Targeted Pseudorandom Generators, Simulation Advice Generators, and Derandomizing Logspace

William M. Hoza[1] Chris Umans[2]

October 10, 2016
Dagstuhl Seminar 16411

[1]University of Texas at Austin
[2]California Institute of Technology

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$AM \subseteq$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$AM \subseteq \qquad \Sigma_2$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\mathsf{AM} \subseteq \bigcap_{\varepsilon>0} \Sigma_2\mathsf{TIME}(2^{n^{\varepsilon}})$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\text{promise-AM} \subseteq \bigcap_{\varepsilon>0} \Sigma_2\mathrm{TIME}(2^{n^{\varepsilon}})$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\text{promise-AM} \subseteq \bigcap_{\varepsilon > 0} \text{i.o.-}\Sigma_2\text{TIME}(2^{n^{\varepsilon}})$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:
$$\text{promise-AM} \subseteq \bigcap_{\varepsilon>0} \text{i.o.-}\Sigma_2\text{TIME}(2^{n^{\varepsilon}})/n^{\varepsilon}$$
  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\text{promise-AM} \subseteq \bigcap_{\varepsilon > 0} \text{i.o.-}\Sigma_2\text{TIME}(2^{n^{\varepsilon}})/n^{\varepsilon}$$

  - Then there is a PRG that gives that same derandomization
- Theorem (Goldreich '11):

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\text{promise-AM} \subseteq \bigcap_{\varepsilon>0} \text{i.o.-}\Sigma_2\text{TIME}(2^{n^\varepsilon})/n^\varepsilon$$

  - Then there is a PRG that gives that same derandomization
- Theorem (Goldreich '11):
  - Assume that $\forall \Pi \in$ promise-BPP, $\forall k \in \mathbb{N}$, $\exists$ deterministic polytime algorithm A for Π s.t. any probabilistic n^k-time algorithm has only an n^{-k} chance of generating an instance on which A fails

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\text{promise-AM} \subseteq \bigcap_{\varepsilon>0} \text{i.o.-}\Sigma_2\text{TIME}(2^{n^\varepsilon})/n^\varepsilon$$

  - Then there is a PRG that gives that same derandomization
- Theorem (Goldreich '11):
  - Assume that $\forall \Pi \in$ promise-BPP, $\forall k \in \mathbb{N}$, $\exists$ deterministic polytime algorithm A for Π s.t. any probabilistic n^k-time algorithm has only an n^{-k} chance of generating an instance on which A fails
  - Then there is a PRG that gives that same derandomization

L vs. BPL

- Best PRG against logspace (Nisan '92): Seed length

$$O(\log^2 n)$$

L vs. BPL

- Best PRG against logspace (Nisan '92): Seed length

$$O(\log^2 n)$$

- Best derandomization (Saks, Zhou '98):

$$\mathrm{BPL} \subseteq \mathrm{DSPACE}(\log^{3/2} n)$$

Main result, simplified version

- **Theorem** (informally stated):

Main result, simplified version

- **Theorem** (informally stated):
  - Assume that for every derandomization of logspace, there exists a PRG strong enough to (nearly) recover derandomization

Main result, simplified version

- **Theorem** (informally stated):
  - Assume that for every derandomization of logspace, there exists a PRG strong enough to (nearly) recover derandomization
  - Then
$$\mathsf{BPL} \subseteq \bigcap_{\alpha>0} \mathsf{DSPACE}(\log^{1+\alpha} n).$$

Main result, simplified version

- **Theorem** (informally stated):
  - Assume that for every derandomization of logspace, there exists a PRG strong enough to (nearly) recover derandomization
  - Then
$$\mathsf{BPL} \subseteq \bigcap_{\alpha > 0} \mathsf{DSPACE}(\log^{1+\alpha} n).$$
- Equivalence of PRGs and derandomization would itself give a derandomization!

How to interpret our result

How to interpret our result

How to interpret our result

Outline

- ✓ Simplified statement of main result
- Proof sketch of main result
  - Saks-Zhou theorem, revisited
- Proof sketch of Saks-Zhou-Armoni theorem
- Stronger version of main result
  - Targeted PRGs
  - Simulation advice generators

When your PRG doesn't output enough bits

When your PRG doesn't output enough bits

- Given: Oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space

When your PRG doesn't output enough bits

- Given: Oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$

When your PRG doesn't output enough bits

- Given: Oracle Gen : $\{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits

When your PRG doesn't output enough bits

- Given: Oracle $\text{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$

When your PRG doesn't output enough bits

- Given: Oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$
- Approach 2: Use Gen as building block in new PRG which outputs m bits

When your PRG doesn't output enough bits

- Given: Oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$
- Approach 2: Use Gen as building block in new PRG which outputs m bits
  - E.g. using techniques of INW: Seed length

$$s + O\left((\log n) \cdot \log\left(\frac{m}{m_0}\right)\right)$$

When your PRG doesn't output enough bits

- Given: Oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$
- Approach 2: Use Gen as building block in new PRG which outputs m bits
  - E.g. using techniques of INW: Seed length

$$s + O\left((\log n) \cdot \log\left(\frac{m}{m_0}\right)\right)$$

  - For $m \gg m_0$, might as well have started from scratch!

When your PRG doesn't output enough bits

- Given: Oracle Gen : $\{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$
- Approach 2: Use Gen as building block in new PRG which outputs m bits
  - E.g. using techniques of INW: Seed length

$$s + O\left((\log n) \cdot \log\left(\frac{m}{m_0}\right)\right)$$

  - For $m \gg m_0$, might as well have started from scratch!
- Approach 3: Use Gen as building block in m-step "simulator"

Randomness-efficient simulators for automata

- Nonuniform model of $\log n$ space: n-state automaton

Randomness-efficient simulators for automata

- Nonuniform model of $\log n$ space: n-state automaton
- $Q^m(q; y) =$ final state if Q starts in state q, reads $y \in \{0,1\}^m$

Randomness-efficient simulators for automata

- Nonuniform model of $\log n$ space: n-state automaton
- $Q^m(q; y) =$ final state if Q starts in state q, reads $y \in \{0,1\}^m$
- *Simulator* for automata: algorithm $\mathrm{Sim}(Q, q, x)$ such that

$$\mathrm{Sim}(Q, q, U_s) \sim_{\varepsilon} Q^m(q; U_m)$$

PRGs for automata

- $\mathrm{Gen}(x)$ is a PRG for automata iff

$$\mathrm{Sim}(Q, q, x) = Q^m(q; \mathrm{Gen}(x))$$

  is a simulator for automata

PRGs for automata

- $\mathrm{Gen}(x)$ is a PRG for automata iff

$$\mathrm{Sim}(Q, q, x) = Q^m(q; \mathrm{Gen}(x))$$

  is a simulator for automata
- Crucial feature: Gen doesn't see "source code" (Q, q)!

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
  - Can construct m-step simulator for n-state automata with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle $\text{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
  - Can construct m-step simulator for n-state automata with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Example 1: Saks-Zhou theorem

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
  - Can construct m-step simulator for n-state automata with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Example 1: Saks-Zhou theorem
  - $m_0 = 2^{\sqrt{\log n}}$, $s = O(\log n \log m_0) = O(\log^{3/2} n)$ (INW)

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle Gen : $\{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
  - Can construct m-step simulator for n-state automata with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Example 1: Saks-Zhou theorem
  - $m_0 = 2^{\sqrt{\log n}}$, $s = O(\log n \log m_0) = O(\log^{3/2} n)$ (INW)
  - Pick $m = n$ (max # coins of ($\log n$)-space algorithm)

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
  - Can construct m-step simulator for n-state automata with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Example 1: Saks-Zhou theorem
  - $m_0 = 2^{\sqrt{\log n}}$, $s = O(\log n \log m_0) = O(\log^{3/2} n)$ (INW)
  - Pick $m = n$ (max # coins of $(\log n)$-space algorithm)
  - Obtain simulator with seed length/space complexity

$$O(\log^{3/2} n + \log^{3/2} n) = O(\log^{3/2} n)$$

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
  - Can construct m-step simulator for n-state automata with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Example 2: Some wishful thinking

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
  - Can construct m-step simulator for n-state automata with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Example 2: Some wishful thinking
  - $m_0 = 2^{\log^{0.7} n}$, $s = O(\log^{1.1} n)$ (no such construction known)

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
  - Can construct m-step simulator for n-state automata with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Example 2: Some wishful thinking
  - $m_0 = 2^{\log^{0.7} n}$, $s = O(\log^{1.1} n)$ (no such construction known)
  - Pick $m = 2^{\log^{0.8} n}$

Saks-Zhou-Armoni transformation

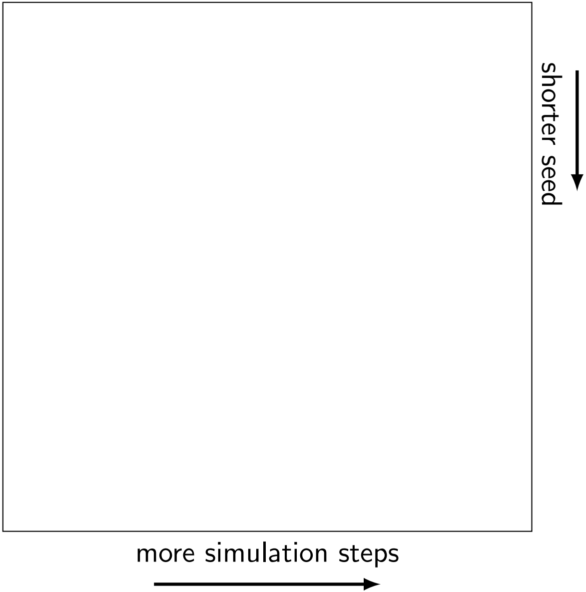
- **Theorem** (implicit in Armoni '98, builds on SZ '98, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
  - Can construct m-step simulator for n-state automata with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

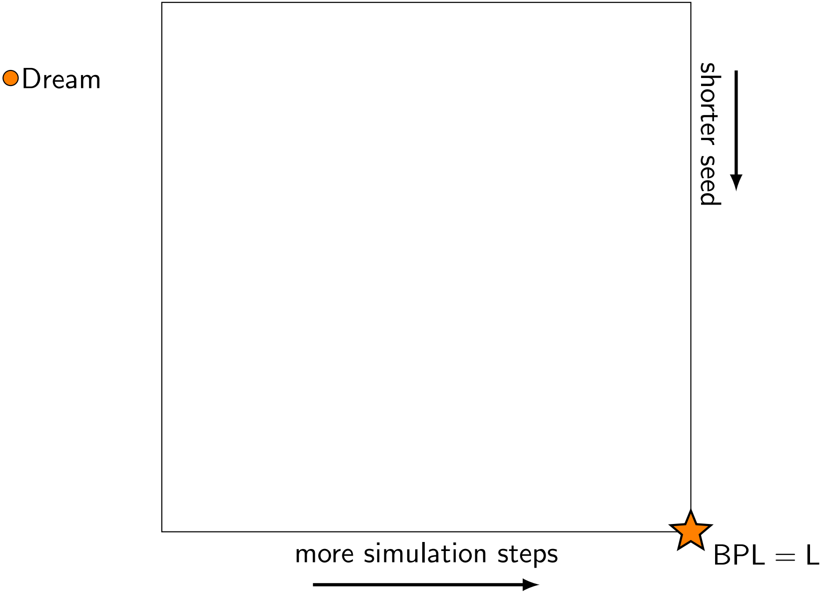
- Example 2: Some wishful thinking
  - $m_0 = 2^{\log^{0.7} n}$, $s = O(\log^{1.1} n)$ (no such construction known)
  - Pick $m = 2^{\log^{0.8} n}$
  - Obtain simulator with seed length/space complexity

$$O(\log^{1.1} n + \log^{1.1} n) = O(\log^{1.1} n)$$

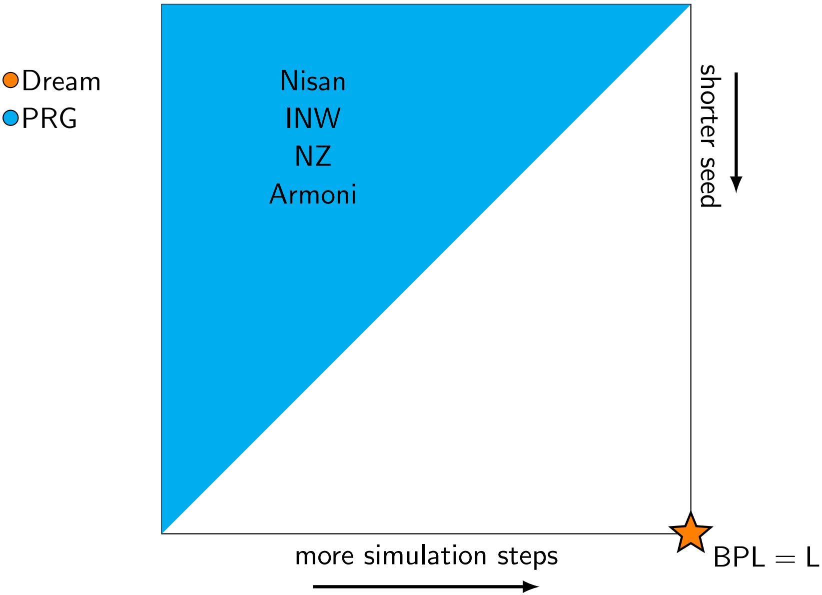
Proof of main result



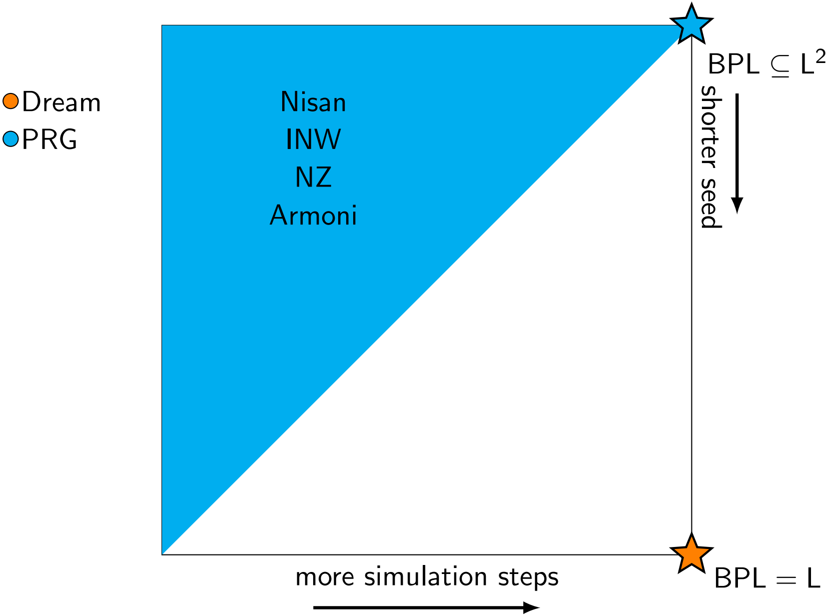
Proof of main result



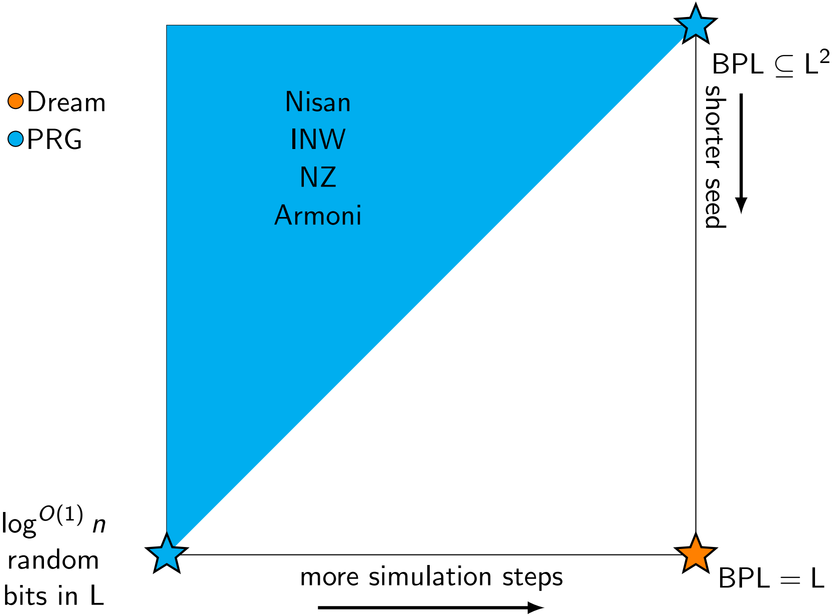
Proof of main result



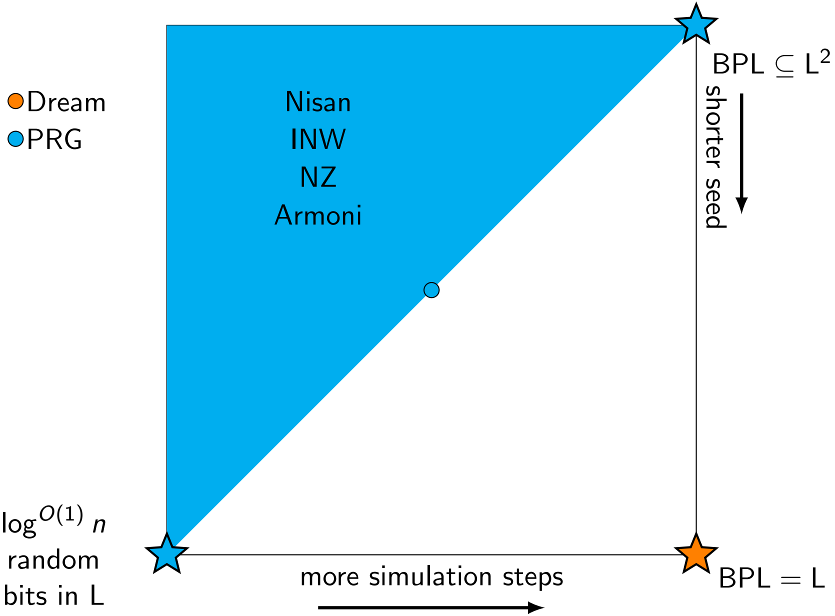
Proof of main result



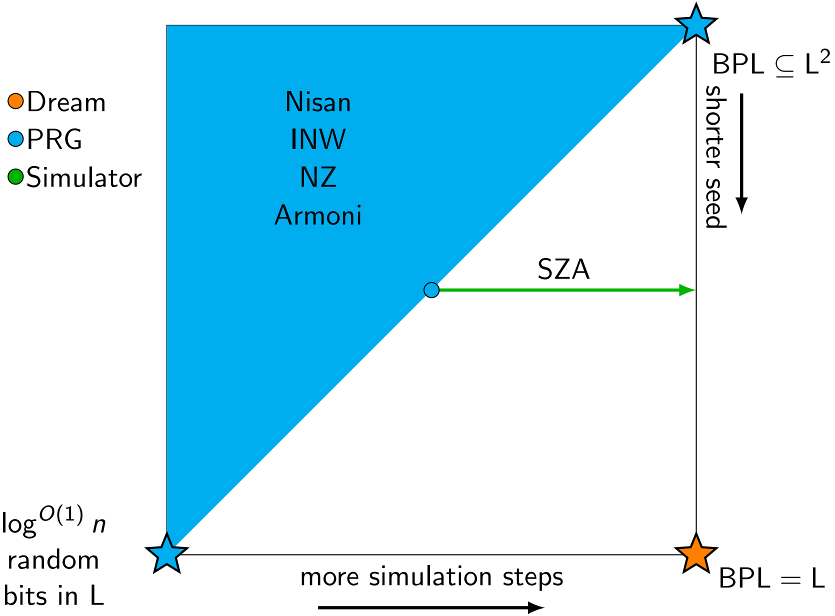
Proof of main result



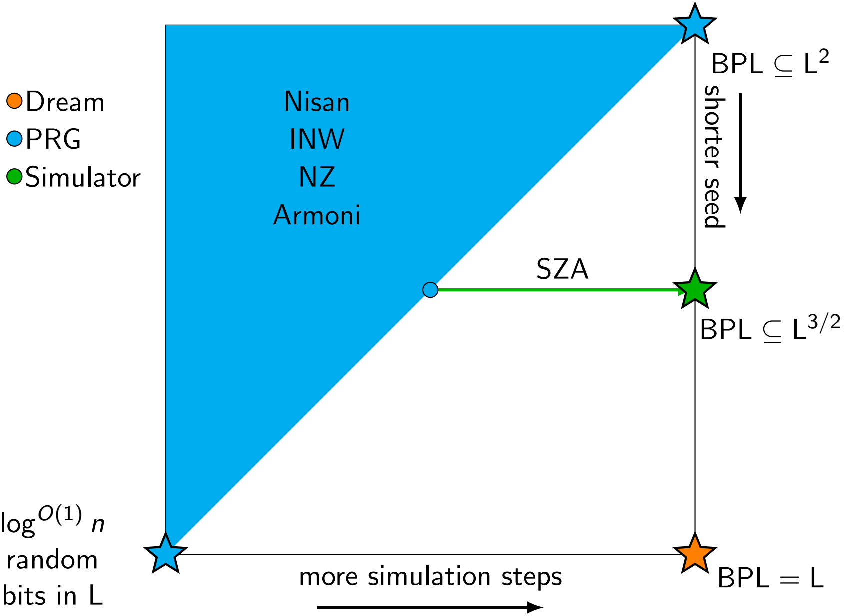
Proof of main result



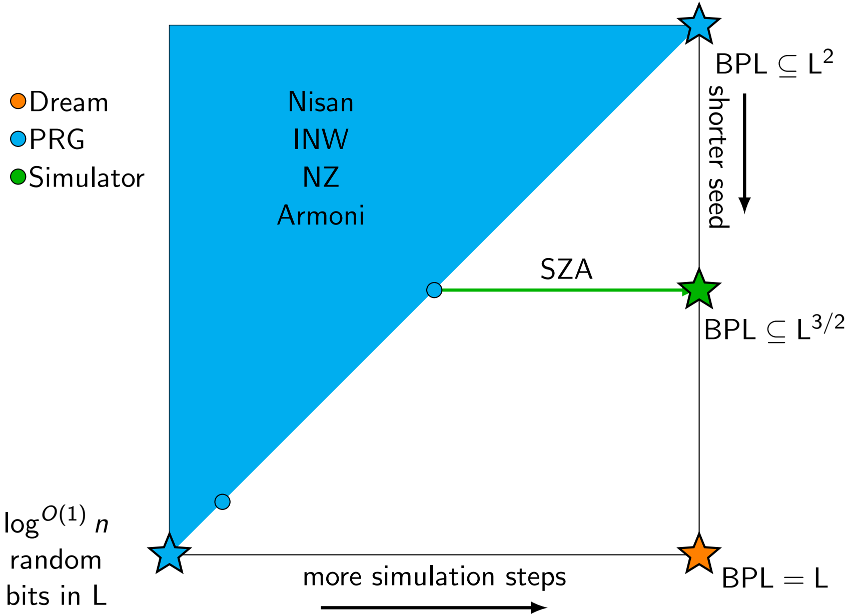
Proof of main result



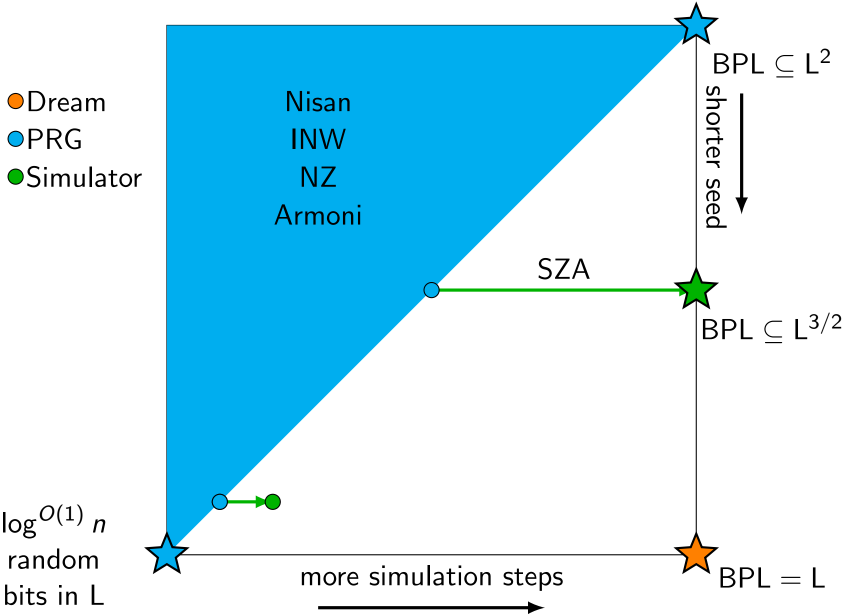
Proof of main result



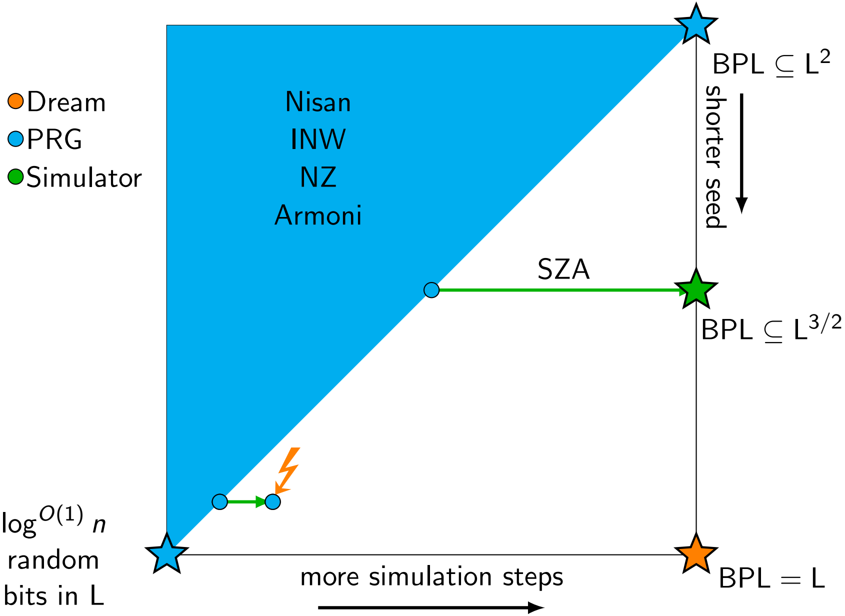
Proof of main result



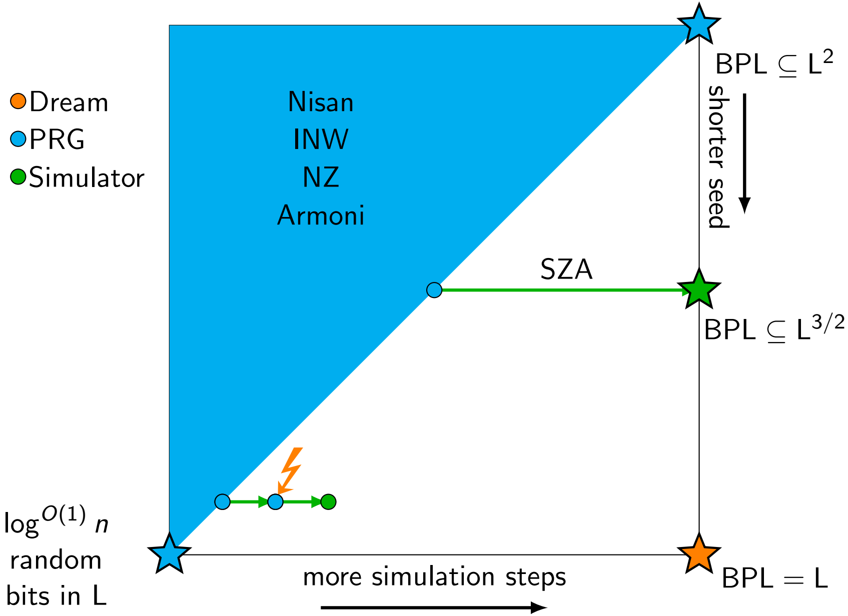
Proof of main result



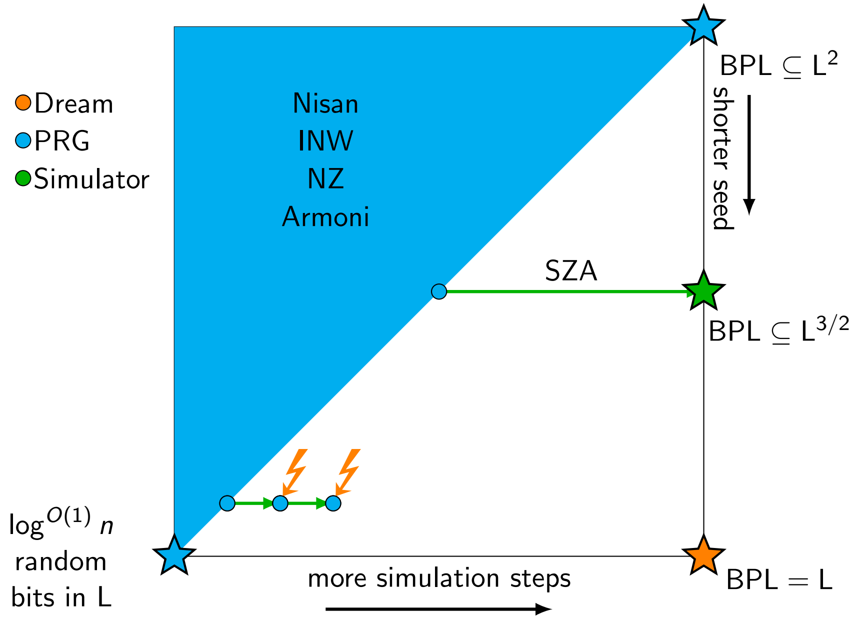
Proof of main result



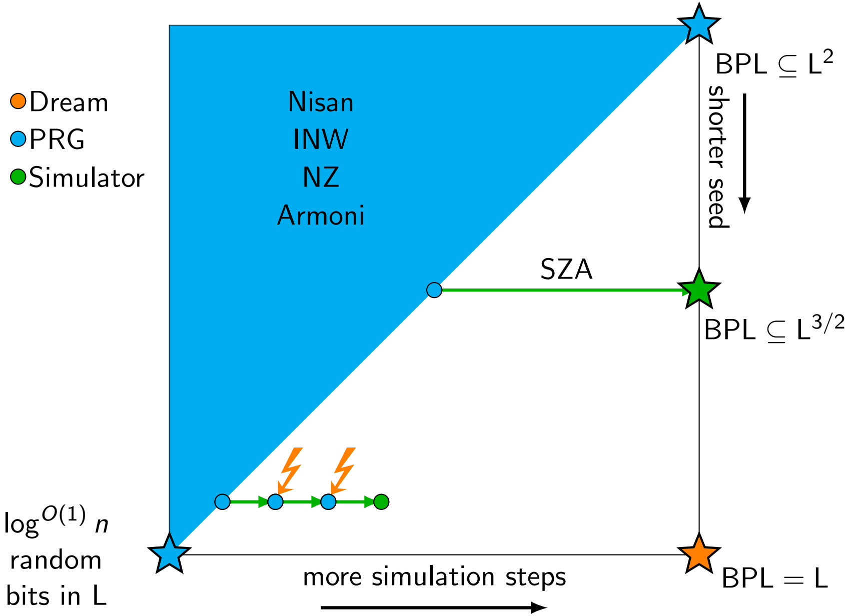
Proof of main result



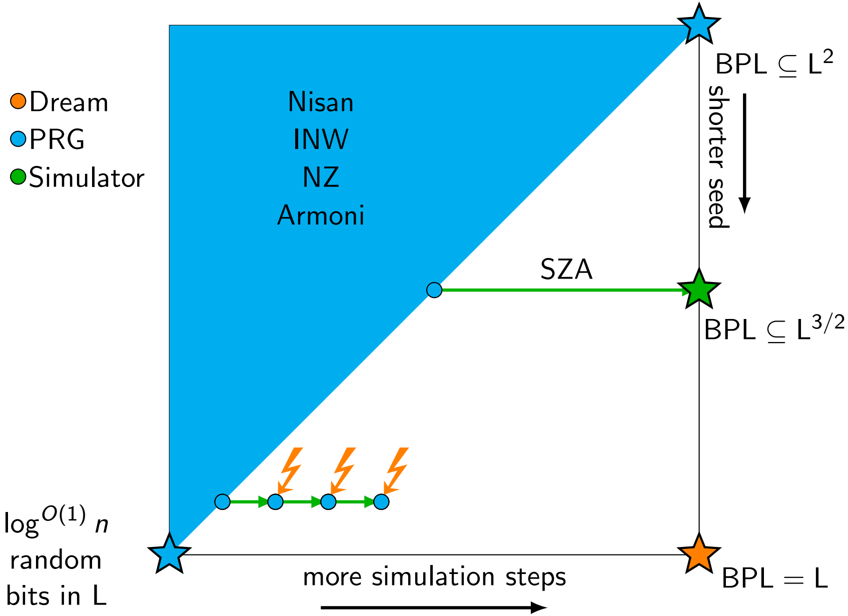
Proof of main result



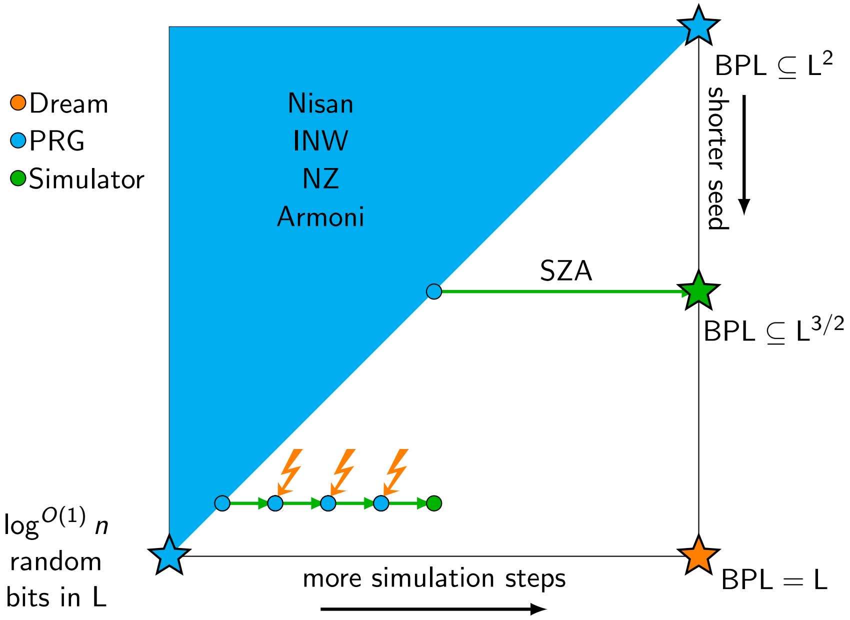
Proof of main result



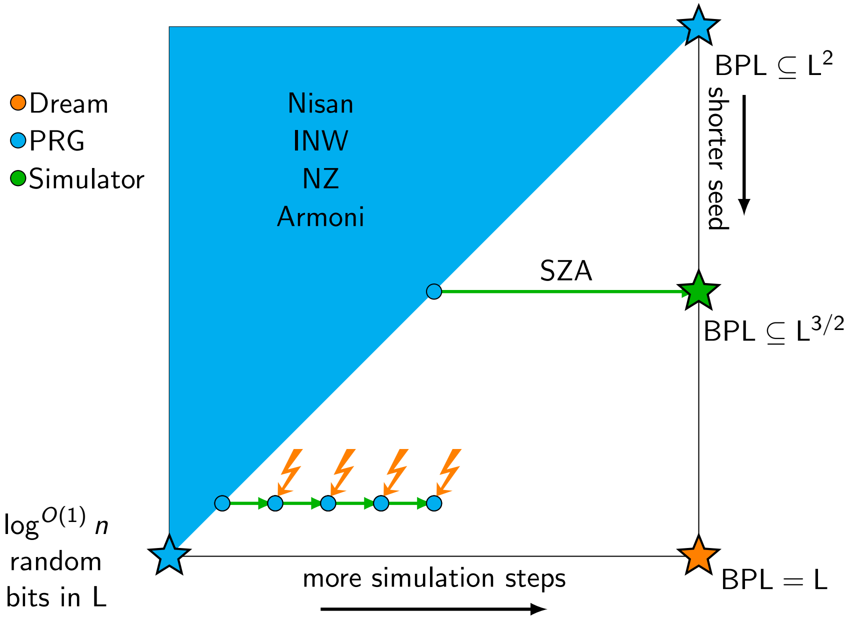
Proof of main result



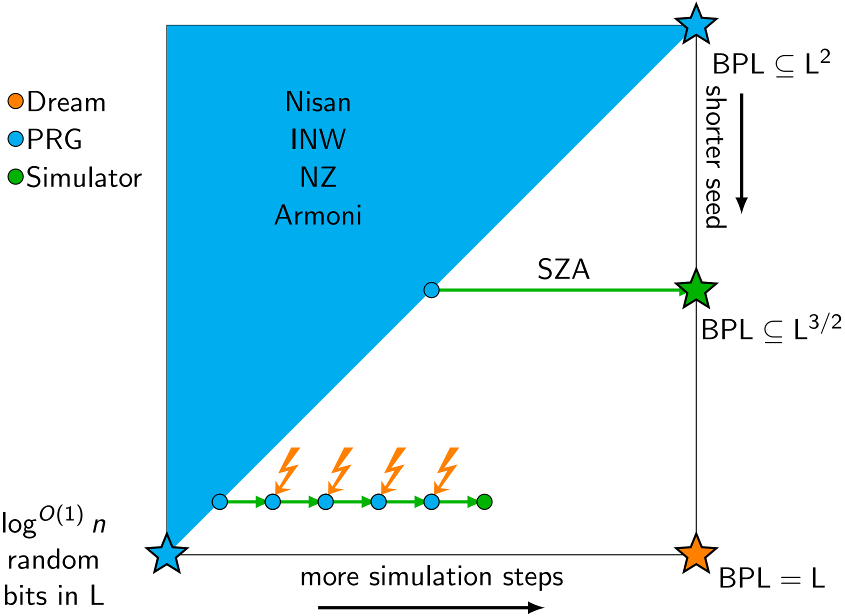
Proof of main result



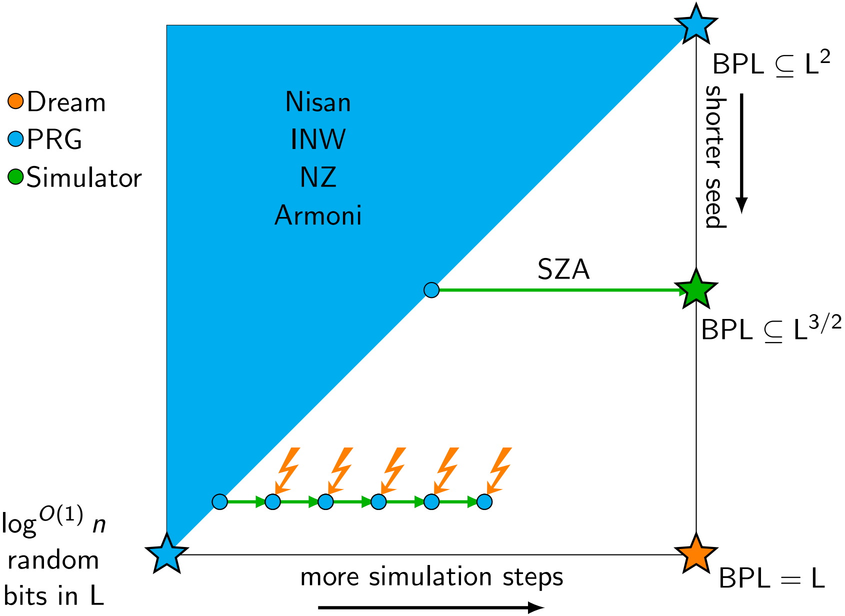
Proof of main result



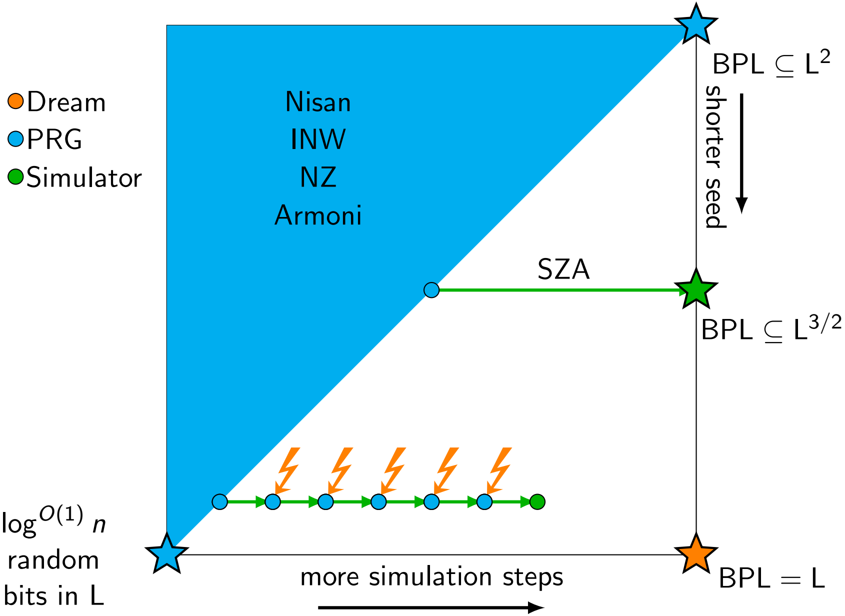
Proof of main result



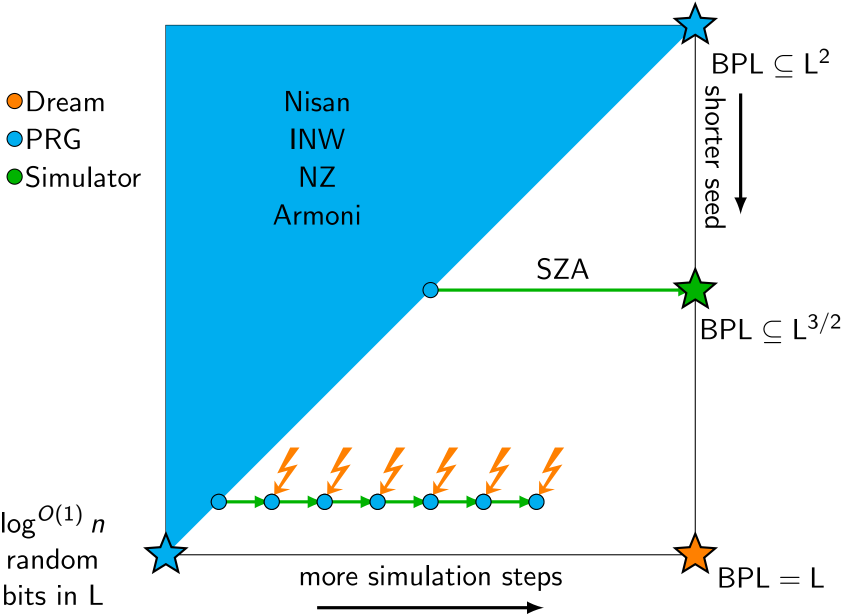
Proof of main result



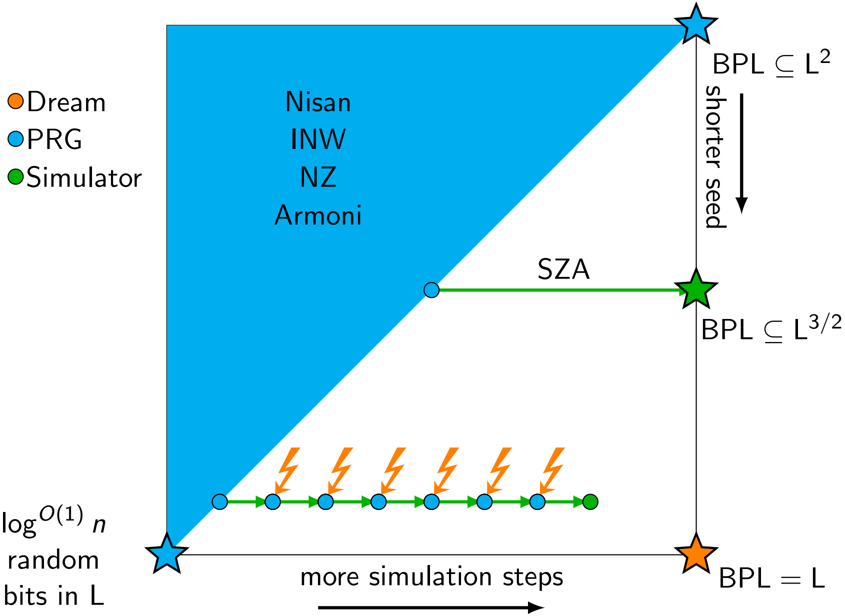
Proof of main result



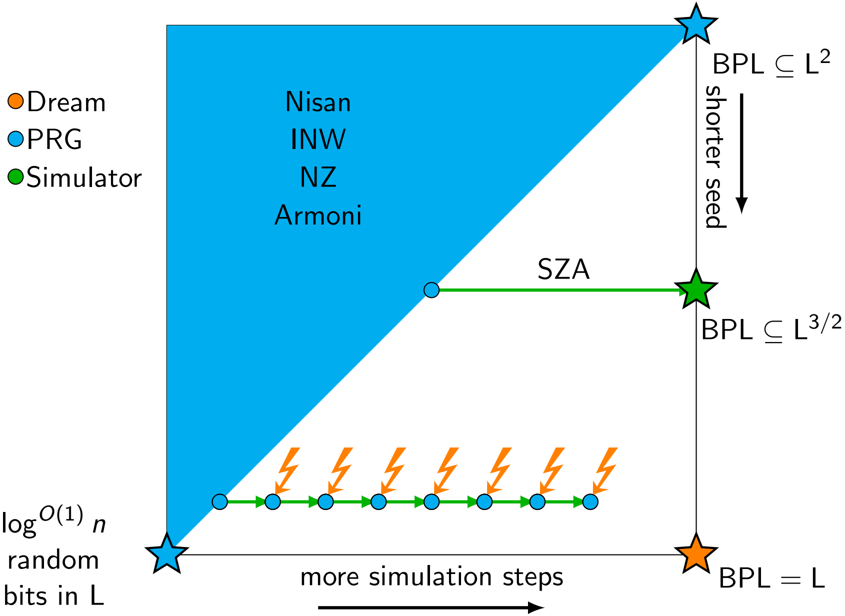
Proof of main result



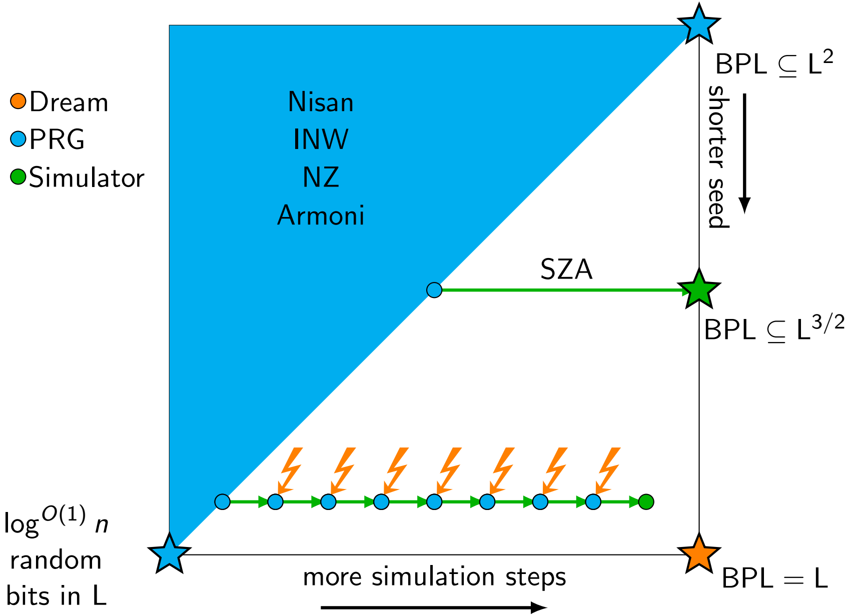
Proof of main result



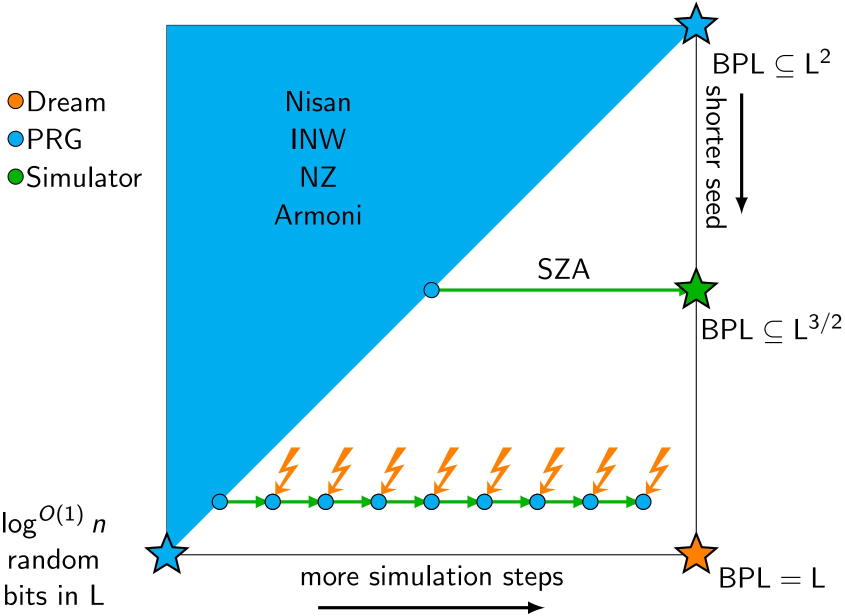
Proof of main result



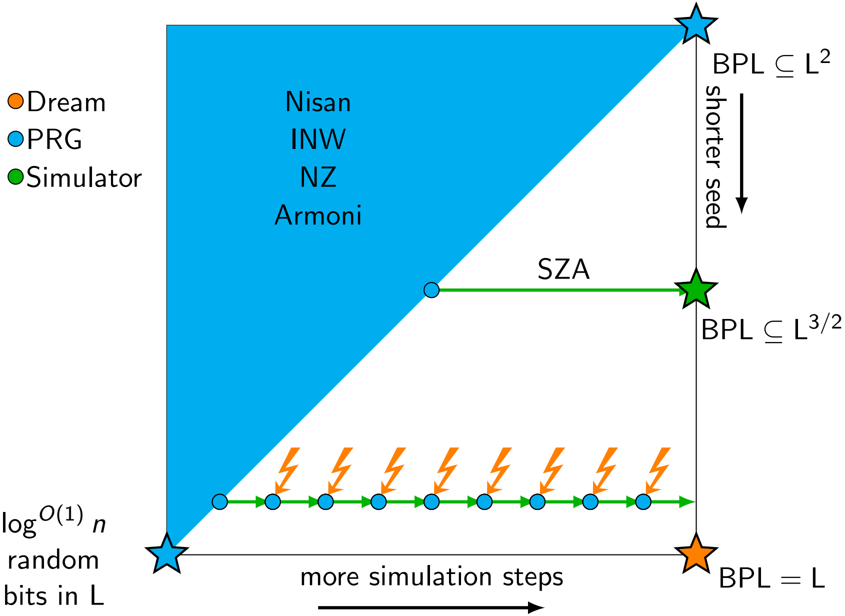
Proof of main result



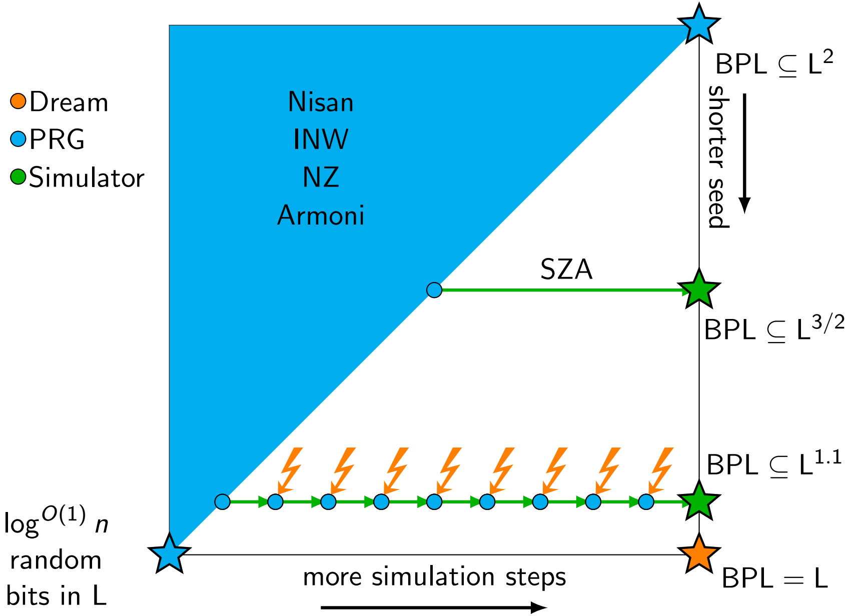
Proof of main result



Proof of main result



Proof of main result



Outline

- ✓ Simplified statement of main result
- ✓ Proof sketch of main result
  - ✓ Saks-Zhou theorem, revisited
- Proof sketch of Saks-Zhou-Armoni theorem
- Stronger version of main result
  - Targeted PRGs
  - Simulation advice generators

Randomness-efficient approximate powering

- Goal: approximate Q_0^m

Randomness-efficient approximate powering

- Goal: approximate Q_0^m
- Easier goal: Use Gen to find automaton $\mathrm{Pow}(Q_0) \approx Q_0^{m_0}$

Randomness-efficient approximate powering

- Goal: approximate Q_0^m
- Easier goal: Use Gen to find automaton $\mathrm{Pow}(Q_0) \approx Q_0^{m_0}$
- First attempt:

$$\mathrm{Pow}(Q_0)(q; y) = Q_0^{m_0}(q; \mathrm{Gen}(y))$$

Randomness-efficient approximate powering

- Goal: approximate Q_0^m
- Easier goal: Use Gen to find automaton $\mathrm{Pow}(Q_0) \approx Q_0^{m_0}$
- First attempt:

$$\mathrm{Pow}(Q_0)(q; y) = Q_0^{m_0}(q; \mathrm{Gen}(y))$$

- But we want $\mathrm{Pow}(Q_0)$ to only read $O(\log n)$ bits at a time

Randomness-efficient approximate powering

- Goal: approximate Q_0^m
- Easier goal: Use Gen to find automaton $\mathrm{Pow}(Q_0) \approx Q_0^{m_0}$
- First attempt:

$$\mathrm{Pow}(Q_0)(q; y) = Q_0^{m_0}(q; \mathrm{Gen}(y))$$

- But we want $\mathrm{Pow}(Q_0)$ to only read $O(\log n)$ bits at a time
- Randomized algorithm:

$$\mathrm{Pow}(Q_0, x)(q; y) = Q_0^{m_0}(q; \mathrm{Gen}(\mathrm{Samp}(x, y)))$$

Randomness-efficient approximate powering

- Goal: approximate Q_0^m
- Easier goal: Use Gen to find automaton $\mathsf{Pow}(Q_0) \approx Q_0^{m_0}$
- First attempt:

$$\mathsf{Pow}(Q_0)(q; y) = Q_0^{m_0}(q; \mathsf{Gen}(y))$$

- But we want $\mathsf{Pow}(Q_0)$ to only read $O(\log n)$ bits at a time
- Randomized algorithm:

$$\mathsf{Pow}(Q_0, x)(q; y) = Q_0^{m_0}(q; \mathsf{Gen}(\mathsf{Samp}(x, y)))$$

- Can achieve $|x| \leq O(s)$, $|y| \leq O(\log n)$

Repeated approximate powering

- Goal: approximate Q_0^m

Repeated approximate powering

- Goal: approximate Q_0^m
- First attempt: For $i = 1$ to $\log_{m_0} m$:

Repeated approximate powering

- Goal: approximate Q_0^m
- First attempt: For $i = 1$ to $\log_{m_0} m$:
  - Pick fresh randomness x_i

Repeated approximate powering

- Goal: approximate Q_0^m
- First attempt: For $i = 1$ to $\log_{m_0} m$:
  - Pick fresh randomness x_i
  - Let $Q_i = \mathrm{Pow}(Q_{i-1}, x_i)$

Repeated approximate powering

- Goal: approximate Q_0^m
- First attempt: For $i = 1$ to $\log_{m_0} m$:
  - Pick fresh randomness x_i
  - Let $Q_i = \mathrm{Pow}(Q_{i-1}, x_i)$
- Randomness complexity: $O(s \cdot \frac{\log m}{\log m_0})$. Too much!

Repeated approximate powering

- Goal: approximate Q_0^m
- First attempt: For $i = 1$ to $\log_{m_0} m$:
  - Pick fresh randomness x_i
  - Let $Q_i = \mathrm{Pow}(Q_{i-1}, x_i)$
- Randomness complexity: $O(s \cdot \frac{\log m}{\log m_0})$. Too much!
- Second attempt: Pick x once, reuse in each iteration

Repeated approximate powering

- Goal: approximate Q_0^m
- First attempt: For $i = 1$ to $\log_{m_0} m$:
  - Pick fresh randomness x_i
  - Let $Q_i = \mathrm{Pow}(Q_{i-1}, x_i)$
- Randomness complexity: $O(s \cdot \frac{\log m}{\log m_0})$. Too much!
- Second attempt: Pick x once, reuse in each iteration
  - Q_i is stochastically dependent on x

Repeated approximate powering

- Goal: approximate Q_0^m
- First attempt: For $i = 1$ to $\log_{m_0} m$:
  - Pick fresh randomness x_i
  - Let $Q_i = \mathrm{Pow}(Q_{i-1}, x_i)$
- Randomness complexity: $O(s \cdot \frac{\log m}{\log m_0})$. Too much!
- Second attempt: Pick x once, reuse in each iteration
  - Q_i is stochastically dependent on x
  - No guarantee that Pow will be accurate

Snap operation

- Solution: Break dependencies by rounding

Snap operation

- Solution: Break dependencies by rounding
- Snap(Q):

Snap operation

- Solution: Break dependencies by rounding
- Snap(Q):
  1. Compute $M =$ transition probability matrix of Q

Snap operation

- Solution: Break dependencies by rounding
- Snap(Q):
  1. Compute $M =$ transition probability matrix of Q
  2. Randomly perturb, round each entry of M

Snap operation

- Solution: Break dependencies by rounding
- Snap(Q):
  1. Compute $M =$ transition probability matrix of Q
  2. Randomly perturb, round each entry of M
  3. Return automaton with resulting transition probability matrix

Snap operation

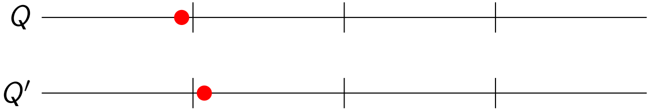
- Solution: Break dependencies by rounding
- Snap(Q):
  1. Compute $M =$ transition probability matrix of Q
  2. Randomly perturb, round each entry of M
  3. Return automaton with resulting transition probability matrix
- Key feature:

$$Q \approx Q' \implies \text{w.h.p. over } r, \mathrm{Snap}(Q, r) = \mathrm{Snap}(Q', r)$$

Snap operation

- Solution: Break dependencies by rounding
- Snap(Q):
  1. Compute $M =$ transition probability matrix of Q
  2. Randomly perturb, round each entry of M
  3. Return automaton with resulting transition probability matrix
- Key feature:

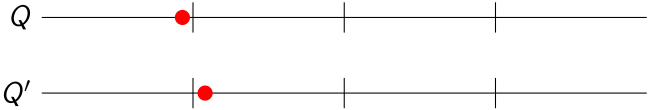
$$Q \approx Q' \implies \text{w.h.p. over } r, \mathrm{Snap}(Q, r) = \mathrm{Snap}(Q', r)$$



Snap operation

- Solution: Break dependencies by rounding
- Snap(Q):
  1. Compute $M =$ transition probability matrix of Q
  2. Randomly perturb, round each entry of M
  3. Return automaton with resulting transition probability matrix
- Key feature:

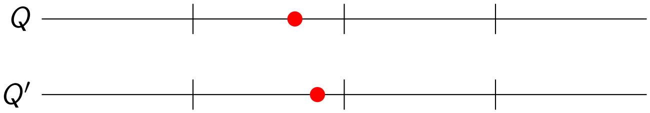
$$Q \approx Q' \implies \text{w.h.p. over } r, \text{Snap}(Q, r) = \text{Snap}(Q', r)$$



Snap operation

- Solution: Break dependencies by rounding
- Snap(Q):
  1. Compute $M =$ transition probability matrix of Q
  2. Randomly perturb, round each entry of M
  3. Return automaton with resulting transition probability matrix
- Key feature:

$$Q \approx Q' \implies \text{w.h.p. over } r, \text{Snap}(Q, r) = \text{Snap}(Q', r)$$



SZA transformation

- To approximate Q_0^m:

SZA transformation

- To approximate Q_0^m:
  1. Pick x randomly

SZA transformation

- To approximate Q_0^m:
  1. Pick x randomly
  2. For $i = 1$ to $\log_{m_0} m$, set $Q_i = \mathrm{Snap}(\mathrm{Pow}(Q_{i-1}, x))$

SZA transformation

- To approximate Q_0^m:
  1. Pick x randomly
  2. For $i = 1$ to $\log_{m_0} m$, set $Q_i = \mathrm{Snap}(\mathrm{Pow}(Q_{i-1}, x))$
- Correctness proof sketch:

SZA transformation

- To approximate Q_0^m:
  1. Pick x randomly
  2. For $i = 1$ to $\log_{m_0} m$, set $Q_i = \mathrm{Snap}(\mathrm{Pow}(Q_{i-1}, x))$
- Correctness proof sketch:
  - Define $\widehat{Q}_i$ by wishful thinking: $\widehat{Q}_0 = Q_0$, $\widehat{Q}_i = \mathrm{Snap}(\widehat{Q}_{i-1}^{m_0})$

SZA transformation

- To approximate Q_0^m:
  1. Pick x randomly
  2. For $i = 1$ to $\log_{m_0} m$, set $Q_i = \mathrm{Snap}(\mathrm{Pow}(Q_{i-1}, x))$
- Correctness proof sketch:
  - Define $\widehat{Q}_i$ by wishful thinking: $\widehat{Q}_0 = Q_0$, $\widehat{Q}_i = \mathrm{Snap}(\widehat{Q}_{i-1}^{m_0})$
  - W.h.p., for all i, $\mathrm{Pow}(\widehat{Q}_i, x) \approx \widehat{Q}_i^{m_0}$ (union bound)

SZA transformation

- To approximate Q_0^m:
  1. Pick x randomly
  2. For $i = 1$ to $\log_{m_0} m$, set $Q_i = \mathrm{Snap}(\mathrm{Pow}(Q_{i-1}, x))$
- Correctness proof sketch:
  - Define $\widehat{Q}_i$ by wishful thinking: $\widehat{Q}_0 = Q_0$, $\widehat{Q}_i = \mathrm{Snap}(\widehat{Q}_{i-1}^{m_0})$
  - W.h.p., for all i, $\mathrm{Pow}(\widehat{Q}_i, x) \approx \widehat{Q}_i^{m_0}$ (union bound)
  - W.h.p., $\mathrm{Snap}(\mathrm{Pow}(\widehat{Q}_i, x)) = \mathrm{Snap}(\widehat{Q}_i^{m_0}) = \widehat{Q}_{i+1}$

SZA transformation

- To approximate Q_0^m:
  1. Pick x randomly
  2. For $i = 1$ to $\log_{m_0} m$, set $Q_i = \mathrm{Snap}(\mathrm{Pow}(Q_{i-1}, x))$
- Correctness proof sketch:
  - Define $\widehat{Q}_i$ by wishful thinking: $\widehat{Q}_0 = Q_0$, $\widehat{Q}_i = \mathrm{Snap}(\widehat{Q}_{i-1}^{m_0})$
  - W.h.p., for all i, $\mathrm{Pow}(\widehat{Q}_i, x) \approx \widehat{Q}_i^{m_0}$ (union bound)
  - W.h.p., $\mathrm{Snap}(\mathrm{Pow}(\widehat{Q}_i, x)) = \mathrm{Snap}(\widehat{Q}_i^{m_0}) = \widehat{Q}_{i+1}$
  - W.h.p., by induction, dreams come true: $Q_i = \widehat{Q}_i$ for all i

SZA transformation

- To approximate Q_0^m:
  1. Pick x randomly
  2. For $i = 1$ to $\log_{m_0} m$, set $Q_i = \mathrm{Snap}(\mathrm{Pow}(Q_{i-1}, x))$
- Correctness proof sketch:
  - Define $\widehat{Q}_i$ by wishful thinking: $\widehat{Q}_0 = Q_0$, $\widehat{Q}_i = \mathrm{Snap}(\widehat{Q}_{i-1}^{m_0})$
  - W.h.p., for all i, $\mathrm{Pow}(\widehat{Q}_i, x) \approx \widehat{Q}_i^{m_0}$ (union bound)
  - W.h.p., $\mathrm{Snap}(\mathrm{Pow}(\widehat{Q}_i, x)) = \mathrm{Snap}(\widehat{Q}_i^{m_0}) = \widehat{Q}_{i+1}$
  - W.h.p., by induction, dreams come true: $Q_i = \widehat{Q}_i$ for all i
- Implement using recursion

Outline

- ✓ Simplified statement of main result
- ✓ Proof sketch of main result
  - ✓ Saks-Zhou theorem, revisited
- ✓ Proof sketch of Saks-Zhou-Armoni theorem
- Stronger version of main result
  - Targeted PRGs
  - Simulation advice generators

Stronger version of main result

- Two key features distinguish PRG from simulator

Stronger version of main result

- Two key features distinguish PRG from simulator
  - Input: no access to "source code" (Q, q)

Stronger version of main result

- Two key features distinguish PRG from simulator
  - Input: no access to “source code” (Q, q)
  - Output: long string for automaton to read vs. state

Stronger version of main result

- Two key features distinguish PRG from simulator
  - Input: no access to "source code" (Q, q)
  - Output: long string for automaton to read vs. state
- Claim: First feature is the one that matters for us

Targeted PRGs

- *Targeted PRG*: Algorithm $\mathsf{Gen}(Q, q, x)$ such that

$$\mathsf{Sim}(Q, q, x) = Q^m(q; \mathsf{Gen}(Q, q, x))$$

is a simulator

Targeted PRGs

- *Targeted PRG*: Algorithm $\mathsf{Gen}(Q, q, x)$ such that

$$\mathsf{Sim}(Q, q, x) = Q^m(q; \mathsf{Gen}(Q, q, x))$$

  is a simulator
- Introduced by Goldreich '11 for BPP

Targeted PRGs

- *Targeted PRG*: Algorithm $\mathsf{Gen}(Q, q, x)$ such that

$$\mathsf{Sim}(Q, q, x) = Q^m(q; \mathsf{Gen}(Q, q, x))$$

  is a simulator
- Introduced by Goldreich '11 for BPP
- Ordinary PRG: Special case that Gen doesn't depend on (Q, q)

Simulation advice generators

- *Simulation advice generator*: Algorithm $\mathsf{Gen}(x)$ such that for some deterministic logspace S,

$$\mathsf{Sim}(Q, q, x) = \mathsf{S}(Q, q, \mathsf{Gen}(x))$$

  is a simulator

Simulation advice generators

- *Simulation advice generator*: Algorithm $\mathsf{Gen}(x)$ such that for some deterministic logspace S,

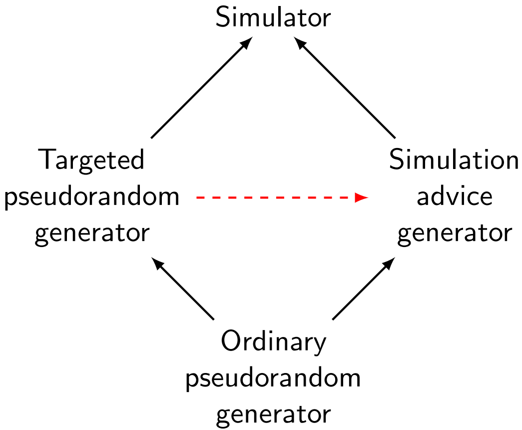
  $$\mathsf{Sim}(Q, q, x) = \mathsf{S}(Q, q, \mathsf{Gen}(x))$$

  is a simulator
- Ordinary PRG: Special case that $\mathsf{S}(Q, q, y) = Q^{|y|}(q; y)$

Four kinds of derandomization

$$
\begin{tikzcd}
 & \text{Simulator} & \\
\text{Targeted pseudorandom generator} \arrow[ur] & & \text{Simulation advice generator} \arrow[ul] \\
 & \text{Ordinary pseudorandom generator} \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

Main result (informally)



Theorem: Dashed arrow transformation exists if and only if

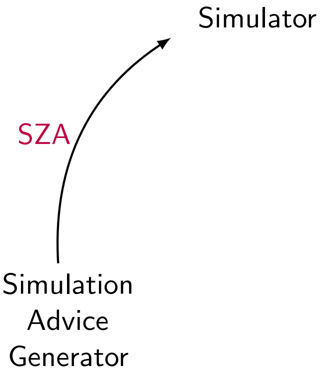
$$\bigcap_{\alpha > 0} \text{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha > 0} \text{promise-DSPACE}(\log^{1+\alpha} n)$$

Proof sketch

- SZA works on simulation advice generator without modification

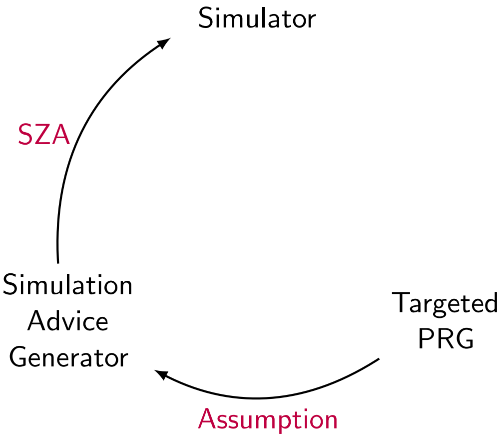
Proof sketch

- SZA works on simulation advice generator without modification



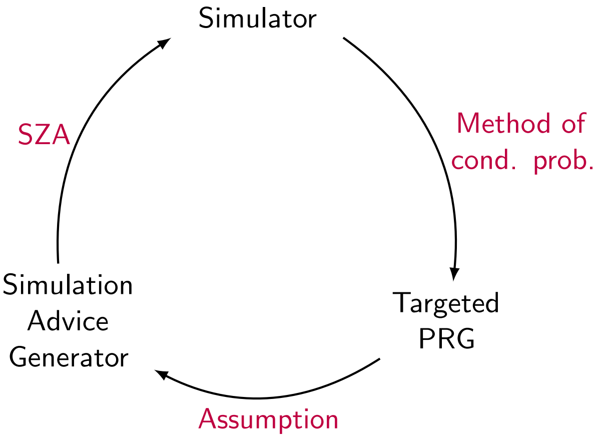
Proof sketch

- SZA works on simulation advice generator without modification



Proof sketch

- SZA works on simulation advice generator without modification



Main result (in painful detail)

- **Theorem**: The following are equivalent:

Main result (in painful detail)

- **Theorem**: The following are equivalent:
  1. $$\bigcap_{\alpha>0} \text{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha>0} \text{promise-DSPACE}(\log^{1+\alpha} n)$$

Main result (in painful detail)

- **Theorem**: The following are equivalent:
  1. $$\bigcap_{\alpha>0} \text{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha>0} \text{promise-DSPACE}(\log^{1+\alpha} n)$$
  2. For all $\mu \in [0,1]$, for all suff. small $\sigma > \eta > 0$, for all $\gamma > 0$:

Main result (in painful detail)

- **Theorem**: The following are equivalent:
  1. $$\bigcap_{\alpha>0} \text{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha>0} \text{promise-DSPACE}(\log^{1+\alpha} n)$$
  2. For all $\mu \in [0,1]$, for all suff. small $\sigma > \eta > 0$, for all $\gamma > 0$:
     - If $\exists$ efficient targeted PRG with parameters

       $$s \leq O(\log^{1+\sigma} n), \quad \log(1/\varepsilon) = \log^{1+\eta} n, \quad \log m \geq \log^{\mu} n$$

Main result (in painful detail)

- **Theorem**: The following are equivalent:
  1. $$\bigcap_{\alpha>0} \text{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha>0} \text{promise-DSPACE}(\log^{1+\alpha} n)$$
  2. For all $\mu \in [0,1]$, for all suff. small $\sigma > \eta > 0$, for all $\gamma > 0$:
     - If $\exists$ efficient targeted PRG with parameters
     $$s \leq O(\log^{1+\sigma} n), \quad \log(1/\varepsilon) = \log^{1+\eta} n, \quad \log m \geq \log^{\mu} n$$
     - Then $\exists$ efficient simulation advice generator with parameters
     $$s' \leq O(\log^{1+\sigma+\gamma} n), \quad \log(1/\varepsilon') = \log^{1+\eta-\gamma} n,$$
     $$\log m' \geq \log^{\mu-\gamma} n, \quad \log a' \leq O(\log^{1+\eta+\gamma} n)$$

Main result (in painful detail)

- **Theorem**: The following are equivalent:
  1. $$\bigcap_{\alpha>0} \text{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha>0} \text{promise-DSPACE}(\log^{1+\alpha} n)$$
  2. For all $\mu \in [0,1]$, for all suff. small $\sigma > \eta > 0$, for all $\gamma > 0$:
     - If $\exists$ efficient targeted PRG with parameters
       $$s \leq O(\log^{1+\sigma} n), \quad \log(1/\varepsilon) = \log^{1+\eta} n, \quad \log m \geq \log^{\mu} n$$
     - Then $\exists$ efficient simulation advice generator with parameters
       $$s' \leq O(\log^{1+\sigma+\gamma} n), \quad \log(1/\varepsilon') = \log^{1+\eta-\gamma} n,$$
       $$\log m' \geq \log^{\mu-\gamma} n, \quad \log a' \leq O(\log^{1+\eta+\gamma} n)$$
- a' = number of advice bits

Main result (in painful detail)

- **Theorem**: The following are equivalent:
  1. $$\bigcap_{\alpha>0} \text{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha>0} \text{promise-DSPACE}(\log^{1+\alpha} n)$$
  2. For all $\mu \in [0,1]$, for all suff. small $\sigma > \eta > 0$, for all $\gamma > 0$:
     - If $\exists$ efficient targeted PRG with parameters
       $$s \leq O(\log^{1+\sigma} n), \quad \log(1/\varepsilon) = \log^{1+\eta} n, \quad \log m \geq \log^{\mu} n$$
     - Then $\exists$ efficient simulation advice generator with parameters
       $$s' \leq O(\log^{1+\sigma+\gamma} n), \quad \log(1/\varepsilon') = \log^{1+\eta-\gamma} n,$$
       $$\log m' \geq \log^{\mu-\gamma} n, \quad \log a' \leq O(\log^{1+\eta+\gamma} n)$$

- a' = number of advice bits
- "Efficient": Space complexity $\leq O$(seed length)

Conclusion

- This material is based upon work supported by
  - NSF GRFP Grant No. DGE-1610403
  - NSF Grant No. NSF CCF-1423544
- Thanks for your attention!
- Any questions?