FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION

OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

on the practical task No. 8

“Practical analysis of advanced algorithms”

Performed by

Victor Voronin

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg

2020

# Goal

Practical analysis of advanced algorithms.

# Formulation of the problem

1. Choose two algorithms (interesting to you and not considered in the course) from the above-mentioned book sections.
2. Analyze the chosen algorithms in terms of time and space complexity, design technique used, etc. Implement the algorithms and produce several experiments. Analyze the results.

# 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 sample random node
* library Nympy for work with array
* library timeit to measurement of execution time of algorithms
* library heapq for implementation of Prim’s algorithm.

**Prim-Jarnık Algorithm**

Prim-Jarnık Algorithm allows to find minimum spanning tree on a weight graph starting from some “root” vertex s. The algorithm begins with some vertex s, defining the initial “cloud” of vertices C. Then, in each iteration, the algorithm choose a minimum-weight edge e = (u,v), connecting a vertex u in the cloud C to a vertex v outside of C. The vertex v is then brought into the cloud C and the process is repeated until a spanning tree is formed, in other words, until all vertices have been visited.

The implementation issues for the Prim-Jarnık algorithm are similar to those for Dijkstra’s algorithm, relying on an adaptable priority queue Q. It initially perform n insertions into Q, later perform n extract-min operations, and may update a total of m priorities as part of the algorithm. Those steps are the primary contributions to the overall running time. With a heap-based priority queue, each operation runs in O(logn) time, and the overall time for the algorithm is O((n + m)logn), which is O(mlogn) for a connected graph. Alternatively, it can achieve O(n2) running time by using an unsorted list as a priority queue.

**Ford-Fulkerson algorithm**

It is an algorithm that finds a maximum flow between source and sink node in directed graph. There are two conditions.

* each edge in the graph has a fixed capacity and a flow through the edge must not exceed the capacity
* total incoming flow into any node must be equal to total outgoing flow from the node

1. Initially set the flow along every edge to 0.
2. Use a pathfinding algorithm like depth-first search (DFS) or breadth-first search (BFS) to find a path P from s to t that has available capacity.
3. Let cap(P) indicate the maximum amount of stuff that can flow along this path. To find the capacity of this path, it needs to look at all edges e on the path and subtract their current flow, fe, from their capacity ce. It will set cap(P) to be equal to the smallest value of ce - fe since this will bottleneck the path.
4. It then augments the flow across the edges in the path P by our cap(P) value.
5. Repeat the process from step 2 until there are no paths left from s to t that have available capacity.

By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. However, there is no certainty that this situation will ever be reached, so the best that can be guaranteed is that the answer will be correct if the algorithm terminates. In the case that the algorithm runs forever, the flow might not even converge towards the maximum flow. However, this situation only occurs with irrational flow values. When the capacities are integers, the runtime of Ford–Fulkerson is bounded by O(Ef), where E is the number of edges in the graph and f is the maximum flow in the graph. This is because each augmenting path can be found in O(E) time and increases the flow by an integer amount of at least 1, with the upper bound f.

# Results

1. Prim’s algorithm

Method get\_AdjMatrix(n, qty\_edge\_required) was implemented to get random adjacency matrix size of n x n. For the code see Appendix 1. Method get\_visualization(adjMatrix) to visualization adjacency matrix with graph. For the code see Appendix 1.

Graph visualization is shown on figure 1.

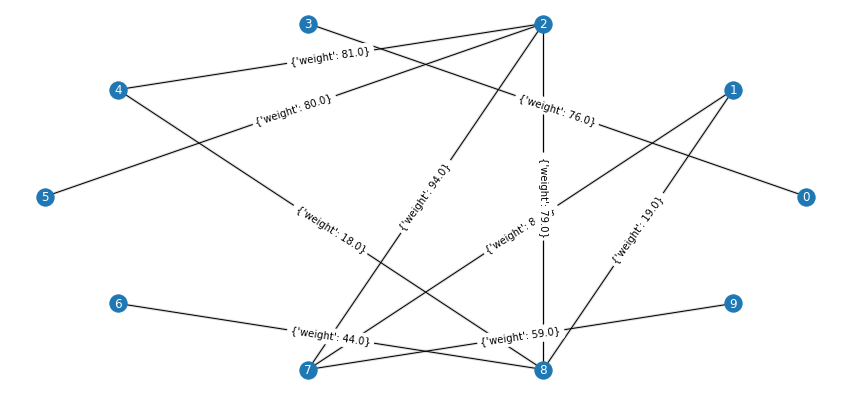


Figure 1 Graph visualization

Method prim(graph, staring\_vertex) has been implemented to find minimum spanning tree on the graph (figure 1) using priority queue. Code of the method is in Appendix 2. Visualization of minimum spanning tree obtained with the method shows by red edges on figure 2.

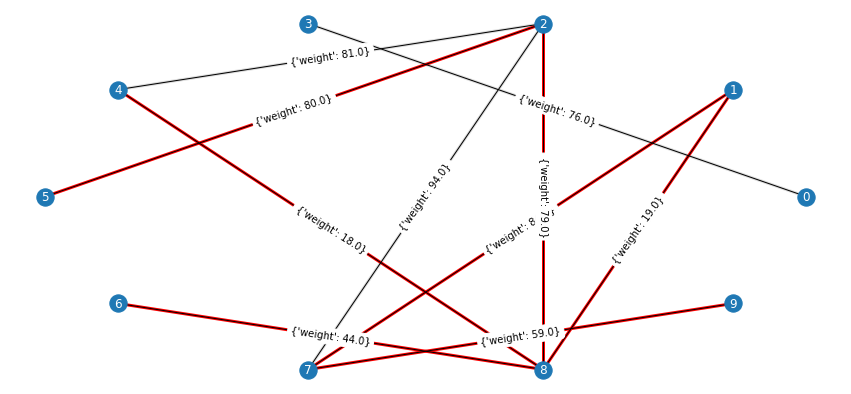


Figure 2 Minimum spanning tree (red edges)

Code in appendix 3 presents the way of time execution measurement and visualization of the results. Parameter n indicating the number of vertices has been varied from 10 to 1000 with step = 10. The number of edges was suggested to equal twice of vertexes number. Figure 3 shows obtained time performance measures where blue line denotes approximation function whereas red line obtained measures. Approximation function is n2 because the algorithm was enacted with priority queue which is default implementation of heapq module.

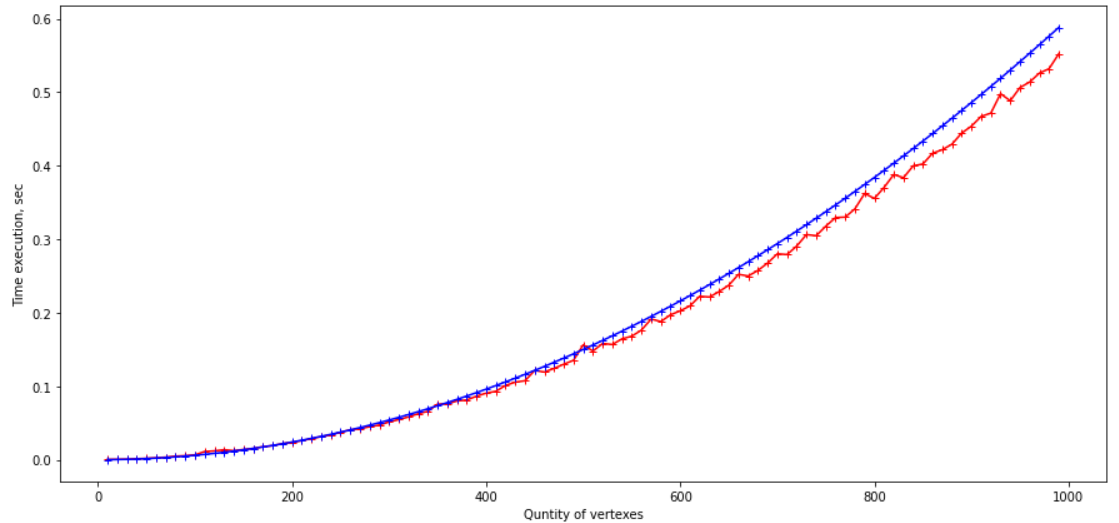


Figure 3 Time execution measurement for Prim’s algorithm (red) and approximation function (blue)

1. Maximum Flow (Ford-fulkerson algorithm)

To find the max flow for the flow network (figure 4) two methods have been enacted: depth\_first\_search(graph, source, sink) and ford\_fulkerson(graph, source, sink). Ford-Fulkerson algorithm detects some path from start vertex to sink vertex with few modified depth-first search and then find out minimum value of flow among every edges of this path. This procedure is repeated until no more new paths can be found.

from 0 to 1

at path ['0', '1'] ; current flow 1

at path ['0', '3', '7', '1'] ; current flow 3

at path ['0', '2', '5', '7', '1'] ; current flow 5

at path [] ; current flow 0

Result: at path ['0', '2', '5', '7', '1']; max flow = 5

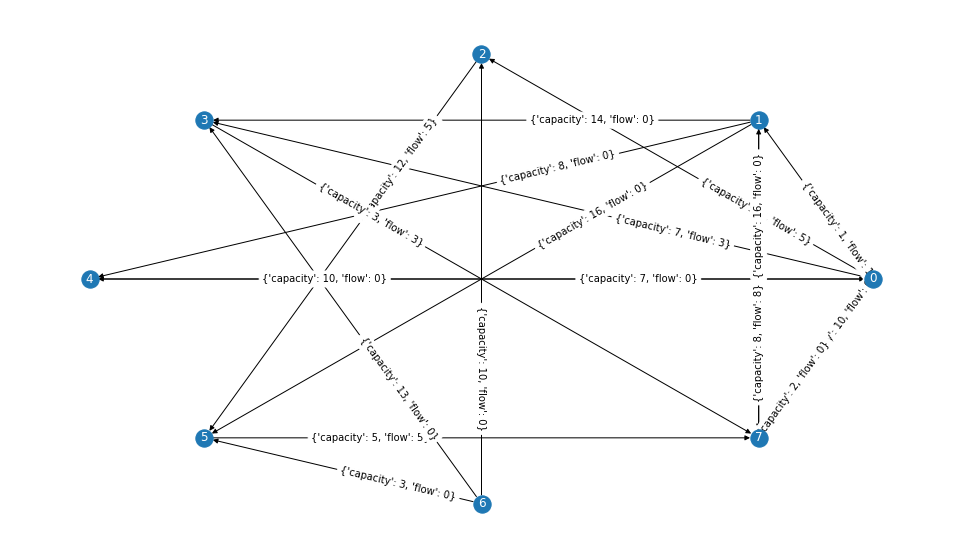


Figure 4 Flow network

Code in appendix 5 presents the way of time execution measurement and visualization of the results. Parameter n indicating the number of vertices has been varied from 10 to 1000 with step = 10. The number of edges was suggested to equal twice of vertexes number. Figure 5 shows obtained time performance measures where blue line denotes approximation function whereas red line obtained measures. Approximation function is O(fE), where f is considered by some almost not changing value lying in interval from 0 to 100. It is known from the general rules of time analysis that a constant can be canceled if there is a rapidly growing function nearby. For instance, O(C\*n2)≈O(n2). Taking this into account, the main increase in the execution time is provided by the increase in the number of edges.

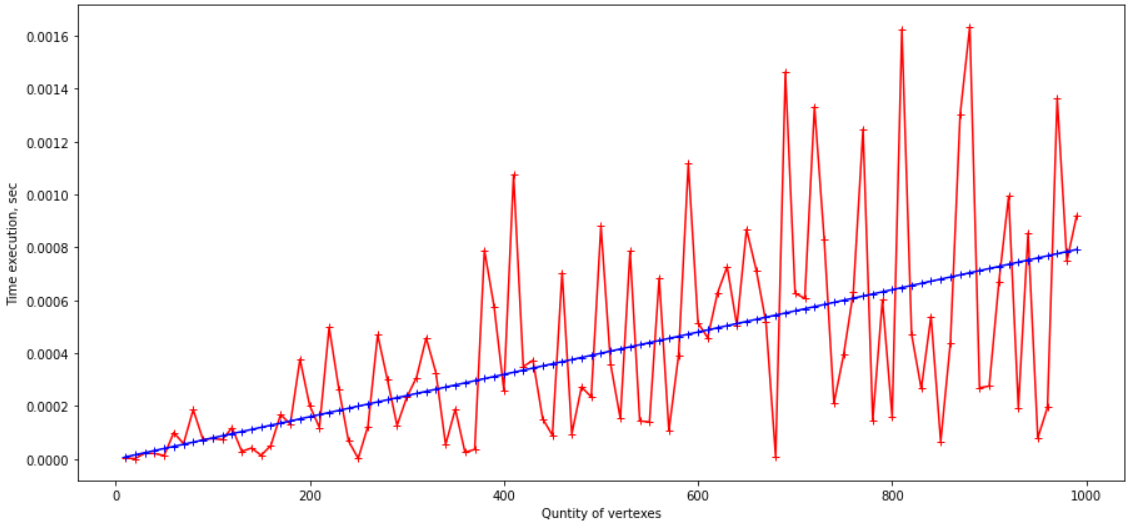


Figure 5 Time execution measurement for Ford-Fulkerson algorithm (red) and approximation function (blue)

# Conclusions

During the execution of the task, Prim’s and Ford-Fulkerson algorithms were implemented. Time analysis of these algorithms has been performed. 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] = numpy.random.randint(0, 100)

      adjMatrix[j][i] = adjMatrix[i][j]

      qty\_edge += 1

  return adjMatrix

def convert(adjMatrix):

    adjList = {}

    for i in range(len(adjMatrix)):

      temp = {}

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

        if adjMatrix[i][j] != 0:

          temp["{0}".format(j)] = adjMatrix[i][j]

      adjList["{0}".format(i)] = temp

    return adjList

**Appendix 2**

def prim(graph, starting\_vertex):

    mst = defaultdict(set)

    visited = set([starting\_vertex])

    edges = [(cost, starting\_vertex, to\_vert) for to\_vert, cost in graph[starting\_vertex].items()]

    heapq.heapify(edges)

    while edges:

      cost, from\_vert, to\_vert = heapq.heappop(edges)

      if to\_vert not in visited:

        visited.add(to\_vert)

        mst[from\_vert].add(to\_vert)

        for to\_next, cost in graph[to\_vert].items():

          if to\_next not in visited:

            heapq.heappush(edges, (cost, to\_vert, to\_next))

    return mst

n = 10

qty\_edge\_required = 10

starting\_vertex = "2"

print("Start from: ", starting\_vertex)

G = nx.Graph(adjMatrix)

prim\_graph = nx.Graph(prim(convert(adjMatrix), starting\_vertex))

prim\_graph\_edge = []

for edge in prim\_graph.edges:

  if int(edge[0]) < int(edge[1]):

    prim\_graph\_edge.append((int(edge[0]), int(edge[1])))

  else:

    prim\_graph\_edge.append((int(edge[1]), int(edge[0])))

color\_map = [] #marking the edges of mst

for edge in G.edges:

  if edge in prim\_graph\_edge:

    color\_map.append('red')

  else:

    color\_map.append('gainsboro')

pos = nx.circular\_layout(G)

plt.figure(3,figsize=(15,7))

labels = {} #create label for vertexes

for k in range(n):

  labels[k] = "{0}".format(k)

nx.set\_edge\_attributes(G, {e: adjMatrix[e[0]][e[1]] for e in G.edges()}, "weight")

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

nx.draw\_networkx\_edge\_labels(G, pos)

nx.draw\_networkx\_edges(G, pos, width=3, edge\_color=color\_map)

nx.draw(G, pos)

**Appendix 3**

#time measurement

def my\_function1():

  nx.Graph(prim(convert(adjMatrix), starting\_vertex))

n\_list = []

approximate\_line = []

for i in range(10, 1000, 10):

  n = i

  qty\_edge\_required = i \* 2

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

  #time\_list.append(timeit.timeit(my\_function1, number = 1))

  n\_list.append(n)

  approximate\_line.append(0.0000006\*n\*n)

plt.figure(figsize=(15,7)) #determed size of graph

plt.xlabel('Quntity of vertexes') #label of x axes

plt.ylabel('Time execution, sec') #label of y axes

plt.plot(n\_list, time\_list, marker="+", label = "Prim's algorithm", color='red')

plt.plot(n\_list, approximate\_line, marker="+", label = "Approximation line", color='blue')

**Appendix 4**

def ford\_fulkerson(graph, source, sink):

    flow, path = 0, True

    max\_flow = 0

    while path:

        # search for path with flow reserve

        path, reserve = depth\_first\_search(graph, source, sink)

        # increase flow along the path

        for v, u in zip(path, path[1:]):

            if graph.has\_edge(v, u):

                graph[v][u]['flow'] += reserve

            else:

                graph[u][v]['flow'] -= reserve

    # show intermediate results

        print('at path', path, '; current flow', reserve)

        if reserve > max\_flow:

          max\_flow = reserve

          best\_path = path

    print('Result: at path', best\_path, '; max flow = ', max\_flow)

def depth\_first\_search(graph, source, sink):

    undirected = graph.to\_undirected()

    explored = {source}

    stack = [(source, 0, dict(undirected[source]))]

    while stack:

        v, \_, neighbours = stack[-1]

        if v == sink:

            break

        # search the next neighbour

        while neighbours:

            u, e = neighbours.popitem()

            if u not in explored:

                break

        else:

            stack.pop()

            continue

        # current flow and capacity

        in\_direction = graph.has\_edge(v, u)

        capacity = e['capacity']

        flow = e['flow']

        neighbours = dict(undirected[u])

        # increase or redirect flow at the edge

        if in\_direction and flow < capacity:

            stack.append((u, capacity - flow, neighbours))

            explored.add(u)

        elif not in\_direction and flow:

            stack.append((u, flow, neighbours))

            explored.add(u)

    # (source, sink) path and its flow reserve

    reserve = min((f for \_, f, \_ in stack[1:]), default=0)

    path = [v for v, \_, \_ in stack]

    return path, reserve

#initial data

n = 8

qty\_edge\_required = 20

#create graph with given data

graph = nx.DiGraph()

for vertex in range(n):

  graph.add\_nodes\_from('{0}'.format(vertex))

for edge in range(qty\_edge\_required):

  vert\_from = '{0}'.format(random.choice(list(graph.nodes)))

  while True:

    vert\_to = '{0}'.format(random.choice(list(graph.nodes)))

    if vert\_to != vert\_from:

      break

  edge = "{0}, {1}".format(vert\_from, vert\_to)

  if edge in list(graph.edges):

    break

  else:

    graph.add\_edges\_from([(vert\_from, vert\_to, {'capacity': numpy.random.randint(1,20), 'flow': 0})])

#random choose two vertexes

source = random.choice(list(graph.nodes))

while True:

  sink = random.choice(list(graph.nodes))

  if sink != source:

    break

#create reliable solution

graph.add\_edges\_from([(source, sink, {'capacity': 1, 'flow': 0})])

#run algorithm

print("from", source, "\t to", sink)

ford\_fulkerson(graph, source, sink)

#create visualization

pos = nx.circular\_layout(graph)

plt.figure(3,figsize=(17,10))

labels = {}

for k in range(n):

  labels["{0}".format(k)] = "{0}".format(k)

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

nx.draw\_networkx\_edge\_labels(graph, pos, label\_pos = 0.7)

nx.draw(graph, pos)

**Appendix 5**

def my\_function2():

  ford\_fulkerson(graph, source, sink)

n\_list = []

time\_list = []

approximate\_line = []

for i in range(10, 1000, 10):

  n = i

  qty\_edge\_required = i \* 2

  graph = nx.DiGraph()

  for vertex in range(n):

    graph.add\_nodes\_from('{0}'.format(vertex))

  for edge in range(qty\_edge\_required):

    vert\_from = '{0}'.format(random.choice(list(graph.nodes)))

    while True:

      vert\_to = '{0}'.format(random.choice(list(graph.nodes)))

      if vert\_to != vert\_from:

        break

    edge = "{0}, {1}".format(vert\_from, vert\_to)

    if edge in list(graph.edges):

      break

    else:

      graph.add\_edges\_from([(vert\_from, vert\_to, {'capacity': numpy.random.randint(1,20), 'flow': 0})])

  source = random.choice(list(graph.nodes))

  while True:

    sink = random.choice(list(graph.nodes))

    if sink != source:

      break

  graph.add\_edges\_from([(source, sink, {'capacity': 1, 'flow': 0})])

  time\_list.append(timeit.timeit(my\_function2, number = 1))

  n\_list.append(n)

  approximate\_line.append(0.00004\*qty\_edge\_required)

plt.figure(figsize=(15,7)) #determed size of graph

plt.xlabel('Quntity of vertexes') #label of x axes

plt.ylabel('Time execution, sec') #label of y axes

plt.plot(n\_list, time\_list, marker="+", label = "Ford-fulkerson algorithm", color='red')

plt.plot(n\_list, approximate\_line, marker="+", label = "Approximation line", color='blue')