Sublinear time local access random generators

Amartya Shankha Biswas (MIT)
Ronitt Rubinfeld (MIT and TAU)
Anak Yodpinyanee (MIT)
Huge random objects: How to generate? 
Up front? 
Locally... on the fly?
Generating large random graph

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Generate “on the fly”?

What if required to be symmetric? $d$-regular? Support “next-neighbor” queries?
A challenge: How to handle dependencies?

Sources of dependencies:
Model, supported queries,...
Some prior work
Implementation of Huge Random Objects


- Model introduced and formalized in [Goldreich-Goldwasser-Nussboim 2003]
  - Generators for random functions, codes, graphs,…
  - Give important primitives
    - e.g. Sampling from binomial distribution, interval-sum queries for functions (see also [Gilbert, Guha, Indyk, Kotidis, Muthukrishnan, Strauss 2002])
  - Generators provide (limited) queries to random graphs with specified property
    - e.g. Planted Hamiltonian cycle
    - Focus on indistinguishable (under small number of queries and poly time) and truthful implementations (more on this by [Naor Nussboim Tromer 05] [Alon Nussboim 07])
Implementations of random $G(n,p)$ graphs
[Goldreich Goldwasser Nussboim 03]

• Graphs generated:
  • Have a specific property e.g.,
    colorability, clique, connectedness,
    bipartiteness

• Queries:
  • Adjacency
  • Up to polylog queries
Implementation I of sparse $G(n,p)$ graph
[GGN]

• Graphs generated:
  • Degree at most polylog
  • Indistinguishable from uniform distribution for few queries

• Queries:
  • Adjacency, all-neighbor
  • Up to polylog queries
Implementation II of sparse $G(n,p)$ graph

[Naor-Nussboim 2007]

• Graphs generated:
  • Degree at most polylog

• Queries:
  • Adjacency, all-neighbor
  • Bound on number stated in paper, but not necessary in some settings

• Graphs generated:
  • essentially a rooted tree/forest structure
  • Highly sequential random process
  • Sparse, but degree not bounded

• Queries:
  • Adjacency
  • Introduce next-neighbor query (implement with \text{polylog}(n) resources)
  • No bound on number

Give local implementation without reconstructing full history!!
Models
Two models for random generation of graphs

**Huge random graphs/objects**  
[Goldreich Goldwasser Nussboim]
- Huge = exponential size
- User will not query more than poly locations
- In some versions, sufficient to generate graph that “looks” random to poly time algorithm

**Big random graphs/objects**  
[Even Levi Medina Rosen]
- Big = poly size
- Might eventually write down the whole graph, but don’t want to pay cost up-front
- End result should be random according to the claimed process
“On-the-fly” Sampler
(Adapted from [Even-Levi-Medina-Rosen 2017])

Random bits → Generation Algorithm → Random Object (in memory) → User

Queries reveal partial information
Eventually, entire object sampled, stored in memory

Standard paradigm

“on-the-fly” sampler
Desiderata:

• Efficiency:
  • Answer queries in polylogarithmic time

• Succinct Representation

• Consistency over future queries:
  • Can store past decisions
  • eventually give complete valid sample

• Distribution equivalence:
  • Output distribution is $\epsilon$-close (in $\ell_1$-distance) to goal distribution

• Not considered today:
  • pseudo-random distributions (indistinguishable from goal distribution, or preserving properties)
  • bounds on number of queries
  • Very succinct representation

Error from implementation issues
Possible queries:

- **Vertex-pair (adjacency):** Is edge \((u,v)\) present?
- **All-Neighbors:** What are all neighbors of \(u\)?
- **Degree:** What is \(\text{degree}(u)\)?
- **ith neighbor:** What is \(ith\) neighbor of \(u\)?
- **Next-neighbor:** What is next neighbor of \(u\)?
- **Random-neighbor:** Output random neighbor of \(u\)?

considered by [GGN] [NN]

considered by [Even Levi Medina Rosen 2017]
today
New Generators
Today’s Goal:
Graph models supporting typical graph queries

\[ G(n,p) \]

Community structure: Stochastic Block Model

Small world graphs
$G(n,p)$ graphs
Vertex-pair query: Is there an edge from u to v?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Generate “on the fly”? toss coins as needed
All-neighbor queries for sparse G(n,p):
Implementation adapted from [NN07]

• Edges defined via “Ports”:
  • For each node, pick “ports”: "1" (green)
  • Ports matched to others on the fly: indicated via edge (red)

• Two equivalent processes:
  • Pick number of edges for each u and sum to get total edges
  • Picking total number of edges and dividing among u’s
    → Compute u’s locations using locally computable interval-summable functions [GGIKMS 02] [GGN03][NN07]

• Given an “all neighbor” query vertex (6), match its ports to other unmatched ports
  • Match each port to random open position in degree sequence
Next-Neighbor Query: what is u’s next neighbor?

Dense case: $p \geq 1/poly(\log n)$

- Algorithm:
  - Start at last found neighbor
  - Go down row, flipping coins to fill empty entries, until find neighbor.
- Time $O(1/p)$.

Sparse case: $p \leq poly(\log n)/n$

- Algorithm: Use “all neighbor” query [Naor Nussboim 07]
- Time $O(E[degree]) = O(polylog n)$

Intermediate case: (e.g. $p = \frac{1}{\sqrt{n}}$)

- Idea: Sample “length of 0’s run” according to hypergeometric distribution $p(1 - p)^i$
- Challenge: some entries already filled in!

Can we do $o(1/p)$ for $p = o(1)$?
Skip-sampling for next-neighbor queries: The case of directed graphs

Algorithm idea:

Pick length of 0-run according to hypergeometric distribution (via binary search on CDF):

$$\sum_{k=0}^{b-a-1} p \ (1 - p)^k = 1 - (1 - p)^{b-a}$$

Fill in next entry \((i, j+k)\) with a 1
Skip-sampling for next-next neighbor queries: Undirected graphs

Algorithm idea:
Pick length of 0-run according to hypergeometric distribution:

$$\sum_{k=0}^{b-a-1} p (1-p)^k = 1 - (1-p)^{b-a}$$

Fill in next entry $(i, j+k)$ with a 1

<table>
<thead>
<tr>
<th>row $i$</th>
<th>0 1 0 0 0 0 1</th>
<th>0 0 0 0 0 1</th>
<th>0 0 0 0 0 1</th>
<th>0 0 0 0 0 1</th>
<th>0 0 0 0 0 1</th>
<th>0 0 0 0 0 1</th>
<th>0 0 0 0 0 1</th>
</tr>
</thead>
</table>

Some are determined by other neighbor? Yields correct distribution? Need to write all 0s?
Implementation of next neighbor queries: (assume no adjacency queries)

• For each node \( i \) maintain:
  1. last seen neighbor \( j \) (row entries 1..\( j \) are determined, and \( j \) is a “1”)
  2. list of “1”s coming before \( j \) (everything else is “0”)
  3. remaining “1”s via min-heap
  4. Keep track of “0”s on row implicitly

![Diagram showing the implementation of next neighbor queries]

Only keep track of 1’s
Skip-sampling for next-neighbor queries

choose $k$ according to geometric distribution
If $j+k > \text{next 1 in i's heap}$, output next 1 in i’s heap
else check if $(i, j+k)$ previously decided by $j+k$
  if 0 then re-roll
  else add $(i, j+k)$ to heaps for $i$ and $j+k$

Why correct distribution?

if “1”, then neighbor has told i about it

if before next “1” and land here, pick new length starting from here

Some are determined by other neighbor?
Local-Access Generators – Difficulties

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(900380)</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**next-neighbor**
- how to sample for **next-neighbor**?
- how to inform (non-)neighbors?
- how to find **next-neighbor** when some choices are already decided?

**vertex-pair**
- how to maintain information without obstructing **next-neighbor**?

**careful analysis can mitigate these .. but**

**random-neighbor**
- how to sample without knowing/committing to a degree?
Random-Neighbor Query: output random neighbor of $i$

**Dense case:** $p \geq 1/poly(\log n)$
- Algorithm:
  - repeat until find neighbor:
    - pick random $j$
    - do vertex pair query on $(i, j)$
- Time $O(1/p)$.

**Sparse case:** $p \leq poly(\log n)/n$
- Algorithm: Use “all neighbor” query [Naor Nussboim 07]
- Time $O(E[\text{degree}]) = O(poly\log n)$

**Intermediate case:** (e.g. $p = \frac{1}{\sqrt{n}}$)
- ???
  - we don’t even know degree?

Can we do $o(1/p)$ for $p = o(1)$?
Implementation of Random-Neighbor queries via Bucketing

Plan: **Equipartition** each row into **contiguous buckets** such that:
- Expected # of neighbors in a bucket is a constant
  - w.h.p. 1/3 of buckets are non-empty
  - w.h.p. no bucket has more than \( \log n \) neighbors

(drumroll...)
- can write down all \( \log n \) neighbors for each bucket! (assuming you can figure them out)

How many buckets?

\[ pn, \text{ each of size } 1/p \]

Note that both size and number of buckets can be big
Random Neighbors with rejection sampling

**Bucketing:**
- Expected number of neighbors in a bucket: $\Theta(1)$ expected, $\leq O(\log n)$ w.h.p.
- $v$: \[0 \; 1 \; 0 \; 0 \; 1 \; \ldots \quad \Rightarrow \quad \text{Select bucket} \]

**Step 1:** Pick a uniform random bucket “fill” this bucket if needed

**Step 2:** Pick a uniform random neighbor $u$

**Step 3:** Return $u$ with probability $\frac{1}{\#\text{neighbors in the bucket}} \cdot \frac{1}{\#\text{neighbors in bucket}} \cdot \frac{\#\text{neighbors in bucket}}{O(\log n)} \approx \frac{\Omega(1/\log n)}{\#\text{neighbors of } v}$

$\mathbb{P}[\text{return } u] = \frac{1}{\#\text{buckets}} \times \frac{1}{\#\text{neighbors in bucket}} \times \frac{\#\text{neighbors in bucket}}{O(\log n)} \approx \frac{\Omega(1/\log n)}{\#\text{neighbors of } v}$

$\mathbb{P}[\text{return any neighbor}] \approx \Omega(1/\log n) \Rightarrow O(\log n)$ iterations suffice
How to fill a bucket?

• Bucket may be *indirectly* filled in certain locations
  • "1" entries reported when created
  • "0" entries not reported but can query from complementary bucket

```
? 1 ? ? ? 0 ? 1 0 ? ?
```

• First, skip-sample in the bucket ignoring the existing entries
```
0 0 1 0 0 1 1 0 0 1 0
```

• Re-insert all *indirectly filled* (red) "1" entries: \{2,8\}
• For each new (green) "1" entry: remove if coincides with indirectly filled "0" entries
```
0 1 1 0 0 1 1 1 0 1 0
```

• Why fast? # of "1" entries is bounded by log n
Nice fact:
Bucketing improves next-neighbor queries too!
Stochastic Block Model
Stochastic Block Model

• R communities each labelled via “color”
  • $P_{ij}$ specifies probability of edges between community i and j
• how to assign colors to nodes?
  • contiguous blocks?
    • Algorithms for SBM are usually concerned with community detection
  • randomly?
    • assume given counts of members of each color
Skip-sampling probabilities

• New requirement
  • count # of members of each color within a specified interval \([a,b]\)
    • E.g., Allows computing CDF of skip-sampling distributions
  • Equivalently: sample from the multivariate hypergeometric distribution
Count generator: Sample colors in an interval  
(see also GGIKMS, GGN, NN)

Tree constructed “lazily”: only as required
Another use: Partially Sampling a Random Walk

Query Height(t) returns position of random walk at time t
Small world graphs
Small-World Model [Kleinberg]

Edges:
- Uniform grid
- Directed long range edge $(u, v)$ with probability $c/d(u, v)^2$

Will answer “All-neighbor queries” (implies implementation for other queries)
Small-World Model: All neighbor queries

- Model:
- Uniform grid
- \((u, v)\) with probability \(\frac{c}{d(u, v)^2}\)

For increasing \(d\):
(1) Sample next \(d\) which has nbrs of distance \(d\)
(2) skip sample among all \(O(d)\) nbrs at distance \(d\)
Future directions

Other random objects?

Support degree, ith neighbor queries?

Local generation without history?
Thank you!