FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION

OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

on the practical task No. 6

“Algorithms on graphs. Path search algorithms on weighted graphs”

Performed by

Victor Voronin

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg

2020

# Goal

The use of path search algorithms on weighted graphs (Dijkstra's, A\* and Bellman-Ford algorithms).

# Formulation of the problem

1. Generate a random adjacency matrix for a simple undirected weighted graph of 100 vertices and 500 edges with assigned random positive integer weights (note that the matrix should be symmetric and contain only 0s and weights as elements). Use Dijkstra's and Bellman-Ford algorithms to find shortest paths between a random starting vertex and other vertices. Measure the time required to find the paths for each algorithm. Repeat the experiment 10 times for the same starting vertex and calculate the average time required for the paths search of each algorithm. Analyse the results obtained.
2. Generate a 10x10 cell grid with 30 obstacle cells. Choose two random non-obstacle cells and find a shortest path between them using A\* algorithm. Repeat the experiment 5 times with different random pair of cells. Analyse 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 sample random node
* library Nympy for work with array
* library timeit to measurement of execution time of algorithms
* library heapq for implementation of Dijkstra’s algorithm.
* Library bellmanford to apply Bellman-Ford algorithm

**Dijkstra’s algorithm**

Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.[16]
3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Bounds of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of the number of edges, denoted |E|, and the number of vertices, denoted |V|, using big-O notation. The complexity bound depends mainly on the data structure used to represent the set Q.

**Bellman-Ford algorithm**

Unlike Dijkstra's algorithm, the Bellman–Ford algorithm can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex s. The presence of such cycles means there is no shortest path, since the total weight becomes lower each time the cycle is traversed.

The algorithm initializes the distance to the source to 0 and all other nodes to infinity. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value. At each iteration i that the edges are scanned, the algorithm finds all shortest paths of at most length i edges (and possibly some paths longer than i edges). Since the longest possible path without a cycle can be |V|-1 edges, the edges must be scanned |V|-1 times to ensure the shortest path has been found for all nodes. A final scan of all the edges is performed and if any distance is updated, then a path of length |V| edges has been found which can only occur if at least one negative cycle exists in the graph.

Bellman–Ford runs in O(|V|\*|E|) time, where |V| and |E| are the number of vertices and edges respectively.

**A\* algorithm**

Starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration of its main loop, A\* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A\* selects the path that minimizes f(n)=g(n)+h(n) where n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal. A\* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended.

Implementations of A\* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a removed node (thus the node with the lowest f value out of all fringe nodes) is a goal node. The f value of that goal is then also the cost of the shortest path since h at the goal is zero in an admissible heuristic.

# Results

1. 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.
2. Graph visualization is shown on figure 1.

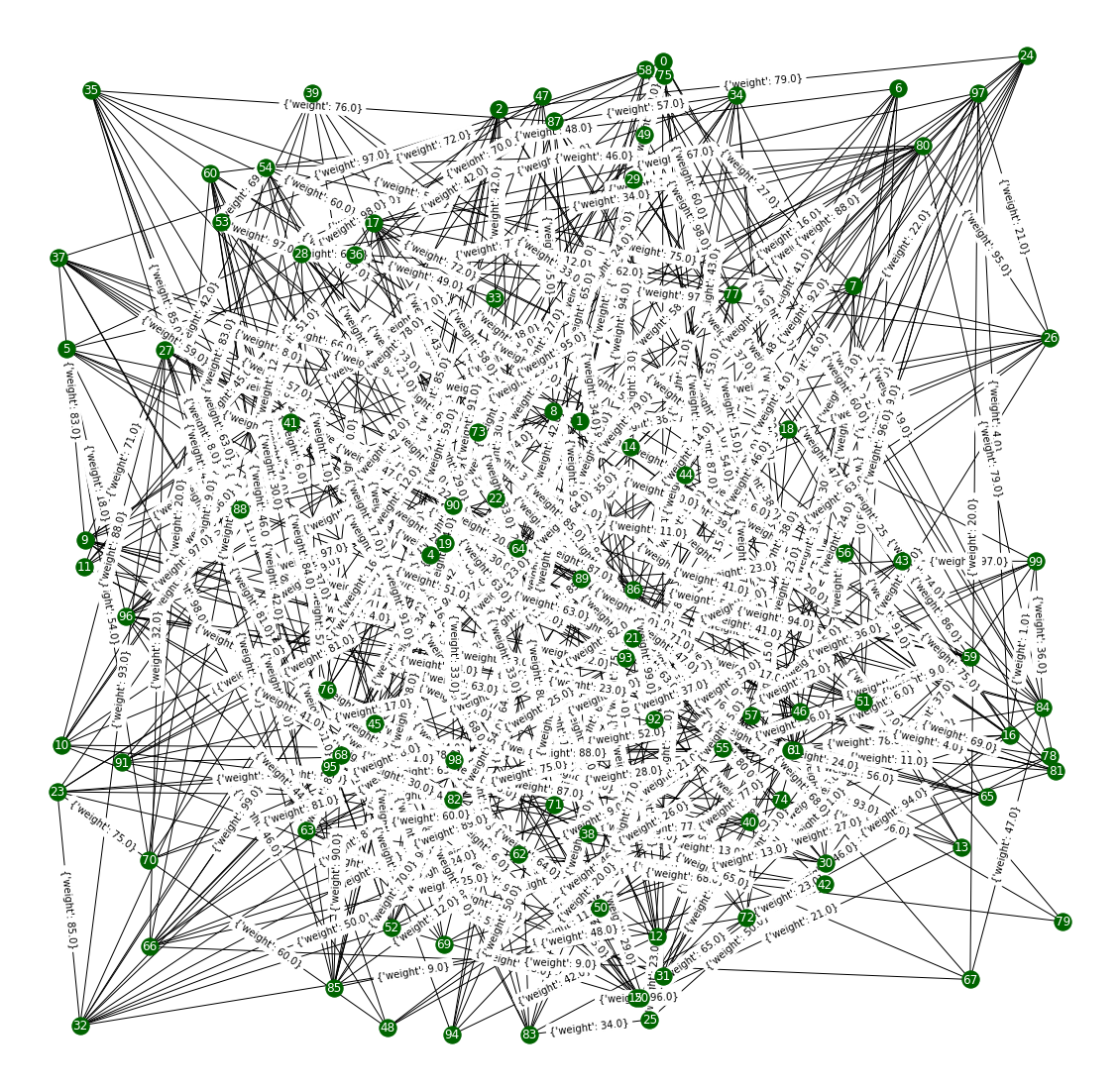


Figure 1 Graph visualization

1. The shortest path between start node and all other in the graph (fig. 1) can be found by few methods:
   1. The first, with Dijkstra’s algorithm. It was implemented with function nx.single\_source\_dijkstra\_path of network library. Execution time can be measurement with function timeit.
   2. The second, with Bellman-Ford algorithm.It was implemented also with function nx.single\_source\_bellman\_ford\_path of networkx library. Measurement time is enacted the same way.

For the graph (fig.1) follow result was obtained with code in Appendix 2:

* Average time: 0.001841113800014682 for 10 attempts of Dijkstra's algorithm
* Average time: 0.004593348200000946 for 10 attempts of Bellman-Ford algorithm

The results demonstrate fourfold difference of execution time between the methods. Largely due to Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and performs relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes all the edges, and does this |V|-1 times, where |V| is the number of vertices in the graph.

1. There is another example of code’s work on simpler graph showing on figure 2. Result for this case:

AdjMatrix:

[[ 0. 0. 0. 22. 0. 0. 0. 80. 34. 11.]

[ 0. 0. 0. 0. 0. 0. 0. 98. 0. 20.]

[ 0. 0. 0. 93. 0. 0. 0. 86. 0. 0.]

[22. 0. 93. 0. 0. 0. 0. 0. 97. 99.]

[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 28.]

[ 0. 0. 0. 0. 0. 0. 0. 71. 0. 0.]

[ 0. 0. 0. 0. 0. 0. 0. 0. 54. 79.]

[80. 98. 86. 0. 0. 71. 0. 0. 85. 0.]

[34. 0. 0. 97. 0. 0. 54. 85. 0. 0.]

[11. 20. 0. 99. 28. 0. 79. 0. 0. 0.]]

Start node for both algorithms: 2

Result:

For Dijkstra's algorithm: {2: [2], 3: [2, 3], 7: [2, 7], 0: [2, 3, 0],

1: [2, 3, 0, 9, 1], 5: [2, 7, 5], 8: [2, 3, 0, 8],

9: [2, 3, 0, 9], 4: [2, 3, 0, 9, 4], 6: [2, 3, 0, 8, 6]}

For Bellman-Ford algorithm: {2: [2], 3: [2, 3], 7: [2, 7],

0: [2, 3, 0], 8: [2, 3, 0, 8], 9: [2, 3, 0, 9], 1: [2, 3, 0, 9, 1],

5: [2, 7, 5], 6: [2, 3, 0, 8, 6], 4: [2, 3, 0, 9, 4]}

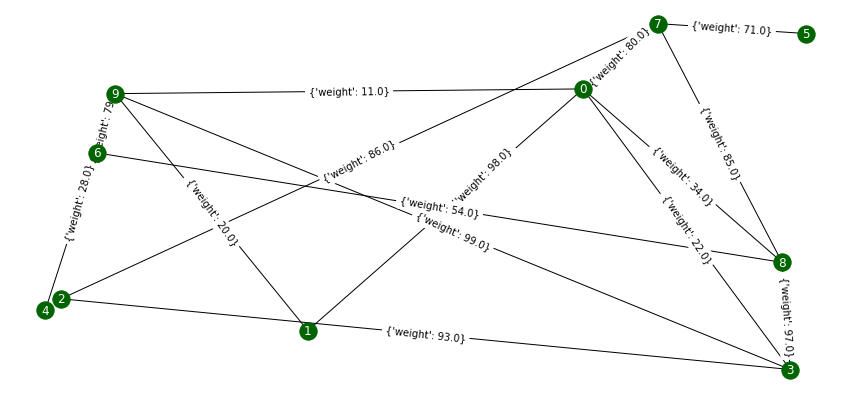


Figure 2 Simpler case of graph

1. There is generated graph-labyrinth on figure 3. Shortest path (marked by red) between two nodes (marked by green colour) was found with method astar\_path(). Method remove\_node(graph, how\_many) to remove nodes from graph in given the amount. Method dist(a, b) is auxiliary method for A\* algorithm and return distance.

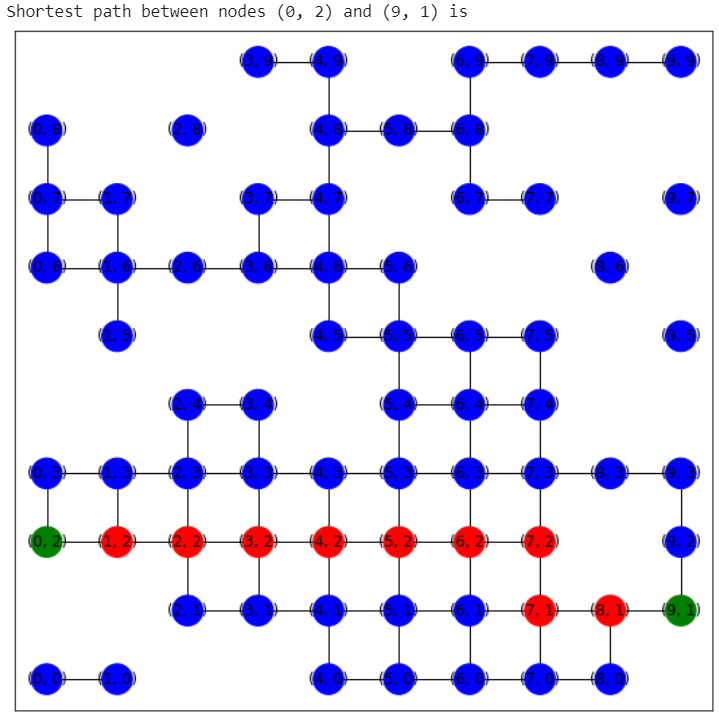


Figure 3 Shortest path was obtained with A\* algorithm

Let generate another labyrinth-graph (fig.4) and will find average time of execution.

Start (8, 5) target (8, 5)

Shortest path between nodes (8, 5) and (8, 5) is

[(8, 5)]

Start (6, 6) target (5, 7)

Shortest path between nodes (6, 6) and (5, 7) is

[(6, 6), (6, 7), (5, 7)]

Start (4, 8) target (1, 4)

Shortest path between nodes (4, 8) and (1, 4) is

[(4, 8), (4, 7), (4, 6), (4, 5), (4, 4), (3, 4), (2, 4), (1, 4)]

Start (7, 2) target (4, 4)

Shortest path between nodes (7, 2) and (4, 4) is

[(7, 2), (6, 2), (5, 2), (4, 2), (4, 3), (4, 4)]

Start (4, 2) target (8, 2)

Shortest path between nodes (4, 2) and (8, 2) is

[(4, 2), (5, 2), (6, 2), (7, 2), (8, 2)]

Average time for 5 attempts: 0.00030044800000723627

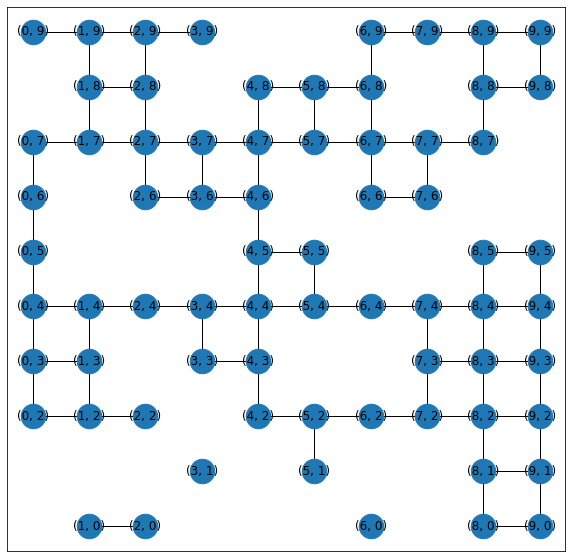


Figure 4 Another labyrinth

The fact that the algorithm is ahead of other algorithms is due only to the size of the graphs. When running all algorithms on graphs with similar parameters, the run time is very close.

# Conclusions

During the execution of the task, weighted random graph was generated and then Dijkstra’s algorithm, Bellman-Ford and A\* for search the shortest path. 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 get\_visualization(adjMatrix):

  n = len(adjMatrix)

  G = nx.Graph(adjMatrix)

  pos = nx.random\_layout(G)

  plt.figure(3,figsize=(15,7))

  labels = {}

  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\_networkx\_edge\_labels(G, pos)

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

Appendix 2

n = 100 #count of nodes

qty\_edge\_required = 500 #count of edges

start = 2 #start node for search the shortest path

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

G = nx.Graph(adjMatrix)

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

get\_visualization(adjMatrix)#draw graph

def my\_function1():

  start = random.choice(list(G)) #change start node each time

  print("start", start)

  nx.single\_source\_dijkstra\_path(G, start, weight='weight')

average\_time\_dijkstra = timeit.timeit(my\_function1, number = 10) / 10

print("Average time: ",average\_time\_dijkstra, "for 10 attempts of Dijkstra's algorithm")

def my\_function2():

  start = random.choice(list(G))

  print("start", start)

  nx.single\_source\_bellman\_ford\_path(G, start, weight='weight')

average\_time\_bellman = timeit.timeit(my\_function2, number = 10) / 10

print("Average time: ", average\_time\_bellman, "for 10 attempts of Bellman-Ford algorithm")

Appendix 3

def dist(a, b):

  (x1, y1) = a

  (x2, y2) = b

  return ((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2) \*\* 0.5

def remove\_node(graph, how\_many):

  for i in range(how\_many):

    random\_node = random.choice(list(graph.nodes))

    graph.remove\_node(random\_node)

  return graph

N=10

G = nx.grid\_graph(dim=[N, N]) # nodes are two-tuples (x,y)

remove\_node(G, 30)

def my\_function3():

  start = random.choice(list(G.nodes))

  target = random.choice(list(G.nodes))

  print("Start", start, "target", target)

  print("Shortest path between nodes {0} and {1} is ".format(start, target

))

  try:

    shortest\_path = (nx.astar\_path(G, start, target, heuristic=dist,weight

="W"))

    print(shortest\_path)

  except:

    print("Path does not exist")

average\_time\_astar = timeit.timeit(my\_function3, number = 5) / 5

print(average\_time\_astar)

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

pos = dict( (n, n) for n in G.nodes() ) #Dictionary of all positions

nx.draw\_networkx(G, pos = pos, node\_size=600)