Some remarks on the "Short lists for short programs" problem

Marius Zimand

CCR, Moscow, September 2013

BIG Question (we'll try some answers later)

- Is randomness 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.

Is randomness useful for computational tasks (2)?

- Task: Input n, Find an n-bit string x with $C(x) \geq n$.
- Not computable, but if we toss a coin n times, we get what we want.
- Task: on input x, find an extension xy such that $C(xy) > C(x)$. It has the same easy solution. We toss just a few coins.
- These examples "showing" the usefulness of randomness are trivial and non-convincing.
- The non-computability of output comes directly (or almost) 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?

Back to business...

Remarks on the "short lists for short programs" problem.

- 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.

- 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?

- 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?
- **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 $\mathrm{poly}(n)$.

Results from [BMVZ]

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.

New results after [BMVZ]

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

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

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

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

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

Lower Bounds: The parameters are essentially optimal.

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

New results after [BMVZ]

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

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

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

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

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

Lower Bounds: The parameters are essentially optimal.

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

New results after [BMVZ]

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

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

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

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

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

Lower Bounds: The parameters are essentially optimal.

[Z] 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: $x \in \{0,1\}^n$, $k \in \mathbf{N}$, $\delta > 0$
Output: list of size $\mathrm{poly}(n/\delta)$, *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 an algorithm that
Input: $x \in \{0,1\}^n$, $k \in \mathbf{N}$, $\delta > 0$
Output: list of size $\mathrm{poly}(n/\delta)$, *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: $x \in \{0,1\}^n$, $k \in \mathbf{N}$, $\delta > 0$
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: $x \in \{0,1\}^n$, $k \in \mathbf{N}$, $\delta > 0$
Output: list of size $\mathrm{poly}(n/\delta)$, *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: $x \in \{0,1\}^n$, $k \in \mathbf{N}$, $\delta > 0$
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, δ)-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:
- $L = \{0,1\}^n$
- $R = \{0,1\}^{k+O(\log(n/\delta)}$
- $D(\textit{left degree}) = \mathrm{poly}(n/\delta)$

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. $B = \{x' \mid C(x') \le k\}$ 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 $\mathrm{poly}(n)$ many other nodes.
- Step 2: most 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, ϵ)-extractor if for any $B \subseteq \{0,1\}^n$ of size $|B| \geq 2^k$ and X unif. distrib in B, and for any $A \subseteq \{0,1\}^m$,

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

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 : $\forall B \subseteq L$ with $|B| \approx K$, most nodes in B share most of their neighbors with only $\mathrm{poly}(n)$ 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, ϵ)-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 whith 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 $1/\epsilon = \mathrm{poly}(n)$, we get our GOAL.

Step 2

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

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

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

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

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 $T = \mathrm{poly}(n)$, .

Lower bounds

parameters of interest:

- T = size of the list
- r = number of random bits
- c = |short program| − |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 = |short program| − |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 = |short program| − |shortest program|.

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

Proof (by Bruno Bauwens)

L_ρ = list when randomness is ρ.
$\mathcal{P}$ = set of c-short programs for x. $\ell = |\mathcal{P}| = O(2^c)$.
• At least half of the lists L_ρ, $\rho \in \{0,1\}^r$ 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

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

•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 bound on c

- T = size of the list
- r = number of random bits
- c = |short program| − |shortest program|.

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

Proof (by Bruno Bauwens)

L_ρ = list when randomness is ρ.
$\mathcal{P}$ = set of c-short programs for x. $\ell = |\mathcal{P}| = O(2^c)$.
• At least half of the lists L_ρ, $\rho \in \{0,1\}^r$ 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

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

•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 bound on c

- T = size of the list
- r = number of random bits
- c = |short program| − |shortest program|.

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

Proof (by Bruno Bauwens)

L_ρ = list when randomness is ρ.
$\mathcal{P}$ = set of c-short programs for x. $\ell = |\mathcal{P}| = O(2^c)$.
• At least half of the lists L_ρ, $\rho \in \{0,1\}^r$ 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

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

•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 bound on T

- T = size of the list
- r = number of random bits
- c = |short program| − |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 bound on T

- T = size of the list
- r = number of random bits
- c = |short program| − |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].

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 δ.

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 task 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).

Thank you.