Chaofan Li <chaof@tamu.edu>

1 Random regular graph generation

1.1 Introduction to regular graphs

A *regular graph* is a graph where each vertex has the same degree. Specifically, it is called an r-regular graph if the same degree is r. For given n and r, r-regular graphs with n vertices exist if and only if

$$r < n \tag{1}$$

$$nr \text{ is even} \tag{2}$$

It is obvious for them as necessary conditions because every edge makes 2 degrees and hence if an r-regular graph has n vertices and m edges, then $nr = 2m$. For every simple graph, there are at most $n(n-1)/2$ edges, that is, the number of edges for the complete graph with n vertices. Then for every regular graph, $m = nr/2 \leqslant n(n-1)/2$, i.e., $r \leqslant n-1$

As for sufficiency, I will propose an algorithm that will definitely generate a random regular graph for any given n and r satisfying the two conditions above, and prove it, and the sufficiency can be proved as such.

1.2 Algorithm for generating random regular graph

Generating a random regular graph is not a trivial work because regular graphs account for only a small fraction of all the graphs. A common method to do this is taking advantage of the *pairing model* as noted in [1], "which starts with nr points in n groups, and choose a random pairing of the points then creates a graph with an edge from i to j if there is a pair containing points in the ith and jth groups. If no duplicated edge or loop (i.e., a pair of points in the same group) occurs, the resulting r-regular graphs occur uniformly at random." However, for graphs with n vertices, the possibility for this of generating an r-regular graph is only $e^{\frac{1-d^2}{4}}$ as $n \to \infty$[2]. The algorithm in [1] eliminates the possibility of generating a multigraph and gets a higher possibility of generating a regular graph but is still not necessarily to generate a regular graph in one pass, because some vertices may be left and cannot be paired.

1.2.1 The algorithm

input : The number of vertices n and the degree r of the random regular graph to be generated
output: A random regular graph

begin
  for *i from 0 to r* **do** `// Create r + 1 empty sets, to store vertices with degree i`
    $sets[i] \leftarrow [\,]$
  end
  for *i from 0 to n - 1* **do** `// Create n 0-degree vertices and store them in sets[0]`
    $sets[0][i] \leftarrow new\_vertex(i+1)$ `// the label of the new vertex is i + 1`
  end
  while *length(sets[r]) < n - 1* **do**
    $v1 \leftarrow pop(random\_vertex(sets[0 \text{ to } r-1]))$ `// Extract a vertex that is not in sets[r]`
    $l \leftarrow r - degree(v1)$ `// v1 needs l more degree`
    for *i from 1 to l* **do**
      $v2 \leftarrow pop(random\_min\_degree\_vertex(sets))$
      `// Extract a random vertex with min degree in sets and not connected with v1`
      $connect(v1, v2)$ `// Degree of v2 increase by 1`
      $add(v2, sets[degree(v2)])$
    end
    $add(v1, sets[D])$
  end
  return $Graph(sets[D])$ `// sets[r] will be an adjacency list, create a graph with it`
end

Algorithm 1: Random regular graph generation

This algotithm (1) will definitely generate a random regular graph, though whether it is uniformly at random is unknown. The basic idea is starting with n 0-degree vertices and $r+1$ vertex sets indexed by $0 \ldots r$ respectively

to store vertices with degree 0 to r, then iteratively randomly selecting a *pivot vertex* from all vertices with degree less than r, randomly connecting several other vertices with minimum total number of degree, to the pivot vertex making its degree increased to r, and updating the vertex sets by moving vertices with degree i to vertex sets i. For example, if we want to generate a 6-regular graph and we have selected a 1-degree vertex, then we will connect this vertex with random five vertices in $sets[0]$, if $sets[0]$ has not that many vertices, selecting from $sets[1]$ then $sets[2]$.

1.2.2 Proof of its correctness

To prove that algorithm (1) will definitely terminates with n vertices in $sets[r]$ (i.e., n r-degree vertices) for any given n and r s.t. $n > r$ and nr is even, we can firstly prove two properties of $sets[0 \dots r]$.

1. Vertices in $sets[0 \dots r-1]$ cannot be connected.

   Proof. If two of them are connected, one must be a pivot vertex, and then after the **for** loop in line 11, the pivot vertex must be in $sets[D]$, which is against that two of them are both in $sets[0 \dots r-1]$ □

2. At most two of $sets[0 \dots r-1]$ are non-empty, and they must have consecutive indices, like $sets[0]$ and $sets[1]$, $sets[4]$ and $sets[5]$.

   Proof. This property can be proved inductively. We know that initially only $sets[0]$ is non-empty. Suppose there are at most two non-empty sets among all $sets$ before the **while** loop in line 8, after one this loop, only three possible cases occur:

   (a) Except the *pivot vertex*, part of the vertices in the smaller indexed set, say $sets[i]$, are moved to $sets[i+1]$ and the vertices in $sets[i+1]$ are all still there.

   (b) Except the *pivot vertex*, all of the vertices in $sets[i]$ are moved to $sets[i+1]$ and vertices in $sets[i+1]$ are all still there. $sets[i]$ becomes empty and only $sets[i+1]$ will be non-empty.

   (c) Except the *pivot vertex*, all of the vertices in $sets[i]$ are moved to $sets[i+1]$ and all or part of vertices in $sets[i+1]$ are moved to $sets[i+2]$. $sets[i]$ becomes empty and $sets[i+1]$ and $sets[i+2]$ becomes non-empty.

   So this property always holds because at the beginning it holds. □

After proving the two properties above, we can prove the correctness of the whole algorithm inductively. Firstly, note that at a certain stage, if only $sets[k]$ is non-empty and $sets[0 \dots k-1]$ are all empty, then by property 2, only $sets[k+1]$ may be non-empty among $sets[k+1 \dots r-1]$. Suppose there are p vertices in $sets[k]$ and q vertices in $sets[k+1]$, then we have some necessary conditions if the algorithm is correct:

$$(p+q)(p+q-1) \geqslant p(r-k) + q(r-k-1) \tag{3}$$

$$p(r-k) + q(r-k-1) \text{ is even} \tag{4}$$

The left part of (3) is simply 2 multipled by the number of edges that can be added if every pair of vertices in $sets[k]$ and $sets[k+1]$ are connected. By property 1, all of the vertices in $sets[0 \dots r-1]$ are not connected with each other. So the left part is exactly the maximum number of degree that can be added. The right part of (3) is just the number of degree lacking to be r-regular graph. Also this number should be even because each edge generates 2 degree and so we have necessary condition (4).

Note that if $k = 0$ and $q = 0$, the necessary conditions (3) and (4) are just the necessary conditions (1) and (2), i.e., the necessary conditions for the $sets[0]$ initialized in line 5 of algorithm (1) to be a candidate as a regular graph. So the necessary conditions hold before the first **while** loop in line 8.

What we can prove is that if the necessary conditions hold, then the **while** loop must be able to be executed and after that the necessary conditions still hold. If we can prove that, then after at most $nr/2$ times of the **while** loop in line 8, the algorithm must terminate with $p = 0$, $q = 0$ and $k = r - 1$.

In each **while** loop, we will select a *pivot vertex* and pair it with at least one other vertices. So, $p + q \geqslant 2$. This pivot vertex can be chosen in $sets[k]$ or $sets[k+1]$, after which there are $p+q-1$ vertices left and $r-k$ or $r-(k+1)$

more vertices need to be selected. So $p+q-1 \geqslant r-k$. The sufficient and necessary conditions for the **while** loop to be able to execute is

$$p+q-1 \geqslant r-k \tag{5}$$
$$p+q \geqslant 2 \tag{6}$$

(3)/$(p+q)$ is $p+q-1 \geqslant r-k-q/(q+p)$, i.e., $p+q-1 \geqslant r-k$, and because $q/(p+q) < 1$, If (3) hold, (5) must hold. As for (6), if it doesn't hold, the program should terminate as intended. So if the necessary conditions (3) and (4) hold, the **while** loop must be able to be executed. After that loop there are still $(p+q-1)(p+q-2)$ vertices left and the right part of (3) will decrease at least $2(r-k)-2$, so we need to prove that $(p+q-1)(p+q-2) \geqslant p(r-k)+q(r-k-1)-2(r-k)+2$. Because for $q \geqslant 2$

$$\begin{aligned}(p+q-1)(p+q-2) &\geqslant (r-k)(p+q-2) \\ &= (r-k)(p-2)+q(r-k) \\ &\geqslant (r-k)(p-2)+q(r-k)+2-q \text{ if } q \geqslant 2 \\ &= p(r-k)+q(r-k-1)-2(r-k)+2\end{aligned} \tag{7}$$

As shown above, for $q \geqslant 2$, the condition holds. For $q=0$, the right part will decrease at least $2(r-k)$ rather than $2(r-k)-2$, it still holds. For $q=1$, (7) should be "strictly greater than", the condition (3) holds too. A special case is that two vertices are more than one vertices are moved to $sets[r]$ in the **while** loop. In that case, after the loop, all vertices not in $sets[r]$ will be in $sets[r-1]$, which is easy to prove that the condition still holds. As for the condition (4), it must hold after one **while** loop, because only even number of degree will be deducted from the left part of (4), i.e., the right part of (3).

So the correctness has been proved inductively by proving that if the necessary conditions (3) and (4) hold then the **while** loop in line line 8 must be able to be executed with no exceptions and after that loop the necessary conditions still hold until there is no vertices in $sets[0 \ldots r-1]$.

1.2.3 Complexity and implementation

The time complexity of this algorithm is $O(nd)$ because we will generate nd random numbers. The space complexity is $O(nd)$ too because storing a graph with adjacency list will cost nd units of space, and the $sets[0 \ldots r][0 \ldots n-1]$, if implemented carefully, cost only three fixed-size array of length n because as proved above only two sets among $sets[0 \ldots r-1]$ will be non-empty.

I have implemented this algorithm in Python and C respectively. In Python I used $r+1$ $sets$ and in C I implemented it with only three fixed-size array. Both works well and fast enough even for the 1000-regular graph with 5000 vertices (less than 1 minute for Python implementation and less than 5 seconds for C implementation), i.e., the graph "with each vertex having edges going to about 20% of the other vertices". The random integer generation procedure I used in the implementation is the `random.randrange` in Python and the `rand()` in `<stdlib.h>`.

References

[1] Angelika Steger and Nicholas C Wormald. Generating random regular graphs quickly. *Combinatorics Probability and Computing*, 8(4):377–396, 1999.

[2] Nicholas C Wormald. Models of random regular graphs. *London Mathematical Society Lecture Note Series*, pages 239–298, 1999.