OR PROGRAMMING EXERCISE

February 17, 2020

1 GETTING THE DATA SORTED

Before we get started, some info: We use python programming language. The problem is to apply a dynamic program to find the shortest path. We are given a text file, representing a given Direct Acyclic Graph. There are different ways of representing a graph, it could a matrix, or a list of lists. In our case we use a dictionary. A dictionary is tupled pair of data represented by a 'key' and its 'value'. For example, consider that node 1 have to out-going arcs to node 2 adn 3 with weith 4 and 5. Putting this in a dictionary gives: dict = {'1':[('2', 4), ('3', 5)]}. Of course a complete dictionary has more enteries.

[1]:
```python
def convert_text_to_dictionairy(filename):

    # PART ONE - IN THIS PART WE READ THE FILE LINE BY LINE AND SAVE IT TO
    # A LIST
    doc = []
    with open(filename) as file:
        for line in file:
            t = line.strip().split('\t')
            doc.append(t)


    # PART TWO - HERE WE FIND OUT WHAT MAIN-NODES WE HAVE,
    # I.E. THE FIRST ENTRY OF EVERY LINE
    main_nodes = []
    for line in doc[1:]:
        main_nodes.append(line[0])


    # PART THREE - YOU WILL NOTICE THAT IN THE PREVIOUS PART THE MAIN_NODES
    # HAS ITEMS THAT OCCUR MULTIPLE TIMES HENCE WE NEED TO USE THE SET()
    # FUNCTION TO OPBTIAN THE UNIQUE ONES.
    set_nodes = set(main_nodes)
    set_nodes_list = list(set_nodes)
    # -> WE NEED TO SORT IT, BECAUSE THE SET() FUNCTION IS IMMUTABLE OBJECT
    #    AND IS NOT SORTED, THATS WHY WE
    # -> CONVERT IT TO A LIST SO THAT WE CAN LATER ON ADD MORE NODES AND
    #    OFCOURSE SORT IT, AS THE ALGORITHM REQUIRES IT
    set_nodes_list.sort()


    # PART FOUR - WE CREATE AN EMPTY DICTIONARY WHERE WE CAN STORE
    #             THE Acyclic Directed Graph.
    graph_dict = {x: [] for x in set_nodes_list}

    # PART FIVE - NOTE THAT WE HAVE SAVED THE DATA TO A LIST CALLED DOC.
    #             WE USE A WHILE LOOP TO POP EVEY ITEM IN THE LIST,
    #             BY POPPING EACH ITEM BY LOOKING BACK
    #             WE CAN CREATE THE DICTIONAIRY
    while set_nodes_list:
        a = set_nodes_list.pop(0)
        for i in range(len(doc[1:])):
            if a == doc[1:][i][0]:
                #-> doc[1:][i][0] = begin node
                #-> doc[1:][i][1] = end node
                #-> doc[1:][i][2] = weight of that arc, between the two nodes
                graph_dict[a].append( (doc[1:][i][1], doc[1:][i][2]) )
```

```python
    # PART SIX - THIS PART IS IMPORTANT BECAUSE NOW THAT WE HAVE A DICTIONAIRY
    # /GRAPH, BUT ONLY THE ONES THAT HAVE OUTGOING ARCS. IN ORDER FOR
    # THE ALGORITHM TO WORK, WE HAVE TO ADD EMPTY OR DUMMT ARC, THAT HAVE ZERO
    # WEIGHT/LENGTH AND WHERE THE BEGIN NODE AND END NODE ARE EQUAL.
    # WE ARE GIVEN THAT THERE ARE N-NUMBER OF NODES IN THE TEXT FILE,
    # SO WE RANGE OVER THE NUMBER OF NODES TO SEE WHICH ONES WE MISSM,
    # AND PUT THEM IN A LIST
    missing_nodes = []
    for node in range(1, int(doc[0][0])+1):
        if str(node) not in graph_dict.keys():
            missing_nodes.append(str(node))


    # PART SEVEN - NOW THAT WE HAVE FOUND THE NODES THAT WE ARE MISSING,
    # WE ADD THEM IN THE DICTIONAIRY USING THE TWO FOR LOOPS
    for node in range(1, int(doc[0][0])+1):
        if str(node) in graph_dict.keys():
            pass
        else:
            graph_dict[str(node)] = (str(node), str(0))

    for node in missing_nodes:
        graph_dict[node] = [graph_dict[node]]

    new_dict = {}
    node_list = [str(x) for x in range(1, int(doc[0][0])+1)]
    while node_list:
        a = node_list.pop(0)
        for g in graph_dict.keys():
            if g == a:
                new_dict[a] = graph_dict[a]


    # PART EIGHT - RETURN A DIRECTED ACYCLIC GRAPH
    return new_dict
```

2 ALGORITHM IMPLEMENTATION

Implementing a Reaching Algorithm (Dynamic Program)

The algorithm:
We prodive a simple describtion of the algorithm. Given a Direct Acyclic Graph $G = (N, A)$, where V are the nodes and A are arcs. Let $l(v)$ be the shortest path from s to v. Moreover, each arc has

cost or weight c_{uv}

- Step 1: sort all the nodes where $u < v, (u, v) \in A$
- Step 2: set $l(1) = 0$
- Step 3: for $v = 2, ...n$:
  $l(v) = min_{(u,v)\in A}\{l(u) + c_{uv}\}$

[2]:
```
def min_path_dynamic_program(graph):

    # NOTE THAT STEP 1 WAS DONE PREVIOUSLY IN PART FOUR.

    node_list = list(graph.keys())
    # THIS IS WHERE WE KEEP TRACK OF DISTANCES TRAVELLED TO EACH NODE,
    # AND THEIR PREDECESSORS.

    # STEP 2: INITIALLY WE SET THE ALL DISTANCES TRAVELLED TO ZERO.
    # RATHER THAN ONLY THE FIRST ONE
    min_path = 0
    distance = {str(x): 0 for x in range(1,len(node_list)+1)}
    pred = {str(x):'' for x in range(1,len(node_list)+1)}
    pred['1'] = '1'


    for v in range(1, len(node_list)):
        current_node = node_list[v]

        backtrack_node_dis = []
        min_value_id = ''

        # BACK TRACKING, WE ITERATIVE OVER THE NEXT NODE,
        # BUT LOOK BACKWARDS FOR POSSIBLE PATHS/CONNECTING ARCS
        for u in range(0, v):
            back_track_node = node_list[u]

            for s in graph[back_track_node]:
                s_id = s[0]
                s_value = int(s[1])

                if s_id == current_node:
                    backtrack_node_dis.append((back_track_node,(s_id, s_value)))


        min_distance = []
        for i in backtrack_node_dis:
            min_distance.append(int(i[1][1]))

        min_value = min(min_distance)
```

```python
        for i in backtrack_node_dis:
            if i[1][1] == min_value:
                min_value_id = i[0]


        distance[current_node] = distance[min_value_id] + int(min_value)
        pred[current_node] = min_value_id


    return distance,pred
```

\section(RUNNING FOR ALL FILES)

[3]:
```python
list_of_files = ['P1-10.1.txt',
                 'P1-10.2.txt',
                 'P1-10.3.txt',
                 'P1-15.1.txt',
                 'P1-15.2.txt',
                 'P1-15.3.txt',
                 'P1-20.1.txt',
                 'P1-20.2.txt',
                 'P1-20.3.txt']
```

[4]:
```python
dictionary_of_graphs = {}

for i,file in enumerate(list_of_files):
    dictionary_of_graphs['graph'+str(i)] = convert_text_to_dictionairy(file)
```

[ ]:
```python

```

[ ]:
```python

```

[5]:
```python
list_of_results = []

for g in dictionary_of_graphs.keys():
    pred, dis = min_path_dynamic_program(dictionary_of_graphs[g])
    list_of_results.append((g,pred, dis))
```

[6]:
```python
for result in list_of_results:
    print(result[0])
    print(f'dis: {result[1]}')
    print('\n')
    print(f'pred: {result[2]}')
```

```python
    print('\n')
```

```
graph0
dis: {'1': 0, '2': 4, '3': 10, '4': 18, '5': 12, '6': 14, '7': 18, '8': 10, '9':
17, '10': 19}


pred: {'1': '1', '2': '1', '3': '1', '4': '3', '5': '1', '6': '1', '7': '3',
'8': '2', '9': '1', '10': '9'}


graph1
dis: {'1': 0, '2': 13, '3': 17, '4': 20, '5': 21, '6': 28, '7': 5, '8': 2, '9':
15, '10': 7}


pred: {'1': '1', '2': '1', '3': '1', '4': '2', '5': '3', '6': '5', '7': '1',
'8': '1', '9': '2', '10': '8'}


graph2
dis: {'1': 0, '2': 19, '3': 6, '4': 10, '5': 27, '6': 5, '7': 29, '8': 9, '9':
15, '10': 30}


pred: {'1': '1', '2': '1', '3': '1', '4': '3', '5': '2', '6': '1', '7': '5',
'8': '6', '9': '8', '10': '5'}


graph3
dis: {'1': 0, '2': 10, '3': 2, '4': 6, '5': 15, '6': 21, '7': 8, '8': 23, '9':
23, '10': 8, '11': 10, '12': 3, '13': 13, '14': 25, '15': 10}


pred: {'1': '1', '2': '1', '3': '1', '4': '3', '5': '3', '6': '5', '7': '3',
'8': '6', '9': '6', '10': '4', '11': '4', '12': '3', '13': '10', '14': '9',
'15': '7'}


graph4
dis: {'1': 0, '2': 3, '3': 3, '4': 6, '5': 10, '6': 6, '7': 13, '8': 7, '9': 7,
'10': 9, '11': 17, '12': 3, '13': 4, '14': 20, '15': 4}


pred: {'1': '1', '2': '1', '3': '1', '4': '3', '5': '2', '6': '3', '7': '6',
'8': '4', '9': '2', '10': '8', '11': '7', '12': '1', '13': '12', '14': '11',
'15': '2'}
```

```
graph5
dis: {'1': 0, '2': 7, '3': 15, '4': 15, '5': 8, '6': 19, '7': 17, '8': 22, '9':
10, '10': 16, '11': 25, '12': 18, '13': 17, '14': 5, '15': 18}


pred: {'1': '1', '2': '1', '3': '2', '4': '1', '5': '2', '6': '3', '7': '4',
'8': '7', '9': '5', '10': '9', '11': '6', '12': '3', '13': '10', '14': '1',
'15': '10'}


graph6
dis: {'1': 0, '2': 14, '3': 18, '4': 27, '5': 30, '6': 40, '7': 20, '8': 41,
'9': 32, '10': 28, '11': 30, '12': 27, '13': 34, '14': 44, '15': 44, '16': 21,
'17': 25, '18': 22, '19': 32, '20': 36}


pred: {'1': '1', '2': '1', '3': '2', '4': '2', '5': '4', '6': '5', '7': '3',
'8': '4', '9': '5', '10': '7', '11': '4', '12': '7', '13': '10', '14': '8',
'15': '8', '16': '7', '17': '7', '18': '16', '19': '10', '20': '13'}


graph7
dis: {'1': 0, '2': 6, '3': 19, '4': 9, '5': 4, '6': 7, '7': 6, '8': 9, '9': 11,
'10': 11, '11': 8, '12': 11, '13': 10, '14': 11, '15': 11, '16': 10, '17': 12,
'18': 20, '19': 8, '20': 6}


pred: {'1': '1', '2': '1', '3': '2', '4': '2', '5': '1', '6': '1', '7': '5',
'8': '5', '9': '4', '10': '7', '11': '7', '12': '6', '13': '8', '14': '11',
'15': '11', '16': '5', '17': '12', '18': '3', '19': '7', '20': '5'}


graph8
dis: {'1': 0, '2': 14, '3': 16, '4': 26, '5': 32, '6': 33, '7': 2, '8': 29, '9':
5, '10': 18, '11': 3, '12': 16, '13': 18, '14': 21, '15': 3, '16': 5, '17': 20,
'18': 36, '19': 22, '20': 22}


pred: {'1': '1', '2': '1', '3': '2', '4': '3', '5': '4', '6': '5', '7': '1',
'8': '4', '9': '7', '10': '3', '11': '7', '12': '2', '13': '3', '14': '10',
'15': '7', '16': '11', '17': '13', '18': '6', '19': '13', '20': '17'}
```

[ ]:

[ ]:

```

```

[ ]:

```

```