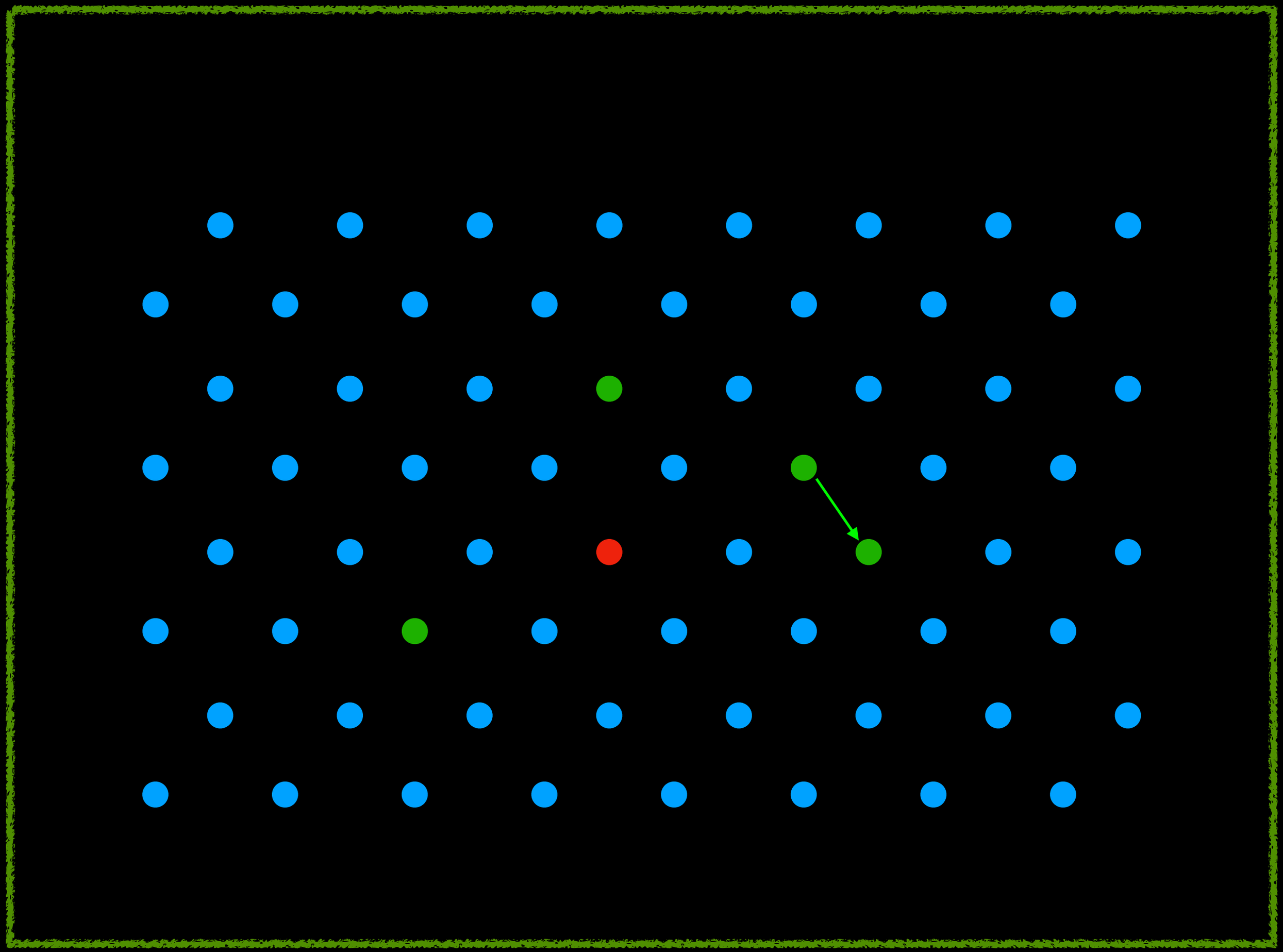


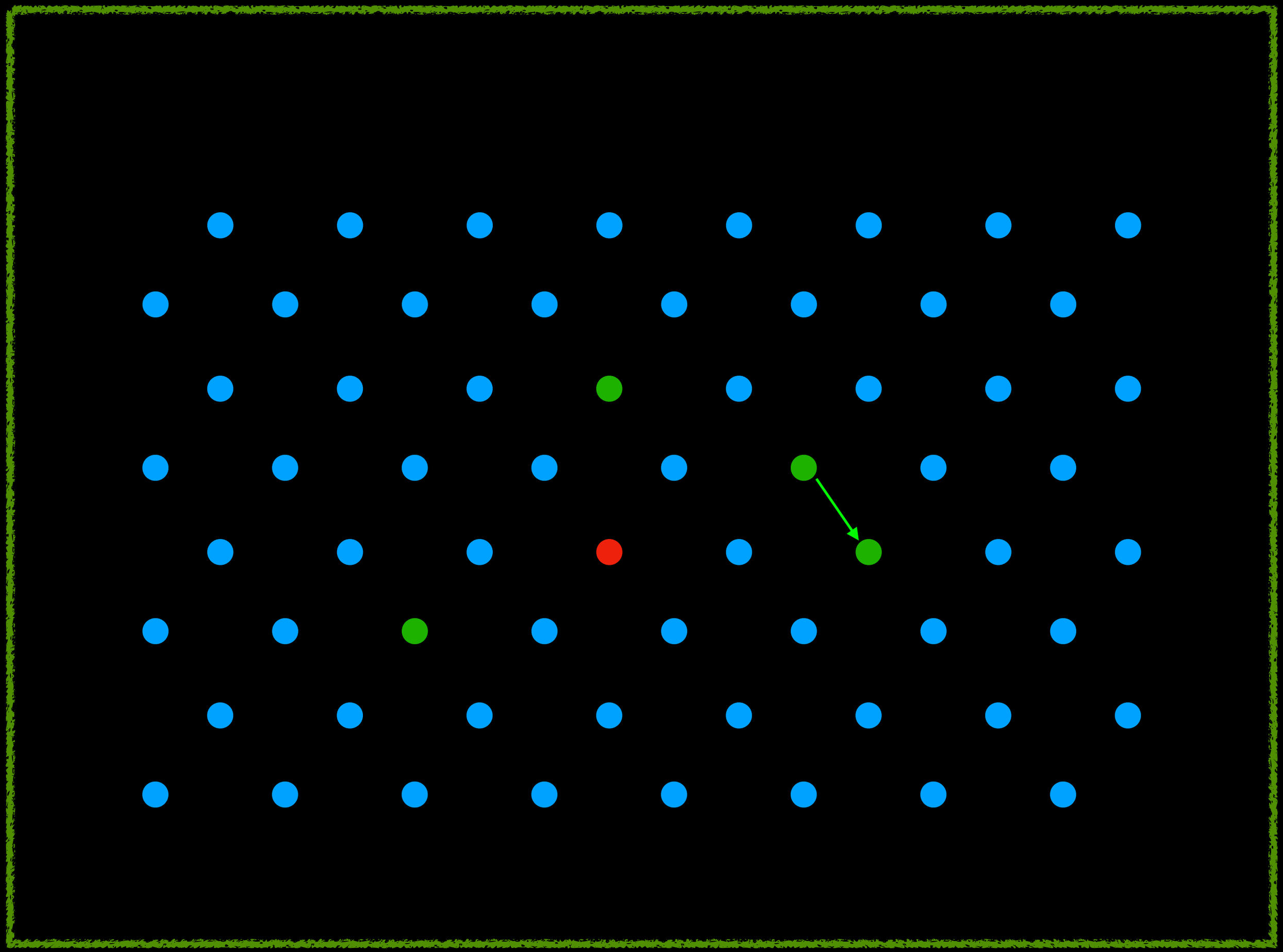


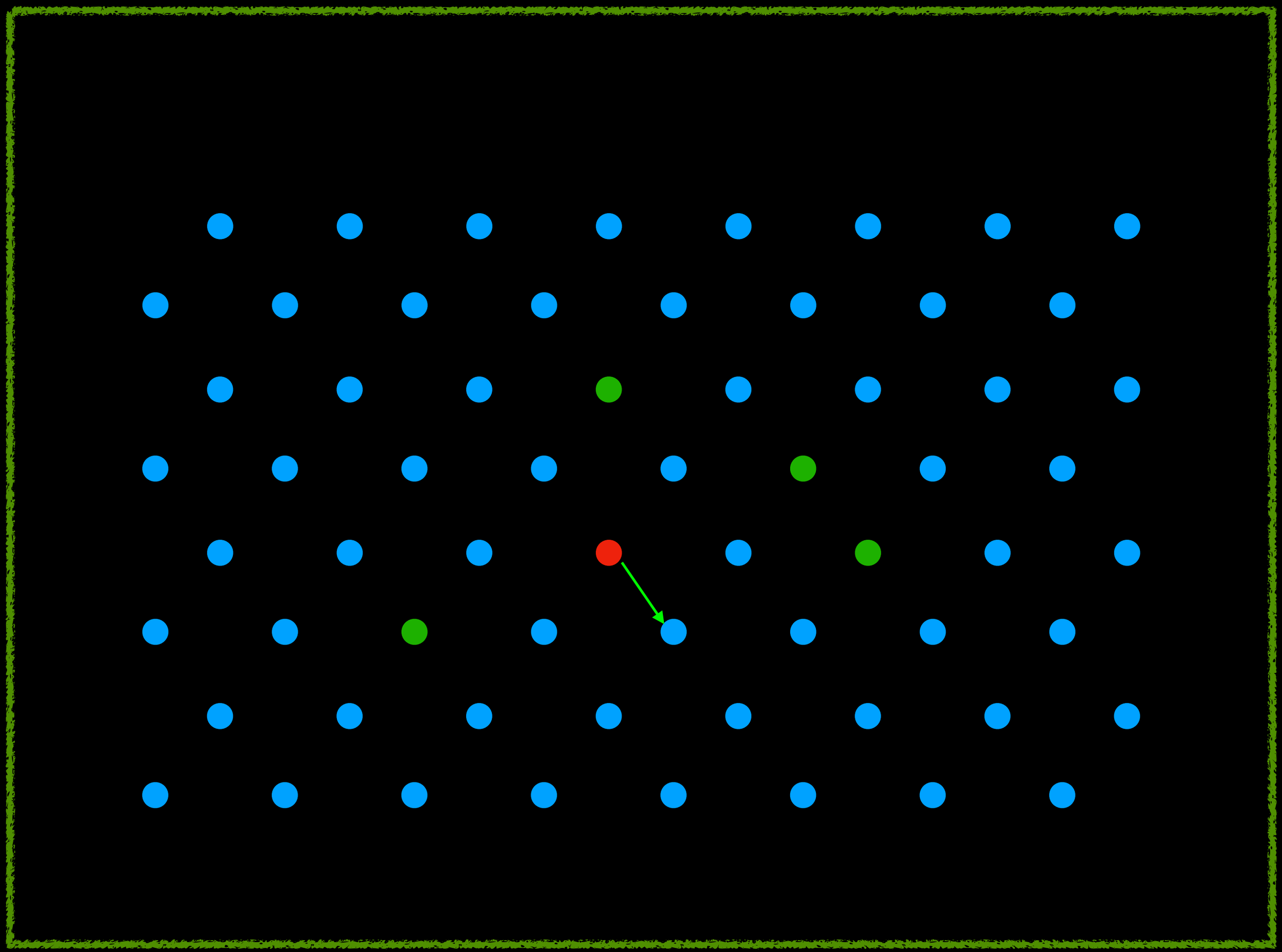


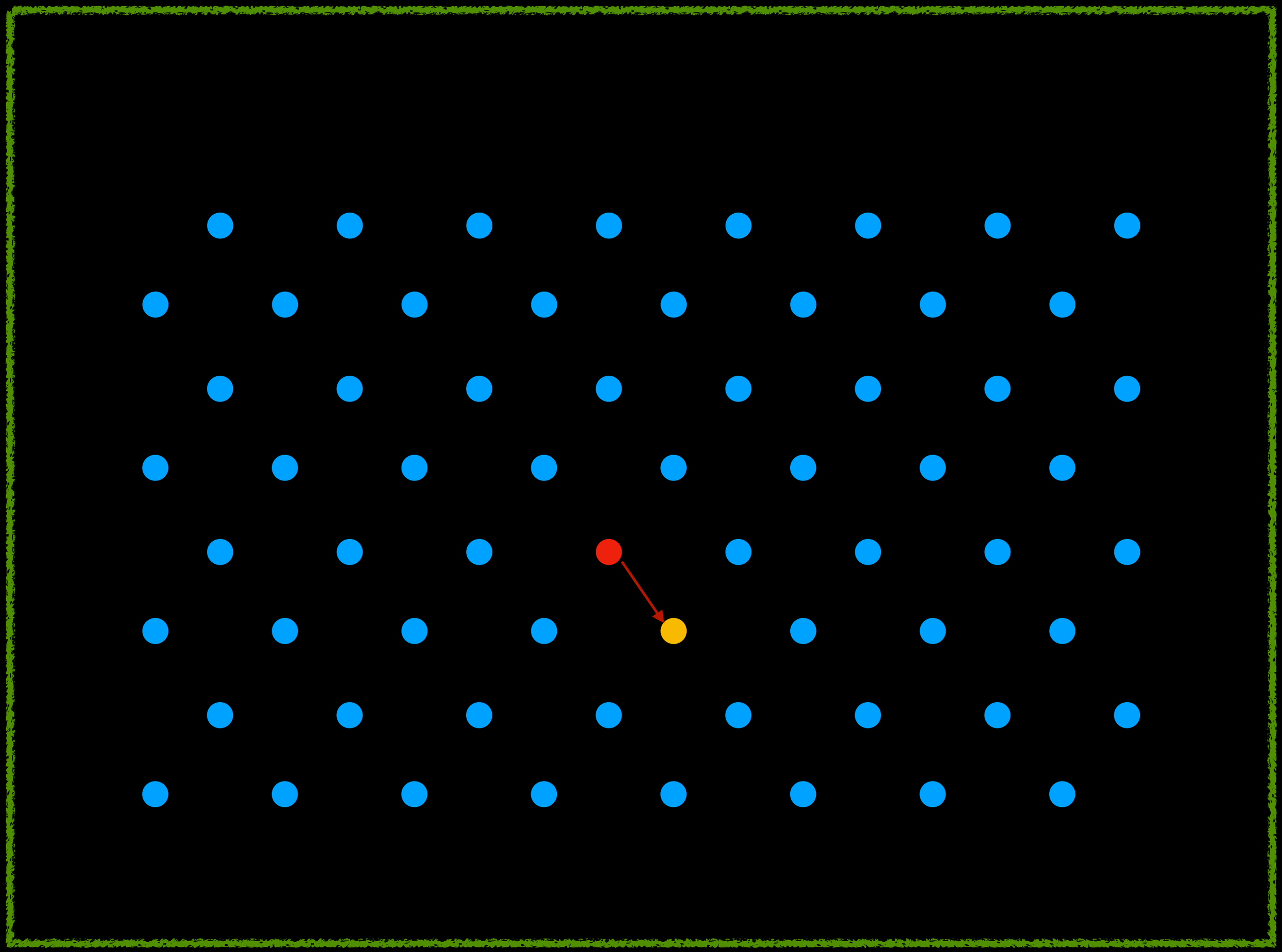












2-Sieve: Algorithm

Input: $\{b_1, \ldots, b_d\} \subset R^d$ a lattice basis
Output: $v \in \mathcal{L}(B)$, a short vector from $\mathcal{L}(B)$

1. Set $N \leftarrow 2^{0.2075d+o(d)}$ and set $\lambda = \Theta(\sqrt{d} \cdot \det(B)^{1/d})$ the target length.

2. Generate a list $L_1 \leftarrow \{x_1, \ldots, x_N\}$ of normalised lattice vectors using an efficient lattice sampling procedure.

3. Construct a list $L_2 \leftarrow \{x'_1, \ldots, x'_N\}$ such that $| < x_i, x'_i > | \geq 1/2$ for $x'_i \in L_1$. If no such $x'_i \in L_1$ exists, set $x'_i = \mathbf{0}$.

4. Construct a list $L_3 \leftarrow \{y_i : y_i \leftarrow \min\{||x_i \pm x'_i||\}$ for all $i \leq N\}$ and normalise its elements except for the last iteration.

5. Swap the labels L_1, L_3. Reinitialise L_2 and L_3 to the zero state by transferring their contents to auxiliary memory.

6. Repeat Steps 3-5 $poly(d)$ times.

7. Output a vector from L_1 of Euclidean norm less than λ.

2-Sieve: Algorithm

Input: $\{b_1, \ldots, b_d\} \subset R^d$ a lattice basis
Output: $v \in \mathcal{L}(B)$, a short vector from $\mathcal{L}(B)$

1. Set $N \leftarrow 2^{0.2075d+o(d)}$ and set $\lambda = \Theta(\sqrt{d} \cdot \det(B)^{1/d})$ the target length.

2. Generate a list $L_1 \leftarrow \{x_1, \ldots, x_N\}$ of normalised lattice vectors using an efficient lattice sampling procedure.

3. Construct a list $L_2 \leftarrow \{x'_1, \ldots, x'_N\}$ such that $| < x_i, x'_i > | \geq 1/2$ for $x'_i \in L_1$. If no such $x'_i \in L_1$ exists, set $x'_i = \mathbf{0}$.

4. Construct a list $L_3 \leftarrow \{y_i : y_i \leftarrow \min\{||x_i \pm x'_i||\}$ for all $i \leq N\}$ and normalise its elements except for the last iteration.

5. Swap the labels L_1, L_3. Reinitialise L_2 and L_3 to the zero state by transferring their contents to auxiliary memory.

6. Repeat Steps 3-5 $poly(d)$ times.

7. Output a vector from L_1 of Euclidean norm less than λ.

L_1

x_1
x_2
x_3
$\vdots$
x_N

2-Sieve: Algorithm

Input: $\{b_1, \ldots, b_d\} \subset R^d$ a lattice basis
Output: $v \in \mathcal{L}(B)$, a short vector from $\mathcal{L}(B)$

1. Set $N \leftarrow 2^{0.2075d+o(d)}$ and set $\lambda = \Theta(\sqrt{d} \cdot \det(B)^{1/d})$ the target length.

2. Generate a list $L_1 \leftarrow \{x_1, \ldots, x_N\}$ of normalised lattice vectors using an efficient lattice sampling procedure.

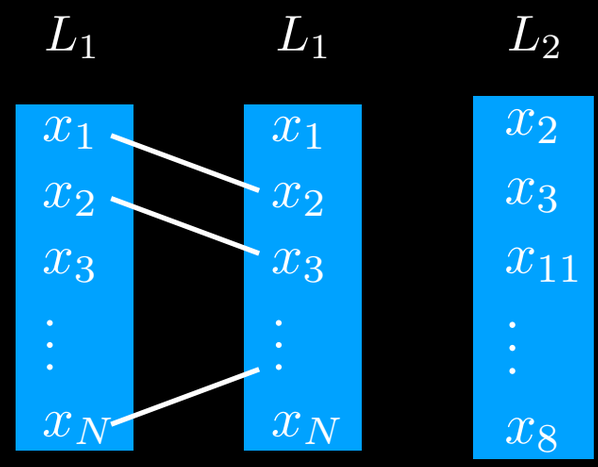
3. Construct a list $L_2 \leftarrow \{x'_1, \ldots, x'_N\}$ such that $| < x_i, x'_i > | \geq 1/2$ for $x'_i \in L_1$. If no such $x'_i \in L_1$ exists, set $x'_i = \mathbf{0}$.

4. Construct a list $L_3 \leftarrow \{y_i : y_i \leftarrow \min\{||x_i \pm x'_i||\}$ for all $i \leq N\}$ and normalise its elements except for the last iteration.

5. Swap the labels L_1, L_3. Reinitialise L_2 and L_3 to the zero state by transferring their contents to auxiliary memory.

6. Repeat Steps 3-5 $poly(d)$ times.

7. Output a vector from L_1 of Euclidean norm less than λ.



2-Sieve: Algorithm

Input: $\{b_1, \ldots, b_d\} \subset R^d$ a lattice basis
Output: $v \in \mathcal{L}(B)$, a short vector from $\mathcal{L}(B)$

1. Set $N \leftarrow 2^{0.2075d+o(d)}$ and set $\lambda = \Theta(\sqrt{d} \cdot \det(B)^{1/d})$ the target length.

2. Generate a list $L_1 \leftarrow \{x_1, \ldots, x_N\}$ of normalised lattice vectors using an efficient lattice sampling procedure.

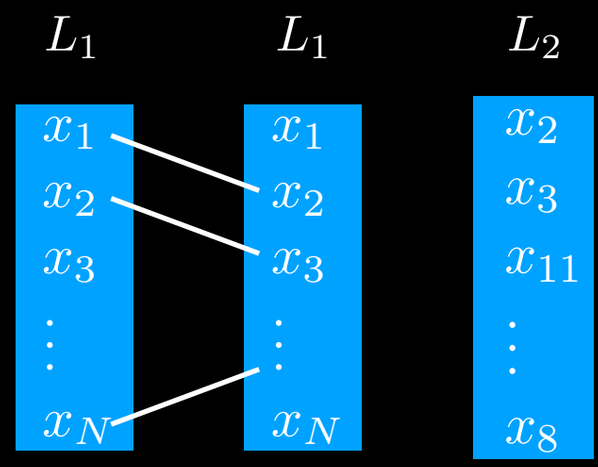
3. Construct a list $L_2 \leftarrow \{x'_1, \ldots, x'_N\}$ such that $| < x_i, x'_i > | \geq 1/2$ for $x'_i \in L_1$. If no such $x'_i \in L_1$ exists, set $x'_i = \mathbf{0}$.

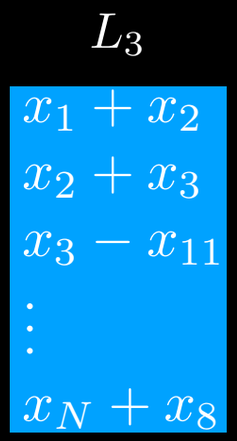
4. Construct a list $L_3 \leftarrow \{y_i : y_i \leftarrow \min\{||x_i \pm x'_i||\}$ for all $i \leq N\}$ and normalise its elements except for the last iteration.

5. Swap the labels L_1, L_3. Reinitialise L_2 and L_3 to the zero state by transferring their contents to auxiliary memory.

6. Repeat Steps 3-5 $poly(d)$ times.

7. Output a vector from L_1 of Euclidean norm less than λ.





2-Sieve: Algorithm

Input: $\{b_1, \ldots, b_d\} \subset R^d$ a lattice basis
Output: $v \in \mathcal{L}(B)$, a short vector from $\mathcal{L}(B)$

1. Set $N \leftarrow 2^{0.2075d+o(d)}$ and set $\lambda = \Theta(\sqrt{d} \cdot \det(B)^{1/d})$ the target length.

2. Generate a list $L_1 \leftarrow \{x_1, \ldots, x_N\}$ of normalised lattice vectors using an efficient lattice sampling procedure.

3. Construct a list $L_2 \leftarrow \{x'_1, \ldots, x'_N\}$ such that $| < x_i, x'_i > | \geq 1/2$ for $x'_i \in L_1$. If no such $x'_i \in L_1$ exists, set $x'_i = \mathbf{0}$.

4. Construct a list $L_3 \leftarrow \{y_i : y_i \leftarrow \min\{||x_i \pm x'_i||\}$ for all $i \leq N\}$ and normalise its elements except for the last iteration.

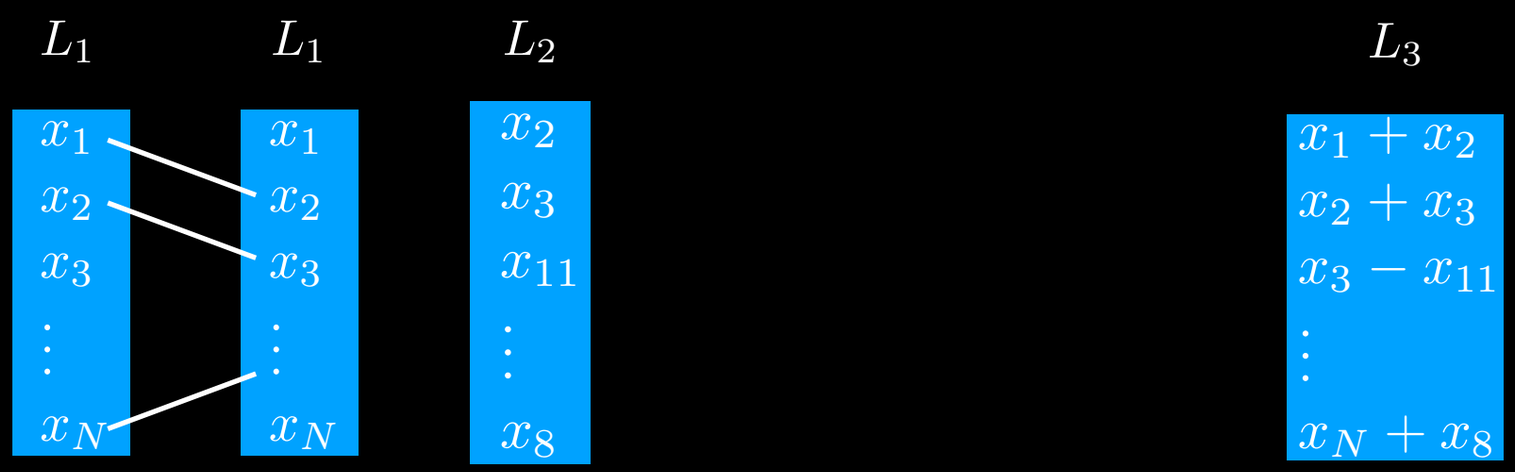
5. Swap the labels L_1, L_3. Reinitialise L_2 and L_3 to the zero state by transferring their contents to auxiliary memory.

6. Repeat Steps 3-5 $poly(d)$ times.

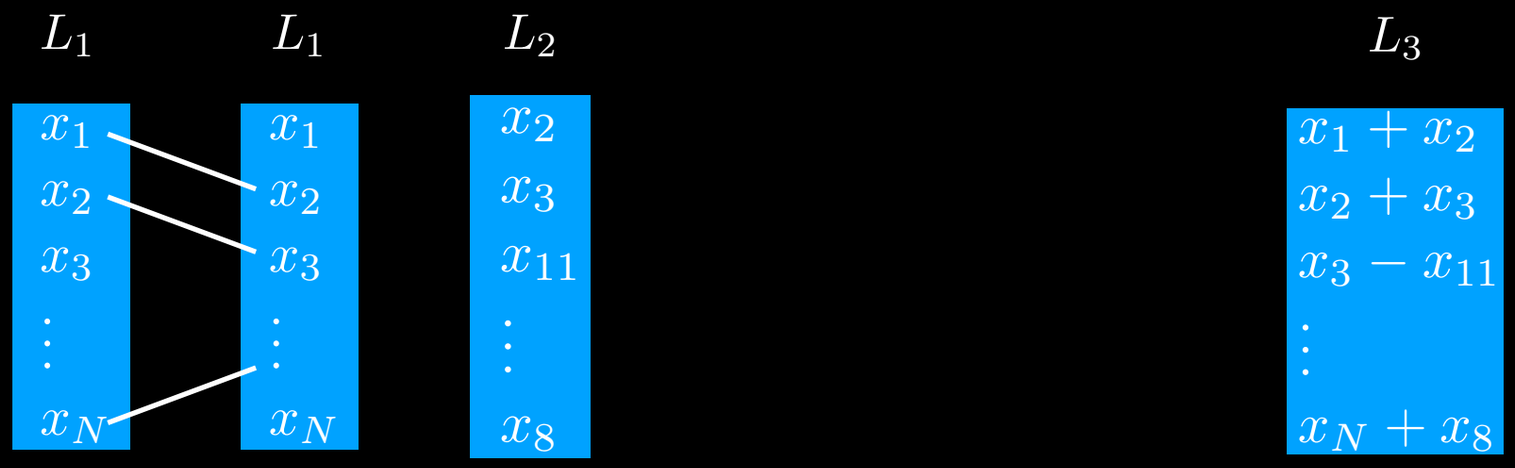
7. Output a vector from L_1 of Euclidean norm less than λ.

L_1

$x_1 + x_2$
$x_2 + x_3$
$x_3 - x_{11}$
$\vdots$
$x_N + x_8$



Idea 1: Pick a vector and search the whole list



Idea 1: Pick a vector and search the whole list

Idea 2: Search the list simultaneously

Given a list L of size N, with each element of the list being d bits, and N functions, $f_i : L \to \{0, 1\}$ for $i = [1...N]$, we wish to find solution vectors $s_i \in [1...N]$.

That is we wish to find s_i for a f_i such that $f_i(x_{s_i}) = 1$ for all $i \in [1...N]$.

Given unitaries $U_{f_i} : |x\rangle |b\rangle \to |x\rangle |b \oplus f_i(x)\rangle$ there exists a quantum algorithm that, for each $i \in [1...N]$, either returns a solution s_i or if there is no such solution, returns *no solution*.

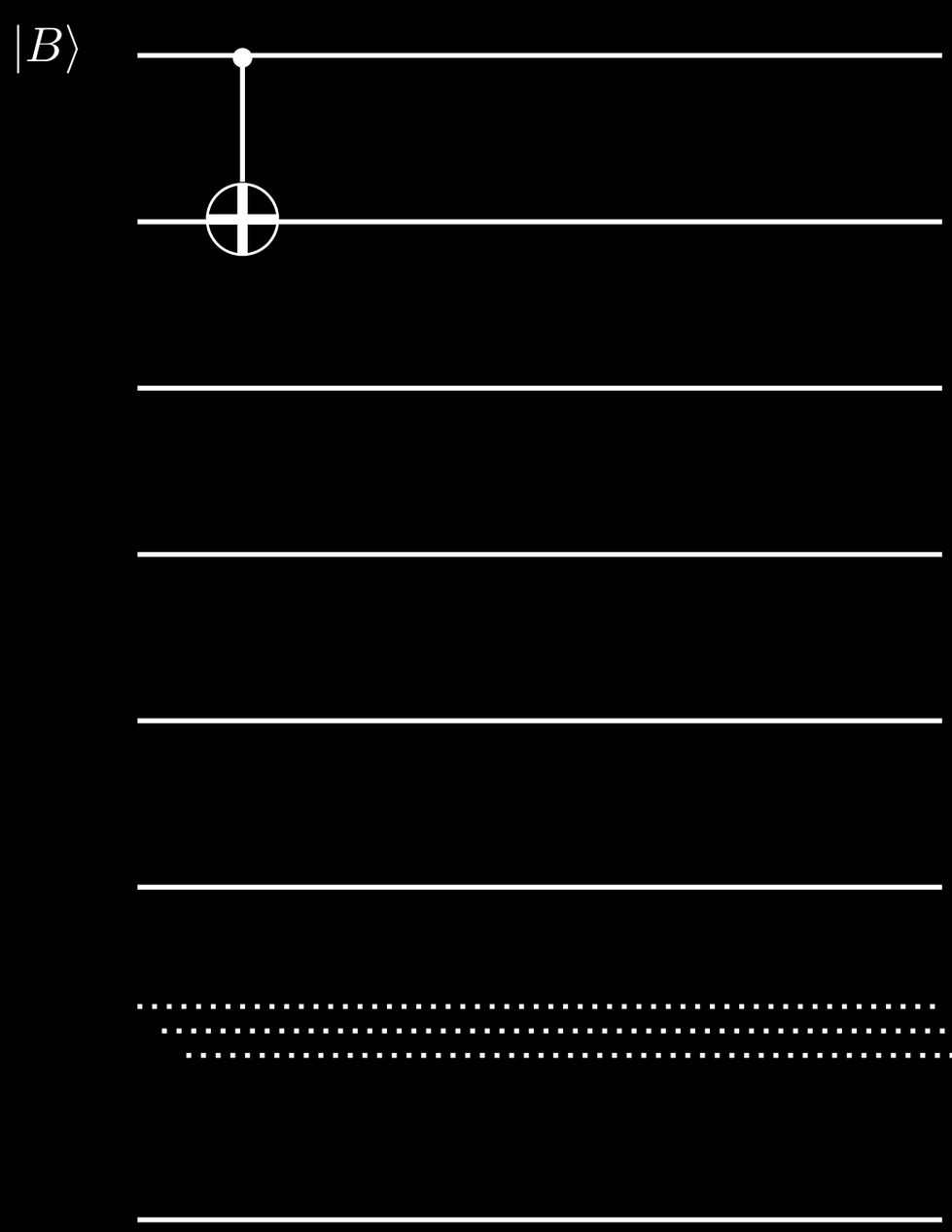
The algorithm succeeds with probability $\Theta(1)$ and, given that each U_{f_i} has depth and width $polylog(N, d)$, this can be implemented using a quantum circuit of width $\widetilde{O}(N)$ and depth $\widetilde{O}(\sqrt{N})$.

Beals R, et al. “Efficient Distributed Quantum Computing”, Proc. Roy. Soc. A, 469.2153, 2013

A quantum algorithm for parallel sieving

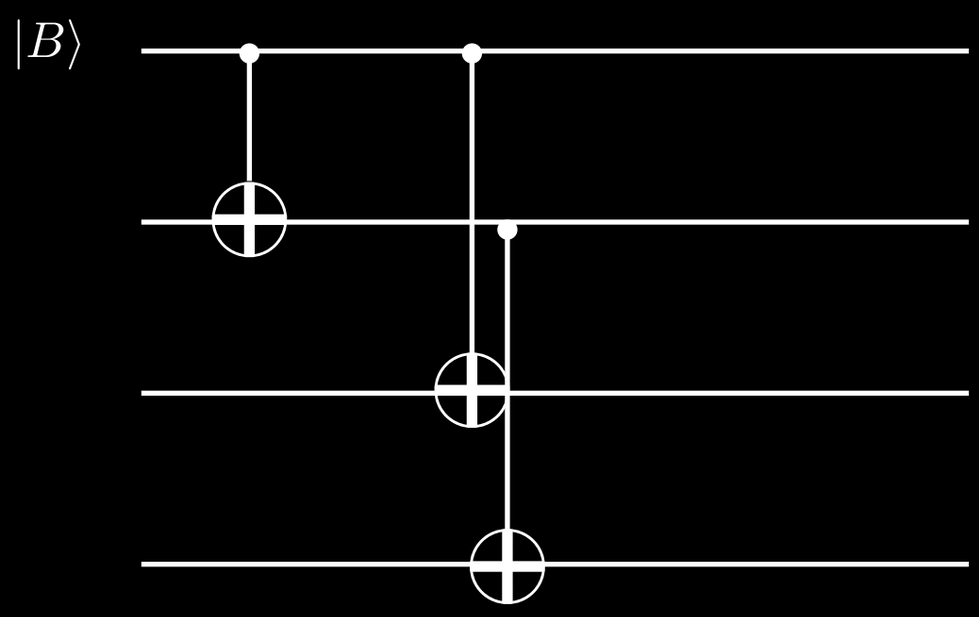
1. Start with N*poly(d) qubits

A quantum algorithm for parallel sieving



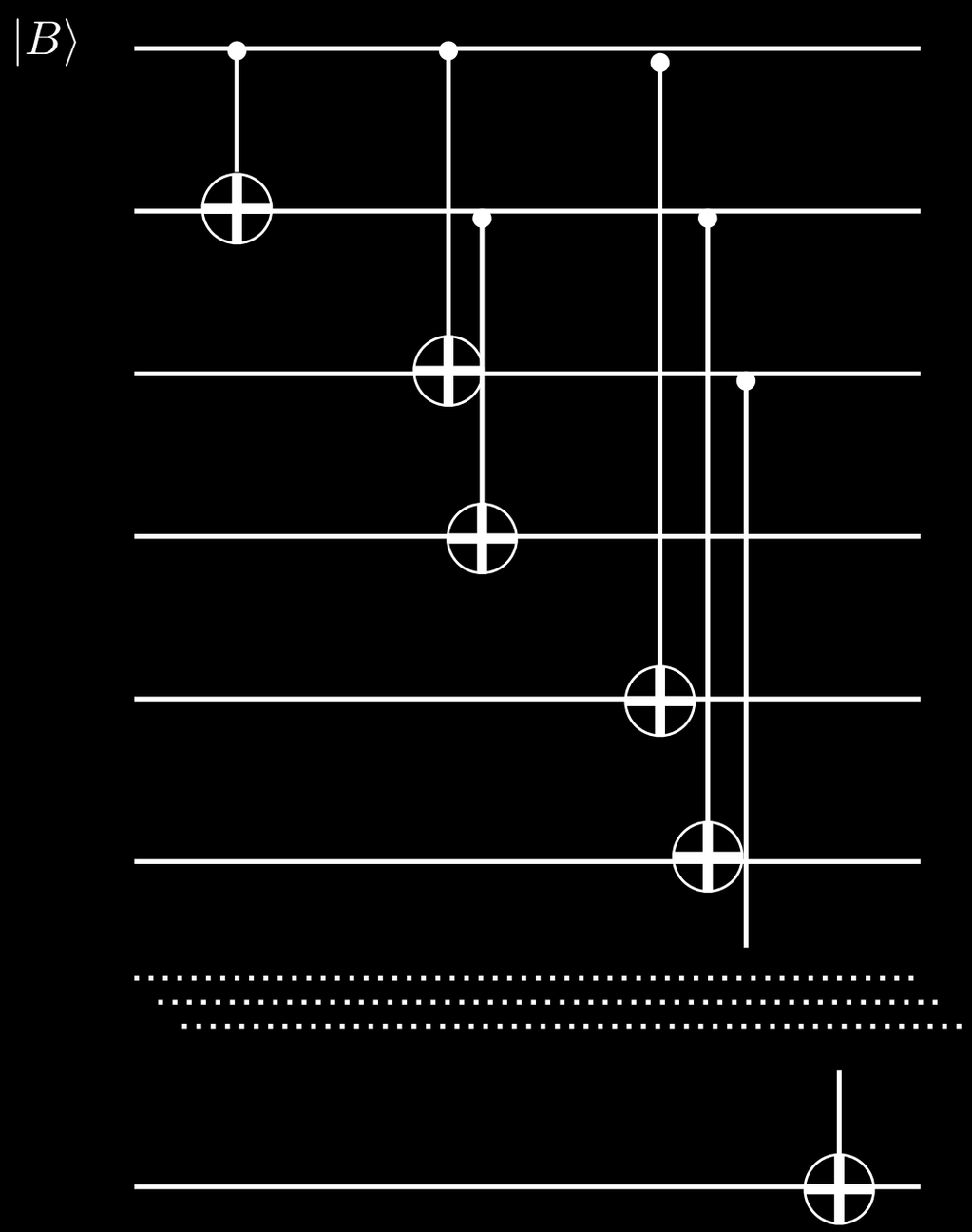
1. Start with N*poly(d) qubits
2. Copy the basis information

A quantum algorithm for parallel sieving



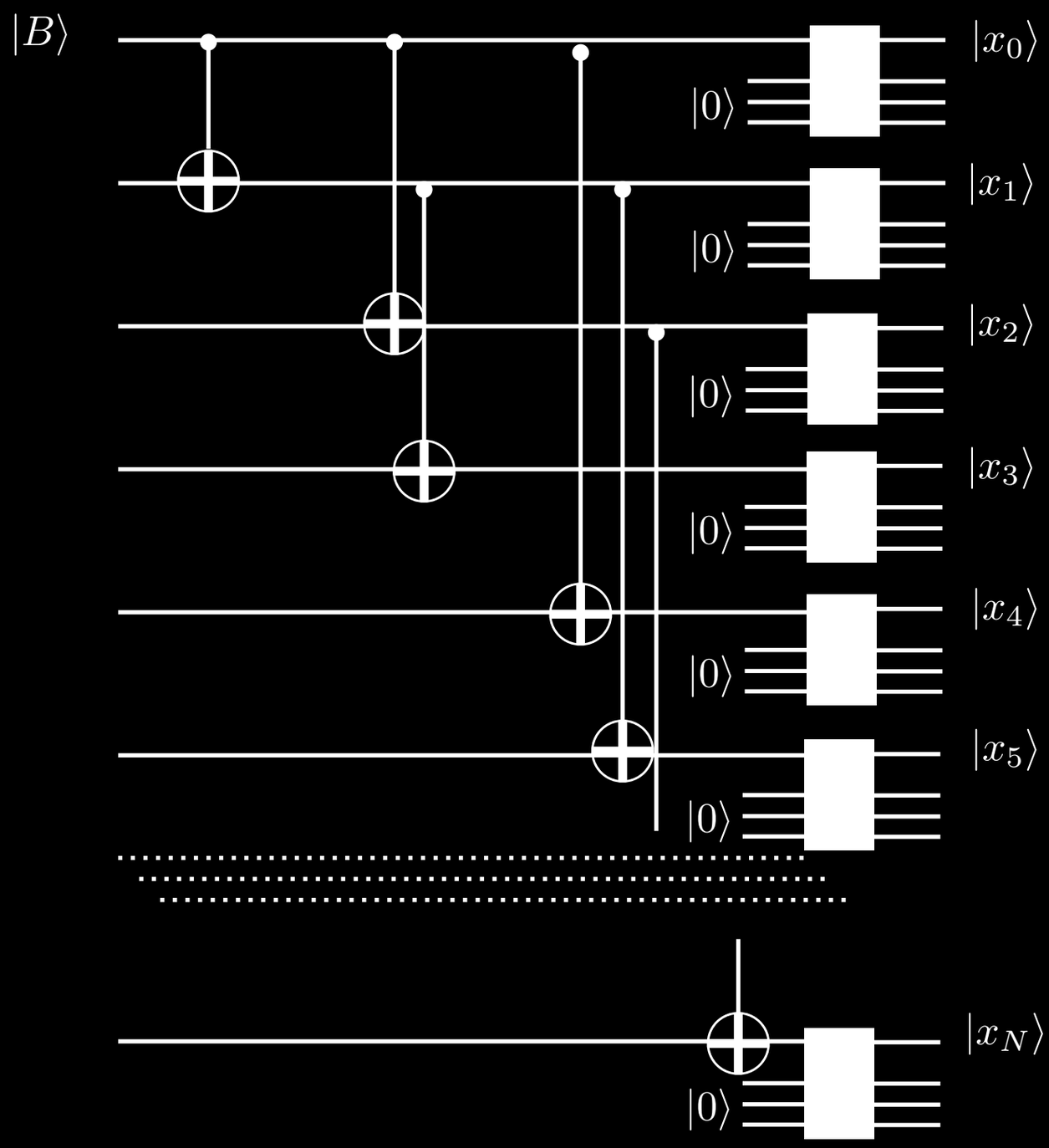
1. Start with N*poly(d) qubits
2. Copy the basis information

A quantum algorithm for parallel sieving



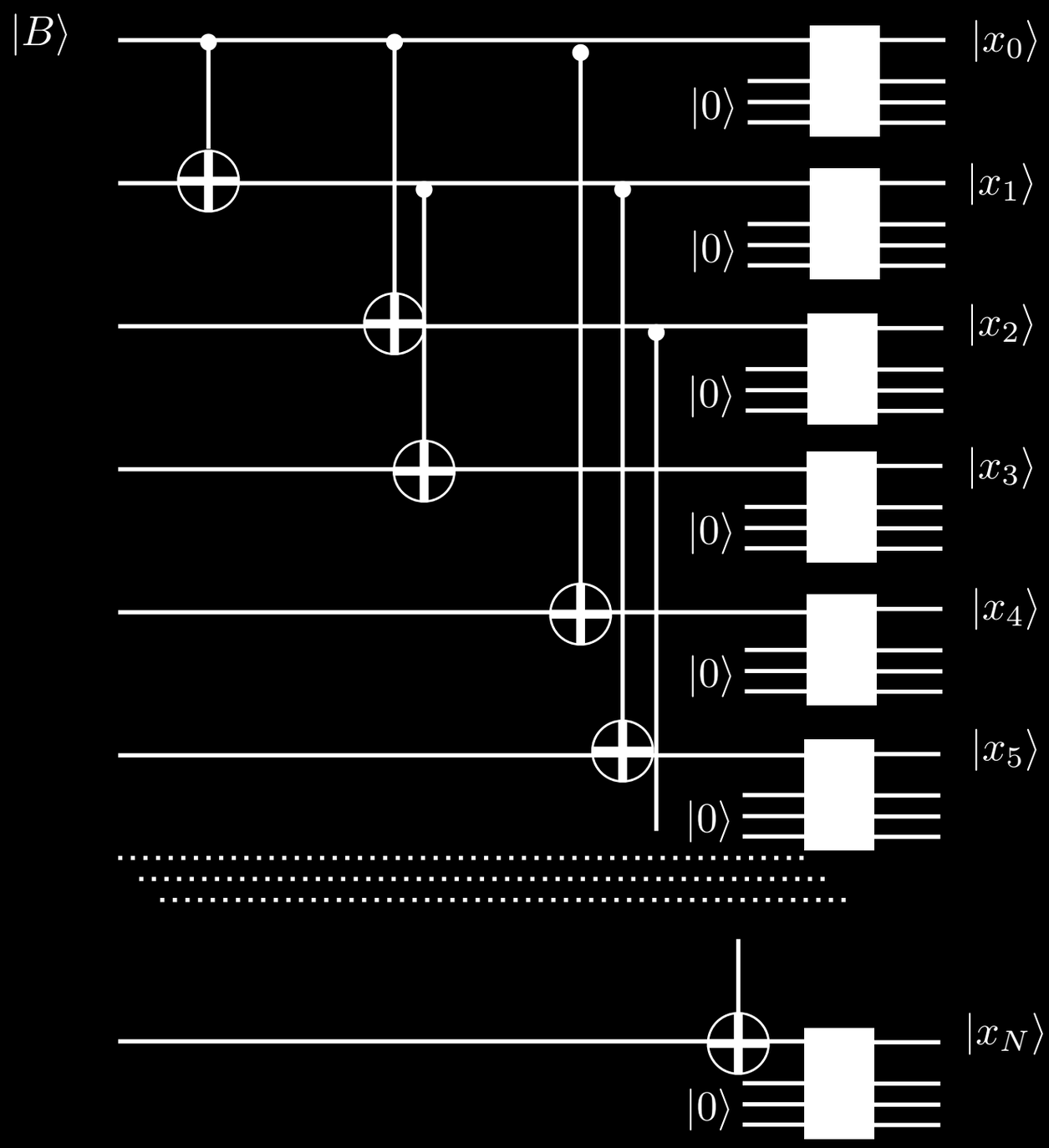
1. Start with N*poly(d) qubits
2. Copy the basis information

A quantum algorithm for parallel sieving



1. Start with N*poly(d) qubits
2. Copy the basis information
3. Sample a vector and normalise it.

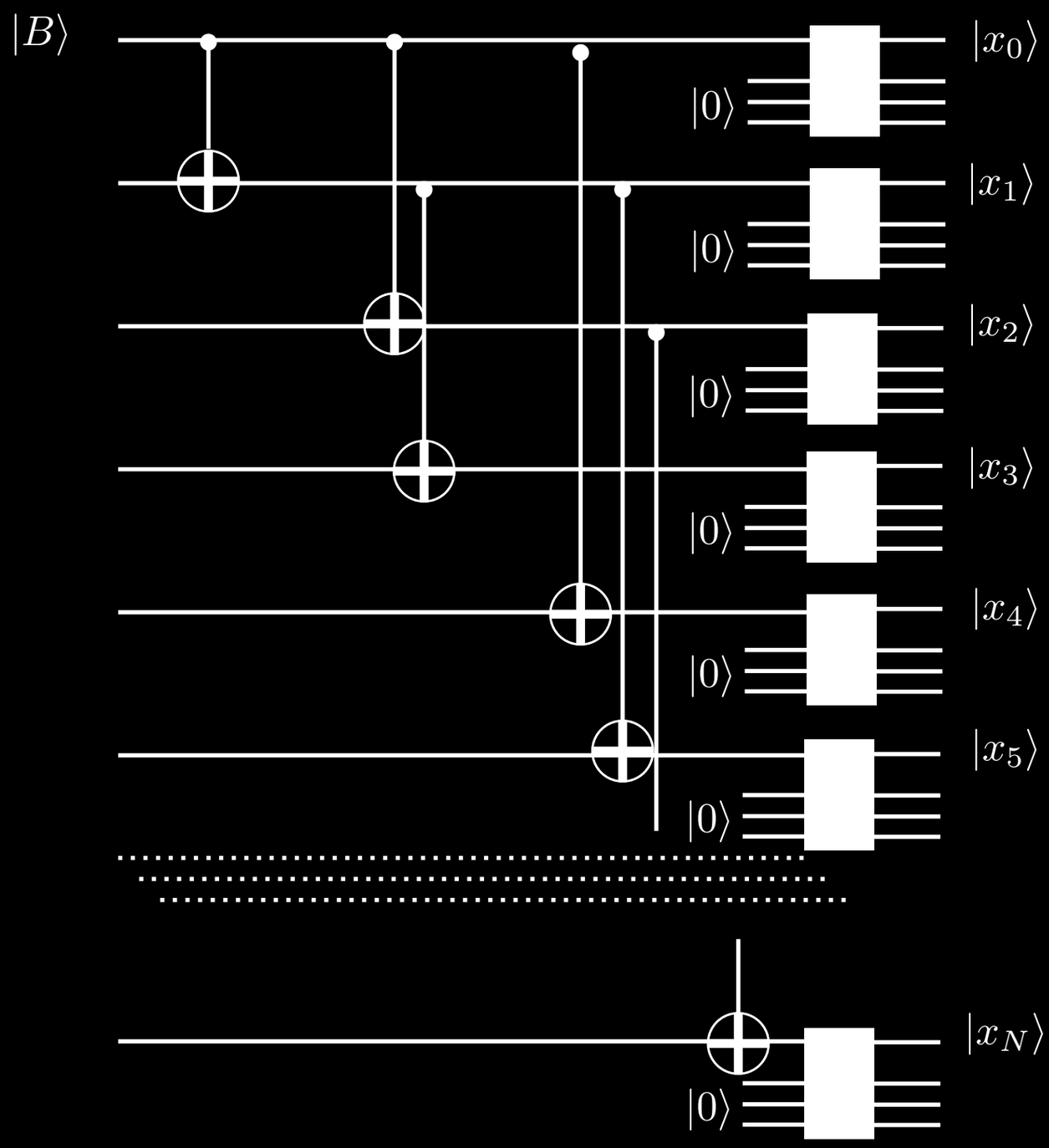
A quantum algorithm for parallel sieving



1. Start with N*poly(d) qubits
2. Copy the basis information
3. Sample a vector and normalise it.

$\tilde{O}(N)$ width
$O(polylog(N))$ depth

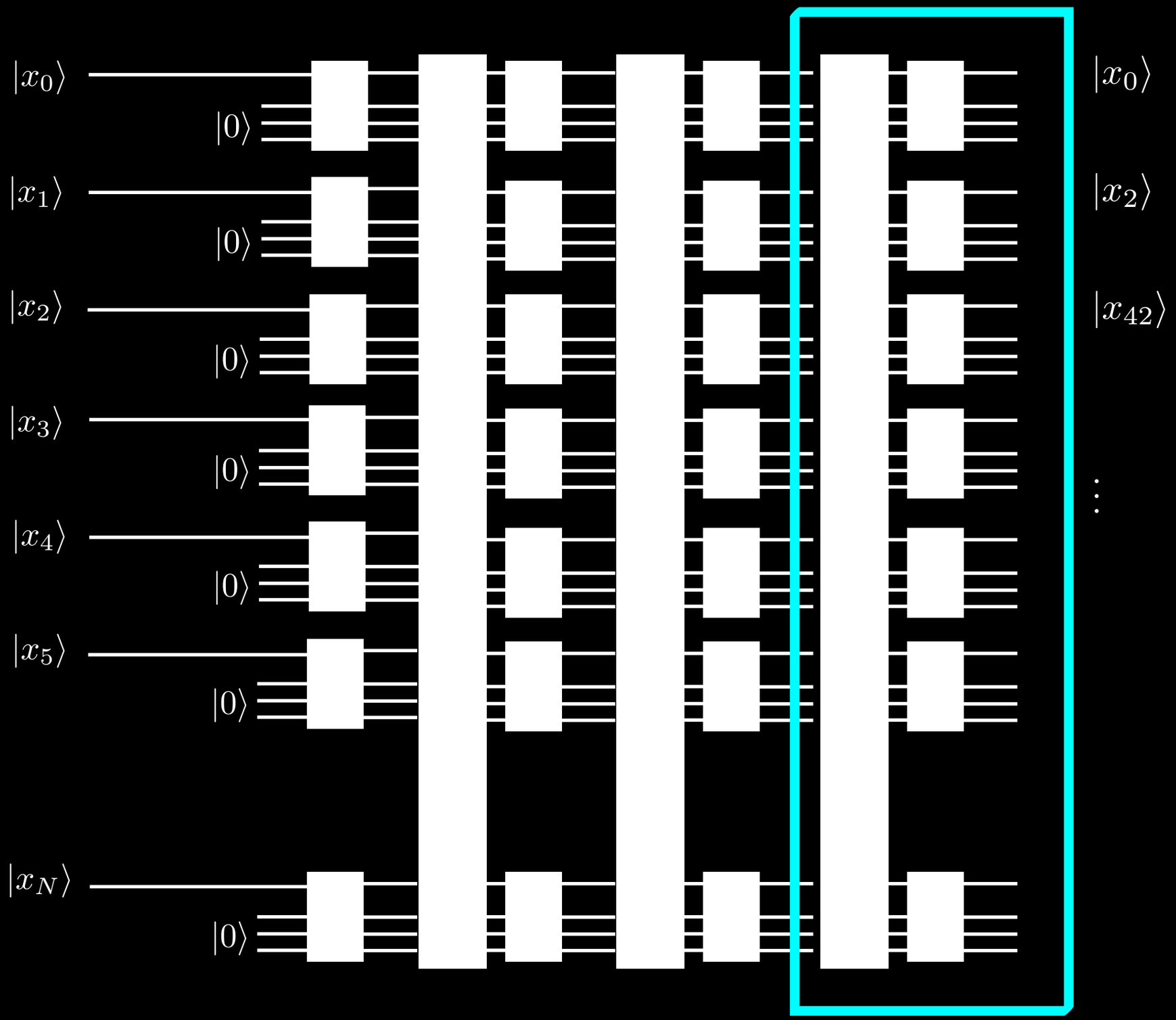
A quantum algorithm for parallel sieving



1. Start with N*poly(d) qubits
2. Copy the basis information
3. Sample a vector and normalise it.

$\tilde{O}(N)$ width
$O(polylog(N))$ depth

A quantum algorithm for parallel sieving



A quantum algorithm for parallel sieving

U_f recognises a solution

$$U_f : |x\rangle |y\rangle \to |x\rangle |y \oplus f(x)\rangle$$

set $|y\rangle = \frac{|0\rangle - |1\rangle}{2}$, and the reduced unitary induces the following transformation

$$|x\rangle \to (-1)^{f(x)} |x\rangle$$

V loads the data

$$V : |j\rangle |0\rangle |y\rangle |x_1 x_2 \dots x_N\rangle \to |j\rangle |x_j\rangle |y\rangle |x_1 x_2 \dots x_N\rangle$$

Quantum Search

$$\begin{aligned}
&|j_1\rangle |0\rangle |y_1\rangle |x_1 \dots x_N\rangle \\
\xrightarrow{V} &|j_1\rangle |x_{j_1}\rangle |y_1\rangle |x_1 \dots x_N\rangle \\
\xrightarrow{\otimes U_{f_i}} &|j_1\rangle |x_{j_1}\rangle |y_1 \oplus f_1(x_{j_1})\rangle |x_1 \dots x_N\rangle \\
\xrightarrow{V} &|j_1\rangle |0\rangle |y_1 \oplus f_1(x_{j_1})\rangle |x_1 \dots x_N\rangle
\end{aligned}$$

Setting $|y_1\rangle = |-\rangle$, and $\mathcal{W} = VUV$ the reduced unitary becomes

$$\mathcal{W} : |j_1\rangle \to (-1)^{f_1(x_{j_1})} |j_1\rangle .$$

The unitary $\mathcal{W}$ can be used as an oracle for Grover's algorithm. Further, using $\mathcal{W}$ as an oracle, requires a circuit width of $\Omega(Nd)$, and depth $\Omega(N(logN + D_f)$ to find one solution with high probability.

A quantum algorithm for parallel sieving

Define $f(x,y) = 1$ if $|< x, y >| \geq 1/2; 0$ otherwise.

$$U_f : |x,y\rangle |b\rangle \rightarrow |x,y\rangle |b \oplus f(x,y)\rangle$$

Now for each 'wire' i, define $f_i(y) = 1$ if $|< x_i, y >| \geq 1/2; 0 otherwise$

$$U_{f_i} : |y\rangle |b\rangle \rightarrow |y\rangle |b \oplus f(x_i,y)\rangle$$

$$V : |j_1 \dots j_N\rangle |y_1 \dots y_N\rangle |x_1 \dots x_N\rangle \rightarrow |j_1 \dots j_N\rangle |y_1 + x_{j_1} \dots y_N + x_{j_N}\rangle |x_1 x_2 \dots x_N\rangle$$

A quantum algorithm for parallel sieving

Parallel Quantum Search

$$
\begin{aligned}
& |j_1 \dots j_N\rangle \, |0 \dots 0\rangle \, |y_1 \dots y_N\rangle \, |x_1 \dots x_N\rangle \\
\xrightarrow{V} & |j_1 \dots j_N\rangle \, |x_{j_1} \dots x_{j_N}\rangle \, |y_1 \dots y_N\rangle \, |x_1 \dots x_N\rangle \\
\xrightarrow{\otimes U_{f_i}^{\otimes N}} & |j_1 \dots j_N\rangle \, |x_{j_1} \dots x_{j_N}\rangle \, |y_1 \oplus f_1(x_{j_1}) \dots y_N \oplus f_N(x_{j_N})\rangle \, |x_1 \dots x_N\rangle \\
\xrightarrow{V} & |j_1 \dots j_N\rangle \, |0 \dots 0\rangle \, |y_1 \oplus f_1(x_{j_1}) \dots y_N \oplus f_N(x_{j_N})\rangle \, |x_1 \dots x_N\rangle
\end{aligned}
$$

Setting $|y_i\rangle = |-\rangle$, for all i, $\mathcal{W} = V U_{f_i}^{\otimes N} V$ the reduced unitary becomes

$$
\mathcal{W} : |j_1 \dots j_N\rangle \to (-1)^{f_1(x_{j_1})} |j_1\rangle \dots (-1)^{f_N(x_{j_N})} |j_N\rangle \, .
$$

The unitary $\mathcal{W}$ can be implemented using a uniform family of circuits of width $N \cdot (logN + d) = N \cdot polylogN$ and depth $O(logNlog(dlogN) + poly(d)) = polylog(N)$. Define R to be a rotation along $|0\rangle$ and run $R \circ \mathcal{W}, \Theta(\sqrt{N})$ times, on the state

$$
\sum_{j_1, \dots j_N} |j_1 \dots j_N\rangle \, |0 \dots 0\rangle \, |- \dots -\rangle \, |x_1 \dots x_N\rangle
$$

A quantum algorithm for parallel sieving

Each wire, i, that had previously sample x_i, thus finds a x_i', such that x_i and x_i' are remarkable close, and computes the difference and sets that as the new vector $y_i = min\{x_i \pm x_i'\}$.
x_i is stored in register $L1[i]$
x_i' is stored in register $L2[i]$
y_i is stored in register $L3[i]$

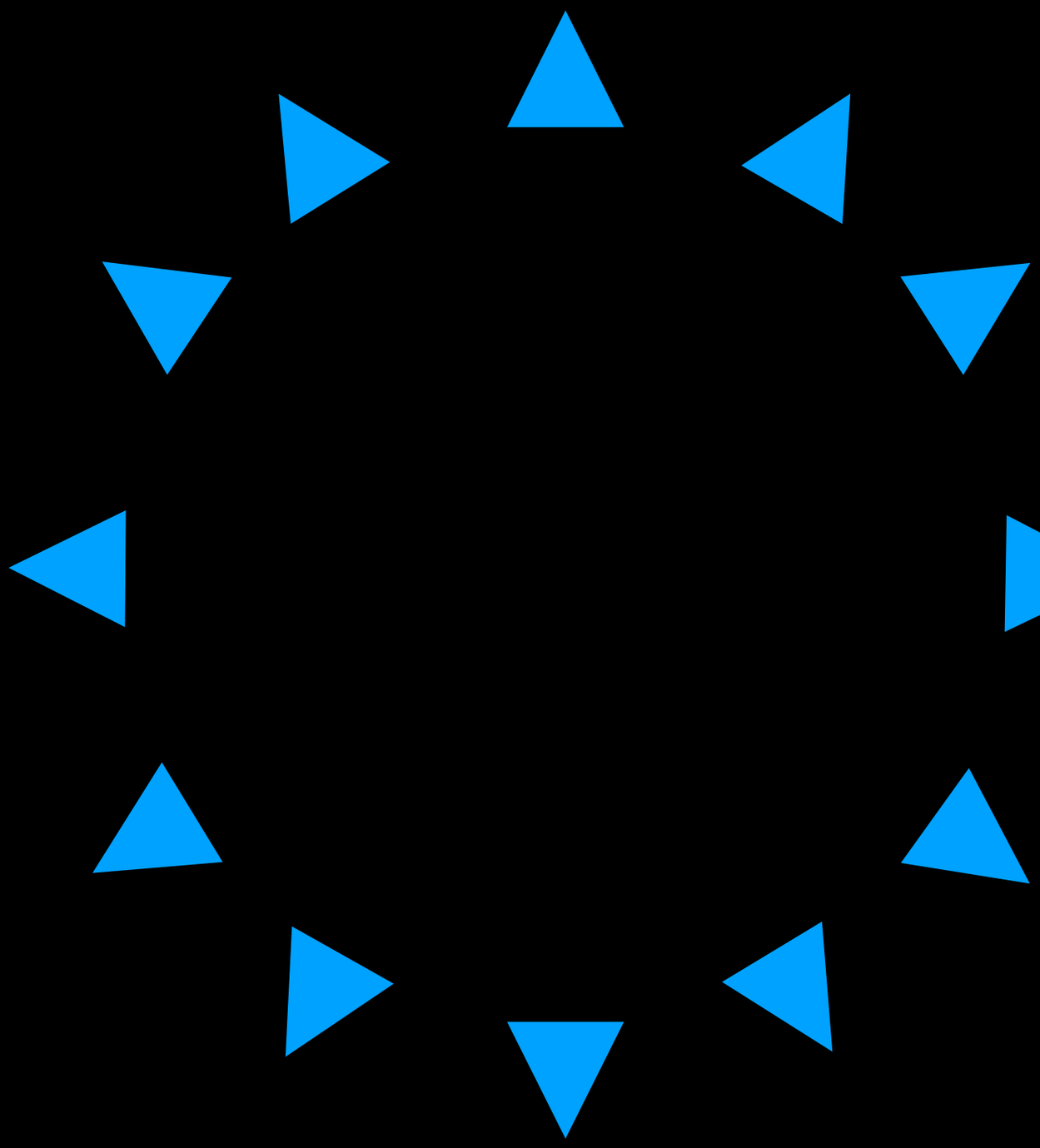
Swap registers L3 and L1. And repeat this process $poly(d)$ times.

Search through L3, and output a vector shorter than the target norm (Euclidean norm less than λ).

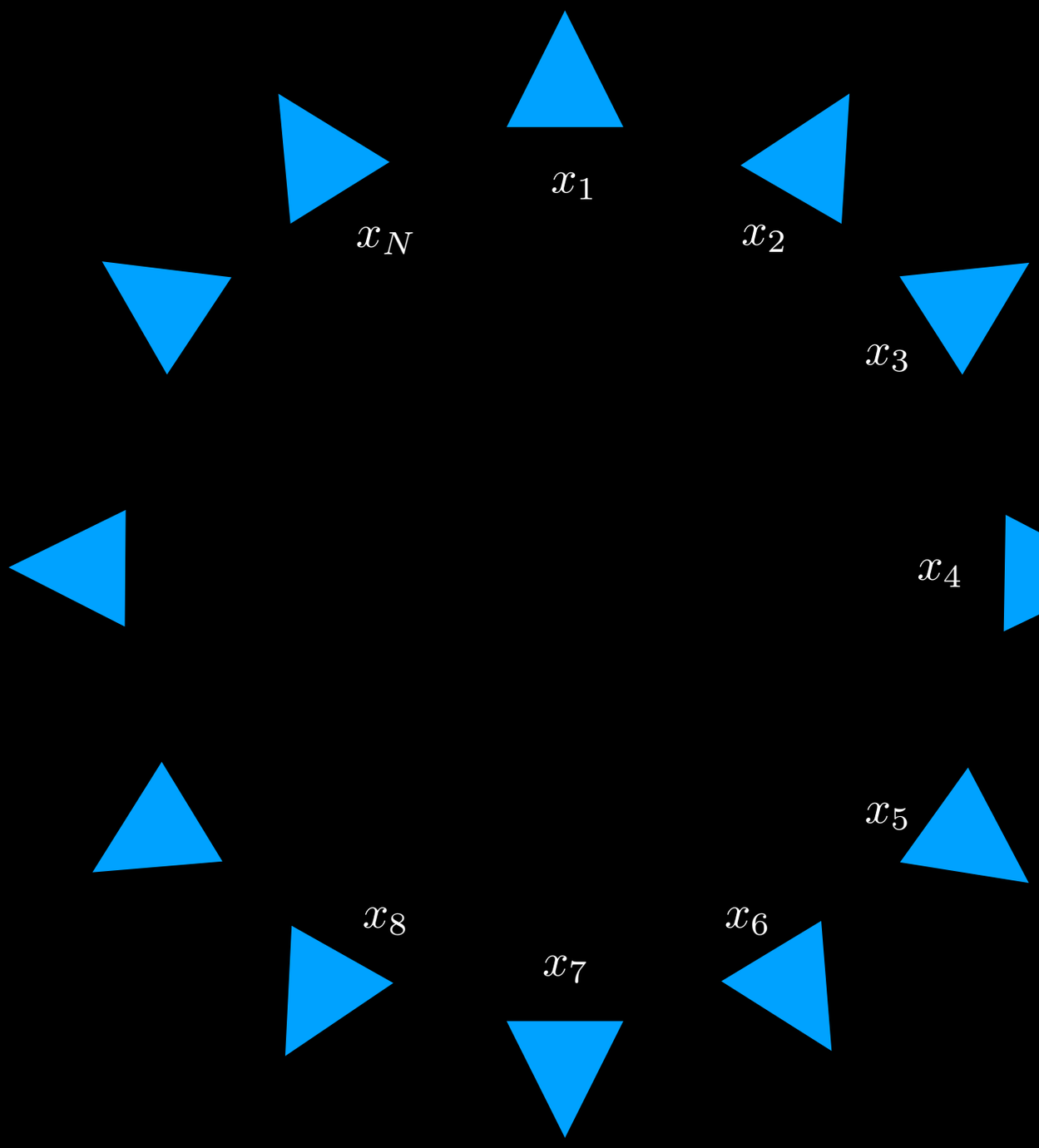
This algorithm can be implemented with a quantum circuit of depth $\tilde{O}(\sqrt{N})$ and width $\tilde{O}(N)$, and recall $N = 2^{0.2075d}$, thus it take the quantum circuit $2^{0.1037d+o(d)}$ time steps and $2^{0.2075d+o(d)}$ memory

A classical algorithm for distributed sieving

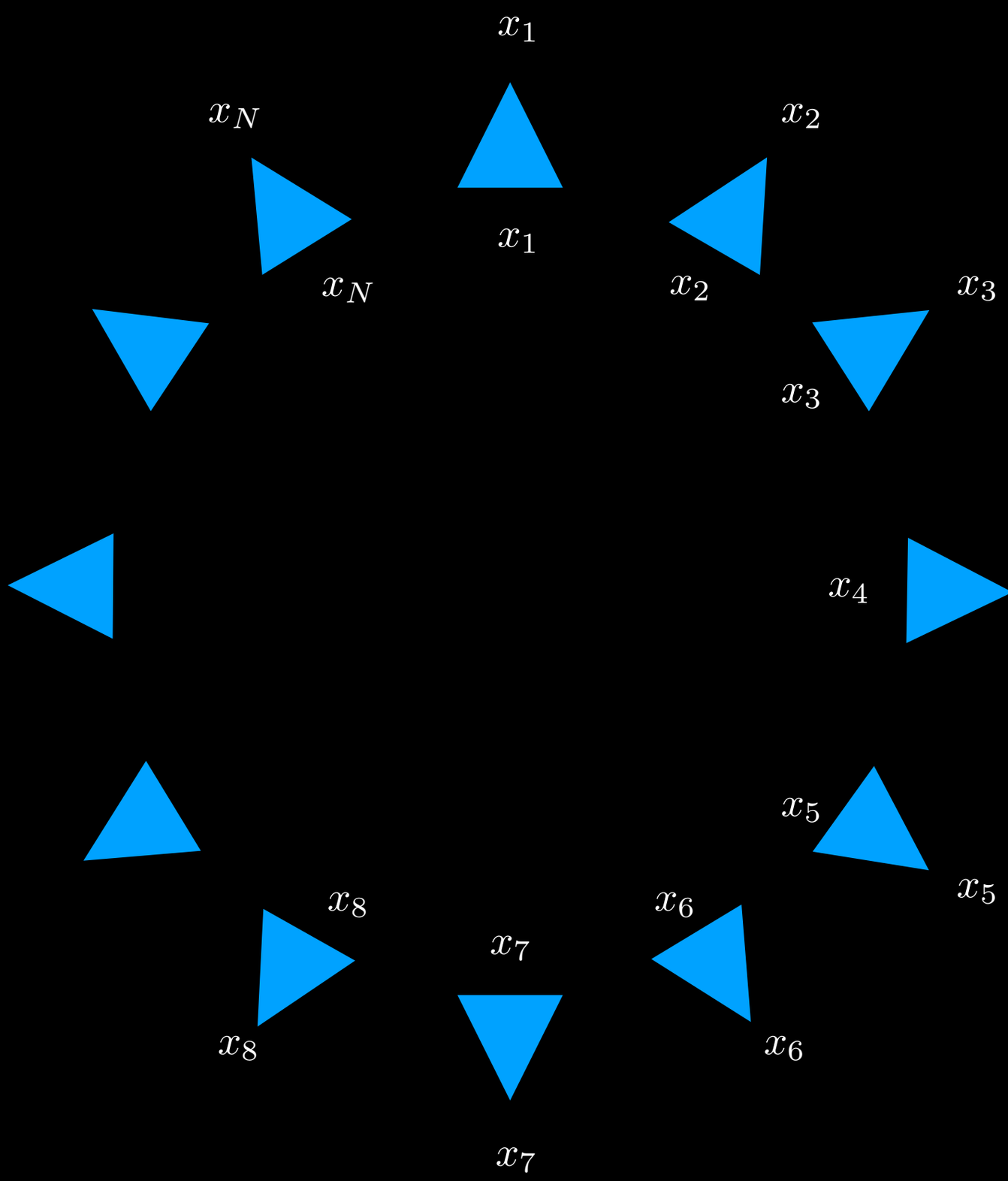
A classical algorithm for distributed sieving



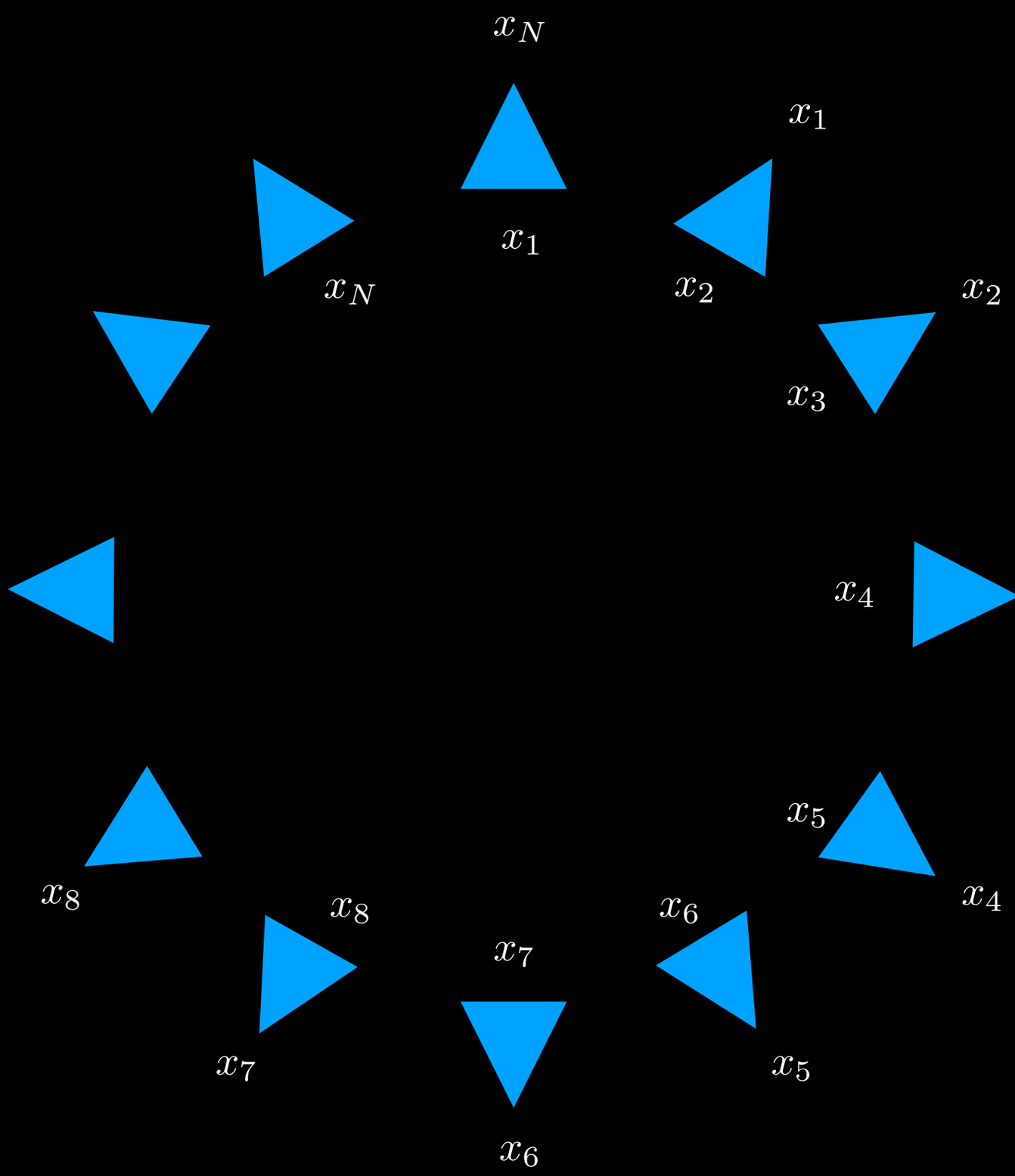
A classical algorithm for distributed sieving



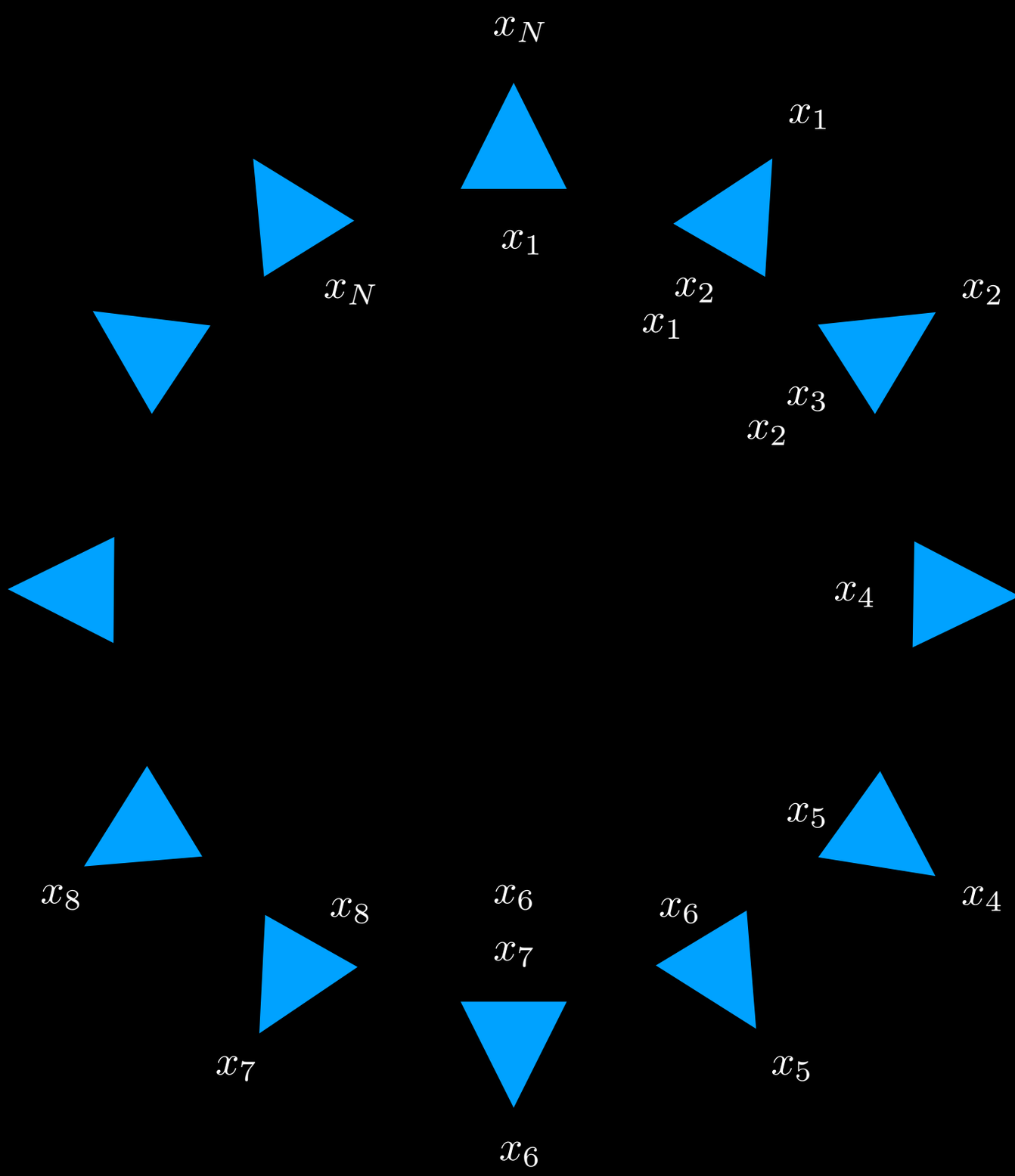
A classical algorithm for distributed sieving



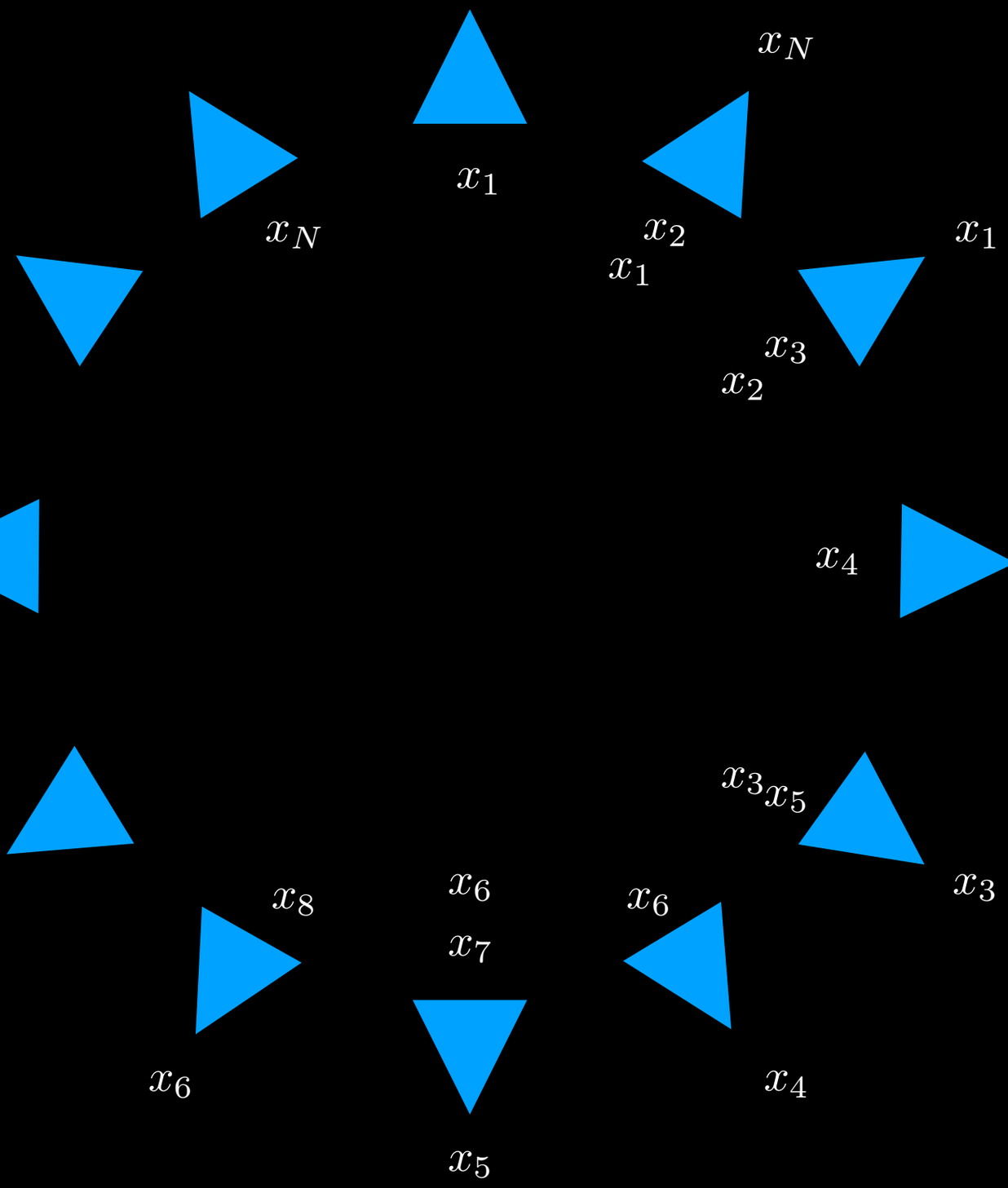
A classical algorithm for distributed sieving



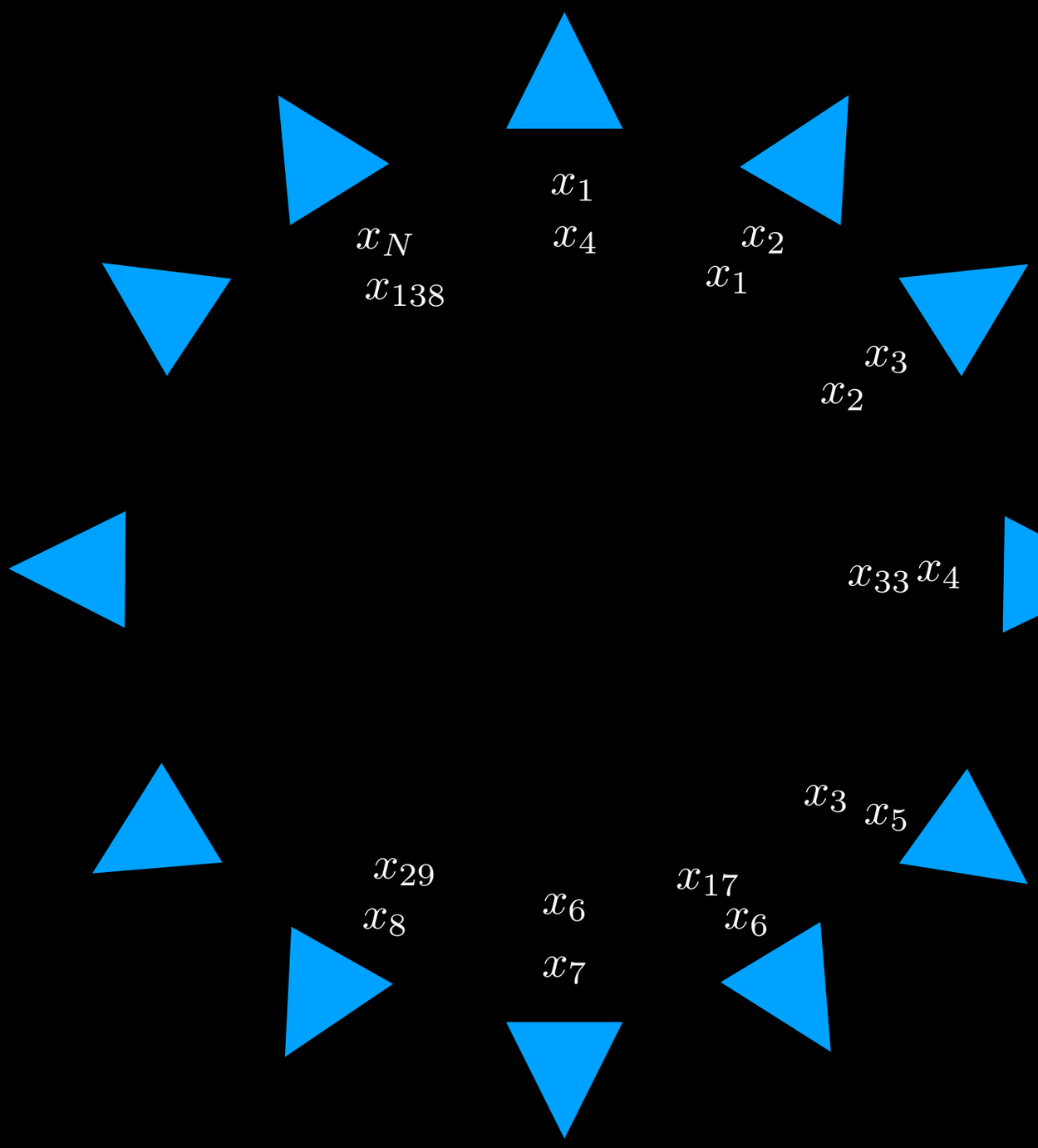
A classical algorithm for distributed sieving



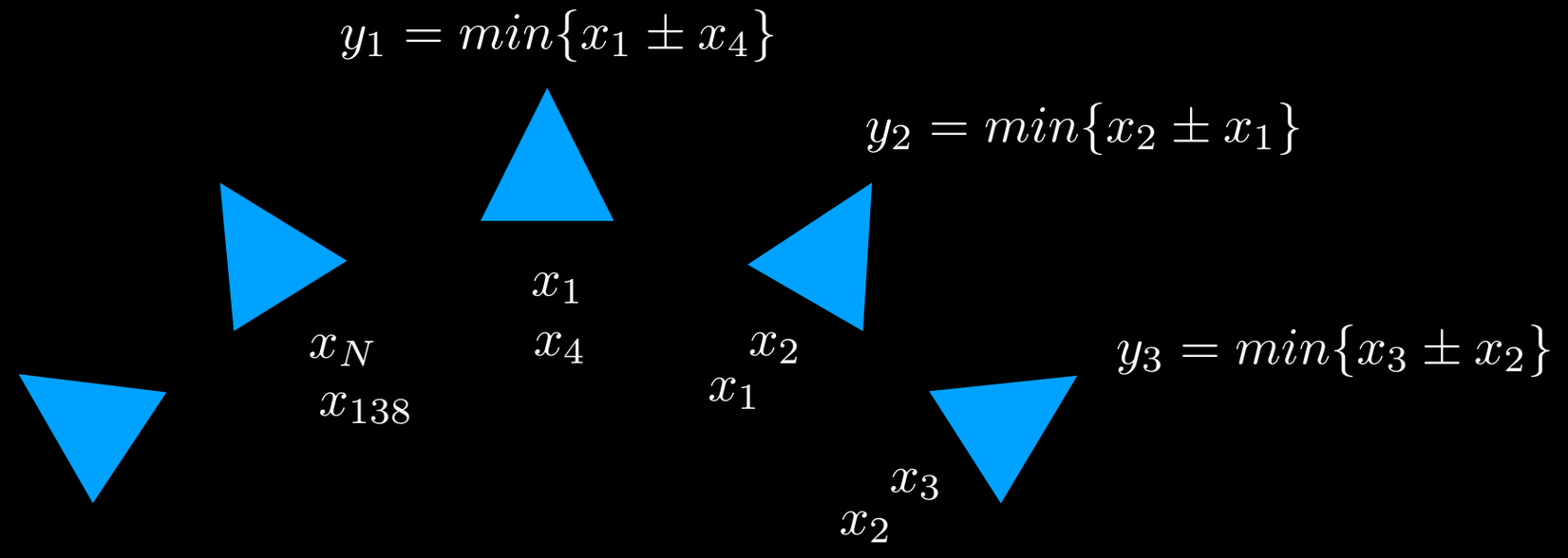
A classical algorithm for distributed sieving



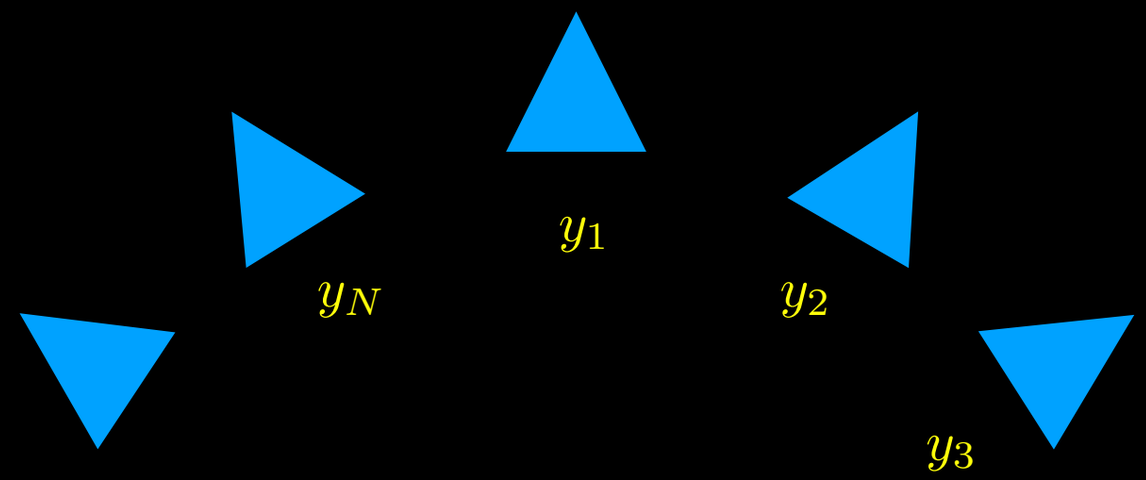
A classical algorithm for distributed sieving

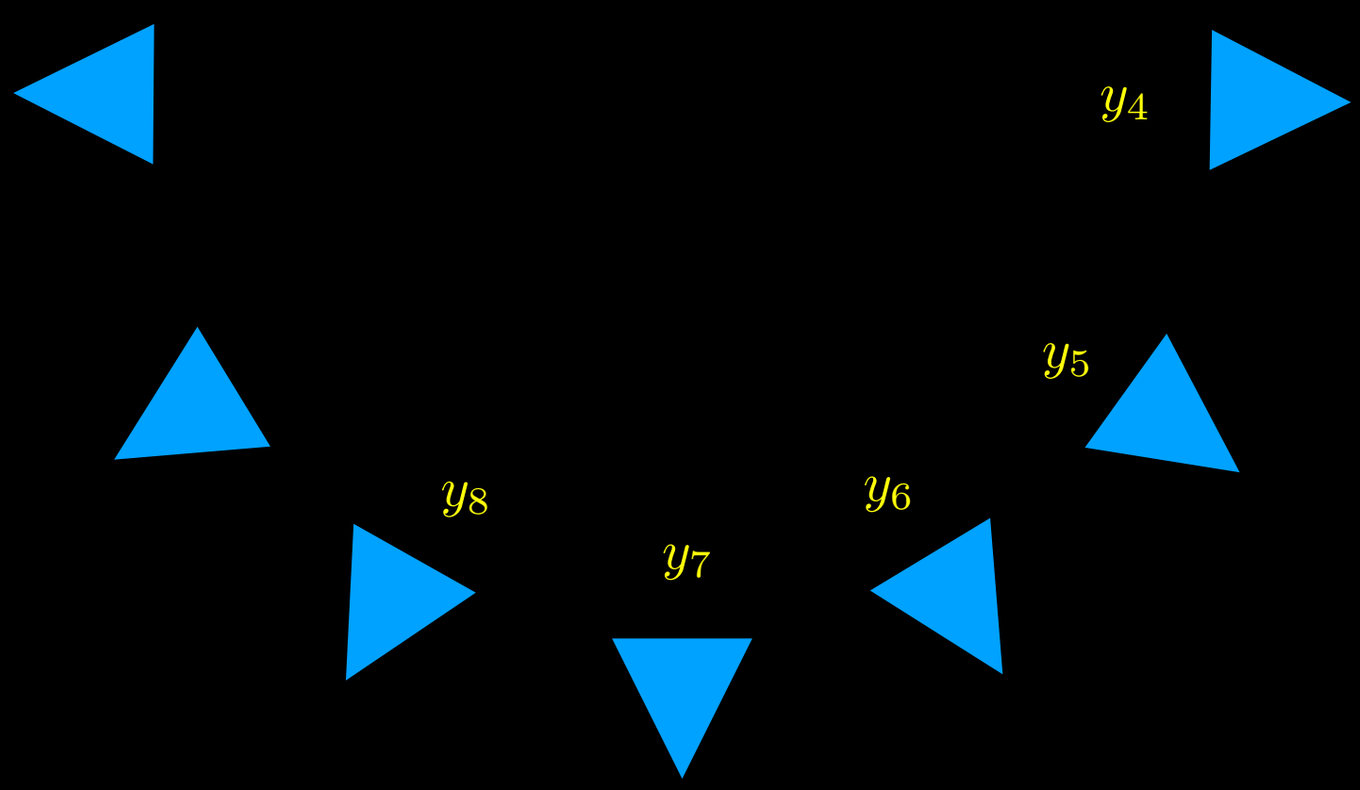


A classical algorithm for distributed sieving



A classical algorithm for distributed sieving

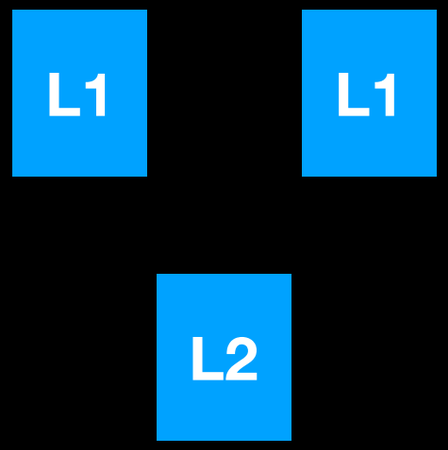




This classical algorithm for distributed sieving can be implemented in $2^{0.2075d+o(d)}$ time steps and $2^{0.2075d+o(d)}$ memory.

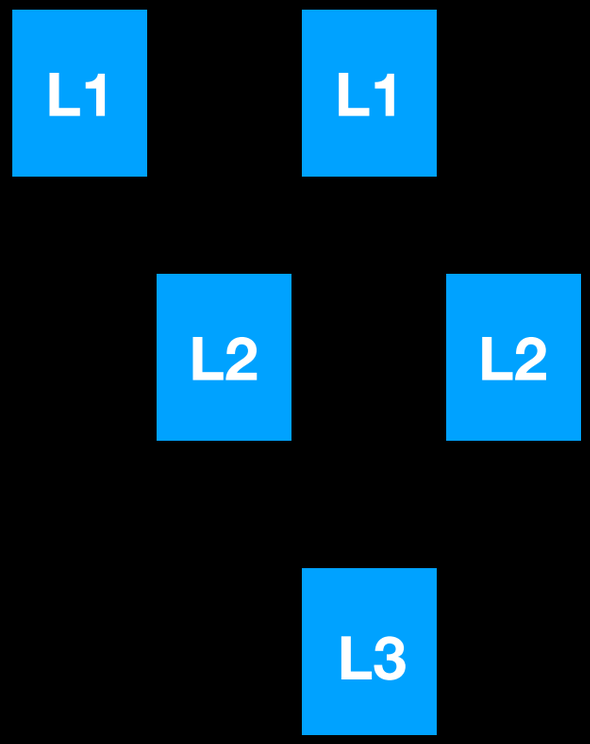
Compare this to the quantum algorithm for parallel sieving that can be implemented in $2^{0.1037d+o(d)}$ time steps and $2^{0.2075d+o(d)}$ memory.

Quantum Algorithms for k-sieve



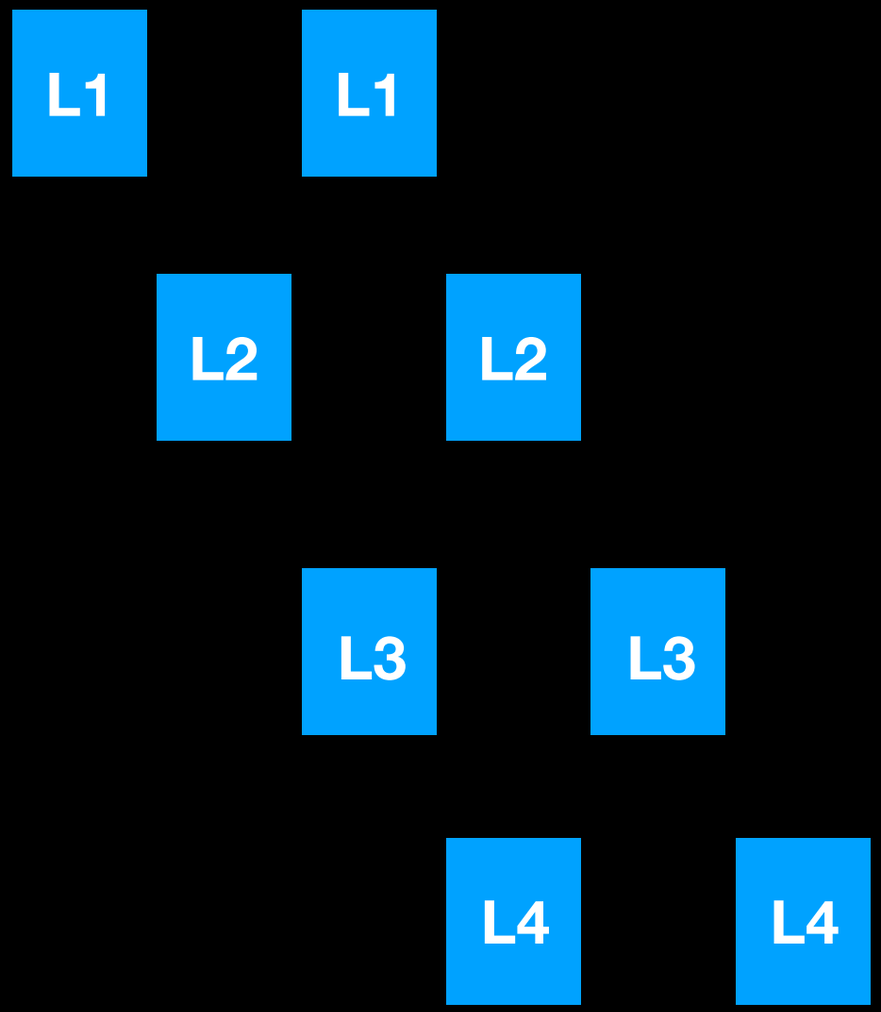
2-sieve

For $x_i, x_j \in L_1$,
the pair (x_i, x_j) is 'good' iff $| < x_i, x_j > | \geq \frac{1}{2}$



2-sieve

For $x_i, x_j \in L_1$,
the pair (x_i, x_j) is 'good' iff $| < x_i, x_j > | \geq \frac{1}{2}$



2-sieve

For $x_i, x_j \in L_1$,
the pair (x_i, x_j) is 'good' iff $| < x_i, x_j > | \geq \frac{1}{2}$

k-sieve

For k=3, $x_i, x_j, x_k \in L_1$,
the tuple (x_i, x_j, x_k) is 'good' iff
$| < x_i, x_j > | \approx \frac{1}{3}$
$| < x_j, x_k > | \approx \frac{1}{3}$
$| < x_k, x_i > | \approx \frac{1}{3}$

L1 L1 L1 L1

L2 L2 L2 L2

L3

k-sieve

For k=3, $x_i, x_j, x_k \in L_1$,
the tuple (x_i, x_j, x_k) is 'good' iff
$|< x_i, x_j >| \approx \frac{1}{3}$
$|< x_j, x_k >| \approx \frac{1}{3}$
$|< x_k, x_i >| \approx \frac{1}{3}$

L1 L1 L1 L1

L2 L2 L2 L2

L3 L3 L3 L3

L

k-sieve

For k=3, $x_i, x_j, x_k \in L_1$,
the tuple (x_i, x_j, x_k) is 'good' iff
$|< x_i, x_j >| \approx \frac{1}{3}$
$|< x_j, x_k >| \approx \frac{1}{3}$
$|< x_k, x_i >| \approx \frac{1}{3}$

Sieving and k-clique listing

x_1

x_3

x_{138}

x_2

x_4

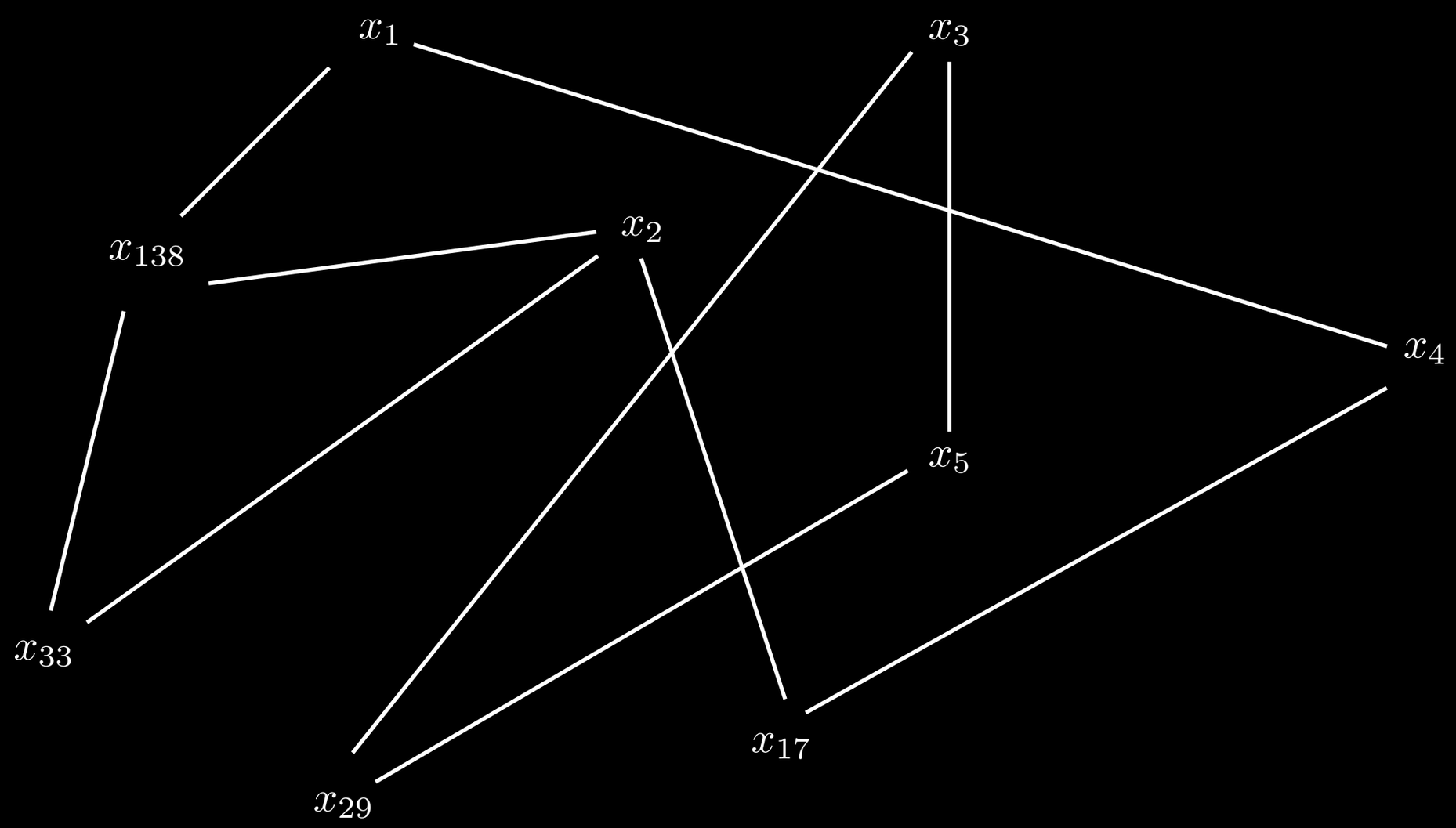
x_5

x_{33}

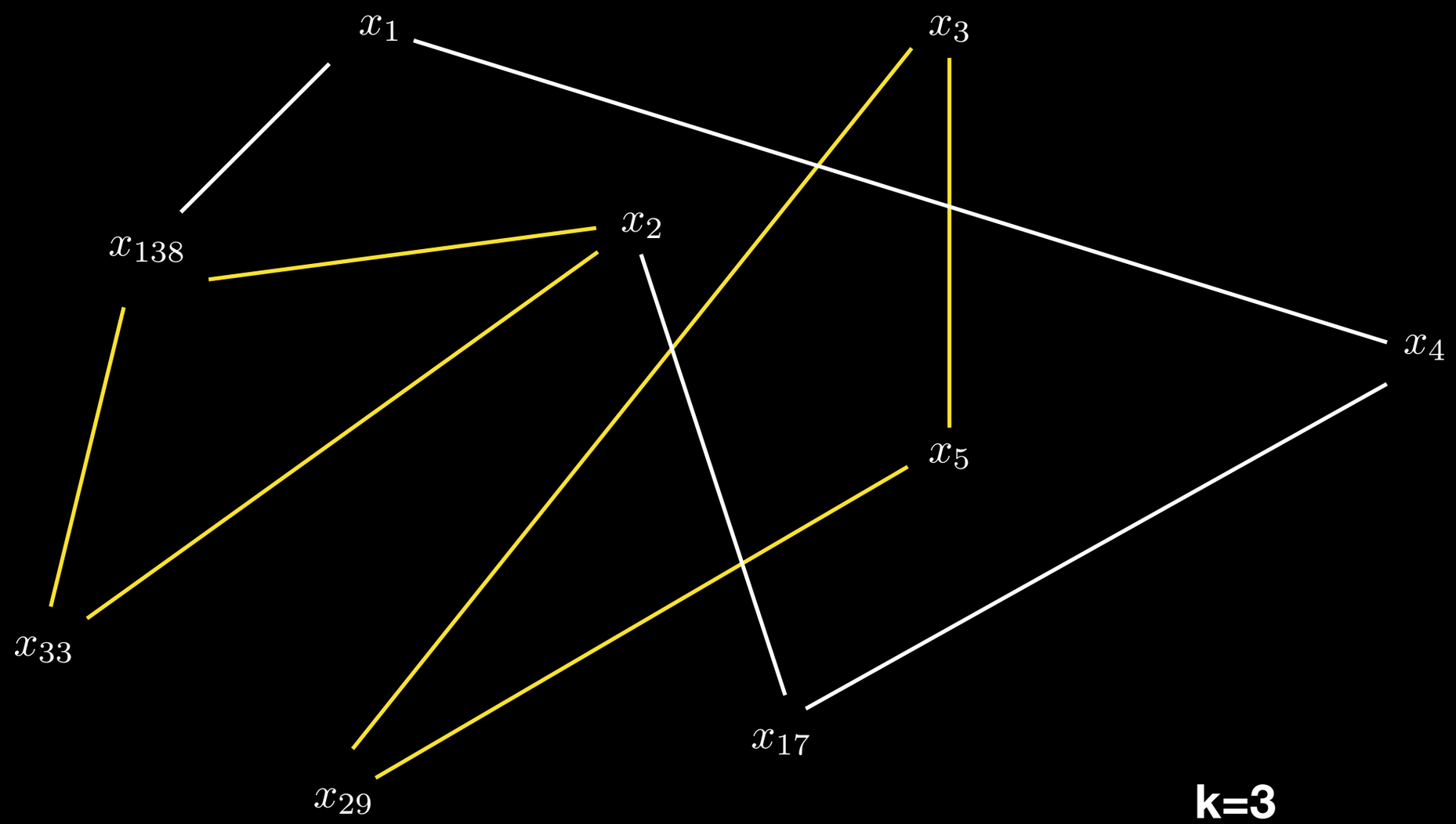
x_{17}

x_{29}

Sieving and k-clique listing



Sieving and k-clique listing



k=3

(x_{29}, x_5, x_3)

(x_{33}, x_2, x_{138})

Sieving and 3-clique listing

Triangle finder:

Buhrman et al. Quantum Algorithms for Element Distinctness,
SIAM J. of Computing, 34(6), 1324-1330, 2005

1. Use quantum search to find an edge $(x_i, x_j) \in E$
2. Use quantum search to find a node x_k, such that (x_i, x_j, x_k) is a triangle.
3. Apply Amplitude Amplification on steps 1-2.

Sieving and 3-clique listing

Triangle finder:

Buhrman et al. Quantum Algorithms for Element Distinctness, SIAM J. of Computing, 34(6), 1324-1330, 2005

1. Use quantum search to find an edge $(x_i, x_j) \in E$
2. Use quantum search to find a node x_k, such that (x_i, x_j, x_k) is a triangle.
3. Apply Amplitude Amplification on steps 1-2.

Amplitude Amplification is a generic way to get speedups for any probabilistic algorithm. Given a Boolean function $f : \{0,1\} \to \{0,1\}$, suppose we have an algorithm that finds a solution $f(x) = 1$ with probability p when acting on the initial state $|0\rangle$.

Classically we would have to execute the algorithm roughly $\frac{1}{p}$ times before finding a solution.

Using Amplitude Amplification, we one need $\frac{1}{\sqrt{p}}$ repetitions.

Sieving and 3-clique listing

Triangle finder:

Buhrman et al. Quantum Algorithms for Element Distinctness,
SIAM J. of Computing, 34(6), 1324-1330, 2005

1. Use quantum search to find an edge $(x_i, x_j) \in E$
2. Use quantum search to find a node x_k, such that (x_i, x_j, x_k) is a triangle.
3. Apply Amplitude Amplification on steps 1-2.

In the graph, corresponding to 3-sieve, we set $N = \tilde{O}((3\sqrt{3}/4)^{d/2})$,
$m = \tilde{O}(n^2(8/9)^{d/2})$, and it is expected there are $\Theta(N)$ triangles.
Step 1, needs $\Theta(\sqrt{(\frac{N^2}{m})})$ time steps.
Step 2, needs $\Theta(\sqrt{N})$ time steps
Step 3, needs $\Theta(\sqrt{\frac{m}{N}})$ time steps.

Total cost: $\Theta(\sqrt{m})$ to find one triangle

Sieving and 3-clique listing

Triangle finder:

Buhrman et al. Quantum Algorithms for Element Distinctness,
SIAM J. of Computing, 34(6), 1324-1330, 2005

1. Use quantum search to find an edge $(x_i, x_j) \in E$
2. Use quantum search to find a node x_k, such that (x_i, x_j, x_k) is a triangle.
3. Apply Amplitude Amplification on steps 1-2.

In the graph, corresponding to 3-sieve, we set $N = \tilde{O}((3\sqrt{3}/4)^{d/2})$,
$m = \tilde{O}(n^2(8/9)^{d/2})$, and it is expected there are $\Theta(N)$ triangles.
Step 1, needs $\Theta(\sqrt{(\frac{N^2}{m})})$ time steps.
Step 2, needs $\Theta(\sqrt{N})$ time steps
Step 3, needs $\Theta(\sqrt{\frac{m}{N}})$ time steps.

Total cost: $\Theta(\sqrt{m})$ to find one triangle

Repeat this $N * logN$ times, to list all triangles.

Sieving and 3-clique listing

Triangle finder:

Buhrman et al. Quantum Algorithms for Element Distinctness,
SIAM J. of Computing, 34(6), 1324-1330, 2005

1. Use quantum search to find an edge $(x_i, x_j) \in E$
2. Use quantum search to find a node x_k, such that (x_i, x_j, x_k) is a triangle.
3. Apply Amplitude Amplification on steps 1-2.

In the graph, corresponding to 3-sieve, we set $N = \tilde{O}((3\sqrt{3}/4)^{d/2})$,
$m = \tilde{O}(n^2(8/9)^{d/2})$, and it is expected there are $\Theta(N)$ triangles.
Step 1, needs $\Theta(\sqrt{(\frac{N^2}{m})})$ time steps.
Step 2, needs $\Theta(\sqrt{N})$ time steps
Step 3, needs $\Theta(\sqrt{\frac{m}{N}})$ time steps.

Total cost: $\Theta(\sqrt{m})$ to find one triangle

Repeat this $N * logN$ times, to list all triangles.

Each sieving step needs $\tilde{O}(\sqrt{m} * N) = 2^{0.3349d+o(d)}$ time steps.
And we need $\tilde{O}(N) = 2^{0.1887d+o(d)}$ classical memory*.
And $poly(d)$ quantum memory.

Sieving and k-clique listing

K-Clique finder:

1. Use quantum search to find an edge $(x_i, x_j) \in E$
2. Use quantum search to find a node x_k, such that (x_i, x_j, x_k) is a triangle.
3. Use quantum search to find a node x_l, such that (x_i, x_j, x_k, x_l) is a 4-clique.

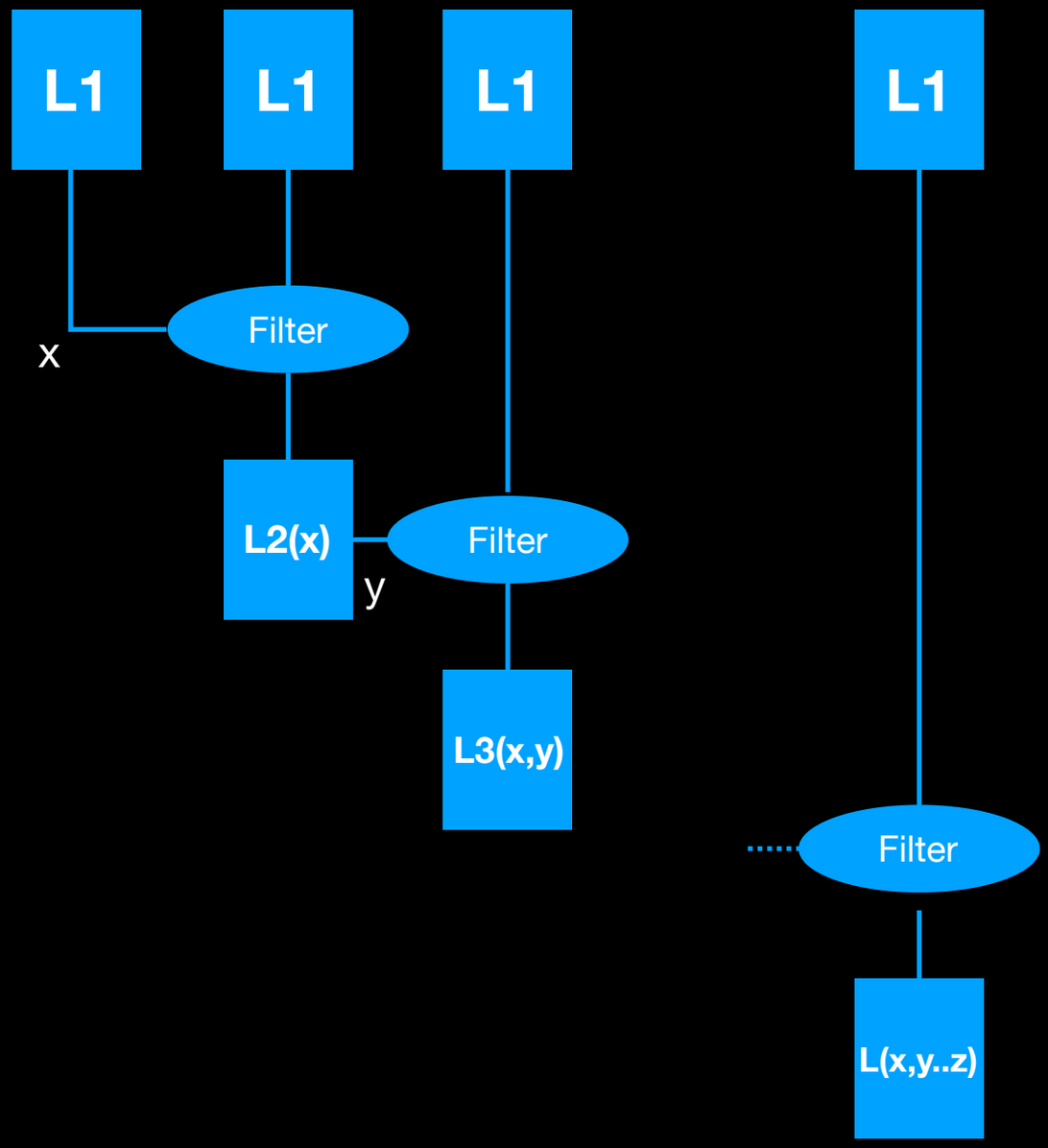
   $\vdots$

k-1. Use quantum search to find a node x_c, such that $(x_i, x_j, x_k, x_l \ldots x_c)$ is a k-clique.

k. Apply Amplitude Amplification on steps $1 - (k-1)$.

If there are N expected k-Cliques, Repeat the k-Clique finder algorithm $N * logN$ times, to list them all.

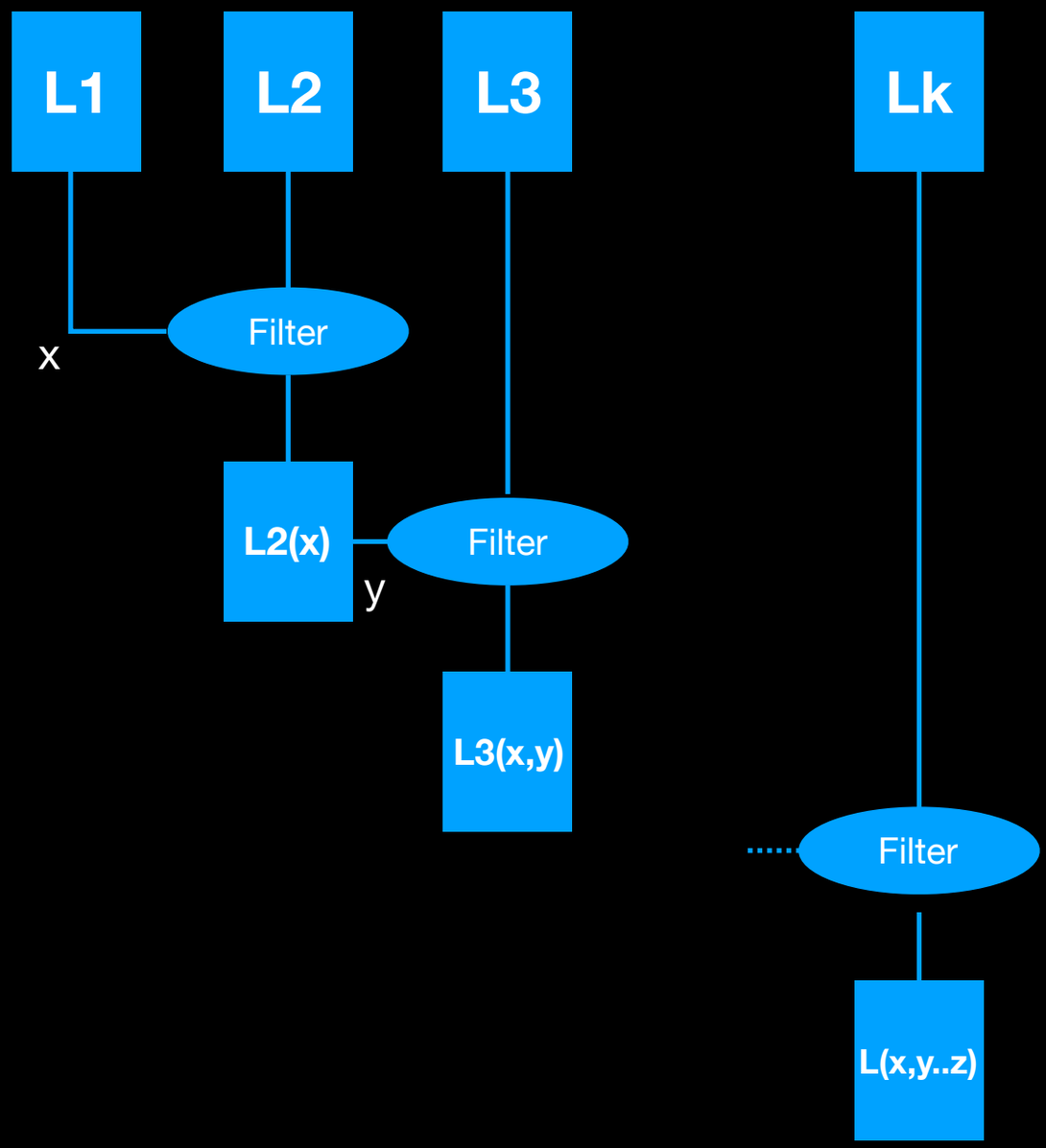
Sieving and k-clique listing



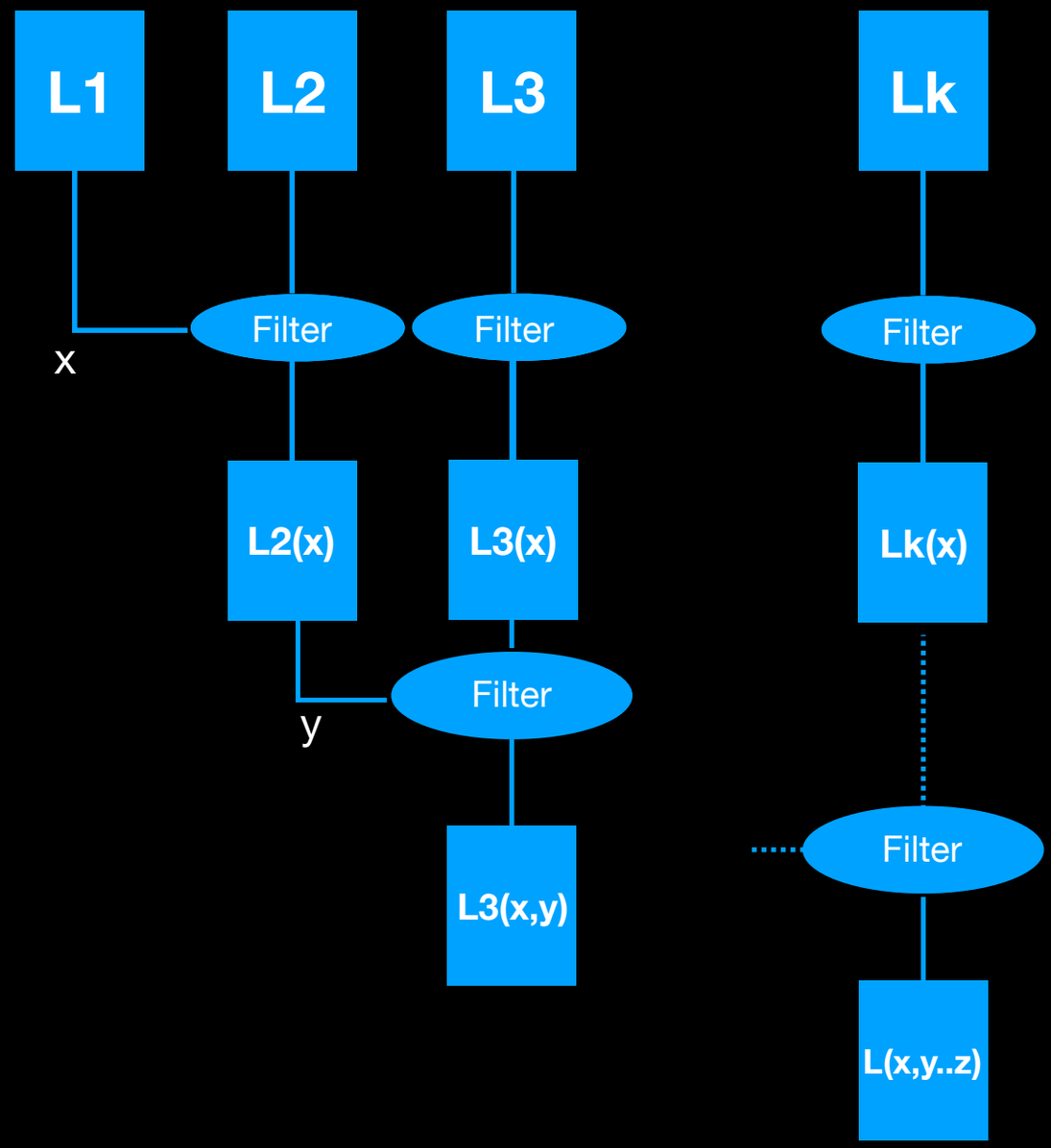
Filter receives as input vectors from the two lists,
and a real number C, the target inner product.
This procedure only passes the vectors whose inner product
is close to C.

Bai Laarhoven Stehlé, Tuple Lattice Sieving,
LMS J. of Comp & Math, 19:146-162, 2016

Sieving and k-clique listing



Bai Laarhoven Stehlé, Tuple Lattice Sieving,
LMS J. of Comp & Math, 19:146-162, 2016



Herold, Kirshanova, Laarhoven, Speed-ups and time-memory
trade-offs for tuple lattice sieving, PKC 2018, (407-436)

Sieving and k-clique listing

Quantum BLS16

K	2	3	4	5	…	28	29	30
TIME	0.3112	0.3306	0.3289	0.3219	…	0.2989	0.2989	0.2989
MEMORY	0.2075	0.1907	0.1796	0.1685		0.1395	0.1395	0.1395

optimising for time

Quantum BLS16+HKL18

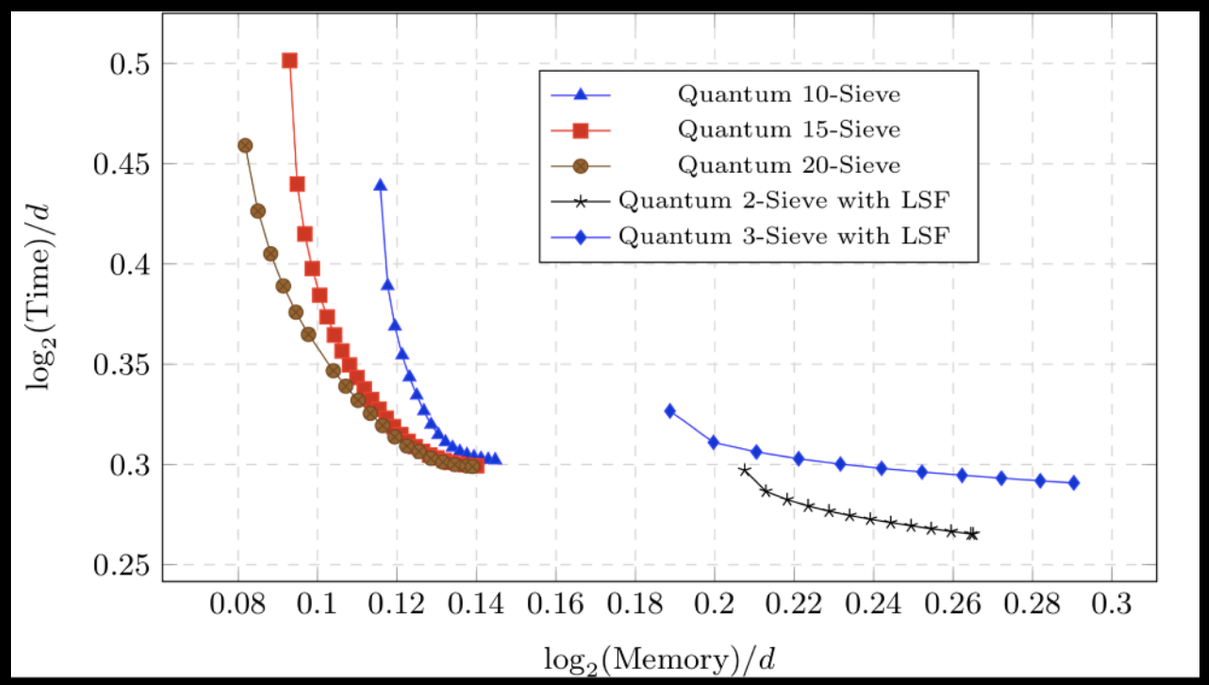
TIME	0.3112	0.3306	0.3197	0.3088	…	0.2989	0.2989	0.2989
MEMORY	0.2075	0.1907	0.1731	0.1638		0.1395	0.1395	0.1395

optimising for time

Quantum BLS16+HKL18

TIME	0.3112	0.3349	0.3215	0.3305	…	0.6352	0.2423	0.6490
MEMORY	0.2075	0.1887	0.1724	0.1587		0.0637	0.0623	0.0609

optimising for memory



Each point on the curve represents (Memory, Time) optimised for time, for a fixed memory

In Summary

Can solve SVP, heuristically, in $2^{0.2989d+o(d)}$ time steps and $2^{1395d+o(d)}$ classical memory, in the QRAM model , requiring $poly(d)$ quantum memory.

Can solve SVP, heuristically, in $2^{1037d+o(d)}$ time steps in the quantum circuit model requiring $2^{0.2075d+o(d)}$ quantum memory.

In Summary

Can solve SVP, heuristically, in $2^{0.2989d+o(d)}$ time steps and $2^{1395d+o(d)}$ classical memory, in the QRAM model , requiring $poly(d)$ quantum memory.

Can solve SVP, heuristically, in $2^{1037d+o(d)}$ time steps in the quantum circuit model requiring $2^{0.2075d+o(d)}$ quantum memory.

Open questions

1. k-sieve for k=f(n)
2. Parallel k-sieve

In Summary

Can solve SVP, heuristically, in $2^{0.2989d+o(d)}$ time steps and $2^{1395d+o(d)}$ classical memory, in the QRAM model , requiring $poly(d)$ quantum memory.

Can solve SVP, heuristically, in $2^{1037d+o(d)}$ time steps in the quantum circuit model requiring $2^{0.2075d+o(d)}$ quantum memory.

Open questions

1. k-sieve for k=f(n)
2. Parallel k-sieve

ia.cr/2019/1016
Quantum algorithms for the Approximate k-List Problem and their Application to Lattice Sieving