Illustrative answers for Assessed coursework 2 for Algorithms and Machine Learning (MATH20017), Autumn 2024

Introduction

This document contains the questions for Part 1 of your assessed coursework for the unit Algorithms and Machine Learning (MATH20017). The marks for this coursework will count 10% towards your final grade.

Please contact henry.reeve@bristol.ac.uk with any questions regarding this document. Whilst I am unable to provide solutions in advance of all work being handed in, I can provide clarification.

The contents of this document should **not** be distributed without permission.

Handing in your coursework

How you present your coursework is important. You should complete your coursework using either Google Colab, a Jupyter notebook, or Rmarkdown. Whichever approach you take you must submit a compressed folder containing all of the following:

(1) A pdf file in which all answers within your notebook are visible.
(2) The notebook used to create the file (either a `.ipynb` file or a `.rmd` file).
(3) Any other files required to create the pdf in (1) e.g. images.

Important: Answers lacking any one of (1),(2),(3) are missing may not be awarded marks.

For answers using mathematical notation you have two options:

1. Include your answers in the Python notebook directly via Tex;
2. Write your answers via pen and paper and include a photo within the final document.

Important: Please ensure that the answers are well presented and clearly readable. Failure to do so can lead to a substantial loss of marks.

The graph class

We shall make use of an adjacency list data structure for a directed graph graph $G = (V, E)$. Hence, we begin by introducing the following `Graph` class. You may wish to begin by checking you understand the class and adding additional comments where useful.

```python
class GraphNode:
    # simple graph node class for adjacency list representation
    def __init__(self, name):
        self.name = name
        self.adjacency_list=[]

    # enable nodes to be compared with one another
    def __lt__(self, other):
        return True
    def __le__(self,other):
        return True


class GraphEdge:
      # graph edge class with from and to nodes

      def __init__(self,from_node,to_node):
          self.from_node= from_node
          self.to_node = to_node

class Graph:
      # simple graph class for adjacency list representation
      def __init__(self):
          self.num_nodes=0
          self.num_edges=0
          self.node_array=[]
          self.node_dictionary={}

      def add_node(self,name):

          new_node=GraphNode(name)
          self.node_array.append(new_node)
          self.node_dictionary[name]=new_node
          self.num_nodes+=1

      def print_nodes(self):

          if self.num_nodes==0:
              print("Empty graph")

          for i in range(self.num_nodes):
              print(self.node_array[i].name)

      def add_edge_by_node(self,from_node,to_node):

          from_node.adjacency_list.append(GraphEdge(from_node,to_node))
          self.num_edges+=1
      def add_edge(self,from_name,to_name):

          from_node=self.node_dictionary[from_name]
          to_node=self.node_dictionary[to_name]
          self.add_edge_by_node(from_node,to_node)

      def print_graph(self):

          if self.num_nodes==0:
              print("Empty graph")
              print("The graph structure is:")

          for i in range(self.num_nodes):
              node=self.node_array[i]
              print("{node} : {direct_descendants}".format(
                  node=node.name,direct_descendants=[
                  edge.to_node.name for edge in node.adjacency_list]))
```

We shall also make use of the graph_from_adjacency_matrix function which takes as input a vector names for the nodes with graph and an adjacency matrix and returns a corresponding graph object, which is an instance of our Graph class.

```
import numpy as np

def graph_from_adjacency_matrix(names_array:list,adjacency_matrix:np.array):

    # function which takes as input a name vector and an adjacency matrix
    # and returns a Graph with adjacency list representation

    # check inputs
    n=len(names_array)
    if adjacency_matrix.shape!=(n,n):
        print("Incongruent inputs")
        return None

    # initialise graph
    output_graph=Graph()

    # add nodes with names given in name vector
    for name in names_array:
        output_graph.add_node(str(name))

    # add edges based upon adjacency matrix
    for i in range(len(names_array)):
        for j in range(len(names_array)):
            if adjacency_matrix[i,j]>0:
                output_graph.add_edge(str(names_array[i]),str(names_array[j]))

    return output_graph
```

Question 1 (10 marks)

In this question we consider the breadth-first-search algorithm. We shall make use of *doubly-ended queue* data structure from the `collections` Python library.First import the `collections` library.

```
from collections import deque
```

Note that you can initialise a new empty doubly-ended `deque([])`. With this data-structure structure we can add an element to the right-end of the queue via the `append` method. We can also remove and return an element from the left-end of the queue via the `popleft()` method. Both of these operations are (typically) $\Theta(1)$ time.

Q1(a) Create a function called `bfs_reachable_nodes` which implements a variant of the breadth-first-search algorithm. Your function should takes as input a graph object G belonging to the `Graph` class implemented above, along with a string `initial_node_name` corresponding to the name of a node within the graph, which we shall refer to as the initial node. The function should return a Python list containing all of the nodes (instances of the `GraphNode` object) within the graph which are reachable from the initial node.

You can begin with the following outline code:

```
def bfs_reachable_nodes(input_graph:Graph,initial_node_name:str)-> list:

    # initialisation

    # find initial node by name and mark all other nodes as not yet visited
    initial_node = None
    for node in input_graph.node_array:
        if node.name==initial_node_name:
            initial_node=node
            node.visited=True
        else:
          node.visited=False

    # exit if the inital node isn't found
    if initial_node is None:
        print("Start node {sn} not found. ".format(sn=initial_node_name))
        return None

    # initialise a list of reachable nodes from the initial node
    list_of_reachable_nodes=[initial_node]

    # initialise queue data structure
    node_queue = deque([initial_node])

    ### EDIT CODE HERE

    return list_of_reachable_nodes
```

Use the following code to test your function.

```
np.random.seed(2024) # set random seed
num_tests=10
num_nodes=10
```

```python
names_v=[chr(ord('A') + i) for i in range(num_nodes)]

for i in range(num_tests):

    adj_mat=np.random.choice([0, 1], size=(num_nodes,num_nodes), p=[0.8,0.2])
    G=graph_from_adjacency_matrix(names_v,adj_mat)
    reachable_nodes=[node.name for node in bfs_reachable_nodes(G,"A")]
    print(f"There are {len(reachable_nodes)} reachable nodes: {', '.join(reachable_nodes)}")
```

Q1(b) Name an application of the breadth-first-search algorithm.

Question 2 (20 marks)

In this question we focus on the depth-first-search algorithm.

Q2(a) Create a function called `dfs_reachable_nodes` which implements a variant of the depth-first-search algorithm. Your function should have the same input and output structure as in the previous question: The input is a graph object G belonging to the `Graph` class implemented above, along with a string `initial_node_name` corresponding to the name of a node within the graph, which we shall refer to as the initial node. The function should return a Python list containing all of the nodes (instances of the `GraphNode` object) within the graph which are reachable from the initial node.

You can begin with the following outline code:

```python
def dfs_reachable_nodes(input_graph:Graph,initial_node_name:str)-> list:

    # find initial node by name and mark all other nodes as not yet visited
    initial_node = None
    for node in input_graph.node_array:
        if node.name==initial_node_name:
            initial_node=node
            node.visited=True
        else:
          node.visited=False

    # exit if the inital node isn't found
    if initial_node is None:
        print("Start node {sn} not found. ".format(sn=initial_node_name))
        return None

    # initialise a list of reachable nodes from the initial node
    list_of_reachable_nodes=[initial_node]

    ### EDIT CODE HERE!

    return list_of_reachable_nodes
```

Run the following code to test your function.

```python
np.random.seed(2024) # set random seed
num_tests=10
num_nodes=10
names_v=[chr(ord('A') + i) for i in range(num_nodes)]

for i in range(num_tests):
    adj_mat=np.random.choice([0, 1], size=(num_nodes,num_nodes), p=[0.8,0.2])
    G=graph_from_adjacency_matrix(names_v,adj_mat)
    reachable_nodes=[node.name for node in dfs_reachable_nodes(G,"A")]
    print(f"There are {len(reachable_nodes)} reachable nodes: {', '.join(reachable_nodes)}")
```

For the next part of the question you may wish to utilise Python's doubly-linked list data structure, the `doubly-ended` queue from Pythons `collections` library.

```
from collections import deque
```

You can create an empty doubly-linked-list as follows:

```
A=deque([])
```

We can add an element to the left end of the queue via the `appendleft()` method, or add an element to the right end via the `append()` method. We can remove and return an element from the left-end of the queue via the `popleft()` method, or from the right end of the queue via the `pop()` method. All of these operations are (typically) $\Theta(1)$ time. You can convert an instance a of the `deque` class to a Python `list` by calling `list(a)`, with time complexity $\Theta(n)$ where n denotes the number of items in a.

Q2(b) Write a Python called `topological_sort` which takes as input a graph object G belonging to the `Graph` which represents directed acyclic graph. Your function should return a standard Python list corresponding to a topological ordering of the nodes. That is, the returned list should contain every node within the graph exactly once. Moreover, the ordering of the nodes within the returned list should be a topological ordering, so if there is an edge $e = (u, w)$ in the graph, then the node w should not occur earlier than the node u within the list.

Run the following test code to check your function.

```
np.random.seed(2024) # set random seed
for j in range(6,30,4):
  names_v=[chr(ord('A') + i) for i in range(j)]
  np.random.shuffle(names_v)
  adj_mat=np.triu(np.ones((j,j)),1)
  G=graph_from_adjacency_matrix(names_v,adj_mat)
  np.random.shuffle(G.node_array)
  top_order=topological_sort(G)
  print(*(node.name for node in top_order), sep=',')
```

Q2(c) Create a function called `topological_sort_DAG_check` which takes as input a graph object G belonging to the `Graph` which represents directed graph. If the input graph object G is acyclic then the function should return a standard Python list corresponding to a topological ordering of the nodes, as in the previous question. However, if the graph object G contains a cycle then the Boolean value `False` should be returned instead

Run the following test code to check your function.

```
np.random.seed(2024) # set random seed
for j in range(9,30):
  names_v=[chr(ord('A') + i) for i in range(j)]
  np.random.shuffle(names_v)
  adj_mat=np.triu(np.ones((j,j)),1)
  if j % 3==0:
    adj_mat[-1,0]=1
  G=graph_from_adjacency_matrix(names_v,adj_mat)
  np.random.shuffle(G.node_array)
  top_order=topological_sort_DAG_check(G)
  if top_order:
      print(f"DAG: {', '.join([node.name for node in top_order])}")
  else:
      print("Cycle found!")
```

Question 3 (20 marks)

In this question we your objective will be to solve a problem involving computing the area of the largest island on a map.

The map is represented by an $n \times n$ NumPy matrix called `grid_map` consisting of zeroes and ones. For each $(i, j) \in \{0, \ldots, n-1\}^2$, `grid_map[i,j]` corresponds to a square metre on the map with the interpretation that `grid_map[i,j]=0` means that the corresponding square metre is mostly water, and `grid_map[i,j]=1` means that the corresponding square metre is mostly land. The outer square of our grid map will consist entirely of water, i.e. `grid_map[i,j]=0` whenever $i \in \{0, n-1\}$ or $j \in \{0, n-1\}$.

Let's now introduce some terms:

Given a pair of grid squares (i, j), $(i', j') \in \{0, \ldots, n-1\}^2$ we shall say there is *a valid move on land* from (i, j) to (i', j') if they are both mostly land (i.e. `grid_map[i,j]=grid_map[i',j']=1`) and $(i', j') \in \{(i-1, j), (i+1, j), (i, j-1), (i, j+1)\}$ (i.e. a move "west", "east", "south" or "north").

By an *island* we mean a maximal collection of grid squares $S \subseteq \{0, 1, \ldots, n-1\}^2$ such that:

- Every $(i, j) \in S$ consists mostly of land i.e. `grid_map[i,j]=1`;
- Given any pair of grid squares on the island a, $b \in S$ we can reach b from a through a sequence of valid moves on land. Thus, there is are tuples $(i_0, j_0), \ldots, (i_q, j_q) \in S$ such that $a = (i_0, j_0)$, $b = (i_q, j_q)$ and for all $k \in \{0, 1, \ldots, q-1\}$ we there is a valid move on land from (i_k, j_k) to (i_{k+1}, j_{k+1}).

The *area* of an island S is the number of grid squares within the island (i.e. the cardinality of S).

Problem: Compute the maximal area of an island

Input: An $n \times n$ NumPy matrix consisting of zeroes and ones called `grid_map`.

Output: The maximum area of a single island within `grid_map`.

Q3. Write a function called `compute_max_island_area` which solves the problem of computing the maximal area of an island within the grid map.

You may wish to begin with the following skeleton code.

```python
def compute_max_island_area(grid_map : np.array)-> int:

    max_island_area=0

    # edit code here

    return max_island_area
```

To check our `compute_max_island_area` function we create the following function to create random grid maps.

```python
from scipy import signal

def random_grid_map(grid_width, land_propensity=0.5,interaction_strength=3):
    # generate random grid map
    grid_map=np.random.choice([0, 1],
      size=(grid_width,grid_width), p=[1-land_propensity,land_propensity])

    grid_map = (0.5+signal.convolve2d(grid_map,
                                      np.ones((interaction_strength,
                                               interaction_strength)))/(
                                                   interaction_strength**2
                                                   )).astype(int)



    # ensure outer squares are all land
    grid_map[0,:]=0
    grid_map[-1,:]=0
    grid_map[:,0]=0
    grid_map[:,-1]=0

    return grid_map
```

As an example, our compute_max_island_area function should behave as follows.

```
np.random.seed(2024) # set random seed

grid_map=random_grid_map(6)

print("The grid map looks as follows:")
```

```
## The grid map looks as follows:
```

```
print(grid_map)
```

```
## [[0 0 0 0 0 0 0 0]
##  [0 0 0 0 0 0 0 0]
##  [0 1 1 1 0 0 0 0]
##  [0 1 1 1 0 0 0 0]
##  [0 1 1 1 1 0 0 0]
##  [0 0 1 1 0 0 0 0]
##  [0 0 0 0 0 0 0 0]
##  [0 0 0 0 0 0 0 0]]
```

```
print("""\nBy applying our function we see that the largest island has area {mx_area}.""".format(
    mx_area=compute_max_island_area(grid_map)))
```

```
##
## By applying our function we see that the largest island has area 12.
```

Now test your compute_max_island_area function by applying the following test code.

```
np.random.seed(2024) # set random seed

num_lakes_list=[]
```

```
for i in range(15):

  grid_map=random_grid_map(100, land_propensity=0.45, interaction_strength=5)
  num_lakes_list.append(compute_max_island_area(grid_map))

print(num_lakes_list)
```

1 Question 4 (25 marks)

In the next two questions we consider greedy algorithms.

Q4(a) Name two examples of greedy algorithms.

Suppose each edge in your graph has an attribute edge_length. We then the *Euclidean norm* of a path $p = (e_0, e_1, \ldots, e_{q-1})$ to be the quantity

$$\text{p.euclidean\_norm} := \sqrt{e_0.\text{edge\_length}^2 + e_1.\text{edge\_length}^2 + \ldots + e_{q-1}.\text{edge\_length}^2}.$$

Q4(b) Create a function called dijkstras_min_euclidean_norm_path which takes as input an adjacency list representation input_graph for a graph in which every edge within the graph has a non-negative edge.edge_length attribute, as well as an initial node object initial_node. Your function should modify the existing graph data structure in-place so that every node node has an additional attribute node.min_euclidean_norm_path_from_start which is equal to the minimum Euclidean norm of a path from the initial node initial_node to the node object itself. If there is no path from initial_node to node then we have node.min_euclidean_norm_path_from_start=np.Inf. We also define the Euclidean norm of the empty path to be 0 so that node.min_euclidean_norm_path_from_start=0.

Ensure that your function has a worst-case run-time of at most $O(n + m\log(n))$ where n and m denote, respectively, the number of nodes and edges within the graph.

You may wish to use the heapq Python library or use the MinHeap class you built in a previous lab if you prefer.

```
import heapq
```

You may wish to utilise the NumPy functions np.sqrt() along with np.Inf and the usual square operation **2.

To test the dijkstras_min_euclidean_norm_path function we first create a function assign_random_edge_lengths to assign edge lengths to a graph.

```
def assign_random_edge_lengths(graph:Graph,edge_length_max:int,
    random_seed=0):

    np.random.seed(random_seed) # set random seed

    for i in range(graph.num_nodes):
        node=graph.node_array[i]
        for edge in node.adjacency_list:
            edge.edge_length=np.random.randint(edge_length_max)+1
```

We also add a function to print out the minimum norms paths for a given graph.

```
def print_dijkstras_min_euclidean_norm_path_algorithm_output(
    graph:Graph,initial_node_name:str):

    initial_node=graph.node_dictionary[initial_node_name]

    dijkstras_min_euclidean_norm_path(graph,initial_node)
    for i in range(graph.num_nodes):
        node=graph.node_array[i]
        print("{node} : {direct_descendants}".format(
          node=node.name,
          direct_descendants=[(edge.to_node.name,edge.edge_length
          ) for edge in node.adjacency_list]))

    for i in range(graph.num_nodes):
        node=graph.node_array[i]
        print(
          "The minimum norm path from node "+
          "{init_node} to node {node} is {norm} ".format(
          node=node.name,norm=node.min_euclidean_norm_path_from_start,
          init_node=initial_node_name))
```

Now run the test code to evaluate the performance of your `dijkstras_min_euclidean_norm_path` function.

```
np.random.seed(2024) # set random seed

num_nodes=20

adj_mat=np.random.choice([0, 1], size=(num_nodes,num_nodes), p=[0.7,0.3])
names_v=[chr(ord('A') + i) for i in range(num_nodes)]

G=graph_from_adjacency_matrix(names_v,adj_mat)

assign_random_edge_lengths(G,10)

print_dijkstras_min_euclidean_norm_path_algorithm_output(G,"A")
```

Q4(c) Justify the claim that once the algorithm implemented by the `dijkstras_min_euclidean_norm_path` function has terminated, for every node in the graph the attribute `node.min_euclidean_norm_path_from_start` will be equal to the minimum Euclidean norm of a path from the initial node `initial_node` to the node object itself.

2 Question 5 (25 marks)

In this problem we consider undirected graphs with n nodes $V = \{0, 1, 2, \ldots, n-1\}$. Let's write $E_{\max}$ for the set of all subsets of the form $\{i, j\} \subseteq V$ with $i < j$. Hence, $E_{\max}$ corresponds to the set of all $n(n-1)/2$ possible undirected edges between nodes V. For each subset $E' \subseteq E_{\max}$ there is a corresponding graph $G' = (V, E')$ with edges in E'.

Q5(a) How many possible graphs are there of the form $G' = (V, E')$ where $E' \subseteq E_{\max}$?

Let's introduce some further concepts. We shall say that the graph $G' = (V, E')$ is *connected* if for any pair of nodes a,$b \in V$, there exists a sequence of edges $e_0, e_1, \ldots, e_{q-1} \in E'$ with a corresponding vertex sequence of vertices $v_0, v_1, \ldots, v_{q-1}, v_q$ such that $a = v_0$, $b = v_q$ and for all $\ell \in \{0, 1, \ldots, q-1\}$, the edge $e_j = \{v_j, v_{j+1}\} \in E'$ goes from v_j to v_{j+1}.

Suppose we have a symmetric $n \times n$ matrix `edge_cost_matrix` with strictly positive entries. For each pair $i, j \in \{0, 1, \ldots, n-1\}$ the quantity `edge_cost_matrix[i,j]` denotes the *edge cost* of an edge from node i to j. Since the graph is symmetric this is equal to the *edge cost* `edge_cost_matrix[j,i]` of an edge from node j to node i.

Given a graph $G' = (V, E')$ the *graph cost* of G' is equal to the sum of the edge costs of the edges $\{i, j\}$ within E' i.e.

$$\text{graph\_cost}(G') := \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \text{edge\_cost\_matrix}[i,j] \cdot \mathbb{1}\left\{\{i,j\} \in E'\right\}.$$

Your objective is to construct an efficient method for determining the minimal cost of a connected undirected graph for a particular edge cost matrix.

Q5(b) Write a function called `min_graph_cost` which takes as input an $n \times n$ symmetric NumPy matrix `edge_cost_matrix` with strictly positive entries and returns a number corresponding to the minimum graph cost $\text{graph\_cost}(G')$ where $G' = (V, E')$ is a connected graph.

Your function should have a worst case time complexity no more than $O(n^2 \log(n))$.

You are encouraged to adapt ideas from Dijkstra's shortest path algorithm.

You may wish to use the `heapq` Python library.

Run the following test code to check your function.

```python
np.random.seed(2024) # set random seed

for n in range(3,31,3):

    edge_cost_matrix=np.random.randint(low=1,high=25, size=(n,n))
    min_cost=min_graph_cost(edge_cost_matrix)
    print(f"The minimum cost for this {n} by {n} edge cost matrix was {min_cost}.")
```

Q5(c) Justify the claim that the algorithm implemented by `min_graph_cost` is guaranteed to return the minimal cost of a connected undirected graph for the given input `edge_cost_matrix`.

End of coursework.