High-performance pseudorandomness

Rasmus Pagh

IT UNIVERSITY OF COPENHAGEN

Work supported by:

DET FRIE FORSKNINGSRÅD

European Research Council
Established by the European Commission
Supporting top researchers from anywhere in the world

Agenda

- Basic motivation and definitions.
- Great theoretical results - useless in practice?
- <u>New developments</u>:
  - Simple tabulation hashing.
  - Semi-explicit unbalanced expanders.
  - Pseudorandom generation.
- Open questions.

Motivating application:
Load balancing

- **Goal**:
  Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

Motivating application:
Load balancing

- **Goal**:
  Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

- **Examples**: Hash tables, SSDs, distributed storage, distributed computation, network routing, parallel algorithms, …

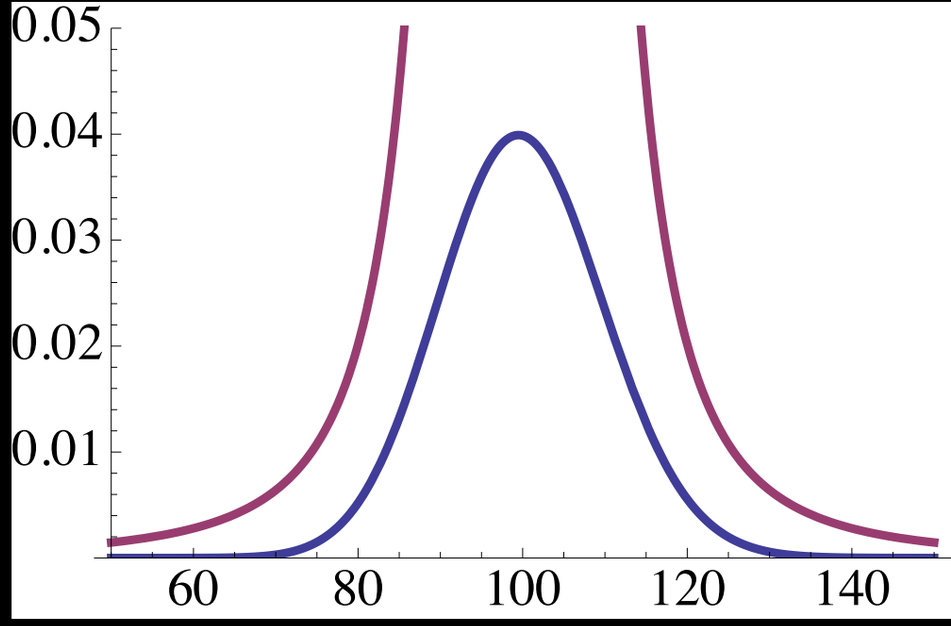
Motivating application:
Load balancing

- **Goal**:
  Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

- **Examples**: Hash tables, SSDs, distributed storage, distributed computation, network routing, parallel algorithms, …

- **Main tool**: Random choice of assignment.

Motivating application:
Load balancing

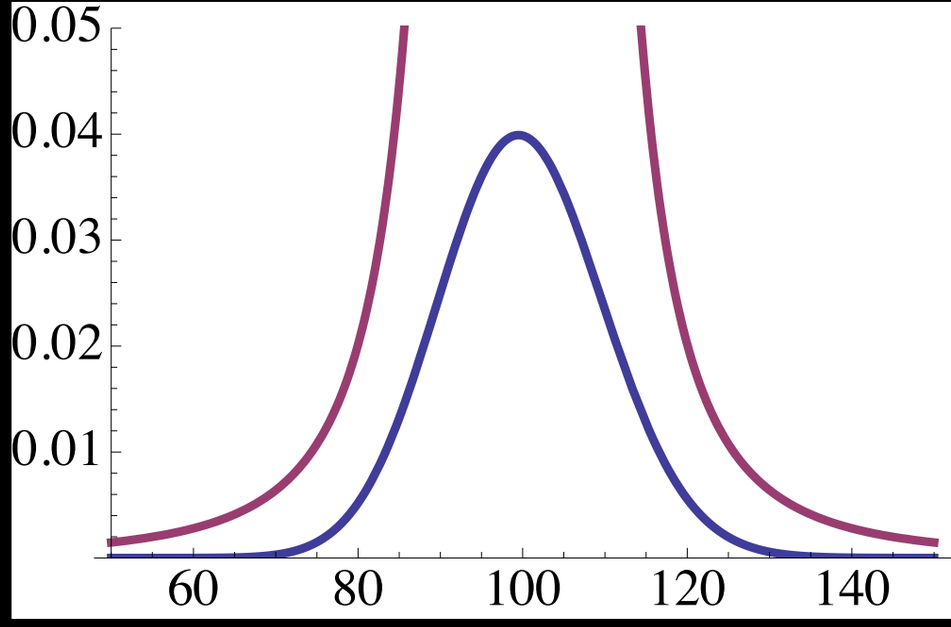


- **Goal**:
  Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

- **Examples**: Hash tables, SSDs, distributed storage, distributed computation, network routing, parallel algorithms, …

- **Main tool**: Random choice of assignment.

Motivating application:
Load balancing



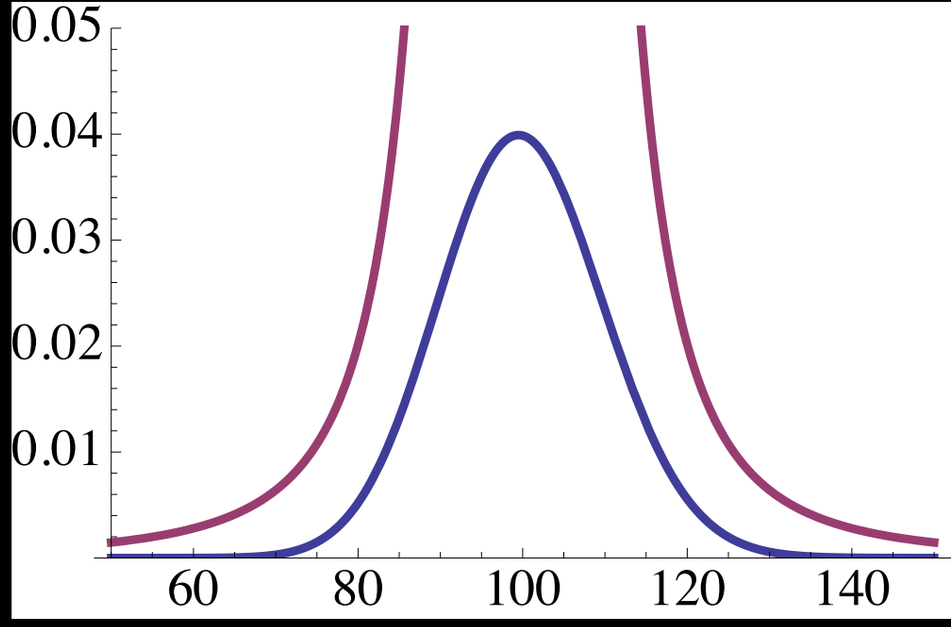
- **Goal**:
  Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

This talk: Items are strings in Σ^n

- **Examples**: Hash tables, SSDs, distributed storage, distributed computation, network routing, parallel algorithms, …

- **Main tool**: Random choice of assignment.

Motivating application:
Load balancing



- **Goal**:
  Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

This talk: Items are strings in Σ^n

- **Exampl**[illegible] ted storage [illegible] ork routing, [illegible]

No significant loss of generality: Can map to (say) 16-byte strings with extremely small risk of collision

- **Main tool**: Random choice of assignment.

Two interfaces to pseudorandomness

- **Basic object**:
  Function f_s: $\Sigma^n \rightarrow D$, indexed by a random seed s.
  (Often implemented with "salt": $f_s(x) = \mathrm{hash}(s;x)$.)

1. **Random access** (hashing):
   Compute $f_s(x)$ for a given string x.

2. **Sequential access** (pseudorandom generator):
   Generate *some* sequence $f_s(x_1)$, $f_s(x_2)$, …

What does theory say?

What does theory say?

On Randomness in Hash Functions

Martin Dietzfelbinger
Technische Universität Ilmenau

Linear hash functions

- ```
  NodeIndex hashfunc(int a, short c)
  { return ((1+8+64)*a + (1+4+16)*c) % HASHSIZE;
  }
  ```

  HASHSIZE is prime
  ```

Linear hash functions

- Associate key x with a vector v_x in F^k, field F.

  For random a in F^k, compute $a \cdot x$.

```
NodeIndex hashfunc(int a, short c)
{ return ((1+8+64)*a + (1+4+16)*c) % HASHSIZE;
}
```

HASHSIZE is prime

Linear hash functions

- Associate key x with a vector v_x in $\mathbf{F}^k$, field $\mathbf{F}$.

  For random a in $\mathbf{F}^k$, compute $a \cdot x$.

This talk: $\mathbf{F}$ is a field with constant time operations

Linear hash functions

- Associate key x with a vector v_x in $\mathbf{F}^k$, field $\mathbf{F}$.

  For random a in $\mathbf{F}^k$, compute $a \cdot x$.
- <u>More examples</u>:

This talk: $\mathbf{F}$ is a field with constant time operations

Linear hash functions

- Associate key x with a vector v_x in $\mathbf{F}^k$, field $\mathbf{F}$.

  For random a in $\mathbf{F}^k$, compute $a \cdot x$.
- <u>More examples</u>:
  - x in $\mathbf{F}$, $v_x = (x^0, x^1, x^2, \ldots, x^{k-1})$.
    Polynomial hash functions.

This talk: $\mathbf{F}$ is a field with constant time operations

Linear hash functions

- Associate key x with a vector v_x in $\mathbf{F}^k$, field $\mathbf{F}$.

  For random a in $\mathbf{F}^k$, compute $a \cdot x$.
- <u>More examples</u>:
  - x in $\mathbf{F}$, $v_x = (x^0, x^1, x^2, \ldots, x^{k-1})$.
    Polynomial hash functions.
  - $x = x_1 \ldots x_n$, $v_x =$ indicator vector of $\{(i, x_i), i = 1, .., n\}$.
    Tabulation hashing.

This talk: $\mathbf{F}$ is a field with constant time operations

Linear hash functions

- Associate key x with a vector v_x in $\mathbf{F}^k$, field $\mathbf{F}$.

  For random a in $\mathbf{F}^k$, compute $a \cdot x$.

- <u>More examples</u>:
  - x in $\mathbf{F}$, $v_x = (x^0, x^1, x^2, \ldots, x^{k-1})$.
    Polynomial hash functions.
  - $x = x_1 \ldots x_n$, $v_x =$ indicator vector of $\{(i, x_i), i = 1, .., n\}$.
    Tabulation hashing.
  - $v_x =$ indicator vector of $\Gamma(x)$, Γ neighbor function of a unique neighbor expander. *"Siegel hashing."*

This talk: $\mathbf{F}$ is a field with constant time operations

Linear hash functions

- Associate key x with a vector v_x in $\mathbf{F}^k$, field $\mathbf{F}$.

  For random a in $\mathbf{F}^k$, compute $a \cdot x$.

- <u>More examples</u>:
  - x in F, $v_x = (x^0, x^1, x^2, \ldots, x^{k-1})$.
  - $\ldots i), i=1,..,n\}$.
  - $v_x =$ indicator vector of $\Gamma(x)$, Γ neighbor function of a unique neighbor expander. *"Siegel hashing."*

This talk: $\mathbf{F}$ is a field with constant time operations

Property: If every set of k vectors v_x is linearly independent, the hash function is k-independent

Two famous, theoretically good linear hash functions

- **Universal hashing** [Carter & Wegman '77]
  Small *expected* load for bucket of every item i.
  Aka. pairwise independence. Provably works for fingerprinting, hash tables with chaining, Bloom filters,…

- u^{ε}**-independence** [Siegel et al. '89, '93]:
  Bucket load obeys Chernoff concentration bounds.
  Provably works for load balancing, cuckoo hashing, linear probing, double hashing, …

- Both have constant evaluation time, but seem to be little used in practice. **Why?**

On Randomness in Hash Functions

Martin Dietzfelbinger
Technische Universität Ilmenau

On Randomness in Hash Functions

Martin Dietzfelbinger
Technische Universität Ilmenau

RAM model of computation

Issues typically neglected in theory of hashing

Things that go beyond the word RAM model:

- Memory hierarchy effects
- Contemporary instruction sets
- Pipelining and small-scale parallelism

Other problems in practice

- **Universal hashing**:

Items are strings in Σ^n

  - Requires seed length proportional to $\log n$.
  - Computationally efficient implementations tend to use $\mathrm{O}(n \log n)$ seed length.

Other problems in practice

- **Universal hashing**:

  Items are strings in Σ^n

  - Requires seed length proportional to $\log n$.
  - Computationally efficient implementations tend to use $O(n \log n)$ seed length.
- u^ε**-independence**:
  - Siegel's method requires a very large (constant) evaluation time and relatively large space.
  - (More efficient solutions would be possible given access to efficient *unbalanced expanders*.)

Prevalent in practice: Deterministic hashing

Prevalent in practice: Deterministic hashing

- Java string hashing (signed 32-bit arithmetic):
  $h(a_1 a_2 \ldots a_n) = a_n + 31\, h(a_1 a_2 \ldots a_{n-1})$

Prevalent in practice: Deterministic hashing

- Java string hashing (signed 32-bit arithmetic):
  $h(a_1 a_2 \ldots a_n) = a_n + 31\, h(a_1 a_2 \ldots a_{n-1})$

- Collisions:
  - $h(Aa) = h(BB) = 2112$ (equivalent substrings)
  - $h(AaAa) = h(AaBB) = h(BBAa) = h(BBBB) = 2095104$
  - …

Prevalent in practice: Deterministic hashing

- Java string hashing (signed 32-bit arithmetic):
  $h(a_1 a_2 \ldots a_n) = a_n + 31\, h(a_1 a_2 \ldots a_{n-1})$

- Collisions:
  - $h(Aa) = h(BB) = 2112$ (equivalent substrings)
  - $h(AaAa) = h(AaBB) = h(BBAa) = h(BBBB) = 2095104$
  - …

- Recent heuristic hash functions, with focus on evaluation time: MurmurHash, CityHash, SipHash.

But (some) people are starting to care!

But (some) people are starting to care!

- Crosby & Wallach: *Denial of Service via Algorithmic Complexity Attacks.* Usenix Security ’03.
  - Follow-ups: Chaos Communication Congress ’11, ’12.
  - Meet-in-the-middle (birthday) attacks find collisions among b-bit hash values in time and space $O(2^{b/2})$.

But (some) people are starting to care!

- Crosby & Wallach: *Denial of Service via Algorithmic Complexity Attacks.* Usenix Security ’03.
  - Follow-ups: Chaos Communication Congress ’11, ’12.
  - Meet-in-the-middle (birthday) attacks find collisions among b-bit hash values in time and space $O(2^{b/2})$.
- Java, C++, C# libraries still use deterministic hashing.
  - But Java falls back to BST for long hash chains!

But (some) people are starting to care!

- Crosby & Wallach: *Denial of Service via Algorithmic Complexity Attacks.* Usenix Security ’03.
  - Follow-ups: Chaos Communication Congress ’11, ’12.
  - Meet-in-the-middle (birthday) attacks find collisions among b-bit hash values in time and space $O(2^{b/2})$.
- Java, C++, C# libraries still use deterministic hashing.
  - But Java falls back to BST for long hash chains!
- **NEW**: Ruby 1.9, Python 3.3, [Perl 5.18] now use *random hashing* [if deterministic hashing fails].

Agenda

- Basic definitions and motivation.
- Great theoretical results - useless in practice?
- <u>New developments</u>:
  - Simple tabulation hashing.
  - Semi-explicit unbalanced expanders.
  - Pseudorandom generation.
- Open questions.

Simple tabulation hashing

- Let $h_1,\ldots,h_n: \Sigma \rightarrow D$ be fully random functions (implemented as lookup tables).
- Simple tabulation [Zobrist '70]:
  - $h(x_1 \ldots x_n) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$.

Simple tabulation hashing

- Let $h_1,\ldots,h_n: \Sigma \rightarrow D$ be fully random functions (implemented as lookup tables).
- Simple tabulation [Zobrist '70]:
  - $h(x_1 \ldots x_n) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$. $\oplus$ is bitwise XOR

Simple tabulation hashing

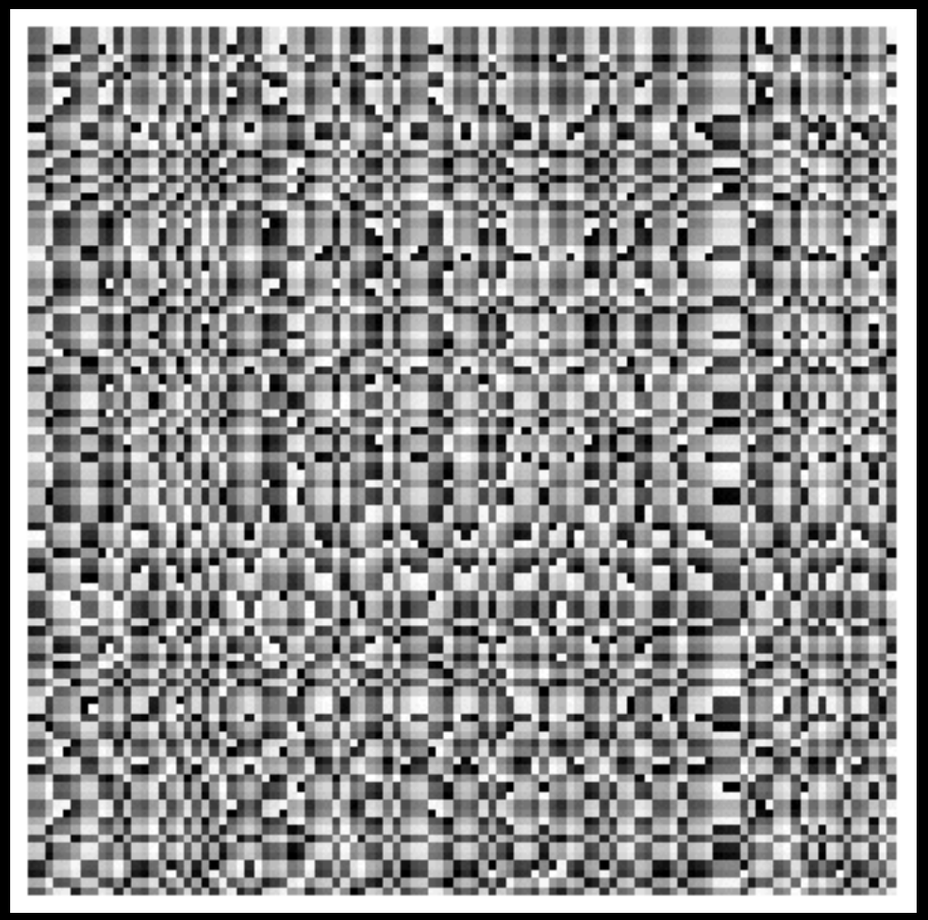
- Let $h_1,\ldots,h_n: \Sigma \rightarrow D$ be fully random functions (implemented as lookup tables).
- Simple tabulation [Zobrist '70]:
  - $h(x_1 \ldots x_n) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$. $\oplus$ is bitwise XOR
  - Only *3-independent.*

Simple tabulation hashing

- Let $h_1,\ldots,h_n: \Sigma \rightarrow D$ be fully random functions (implemented as lookup tables).
- Simple tabulation [Zobrist '70]:
  - $h(x_1 \ldots x_n) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$. $\oplus$ is bitwise XOR
  - Only *3-independent.*
  - [Pătraşcu-Thorup, J. ACM '12]: Yields Chernoff bounds for constant n (space $n\Sigma$).

The power of simple tabulation hashing

- Gives many properties of full randomness
  - But not all, e.g., dynamic cuckoo hashing is slow for some data.



a_2

a_1

Double tabulation

Mikkel Thorup on TV

Double tabulation

Double tabulation

[Thorup, FOCS '13]

- Let $h_1,\ldots,h_n: \Sigma \to \Sigma^6$, $g_1,\ldots,g_{6n}: \Sigma \to D$ be fully random functions (impl. as lookup tables).

- Double tabulation:
  - $(y_1 \ldots y_{6n}) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$.
  - $h(x_1 \ldots x_n) = g_1(y_1) \oplus \ldots \oplus g_{6n}(y_{6n})$.

Double tabulation

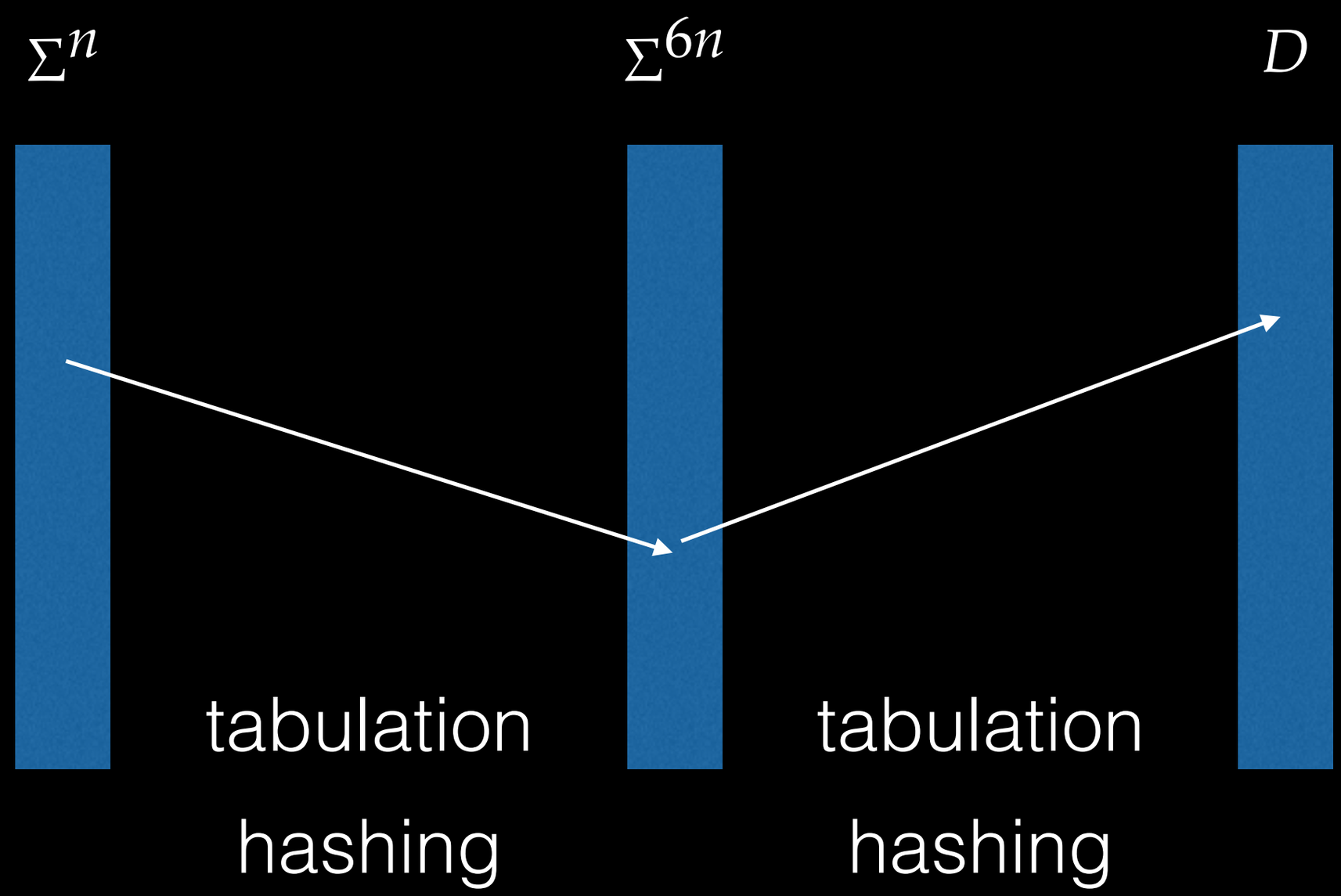
[Thorup, FOCS '13]

- Let $h_1,\ldots,h_n: \Sigma \to \Sigma^6$, $g_1,\ldots,g_{6n}: \Sigma \to D$ be fully random functions (impl. as lookup tables).

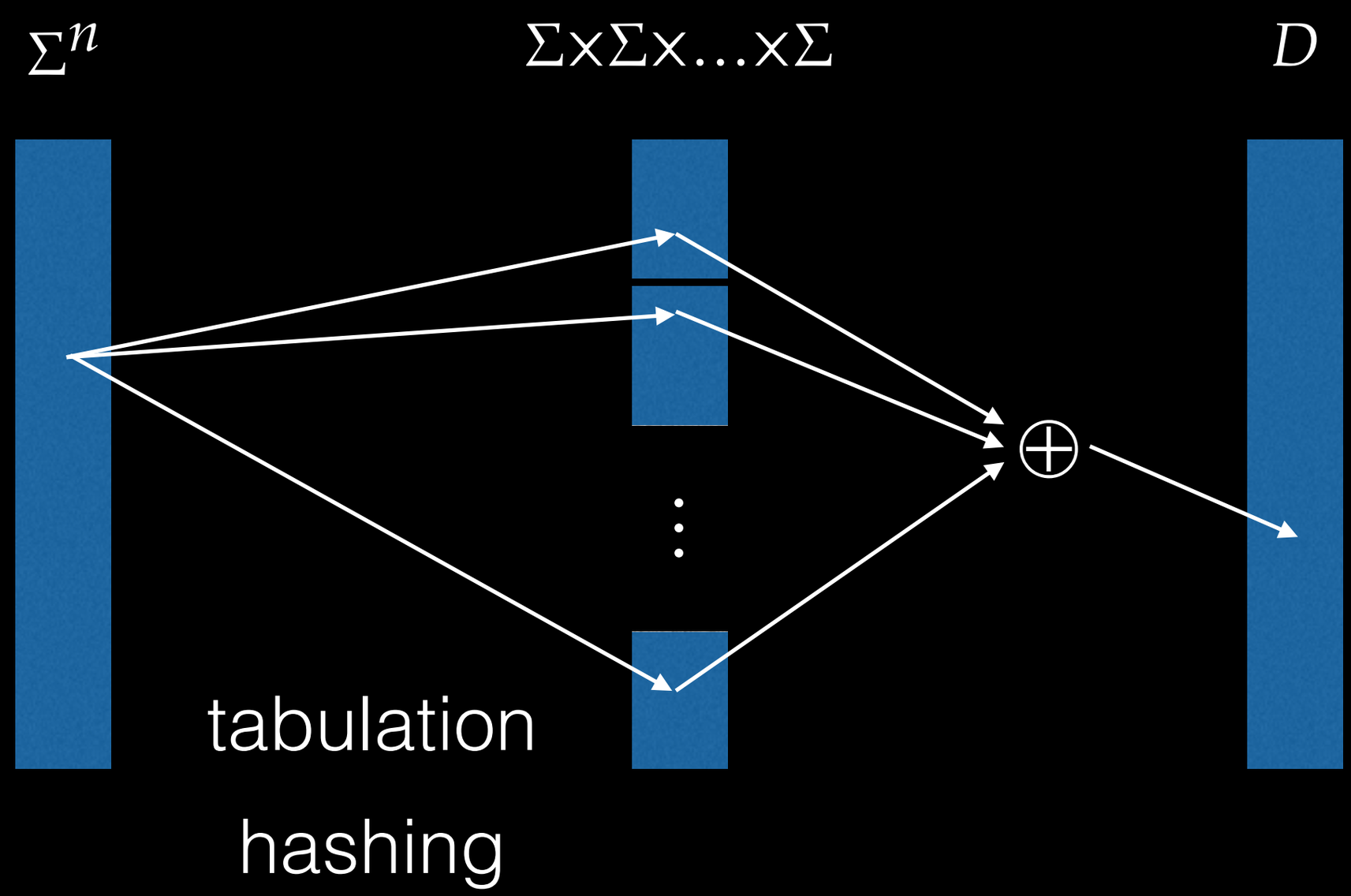
- Double tabulation:
  - $(y_1 \ldots y_{6n}) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$.
  - $h(x_1 \ldots x_n) = g_1(y_1) \oplus \ldots \oplus g_{6n}(y_{6n})$.

- **Main result**: Gives $k = \Sigma^{\Omega(1)}$ independence! (whp. over choice of $h_1,\ldots,h_n$)

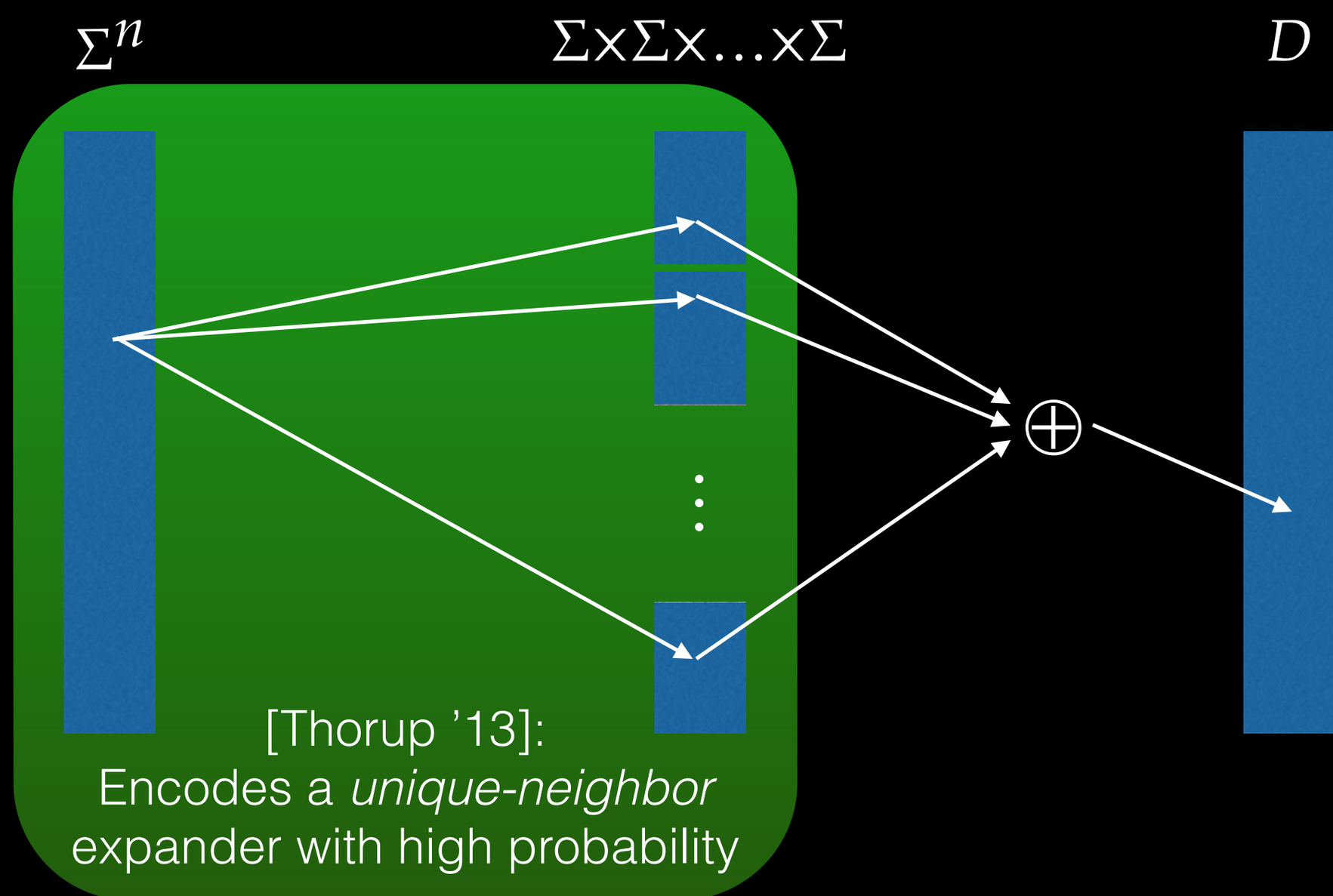
Why double tabulation gives high independence



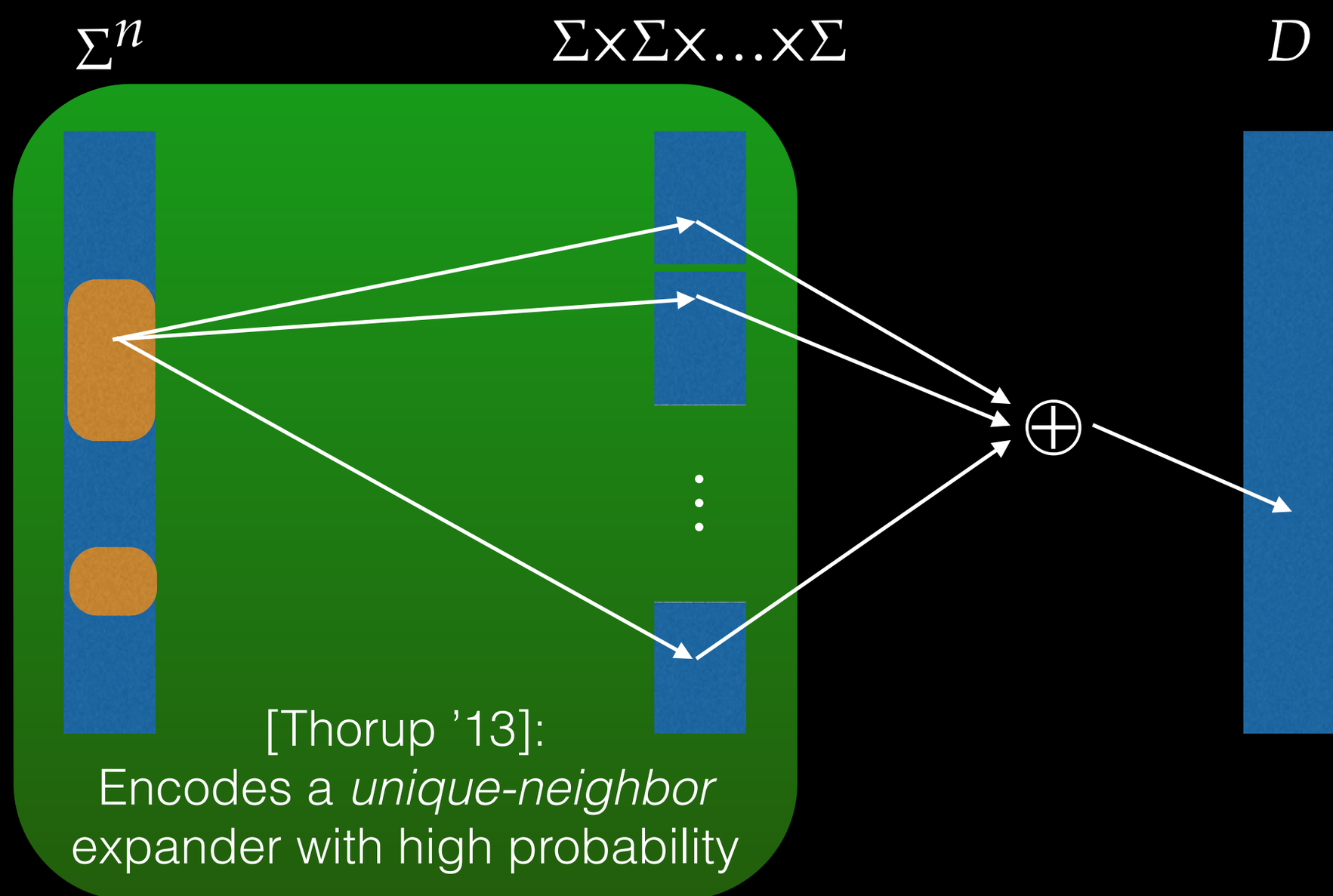
Why double tabulation gives high independence



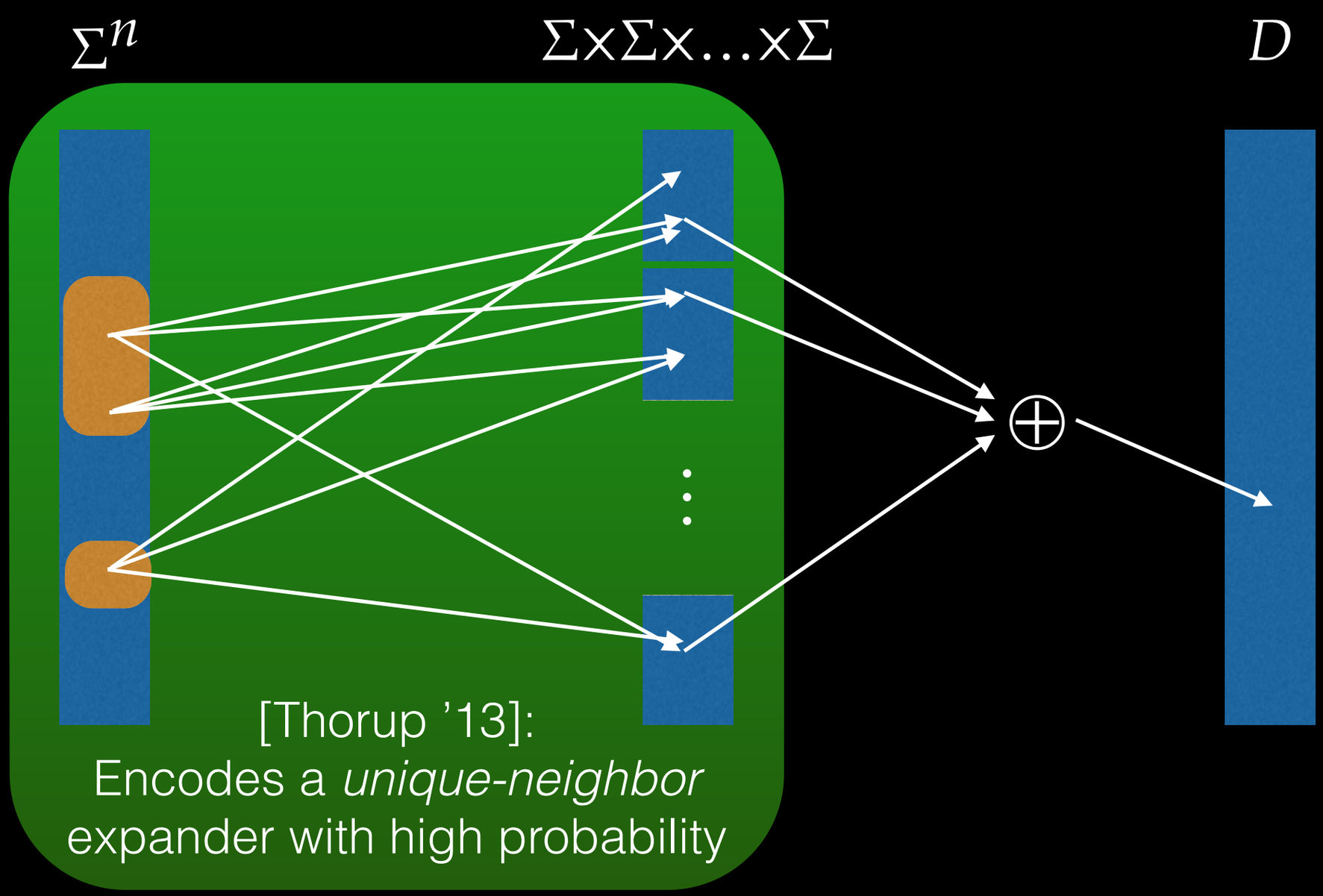
Why double tabulation gives high independence



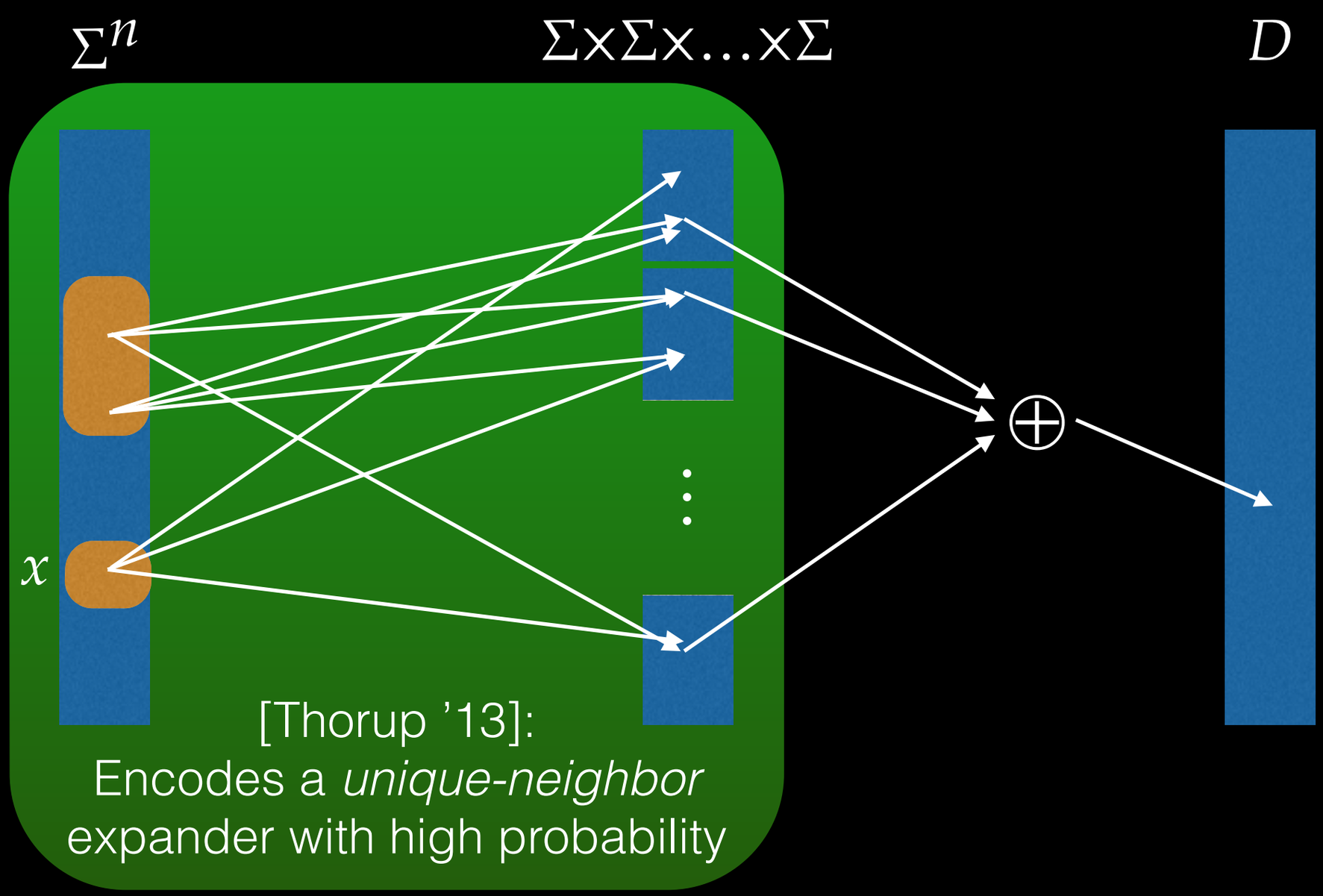
Why double tabulation gives high independence



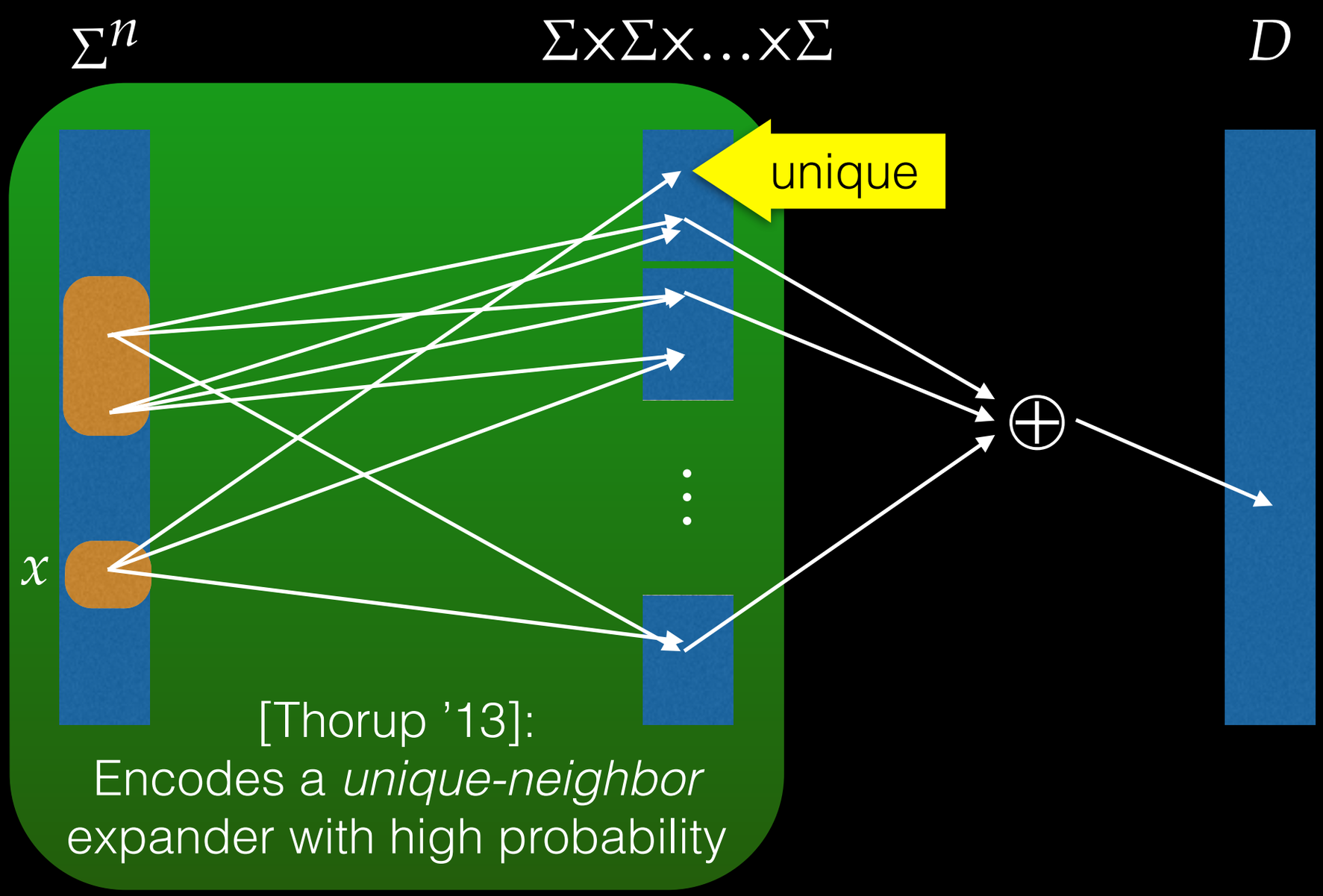
Why double tabulation gives high independence



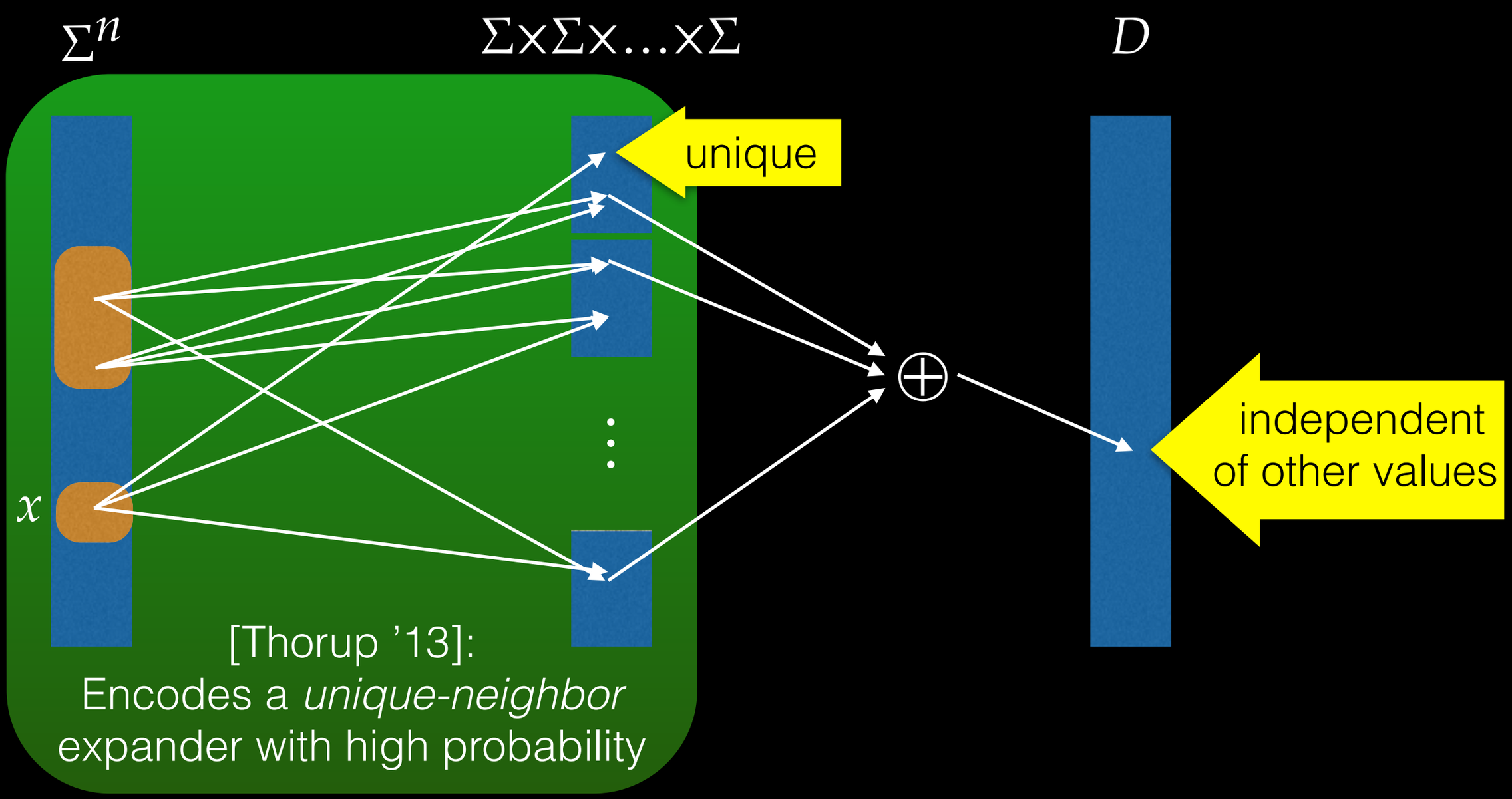
Why double tabulation gives high independence



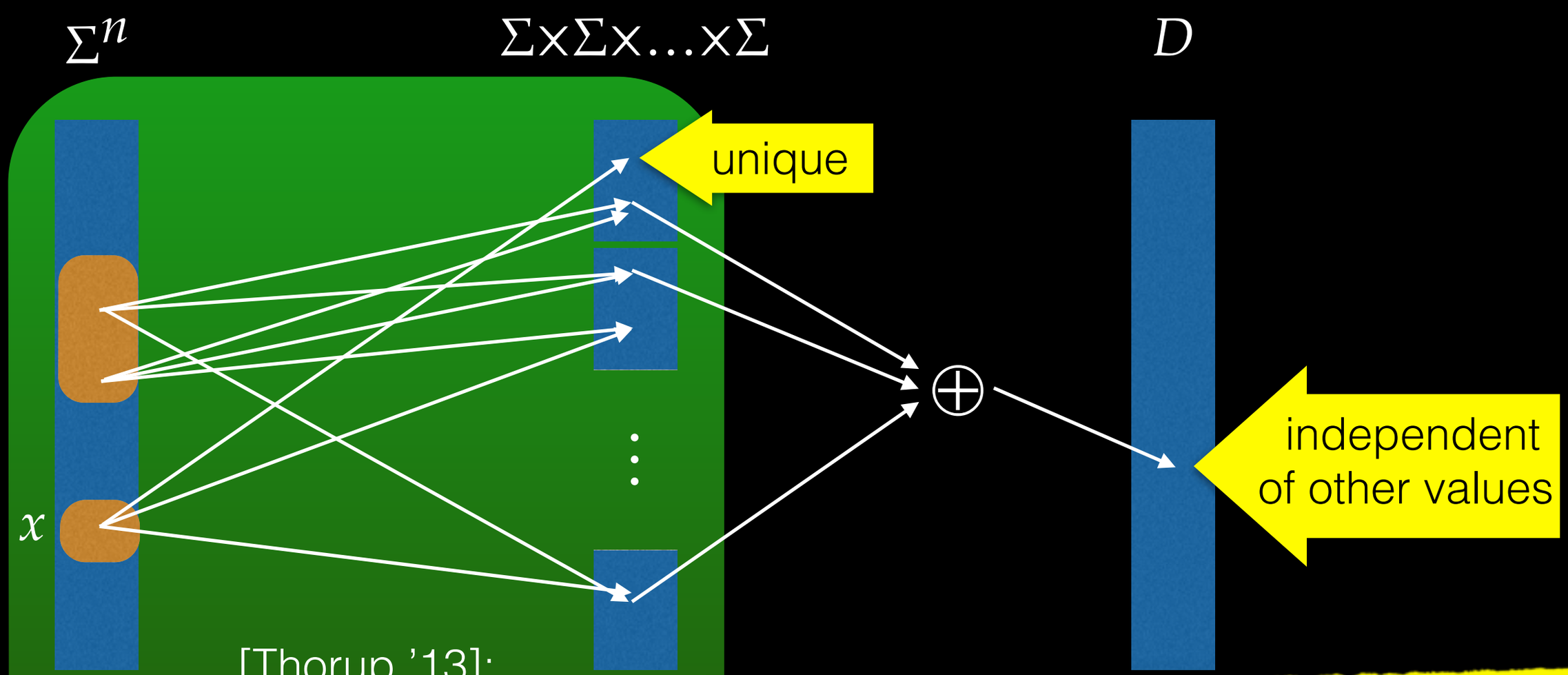
Why double tabulation gives high independence



Why double tabulation gives high independence



Why double tabulation gives high independence

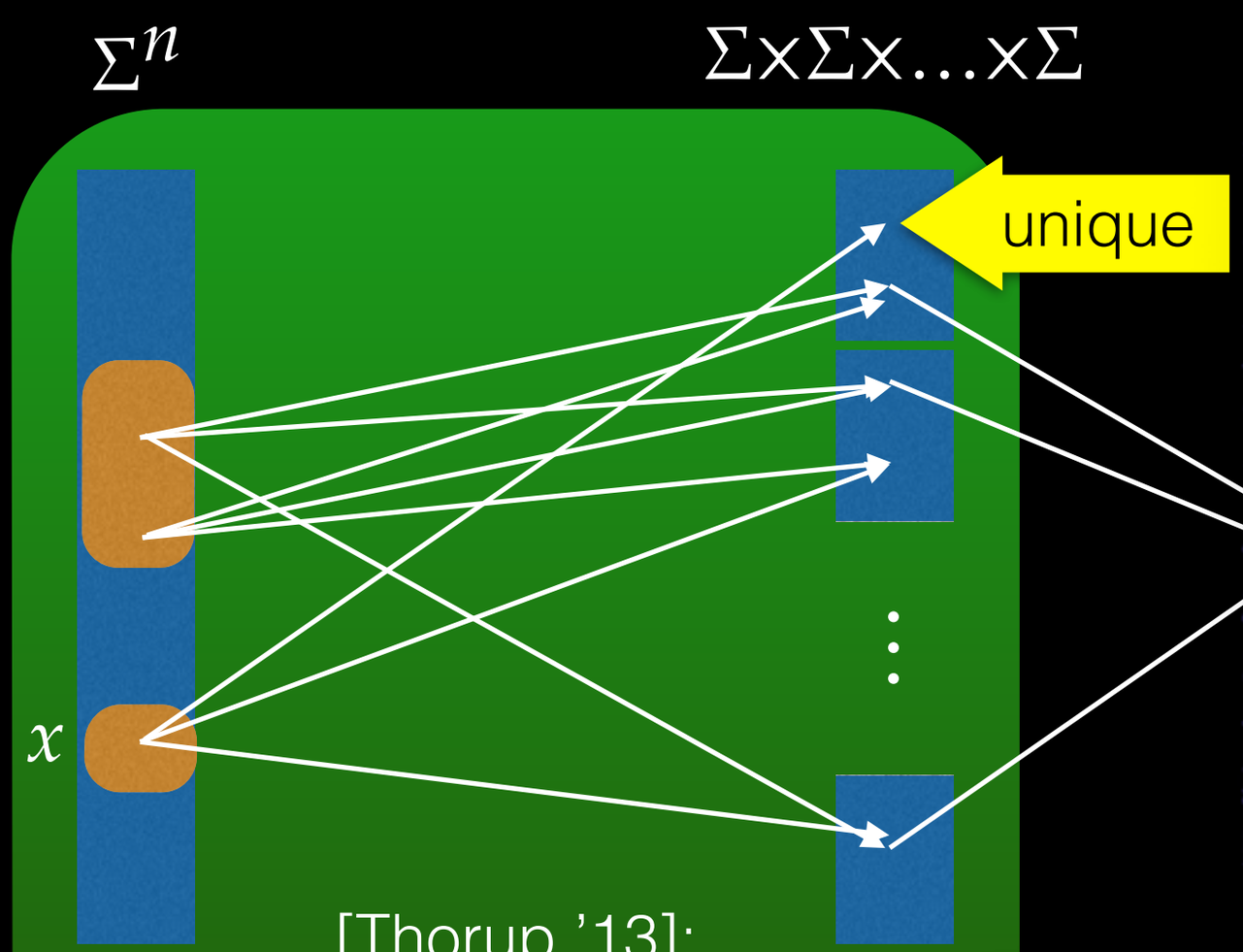


[Thorup '13]:
Encodes a *unique-neighbor* expander with high probability

$\Rightarrow$ every set of k vectors v_x lin. indep.

$\Rightarrow$ hash function is k-independent

Why double tabulation gives high independence



unique

D

"Semi-explicit" expander: Description needs many random bits, but much fewer than an adjacency list representation.

[Thorup '13]:
Encodes a *unique-neighbor* expander with high probability

$\Rightarrow$ every set of k vectors v_x lin. indep.

$\Rightarrow$ hash function is k-independent

How fast is it?

- Time-space trade-off.
- Parameters in [Thorup ’13],100-independence:
  - 32-bit keys: 22 fast lookups (in L3 cache).
  - 64-bit keys: 27 slow lookups (out of cache) **or** 52 fast lookups (in L3 cache).

How fast is it?

- Time-space trade-off.
- Parameters in [Thorup ’13],100-independence:
  - 32-bit keys: 22 fast lookups (in L3 cache).
  - 64-bit keys: 27 slow lookups (out of cache) **or** 52 fast lookups (in L3 cache).
- Typical latency:
  - L3 cache access 3 ns.
  - RAM memory access 15 ns.

How fast is it?

- Time-space trade-off.
- Parameters in [Thorup ’13],100-independence:
  - 32-bit keys: 22 fast lookups ~ 60 ns
  - 64-bit keys: 27 slow lookups ~ 400 ns **or** 52 fast lookups ~150 ns
- Typical latency:
  - L3 cache access 3 ns.
  - RAM memory access 15 ns.

Comparing to polynomial hashing

Comparing to polynomial hashing

- Polynomial k-independent hash function (assuming key x from field F):

$$p(x) = \sum_{i=0}^{k-1} a_i x^i$$

Comparing to polynomial hashing

- Polynomial k-independent hash function (assuming key x from field F):

$$p(x) = \sum_{i=0}^{k-1} a_i x^i$$

- Divide-and-conquer Horner's rule:

$$p(x) = x p_{\text{odd}}(x^2) + p_{\text{even}}(x^2)$$

Reduces data dependencies!

work by Tobias Christiani

Implementing field operations

work by Tobias Christiani

- For GF(2^{64}): Use new CMUL instruction with sparse irreducible polynomial.

Implementing field operations

work by Tobias Christiani

- For GF(2^{64}): Use new CMUL instruction with sparse irreducible polynomial.
  - Time for 100-independence ca. 300 ns

Implementing field operations

work by Tobias Christiani

- For GF(2^{64}): Use new CMUL instruction with sparse irreducible polynomial.
  - Time for 100-independence ca. 300 ns
- For GF(2^{61}-1): Use 128-bit registers and special code for modulo (Mersenne prime).

Implementing field operations

work by Tobias Christiani

- For GF(2^{64}): Use new CMUL instruction with sparse irreducible polynomial.
  - Time for 100-independence ca. 300 ns
- For GF(2^{61}-1): Use 128-bit registers and special code for modulo (Mersenne prime).
  - Time for 100-independence ca. 100 ns

Implementing field operations

work by Tobias Christiani

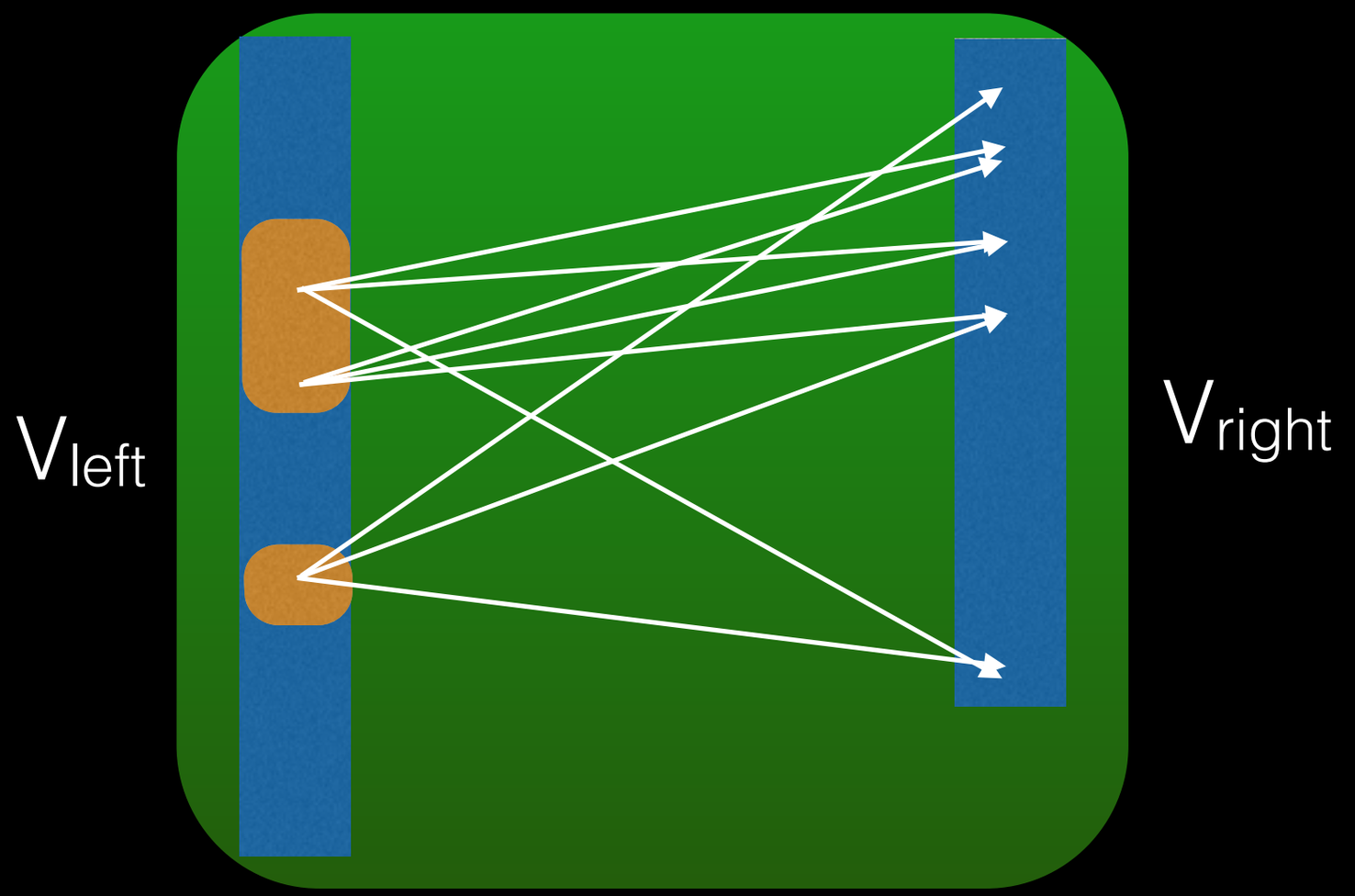
- For GF(2^{64}): Use new CMUL instruction with sparse irreducible polynomial.
  - Time for 100-independence ca. 300 ns
- For GF(2^{61}-1): Use 128-bit registers and special code for modulo (Mersenne prime).
  - Time for 100-independence ca. 100 ns

Double tabulation hashing wins for 32-bit keys, loses for 64-bit keys!

Agenda

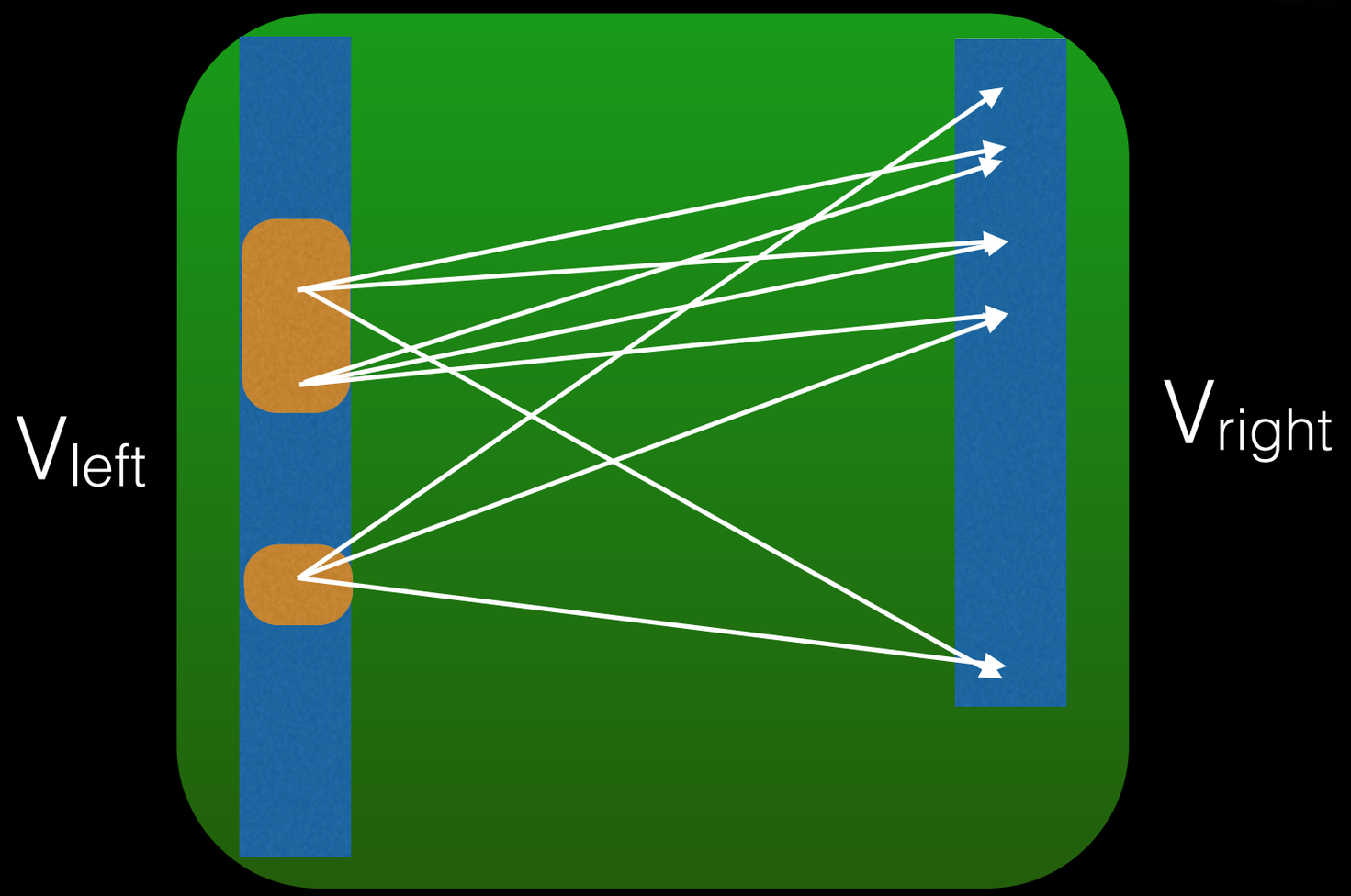
- Basic definitions and motivation.
- Great theoretical results - useless in practice?
- New developments:
  - Simple tabulation hashing.
  - Semi-explicit unbalanced expanders.
  - Pseudorandom generation.
- Open questions.

Explicit constructions of O(1)-degree expanders



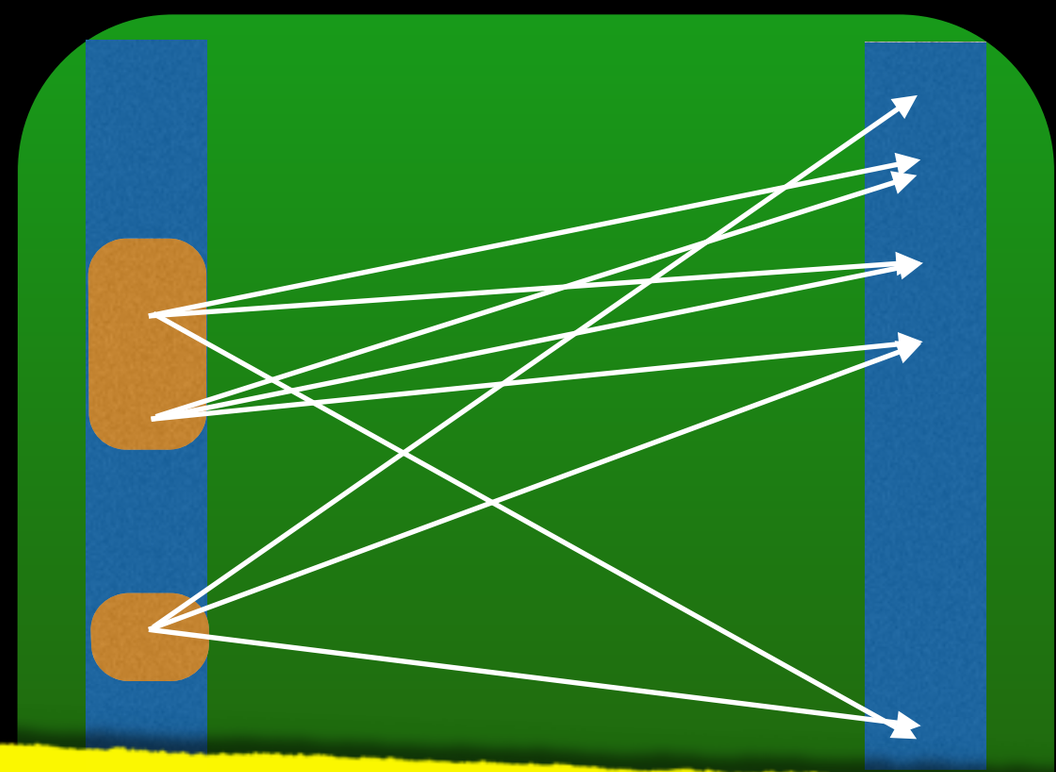
- Known to exist: Degree 3, $V_{right} < (V_{left})^{3/4}$, $k=(V_{left})^{1/4}$
- Explicitly [Capalbo '02]: Degree 3, $V_{right} < (21/22)\ V_{left}$

Can turn n numbers that are k-independent into $(22/21)n$ numbers that are $k/3$-independent



- Known to exist: Degree 3, $V_{right} < (V_{left})^{3/4}$, $k=(V_{left})^{1/4}$
- Explicitly [Capalbo '02]: Degree 3, $V_{right} < (21/22)\ V_{left}$

Can turn n numbers that are k-independent into $(22/21)n$ numbers that are $k/3$-independent

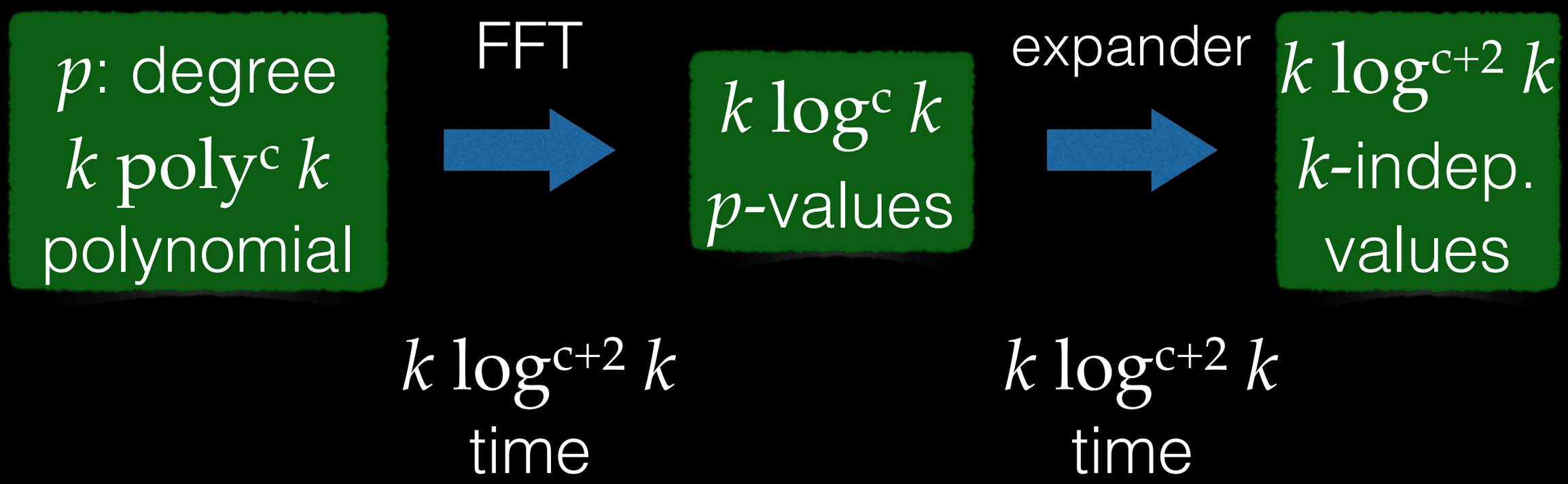


By repetition, can turn k numbers that are k-independent into k polylog k numbers that are k/polylog k-independent

- Known $V_{left})^{1/4}$
- Explic /22) V_{left}

Combining with FFT

- Using FFT it is possible to evaluate a degree-d polynomial on d inputs in $\sim d \log^2 d$ time.



New generator

- **Theorem**: Given preprocessing time and space $k \operatorname{polylog} k$ we can generate a sequence of k-independent random variables from $\mathbf{F}$ of length up to $|\mathbf{F}|$ in O(1) time/value.

joint work with Tobias Christiani, to appear at FOCS 2014

Experimental results

Experimental results

Assume that expander graph is cheaply available (e.g. read edges from SSD)

Experimental results

indep.

time/value

k	Horner ns	FFT ns	FFT + Expander c	m	d	δ	ns
2^5	177	243	64	2^{13}	8	10^{-7}	15
2^6	361	294	64	2^{14}	8	10^{-8}	16
2^7	730	338	64	2^{15}	8	10^{-9}	19
2^8	1470	375	64	2^{16}	8	10^{-10}	23
2^9	2950	412	64	2^{17}	8	10^{-11}	24
2^{10}	5902	449	64	2^{18}	8	10^{-12}	25
2^{11}	11808	487	32	2^{18}	8	10^{-12}	35
2^{12}	23627	523	64	2^{18}	16	10^{-29}	43
2^{13}	47183	561	32	2^{18}	16	10^{-29}	54
2^{14}	94429	599	64	2^{22}	8	10^{-15}	68
2^{15}	188258	638	64	2^{23}	8	10^{-16}	69

Experimental results

indep.

time/value

k	Horner ns	FFT ns	FFT + Expander c	m	d	δ	ns
2^5	177	243	64	2^{13}	8	10^{-7}	15
2^6	361	294	64	2^{14}	8	10^{-8}	16
2^7	730	338	64	2^{15}	8	10^{-9}	19
2^8	1470	375	64	2^{16}	8	10^{-10}	23
2^9	2950	412	64	2^{17}	8	10^{-11}	24
2^{10}	5902	449	64	2^{18}	8	10^{-12}	25
2^{11}	11808	487	32	2^{18}	8	10^{-12}	35
2^{12}	23627	523	64	2^{18}	16	10^{-29}	43
2^{13}	47183	561	32	2^{18}	16	10^{-29}	54
2^{14}	94429	599	64	2^{22}	8	10^{-15}	68
2^{15}	188258	638	64	2^{23}	8	10^{-16}	69

Experimental results

indep.

time/value

k	Horner ns	FFT ns	FFT + Expander c	m	d	δ	ns
2^5	177	243	64	2^{13}	8	10^{-7}	15
2^6	361	294	64	2^{14}	8	10^{-8}	16
2^7	730	338	64	2^{15}	8	10^{-9}	19
2^8	1470	375	64	2^{16}	8	10^{-10}	23
2^9	2950	412	64	2^{17}	8	10^{-11}	24
2^{10}	5902	449	64	2^{18}	8	10^{-12}	25
2^{11}	11808	487	32	2^{18}	8	10^{-12}	35
2^{12}	23627	523	64	2^{18}	16	10^{-29}	43
2^{13}	47183	561	32	2^{18}	16	10^{-29}	54
2^{14}	94429	599	64	2^{22}	8	10^{-15}	68
2^{15}	188258	638	64	2^{23}	8	10^{-16}	69

Experimental results

indep.

time/value

k	Horner ns	FFT ns	FFT + Expander c	m	d	δ	ns
2^5	177	243	64	2^{13}	8	10^{-7}	15
2^6	361	294	64	2^{14}	8	10^{-8}	16
2^7	730	338	64	2^{15}	8	10^{-9}	19
2^8	1470	375	64	2^{16}	8	10^{-10}	23
2^9	2950	412	64	2^{17}	8	10^{-11}	24
2^{10}	5902	449	64	2^{18}	8	10^{-12}	25
2^{11}	11808	487	32	2^{18}	8	10^{-12}	35
2^{12}	23627	523	64	2^{18}	16	10^{-29}	43
2^{13}	47183	561	32	2^{18}	16	10^{-29}	54
2^{14}	94429	599	64	2^{22}	8	10^{-15}	68
2^{15}	188258	638	64	2^{23}	8	10^{-16}	69

Experimental results

indep.

time/value

k	Horner ns	FFT ns	FFT + Expander c	m	d	δ	ns
2^5	177	243	64	2^{13}	8	10^{-7}	15
2^6	361	294	64	2^{14}	8	10^{-8}	16
2^7	730	338	64	2^{15}	8	10^{-9}	19
2^8	1470	375	64	2^{16}	8	10^{-10}	23
2^9	2950	412	64	2^{17}	8	10^{-11}	24
2^{10}	5902	449	64	2^{18}	8	10^{-12}	25
2^{11}	11808	487	32	2^{18}	8	10^{-12}	35
2^{12}	23627	523	64	2^{18}	16	10^{-29}	43
2^{13}	47183	561	32	2^{18}	16	10^{-29}	54
2^{14}	94429	599	64	2^{22}	8	10^{-15}	68
2^{15}	188258	638	64	2^{23}	8	10^{-16}	69

	ms	ms	c	m	a	δ	ms
2^5	177	243	64	2^{13}	8	10^{-7}	15
2^6	361	294	64	2^{14}	8	10^{-8}	16
2^7	730	338	64	2^{15}	8	10^{-9}	19
	1470	375	64	2^{16}	8	10^{-1}	
2^9	2950	412	64	2^{17}	8	10^{-11}	24
2^{10}	5902	449	64	2^{18}	8	10^{-12}	25
2^{11}	11808	487	32	2^{18}	8	10^{-12}	35
2^{12}	23627	523	64	2^{18}	16	10^{-29}	43
2^{13}	47183	561	32	2^{18}	16	10^{-29}	54
2^{14}	94429	599	64	2^{22}	8	10^{-15}	68
2^{15}	188258	638	64	2^{23}	8	10^{-16}	69
2^{16}	376143	678	64	2^{24}	8	10^{-17}	77
2^{17}	751781	719	64	2^{25}	8	10^{-18}	85
2^{18}	1505016	765	64	2^{26}	8	10^{-19}	93
2^{19}	3015969	808	32	2^{26}	8	10^{-19}	110
2^{20}	6082313	864	64	2^{26}	16	10^{-46}	175

indep.

time/value

Some open questions

- Can “semi-explicit” expanders make a practical impact in other settings where expander graphs are used?
- Is there a single method for, say, 100-independence that is superior for all key lengths?

Thanks

The following people contributed information for preparing this talk:

- Tobias Christiani, Martin Dietzfelbinger, Daniel Lemire, Ilya Mironov, Udi Wieder