On the complexity of strings and the power of randomized algorithms

Marius Zimand

University of Bucharest, March 28, 2014
Randomized Algorithms

- Cooking: sauteing onions.
- Polling
- Monte Carlo algorithms in numerical analysis
- Sorting and searching, combinatorial algorithms, primality checking, ....
BIG QUESTION (we’ll try some answers later)

- Is randomness intrinsically useful?

- Yes, sure: Game Theory, Cryptography (randomness is in the model)

- What about computational tasks? Is there a computational task that can be solved with randomness, but cannot be solved without?

  (Computational task: Given an input $x$, find a solution $y$ that satisfies a predicate $P(x, y)$)
Is randomness useful for computational tasks?

- Common perception: “What can be done using randomness, can also be done without, but maybe slower.”

- It is now believed that $P = BPP$.

- If the solution of the task is unique, then we can find it by deterministic simulation.

- [de Leeuw, Moore, Shannon, Shapiro’56] If a function can be computed with probability $\alpha > 0$, then it is computable.
Complexity of strings

- 0000000000000000
Complexity of strings

- 0000000000000000
  Print a list of sixteen 0
Complexity of strings

- 0000000000000000
  Print a list of sixteen 0

- 1001001111101100
Complexity of strings

- 0000000000000000
  Print a list of sixteen 0

- 100100111101100
  Print twice a 1 followed by two 0, then five 1, one 0, two 1, two 0
Complexity of strings

- 0000000000000000
  Print a list of sixteen 0

- 100100111101100
  Print twice a 1 followed by two 0, then five 1, one 0, two 1, two 0

- Kolmogorov complexity of a string = the length of the shortest algorithm that can generate it.
  Notation: $C(x)$. 

- If $|x| = n$, then $C(x) \leq n + O(1)$. 

- The number of strings with complexity less than $k$ is at most $2^k$. 

Marius Zimand  
Complexity and Randomized Computation  
2013 5 / 32
Complexity of strings

- 0000000000000000
  Print a list of sixteen 0

- 1001001111101100
  Print twice a 1 followed by two 0, then five 1, one 0, two 1, two 0

- Kolmogorov complexity of a string = the length of the shortest algorithm that can generate it.
  Notation: \( C(x) \).

- If \( |x| = n \), then \( C(x) \leq n + O(1) \).
  The number of strings with complexity less than \( k \) is at most \( 2^k \).
Is randomness useful for computational tasks (2)?

- Task: Input $n$, Find an $n$-bit string $x$ with $C(x) \geq n/2$.
- Not computable, but if we toss a coin $n$ times, we get what we want.
- This example “showing” the usefulness of randomness is trivial and unconvincing.
- The non-computability of output comes directly from non-computability of the random coins.
The really interesting questions:

Are there **non-trivial** tasks solvable with randomness, but not solvable without?

If YES, how **little** randomness is needed to solve a **non-trivial** task?
List approximation for short programs

- $U$ - universal TM, $U(p) = x$, we say $p$ is a program for $x$.
- $C(x) = \min\{|p| \mid p \text{ program for } x\}$.
- $C(x)$ - canonical example of an uncomputable function.
- Finding a shortest program for $x$: also uncomputable.
List approximation for short programs

- $U$ - universal TM, $U(p) = x$, we say $p$ is a program for $x$.
- $C(x) = \min\{|p| \mid p \text{ program for } x\}$.
- $C(x)$ - canonical example of an **uncomputable** function.
- Finding a shortest program for $x$: also uncomputable.

**Question:** Is it possible to compute a short list containing a short program for $x$?
List approximation for short programs

- \( U \) - universal TM, \( U(p) = x \), we say \( p \) is a program for \( x \).
- \( C(x) = \min\{|p| \mid p \text{ program for } x\} \).
- \( C(x) \) - canonical example of an \textbf{uncomputable} function.
- Finding a shortest program for \( x \): also uncomputable.

\textbf{Question}: Is it possible to compute a short list containing a short program for \( x \)?

\textbf{Question}: Is it possible to compute a short list containing a short program for \( x \) in short time?
DEFINITION. \( p \) is a \( c \)-short program for \( x \) if \( U(p) = x \) and \( |p| \leq C(x) + c \).

DEFINITION. A function \( f \) is a list approximator for \( c \)-short programs if \( \forall x, f(x) \) is a finite list containing a \( c \)-short program for \( x \).
Results from [BMVZ]

- There exists a computable list approximator $f$ for $O(1)$-short programs, with list size $O(n^2)$.

- For any computable list approximator for $c$-short programs, list size is $\Omega(n^2/(c + 1)^2)$.

- There exists a poly.-time computable list approximator for $O(\log n)$-short programs, with list size $\text{poly}(n)$.  

Marius Zimand

Complexity and Randomized Computation

2013 10 / 32
Results from [BMVZ 2013]

What about lists containing a shortest program?  
Answer: It depends on the universal machine.

- For some $U$, any computable list containing a shortest program for $x$ has size $2^n - O(1)$.

- For some $U$, there is a computable list of size $O(n^2)$ containing a shortest program.
Results after [BMVZ 2013]

[Teutsch] There exists a poly.-time computable list approximator for $O(\log n)$ $O(1)$ -short programs, with list size $\text{poly}(n)$.

See also [Z]: Short lists with short programs in short time - a short proof.

[BZ] There exists a randomized computable list approximator for $O(1)$ $O(\log n)$ -short programs, with list size $n^2$.

Lower Bounds: The parameters are essentially optimal.

[BZ] There exists a randomized poly.-time approximator for $O(\log^2 n)$-short programs with list size $n$. 
Results after [BMVZ 2013]

[Teutsch] There exists a poly.-time computable list approximator for $O(\log n)$ $O(1)$-short programs, with list size $\text{poly}(n)$.

See also [Z]: Short lists with short programs in short time - a short proof.

[BZ] There exists a randomized computable list approximator for $O(1)$ $O(\log n)$-short programs, with list size $n^2$.

Lower Bounds: The parameters are essentially optimal.

[BZ] There exists a randomized poly.-time approximator for $O(\log^2 n)$-short programs with list size $n$. 
Theorem

There exists an algorithm that

Input: $x \in \{0, 1\}^n$, $k \in \mathbb{N}$, $\delta > 0$

Output: list of size $\text{poly}(n/\delta)$, each element of length $k + O(\log(n/\delta))$

If $k = C(x)$ then $(1 - \delta)$ of the elements are programs for $x$. 

From here we get the $n$-sized list containing a short program for $x$ with prob. $(1 - \delta)$: Run the algorithm for each $k = 1, 2, \ldots, n$ and pick one random element from each list.
Theorem

There exists an algorithm that

Input: $x \in \{0, 1\}^n$, $k \in \mathbb{N}$, $\delta > 0$

Output: list of size $\text{poly}(n/\delta)$, each element of length $k + O(\log(n/\delta))$

If $k = C(x)$ then $(1 - \delta)$ of the elements are programs for $x$.

Theorem

There exists a poly-time algorithm that

Input: $x \in \{0, 1\}^n$, $k \in \mathbb{N}$, $\delta > 0$

Output: list of size $2^{\log^2(n/\delta)}$, each element of length $k + O(\log^2(n/\delta))$

If $k = C(x)$ then $(1 - \delta)$ of the elements are programs for $x$.

(each element of the list printed in poly time).
Theorem

There exists an algorithm that
Input: $x \in \{0, 1\}^n$, $k \in \mathbb{N}$, $\delta > 0$
Output: list of size $\text{poly}(n/\delta)$, each element of length $k + O(\log(n/\delta))$

If $k = C(x)$ then $(1 - \delta)$ of the elements are programs for $x$.

Theorem

There exists a poly-time algorithm that
Input: $x \in \{0, 1\}^n$, $k \in \mathbb{N}$, $\delta > 0$
Output: list of size $2^{\log^2(n/\delta)}$, each element of length $k + O(\log^2(n/\delta))$

If $k = C(x)$ then $(1 - \delta)$ of the elements are programs for $x$.

(each element of the list printed in poly time).

From here we get the $n$-sized list containing a short program for $x$ with prob. $(1 - \delta)$:
Run the algorithm for each $k = 1, 2, \ldots, n$ and pick one random element from each list.
**Key tool:** bipartite graphs $G = (L, R, E \subseteq L \times R)$ with the rich owner property:

For any $B \subseteq L$ of size $|B| \approx K$, most $x$ in $B$ own most of their neighbors (these neighbors are not shared with any other node from $B$).
• \( x \in B \) owns \( y \in N(x) \) w.r.t. \( B \) if \( N(y) \cap B = \{x\} \).

• \( x \in B \) is a rich owner if \( x \) owns \((1 - \delta)\) of its neighbors w.r.t. \( B \).

• \( G = (L, R, E \subseteq L \times R) \) has the \((K, a, \delta)\)-rich owner property if for all \( B \) with \( K \leq |B| \leq a \cdot K \), \((1 - \delta)\) of the elements of \( B \) are rich owners w.r.t. \( B \).
Theorem

There exists a computable (uniformly in n) graph with the rich owner property for 
\((2^k, a = O(1), \delta)\) with:

- \(L = \{0, 1\}^n\)
- \(R = \{0, 1\}^{k+O(\log(n/\delta))}\)
- \(D(\text{left degree}) = \text{poly}(n/\delta)\)

Similar for poly-time \(G\) but overhead for \(R\) is \(O(\log^2(n/\delta))\) and \(D = 2^{O(\log^2(n/\delta))}\).

We obtain our lists:

- List for \(x\): \(N(x)\)
- Any \(p \in N(x)\) owned by \(x\) w.r.t. \(B = \{x' \mid C(x') \leq k\}\) is a program for \(x\).

How to construct \(x\) from \(p\): Enumerate \(B\) till we find an element that owns \(p\). This is \(x\).
Building graphs with the rich owner property

- Step 1: most neighbors of $x$ are shared with only $\text{poly}(n)$ many other nodes.
- Step 2: most neighbors of $x$ are shared with no other nodes.

Step 1 is done with extractors that have small entropy loss. Step 2 is done by hashing.
extractors

$E : \{0, 1\}^n \times \{0, 1\}^d \rightarrow \{0, 1\}^m$ is a $(k, \epsilon)$-extractor if for any $B \subseteq \{0, 1\}^n$ of size $|B| \geq 2^k$ and $X$ unif. distrib in $B$, and for any $A \subseteq \{0, 1\}^m$,

$$|\text{Prob}(E(X, U_d) \in A) - \text{Prob}(A)| \leq \epsilon,$$

or in other words

$$\left| \frac{|E(B, A)|}{2^k \cdot 2^d} - \frac{|A|}{2^m} \right| \leq \epsilon$$

The entropy loss is $s = k + d - m$. 
Step 1

**GOAL :** \( \forall B \subseteq L \text{ with } |B| \approx K, \) most nodes in \( B \) share most of their neighbors with only \( \text{poly}(n) \) other nodes from \( B \).

We can view an extractor \( E \) as a bipartite graph \( G_E \) with \( L = \{0, 1\}^n \), \( R = \{0, 1\}^m \) and left-degree \( D = 2^d \).

If \( E \) is a \((k, \epsilon)\)-extractor, then for any \( B \subseteq L \) of size \( |B| \approx 2^k \):
most \( x \in B \) share most of their neighbors with only \( O(1/\epsilon \cdot 2^s) \) other nodes in \( B \).

By the probabilistic method: There are extractors with entropy loss \( s = O(\log(1/\epsilon)) \) and log-left degree \( d = O(\log n/\epsilon) \).

[Guruswami, Umans, Vadhan, 2009] Poly-time extractors with entropy loss \( s = O(\log(1/\epsilon)) \) and log-left degree \( d = O(\log^2 n/\epsilon) \).

So for \( 1/\epsilon = \text{poly}(n) \), we get our GOAL.
**Step 2**

**GOAL:** Reduce sharing most neighbors with $\text{poly}(n)$ other nodes, to sharing them with no other nodes.

Let $x_1, x_2, \ldots, x_{\text{poly}(n)}$ be $n$-bit strings.

Consider $p_1, \ldots, p_T$ the first $T$ prime numbers, where $T = (1/\delta) \cdot n \cdot \text{poly}(n)$.

For every $x_i$, for $(1 - \delta)$ of the $T$ prime numbers, $(x_i \mod p)$ is unique in $(x_1 \mod p, \ldots, x_{\text{poly}(n)} \mod p)$.

In this way, by "splitting" each edge into $T$ new edges we reach our GOAL.

Cost: overhead of $O(\log n)$ to the right nodes and the left degree increases by a factor of $T = \text{poly}(n)$.
Lower bounds for probabilistic list approximation

parameters of interest:

- $T =$ size of the list
- $r =$ number of random bits
- $c =$ $|\text{short program}| - |\text{shortest program}|$.

Main result: $T = n$, $r = O(\log n)$, $c = O(\log n)$.

Lower bounds: essentially, no parameter can be reduced while conserving the other two.
lower bound on $r$

- $T$ = size of the list
- $r$ = number of random bits
- $c = |\text{short program}| - |\text{shortest program}|$.

If $T = n$ and $c = O(\log n)$, then $r > \log n - O(\log \log n)$.

Proof. If $r$ would be smaller, we would deterministically get a list of size $< n^2/c^2$, contradicting the lower bound [BMVZ].
lower bound on $r$

- $T =$ size of the list
- $r =$ number of random bits
- $c =$ $|\text{short program}| - |\text{shortest program}|$.

If $T = n$ and $c = O(\log n)$, then $r > \log n - O(\log \log n)$.

Proof. If $r$ would be smaller, we would deterministically get a list of size $< n^2/c^2$, contradicting the lower bound [BMVZ].
lower bound on $c$

- $T =$ size of the list
- $r =$ number of random bits
- $c = |\text{short program}| − |\text{shortest program}|$.

If $T = n$, then $c = O(\log n)$.

Proof

$L_\rho =$ list when randomness is $\rho$.

$P =$ set of $c$-short programs for $x$. $\ell = |P| = O(2^c)$.

- At least half of the lists $L_\rho$, $\rho \in \{0,1\}^r$ contain an element of $P$.
- So some element of $P$ appears in $1/2\ell$ of the lists.
- For each $m = 1, 2, \ldots, n$, select strings of length between $m$ and $m + c$ appearing in $1/2\ell$ of the lists. A $c$-short program will be here.
- Let $s_m$ be the number of elements selected at iteration $m$. The elements selected at iteration $m$ occur at least $s_m \cdot \frac{2^r}{2\ell}$ times.
- So

$$2^r \cdot T \geq s_1 \cdot \frac{2^r}{2\ell} + s_2 \cdot \frac{2^r}{2\ell} + \ldots + s_n \cdot \frac{2^r}{2\ell}.$$ 

- So, $s_1 + s_2 + \ldots + s_n \leq T \cdot 2\ell$.
- By [BMVZ] lower bound, the total number of selected elements is $\Omega(n^2/c^2)$.
- So $T \cdot 2\ell = \Omega(n^2/c^2)$, and the conclusion follows.
Lower bounds for the probabilistic approximation

lower bound on $c$

- $T = \text{size of the list}$
- $r = \text{number of random bits}$
- $c = |\text{short program}| - |\text{shortest program}|$.

If $T = n$, then $c = O(\log n)$.

Proof

$L_\rho = \text{list when randomness is } \rho$.

$\mathcal{P} = \text{set of } c\text{-short programs for } x$. $\ell = |\mathcal{P}| = O(2^c)$.

- At least half of the lists $L_\rho$, $\rho \in \{0, 1\}^r$ contain an element of $\mathcal{P}$.
- So some element of $\mathcal{P}$ appears in $1/2\ell$ of the lists.
- For each $m = 1, 2, \ldots, n$, select strings of length between $m$ and $m + c$ appearing in $1/2\ell$ of the lists. A $c\text{-short program will be here}.$
- Let $s_m$ be the number of elements selected at iteration $m$. The elements selected at iteration $m$ occur at least $s_m \cdot \frac{2^r}{2\ell}$ times.
- So $2^r \cdot T \geq s_1 \cdot \frac{2^r}{2\ell} + s_2 \cdot \frac{2^r}{2\ell} + \ldots + s_n \cdot \frac{2^r}{2\ell}$.
- So, $s_1 + s_2 + \ldots + s_n \leq T \cdot 2\ell$.
- By [BMVZ] lower bound, the total number of selected elements is $\Omega(n^2/c^2)$.
- So $T \cdot 2\ell = \Omega(n^2/c^2)$, and the conclusion follows.
Lower bounds for the probabilistic approximation

**lower bound on** \( c \)

- \( T \) = size of the list
- \( r \) = number of random bits
- \( c = |\text{short program}| - |\text{shortest program}|. \)

If \( T = n \), then \( c = O(\log n) \).

**Proof**

\( L_\rho \) = list when randomness is \( \rho \).
\( \mathcal{P} \) = set of \( c \)-short programs for \( x \). \( \ell = |\mathcal{P}| = O(2^c) \).

- At least half of the lists \( L_\rho, \rho \in \{0, 1\}^r \) contain an element of \( \mathcal{P} \).
- So some element of \( \mathcal{P} \) appears in \( 1/2\ell \) of the lists.
- For each \( m = 1, 2, \ldots, n \), select strings of length between \( m \) and \( m + c \) appearing in \( 1/2\ell \) of the lists. A \( c \)-short program will be here.
- Let \( s_m \) be the number of elements selected at iteration \( m \). The elements selected at iteration \( m \) occur at least \( s_m \cdot \frac{2^r}{2\ell} \) times.
- So

\[
2^r \cdot T \geq s_1 \cdot \frac{2^r}{2\ell} + s_2 \cdot \frac{2^r}{2\ell} + \cdots + s_n \cdot \frac{2^r}{2\ell}.
\]

So, \( s_1 + s_2 + \ldots + s_n \leq T \cdot 2\ell \).
- By [BMVZ] lower bound, the total number of selected elements is \( \Omega(n^2/c^2) \)
- So \( T \cdot 2\ell = \Omega(n^2/c^2) \), and the conclusion follows.
lower bound on $T$

- $T =$ size of the list
- $r =$ number of random bits
- $c = |\text{short program}| - |\text{shortest program}|.$

$T = \Omega(n/c)$.

Proof. If $T$ were smaller, we could obtain a list of lengths of sublinear size containing $C(x)$. Contradicts lower bound from [Beigel et al., 2006].
Lower bounds for the probabilistic approximation

lower bound on $T$

- $T =$ size of the list
- $r =$ number of random bits
- $c =$ $\left| \text{short program} \right| - \left| \text{shortest program} \right|$.

$T = \Omega(n/c)$.

Proof. If $T$ were smaller, we could obtain a list of lengths of sublinear size containing $C(x)$. Contradicts lower bound from [Beigel et al., 2006].
Lower bounds for deterministic list approximation

Lower bound
Th. If a computable list contains a $c$-short program for $x$, then its size is $\Omega(n^2/(c + 1)^2)$.
Lower bounds for deterministic list approximation

Lower bound
Th. If a computable list contains a $c$-short program for $x$, then its size is $\Omega(n^2/(c + 1)^2)$. 

Bipartite graphs with online matching with overhead $c$. 

\[
\begin{array}{c}
\text{strings} \\
x \\
\end{array} \quad \begin{array}{c}
\text{short programs} \\
\end{array}
\]
Lower bounds for deterministic list approximation

Lower bound
Th. If a computable list contains a $c$-short program for $x$, then its size is $\Omega(n^2/(c+1)^2)$.

Bipartite graphs with online matching with overhead $c$.

- matching requests arrive one by one
- request $(x, k)$: match $x \in LEFT$ with a free node $y \in N(x)$ s.t. $|y| \leq k + c$
- Promise: $k \leq |x|$ and $\forall k$ there are $\leq 2^k$ requests $(*, k)$
- Requirement: all requests should be satisfied online (before seeing the next request).
Lower bounds for deterministic list approximation

Lower bound

Th. If a computable list contains a $c$-short program for $x$, then its size is $\Omega(n^2/(c + 1)^2)$.

Bipartite graphs with online matching with overhead $c$.

- matching requests arrive one by one
- request $(x, k)$: match $x \in LEFT$ with a free node $y \in N(x)$ s.t. $|y| \leq k + c$.
- Promise: $k \leq |x|$ and $\forall k$ there are $\leq 2^k$ requests $(\ast, k)$.
- Requirement: all requests should be satisfied online (before seeing the next request).
Lower bounds for deterministic list approximation

**Lower bound**

**Th.** If a computable list contains a $c$-short program for $x$, then its size is $\Omega(n^2/(c + 1)^2)$.

Bipartite graphs with online matching with overhead $c$.

- matching requests arrive one by one
- request $(x, k)$: match $x \in LEFT$ with a free node $y \in N(x)$ s.t. $|y| \leq k + c$.
- Promise: $k \leq |x|$ and $\forall k$ there are $\leq 2^k$ requests $(\ast, k)$.
- Requirement: all requests should be satisfied online (before seeing the next request).
Short lists - combinatorial characterization

Theorem

∃ (poly time) computable G with on-line matching with overhead c and the matching strategy is computable
IFF (almost)
∃ (poly time) computable list of size \text{deg}(x) containing a c + O(1)-short program for x

Proof. (⇒ only)

• Enumerate strings in \text{LEFT}(G) as they are produced by \text{U}.
• Say, at step s, \text{U}(q) = x, with |q| = k.
• Make request (x, k).
• x is matched with y of length k + c.
• y is a program for x.
• Why: on input y, re-play the matching process till some left string is matched to it; output this left string, which will be x.

So when q is the shortest program for x, we get a program for x of length C(x) + c.
The list consists of x’s neighbors in G.
Theorem

∃ (poly time) computable G with on-line matching with overhead c and the matching strategy is computable

IFF (almost)

∃ (poly time) computable list of size deg(x) containing a c + O(1)-short program for x

Proof. (⇒ only)

• Enumerate strings in LEFT(G) as they are produced by U.
• Say, at step s, U(q) = x, with |q| = k.
• Make request (x, k).
• x is matched with y of length k + c.
• y is a program for x.
• Why: on input y, re-play the matching process till some left string is matched to it; output this left string, which will be x.

So when q is the shortest program for x, we get a program for x of length C(x) + c.

The list consists of x’s neighbors in G.
Short lists - combinatorial characterization

Theorem

\[ \exists \ (\text{poly time}) \text{ computable } G \text{ with on-line matching with overhead } c \text{ and the matching strategy is computable} \]

IFF (almost)

\[ \exists \ (\text{poly time}) \text{ computable list of size } \deg(x) \text{ containing a } c + O(1)\text{-short program for } x \]

Proof. (⇒ only)

- Enumerate strings in \( \text{LEFT}(G) \) as they are produced by \( U \).
- Say, at step \( s \), \( U(q) = x \), with \( |q| = k \).
- Make request \((x, k)\).
- \( x \) is matched with \( y \) of length \( k + c \).
- \( y \) is a program for \( x \).
- Why: on input \( y \), re-play the matching process till some left string is matched to it; output this left string, which will be \( x \).

So when \( q \) is the shortest program for \( x \), we get a program for \( x \) of length \( C(x) + c \).

The list consists of \( x \)'s neighbors in \( G \).
Lower bounds - 2

Th. If a computable list contains a $c$-short program for $x$, then its size is $\Omega(n^2/(c + O(1))^2)$.

- $x \rightarrow$ list $f(x) = \{y_1, \ldots, y_t\}$ contains a $c$-short program for $x$.
- bipartite $G$: $\text{LEFT} = \{0, 1\}^n$, $\forall x \in \text{LEFT}$, $(x, y_i) \in E$ for all $y_i \in f(x)$.
- $G$ has on-line matching with overhead $c \Rightarrow$ also has off-line matching with overhead $c$.
- $G[\ell, k]$ is $G$ from which we cut right nodes $y$ with $|y| < \ell$ or $|y| > k$.
- $\forall k, G[0, k + c]$ is $(2^k, 2^k)$-expander
- So, $\forall k, G[k - 1, k + c]$ is $(2^k, 2^{k-1} + 1)$-expander.

**LEMMA**

Graph $G$ has $|\text{LEFT}| = 2^\ell$, $|\text{RIGHT}| = 2^{k+c}$, and is a $(2^k, 2^{k-1} + 1)$-expander. Then $\exists x \in \text{LEFT}$ with $\deg(x) \geq \min(2^{k-2}, \frac{\ell-k}{c+2})$. 
• Take $k \in (n/4, n/2]$. By LEMMA (with $\ell = 3n/4$), in $G[k - 1, k + c]$, all $x \in LEFT$, except $2^{3n/4}$, have $\deg(x) \geq \frac{n}{4(c+3)}$. 

\begin{itemize}
  \item [0, 1]^n
  \item $[k - 1, k + c]$
  \item $\deg \geq \frac{n}{4(c+3)}$
\end{itemize}
• Take $k \in (n/4, n/2]$. By LEMMA (with $\ell = 3n/4$), in $G[k - 1, k + c]$, all $x \in \text{LEFT}$, except $2^{3n/4}$, have

$\deg(x) \geq \frac{n}{4(c+3)}$.

• Pick $n/4 < k_1 < k_2 < \ldots < k_s < n/2$, and $(c + 2)$ apart from each other;

$s \approx \frac{n}{4(c+2)}$.

• In each $G[k_i - 1, k_i + c]$, all left nodes, except $2^{3n/4}$, have $\deg \geq \frac{n}{4(c+3)}$.

• The RIGHT sets are disjoint.

• So, $\exists x \in \text{LEFT}$, with

$\deg(x) \geq s \cdot \frac{n}{4(c+3)} = \Omega\left(\frac{n^2}{(c+3)^2}\right)$.
SUMMARY

- One can compute a $O(n^2)$-sized list containing a $O(1)$-short program.
- Any computable list containing a $c$-short program has size $\Omega(n^2/(c + 1)^2)$.
- One can probabilistically compute an $n$-sized list containing a $O(\log n)$-short program. Parameters are essentially optimal.
- One can probabilistically compute in polynomial time an $n$-sized list containing a $O(\log^2 n)$-short program.
- One can compute in poly time a poly$(n)$-sized list containing a $O(1)$-short program.
Back to our BIG QUESTIONS

Are there non-trivial tasks solvable with randomness, but not solvable without?

If YES, how little randomness is needed to solve a non-trivial task?
Back to our BIG QUESTIONS

Are there non-trivial tasks solvable with randomness, but not solvable without?

If YES, how little randomness is needed to solve a non-trivial task?

Task: Given $x \in \{0, 1\}^n$ compute a list of $n$ elements that contains an $(O \log n)$-short program for $x$.

The task is not solvable deterministically (recall the $\Omega(n^2/c^2)$ lower bound for $c$-short programs [BMVZ]).

The task can be done probabilistically, with prob. error $\delta$.

The number of random bits is $O(\log n/\delta)$.

The similar task for $(O \log^2 n)$-short program for $x$ can be solved in probabilistic polynomial time with $O(\log^2 n)$ random bits.
Open Question

Are there non-trivial tasks that can be solved with $o(\log n)$ random bits, but cannot be solved deterministically?

Task: Defined by a predicate $P$. Given $x$, find a “solution” $y$ such that $P(x, y)$ is true.

The task is **trivial** if for some very simple function $g$, $g(x, r)$ is a solution for most $r$.

”very simple function”: projection + permutation (or maybe $NC_0$).
Multumesc pentru atentie.