UDC 004.02+004.021+519.17

GENERATING UNDIRECTED GRAPHS WITH A FIXED VERTEX DEGREE FOR TESTING PURPOSES

V. Chernyakhivskyy

Ivan Franko National University of Lviv,
Universytets'ka Str., 1, Lviv, 79000, Ukraine,
e-mail: volodymyr.chernyakhivskyy@lnu.edu.ua

The analysis of problem-solving tasks and construction of graphs have been leading to the development of new algorithms and software implementations. To verify the algorithm, it is necessary to make test data in such a way, that all values for the test are predetermined. The values for testing algorithms via graphs are: the number of vertices in the graph; the vertex degree; connectivity of the graph; permissibility of multiple edges and loops; the weight of the graph edges. If the values are known for the test graph then we can calculate the expected results of the designed algorithm and match them with the actual ones in a well-posed manner.

The algorithm and some elements of software implementation of generating finite undirected graphs, with such values as the number of vertices n and the degree of the vertices $k = const$, which correspond to correctness of additional conditional statements, are presented in the article.

The general scheme of the algorithm is as follows. Create two lists of vertices L1 and L2. The list L1 is initially empty; in the list L2 we have all vertices of the graph from 1 to n. We take the first vertex in L2, we build all k edges to it and transfer it to the list L1. Repeat the same for each next vertex of the L2 list, except the last one. The last vertex L2 is just transferred to L1. The algorithm immediately builds the adjacency matrix by modeling list operations with L1 and L2.

According to the algorithm, the adjacency matrix will always be the same for fixed n and k when the algorithm is repeated. To obtain different matrices, the rule of graph isomorphism is used: perform the permutations of the rows and columns of the resulting matrix.

Key words: graph, test, algorithm, isomorphism, Python.

1. Introduction

The analysis of problem-solving tasks and construction of graphs have been leading to the development of new algorithms and software implementations. To verify the algorithm, it is necessary to make test data in such a way, that all values for the test are predetermined. The values for testing algorithms via graphs are: the number of vertices in the graph; the vertex degree; connectivity of the graph; permissibility of multiple edges and loops; the weight of the graph edges. If the values are known for the test graph then we can calculate the expected results of the designed algorithm and match them with the actual ones in a well-posed manner.

The algorithm and some elements of software implementation of generating finite undirected graphs, with such values as the number of vertices n and the degree of the vertices $k = const$, which correspond to correctness of additional conditional statements, are presented in the article.

2. The overview and task formulation

There are different approaches to the problem of constructing graphs and their applications published in the literature. Thus, in paper [1], an algorithm for generating nets is presented; the geometric spatial graph is the basis. The nets generator is used to predict the crystal structure and design at the design stage. In paper [2] the problem of generating all triangulated planar graphs is considered. The algorithm for generating a three-connected planar graph of a given number of vertices n is described. In paper [3], a defined algorithm for a given arbitrary graph G, and a positive integer k, can find all the subgraphs of order k. Such results can be used for the structure of primary partial solutions of designing and testing graph models in various subject areas. In paper [4], an algorithm for generating random graphs is presented. The parameters are: the number of vertices d and the total number of edges of the graph t. Such an algorithm provides a general method of generating, which has only two specified values for the characteristics of the graph itself.

Similar methods and algorithms for the modeling of applied tasks via graphs, including testing, remain relevant. The range of applying the results of modeling has been expanding. Algorithms are improving by reducing the computational complexity.

The aim of the present research is to obtain a method for designing test graphs for proof verification of various methods and algorithms for problem-solving tasks via graphs. The test graphs will be used in computational experiments, so in fact, it is necessary to have a numerical representation of the graph, rather than a graph as a geometric image. It is assumed that the graph is represented with adjacency matrix. Thus, the aim can be formulated in the following way: to obtain the method for building an adjacency matrix with specified values of the graph structure.

Input:

n - number of vertices in the graph

$k = const$ - the same degrees of all vertices

$V_i \; in \; set(1, n)$ - vertices are numbered from 1 to n

$path(V_i, V_j) = 1, i \neq j$ - connected graph, there is a path between each pair of vertices

$d(V_t, V_p) = d(V_p, V_t)$ - all edges are of equal weight or unweighted graph

$d(V_p, V_p) = -1$ - the graph has no loops

Output:

$graph \; G_n = \{(V_t, V_p)\}, G_n \leftrightarrow (input \; def \; property), G_n = [V_{tp}]$ - obtain all the edges of the constructed graph represented by the adjacency matrix.

Note, in many cases of the computational experiments a planar graph is optional, for example, testing search tasks via graphs.

3. Algorithm for graph generation

The general scheme of the algorithm is as follows. Create two lists of vertices L1 and L2. The list L1 is initially empty; in the list L2 we have all vertices of the graph from 1 to n (from 0 to n-1 according to the rules of the algorithmic language). We take the first vertex in L2, we build all k edges to it and transfer it to the list L1. Repeat the same for each next vertex of the L2 list, except the last one. The last vertex L2 is just transferred to L1. As a result, there will be all vertices of the graph with definite edges $[(V_i, V_{i1}), (V_i, V_{i2}), ..., (V_i, V_{ik})]$ in the list L1.

Now we define the complete and accurate algorithm for generation. The elements

of the algorithm are written in the notation of Python [5]. The algorithm immediately builds the adjacency matrix by modeling list operations with L1 and L2.

Step 1. Determine the values n, k of the structure of the graph, the initial zero adjacency matrix *mta* and the fixed vector *adj* of the degree of vertices:

```
n = int(input('Number of vertices ='))
if n<3: raise RuntimeError("n=" + str(n) + "\ Need n>=3")
k = int(input('Specified degree of vertices ='))
if k<2:
  raise RuntimeError("The degree of vertices must be not less than 2")
if k>=n: raise RuntimeError("The degree of vertices must be less than
            the number of vertices")
mta = [ ]  # basic adjacency matrix
for i in range (n): mta.append([0] * n) # zero initialization
isomta1 = [ ]  # adjacency matrix, isomorphism, rows are permuted
for i in range(n): isomta1.append([0] * n) # zero initialization
isomta2 = [ ]  # adjacency matrix, isomorphism, columns are permuted
for i in range(n): isomta2.append([0] *n ) # zero initialization
adj = [0] * n  # vector of degree of vertices:\ zero initialization
```

Step 2. Define the function of *nextfirstmin()* search for the vertex with the least number of constructed edges with additional search terms:

```
# Find the first vertex after fm with the least number of edges
#   at the time of the search
def nextfirstmin(fm):
  temp = [ n+1 if i<=fm else adj[i] for i in range(n) ]
  return(temp.index(min(temp))) # return the number of the vertex
```

Let's clarify the structure of the algorithm. If it is necessary to construct edges to the vertex V, we search the pair points among the vertices V+1, V+2, V+3 and then in the ascending order of the number of vertices. The very first vertex with the minimum number of edges at the time of the search is the result of the search. That is, we perform a "forward search" strictly in the ascending order of number. We construct all necessary edges for V. Vertex of V is definitely defined by its edges. If this is done, then, obviously, the next vertex V+1 may have a pair points only among the vertices V+2, V+3, V+4 and so on.

In case all edges of the vertices are built up to $fm-1$ inclusively, then the search for fm next pair points must be performed starting with $fm+1$. The *temp* list must demonstrate the complete list of the number of current links for each vertex for each step of the algorithm. In order to obtain the index of the next pair points correctly, the previously constructed vertices till fm incl, we determine the number by $n+1$, which is greater than n (according to the input it is also greater than k). Thus, the result of the search is guaranteed only among the vertices $fm+1$, $fm+2$, $fm+3$ and so on, because the degree s of each vertex at each step of the algorithm $s \le k < n$.

In Python it is presented with two lines of function *nextfirstmin()*.

In this interpretation, the part of the list of *temp* within $[0-fm]$ of the Python indexes is the list L1, and the part of the *temp* within $[(fm+1)-(n-1)]$ is the L2 list.

Step 3. Define the function *rib()* of connection of edges of the pair of specified vertices:

```
def rib(a,b): # connect with the edge of the vertex a, b
  # control of the multiplicity of edges
  if mta[a][b]==1: print('Error-multiple edge !!', a, b)
  mta[a][b]=1; mta[b][a]=1  # undirected graph
  # the current state of the vector of degree of the vertices
  adj[a]+=1; adj[b]+=1
```

The function fixes the undirected edge in the adjacency matrix with the pair $mta[a][b]$ and $mta[b][a]$ and performs the control of the multiplicity of the edges $mta[a][b] == 1$ for debugging the algorithm. The current state of the corresponding elements of the vector $adj[]$ of the degrees of vertices is increased by 1.

Step 4. Detect the basic cycle of the structure of the edges of all vertices:

```
for a in range(n-1):
  # we look through all vertices in turn [0-(n-2)],
  #   except the last one
  # control: how many edges are needed to be constructed
  print(k-adj[a],end='')
  while adj[a]<k:  # construct all 'k' edges to the vertex 'a'
    b=nextfirstmin(a)  # next the first one after 'a'
    rib(a,b)  # connect with the edge
```

Scanning the vertices for the building of the edges is performed in ascending order of numbers of vertices *for a in range*$(n-1)$. The vertex with least number of edges $nextfirstmin(a)$ is always the pair point for the adjacent edge for a at each step of the search. At the beginning, each vertex has zero edges. That is why the very first vertex without edges will be connected by the edge of the one already connected vertex. In this way, we guarantee the connectivity of the whole graph after the construction of all edges.

Note, by the run of algorithm the last vertex $n-1$ obtains automatically all needed edges with accuracy (0;-1) depending on the parity of the total number of edges of the graph.

An example of iteration of step 4 for the first three vertices at n=8, k=4 is shown in *Table 1,a* and (iteration 1), *Table 1,b* (iteration 2), *Table 1,c* in (iteration 3). The result of the complete structure for all vertices is presented in *Table 1,d*.

Table 1

Example of iteration at $n = 8, k = 4$

0 1 1 1 1 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

a) iter.1

0 1 1 1 1 0 0 0
1 0 0 0 0 1 1 1
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0

b) iter.2

0 1 1 1 1 0 0 0
1 0 0 0 0 1 1 1
1 0 0 1 1 1 0 0
1 0 1 0 0 0 0 0
1 0 1 0 0 0 0 0
0 1 1 0 0 0 0 0
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0

c) iter.3

0 1 1 1 1 0 0 0
1 0 0 0 0 1 1 1
1 0 0 1 1 1 0 0
1 0 1 0 0 0 1 1
1 0 1 0 0 1 1 0
0 1 1 0 1 0 0 1
0 1 0 1 1 0 0 1
0 1 0 1 0 1 1 0

d) iter.n-1

Taking into account the already constructed edges in the previous iterations, the control number of edges that must be construct for each vertex is carried out:

```
print(k-adj[a],end='')
```

As a result of the control at $n = 8, k = 4$ we get (4 3 3 2 2 1 1).

Step 5. According to the algorithm, the adjacency matrix will always be the same for fixed n and k when the algorithm is repeated. To obtain different matrices, the rule of graph isomorphism is used: perform the permutations of the rows and columns of the resulting matrix. To calculate the vector of permutation *rearr*[] we use the standard function *randint*() the Python library [5] to generate random numbers:

```
import random
rearr=[ ]; k=0; c=0  # c - control of the generator (will be ~2-3 times
                           greater than n)
while k<n-1:
  newpos = random.randint(0,n-1); c+=1
  if newpos not in rearr: rearr.append(newpos); k+=1
# add the last element to the vector "manually"
rearr += list(set(range(0,n)) - set(rearr))
```

New positions of permutation are generated for all rows (columns) except the last one. We calculate the last position as the difference of sets $\{0, 1, ..., (n-1)\} - \{rearr\}$. When a new position is added, the repetition is under control (*newpos not in rearr*). For example, repeated run of the algorithm for $n = 8, k = 4$ can provide such vectors of permutation and meter measurements c the function call *randint*():

```
[1, 2, 3, 5, 4, 6, 0, 7] ,   c= 29
[4, 1, 5, 3, 0, 6, 2, 7] ,   c= 9
[0, 3, 1, 6, 4, 7, 5, 2] ,   c= 11
[2, 4, 3, 6, 5, 7, 1, 0] ,   c= 14
```

In accordance with the achieved vector of permutation the program code of the permutations of the rows and columns of the matrix *mta*[ ][ ] in an arbitrary manner is presented. For example:

```
# step 1: rearrange the rows
for i in range(0,n): isomta1[rearr[i]] = mta[i][:]
# step 2: rearrange the columns of the isomta1 matrix obtained in step 1
for col in range(n):
  for row in range(n):
    isomta2[row][rearr[col]] = isomta1[row][col]
```

Step 6. Save the constructed adjacency matrix as the text file, taking into account the permutations of the rows and columns of step 5. There are various methods of formatting as a text file, depending on the usage of the adjacency matrix. Python provides a lot of formatting capabilities.

4. Some results of calculating experiments

Examples of calculations of adjacency matrices of graphs are presented in *Table 2,a*, in *Table 2,b*, and *Table 2,c*. The values of the structure of the graph are indicated in the signature tables.

Table 2

Examples of calculations of adjacency matrices

0 0 0 0 1 1 0 0 1 0	0 1 0 0 1 1 0 1 1 1 1	0 0 1 0 0 0 1 1 0 0 0 1 1 0
0 0 1 0 0 0 0 1 1 0	1 0 1 0 1 0 0 1 1 1 1	0 0 1 1 1 0 0 0 0 1 0 0 1 0
0 1 0 1 0 0 1 0 0 0	0 1 0 1 1 1 1 0 0 1 1	1 1 0 1 0 0 0 1 1 0 0 0 0 0
0 0 1 0 0 0 1 0 0 1	0 0 1 0 1 1 1 1 1 0 1	0 1 1 0 0 1 0 0 0 1 1 0 0 0
1 0 0 0 0 0 0 0 1 1	1 1 1 1 0 0 0 1 0 1 1	0 1 0 0 0 1 0 0 1 0 0 1 0 1
1 0 0 0 0 0 1 1 0 0	1 0 1 1 0 0 1 1 1 0 1	0 0 0 1 1 0 0 0 1 1 0 0 1 0
0 0 1 1 0 1 0 0 0 0	0 0 1 1 0 1 0 1 1 1 0	1 0 0 0 0 0 0 0 0 0 1 1 1 1
0 1 0 0 0 1 0 0 0 1	1 1 0 1 1 1 1 0 1 0 0	1 0 1 0 0 0 0 0 1 1 0 0 0 1
1 1 0 0 1 0 0 0 0 0	1 1 0 1 0 1 1 1 0 1 0	0 0 1 0 1 1 0 1 0 0 1 0 0 0
0 0 0 1 1 0 0 1 0 0	1 1 1 0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1 0 0 1 0 0 0
	1 1 1 1 1 1 0 0 0 1 0	0 0 0 1 0 0 1 0 1 1 0 1 0 0
		1 0 0 0 1 0 1 0 0 0 1 0 0 1
		1 1 0 0 0 1 1 0 0 0 0 0 0 1
		0 0 0 0 1 0 1 1 0 0 0 1 1 0
a) $n = 10, k = 3$	*b*) $n = 11, k = 7$	*c*) $n = 14, k = 5$

5. Conclusion

The aim of the paper is to determine the general algorithm and elements of software implementation of the structure of the test graphs defined by the adjacency matrix. Test graphs or adjacency matrices can be used for tasks that are modeled via graphs. The obtained test matrices can be saved in a binary file without converting them to a text form using the Python language tools.

References

1. *McColm G.* Generating Geometric Graphs Using Automorphisms / G. McColm // Journal of Graph Algorithms and Applications. – 2012. – Vol. 16, No. 2. – P. 507–541.
2. *Parvez M. T.* Generating All Triangulations of Plane Graphs / M. T. Parvez, Md. S. Rahman, S. Nakano // Journal of Graph Algorithms and Applications. – 2011. – Vol. 15, No. 3. – P. 457–482.
3. *Elbassioni K.* A Polynomial Delay Algorithm for Generating Connected Induced Subgraphs of a Given Cardinality / K. Elbassioni // Journal of Graph Algorithms and Applications. – 2015. – Vol. 19, No. 1. – P. 273–280.
4. *Bayati M.* A Sequential Algorithm for Generating Random Graphs / M. Bayati, J. H. Kim, A. Saberi // Discrete Applied Mathematics. – 2010. – Vol. 58, Issue 4. – P. 860–910.
5. *Python Software Foundation [US].* Python 3.7.3 documentation [Electronic resource]. Available from: https://docs.python.org/3/

Article: received 05.08.2019
revised 20.11.2019
printing adoption 20.11.2019

ГЕНЕРУВАННЯ ТЕСТОВИХ НЕОРІЄНТОВАНИХ ГРАФІВ ФІКСОВАНОГО СТЕПЕНЯ ВЕРШИН

В. Черняхівський

Львівський національний університет імені Івана Франка,
вул. Університеська, 1, Львів, 79000,
e-mail: volodymyr.chernyakhivskyy@lnu.edu.ua

Розв'язування задач аналізу і конструювання графів приводить до будови нових алгоритмів і програмної реалізації. Для перевірки алгоритму потрібно будувати тестові дані такого виду, щоб мати наперед задані значення параметрів тестів. Для випадку тестування алгоритмів на графах такими параметрами є: кількість вершин графа; степені вершин; зв'язність графа; допустимість кратних ребер і петель; вага ребер. Якщо тестовий граф має відомі значення параметрів, тоді можна обчислювати теоретичні результати побудованого алгоритму і коректно зіставляти з реально отриманими.

Викладено алгоритм і елементи програмної реалізації задачі генерування скінчених неорієнтованих графів, параметрами яких є кількість вершин n і степені вершин $k = const$ і які відповідають додатковим умовам коректності.

Загальна схема алгоритму така. Створюємо два списки вершин L1 і L2. Список L1 – початково пустий, в списку L2 – всі вершини графа від 1 до n. Беремо найпершу за номером вершину в L2, будуємо до неї всі k ребер і переносимо в список L1. Повторюємо описану операцію для кожної наступної вершини списку L2, крім останньої. Останню вершину L2 просто переносимо в L1. Алгоритм зразу будує матрицю суміжності, моделюючи операції зі списками L1 і L2.

Відповідно до алгоритму матриця суміжності завжди буде однаковою для фіксованих n, k при повторному виконанні алгоритму. Щоб отримати різні матриці, використаємо правило ізоморфізму графа: виконаємо перестановки рядків і стовпців отриманої матриці.

Ключові слова: граф, тест, алгоритм, ізоморфізм, Python.