Revisiting Time-Space Tradeoffs for Function Inversion

Spencer Peters

Noah S.D.

Siyao Guo

Sasha Golovnev

Function Inversion

- Given a function

$$\{1, 2, \ldots, N\} =: [N] \longrightarrow \boxed{f} \longrightarrow [N]$$

Function Inversion

- Given a function, and a point y in its range,

$$\{1, 2, \ldots, N\} =: [N] \longrightarrow \boxed{f} \longrightarrow y \in [N]$$

Function Inversion

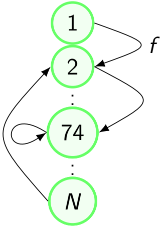
- Given a function, and a point y in its range, find x with $f(x) = y$.

$$\{1, 2, \ldots, N\} =: [N] \ni x \longrightarrow \boxed{\; f \;} \longrightarrow y \in [N]$$

Function Inversion

- Given a function, and a point y in its range, find x with $f(x) = y$.

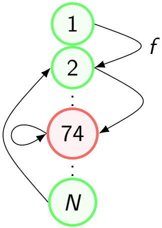
$$\{1, 2, \ldots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$



Function Inversion

- Given a function, and a point y in its range, find x with $f(x) = y$.

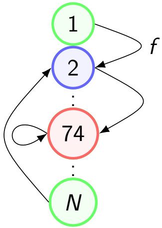
$$\{1, 2, \ldots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$



Function Inversion

- Given a function, and a point y in its range, find x with $f(x) = y$.

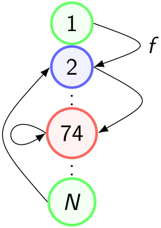
$$\{1, 2, \ldots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$



Function Inversion

- Given a function, and a point y in its range, find x with $f(x) = y$.

$$\{1, 2, \dots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$

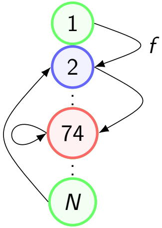


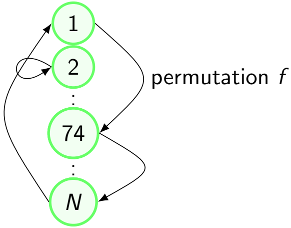
- The study of this *black-box* function inversion problem was initiated by Martin Hellman  in 1980 [Hel80].

Function Inversion

- Given a function, and a point y in its range, find x with $f(x) = y$.

$$\{1, 2, \ldots, N\} =: [N] \ni x \longrightarrow \boxed{f} \longrightarrow y \in [N]$$

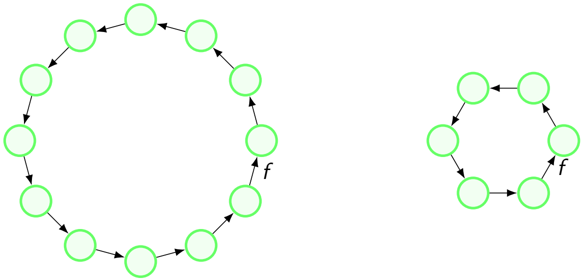




- The study of this *black-box* function inversion problem was initiated by Martin Hellman  in 1980 [Hel80].

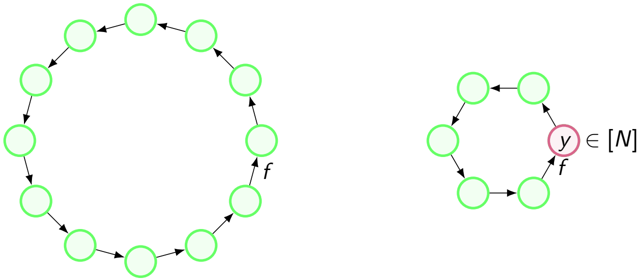
Hellman's algorithm

- If f is a permutation, its *graph* is a disjoint union of cycles.



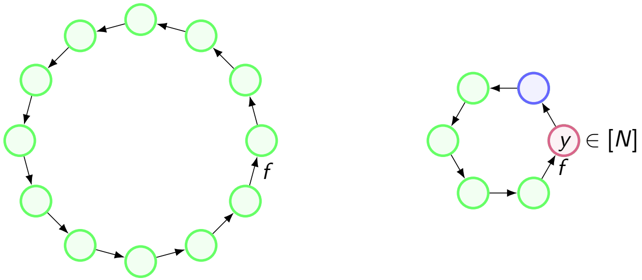
Hellman's algorithm

- If f is a permutation, its *graph* is a disjoint union of cycles.



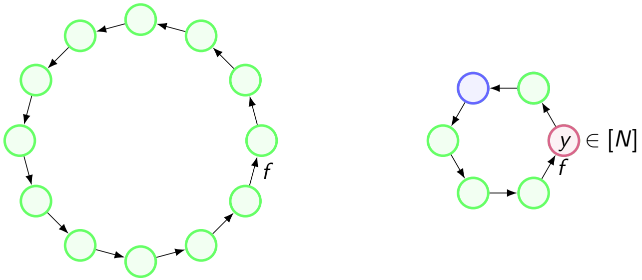
Hellman's algorithm

- If f is a permutation, its *graph* is a disjoint union of cycles.



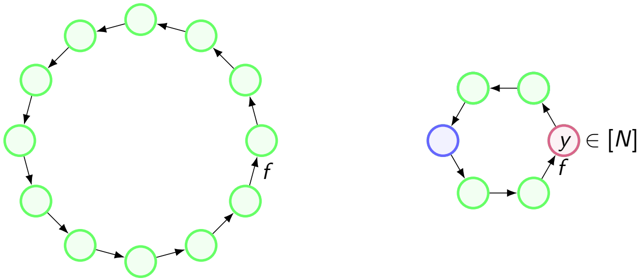
Hellman's algorithm

- If f is a permutation, its *graph* is a disjoint union of cycles.



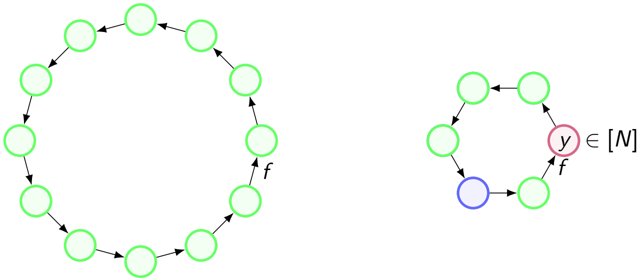
Hellman's algorithm

- If f is a permutation, its *graph* is a disjoint union of cycles.



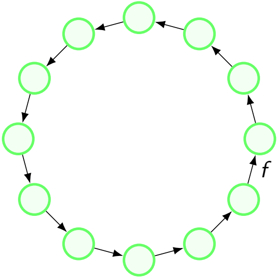
Hellman's algorithm

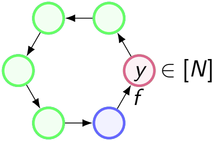
- If f is a permutation, its *graph* is a disjoint union of cycles.



Hellman's algorithm

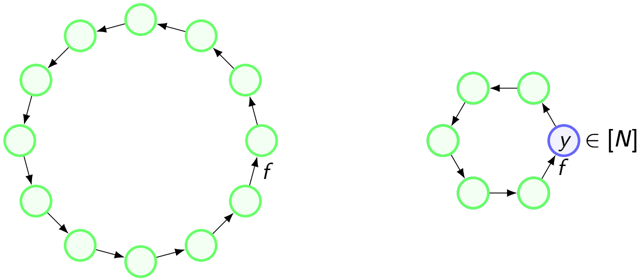
- If f is a permutation, its *graph* is a disjoint union of cycles.





Hellman's algorithm

- If f is a permutation, its *graph* is a disjoint union of cycles.

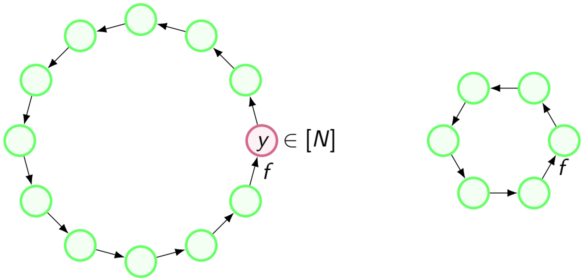


Hellman's algorithm

- What if y is on a large cycle?

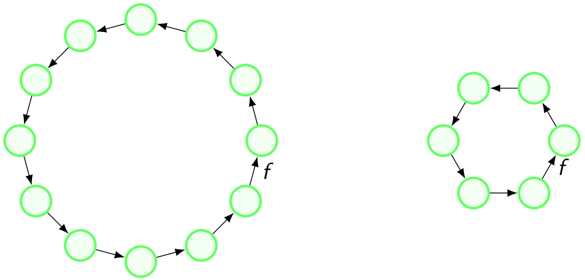
Hellman's algorithm

- What if y is on a large cycle?



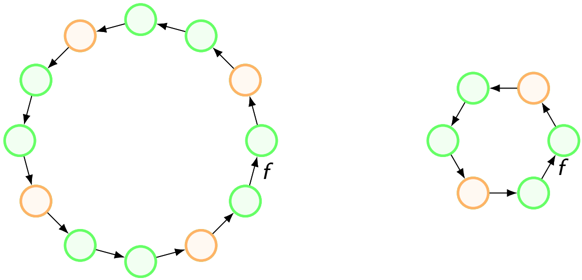
Hellman's algorithm

- What if y is on a large cycle?



Hellman's algorithm

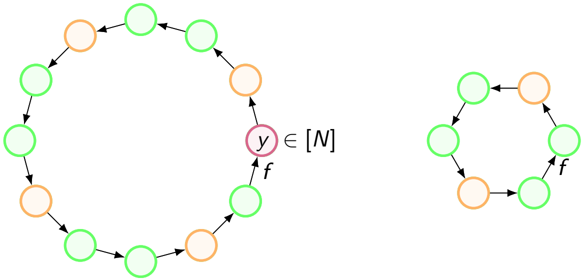
- What if y is on a large cycle?



- In a preprocessing step, store points uniformly spaced around each cycle.

Hellman's algorithm

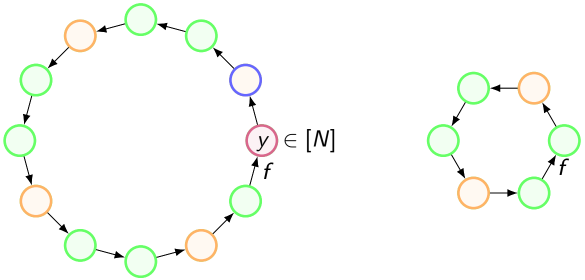
- What if y is on a large cycle?



- In a preprocessing step, store points uniformly spaced around each cycle.

Hellman's algorithm

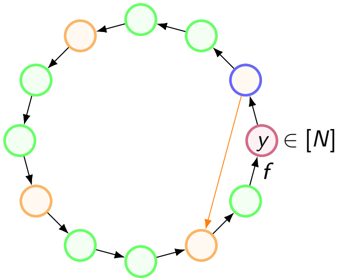
- What if y is on a large cycle?



- In a preprocessing step, store points uniformly spaced around each cycle.

Hellman's algorithm

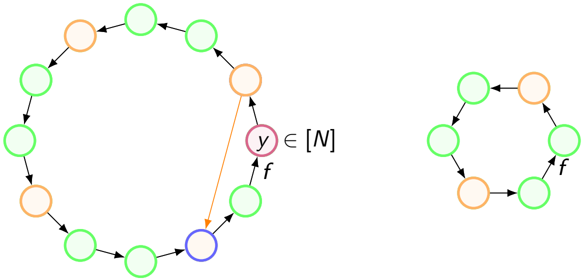
- What if y is on a large cycle?



- In a preprocessing step, store points uniformly spaced around each cycle.

Hellman's algorithm

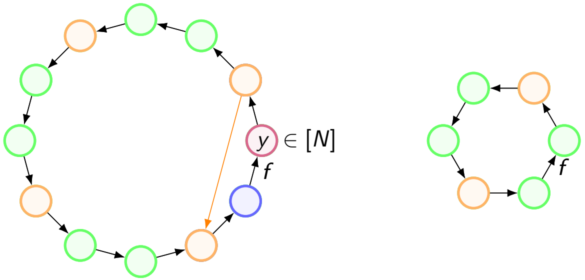
- What if y is on a large cycle?



- In a preprocessing step, store points uniformly spaced around each cycle.

Hellman's algorithm

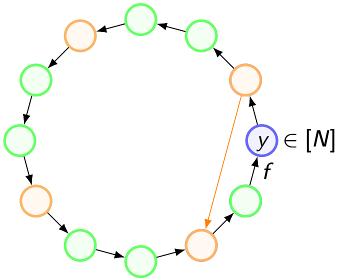
- What if y is on a large cycle?

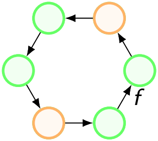


- In a preprocessing step, store points uniformly spaced around each cycle.

Hellman's algorithm

- What if y is on a large cycle?





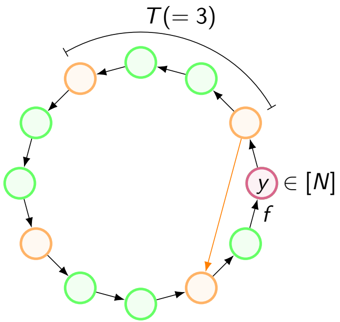
- In a preprocessing step, store points uniformly spaced around each cycle.

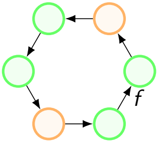
Analysis

- If we space the stored points T hops apart:

Analysis

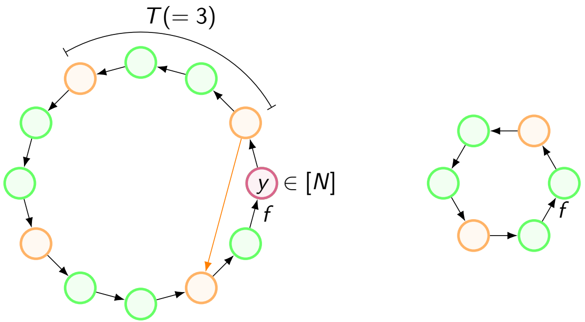
- If we space the stored points T hops apart:





Analysis

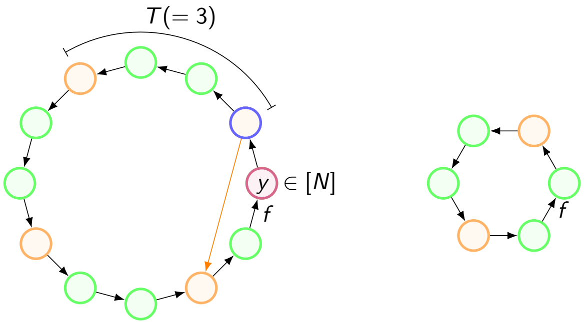
- If we space the stored points T hops apart:



- We need T evaluations of f to invert y.

Analysis

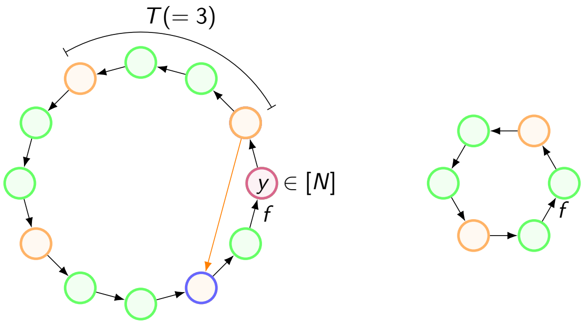
- If we space the stored points T hops apart:



- We need T evaluations of f to invert y.

Analysis

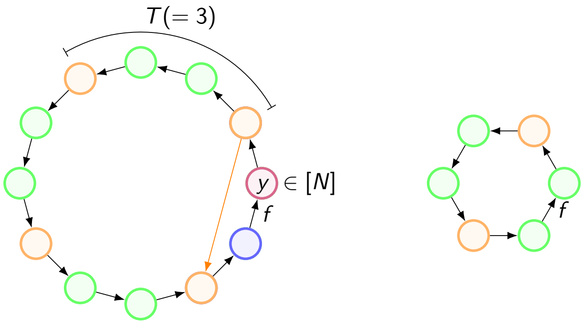
- If we space the stored points T hops apart:



- We need T evaluations of f to invert y.

Analysis

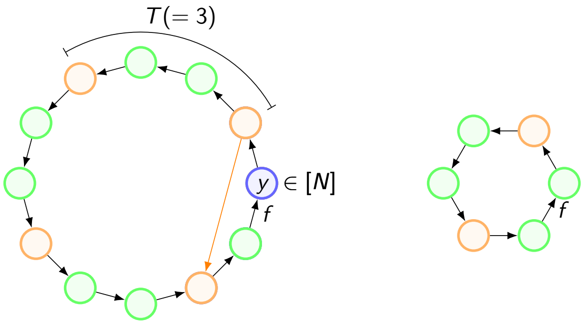
- If we space the stored points T hops apart:



- We need T evaluations of f to invert y.

Analysis

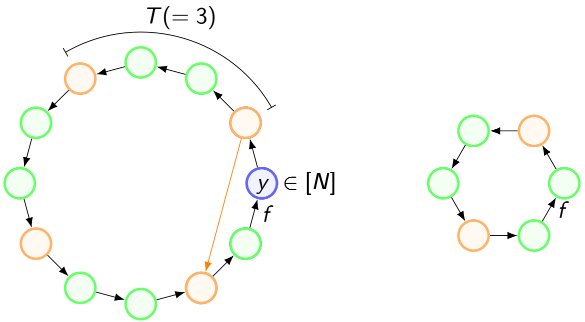
- If we space the stored points T hops apart:



- We need T evaluations of f to invert y.

Analysis

- If we space the stored points T hops apart:



- We need T evaluations of f to invert y.
- We need to store about N/T points total.

Stepping back

- **Goal:** design a pair of algorithms $(\mathcal{P}, \mathcal{A})$ such that for all f and all $y \in f([N])$,

$$\Pr[\alpha \leftarrow \mathcal{P}(f); x \leftarrow \mathcal{A}^f(\alpha, y); f(x) = y] \geq \text{constant}.$$

Stepping back

- **Goal:** design a pair of algorithms $(\mathcal{P}, \mathcal{A})$ such that for all f and all $y \in f([N])$,

$$\Pr[\alpha \leftarrow \mathcal{P}(f); x \leftarrow \mathcal{A}^f(\alpha, y); f(x) = y] \geq \text{constant}.$$

- In this model, $\mathcal{P}$ and $\mathcal{A}$ have **unbounded** computational power.

Stepping back

- **Goal:** design a pair of algorithms $(\mathcal{P}, \mathcal{A})$ such that for all f and all $y \in f([N])$,

$$\Pr[\alpha \leftarrow \mathcal{P}(f); x \leftarrow \mathcal{A}^f(\alpha, y); f(x) = y] \geq \text{constant.}$$

- In this model, $\mathcal{P}$ and $\mathcal{A}$ have **unbounded** computational power.
- We aim to **minimize** the bitlength S of α, and the number of queries T that $\mathcal{A}$ makes to f.

Application scenarios

- The NSA wants to break a widely used cryptographic function, such as AES-128

Application scenarios

- The NSA wants to break a widely used cryptographic function, such as AES-128
- Hackers want to recover passwords from a stolen database of password hashes (Rainbow Tables)

Application scenarios

- The NSA wants to break a widely used cryptographic function, such as AES-128
- Hackers want to recover passwords from a stolen database of password hashes (Rainbow Tables)
- Theoretical computer scientists want better algorithms for 3-SUM [GGH$^+$20], 

Application scenarios

- The NSA wants to break a widely used cryptographic function, such as AES-128
- Hackers want to recover passwords from a stolen database of password hashes (Rainbow Tables)
- Theoretical computer scientists want better algorithms for 3-SUM [GGH+20],     
  multiparty pointer jumping [CK19],  

Application scenarios

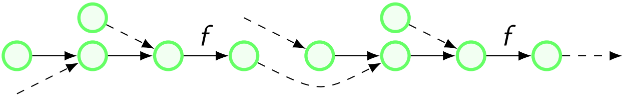
- The NSA wants to break a widely used cryptographic function, such as AES-128
- Hackers want to recover passwords from a stolen database of password hashes (Rainbow Tables)
- Theoretical computer scientists want better algorithms for 3-SUM [GGH+20],  multiparty pointer jumping [CK19],  systematic substring search [CK19], ...

Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.

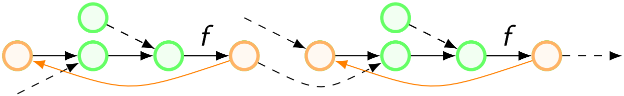
Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.



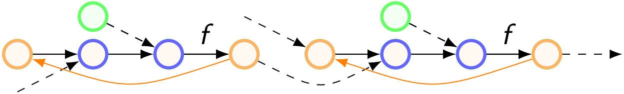
Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.



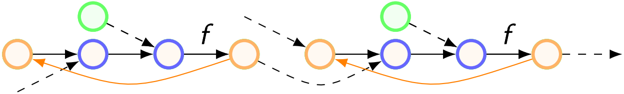
Beyond permutations

- Preprocessing stores the endpoints of disjoint paths.



Beyond permutations

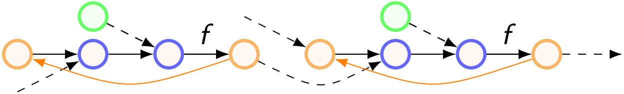
- Preprocessing stores the endpoints of disjoint paths.



- No longer possible to cover the entire range.

Beyond permutations

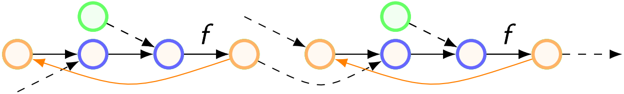
- Preprocessing stores the endpoints of disjoint paths.



- No longer possible to cover the entire range.
- But, can still cover a small fraction with disjoint paths.

Beyond permutations

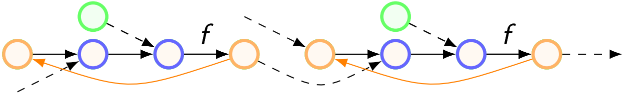
- Preprocessing stores the endpoints of disjoint paths.



- No longer possible to cover the entire range.
- But, can still cover a small fraction with disjoint paths.
- To boost the coverage, observe that inverting $g \circ f$ on $g(y)$ is often enough to invert f on y.

Beyond permutations

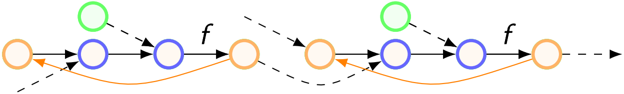
- Preprocessing stores the endpoints of disjoint paths.



- No longer possible to cover the entire range.
- But, can still cover a small fraction with disjoint paths.
- To boost the coverage, observe that inverting $g \circ f$ on $g(y)$ is often enough to invert f on y.
- So, can repeatedly apply the basic scheme to many compositions $g_i \circ f$, for suitably chosen “rerandomization” functions g_i.

Beyond permutations

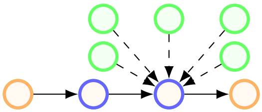
- Preprocessing stores the endpoints of disjoint paths.



- No longer possible to cover the entire range.
- But, can still cover a small fraction with disjoint paths.
- To boost the coverage, observe that inverting $g \circ f$ on $g(y)$ is often enough to invert f on y.
- So, can repeatedly apply the basic scheme to many compositions $g_i \circ f$, for suitably chosen "rerandomization" functions g_i.
- For *random* functions, Hellman showed (heuristically) this can be made to work. 

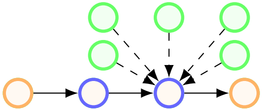
Beyond permutations

- Hellman's argument fails for arbitrary functions.



Beyond permutations

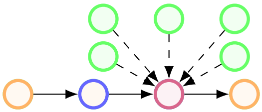
- Hellman's argument fails for arbitrary functions.



- Arbitrary functions can have "junction points" with many inverses.

Beyond permutations

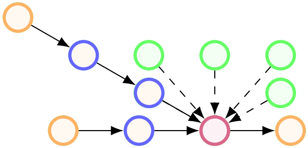
- Hellman's argument fails for arbitrary functions.



- Arbitrary functions can have “junction points” with many inverses.

Beyond permutations

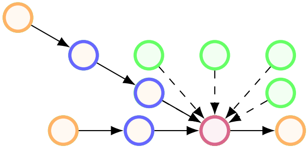
- Hellman's argument fails for arbitrary functions.



- Arbitrary functions can have "junction points" with many inverses.

Beyond permutations

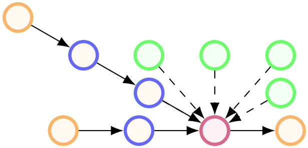
- Hellman's argument fails for arbitrary functions.



- Arbitrary functions can have "junction points" with many inverses.
- Paths collide at these points, causing all sorts of problems.
- Fiat and Naor deal with this by storing $\alpha = (\alpha', L)$, where L contains junction points along with their inverses.

Beyond permutations

- Hellman's argument fails for arbitrary functions.



- Arbitrary functions can have "junction points" with many inverses.
- Paths collide at these points, causing all sorts of problems.
- Fiat and Naor deal with this by storing $\alpha = (\alpha', L)$, where L contains junction points along with their inverses.
- Intuitively, α' is the data structure for a *restriction* of f that avoids the junction points in L.
- More precisely, the "rerandomization" functions are sampled using rejection sampling so that their range is $[N] - L$.

The Story So Far

Result		Applies To	Tradeoff	Key Point
Hellman 	1980	permutations	$T \lesssim N/S$	$S = T \lesssim \sqrt{N}$

The Story So Far

Result	Applies To	Tradeoff	Key Point
Hellman 1980 	permutations	$T \lesssim N/S$	$S = T \lesssim \sqrt{N}$
Yao 1990 	permutations	$T \gtrsim N/S$	$S = T \gtrsim \sqrt{N}$

The Story So Far

Result		Applies To	Tradeoff	Key Point
Hellman 	1980	permutations	$T \lesssim N/S$	$S = T \lesssim \sqrt{N}$
Yao 1990		permutations	$T \gtrsim N/S$	$S = T \gtrsim \sqrt{N}$
Hellman 	1980	random f	$T \lesssim N^2/S^2$	$S = T \lesssim N^{2/3}$

The Story So Far

Result	Applies To	Tradeoff	Key Point
Hellman 1980 	permutations	$T \lesssim N/S$	$S = T \lesssim \sqrt{N}$
Yao 1990 	permutations	$T \gtrsim N/S$	$S = T \gtrsim \sqrt{N}$
Hellman 1980 	random f	$T \lesssim N^2/S^2$	$S = T \lesssim N^{2/3}$
Fiat-Naor 1991  	all functions	$T \lesssim N^3/S^3$	$S = T \lesssim N^{3/4}$

The Story So Far

Result	Applies To	Tradeoff	Key Point
Hellman 1980 	permutations	$T \lesssim N/S$	$S = T \lesssim \sqrt{N}$
Yao 1990 	permutations	$T \gtrsim N/S$	$S = T \gtrsim \sqrt{N}$
Hellman 1980 	random f	$T \lesssim N^2/S^2$	$S = T \lesssim N^{2/3}$
Fiat-Naor 1991  	all functions	$T \lesssim N^3/S^3$	$S = T \lesssim N^{3/4}$

- Q: Can we improve Fiat-Naor? Can we improve Yao's lower bound?

The Story So Far

Result	Applies To	Tradeoff	Key Point
Hellman 1980 	permutations	$T \lesssim N/S$	$S = T \lesssim \sqrt{N}$
Yao 1990 	permutations	$T \gtrsim N/S$	$S = T \gtrsim \sqrt{N}$
Hellman 1980 	random f	$T \lesssim N^2/S^2$	$S = T \lesssim N^{2/3}$
Fiat-Naor 1991  	all functions	$T \lesssim N^3/S^3$	$S = T \lesssim N^{3/4}$

- Q: Can we improve Fiat-Naor? Can we improve Yao's lower bound?
- A: Sort of and sort of!

Part 1: A Simple Improvement to Fiat-Naor

Result		Applies To	Tradeoff	Key Point
Hellman 	1980	permutations	$T \lesssim N/S$	$S = T \lesssim \sqrt{N}$
Yao 1990		permutations	$T \gtrsim N/S$	$S = T \gtrsim \sqrt{N}$
Hellman 	1980	random f	$T \lesssim N^2/S^2$	$S = T \lesssim N^{2/3}$
Fiat-Naor 	1991 	all functions	$T \lesssim N^3/S^3$	$S = T \lesssim N^{3/4}$

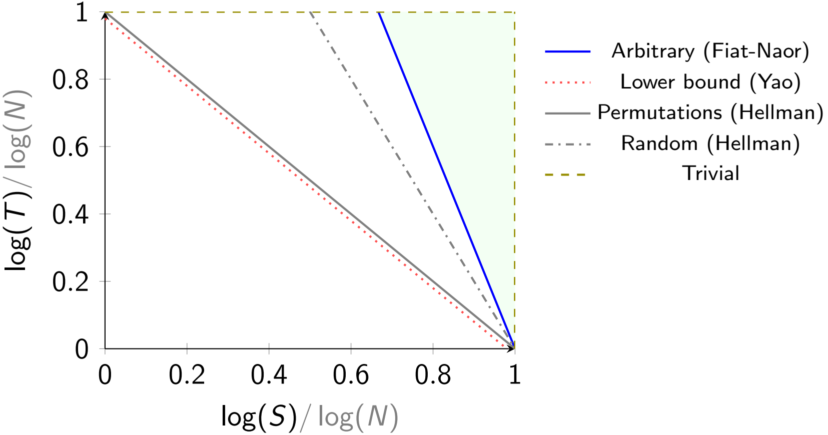
Part 1: A Simple Improvement to Fiat-Naor

Result		Applies To	Tradeoff	Key Point
Hellman 	1980	permutations	$T \lesssim N/S$	$S = T \lesssim \sqrt{N}$
Yao 1990		permutations	$T \gtrsim N/S$	$S = T \gtrsim \sqrt{N}$
Hellman 	1980	random f	$T \lesssim N^2/S^2$	$S = T \lesssim N^{2/3}$
Fiat-Naor 	1991 	all functions	$T \lesssim N^3/S^3$	$S = T \lesssim N^{3/4}$
This work		all functions	$T \lesssim N^3/(S^2 T)$	$S = T \lesssim N^{3/4}$

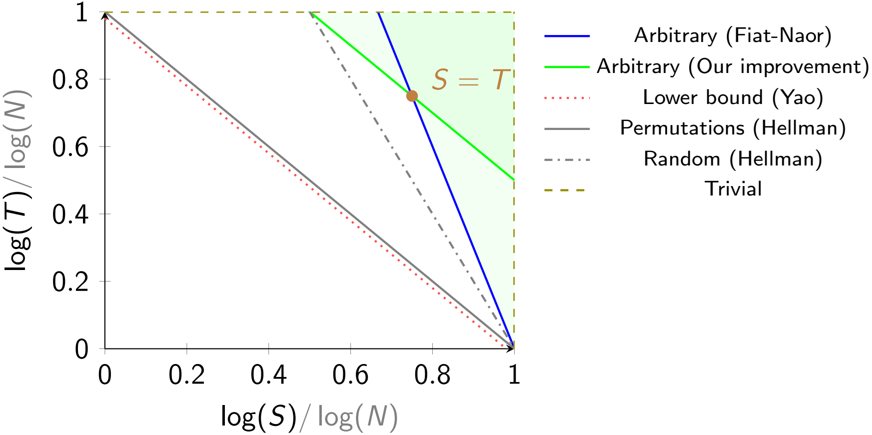
Part 1: A Simple Improvement to Fiat-Naor

Result		Applies To	Tradeoff	Key Point
Hellman 	1980	permutations	$T \lesssim N/S$	$S = T \lesssim \sqrt{N}$
Yao 1990 		permutations	$T \gtrsim N/S$	$S = T \gtrsim \sqrt{N}$
Hellman 	1980	random f	$T \lesssim N^2/S^2$	$S = T \lesssim N^{2/3}$
Fiat-Naor  	1991	all functions	$T \lesssim N^3/S^3$	$S = T \lesssim N^{3/4}$
This work		all functions	$T \lesssim N^3/(S^2T)$ $T \lesssim N^{3/2}/S$	$S = T \lesssim N^{3/4}$

A Simple Improvement to Fiat-Naor



A Simple Improvement to Fiat-Naor



A Simple Improvement to Fiat-Naor

- Recall that Fiat and Naor's preprocessing and online algorithms must agree on a list L of "junction points".

A Simple Improvement to Fiat-Naor

- Recall that Fiat and Naor's preprocessing and online algorithms must agree on a list L of "junction points".
- We observe that the tradeoff $T \lesssim N^3/S^3$ comes from

$$T \lesssim \frac{1}{|L|} \cdot \frac{N^3}{S^2}.$$

A Simple Improvement to Fiat-Naor

- Recall that Fiat and Naor's preprocessing and online algorithms must agree on a list L of "junction points".
- We observe that the tradeoff $T \lesssim N^3/S^3$ comes from

$$T \lesssim \frac{1}{|L|} \cdot \frac{N^3}{S^2}.$$

- Fiat and Naor get $|L| \simeq S$, but this is the hard limit, since L needs to fit into S-bit advice α.
- Or does it?

A Simple Improvement to Fiat-Naor

- Fiat and Naor's list L actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.

A Simple Improvement to Fiat-Naor

- Fiat and Naor's list L actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.
- Our idea: Instead of reading L from α, $\mathcal{A}$ recovers L by evaluating f on the same random points x_i.

A Simple Improvement to Fiat-Naor

- Fiat and Naor's list L actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.
- Our idea: Instead of reading L from α, $\mathcal{A}$ recovers L by evaluating f on the same random points x_i.
- This allows $|L| \simeq T$, so we can get $T \lesssim N^3/(S^2 T)$, or

$$T \lesssim N^{3/2}/S.$$

A Simple Improvement to Fiat-Naor

- Fiat and Naor's list L actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.
- Our idea: Instead of reading L from α, $\mathcal{A}$ recovers L by evaluating f on the same random points x_i.
- This allows $|L| \simeq T$, so we can get $T \lesssim N^3/(S^2 T)$, or

$$T \lesssim N^{3/2}/S.$$

- But I've cheated here...

A Simple Improvement to Fiat-Naor

- Fiat and Naor's list L actually consists of images $f(x_i)$ of random points $x_i \sim [N]$.
- Our idea: Instead of reading L from α, $\mathcal{A}$ recovers L by evaluating f on the same random points x_i.
- This allows $|L| \simeq T$, so we can get $T \lesssim N^3/(S^2 T)$, or

$$T \lesssim N^{3/2}/S.$$

- But I've cheated here...
- How do $\mathcal{A}$ and $\mathcal{P}$ agree on the *same* list of random values x_i?

Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality.

Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality.
- More precisely, shared randomness costs an additive $O(\log N)$ in S.

Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality.
- More precisely, shared randomness costs an additive $O(\log N)$ in S.
- The proof adapts *Newman's lemma* [New91]  from communication complexity.

Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality.
- More precisely, shared randomness costs an additive $O(\log N)$ in S.
- The proof adapts *Newman's lemma* [New91]  from communication complexity.
- The idea: rather than storing the random bits in full, store a random *index* into a fixed list of strings.

Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality.
- More precisely, shared randomness costs an additive $O(\log N)$ in S.
- The proof adapts *Newman's lemma* [New91]  from communication complexity.
- The idea: rather than storing the random bits in full, store a random *index* into a fixed list of strings.
- If the number $\Sigma = O(N)$ of fixed strings is large enough, this will be almost as good as fully random.

Sharing Randomness

- We show that we can assume *shared randomness* without loss of generality.
- More precisely, shared randomness costs an additive $O(\log N)$ in S.
- The proof adapts *Newman's lemma* [New91]  from communication complexity.
- The idea: rather than storing the random bits in full, store a random *index* into a fixed list of strings.
- If the number $\Sigma = O(N)$ of fixed strings is large enough, this will be almost as good as fully random.
- In practice, can instantiate a random oracle.

Part 2: A Lower Bound

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.

Part 2: A Lower Bound

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.

Part 2: A Lower Bound

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.
- Even improving Yao's bound just for *non-adaptive* algorithms would do it!

Part 2: A Lower Bound

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.
- Even improving Yao's bound just for *non-adaptive* algorithms would do it!
- $\mathcal{A}$ is non-adaptive if its evaluation points $x_1, \ldots, x_T$ are chosen up front, before any evaluations of f are seen.

Part 2: A Lower Bound

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.
- Even improving Yao's bound just for *non-adaptive* algorithms would do it!
- $\mathcal{A}$ is non-adaptive if its evaluation points $x_1, \ldots, x_T$ are chosen up front, before any evaluations of f are seen.
- Non-adaptive algorithms seem very weak. Hellman's algorithm is *very* adaptive.

Part 2: A Lower Bound

- Recall that Yao's lower bound (for inverting *arbitrary* functions) hasn't been improved in 30+ years.
- Corrigan-Gibbs and Kogan   [CK19]: any small improvement $\implies$ new lower bounds in circuit complexity.
- Even improving Yao's bound just for *non-adaptive* algorithms would do it!
- $\mathcal{A}$ is non-adaptive if its evaluation points $x_1, \ldots, x_T$ are chosen up front, before any evaluations of f are seen.
- Non-adaptive algorithms seem very weak. Hellman's algorithm is *very* adaptive.
- Corrigan-Gibbs and Kogan speculated that there is no non-adaptive algorithm with

$$S = o(N \log N) \text{ and } T = o(N).$$

A Simple Non-Adaptive Algorithm

- Corrigan-Gibbs and Kogan speculated that there is no non-adaptive algorithm with

$$S = o(N \log N) \text{ and } T = o(N).$$

- We observe that there IS in fact a very simple algorithm, that (barely!) outperforms the trivial inverter.

A Simple Non-Adaptive Algorithm

- Corrigan-Gibbs and Kogan speculated that there is no non-adaptive algorithm with

$$S = o(N \log N) \text{ and } T = o(N).$$

- We observe that there IS in fact a very simple algorithm, that (barely!) outperforms the trivial inverter.
- For each range element, just store all but $\log(T)$ bits of one of its inverses (implicitly defining T candidate inverses). This achieves $S = O(N \log(N/T))$.

A Simple Non-Adaptive Algorithm

- Corrigan-Gibbs and Kogan speculated that there is no non-adaptive algorithm with

$$S = o(N \log N) \text{ and } T = o(N).$$

- We observe that there IS in fact a very simple algorithm, that (barely!) outperforms the trivial inverter.
- For each range element, just store all but $\log(T)$ bits of one of its inverses (implicitly defining T candidate inverses). This achieves $S = O(N \log(N/T))$.

A Matching Lower Bound

- The very simple algorithm is a *guess-and-check algorithm*. That is, it can be thought of as making *no* queries to f, and returning T candidate inverses, one of which is guaranteed to be correct.

A Matching Lower Bound

- The very simple algorithm is a *guess-and-check algorithm*. That is, it can be thought of as making *no* queries to f, and returning T candidate inverses, one of which is guaranteed to be correct.
- We show that all *guess-and-check* algorithms satisfy the matching lower bound $S = \Omega(N \log(N/T))$.

A Matching Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters S, T. We use a compression argument.

A Matching Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters S, T. We use a compression argument.
- Theorem: Can encode any permutation f using $S + N \log T$ bits.

A Matching Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters S, T. We use a compression argument.
- Theorem: Can encode any permutation f using $S + N \log T$ bits.

$$\begin{aligned} &\implies S + N \log T = \Omega(N \log N) \\ &\implies S = \Omega(N \log(N/T)). \end{aligned}$$

A Matching Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters S, T. We use a compression argument.
- Theorem: Can encode any permutation f using $S + N \log T$ bits.

$$\implies S + N \log T = \Omega(N \log N)$$
$$\implies S = \Omega(N \log(N/T)).$$

- Proof:
  - Encoder computes $\alpha \leftarrow \mathcal{P}(f)$.

A Matching Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters S, T. We use a compression argument.
- Theorem: Can encode any permutation f using $S + N \log T$ bits.

$$\implies S + N \log T = \Omega(N \log N)$$
$$\implies S = \Omega(N \log(N/T)).$$

- Proof:
  - Encoder computes $\alpha \leftarrow \mathcal{P}(f)$.
  - For each $y \in [N]$, encoder runs $\mathcal{A}(\alpha, y)$ and receives $x_1, \ldots, x_T$. It writes down the $i_y \in [T]$ that satisfies $f(x_{i_y}) = y$.

A Matching Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters S, T. We use a compression argument.
- Theorem: Can encode any permutation f using $S + N \log T$ bits.

$$\implies S + N \log T = \Omega(N \log N)$$
$$\implies S = \Omega(N \log(N/T)).$$

- Proof:
  - Encoder computes $\alpha \leftarrow \mathcal{P}(f)$.
  - For each $y \in [N]$, encoder runs $\mathcal{A}(\alpha, y)$ and receives $x_1, \ldots, x_T$. It writes down the $i_y \in [T]$ that satisfies $f(x_{i_y}) = y$.
  - Encoding is $(\alpha, i_1, \ldots, i_N)$.

A Matching Lower Bound

- For simplicity, assume a guess-and-check algorithm that always succeeds, with parameters S, T. We use a compression argument.
- Theorem: Can encode any permutation f using $S + N \log T$ bits.

$$\implies S + N \log T = \Omega(N \log N)$$
$$\implies S = \Omega(N \log(N/T)).$$

- Proof:
  - Encoder computes $\alpha \leftarrow \mathcal{P}(f)$.
  - For each $y \in [N]$, encoder runs $\mathcal{A}(\alpha, y)$ and receives $x_1, \ldots, x_T$. It writes down the $i_y \in [T]$ that satisfies $f(x_{i_y}) = y$.
  - Encoding is $(\alpha, i_1, \ldots, i_N)$.
  - For each y, decoder again runs $\mathcal{A}(\alpha, y)$ and receives $x_1, \ldots, x_T$. It sets $f^{-1}(y) = x_{i_y}$.

Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

algorithm for inverting arbitrary functions.

Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

algorithm for inverting arbitrary functions.

- Shared randomness is essentially free

Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

  algorithm for inverting arbitrary functions.

- Shared randomness is essentially free

- We observe there is a very simple nontrivial non-adaptive function inversion algorithm, and show it is best possible among *guess-and-check* algorithms.

Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

  algorithm for inverting arbitrary functions.

- Shared randomness is essentially free

- We observe there is a very simple nontrivial non-adaptive function inversion algorithm, and show it is best possible among *guess-and-check* algorithms.

- Not in this talk:
  - Search-to-decision reductions

Function Inversion Revisited: Our Results

- Improving Fiat-Naor: a

$$T \lesssim N^{3/2}/S$$

  algorithm for inverting arbitrary functions.

- Shared randomness is essentially free
- We observe there is a very simple nontrivial non-adaptive function inversion algorithm, and show it is best possible among *guess-and-check* algorithms.
- Not in this talk:
  - Search-to-decision reductions
  - A larger codomain does not make the problem harder (us and [CK19])  

Open Problems

- Only one real open problem–

Open Problems

- Only one real open problem–**close the gap!**

Open Problems

- Only one real open problem–**close the gap!**
- Moonshots:
  - Improve Yao's lower bound against (general) non-adaptive algorithms?

Open Problems

- Only one real open problem–**close the gap!**
- Moonshots:
  - Improve Yao's lower bound against (general) non-adaptive algorithms?
  - $S^2 T = N^2$ algorithm for worst-case function inversion?

Open Problems

- Only one real open problem–**close the gap!**
- Moonshots:
  - Improve Yao's lower bound against (general) non-adaptive algorithms?
  - $S^2 T = N^2$ algorithm for worst-case function inversion?
- Possibly more tractable:
  - Better algorithms for inverting a small fraction of the range? That is, improving on De, Trevisan and Tulsiani [DTT10]?

Open Problems

- Only one real open problem–**close the gap!**
- Moonshots:
  - Improve Yao's lower bound against (general) non-adaptive algorithms?
  - $S^2T = N^2$ algorithm for worst-case function inversion?
- Possibly more tractable:
  - Better algorithms for inverting a small fraction of the range? That is, improving on De, Trevisan and Tulsiani [DTT10]?
  - Better algorithms for injective functions?

Thank you!

I'm happy to take additional questions offline. You can ping me at `speters@cs.cornell.edu`.

References I

Henry Corrigan-Gibbs and Dmitry Kogan.
The function-inversion problem: Barriers and opportunities.
In *TCC*, 2019.

Anindya De, Luca Trevisan, and Madhur Tulsiani.
Time space tradeoffs for attacks against one-way functions and PRGs.
In *CRYPTO*, 2010.

Alexander Golovnev, Siyao Guo, Thibaut Horel, Sunoo Park, and Vinod Vaikuntanathan.
Data structures meet cryptography: 3SUM with preprocessing.
In *STOC*, 2020.

References II

Alexander Golovnev, Siyao Guo, Spencer Peters, and Noah Stephens-Davidowitz.
Revisiting time-space tradeoffs for function inversion.
Available at https://eccc.weizmann.ac.il/report/2022/145/, 2022.

Martin Hellman.
A cryptanalytic time-memory trade-off.
IEEE Transactions on Information Theory, 26(4):401–406, 1980.

Ilan Newman.
Private vs. common random bits in communication complexity.
Information processing letters, 39(2):67–71, 1991.