



HIT220 ALGORITHMS AND COMPLEXITY

Water Network Assignment 3 Part C

Group 51:

Member

Muhammed Abed ID: s361222

Thi Minh Chau Pham ID:s361375

Manish Rijal ID: s366267

Supin Shrestha :s360911

Question 1:

What cycles exist in the graph and how can you find them from code. Provide the *function* `list_cycles()` -> [List of nodes, List2 of nodes, etc]

Answer

Assumption

- Data Loading: The code reads data from a CSV file using Pandas.
- Data Processing: It processes the data to create an adjacency list representing a graph.
- Graph Structure: The graph is represented as a Python dictionary with nodes and their linked nodes.
- Data Format Assumption: Assumes the input CSV file has 'Node', 'type', and 'linked' columns. Data from a CSV file called 'water_data.csv' ,(attached in the submission),

Description of how the problem solved

- The answer uses an algorithm based on Depth-First Search to identify cycles in a graph.
- **'visited'** dictionary marks which nodes have been visited. The visited dictionary helps ensure each node is explored only once
- The stack keeps track of the path taken during the exploration.
- When a node is found that is both in the stack and has already been visited, a cycle is identified.
- From the expected output, the code seems to have identified four cycles in the graph. This suggests that the provided data has some interlinked nodes, causing cyclic paths.

Code

```
import pandas as pd

# Read data from the CSV file
data_path = 'water_data.csv'
water_data_df= pd.read_csv(data_path)

# Process data and generate a graph
def process_data(data_frame):

    # Initialize an empty dictionary 'graph' to represent the adjacency list.
    graph = {}
    # Iterate through each row in the DataFrame.
    # Extract 'Node', 'type', and 'linked' values from the row.
    for index, row in data_frame.iterrows():
        node = int(row['Node'])
        node_type = row['type']
```

Group 51

```
    linked = int(row['linked'])
    # If 'type' is 'headwater', initialize an empty list for the node in the 'graph' dictionary.
    graph[node] = [] if node_type == 'headwater' else graph.get(node, [])
    # If 'linked' is not 0, add 'node' to the list of the linked node in the 'graph' dictionary.
    graph.setdefault(linked, []).append(node) if linked != 0 else None
    # Return the 'graph' dictionary.
    return graph

# Function to extract and list cycles in the graph
def list_cycles(graph):

    # The 'dfs' function recursively explores the graph and detects cycles by checking if a node is visited
    # and is already in the stack.
    def dfs(node, visited, stack):
        # Initialize a 'visited' dictionary to keep track of visited nodes.
        # Initialize an empty 'stack' to keep track of the current path.
        visited[node] = True
        stack.append(node)
        # Iterate through each node in the 'graph'
        # If the node is not visited, call the 'dfs' function to search for cycles starting from that node.
        for neighbor in graph[node]:
            if not visited[neighbor]:
                if dfs(neighbor, visited, stack):
                    return True
            # Append any detected cycles to the 'cycles' list.
            # Return the list of cycles
            elif neighbor in stack:
                cycle = stack[stack.index(neighbor):]
                cycles.append(cycle)

        stack.pop()
        return False

    visited = {node: False for node in graph}
    cycles = []

    for node in graph:
        dfs(node, visited, []) if not visited[node] else None
    return cycles

#Process the data
graph = process_data(water_data_df)

# Find and extract cycles in the graph
graph_cycles = list_cycles(graph)

print("Cycles Identified in the data:", graph_cycles)
```

Group 51

```
"""
Expected Output:
    Cycles Identified in the data: [[33, 34, 59, 35, 2, 36, 37, 39, 38, 49, 46, 50], [61, 62], [33, 34, 59, 35,
2, 36, 37, 39, 38, 49, 46, 50, 61, 62], [50, 63]]
    Identified 4 cycles
"""
```

Output

```
[Running] python -u "/Users/phamthiminhchau/Documents/Algorithms and Complexity/Group assignment 3.3/question 1.py"
Cycles Identified in the data: [[33, 34, 59, 35, 2, 36, 37, 39, 38, 49, 46, 50], [50, 61], [61, 62], [33, 34, 59, 35, 2, 36, 37, 39, 38, 49, 46, 50, 61, 62], [50, 63]]
```

Question 2:

We are flying a drone over the country and want to check all the nodes (junctions and headwaters) in any area. We will start at top right point of area. Give the order of nodes we should flyover to see all nodes in shortest time.

Explain what method you use.

Provide flight path $((x_1, y_1), (x_2, y_2)) \rightarrow \text{List of nodes}$

Assumption

Data from a CSV file called 'nodes.csv' , (attached in the submission). Assumes the input CSV file has 'Node', 'type', 'x' coordinate, and 'y' coordinate columns

The Euclidean distance, d , between the two points is given by the formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Method Used

1. The Nearest Neighbor Algorithm is the method used in the Python code to find the order of nodes that the drone can visit in the least amount of time.
2. The code starts with an initial node (in the top right spot) and an empty flight path.

3. At each step, the algorithm chooses the unvisited node that is closest to the present node in terms of Euclidean distance. It figures out how far each unvisited node is from the current node and picks the one that is the closest.
4. Path Construction: The nearest node chosen is added to the flight path, and the process keeps going by choosing the next closest node that hasn't been visited yet. This continues until all nodes have been visited.
5. Path Completion: When the programmer has gone to all the nodes, it adds the starting point back to the path to make a closed loop. It also figures out how far it is to go around the loop.
6. The code shows the flight path, including information about each node, its coordinates, and the distance between each node. It also shows the total distance travelled, which shows which path was the quickest.

Description of how the problem solved (Steps for the Python Code)

1. It starts by reading data from a CSV file called 'nodes.csv' and stores this data in a list.
2. The code assumes there are points with coordinates in the CSV file and designates a starting point.
3. It calculates the distance between two points using a formula.
4. Filters the points to keep only those labelled as 'junction' or 'headwater.'
5. Initializes a path and distance variable to keep track of the best route.
6. The code uses a method called the Nearest Neighbour Algorithm to find the shortest path that visits all the filtered points and returns to the starting point.
7. It prints out the path, showing each point's type and coordinates, along with the distance to the next point.
8. Finally, it prints the total distance of the entire path.

Code

```
import csv
import math

# Read data from the CSV file and store it as a list of tuples
nodes = []
with open('nodes.csv', mode='r') as csv_file:
    csv_reader = csv.reader(csv_file)
    next(csv_reader) # Skip the header row
    for row in csv_reader:
        if len(row) >= 4: # Check if the row has at least 4 columns
            node_type = row[0]
            x = float(row[2])
            y = float(row[3])
```

```

        nodes.append((node_type, x, y))

# Define the starting point as the top-right corner
start_x, start_y = 70, 100

# Function to calculate Euclidean distance between two points
def calculate_distance(x1, y1, x2, y2):
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

# Filter nodes to include only junctions and headwaters
filtered_nodes = [(node_type, x, y) for node_type, x, y in nodes if node_type in ('junction', 'headwater')]

# Initialize variables to track the best path and its total distance
best_path = [filtered_nodes[0]] # Start with the first node
best_distance = 0

# Function to find the nearest unvisited node
def find_nearest_unvisited(current_node, unvisited_nodes):
    min_distance = float('inf')
    nearest_node = None

    for node in unvisited_nodes:
        distance = calculate_distance(current_node[1], current_node[2], node[1], node[2])
        if distance < min_distance:
            min_distance = distance
            nearest_node = node

    return nearest_node

# Build the path using the Nearest Neighbor Algorithm
unvisited_nodes = filtered_nodes[1:] # Exclude the first node (already added)
while unvisited_nodes:
    current_node = best_path[-1]
    nearest_node = find_nearest_unvisited(current_node, unvisited_nodes)

    if nearest_node:
        best_distance += calculate_distance(current_node[1], current_node[2], nearest_node[1],
nearest_node[2])
        best_path.append(nearest_node)
        unvisited_nodes.remove(nearest_node)

# Add the distance back to the starting point
best_distance += calculate_distance(best_path[-1][1], best_path[-1][2], start_x, start_y)
best_path.append(('sea entrance', start_x, start_y)) # Add the starting point to the end of the path

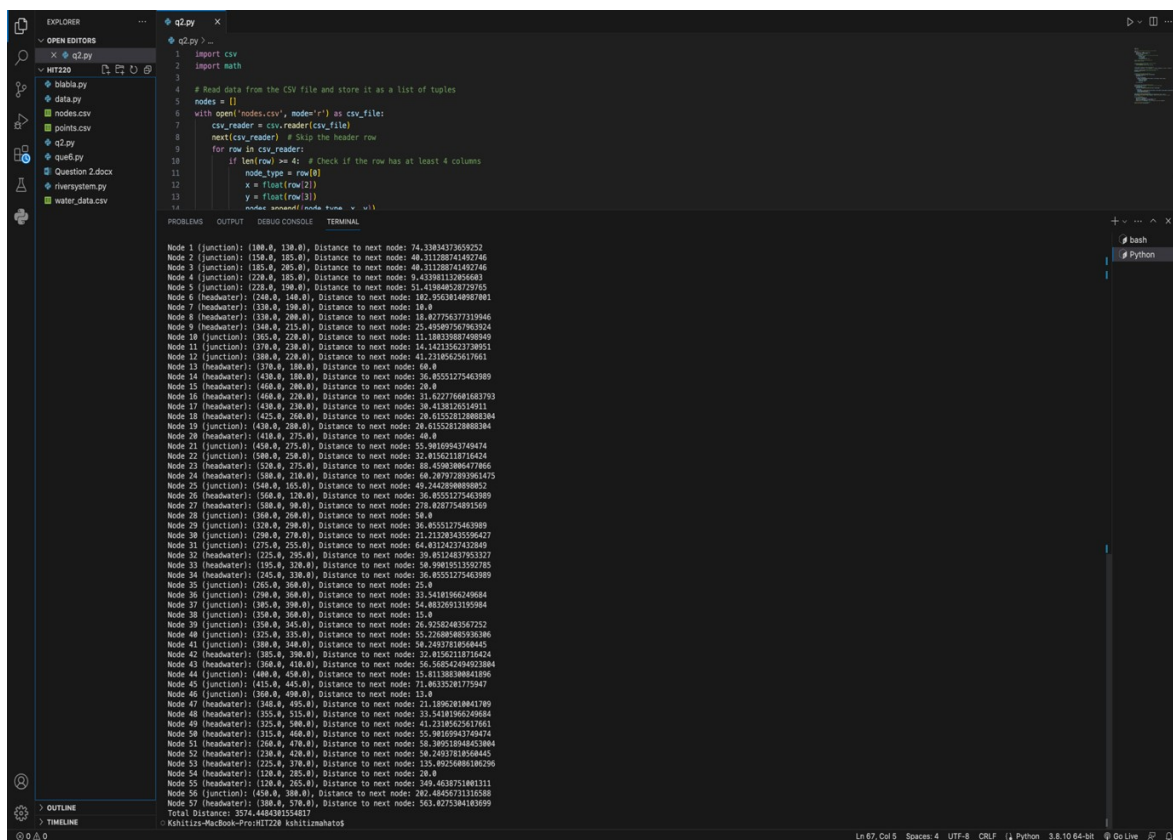
```

Group 51

```
# Print the flight path
for i in range(len(best_path) - 1):
    node_type, x, y = best_path[i]
    next_x, next_y = best_path[i + 1][1], best_path[i + 1][2]
    distance = calculate_distance(x, y, next_x, next_y)
    print(f"Node {i + 1} ({node_type}): ({x}, {y}), Distance to next node: {distance}")

# Print the total distance of the flight path
print(f"Total Distance: {best_distance}")
```

Output



The screenshot shows a code editor with a file explorer on the left and a terminal at the bottom. The file explorer shows a project named 'q2.py' with files like 'nodes.csv', 'points.csv', 'q2.py', 'Question 2.docx', 'riversystem.py', and 'water_data.csv'. The code editor shows a Python script that reads data from 'nodes.csv' and prints the flight path and total distance. The terminal shows the output of the script, which is a list of nodes and their distances to the next node, followed by the total distance.

```
Node 1 (junction): (380.0, 130.0), Distance to next node: 74.33834373659252
Node 2 (junction): (350.0, 185.0), Distance to next node: 40.3128871492746
Node 3 (junction): (355.0, 255.0), Distance to next node: 40.3128871492746
Node 4 (junction): (220.0, 185.0), Distance to next node: 9.43398113295468
Node 5 (junction): (220.0, 190.0), Distance to next node: 51.41040287379765
Node 6 (headwater): (240.0, 140.0), Distance to next node: 102.95038149087801
Node 7 (headwater): (330.0, 190.0), Distance to next node: 18.0
Node 8 (headwater): (330.0, 200.0), Distance to next node: 15.827756177319946
Node 9 (headwater): (340.0, 215.0), Distance to next node: 25.49097567963924
Node 10 (junction): (305.0, 220.0), Distance to next node: 11.18033927490949
Node 11 (junction): (370.0, 230.0), Distance to next node: 14.52135622709551
Node 12 (junction): (380.0, 220.0), Distance to next node: 41.2310525617661
Node 13 (headwater): (370.0, 180.0), Distance to next node: 60.0
Node 14 (headwater): (430.0, 180.0), Distance to next node: 36.05551275463989
Node 15 (headwater): (460.0, 200.0), Distance to next node: 20.0
Node 16 (headwater): (460.0, 220.0), Distance to next node: 31.62277681683793
Node 17 (headwater): (430.0, 230.0), Distance to next node: 30.4138126514911
Node 18 (headwater): (425.0, 240.0), Distance to next node: 20.015252120880384
Node 19 (junction): (430.0, 200.0), Distance to next node: 26.015252120880384
Node 20 (headwater): (410.0, 275.0), Distance to next node: 40.0
Node 21 (junction): (450.0, 275.0), Distance to next node: 55.90109943740474
Node 22 (junction): (500.0, 250.0), Distance to next node: 32.01562118716424
Node 23 (headwater): (520.0, 275.0), Distance to next node: 68.4090308477806
Node 24 (headwater): (500.0, 250.0), Distance to next node: 60.207927093061475
Node 25 (junction): (540.0, 165.0), Distance to next node: 49.24428908908852
Node 26 (headwater): (500.0, 120.0), Distance to next node: 36.05551275463989
Node 27 (headwater): (500.0, 90.0), Distance to next node: 270.828754091569
Node 28 (junction): (360.0, 260.0), Distance to next node: 50.0
Node 29 (junction): (520.0, 260.0), Distance to next node: 36.05551275463989
Node 30 (junction): (200.0, 270.0), Distance to next node: 21.21208435596427
Node 31 (junction): (275.0, 255.0), Distance to next node: 64.0324274232045
Node 32 (headwater): (225.0, 265.0), Distance to next node: 50.0324274232045
Node 33 (headwater): (135.0, 320.0), Distance to next node: 50.99015513529285
Node 34 (headwater): (265.0, 320.0), Distance to next node: 36.05551275463989
Node 35 (junction): (265.0, 360.0), Distance to next node: 25.0
Node 36 (junction): (200.0, 360.0), Distance to next node: 33.54101964249684
Node 37 (junction): (360.0, 360.0), Distance to next node: 56.0126513350984
Node 38 (junction): (330.0, 360.0), Distance to next node: 15.0
Node 39 (junction): (320.0, 245.0), Distance to next node: 26.02024033657252
Node 40 (junction): (325.0, 335.0), Distance to next node: 55.22408090593646
Node 41 (junction): (380.0, 340.0), Distance to next node: 50.24937810540445
Node 42 (headwater): (385.0, 300.0), Distance to next node: 52.01562118716424
Node 43 (headwater): (360.0, 410.0), Distance to next node: 56.508542494923804
Node 44 (junction): (480.0, 450.0), Distance to next node: 15.8138030843196
Node 45 (junction): (415.0, 445.0), Distance to next node: 71.0613521775947
Node 46 (junction): (380.0, 490.0), Distance to next node: 13.0
Node 47 (headwater): (500.0, 450.0), Distance to next node: 21.00523084181700
Node 48 (headwater): (355.0, 515.0), Distance to next node: 33.54101964249684
Node 49 (headwater): (325.0, 500.0), Distance to next node: 41.2310525617661
Node 50 (headwater): (215.0, 400.0), Distance to next node: 55.80109943740474
Node 51 (headwater): (260.0, 470.0), Distance to next node: 50.399510940453804
Node 52 (headwater): (230.0, 420.0), Distance to next node: 50.24937810540445
Node 53 (headwater): (225.0, 370.0), Distance to next node: 33.54101964249684
Node 54 (headwater): (120.0, 285.0), Distance to next node: 20.0
Node 55 (headwater): (120.0, 265.0), Distance to next node: 40.4038731001311
Node 56 (junction): (450.0, 380.0), Distance to next node: 202.48456731310588
Node 57 (headwater): (380.0, 570.0), Distance to next node: 503.0275304183699
Total Distance: 3076.4684035504017
```

Question 3:

Water from each headwaters has different chemical composition. We will feed you the sequence of junctions a chemical mix passes and its concentration in a tuple, maybe not on the same creek. Provide the likely headwater source of that chemical. Note water might be seeping from other headwaters into a creek.

Group 51

Marks for simplified coding

For instance [(58,3),(55,10),(52,5)] gives 25

[(57,10),(56,5),(55,2)] gives 22,21

Provide *chemical_source([(junction1,conc1), (junction2,conc2), etc]) -> node number/s*

Answer

Assumptions

- Junctions cannot be a source for the chemicals

Description of how the problem solved

- From previous assignment, some codes have been kept as they were needed for this question.
- The sequence in which chemical passes is checked if it is flowing downstream via directly connected junctions.
 - If connected, then we find the junction with highest concentration.
 - Otherwise, we find possible sources, calculate distance from each junction and the distance between the edge connecting the possible headwater and junction.
 - All the distances to each source's edge is summed up. For example, distance between 58 and edge of 25, 55 and edge of 25, 52 and edge of 25.
 - Source with lowest sum is most likely the source.
- Most likely source printed.

Code

```
#===== 3.1 =====  
#Question 1: Store the Data and Calculate Traveling Distance (Muhammed Abed)  
import csv  
import math  
  
# Read the data from the CSV file and store it as a list of dictionaries  
with open('water_data_q3.csv', 'r') as water_data:  
    dict_reader = csv.DictReader(water_data)  
    list_of_nodes = list(dict_reader)  
  
#initiate flow rate and distance for each node  
for node in list_of_nodes:  
    node['flow rate'] = 0  
    node['distance'] = 0  
  
#create a dictionary with nodes and their linked nodes
```


Group 51

```
riverSystem = {}
riverSystem[0]=[]
riverSystem[0].append(1)
for path in list_of_nodes:
    if path['water'] == 'Yes':
        for i in list_of_nodes:
            if i['Node'] == path['linked'] and i['water'] == 'Yes':
                x = int(path["linked"])
                y = int(path["Node"])
                #checks if the start point of the current river segment is not already a key in the
riverSystem dictionary.
                # If it's not in the dictionary, it means that this is the first time this starting point is
encountered.
                if x not in riverSystem:
                    #If the start point is not in the dictionary, initialize an empty list as the value for that key.
                    # This list will be used to store the ending points of river segments that start at the same
point.

                    riverSystem[x] = []
                    #appends the end point of the current river segment to the list associated with the start
point in the riverSystem dictionary.
                    # This creates a mapping between the starting point and all the ending points of river
segments that originate from it.
                    if y not in riverSystem[x]:
                        riverSystem[x].append(y)

#merge sort used to create a list in ascending order of node number
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    middle = len(arr) // 2
    left_half = arr[:middle]
    right_half = arr[middle:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)

def merge(left, right):
    result = []
    left_index, right_index = 0, 0

    while left_index < len(left) and right_index < len(right):
        left_node = int(left[left_index]['Node'])
```

```

    right_node = int(right[right_index]['Node'])

    if left_node < right_node:
        result.append(left[left_index])
        left_index += 1
    else:
        result.append(right[right_index])
        right_index += 1

    result.extend(left[left_index:])
    result.extend(right[right_index:])
    return result

# Sorting the list using merge sort
sorted_data = merge_sort(list_of_nodes)

#Create list containing only waterways.
river_nodes = []
for entry in sorted_data:
    if entry['water']=='Yes':
        river_nodes.append(entry)

#===== 3.3 =====
#Question 3

#Function to calculate distance between edge and a node
def distance_point_to_line(x, y, x1, y1, x2, y2):
    #calculate the differences between the coordinates of the point (x, y)
    #and the coordinates of one endpoint (x1, y1).

    A = x - x1
    B = y - y1
    #calculate the differences between the coordinates of the
    # two endpoints (x2 - x1 and y2 - y1).
    C = x2 - x1
    D = y2 - y1

    # calculate the dot product between vectors A and C and
    dot_product = A * C + B * D
    # find the squared length of vector CD
    length_squared = C * C + D * D
    # initialize a variable param to -1.
    param = -1

    #checks if the length_squared is not zero (to avoid division by zero)
    #and calculates the parameter param based on the dot product and the squared length.
    if length_squared != 0:

```

Group 51

```
param = dot_product / length_squared

# calculate the coordinates (closest_x and closest_y) of the closest point on the
# line segment to the given point (x, y) based on the parameter param.
if param < 0:
    closest_x, closest_y = x1, y1
elif param > 1:
    closest_x, closest_y = x2, y2
else:
    closest_x = x1 + param * C
    closest_y = y1 + param * D

#calculate distance using Pythagoras' Theorem
dx = x - closest_x
dy = y - closest_y
distance = math.sqrt(dx**2 + dy**2)
return distance

#function to check if the junctions are connected directly to each other
def downstream_check(tuple_list, entries, start):
    child = tuple_list[start][0]
    if start == entries-1:
        return True
    #check if next junction in order is linked to current junction
    elif tuple_list[start+1][0] == int(river_nodes[child-1]['linked']):
        valid = downstream_check(tuple_list, entries, start+1)
        if valid:
            return True
        else:
            return False
    else:
        return False

#List all possible sources by seeing thhe headwaters connected to nodes.
def possible_sources(tuple_list):
    sources=[]
    for i in tuple_list:
        for j in riverSystem[i[0]]:
            #since junctions cannot be a source, not added to list.
            if river_nodes[j-1]['type'] == 'headwater':
                sources.append([i[0],j])
    return sources

def chemical_source(tuple_list):
    entries = len(tuple_list)
    #returns True if junctions directly connected
    valid = downstream_check(tuple_list, entries, 0)
```

```

max_conc = 0
if valid:
    #If directly connected, junction with highest concentration is closest to source
    for i in tuple_list:
        if int(i[1])>max_conc:
            max_conc=int(i[1])
            closest_node=i[0]
    n1, n2 = riverSystem[closest_node][0], riverSystem[closest_node][1]
    if river_nodes[n1-1]['type'] == 'headwater' and river_nodes[n2-1]['type']=='headwater':
        return n1, n2
    elif river_nodes[n1-1]['Type'] == 'headwater':
        return n1
    else:
        return n2

#if not connected, checks distance between closest edge and junction
#This is because chemical may have seeped
else:
    #call function to list all possible sources and
    #the junctions they are connect to
    river_sources = possible_sources(tuple_list)

    min_dist = float('inf')
    closest_node = 0
    #Calculates distance between each node and possible edges
    for j in river_sources:

        dist=0 #reset distance
        hwater = j[1]
        junction = j[0]
        x1 = int(river_nodes[hwater-1]['x'])
        y1 = int(river_nodes[hwater-1]['y'])
        x2 = int(river_nodes[junction-1]['x'])
        y2 = int(river_nodes[junction-1]['y'])
        for i in tuple_list:
            node = int(i[0])
            x = int(river_nodes[node-1]['x'])
            y = int(river_nodes[node-1]['y'])
            dist += distance_point_to_line(x, y, x1, y1, x2, y2)
        if dist<min_dist:
            #update minimum distance and closest node
            min_dist = dist
            closest_node = int(j[1])

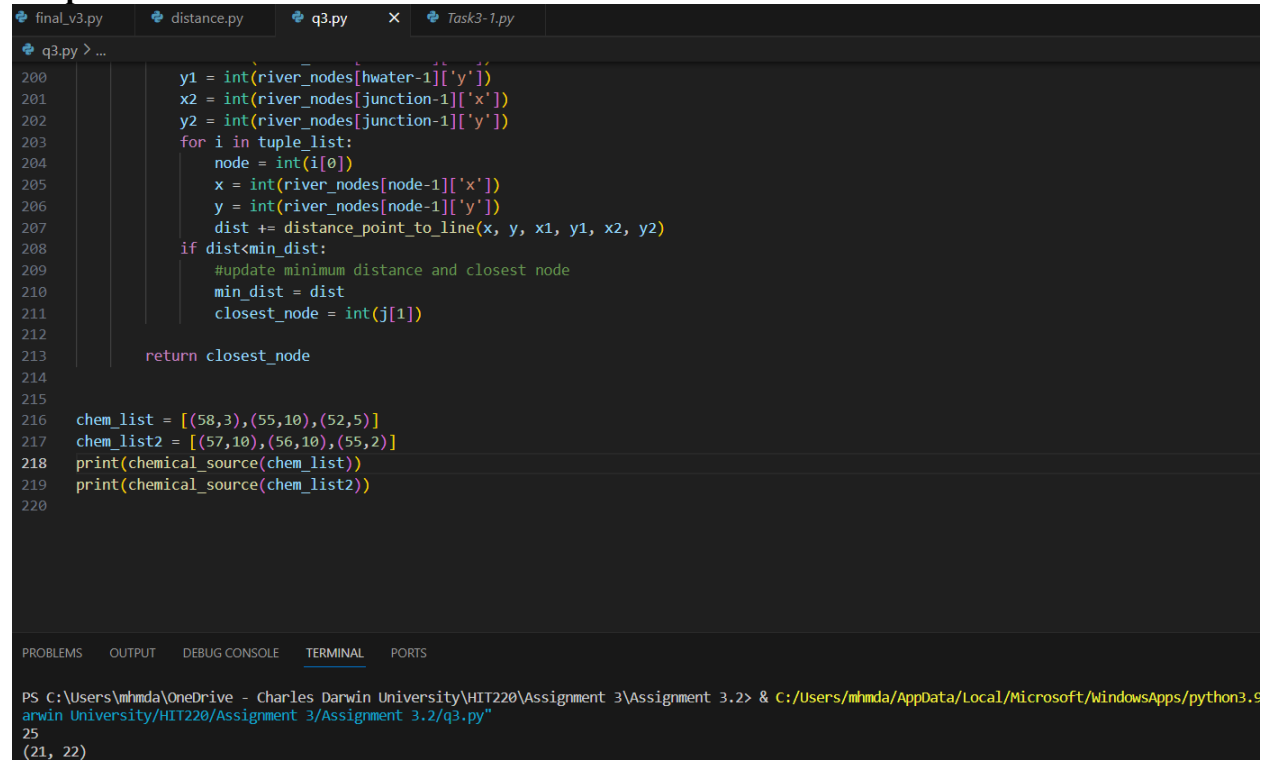
    return closest_node

```

Group 51

```
chem_list = [(58,3),(55,10),(52,5)]
chem_list2 = [(57,10),(56,10),(55,2)]
print(chemical_source(chem_list))
print(chemical_source(chem_list2))
```

Output



The screenshot shows a code editor with several tabs: 'final_v3.py', 'distance.py', 'q3.py', and 'Task3-1.py'. The 'q3.py' tab is active, displaying Python code. The code defines two lists, 'chem_list' and 'chem_list2', and calls a function 'chemical_source' on each. The code is as follows:

```
200 y1 = int(river_nodes[hwater-1]['y'])
201 x2 = int(river_nodes[junction-1]['x'])
202 y2 = int(river_nodes[junction-1]['y'])
203 for i in tuple_list:
204     node = int(i[0])
205     x = int(river_nodes[node-1]['x'])
206     y = int(river_nodes[node-1]['y'])
207     dist += distance_point_to_line(x, y, x1, y1, x2, y2)
208 if dist < min_dist:
209     #update minimum distance and closest node
210     min_dist = dist
211     closest_node = int(j[1])
212
213 return closest_node
214
215
216 chem_list = [(58,3),(55,10),(52,5)]
217 chem_list2 = [(57,10),(56,10),(55,2)]
218 print(chemical_source(chem_list))
219 print(chemical_source(chem_list2))
220
```

Below the code editor is a terminal window showing the output of the program. The output is:

```
PS C:\Users\mhmda\OneDrive - Charles Darwin University\HIT220\Assignment 3\Assignment 3.2> & C:/Users/mhmda/AppData/Local/Microsoft/WindowsApps/python3.9
arwin University/HIT220/Assignment 3/Assignment 3.2/q3.py"
25
(21, 22)
```

Question 4:

Create a Trie of the names of the rivers and creeks on the map. Encode each of the node characters with a binary code-word as shown in class, to form the smallest encoding Provide code_string(s) -> encoded string - a function that returns the code for a word entered from the alphabet collected from the river names

Answer

Assumption

The names of the rivers and creeks are stored in a CSV file called 'List of River and Creek.csv'.
(attached in the submission)

Description of how the problem solved

A Trie, also known as a prefix tree, is a tree-like data structure used for storing a dynamic set of strings. To encode each of the node characters with a binary codeword, we can use Huffman coding which is an optimal prefix coding method used for lossless data compression.

The steps to achieve this are:

1. Construct a frequency table for each character appearing in the river names.
2. Use the frequency table to create a Huffman tree, which provides us the binary code for each character.
3. Create a function `code_string` which returns the Huffman encoded string for a given word.

Code

```
import heapq
import csv

# Step 1: Constructing the frequency table
# The function takes a list of names as an input.
# It returns a dictionary where the keys are characters
# and the values are the frequency counts of those characters in the provided names.
def calc_freq(names):
    freq = {}
    for name in names:
        for char in name:
            if char not in freq:
                freq[char] = 0
            freq[char] += 1
    return freq

# Step 2: Create Huffman tree and codes
# The function takes the frequency table (created from Step 1) as input.
# It constructs the Huffman tree using a priority queue (heap).
def build_huffman_tree(freq):
    # Nodes in the heap consist of a weight (frequency of the character)
    # and a pair with the character itself and its Huffman code.
    heap = [[weight, [char, ""]] for char, weight in freq.items()]
    heapq.heapify(heap)
    # It follows the basic steps of Huffman coding algorithm:
    # selecting the two nodes with the lowest frequency, combining them into a single node,
```

Group 51

and repeating the process until there's only one node left which represents the root of the Huffman tree.

```
while len(heap) > 1:
    lo = heapq.heappop(heap)
    hi = heapq.heappop(heap)
    for pair in lo[1:]:
        pair[1] = '0' + pair[1]
    for pair in hi[1:]:
        pair[1] = '1' + pair[1]
    heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

return sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]), p))
```

Step 3 : Code_string function

The function takes a word and the Huffman codes as inputs.

It returns the Huffman encoded string for the word.

```
def code_string(word, huffman_codes):
```

```
    code = ""
    for char in word:
        for hc in huffman_codes:
            if hc[0] == char:
                code += hc[1]
                break
    return code
```

Open the CSV file and store the names of rivers and creeks in a list named river_names

```
river_names = []
```

```
with open('List of River and Creek.csv', 'r') as csvfile:
```

```
    csvreader = csv.reader(csvfile)
```

```
    # Assuming the names of rivers and creeks are in the first column
```

```
    # Skip the header if there's one
```

```
    next(csvreader)
```

```
    for row in csvreader:
```

```
        river_names.append(row[0])
```

Example Usage:

```
freq_table = calc_freq(river_names)
```

```
huffman_codes = build_huffman_tree(freq_table)
```

```
for word in river_names:
```

Group 51

```
encoded_word = code_string(word, huffman_codes)
print(f"Encoded Word for {word}: {encoded_word}")
```

Output

```
[Running] python -u "/Users/phamthiminhchau/Documents/Algorithms and Complexity/Group assignment 3.3/v2 Question 4.py"
Encoded Word for DALY RIVER: 1100000101001100000101011100010010111011
Encoded Word for CHILLING & MULDIRA CREEKS: 0011010011000100111001110001101001010110100001010111001001001001111000100010001010011011111111000111011
Encoded Word for HAYWARD CREEK: 010010010000001100110001001111000101001101111111110001
Encoded Word for FISH RIVER: 1100111100011011101001101011100010010111011
Encoded Word for BAMBOO CREEK: 000011100101100100000111010110101110100110111111110001
Encoded Word for GREEN ANT CREEK: 010101011111111110101010010110100100010100110111111110001
Encoded Word for DOUGLAS RIVER: 110000101101010001010110011001011011101011100010010111011
Encoded Word for HAYES CREEK : 010010010000001111101110100110111111110001101
Encoded Word for MIDDLE CREEK : 110010100011000110001001111110100110111111110001101
Encoded Word for STRAY CREEK : 110110100001100100000010100110111111110001101
Encoded Word for BRADSHAW CREEK : 00001110110010110001101101001001011001101010011011111110001101
Encoded Word for DEAD HORSE CREEK : 11000111001011000101010010101111101111111010011011111110001101
```