jjFEDERAL 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

*Mariia Koroleva*

*J42322c*

Accepted by

Dr Petr Chunaev

St. Petersburg

2021

***Goal***

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

**Formulation of the problem**

***I.*** *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. Analyze the results obtained.*

***II.*** *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. Analyze the results obtained.*

***III.*** *Describe the data structures and design techniques used within the algorithms.*

**Brief theoretical part**

***Dijkstra’s algorithm*** is an algorithm, that, given a weighted graph (with positive weights) and a source vertex, finds shortest paths from the source to all other vertices.

***Bellman-Ford algorithm***is an algorithm, that, *g*iven a weighted graph (possibly directed and with negative weights) and a source vertex s, finds shortest paths from s to all vertices in the graph.

***A\* algorithm*** is an algorithm, that, given a weighted graph (with positive weights), a source vertex and a target vertex, finds a shortest path from the source to the target. At each iteration, A\* determines how to extend the path basing on the cost of the current path from the source and an estimate of the cost required to extend the path to the target

**Results**

After repeating every experiment 10 times the following results were obtained:

Average time required for the paths search of Dijkstra’s algorithm is : 222.6 nanoseconds

Average time required for the paths search of Bellman-Ford algorithm is : 228.1 nanoseconds

As we see, the BFA is less time-efficient, however that is compensated with features that allow to obtain the results for directed and negative-weighted graphs.

A\* algorithm results:

Our grid:

[[1., 1., 1., 0., 1., 0., 0., 0., 0., 0.],

[0., 0., 1., 0., 0., 0., 1., 0., 0., 0.],

[1., 0., 0., 0., 0., 0., 0., 1., 0., 1.],

[0., 1., 0., 0., 0., 0., 1., 1., 0., 0.],

[1., 1., 0., 1., 0., 1., 1., 0., 1., 0.],

[0., 0., 0., 1., 0., 0., 1., 0., 1., 0.],

[0., 0., 0., 0., 0., 1., 0., 1., 0., 0.],

[0., 0., 0., 0., 1., 0., 0., 1., 0., 1.],

[0., 0., 0., 0., 0., 1., 0., 1., 0., 0.],

[0., 0., 0., 0., 1., 0., 0., 0., 0., 1.]]

Node representation of the grid:

[[-1., -1., -1., 3., -1., 5., 6., 7., 8., 9.],

[10., 11., -1., 13., 14., 15., -1., 17., 18., 19.],

[-1., 21., 22., 23., 24., 25., 26., -1., 28., -1.],

[30., -1., 32., 33., 34., 35., -1., -1., 38., 39.],

[-1., -1., 42., -1., 44., -1., -1., 47., -1., 49.],

[50., 51., 52., -1., 54., 55., -1., 57., -1., 59.],

[60., 61., 62., 63., 64., -1., 66., -1., 68., 69.],

[70., 71., 72., 73., -1., 75., 76., -1., 78., -1.],

[80., 81., 82., 83., 84., -1., 86., -1., 88., 89.],

[90., 91., 92., 93., -1., 95., 96., 97., 98., -1.]]

Expriment 0

Chosen nodes(start-finish): 35 11

Shortest route: [35, 24, 13, 22, 11]

Expriment 1

Chosen nodes(start-finish): 95 44

Shortest route: [95, 84, 73, 64, 54, 44]

Expriment 2

Chosen nodes(start-finish): 55 70

Shortest route: [55, 64, 73, 72, 71, 70]

Expriment 3

Chosen nodes(start-finish): 52 62

Shortest route: [52, 62]

Expriment 4

Chosen nodes(start-finish): 42 88

Shortest route: [42, 52, 63, 64, 75, 86, 97, 88]

All the received paths are correct and really efficient and the algorithm did not take very long to get these right results.

**Conclusions**

Through this work, the following goals were achieved:

* detailed analysis of the work of algorithms by implementing in python;
* gained valuable experience with data structures such as graphs and grids;
* studied different methods of their representation and python representation (adjacency matrix, adjacency list).

**Appendix**

**# Part I**

import numpy as np

import math

**def** random\_adj\_matrix\_w(num\_verts, num\_edges):

    '''

    Generates a random adjacency matrix for a simple undirected weighted graph

    Parameters:

    ----------

    num\_verts - int, number of vertices

    num\_edges - int, number of edges

    Output:

    result - 2D array

    '''

    num\_sells\_triangle = int((num\_verts - 1) \* num\_verts / 2) *# counting the number of sells that will be symmetric*

    edge\_indices = np.random.choice(range(num\_sells\_triangle), num\_edges, replace=False) *# picking random connections (pairs)*

    triangle\_list = np.zeros(num\_sells\_triangle) *# initializing the list*

    triangle\_list[edge\_indices] = np.random.randint(1, 10, num\_edges) *# generating weights*

    result = np.zeros((num\_verts, num\_verts)) *# initializing the result matrix*

    triangle\_indices = np.triu\_indices(num\_verts, k=1) *# all the indices of the upper triangle (the diagonal is not included)*

    result[triangle\_indices] = triangle\_list *# pasting the random ones into the result matrix*

    result = result + np.rot90(np.fliplr(result)) *# pasting symmetric ones*

    return result

*# Generating a random adjecency matrix of a weighted grapgh*

m = random\_adj\_matrix\_w(100, 500)

**def** adj\_matrix\_to\_list(matrix):

  '''

  Transfer adjecency matrix into adjecency list

  Paramers:

  matrix - 2D array

  Output:

  result - 1D array

  '''

  result = [set(j for j, cell in enumerate(row) if cell != 0) for row in matrix]

  return result

**def** DA(graph\_matrix, source):

  '''

  Finds the shortest paths between a source vertice to other vertices using Dijkstra's algorithm.

  Params:

  ------

  graph\_matrix: array, graph's adjecency matrix

  source: int, index number of a source vertice

  Output:

  -----

  dists: a dictionary where keys are node numbers and values are distances from the source node to a given node

  paths: a dictionary of the shortest routes from the source vertice to other vertices

  '''

*# Creating an SPT set*

  adj\_list = adj\_matrix\_to\_list(graph\_matrix)

  spt = [(0, source)] *# format (distance, vertice)*

  paths = {i: None for i in range(len(adj\_list))}

  dists = {i: float('infinity') for i in range(len(adj\_list))} *# initializing distances dictionary*

  dists[source] = 0

  while spt or spt == [(0, 0)]:

    new\_dist, new\_node = min(spt) *# picking a new vertice*

    spt.remove(min(spt)) *#removing it from spt*

    if new\_dist > dists[new\_node]:

      continue

    for neighbour in adj\_list[new\_node]:

         weight = graph\_matrix[new\_node, neighbour]

         dist = new\_dist + weight

         if dist < dists[neighbour]:  *# Updating distance values*

            dists[neighbour] = dist

            paths[neighbour] = new\_node

            spt.append((dist, neighbour))

*# def get\_path(start, finish, paths):*

*#   previous = paths[finish]*

*#   res = [previous]*

*#   while previous != None:*

*#       previous = paths[previous]*

*#       res.insert(0, previous)*

*#   return res*

*# routes = {i: get\_path(0, i, DA(graph\_m, 0)[1]) for i in range(len(graph\_list))}*

  return dists, paths

**def** edges(matrix):

  '''

  Getting the list of edges

  Params:

  matrix: graph's adjecency matrix

  Output:

  result: array with edges with (start, finish)

  '''

  result = [[(i, j) for j in range(len(matrix)) if matrix[i][j] != 0] for i in range(matrix.shape[0])]

  return result

**def** BFA(graph, source):

  '''

  Finds the shortest paths between a source vertice to other vertices using Bellman-Ford algorithm.

  Params:

  ------

  graph\_matrix: array, graph's adjecency matrix

  source: int, index number of a source vertice

  Output:

  -----

  dists: a dictionary where keys are node numbers and values are distances from the source node to a given node

  paths: a dictionary of the shortest routes from the source vertice to other vertices

  '''

  adj\_list = adj\_matrix\_to\_list(graph)

  edge\_list = edges(graph)

  paths = {i: None for i in range(len(adj\_list))}

  dists = {i: float('infinity') for i in range(len(adj\_list))}

  dists[source] = 0

  for i in range(len(adj\_list) - 1):

    i += 1

    for sublist in edge\_list:

      for edge in sublist:

        node, neighbour = edge

        weight = graph[node, neighbour]

        dist = dists[node] + weight

        if dist < dists[neighbour]:

          paths[neighbour] = node

          dists[neighbour] = dist

  return dists, paths

import time

*# Time Function*

**def** time\_func(alg):

  '''

  Measures the time required for an algorithm

  Params:

  alg - a measured function

  Output:

  t - time, required for the algorithm

  '''

  t = time.perf\_counter\_ns()

  alg

  t = time.perf\_counter\_ns() - t

  return t

*# Average time*

**def** avg\_time(alg, n):

    '''

    Measures the average time for an algorithm

    Params:

    -----

    alg - the measured algorithm

    n - number of experiment repetitions

    Output:

    res - average time, required for the algorithm

    '''

    time\_array = np.zeros(n)

    for i in range(n):

      time\_array[i] = time\_func(alg)

    res = np.mean(time\_array)

    return res

*# Finding the shortest distances*

start = np.random.randint(0, len(m))

print('Average time required for the paths search of DA is : {} nanoseconds'.format(avg\_time(DA(m, start), 10)))

print('Average time required for the paths search of BFA is : {} nanoseconds'.format(avg\_time(BFA(m, start), 10)))

**# Part II**

**def** generate\_cell\_grid(shape, obsts\_num):

  '''

  Generates a cell grid with obstacle cells

  Params:

  -------

  shape: tuple, shape of a cell grid

  obsts\_num: int, number of obstacles

  Output:

  grid: 2D array, where 1s are the obstacle cells

  obsts: array of integers, obstacles' indices

  cells: array of integers, non-obstacles' indices

  '''

  grid = np.zeros(shape)

*# Allocating a number for each cell for our convinience (that will help to consider them as nodes further)*

  cells = list(range(shape[0]\*shape[1]))

*# Picking random cells as obstacles*

  obsts = np.random.choice(range(shape[0]\*shape[1]), obsts\_num, replace=False)

  i, j = 0, 0

  for n in range(shape[0]\*shape[1]):

    if n in obsts:

      grid[i][j] = 1

      cells.remove(n)

    j += 1

    if j % shape[0] == 0:

      i += 1

      j = 0

  return grid, obsts, cells

**def** grid\_to\_grapgh(grid):

    '''

    Provides another way of visualizing a grid, where obstacle cells are equal to -1

    and non-obstacles are equal to node numbers

    Params:

    -------

    grid: 2D array, where 1s are the obstacle cells

    Output:

    result: 2D array, a transformed grid as written above

    '''

    shape =grid[0].shape

    obsts = grid[1]

    i, j = 0, 0

    result = grid[0]

    for n in range(shape[0]\*shape[1]):

      result[i][j] = n

      if n in obsts:

         result[i][j] = -1

      j += 1

      if j % shape[0] == 0:

        i += 1

        j = 0

    return result

**def** grid\_adj\_list(grid):

    '''

    Makes an adjecencent list of a graph that is considered in a grid

    Params:

    grid: 2D array, where obstacle cells are equal to -1

    and non-obstacles are equal to node numbers

    Output:

    res: 1D array, adjecency list

    '''

    shape =grid.shape

    corners = [0, shape[0] - 1, shape[0]\*shape[1] - shape[0], shape[0]\*shape[1] - 1]

    res = []

    for n in range(shape[0]\*shape[1]):

        i, j = n // shape[0], n % shape[0]

        curr = grid[i][j]

        neighbours = set()

        if curr < 0:

            pr\_neighbours = []

*# corner check*

        elif curr in corners:

            if curr == 0:

                pr\_neighbours = [grid[i][j+1], grid[i+1][j], grid[i+1][j+1]]

            elif curr == corners[1]:

                pr\_neighbours = [grid[i][j-1], grid[i+1][j-1], grid[i+1][j]]

            elif curr == corners[2]:

                pr\_neighbours = [grid[i-1][j], grid[i-1][j+1], grid[i][j+1]]

            elif curr == corners[3]:

                pr\_neighbours = [grid[i-1][j-1], grid[i-1][j], grid[i][j-1]]

*# border check*

        elif i == 0:

            pr\_neighbours = [grid[i][j-1], grid[i][j+1], grid[i+1][j-1], grid[i+1][j],  grid[i+1][j+1]]

        elif i == shape[0] - 1:

            pr\_neighbours = [grid[i-1][j-1], grid[i-1][j], grid[i-1][j+1], grid[i][j-1], grid[i][j+1]]

        elif j == 0:

            pr\_neighbours = [grid[i-1][j], grid[i-1][j+1], grid[i][j+1], grid[i+1][j],  grid[i+1][j+1]]

        elif j == shape[1] - 1:

            pr\_neighbours = [grid[i-1][j-1], grid[i-1][j], grid[i][j-1], grid[i+1][j-1], grid[i+1][j]]

        else:

            pr\_neighbours = [grid[i-1][j-1], grid[i-1][j], grid[i-1][j+1], grid[i][j-1], grid[i][j+1], grid[i+1][j-1], grid[i+1][j],  grid[i+1][j+1]]

        for node in pr\_neighbours:

            if node > 0:

                neighbours.add(node)

        res.append(neighbours)

    return res

**def** adj\_list\_to\_matrix(adj\_list, shape):

    '''

    Transforms graph's adjecency list into adjecency matrix

    Params:

    adj\_list: 1D array, adjecency list

    shape: tuple, shape of a matrix to transform to (n\*m must be equal to len(adj\_list))

    Output:

    result: 2D array, adjecency matrix

    '''

    result = np.zeros((shape[0]\*shape[1], shape[0]\*shape[1]))

    for i in range(shape[0]\*shape[1]):

        for j in range(shape[0]\*shape[1]):

          if i==j or adj\_list[i] == set():

              continue

          else:

              neighbours = adj\_list[i]

          if j in neighbours:

              result[i][j] = 1

    return result

**def** distance(p1, p2):

    '''

    Counts eucleadian distance

    '''

    res = math.sqrt( ((int(p1[0])-int(p2[0]))\*\*2)+((int(p1[1])-int(p2[1]))\*\*2) )

    return res

**def** coordinates(node, grid):

    shape = grid[0].shape

    i, j = node // shape[0], node % shape[0]

    return i, j

**def** A\_star(start, finish, adj\_matrix):

    '''

    Finds the shortest path in a graph from start to finish by use of A\* algorithm

    Params:

    start: integer, start node index

    finish: integer, finish node index

    adj\_matrix: 2D array, graph's adjecency matrix

    Output:

    path: 1D array, the shortest route form start to finish

    '''

    adj\_list = adj\_matrix\_to\_list(adj\_matrix)

    c\_t = coordinates(finish, grid)

    c\_s = coordinates(start, grid)

    dists = {i: float('infinity') for i in range(len(adj\_list))}

    dists[start] = 0

    spt = [(distance(c\_s, c\_t), start)] *# format (distance, vertice)*

    path = []

    new\_node = -1

    while new\_node != finish:

      new\_dist, new\_node = min(spt) *# picking a new vertice*

      spt = [] *#removing it from spt*

      path.append(new\_node)

      for neighbour in adj\_list[new\_node]:

          weight = adj\_matrix[new\_node, neighbour]

          g = dists[new\_node] + weight

          if g < dists[neighbour]:  *# Updating distance values*

              dists[neighbour] = g

          c\_k = coordinates(neighbour, grid)

          h = distance(c\_k, c\_t)

          f = g + h

          spt.append((f, neighbour))

    return path

grid = generate\_cell\_grid((10, 10), 30)

grid[0]

node\_grid = grid\_to\_grapgh(grid)

node\_grid

graph\_list = grid\_adj\_list(node\_grid)

graph\_m = adj\_list\_to\_matrix(graph\_list, (10,10))

*# Conducting 5 experiments by picking a pair of random points*

for i in range(5):

  a, b = np.random.choice(grid[2], 2, replace=False)

  print('Expriment ', i)

  print('Chosen nodes(start-finish):', a, b)

  result = A\_star(a, b, graph\_m)

  print('Shortest route: ', result, '\n')