Improved Classical and Quantum Algorithms for Subset-Sum

Xavier Bonnetain[1], Rémi Bricout[2,3], André Schrottenloher[3], **Yixin Shen**[4]

[1] Institute for Quantum Computing, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, Canada
[2] Sorbonne Université, Collège Doctoral, F-75005 Paris, France
[3] Inria, France
[4] Université de Paris, IRIF, CNRS, F-75013 Paris, France

April 10, 2020

Outline

Introduction

The Subset-Sum Problem

Problem

Given: $\mathrm{a} = (a_1, \ldots, a_{\mathrm{n}})$ *a vector of* ℓ*-bit integers, and an* ℓ*-bit target* S*, find* $\mathrm{e} = (e_1, \ldots, e_{\mathrm{n}}) \in \{0,1\}^{\mathrm{n}}$ *such that* $\mathrm{e} \cdot \mathrm{a} = \sum_i e_i a_i = S \mod 2^{\ell}$.

- The decision version is NP-complete
- The low-density case ($\ell \gg \mathrm{n}$) is related to lattice SVP
- The high-density case ($\ell \ll \mathrm{n}$) is solvable efficiently
- The density-1 case ($\ell \simeq \mathrm{n}$) is hard

Subset-sums in (post-quantum) cryptography

- Repeatedly used as a hard problem for post-quantum cryptography [a]
- Similar techniques that we will see in this presentation apply to other problems (generic decoding algorithms) [b]
- Solving subset-sums is also useful in quantum hidden shift algorithms [c]

[a] Lyubashevsky, Palacio, and Segev, *"Public-Key Cryptographic Primitives Provably as Secure as Subset Sum"*, TCC 10

[b] Kachigar and Tillich, *"Quantum Information Set Decoding Algorithms"*, PQCrypto 17

[c] Bonnetain, *Improved Low-qubit Hidden Shift Algorithms*, 2019

The random Subset-Sum Problem

Problem

Given: $\mathsf{a} = (a_1, \ldots, a_\mathsf{n})$ *a vector of* n*-bit integers, and an* n*-bit target* S*, find* $\mathsf{e} = (e_1, \ldots, e_\mathsf{n}) \in \{0,1\}^\mathsf{n}$ *such that* $\mathsf{e} \cdot \mathsf{a} = \sum_i e_i a_i = S \mod 2^\mathsf{n}$; ***where*** a, S ***are selected uniformly at random.***

- Classical and quantum algorithms run in time $\widetilde{\mathcal{O}}\left(2^{\beta \mathsf{n}}\right)$: we are interested in the value of β
- In this talk, we optimize the **time exponent** (not the memory)
- We consider that e has Hamming weight $\frac{\mathsf{n}}{2}$

Classical algorithms

The time is $\widetilde{\mathcal{O}}\left(2^{\beta n}\right)$.

Technique	β	Ref.
MIM	0.5	HS74 (Slide 8)
4-list merge	0.5	SS81 (Slide 9)
$\{0,1\}$	0.3370	HGJ10 (Slide 18)
$\{-1,0,1\}$	0.2909	BCJ11 (Slide 23)
$\{-1,0,1\}$ + NN	0.287	Ilya Ozerov's PhD thesis
$\{-1,0,1,2\}$	0.283	Ours

Classical algorithm: meet-in-the-middle

Cut the solution $e = \underbrace{(\ \underbrace{0\dots0}_{n/2 \text{ bits}} \mid \underbrace{*\dots*}_{n/2 \text{ bits}}\ )}_{2^{n/2} \text{ choices: } L_l} + \underbrace{(\ \underbrace{*\dots*}_{n/2 \text{ bits}} \mid \underbrace{0\dots0}_{n/2 \text{ bits}}\ )}_{2^{n/2} \text{ choices: } L_r}$

Then find $e_l \in L_l, e_r \in L_r$ s.t. $e_l \cdot a = -e_r \cdot a + S \mod 2^n$.

Complexities

Time: $\mathcal{O}\left(2^{n/2}\right)$ (best worst-case time); **Memory:** $\mathcal{O}\left(2^{n/2}\right)$

Horowitz and Sahni, *"Computing Partitions with Applications to the Knapsack Problem"*, J. ACM

Schroeppel and Shamir's 4-list merging

By cutting in 4 instead of 2, we can decrease the memory to $2^{n/4}$.

$$e = \underbrace{(*|0|0|0)}_{e_0 \in L_0} + \underbrace{(0|*|0|0)}_{e_1 \in L_1} + \underbrace{(0|0|*|0)}_{e_2 \in L_2} + \underbrace{(0|0|0|*)}_{e_3 \in L_3}$$

We now look for $(e_0, e_1, e_2, e_3) \in L_0 \times L_1 \times L_2 \times L_3$ s.t.

$$a \cdot (e_0 + e_1 + e_2 + e_3) = S \mod 2^n$$

$\Rightarrow$ there is still one solution among $2^{n/4} \times 2^{n/4} \times 2^{n/4} \times 2^{n/4}$.

Schroeppel and Shamir, "*A $T = O(2^{n/2})$, $S = O(2^{n/4})$ Algorithm for Certain NP-Complete Problems*", SIAM 81

Schroeppel and Shamir's 4-list merging (ctd.)

- Choose an $n/4$-bit number $c \mod M$
- Repeat for every value of c:

$e_0 \in L_0$ | $e_1 \in L_1$ | $e_2 \in L_2$ | $e_3 \in L_3$

$e_0 + e_1$ s. t. $a \cdot (e_0 + e_1) = (c + S) \mod M$ Size $2^{n/4}$

$e_2 + e_3$ s. t. $a \cdot (e_2 + e_3) = -c \mod M$ Size $2^{n/4}$

Solution with prob. $2^{-n/4}$

Complexities

Time: $\mathcal{O}\left(2^{n/4} \times 2^{n/4}\right)$

Memory: $\mathcal{O}\left(2^{n/4}\right)$

Representations

Breaking the $2^{n/2}$ bound

- When we have **one** solution among 2^n tuples, we don't know of any better time than $2^{n/2}$
- The idea of Howgrave-Graham and Joux (HGJ): cut e with respect to its Hamming weight

Suppose that e is of weight $n/2$ (worst case). Write for example:

$$\underbrace{e}_{\substack{\text{Weight}\\ n/2}} = \underbrace{e_0}_{\substack{\text{Weight}\\ n/8}} + \underbrace{e_1}_{\substack{\text{Weight}\\ n/8}} + \underbrace{e_2}_{\substack{\text{Weight}\\ n/8}} + \underbrace{e_3}_{\substack{\text{Weight}\\ n/8}}$$

notice that: $\binom{n}{n/8}^4 \simeq 2^{2.174n} \ggg \binom{n}{n/2} \simeq 2^n$: many solution tuples!

Howgrave-Graham and Joux, *"New Generic Algorithms for Hard Knapsacks"*, EC 10

Notations

We introduce **distributions** and **weight constraints**.

Distributions

$e \in D^n[\alpha]$ if e contains αn "1" and $(1-\alpha)n$ "0".

Then if $e_1 \in D^n[\alpha_1], e_2 \in D^n[\alpha_2]$ we have $e_1 + e_2 \in D^n[\alpha_1 + \alpha_2]$ **with some probability** (to be continued).

Weight constraints

e has "a cn-bit weight constraint" if we are able to constrain the (knapsack) weight of e as $e \cdot a = s \mod M$ for (previously) chosen cn-bit integers M and s.

If e_1 has a cn-bit-weight cons. and e_2 has a cn-bit-weight cons., then $e_1 + e_2$ as well by linearity (but the precise moduli don't matter!)

Example: Schroeppel-Shamir

$L_1, 0$
$\{0\}^{n/4} \times D^{n/4}[\frac{1}{2}] \times \{0\}^{n/2}$

$L_2, 0$
$\{0\}^{n/2} \times D^{n/4}[\frac{1}{2}] \times \{0\}^{n/4}$

$L_0, 0$
$D^{n/4}[\frac{1}{2}] \times \{0\}^{3n/4}$

$L_3, 0$
$\{0\}^{3n/4} \times D^{n/4}[\frac{1}{2}]$

$L_0^1, c = \frac{1}{4}$
$D^{n/2}[\frac{1}{2}] \times \{0\}^{n/2}$

$L_1^1, c = \frac{1}{4}$
$\{0\}^{n/2} \times D^{n/2}[\frac{1}{2}]$

$L^0, 1$

Generic layout

Solution

The solution e is the only vector with an n-bit weight constraint and $e \in D^{n}[1/2]$.

- We start from vectors e with a 0-bit weight constraint and distributions $D^{n}[\alpha]$
- We sum them, trying to increase the weight constraint ("merging")
- Eventually we get to a n-bit weight constraint and a distribution $D^{n}[1/2]$

Merging and filtering

List L_2
$\subseteq D^{n}[\alpha_2]$
cons. cn

List L_1
$\subseteq D^{n}[\alpha_1]$
cons. cn

List L
size $|L| = |L_1||L_2|/(2^{nd})$
cons. $(c+d)n$

List $L^f \subseteq D^{n}[\alpha_1 + \alpha_2]$
size $|L^f| = |L| \times p$
cons. $(c+d)n$

Step 1: merging

Find pairs with more constrained weights.

We produce L in time $\max(\min(|L_1|, |L_2|), |L|)$.

Step 2: filtering

Keep only the $e_1 + e_2$ that conform to the expected distribution.

p is the "filtering probability" for: $D^{n}[\alpha_1] \times D^{n}[\alpha_2] \to D^{n}[\alpha_1 + \alpha_2]$

Merging and filtering (ctd.)

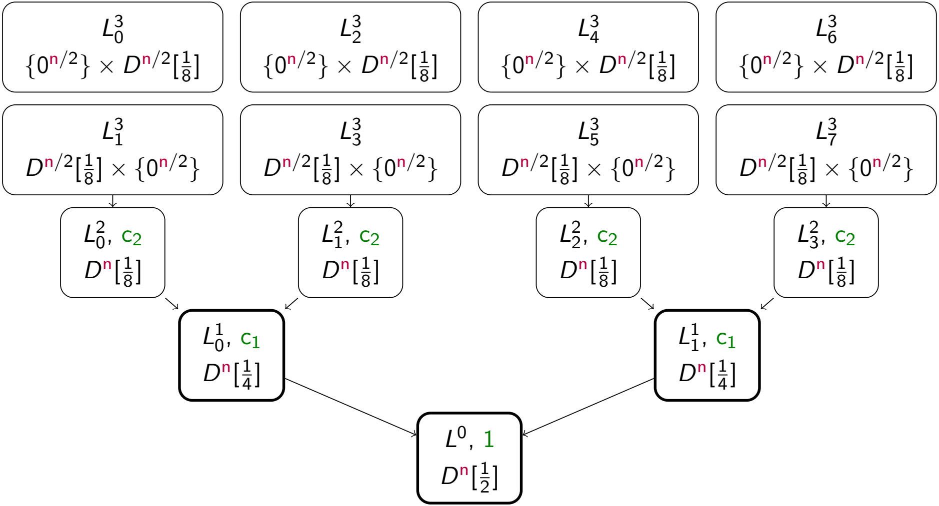
Heuristic

The vectors in L^f are uniformly distributed in $D^n[\alpha_1 + \alpha_2]$.

Approximations

- Representation sets have size: $D^n[\alpha] = \binom{n}{\alpha n} \simeq 2^{h(\alpha)n}$
- $h(x) = -x \log x - (1-x)\log(1-x)$
- In general we have a filtering probability $D^n[\alpha_1] \times D^n[\alpha_2] \to D^n[\alpha_1 + \alpha_2]$ of: $\simeq 2^{(h(\alpha_1/(1-\alpha_2)) - h(\alpha_1))n}$

The HGJ algorithm



HGJ step 3: left-right split with a modulus

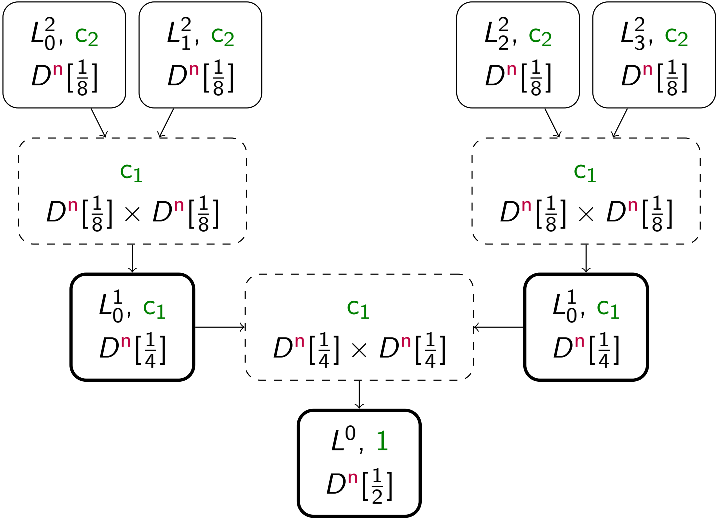
$$\begin{array}{c} L_6^3 \\ \{0^{n/2}\} \times D^{n/2}[\tfrac{1}{8}] \end{array} \qquad \begin{array}{c} L_7^3 \\ D^{n/2}[\tfrac{1}{8}] \times \{0^{n/2}\} \end{array}$$

$$\searrow \quad \swarrow$$

$$\begin{array}{c} L_3^2,\ c_2 \\ D^n[\tfrac{1}{8}] \end{array}$$

At this level we can afford to **merge without filtering**: find vectors $e \in D^n[\frac{1}{8}]$ with a c_2-weight constraint (on $e \cdot a$).

HGJ step 2 and 1: merge and filter



An optimization problem

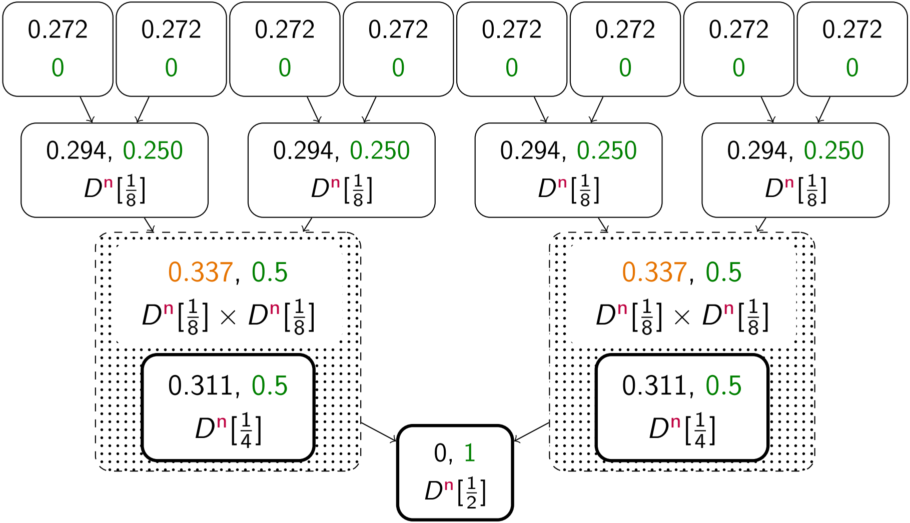
We write all parameters in $\log_2$, relative to n:

$$\begin{array}{c||c} |D^{n/2}[\frac{1}{8}]| & h(1/8)/2 \simeq 0.2718 \\ |L_i^j| & \ell_i^j \\ |L_1||L_2|/(2^{nd}) & \ell_1 + \ell_2 - d \\ |L| \times p & \ell + \mathrm{pf} \end{array}$$

We compute the filtering probabilities:

- $D^n[0, \frac{1}{8}] \times D^n[0, \frac{1}{8}] \to D^n[0, \frac{1}{4}]$: $2^{-0.02585n}$
- $D^n[0, \frac{1}{4}] \times D^n[0, \frac{1}{4}] \to D^n[0, \frac{1}{2}]$: $2^{-0.12256n}$

Optimized parameters for HGJ



Better representations: BCJ

Sample distributions with $\{-1, 0, 1\}$.

- Of course we still need to obtain $D^{n}[\frac{1}{2}]$ in the end
- The "-1" need to be canceled out by "1"
- The "-1" shouldn't sum up to "-2"!
- More parameters, new filtering probabilities
- Improvement on the time exponent: $0.291 < 0.337$

Becker, Coron, and Joux, *"Improved Generic Algorithms for Hard Knapsacks"*, EC 11

New results

Idea 1: do not saturate the lists

Starting lists are **not equal** to $D^{n/2}[*]$ but **sampled** u.a.r. from it.

BCJ without saturation: $0.289 < 0.291$

Idea 2: still better representations

Why stop at "-1"? Add some "2".

- Of course we still need to obtain $D^{n}[\frac{1}{2}]$ in the end
- Some "1" can sum up to "2" (but not too much)
- A "-1" and a "2" give a "1"

BCJ without saturation and with "2": $0.283 < 0.289$

Bonnetain et al., *Improved Classical and Quantum Algorithms for Subset-Sum*, ePrint 2020/168

Going Quantum

Classical search

$$\text{Let } \underbrace{X}_{\substack{\text{Search space,}\\ \text{size } N}} = \underbrace{G}_{\substack{\text{Good ones,}\\ \text{size } T}} \cup \underbrace{B}_{\substack{\text{Bad ones, size}\\ N-T}}$$

Let `Sample` and `Test` be functions to *sample x from X* and *test if $x \in G$*, in time $t_{\texttt{Sample}}$ and $t_{\texttt{Test}}$.

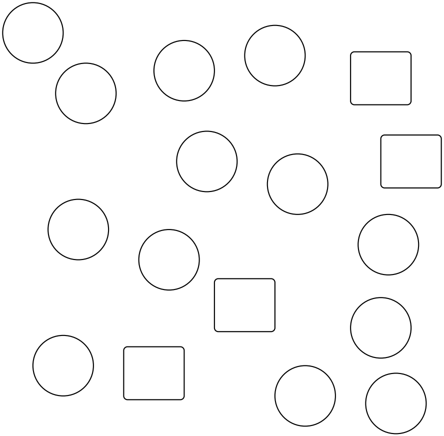
There exists a function $\texttt{Sample}_G$ that samples from G in time:

$$\frac{N}{T}\left(t_{\texttt{Sample}} + t_{\texttt{Test}}\right)$$

$\Rightarrow$ we transform a sampling procedure for the "search space" into a sampling procedure for the "solution space".

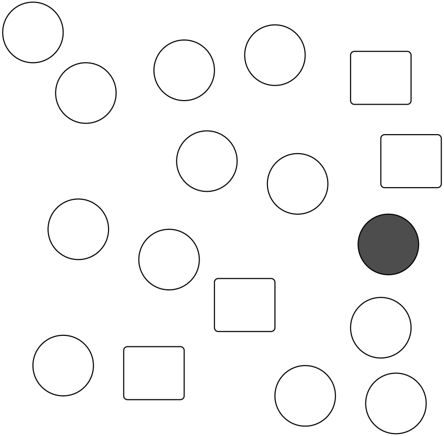
Classical search vs. quantum search

In the classical realm, we test elements x at random until we have found (a random) $x \in G$.



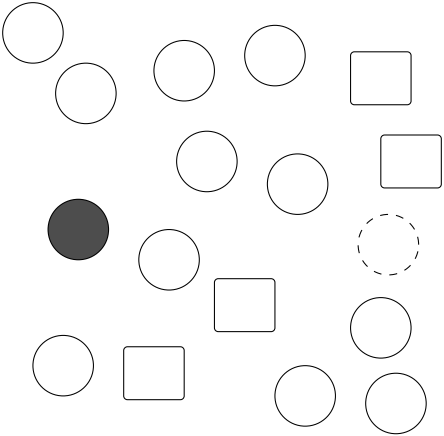
Classical search vs. quantum search

In the classical realm, we test elements x at random until we have found (a random) $x \in G$.



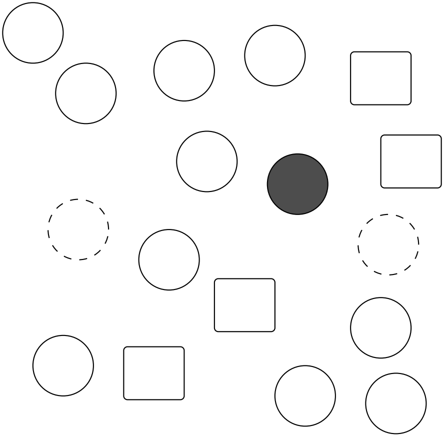
Classical search vs. quantum search

In the classical realm, we test elements x at random until we have found (a random) $x \in G$.



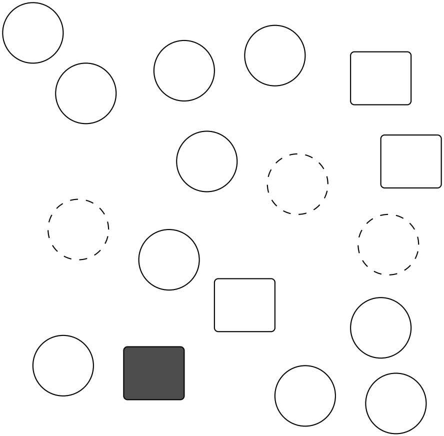
Classical search vs. quantum search

In the classical realm, we test elements x at random until we have found (a random) $x \in G$.



Classical search vs. quantum search

In the classical realm, we test elements x at random until we have found (a random) $x \in G$.



Quantum search (amplitude amplification)

$$\underbrace{X}_{\substack{\text{Search space,}\\ \text{size } N}} = \underbrace{G}_{\substack{\text{Good ones,}\\ \text{size } T}} \cup \underbrace{B}_{\substack{\text{Bad ones, size}\\ N-T}}$$

Let `QSample` and `QTest` be quantum algorithms to **quantumly sample X** and **quantumly test if $x \in G$**, in time $t_{\mathtt{Sample}}$ and $t_{\mathtt{Test}}$.

There exists an algorithm $\mathtt{QSample}_G$ that samples G in time:

$$\sqrt{\frac{N}{T}}\left(t_{\mathtt{QSample}} + t_{\mathtt{QTest}}\right)$$

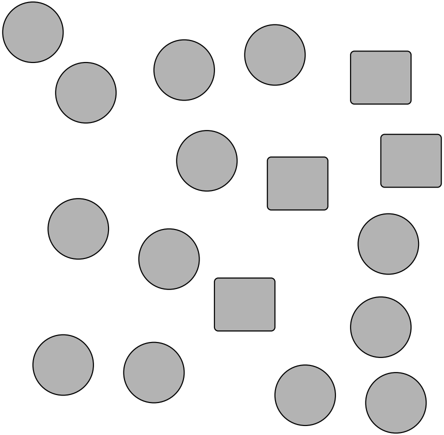
Quantum test: means testing **any $x \in X$ in superposition**

Quantum sample: means producing a uniform superposition of X

Brassard et al., *"Quantum amplitude amplification and estimation"*, 2002

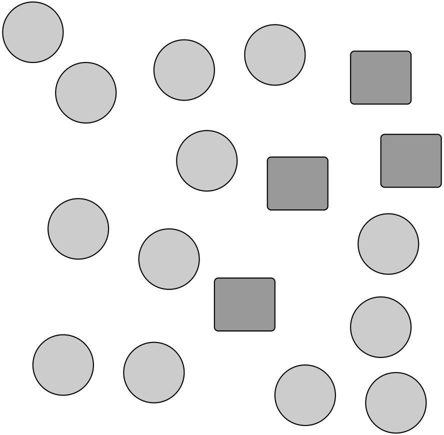
Classical search vs. quantum search

In the quantum realm, we move globally from X to G.



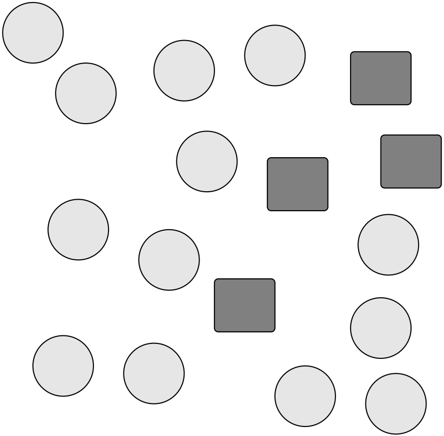
Classical search vs. quantum search

In the quantum realm, we move globally from X to G.



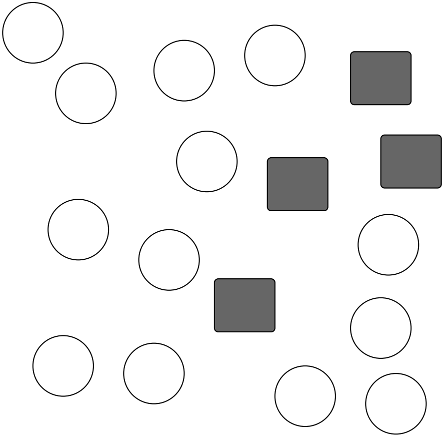
Classical search vs. quantum search

In the quantum realm, we move globally from X to G.



Classical search vs. quantum search

In the quantum realm, we move globally from X to G.



Interlude: quantum memory models

What happens if X is in memory?

Classical sample: only reads a single $x \in X$ (easy)

Quantum sample: must read all of X in superposition (maybe not easy): this is **quantum random access**

	Quantum random access	
Classical write	Classical memory quantum random access **QRACM**	$\Rightarrow$ **This section**
Quantum write	Quantum memory quantum random access **QRAQM**	Previous quantum subset-sum algos

Quantum algorithms for subset-sum

The time is $\widetilde{\mathcal{O}}\left(2^{\beta n}\right)$.

Technique	β	Ref.	Classical version	Model
MIM	0.3334	BHT98	HS74	QRACM
4-list merge	0.3	BJLM13	SS81	QRAQM
$\{0,1\}$	0.241	BJLM13	HGJ10	QRAQM + conj.
$\{0,1\}$	**0.2356**	**Ours**	**HGJ10**	**QRACM**
$\{-1,0,1\}$	0.226	HM18	BCJ11	QRAQM + conj.

"Sampling-and-filtering"

List L_1
size $|L_1|$
cons. c

List L_2
size $|L_2|$
cons. c

List L
size $|L| = |L_1||L_2|/(2^{\mathrm{nd}})$
cons. $\mathrm{c} + \mathrm{d}$

List L^f
size $|L^f| = |L| \times p$
cons. $\mathrm{c} + \mathrm{d}$

Let's separate:

- **The sampled list**
- **The intermediate list**

We turn samples from L_1 into:

- Samples from L:

$$t_{\mathtt{Sample}(L)} = \max\left(\frac{2^{\mathrm{nd}}}{|L_2|}, 1\right) t_{\mathtt{Sample}(L_1)}$$

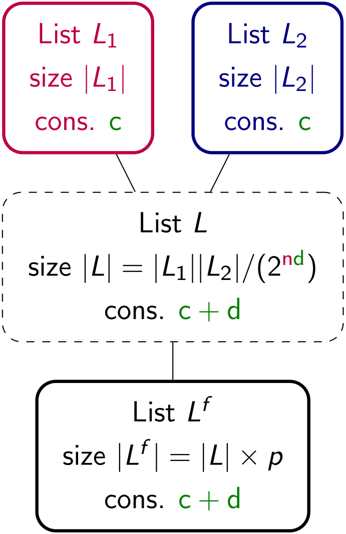
(find an element with same modulus)

- Samples from L^f:

$$t_{\mathtt{Sample}(L_f)} = \frac{1}{p} t_{\mathtt{Sample}(L)}$$

(wait until the filter is passed)

Quantum “sampling-and-filtering”



Assume that we have quantum samples from L_1.

Then we have:

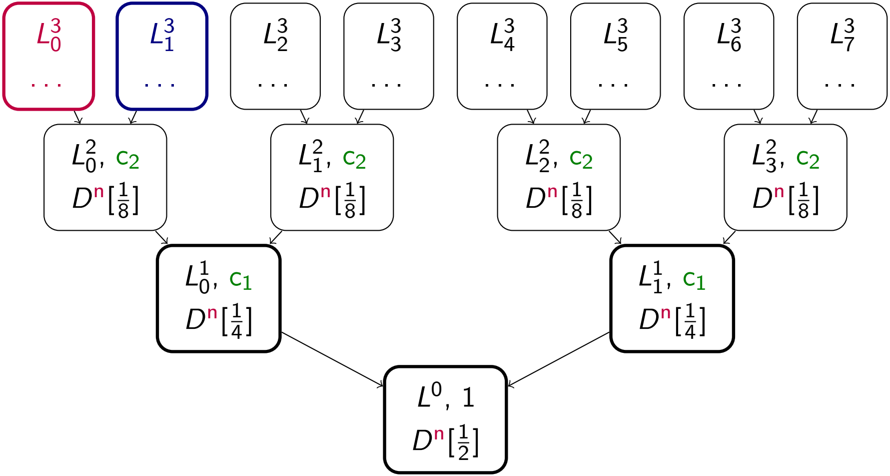
- Quantum samples from L:

$$t_{\mathtt{QSample}(L)} = \max\left(\sqrt{\frac{2^{\mathrm{nd}}}{|L_2|}}, 1\right) t_{\mathtt{QSample}(L_1)}$$

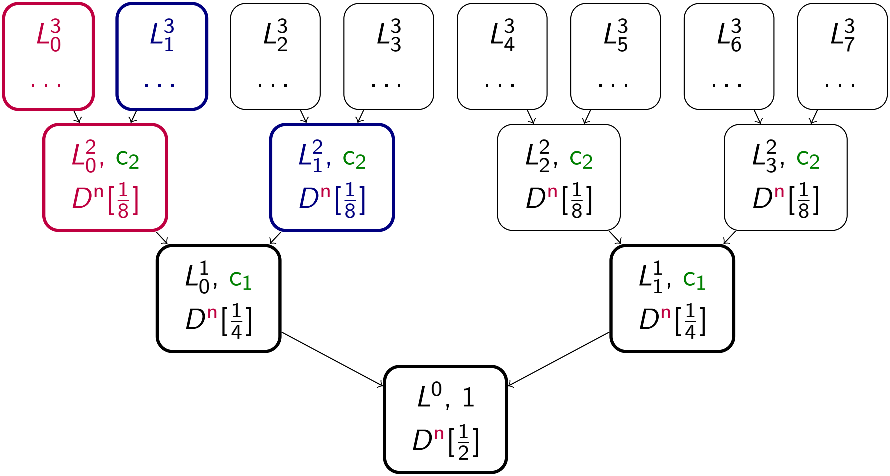
- Quantum samples from L^f:

$$t_{\mathtt{Sample}(L_f)} = \sqrt{\frac{1}{p}} t_{\mathtt{Sample}(L)}$$

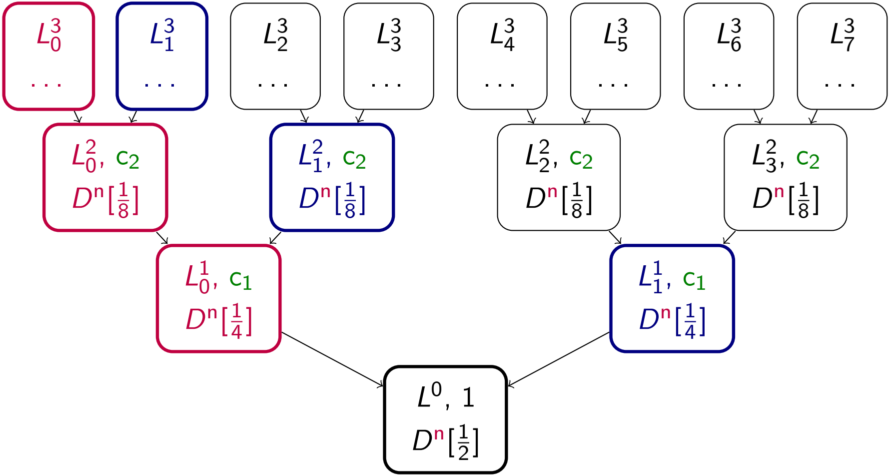
HGJ in the “sampling” framework



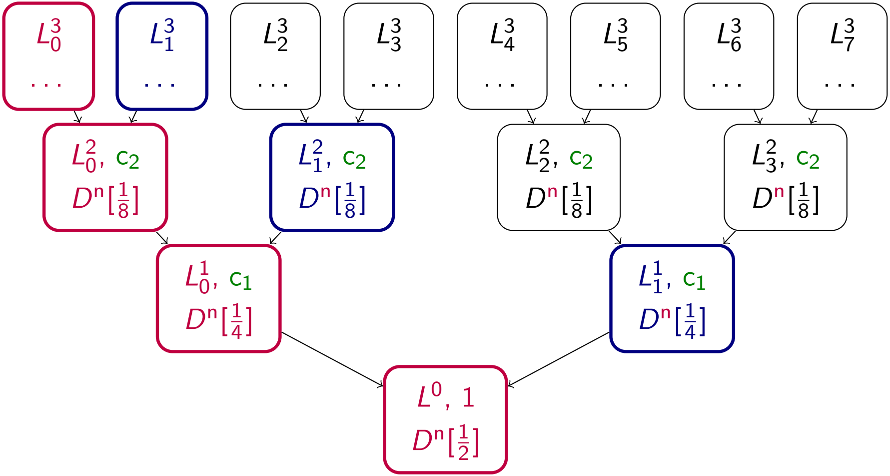
HGJ in the “sampling” framework



HGJ in the “sampling” framework



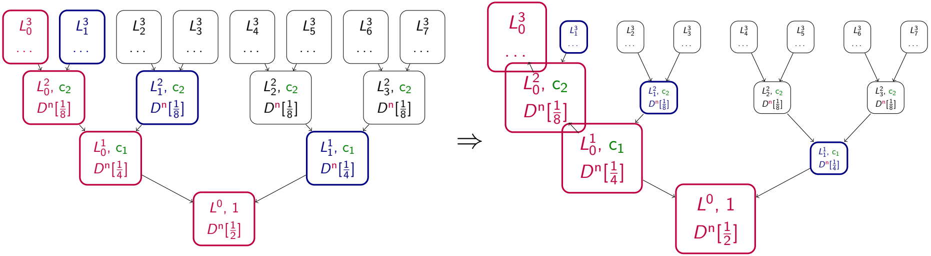
HGJ in the "sampling" framework



Using quantum search

Quantum search will square-root the sampling time of L^0. But it's useless if the intermediate lists cost the same as before.

$\Rightarrow$ we make the tree unbalanced.



Details and result

- Unbalanced left-right split of L_0^3 and L_1^3, unbalanced weights for the lists
- L_1^3, L_1^2, L_1^1 are intermediate lists stored in QRACM (classical data with quantum random access)
- **only poly(n) quantum storage needed**

$$\underbrace{0.226 \text{ (HM18)} \;<}_{\text{We use only } \{0,1\} \text{ representations}} \quad 0.2356 \quad \underbrace{< 0.241 \text{ (BJLM13)}}_{\text{We filter more efficiently}}$$

Subset-Sum with Quantum Walks

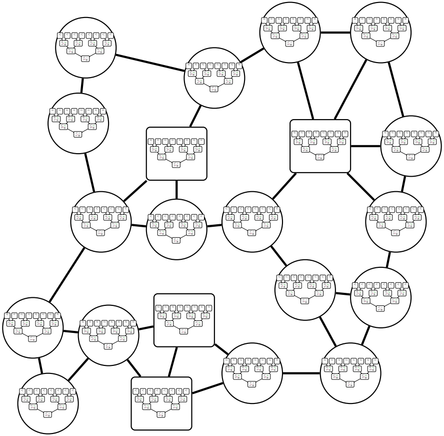
Quantum algorithms for subset-sum

The time is $\widetilde{\mathcal{O}}\left(2^{\beta n}\right)$.

Technique	β	Ref.	Classical version	Model
MIM	0.3334	BHT98	HS74	QRACM
4-list merge	0.3	BJLM13	SS81	QRAQM
$\{0,1\}$	0.241	BJLM13	HGJ10	QRAQM + **conj.**
$\{0,1\}$	0.2356	Ours	HGJ10	QRACM
$\{-1,0,1\}$	0.226	HM18	BCJ11	QRAQM + **conj.**
$\{-1,0,1,2\}$	**0.2156**	**Ours**		**QRAQM + conj.**
$\{-1,0,1,2\}$	**0.2182**	**Ours**		**QRAQM**

A classical walk for HGJ

Reduce the HGJ merging tree to a smaller tree, with smaller starting lists. Now L^0 does not always contain a solution.



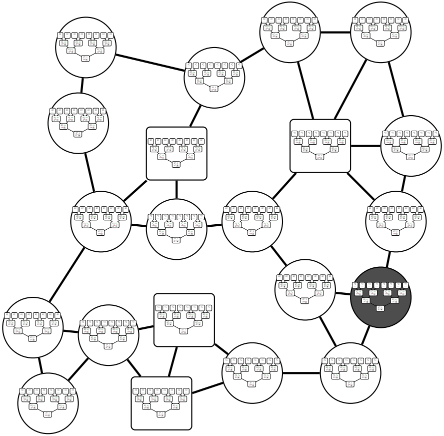
Walking on the graph

We use a (regular, undirected) Johnson graph $J(D, L)$.

- A vertex contains a product of 8 small lists $L_i^3, 0 \leq i \leq 7$, of size $|L_i^3| = L$, chosen among distributions $|D^i| = D$, **and** the whole tree built from these lists.
- There are $\binom{D}{L}^8$ vertices.
- Some vertices are **marked**: they contain the knapsack solution.
- We go from one to another by changing an element in a list L_i^3 **and** updating the tree.

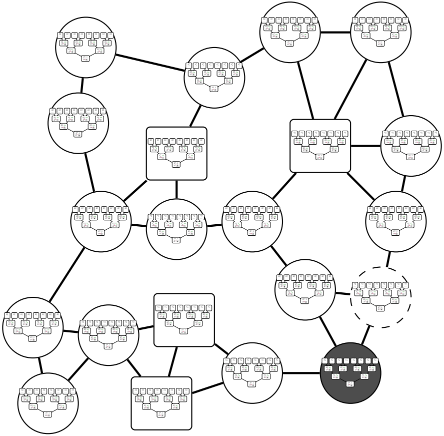
Classical random walk

We move to random neighbors until we find a marked vertex.



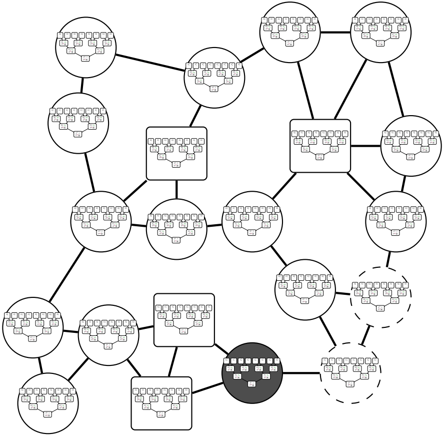
Classical random walk

We move to random neighbors until we find a marked vertex.



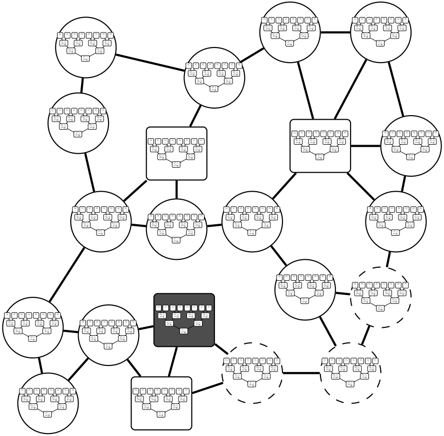
Classical random walk

We move to random neighbors until we find a marked vertex.



Classical random walk

We move to random neighbors until we find a marked vertex.



Cost of a classical random walk

We need procedures:

- To **setup** a starting arbitrary vertex (S)
- To **move** from one vertex to one of its neighbors (U)
- To **check** if a vertex is marked (trivial) (C)

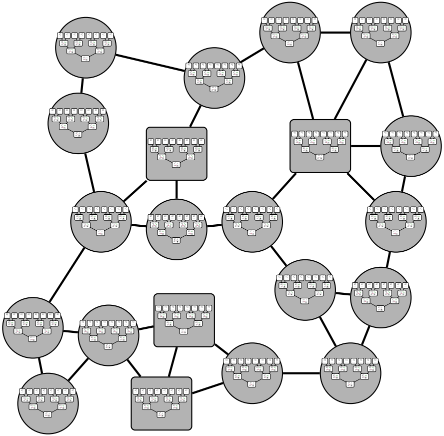
We will find a marked vertex in time:

$$\mathsf{S} + \underbrace{\frac{1}{\epsilon}}_{\substack{\epsilon \text{ proportion of} \\ \text{marked vertices}}} \left(\underbrace{\frac{1}{\delta}}_{\substack{\delta \text{ spectral gap} \\ \text{of the graph}}} \mathsf{U} + \mathsf{C} \right)$$

where $\frac{1}{\delta}$ is the number of updates before we reach a new uniformly random vertex. In a Johnson graph $J(D, L)$, $\frac{1}{\delta} \simeq L$. (We need to replace all elements.)

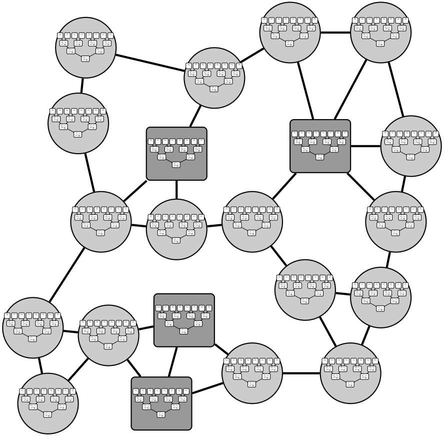
Quantum walk

As in quantum search, the walk transforms a uniform superposition over the whole graph into a superposition over marked vertices.



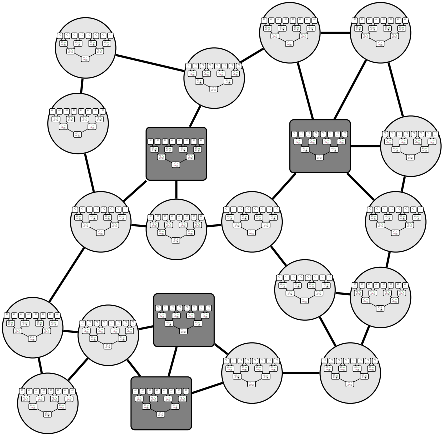
Quantum walk

As in quantum search, the walk transforms a uniform superposition over the whole graph into a superposition over marked vertices.



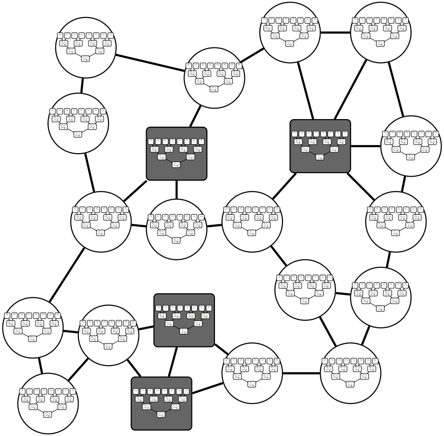
Quantum walk

As in quantum search, the walk transforms a uniform superposition over the whole graph into a superposition over marked vertices.



Quantum walk

As in quantum search, the walk transforms a uniform superposition over the whole graph into a superposition over marked vertices.



Time of a quantum walk (MNRS framework)

- The **setup** now requires to create a superposition over **all** vertices
- As in quantum search, we perform $\sqrt{\frac{1}{\epsilon}}$ steps instead of $\frac{1}{\epsilon}$
- But the mixing is also accelerated!

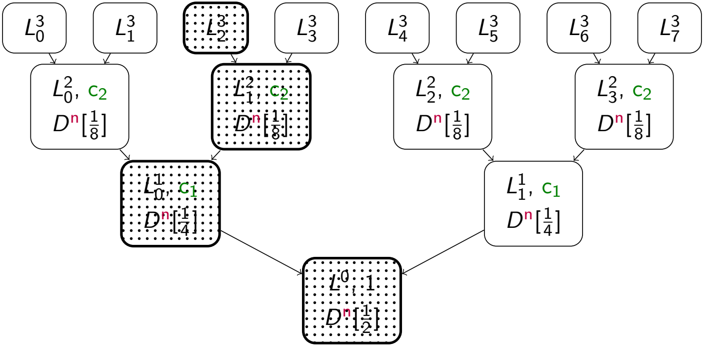
$$\mathsf{S} + \underbrace{\sqrt{\frac{1}{\epsilon}}}_{\text{Walk steps}} \left(\underbrace{\sqrt{\frac{1}{\delta}}\mathsf{U}}_{\text{Mixing time}} + \mathsf{C} \right)$$

- The **Update** handles all vertices and all edges in superposition

Magniez et al., *"Search via quantum walk"*, SIAM 11

Tracking the updates

- The update U must replace one element in a lower-level list **and** update the merging tree data structure.



- **On average,** there is a single replacement to make at each level (no problem classically).

Superposition updates

- The update needs to take a fixed time.
- Since we are handling all vertices **and all edges** in superposition, there are cases when updating the tree would cost an exponential time.

Can we abort the bad cases?

Not in the MNRS framework: the data structure (the tree) must depend only on the vertex (the initial lists).

The quantum walk conjecture of Helm and May (TQC18)

With an update of expected time $\mathcal{O}(1)$, we can still do the runtime analysis as if it had an exact time $\mathcal{O}(1)$.

Helm and May, *"Subset Sum Quantumly in 1.17^n"*, TQC 18

New results

- We have modified the **data structure** to guarantee the update time
- This reduces (marginally) the number of marked vertices

Fact

Previous quantum walks for subset-sum do not require the update conjecture.

- However, this data structure is not enough for our best algorithms . . .

Summary of quantum walk results

Technique	Time	Ref.	Classical version	Model
$\{0,1\}$	0.241	BJLM13	HGJ10	QRAQM + **conj.**
$\{-1,0,1\}$	0.226	HM18	BCJ11	QRAQM + **conj.**
$\{-1,0,1,2\}$	0.2156	Ours		QRAQM + **conj.**
$\{-1,0,1,2\}$	0.2182	Ours		QRAQM

Conclusion and open questions

On classical algorithms

More symbols and nearest-neighbor techniques should improve the exponent $\Rightarrow$ how far could we go?

On quantum algorithms with quantum search

Better representations should improve the exponent . . . if we manage to make the optimization converge.

Conclusion and open questions (ctd.)

On quantum walks

- The update conjecture can be removed from previous works . . . but not completely from ours
- It seems that we still need to adapt the MNRS Quantum Walk framework

ePrint 2020/168

Thank you!