Isogeny School 2021
Week 8: Quantum Claw-Finding

Sam Jaques

August 30, 2021

1 Isogenies as Claw Finding

Problem 1.1 (Claw-finding)**.** *Let $f : X \to S$ and $g : Y \to S$ be two functions. Find $x \in X$ and $y \in Y$ such that $f(x) = g(y)$, if it exists.*

To connect this to isogenies, we imagine two curves E_0 and E_1 where there is a secret isogeny $\phi : E_0 \to E_1$ of degree ℓ^e. We let X be the set of all isogenies $\phi_i : E_0 \to E_i$ of degree $\ell^{e/2}$ and Y be the set of all isogenies $\phi_j : E_1 \to E_j$ of degree $\ell^{e/2}$. The functions f and g will take an isogeny as input and output the j-invariant of the image curve (so the set S is the set of all possible j-invariants). A claw (x, y) implies a single curve E_{01} and two isogenies $\phi_x : E_0 \to E_{01}$ and $\phi_y : E_1 \to E_{01}$. From this, we can compute $\hat{\phi}_y \circ \phi_x : E_0 \to E_1$, an isogeny of degree ℓ^e.

For SIDH, $|X| = \ell^{e/2} \approx p^{1/4}$. There are approximately $p/12$ supersingular isogeny classes, up to isomorphism, so $|S| \approx p/12$. There are $(\ell + 1)\ell^{e/2-1} \approx \ell^{e/2} \approx p^{1/4}$ isogenies of degree $\ell^{e/2}$ from E_0 (and the same from E_1). This means $|X| = |Y| \ll |S|$, which implies the claw is very likely to be unique.

From this point on, we will focus on the claw-finding problem, and three "families" of attack: Grover search, quantum walks, and Multi-Grover. It's important to remember this claw-finding perspective throws aways almost all of the rich structure of the isogenies. So far, algebraic quantum attacks either do not apply to SIDH parameters (see torsion-point attacks) or their performance is the same or worse than generic attacks (e.g., [4]).

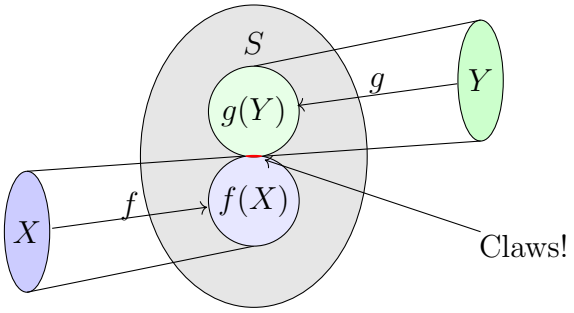


Figure 1: The claw-finding problem.

Type	**Variant**	**Specialty**
Grover	Tiny-Claw[5]	Lowest "depth-width" cost*
	Parallel Tiny-claw[2]	Offsets a lot of the query cost
Random Walk[9]	Tani[10]	Lowest number of queries*
	Distinguished Points[7]	Lowest gate cost*
Multi-Grover[3]	Distinguished Points[7]	Best parallelism

Table 1: Overview of state-of-the-art quantum claw-finding algorithms. Under reasonable limits on total runtime, all of them perform worse than classical van Oorschot-Wiener, even without accounting for the overheads of quantum computing. * indicates that the claim only holds with no runtime limit.

2 Grover's Search

We can apply Grover's algorithm directly to claw-finding. The search space is all pairs $(x, y) \in X \times Y$. If we assume there is exactly one claw, we require $O(\sqrt{|X||Y|})$ iterations. For SIDH, this is $O(p^{1/4})$.

To parallelize generically, we cannot do better than dividing the search space: With P quantum machines, we partition the search space into P subsets and each machine searches one subset. This means each machine needs $O(\sqrt{\frac{|X||Y|}{P}})$ iterations. Adding up the cost over all machines, the total cost is P times this value, or $O(\sqrt{|X||Y|P})$. This means **the total cost increases with the amount of parallelism**.

Exercise: If each iteration takes 1 unit of time, find the cost to complete a Grover search in a fixed time T (as a function of $|X|$, $|Y|$, and T).

3 Quantum Random Walks

3.1 Classical Meet-in-the-middle

The following classical attack is not the most efficient, but it will be analogous to a quantum walk.

Choose a memory parameter R. Construct a list L_x of random elements $(x, f(x))$ and a list L_y of elements $(y, g(y))$ such that $|L_x| = |L_y| = R$. Sort it and check for any claws. Then, until a claw is found, repeat:

- Delete a random element of L_x. Choose a new random element of X, compute $(x, f(x))$, then (a) check if $f(x) = g(y)$ for any $y \in L_y$; (b) insert $(x, f(x))$ into L_x. Repeat with the roles of X and Y switched.

We split the cost into set-up, update, and check costs. The set-up, to initially construct the lists, will be $O(R \log R)$ if we ignore the costs to evalute f and g and give cost $O(1)$ to access memory. The update, to delete an element and insert a new one, is $O(\log R)$. The check, to look for $f(x) = g(y)$, is $O(\log R)$ as well.

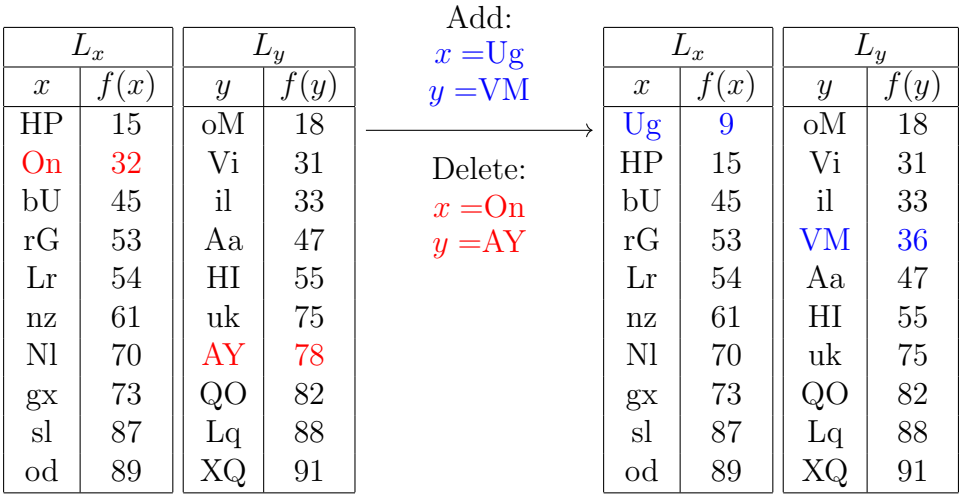
This algorithm detects a claw (x, y) if $x \in L_x$ and $y \in L_y$ at any point. The probability of this for a random set will be $\frac{R}{|X|}\frac{R}{|Y|}$. However, the probability of the claw being in $L_x \times L_y$ after replacing only one element in each is highly correlated to the probability *before* the replacement. Therefore, we expect to need to repeat about R replacements before both lists look "fresh" and we have the desired probability of containing the claw.

Together, this gives a total cost (in runtime) of

$$O\left(\underbrace{R\log R}_{:=\mathsf{S}} + \underbrace{\frac{|X||Y|}{R^2}}_{:=1/\epsilon}\underbrace{R}_{:=1/\delta}\left(\underbrace{\log R}_{:=\mathsf{U}} + \underbrace{\log R}_{:=\mathsf{C}}\right)\right) \tag{1}$$

Optimizing gives $R \approx |X|$, for a total cost of $\tilde{O}(|X|)$. For SIDH, this means $p^{1/4+o(1)}$ – but it requires $p^{1/4+o(1)}$ bits of memory as well.

Exercise: Suppose there are T pairs of claws $(x_1, y_1), \ldots, (x_T, y_T)$, with $T \ll R \ll |X| = |Y|$, such that $f(x_i) = f(x_j)$ for $i \neq j$. What is the runtime of the previously-described algorithm to find any claw?



L_x	
x	$f(x)$
HP	15
On	32
bU	45
rG	53
Lr	54
nz	61
Nl	70
gx	73
sl	87
od	89

L_y	
y	$f(y)$
oM	18
Vi	31
il	33
Aa	47
HI	55
uk	75
AY	78
QO	82
Lq	88
XQ	91

L_x	
x	$f(x)$
Ug	9
HP	15
bU	45
rG	53
Lr	54
nz	61
Nl	70
gx	73
sl	87
od	89

L_y	
y	$f(y)$
oM	18
Vi	31
il	33
VM	36
Aa	47
HI	55
uk	75
QO	82
Lq	88
XQ	91

Figure 2: A single random iteration. The list on the left has no claw and it is easy to check after each addition/deletion that the new elements do not create one.

3.2 Quantum Walk

We can re-frame the previous algorithm as a *random walk* on a graph[1]. The vertices of the graph will be the pairs of lists L_x and L_y. We want each edge to represent replacing an element from each list, so there will be an edge between (L_x, L_y) and (L'_x, L'_y) if and only if both L_x and L'_x each differ by exactly one element, and L_y and L'_y also differ by exactly one element. We say a vertex is *marked* if there is some $x \in L_x$ and $y \in L_y$ such that $f(x) = g(y)$ (i.e. (x, y) is a claw).

There is a quantum algorithm, analogous to Grover's algorithm, which performs a search for marked vertices on a graph [9]. It is almost identical to the classical algorithm of the last section, except instead of a *random* step, it takes a *superposition* of steps. Instead of checking for marked vertices, it adds a phase of -1 to any marked vertices. The complexity is

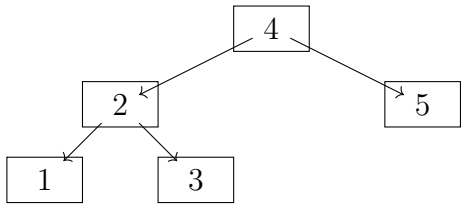
$$O\left(\mathsf{S} + \frac{1}{\sqrt{\epsilon}}\left(\frac{1}{\sqrt{\delta}}\mathsf{U} + \mathsf{C}\right)\right) \tag{2}$$

where all the terms are defined in Equation 1. The ϵ and δ terms have a

[1]It will be a product of two *Johnson graphs*. The exact definition of a Johnson graph is unimportant here.

quadratic speed-up, which you should think of as the same as the quadratic speed-up of Grover's algorithm.

The algorithm requires us to implement the classical steps in a quantum computer. Generically this is possible at the same cost, with some caveats.



(a) A binary tree.

Memory address	Data	Pointer to left child	Pointer to right child
0x01	4	0x02	0x05
0x02	2	0x03	0x04
0x03	1	null	null
0x04	3	null	null
0x05	5	null	null

Memory address	Data	Pointer to left child	Pointer to right child
0x01	4	0x03	0x02
0x02	5	null	null
0x03	2	0x04	0x05
0x04	1	null	null
0x05	3	null	null

(b) Two equivalent memory layouts for the binary tree. An algorithm to build a binary tree might insert new data at the next free slot in allocated memory and arrange pointers from there. This means if the data was inserted in the order $(4, 2, 1, 3, 5)$ it would produce the first layout and $(4, 5, 2, 1, 3)$ would produce the second.

Figure 3: The same data structure, with equivalent but not identical representations of the data in memory.

The quantum random walk (like all quantum algorithms) requires interference between states. The quantum state of a pair of lists (L_x, L_y) will be a ket vector $|L_x\rangle\,|L_y\rangle$, which specifies the precise memory layout of the lists. This must always be exactly the same for interference to occur. In particular, this means the memory layout must not depend on how the list was constructed. This is called "history independence".

To store a sorted list for fast insertion and deletion, good classical choices are a binary tree or a hash table. Neither works directly for a quantum computer, because they are history dependent (see Figure 3). Two approaches to solve this are to create a superposition over all possible layouts [1, 8], or use a fixed layout, so the elements must be re-sorted with every insertion or deletion [6].

Exercise: Show that if we set $R = 1$, the graph as defined at the start of this section is a complete graph, and that the cost of a quantum random walk on this complete graph is the same as Grover's algorithm.

3.3 Quantum Cost Models

The quantum algorithm repeatedly accesses memory, so we must understand the costs of quantum memory[2]. Quantum circuits are often described in the same way as classical circuits, with wires between quantum gates. In a classical circuit, a gate is an object (a set of transistors) through which a signal can propagate at some time and energy cost. In contrast, a "gate" in a quantum computer today is an action performed on a static qubit (e.g.: a laser pulse fired at a small ring of superconducting metal). This action will be controlled by a classical computer, and hence incurs some cost in time, energy, and classical computation.

This matters for memory access. Classically, a carefully-constructed binary tree of switches can access N words of memory and only send a signal through $O(\lg N)$ gates. However, the entire circuit needs $\Theta(N)$ gates. For a quantum computer, this means a cost (energy, and/or classical control cost) of $\Theta(N)$ for each memory access. Moreover, a quantum computer must apply every gate, since the memory access may be in superposition, and in a vague sense, it must access all words of memory in one request.

In short, a sensible and conservative model of the cost of a quantum computer is the number of total gates, not the time to perform the computation. Under this model, the update cost in the quantum random walk is $\Theta(R)$. In fact, re-sorting the list is one of the cheapest options. This leads to a total cost of

$$O\left(R + \frac{\sqrt{|X||Y|}}{\sqrt{R}}\left(R+1\right)\right) \tag{3}$$

[2]This section follows [6].

This increases at all values of R, implying the optimal choice is $R = 1$: Grover's algorithm[3].

Exercise: The action of memory access maps an index i and a list L to $L[i]$. Show that a classical circuit requires $\Omega(|L|)$ gates to do this, if the individual gates have at most 2 inputs.

4 Multi-Grover

Quantum walks do not parallelize well. Instead, the **Multi-Grover** algorithm uses a Grover search over *lists* of R pairs of elements $(x, y) \in X \times Y$ [3]. To check whether a list contains a claw, we use R quantum processors. Each is given one pair (x, y) from the list. The processors act as a sorting network and look for any claws. The cost of this depends on the physical layout of the processors; for now, assume it costs $O(R \log R)$ gates and time $O(\log R)$.

The probability of selecting the claw out of R processors is approximately $\frac{R^2}{|X||Y|}$ (as long as $R \ll |X|$), so the number of grover iterations is the square root of this. Then the total cost is:

$$O\left(\underbrace{\frac{\sqrt{|X||Y|}}{R}}_{\text{number of iterations}} \cdot \underbrace{R \log R}_{\text{cost of each iteration}}\right) = O\left(\sqrt{|X||Y|} \log R\right) \qquad (4)$$

This is approximately the same cost as Grover's algorithm; *however* the total *runtime* is lower by a factor of R. Hence, this parallelizes the attack.

Exercise: In a 2-dimensional grid, a sorting network costs $O(R^{3/2})$ and runs in time $O(R^{1/2})$ to sort. Calculate the total cost and run-time of the Multi-Grover algorithm in 2-dimensions. How well does it parallelize?

References

[1] A. Ambainis, *Quantum walk algorithm for element distinctness*, SIAM J. Computing **37** (2007), 210–239.

[2] Reza Azarderakhsh, Jean-Franois Biasse, Rami El Khatib, Brandon Langenberg, and Benjamin Pring, *Parallelism strategies for the tuneable*

[3] We ignored oracle costs; the quantum walk can help offset oracle costs.

golden-claw finding problem, International Journal of Computer Mathematics: Computer Systems Theory **0** (2021), no. 0, 1–27.

[3] Robert Beals, Stephen Brierley, Oliver Gray, Aram W. Harrow, Samuel Kutin, Noah Linden, Dan Shepherd, and Mark Stather, *Efficient distributed quantum computing*, Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences **469** (2013), no. 2153, 20120686.

[4] Jean-François Biasse, David Jao, and Anirudh Sankar, *A quantum algorithm for computing isogenies between supersingular elliptic curves*, Progress in Cryptology – INDOCRYPT 2014 (Cham) (Willi Meier and Debdeep Mukhopadhyay, eds.), Springer International Publishing, 2014, pp. 428–442.

[5] Jean-François Biasse and Benjamin Pring, *A framework for reducing the overhead of the quantum oracle for use with grover's algorithm with applications to cryptanalysis of SIKE*, Journal of Mathematical Cryptology **15** (2020), no. 1, 143–156.

[6] S. Jaques and J. M. Schanck, *Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE*, CRYPTO 2019, LNCS 11693, 2019, pp. 32–61.

[7] Samuel Jaques and André Schrottenloher, *Low-gate quantum golden collision finding*, Selected Areas in Cryptography (Cham) (Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O'Flynn, eds.), Springer International Publishing, 2021, pp. 329–359.

[8] S. Jeffery, *Frameworks for quantum algorithms*, Ph.D. thesis, University of Waterloo, Waterloo, Ontario, Canada, 2014.

[9] Frédéric Magniez, Ashwin Nayak, Jérémie Roland, and Miklos Santha, *Search via quantum walk*, SIAM Journal on Computing **40** (2011), no. 1, 142–164.

[10] S. Tani, *An improved claw finding algorithm using quantum walk*, Mathematical Foundations of Computer Science – MFCS 2007, LNCS 4708, pp. 548–558.