Generating random graphs

Giovanni Righini

Università degli Studi di Milano

Random graphs: requirements

Given a vertex set N, generate a random graph/digraph of given density.

The density is defined as

$$\rho = \frac{m}{m_{max}}$$

where $m_{max} = n(n-1)/2$ for undirected graphs and $m_{max} = n(n-1)$ for directed graphs.

In a random graph/digraph it is required that

- all edges/arcs have the same probability of being selected;
- such probabilities are independent;
- multiple copies of edges/arcs are not allowed;
- self-loops are not allowed.

Assumptions

Assumption 1. A generator of pseudo-random numbers is available: we can draw integer numbers at random within any specified interval with uniform probability distribution.

Assumption 2. The computational complexity of this operation is $O(1)$, i.e. it is an atomic operation whose computing time does not depend on the interval width.

Remark. This is not a combinatorial optimization problem. It is combinatorial, but it does not imply to optimize any objective function.

Observation

The integer number p of edges/arcs to be selected is given by $p = \rho m$, possibly rounded.

If $\rho > 1/2$, i.e. $p > m/2$, it is faster to extract $m - p$ items from the ground set, instead of p, i.e. to select the edges/arcs of the complement graph.

Hence, without loss of generality, we assume

$$p \leq m/2$$

from now on.

Algorithm 1: general description

Since

- we want to select p elements from a set of m,
- we have a very efficient sub-routine to select an element at random,

a possible algorithm is:

- For p times, generate a random number in $[1, \ldots, m]$.

Algorithm 1: pseudo-code

Algorithm 1 Select p elements independently

for $k = 1, \ldots, p$ **do**
  $S[k] \leftarrow Random(1, m)$
return S

- k is an iteration counter;
- $Random(a, b)$: sub-routine that generates a random number between two integers a and b with u.p.d.;
- S: vector where the p selected elements are recorded.

Algorithm 1: correctness and complexity

Computational complexity: $O(p)$ in both time and space (best possible!).

Correctness: no guarantee that p distinct elements be selected.

Repetitions may occur (and they are likely to occur, especially for relatively large values of p).

Trade-off between the computational complexity (excellent) and the guarantee that the algorithm provides (weak).

Algorithm 2: general description

Avoid repetitions by explicitly selecting a subset of distinct elements.

For instance: select a set of p consecutive elements from the ground set $\{1, \ldots, m\}$, starting from a randomly selected initial element.

Algorithm 2: pseudo-code

Algorithm 2 Select a subset of p elements

$i \leftarrow Random(1, m)$
for $k = 1, \ldots, p$ **do**
  $S[k] \leftarrow i$
  if $i = m$ **then**
    $i \leftarrow 1$
  else
    $i \leftarrow i + 1$
return S

Algorithm 2: correctness and complexity

Complexity: $O(p)$ in time and space.

Correctness:

- all selected elements are distinct: no multiple copies are generated;
- all elements have the same probability of being selected;
- no self-loops are generated;
- the probabilities of being selected are not independent, as required.

So, we really need to extract each element by a distinct call to *Random*().

Algorithm 3: general description

Instead of avoiding repetitions a priori, we can detect them as soon as one occurs.

To immediately detect repetitions, we can keep a vector of binary flags associated with the elements of the ground set.

Whenever an element is selected, its flag is set to 1.

If an element whose flag is set to 1 is selected again, this is detected in $O(1)$ and the element is not inserted in S the second time.

Algorithm 3: pseudo-code

Algorithm 3 Discard the last element when a repetition occurs.

for $i = 1, \ldots, m$ **do**
  $f[i] \leftarrow 0$
$k \leftarrow 0$
while $k < p$ **do**
  $i \leftarrow Random(1, m)$
  if $f[i] = 0$ **then**
    $k \leftarrow k + 1$
    $S[k] \leftarrow i$
    $f[i] \leftarrow 1$
return S

Algorithm 3: correctness and complexity

Correctness. All requirements are now satisfied.

Complexity (space). $O(m)$, because it needs to allocate an array of m binary flags.

Complexity (time). The worst case is an infinite number of wasted iterations.

The average case must be analyzed.

- The operations within the while loop take $O(1)$ time.
- The number of times the while loop iterations are successful is p.
- We must evaluate the number of wasted iterations of the while loop.

Algorithm 3: average-case time complexity

Compute the probability of occurrence of a wasted iteration $\forall k = 0, \ldots, p-1$, with $p \leq m/2$.

Definition. W_k: number of wasted iterations when k elements have been selected.

$$P(W_k = w) = \left(\frac{k}{m}\right)^w \frac{m-k}{m}.$$

The expected value of W_k is given by

$$E[W_k] = \sum_{w=0}^{\infty} w \left(\frac{k}{m}\right)^w \frac{m-k}{m} = \frac{m-k}{m} \sum_{w=0}^{\infty} w \left(\frac{k}{m}\right)^w.$$

Setting $\eta = \dfrac{k}{m}$, the sum can be computed as follows:

$$\sum_{w=0}^{\infty} w\eta^w = \eta \sum_{w=0}^{\infty} w\eta^{w-1} = \eta \sum_{w=0}^{\infty} \frac{\partial(\eta^w)}{\partial\eta} = \eta \frac{\partial\left(\sum_{w=0}^{\infty} \eta^w\right)}{\partial\eta}.$$

Algorithm 3: average-case time complexity

Observing that $\eta < 1$,

$$\eta \frac{\partial \left(\sum_{w=0}^{\infty} \eta^w \right)}{\partial \eta} = \eta \frac{\partial \left(\frac{1}{1-\eta} \right)}{\partial \eta} = \eta \frac{1}{(1-\eta)^2} = \frac{\eta}{(1-\eta)^2}.$$

Replacing the sum back into the original expression, and replacing η with $\frac{k}{m}$,

$$E[W_k] = \frac{m-k}{m} \frac{\frac{k}{m}}{\left(\frac{m-k}{m} \right)^2} = \frac{k}{m-k}.$$

So, the expected number of wasted iterations increases more than linearly as k increases from 0 to $p-1$.

Algorithm 3: average-case time complexity

The expected total number of wasted iterations is

$$E[W] = \sum_{k=0}^{p-1} E[W_k] = \sum_{k=1}^{p-1} \frac{k}{m-k} = \sum_{k=1}^{p-1} \frac{k-m+m}{m-k} =$$

$$= \sum_{k=1}^{p-1} \left(-1 + \frac{m}{m-k} \right) = (1-p) + \sum_{k=1}^{p-1} \frac{m}{m-k} =$$

$$= (1-p) + m \sum_{k=1}^{p-1} \frac{1}{m-k}.$$

Let define $h = m - k$. Then we have

$$E[W] = (1-p) + m \sum_{k=1}^{p-1} \frac{1}{m-k} = (1-p) + m \left(\sum_{h=1}^{m-1} \frac{1}{h} - \sum_{h=1}^{m-p} \frac{1}{h} \right).$$

Algorithm 3: average-case time complexity

For the harmonic sum stopped at term a the following bounds hold:

$$\log(a+1) < \sum_{h=1}^{a} \frac{1}{h} < \log(a+1) + \gamma,$$

being $\gamma = 0.5772...$ the Euler-Mascheroni constant. Hence

$$\begin{aligned} E[W] &< 1 - p + m(\log(m) + \gamma - \log(m - p + 1)) = \\ &= 1 - \rho m + \gamma m + m \log \frac{m}{m - p + 1} = \\ &= 1 + m(\gamma - \rho) + m \log \frac{m}{m(1-\rho) + 1}. \end{aligned}$$

Being $\gamma > 0.5$ and $\rho \leq 0.5$, the term $(\gamma - \rho)$ is positive.

Therefore, for each fixed $\rho < 1/2$, when m grows to infinity we have

$$E[W] = O(m).$$

Since the number of successful iterations is equal to $p = \rho m$, then the expected total number of iterations is $O(m)$ for each given ρ.

Algorithm 3: correctness and complexity

Correctness. All requirements are satisfied.

Complexity (space). $O(m)$.

Complexity (time). The worst case is an infinite number of wasted iterations.

The average case is $O(m)$.

Drawbacks of Algorithm 3:

- its worst-case time complexity cannot be bounded;
- its $O(m)$ space complexity can be impractical for large graphs.

Algorithm 4: general description

Instead of a binary flag for each element, we can use a hash table to detect repetitions:

$$h : \{1, \ldots, m\} \mapsto \{1, \ldots, r\}$$

We keep a binary flag for each value $c \in \{1, \ldots, r\}$ and a corresponding list $L[c]$ $\forall c$.

Every time a number $i \in \{1, \ldots, m\}$ is generated at random

- compute $c = h(i)$;
- if the flag of c is 0, there is no repetition:
  - append i to the list $L[c]$
  - set the flag of c to 1.
- if the flag of c is 1, there can be a repetition:
  - scan $L[c]$ to possibly detect it.

Algorithm 4: pseudo-code

Algorithm 4 Using a hash function to detect repetitions.

for $c = 1, \ldots, r$ **do**
  $f[c] \leftarrow 0$
  $L[c] \leftarrow \emptyset$
$k \leftarrow 0$
while $k < p$ **do**
  $i \leftarrow Random(1, m)$
  $c \leftarrow h(i)$
  if $(f[c] = 0) \vee (i \notin L[c])$ **then**
    $k \leftarrow k + 1$
    $L[c] \leftarrow L[c] \cup \{i\}$
    $f[c] \leftarrow 1$
$S \leftarrow \emptyset$
for $c = 1, \ldots, r$ **do**
  $S \leftarrow S \cup L[c]$
return S

Algorithm 4: correctness and complexity

Space complexity. $O(r + p)$, because we need r flags (instead of m) and at most p elements in the lists.

(Average) time complexity. The hash function can be chosen, so that each value of $h(i)$ has the same probability. For instance: $h(i) = i \bmod r$.

Each scan takes in average $O(k/r)$ time, where k is the number of already selected elements.

The number of potential conflicts in the hash table decreases for increasing r. Hence, a suitable trade-off between time and space must be defined by suitably selecting r.

Variation. Replace lists L with binary trees.

- insertion: $O(\log(k/r))$ instead of $O(1)$;
- search: $O(\log(k/r))$ instead of $O(k/r)$.

A suitable trade-off must be found also in this case.

Algorithm 5: general description

Avoid repetitions a priori by generating random numbers in a range of decreasing width, corresponding to the subset of not-yet-selected elements of the ground set.

Problem: the *Random*(a, b) procedure requires a range with no "holes" between a and b. Therefore we need to pack the not-yet-selected elements so that their indices cover a range of integers with no missing values.

Keep an array v of m integers, one for each element of the ground set.

- Initialize $v[i] = i \; \forall i = 1, \ldots, m$.
- For each iteration $k = 1, \ldots, p$, generate a random number $i \in [k + 1, \ldots, m]$ (instead of $[1, \ldots, m]$).
- Swap $v[i]$ and $v[k]$.

Algorithm 5: pseudo-code

Algorithm 5 Selection of suitably indexed elements, to avoid repetitions.

for $i = 1, \ldots, m$ **do**
  $v[i] \leftarrow i$
for $k = 1, \ldots, p$ **do**
  $i \leftarrow Random(k, m)$
  $w \leftarrow v[i]$
  $v[i] \leftarrow v[k]$
  $S[k] \leftarrow w$
return S

Algorithm 5: correctness and complexity

Correctness. After each iteration k

- the first k positions of v contain the selected elements,
- there are no duplicates in v,
- the random extraction only involves elements not yet selected.

Space complexity (worst-case): $O(m)$.

Time complexity (worst-case).

- Initialization: $O(m)$;
- Loop: $O(p)$, because each iteration takes constant time.

Initialization is the bottleneck.

Algorithm 6: general description

Store only the elements of v for which $v[i] \neq i$.

Assume we have a list of such elements; each record in the list has two fields, π and e: they indicate an element of v where $v[\pi] = e$ and $\pi \neq e$. For all positions π not contained in the list, $v[\pi] = \pi$.

Every time a position i is generated at random in iteration k, the list is scanned and the following two tests are done.

- Test 1: if the list contains a record with $\pi = i$, let e' be the corresponding element;
- Test 2: if the list contains a record with $\pi = k$, let e'' be the corresponding element.

Four cases can occur:

- Both tests fail: select i; append (i, k) to the list.
- Test 1 succeeds, Test 2 fails: select e'; update (i, e') to (i, k).
- Test 1 fails, Test 2 succeeds: select i; update (k, e'') to (i, e'').
- Both tests succeed: select e'; update (i, e') to (i, e''); delete (k, e'').

Algorithm 6: correctness and complexity

Correctness. The output is the same as for Algorithm 5.

Space complexity: $O(p)$, instead of $O(m)$.

Time-complexity.

- Linked list: it must be scanned at each iteration.
  Each scan takes $O(p)$.
  The number of iterations is p.
  Therefore: $O(p^2)$.
- AVL tree (balanced binary tree):
  each iteration takes $O(\log p)$.
  The number of iterations is p.
  Therefore: $O(p \log p)$.

Coding the arcs of digraphs

So far, we have assumed that each edge/arc of the ground set is represented by an integer number in a range with no holes.

Digraphs. Numbering arcs, rows and columns (i.e. nodes) starting from 0 instead of 1:

$$r = \begin{cases} i(n-1)+j-1 & \text{if } i > j \\ i(n-1)+j & \text{if } i < j \end{cases}$$

$$i = r \div (n-1)$$

$$j = \begin{cases} r \bmod (n-1)+1 & \text{if } r \geq in \\ r \bmod (n-1) & \text{if } r < in \end{cases}$$

	0	1	2	3	4	5
0	x	0	1	2	3	4
1	5	x	6	7	8	9
2	10	11	x	12	13	14
3	15	16	17	x	18	19
4	20	21	22	23	x	24
5	25	26	27	28	29	x

Coding the egdes of graphs

Graphs. The ground set of undirected graphs contains only one edge for each unordered pair of distinct vertices.

Hence, only half of an adjacency matrix is needed.

To avoid wasting memory space, instead of using a triangular part of the adjacency matrix, we can use the top half of it, which is a rectangular matrix.

If n is even, use rows from 0 to $n/2 - 1$.
If n is odd, use rows from 0 to $(n + 1)/2 - 1$.

The elements of the upper triangle in the bottom part of the adjacency matrix (i.e. those with $i < j$ and $i > n/2 - 1$) are put in one-to-one correspondence with the elements under the main diagonal in the rectangular matrix (i.e. those with $i > j$ and $i \leq n/2 - 1$).

Coding the egdes of graphs

The correspondence between an index $r = 0..n(n-1)/2-1$ and an edge $[i,j]$ of the graph with $i \leq n/2-1$, is the same as for digraphs, but...

- ...when coding $[i,j]$ with $i < j$ to r, if $i > n/2-1$, then $[i,j]$ is replaced by $[i',j']$ with

  $i' = (n-1)-i \qquad j' = (n-1)-j.$

- ...when decoding r to $[i,j]$, if $i > j$ then $[i,j]$ is replaced by $[i',j']$, as above.

Coding/decoding operations still take $O(1)$ in both directions.

	0	1	2	3	4	5
0	x	0	1	2	3	4
1	5	x	6	7	8	9
2	10	11	x	12	13	14
3				x	(11)	(10)
4					x	(5)
5						x

	0	1	2	3	4
0	x	0	1	2	3
1	4	x	5	6	7
2	8	9	x	(9)	(8)
3				x	(4)
4					x