FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION

OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

on the practical task No. 5

“Algorithms on graphs. Introduction to graphs and basic algorithms on graphs”

Performed by

Victor Voronin

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg

2020

# Goal

The use of different representations of graphs and basic algorithms on graphs.

# Formulation of the problem

1. Generate a random adjacency matrix for a simple undirected unweighted graph with 100 vertices and 200 edges (note that the matrix should be symmetric and contain only 0s and 1s as elements). Transfer the matrix into an adjacency list. Visualize the graph and print several rows of the adjacency matrix and the adjacency list. Which purposes is each representation more convenient for?
2. Use Depth-first search to find connected components of the graph and Breadth-first search to find a shortest path between two random vertices. Analyze the results obtained.
3. Describe the data structures and design techniques used within the algorithms.

# Brief theoretical part

To solve the task, it is supposed to use the following standard libraries:

* libraries network and matplotlib.pyplot for visualization of the graph
* library random to generate random adjective matrix
* sys.setrecursionlimit(1500) to gain the recursion limit
* library Nympy for work with array

**Depth first search (DFS)**

Initially all vertices are marked unvisited (false). The DFS algorithm starts at a vertex u in the graph. By starting at vertex u it considers the edges from u to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex u. If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That means the new vertex becomes the current vertex. Follow this process until we reach the dead-end. At this point start backtracking.

An execution of depth-first search can be used to analyze the structure of a graph, based upon the way in which edges are explored during the traversal.

Proposition: Let G be an undirected graph on which a DFS traversal starting at a vertex s has been performed. Then the traversal visits all vertices in the connected component of s, and the discovery edges form a spanning tree of the connected component of s.

Justification: Suppose there is at least one vertex w in s’s connected component not visited, and let v be the first unvisited vertex on some path from s to w (we may have v = w). Since v is the first unvisited vertex on this path, it has a neighbor u that was visited. But when we visited u, we must have considered the edge (u,v); hence, it cannot be correct that v is unvisited. Therefore, there are no unvisited vertices in s’s connected component.

**Breadth first search**

Initially, BFS starts at a given vertex, which is at level 0. In the first stage it visits all vertices at level 1 (that means, vertices whose distance is 1 from the start vertex of the graph). In the second stage, it visits all vertices at the second level. These new vertices are the ones which are adjacent to level 1 vertices. BFS continues this process until all the levels of the graph are completed. Generally, queue data structure is used for storing the vertices of a level.

# Results

1. Method get\_AdjMatrix(n, qty\_edge\_required) was implemented to get random adjacency matrix size of n x n. Method convert(adjMatrix) was enacted to convert the adjacency matrix to appropriate adjacency list. For the code see Appendix 1.
2. The first ten columns of the adjacency matrix and some part of the adjacency list are presented on figure 1. Graph visualization is shown on figure 2.

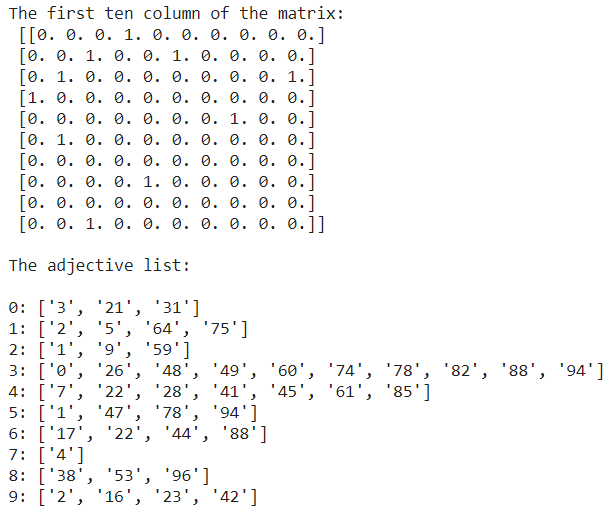


Figure 1 Some parts of the adjective matrix and list

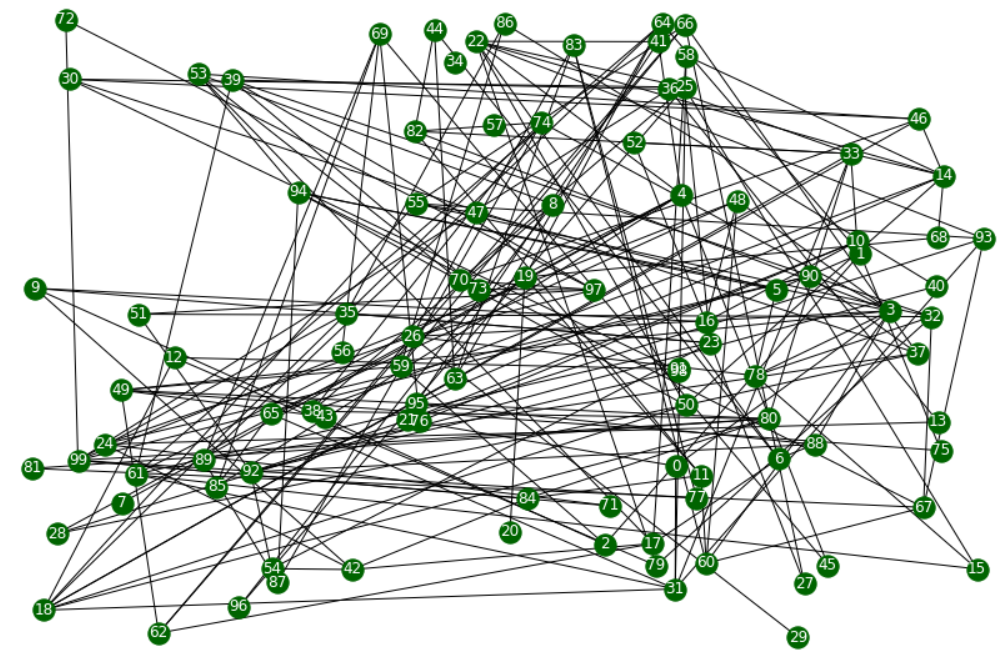


Figure 2 Graph visualization

1. Graph connectivity analysis was carried out by recursion method test\_connected(qnty\_nodes, graph, node) and bfs. This method uses depth first search to analyze connectivity of the graph. See code in Appendix 2. Each time when the method found not-visited node, procedure of DFS is executed again with this node as start point.

Non-recursion method has been enacted also. It uses stack which However, if count of edges e were more than 50 result can not be obtained with this variation of method. Its code can be found in Appendix 3.

Path of DFS:

2->1->5->47->55->18->12->79->22->4->7->28->50->14->33->10->16->9->23->25->15->61->31->0->3->26->24->13->36->39->32->51->92->21->93->67->60->11->19->20->64->63->44->6->17->42->54->84->80->38->8->53->46->30->37->66->27->59->49->43->62->75->86->35->99->48->65->85->69->45->56->76->89->87->74->82->94->71->72->97->70->73->77->83->90->40->98->96->81->58->88->95->41->68->78->29->52->91

Graph is not connected

In terms of its running time, depth-first search (DFS) is an efficient method for traversing a graph. Note that DFS is called at most once on each vertex (since it gets marked as visited), and therefore every edge is examined at most twice for an undirected graph, once from each of its end vertices, and at most once in a directed graph, from its origin vertex. If we let ns ≤ n be the number of vertices reachable from a vertex s, and ms ≤ m be the number of incident edges to those vertices, a DFS starting at s runs in O(ns+ms) time, if the graph is represented by a data structure such that creating and iterating through the incident edges(v) takes O(deg(v)) time, and get the edge from vertex u to vertex v, if one exists takes O(1) time. The adjacency list structure is one such structure, but the adjacency matrix structure is not. In summary, time performance of DFS depends on structure which was used for presentation of the graph.

1. The shortest path between two nodes in the graph (fig. 2) can be found by breadth first search (BFS). To do that, data structure such as queue was used. Starting with the source vertex in the queue, it repeatedly removes the vertex from the front of the queue and insert any of its unvisited neighbors to the back of the queue. For code see Appendix 4

Shortest path between 2 and 8 is ['2', '1', '5', '94', '53', '8']

Time complexity of BFS is also depends on realization of presentation of the graph: O(V + E), if we use adjacency lists for representing the graphs, and O(V2) for adjacency matrix representation.

1. Operation of the algorithm in various cases. See figure 3 and 4

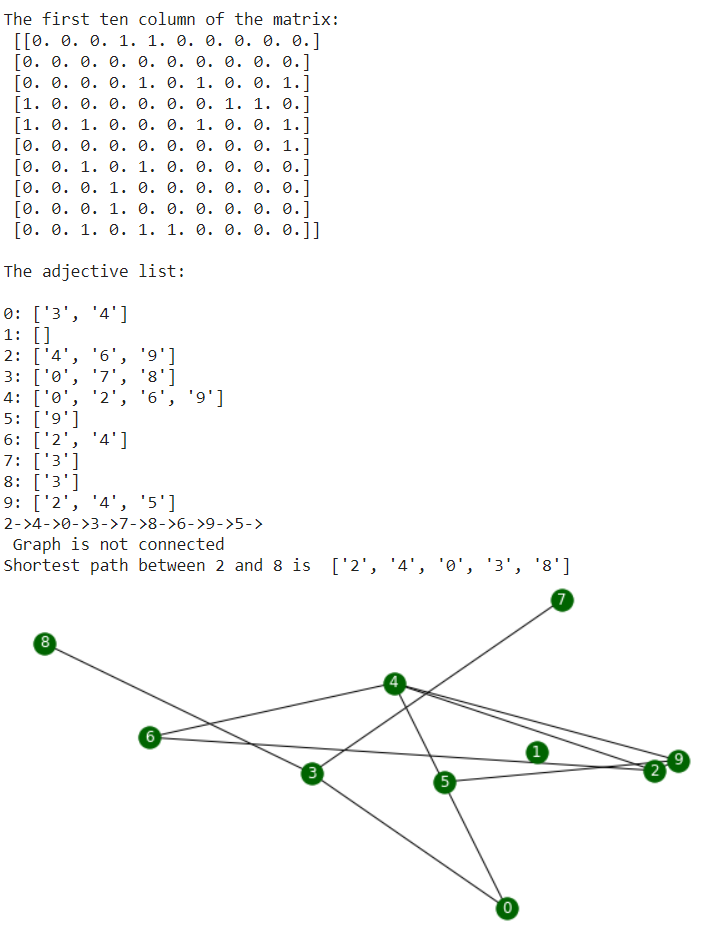


Figure 3 Unconnected graph

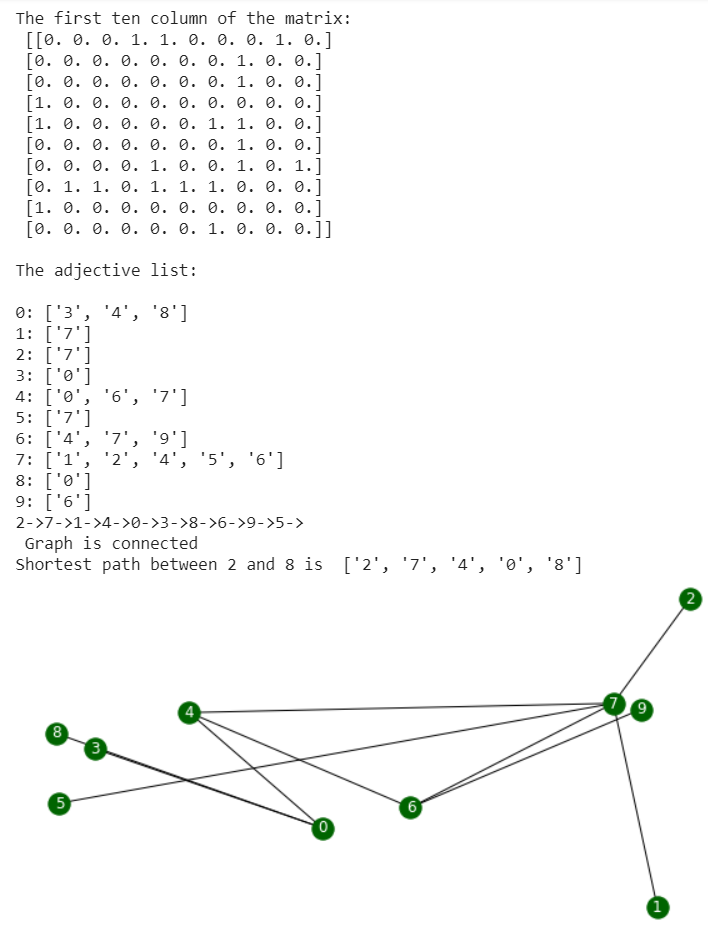
­

Figure 4 Connected graph

The most significant advantage of an adjacency matrix is that any edge (u,v) can be accessed in worst-case O(1) time. However, several operations are less efficient with an adjacency matrix. For example, to find the edges incident to vertex v, we must presumably examine all n entries in the row associated with v; an adjacency list can locate those edges in optimal O(deg(v)) time. Adding or removing vertices from a graph is problematic, as the matrix must be resized. Furthermore, the O(n2) space usage of an adjacency matrix is typically far worse than the O(n + m) space required of the other representations. Although, in the worst case, the number of edges in a dense graph will be proportional to n2, most real-world graphs are sparse. In such cases, use of an adjacency matrix is inefficient. However, if a graph is dense, the constants of proportionality of an adjacency matrix can be smaller than that of an adjacency list.

# Conclusions

During the execution of the task, simple undirected unweighted random graph was generated and then depth first and breath first searches were enacted to find connection or the shortest path between two nodes of the graph. The results obtained were analysed.

Appendix 1

def get\_AdjMatrix(n, qty\_edge\_required):

  adjMatrix = numpy.zeros((n, n))

  qty\_edge = 0

  while qty\_edge < qty\_edge\_required:

    i = numpy.random.randint(0, n)

    j = numpy.random.randint(0, n)

    if i != j and adjMatrix[i][j] == 0:

      adjMatrix[i][j] = 1

      adjMatrix[j][i] = 1

      qty\_edge += 1

  return adjMatrix

def convert(adjMatrix):

    #adjList = {}

    adjList = defaultdict(list)

    for i in range(len(adjMatrix)):

      for j in range(len(adjMatrix[i])):

        if adjMatrix[i][j]== 1:

            temp = "{0}".format(j)

            adjList["{0}".format(i)].append(temp)

    return adjList

Appendix 2

def dfs(visited, graph, node):

    if node not in visited:

      print(node, end = "->")

      visited.append(node)

      for neighbour in graph[node]:

        dfs1(visited, graph, neighbour)

    return visited

def test\_connected1(visited, qnty\_nodes):

  if len(visited) == qnty\_nodes:

    print("\n","Graph is connected")

  else:

    print("\n","Graph is not connected")

Appendix 3

def test\_connected(qnty\_nodes, graph, node):

    iteration = 1

    visited = [node]

    stack = [node]

    while stack:

        iteration += 1

        node = stack[-1]

        if node not in visited:

            visited.extend(node)

        remove\_from\_stack = True

        for next in graph[node]:

            if next not in visited:

                stack.extend(next)

                remove\_from\_stack = False

                break

        if remove\_from\_stack:

            stack.pop()

    if len(visited) == qnty\_nodes:

      result = "Graph is connected"

    else:

      result = "Graph is not connected"

    return result, visited, iteration

Appendix 4

def short\_path(graph, start, end):

  # maintain a queue of paths

  queue = []

  # push the first path into the queue

  queue.append([start])

  try:

    while queue:

        # get the first path from the queue

        path = queue.pop(0)

        # get the last node from the path

        node = path[-1]

        # path found

        if node == end:

            return path

        # enumerate all adjacent nodes, construct a new path and push it into the queue

        for adjacent in graph.get(node, []):

            new\_path = list(path)

            new\_path.append(adjacent)

            queue.append(new\_path)

  except:

   result = ("not exist")

   return result

Appendix 5

# initial data

n = 100

qty\_edge\_required = 200

# Adjacency matrix and print of it

adjMatrix = get\_AdjMatrix(n, qty\_edge\_required)

print("The first ten column of the matrix:","\n",adjMatrix[:10, :10],"\n")

# Adjacency list and print of it

adjList = convert(adjMatrix)

print("The adjective list:","\n")

for i in range(10):

  temp = "{}".format(i)

  print ("{0}: {1}".format(i, adjList[temp]))

# Analyze of graph

start = "2"

end = "8"

# DFS

visited = []

test\_connected(dfs(visited, adjList, start), n)

# BFS

print( "Shortest path between {0} and {1} is ".format(start, end),short\_path(adjList, start, end))

#Visualisation

G = nx.Graph(adjMatrix) # create graph iaw the adjacency matrix

pos = nx.random\_layout(G)

plt.figure(3,figsize=(15,10))

labels = {} # to create labels of graph nodes

for k in range(n):

  labels[k] = "{0}".format(k)

nx.draw\_networkx\_labels(G, pos, labels, font\_size=12, font\_color="white")

nx.draw(G, pos, node\_size = 300, node\_color = "darkgreen", edge\_color = "black")