CSC263 – Problem Set 4

Remember to write your **full name** and **student number** prominently on your submission. To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Remember that you are required to submit your problem sets as both LaTeX `.tex` source files and `.pdf` files. There is a 10% penalty on the assignment for failing to submit both the `.tex` and `.pdf`.

Due April 1, 2019, 22:00; required files: ps4.pdf, ps4.tex, party.py, game.py

Answer each question completely, always justifying your claims and reasoning. Your solution will be graded not only on correctness, but also on clarity. Answers that are technically correct that are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

You may work in groups of up to THREE to complete these questions.

Authors: Junwen Shen(1004299190), Mengning Yang(1002437552), Jianhao Tian(1001354465)

1. **[12]** You are given a weighted, connected, undirected graph $G = (V, E)$ and one of its minimum spanning trees $T \subseteq E$. Now consider the following two scenarios where we modify the graph and want to get an updated MST efficiently.

   (a) **[6]** A new edge (u, v) with weight $w_{u,v}$ $(u, v \in V)$ is added to G, resulting in a new graph $G' = (V, E \cup \{(u, v)\})$. How do you efficiently find a minimum spanning tree T' of G'? Describe and justify your algorithm in concise and precise English, and analyse its runtime. To get full marks, your algorithm's worst-case runtime must be in $\mathcal{O}(|V|)$.

   (b) **[6]** An edge $(u, v) \in E$ is removed from G, resulting in a new graph $G' = (V, E - (u, v))$. Assume that G' is still connected. How do you efficiently find a minimum spanning tree T' of G'? Describe and justify your algorithm in concise and precise English, and analyse its runtime. To get full marks, you must make your algorithm as fast as possible.

   Answer
   (a) Case 1: edge (u, v) is already in the MST T'
   We will loop through the MST, if edge (u, v) is already in the tree, then we simply compare the weight of the new edge with the existing one, and replace the one with heavier weight with the one with lighter weight.
   case 2: edge (u, v) is not in the MST T'
   case 2.1: u, v are on the same path
   We start by looking for v from u and u's ancestors (u path), and keep looking for u from v and v's ancestors (v path), if we can find v on u path or find u on v path, then we know that u, v are on the same path. Add the new edge to the tree, $T' = (V, E \cup \{(u, v)\})$, then a cycle must be formed. Then we will search through the cycle and delete the edge with largest weight.
   case 2.2: u, v are on different paths
   We start by looking for v on u path, and keep looking for u on v path, if we can't find v on u path nor we can't find u on v path, then we know that u, v are on different paths. Then we will put all the edges on u path and v path into a set, and delete all the edges that are same from u path and v path, then add the new edge into the set, now all the edges left in this set must form a cycle. Then we will search through the cycle and delete the edge with largest weight.
   Run time Analysis: In the worst case, we will traverse through the whole tree comparing tree nodes, and do some constant operations like delete an edge. If we need to search through a cycle or add all the nodes into a set, which also takes $\mathcal{O}(|V|)$. So the worst case run time is some constants times $\mathcal{O}(|V|)$.

   (b) case 1: edge (u, v) is not in the MST T'
   We will loop through the MST, if edge (u, v) is not in the tree, then we do nothing. so the worst case run time is $\mathcal{O}(|V|)$.
   case 2: edge (u, v) is in the MST T'
   If edge (u, v) is in the MST T', then u and v must be on the same path, because if they are on different paths then there must be a cycle, then T' wouldn't be a tree anymore. We will make two empty sets, set u and set v. These two sets are used to store the nodes that are on u's subtree and v's subtree (two separate trees after deleting edge (u, v)). And we will make a copy of MST T' to allow us to modify the tree. We will put node u in set u and node v in set v.

If u is v's parent, after deleting the edge (u, v), all the ancestors of u must be in the set u (the upper sub-tree), so we keep searching for u's ancestors and put them in the set u, and delete all these nodes from the copy of the MST T'. Then, search through the nodes left in the copy of the MST T' to see if they are in either set u or set v. If the node is in neither set u nor set v, then keep searching for this node's parent, until you find it in set u or set v. If you find it in set v, then put all the nodes on this path into the set v and delete these nodes in the copy MST T', and if you find it in set u, then put all the nodes on this path into the set u and delete these nodes in the copy MST T'. After the copy MST T' become empty, we have separated the MST T' into two sets indicating the two sub-trees. Then for each edge in the set E of the graph, check if one of the vertex is in set u and the other vertex is in set v, if not, then keep searching, if it is, meaning the edge can connect these two sub-trees. Keep searching and checking until we find the one edge that can connect two sub-trees and with the minimum weight, and make the two sub-tree a new MST again. Vise versa if v is u's parent.
Run time analysis: We first traversed through the tree splitting the tree into two sets and delete the node in the MST copy T', this takes $\mathcal{O}(|V|)$. Then we loop through the set E to check if the vertex of the edge is either set u or set v and pick the edge with minimal weight, which takes $\mathcal{O}(|E|)$. So the overall runtime is $\mathcal{O}(|V + E|)$.

2. **[12]** In this question, you will solve the **Two Party Problem**.

There are two parties occurring in the same evening: a math party and a CS party. Each student chooses which party to attend. We are interested in answering questions of the form "is student x attending the same party as student y?". Unfortunately, students are embarrassed about attending these kinds of parties, so we are never explicitly told that two students are attending the same party. Instead, we are forced to learn only from statements of the form "student x is **NOT** attending the same party as student y".

The function `solve_party` takes a list of commands, where each command is an `add` command or a `tell` command. Your task is to process the commands in order and return the list of results from the `tell` commands.

The `add` commands give us the "student x is **NOT** attending the same party as student y" information. Specifically, an `add` command is a string of the form `add x y`, and tells us that student `x` and student `y` are **NOT** attending the same party.

The `tell` commands ask us for information. A `tell` command is a string of the form `tell x y`, and asks us to say what we know about the parties that student `x` and student `y` are attending. Each `tell` command results in one of three strings being appended to the list that is returned:

- `same`, if `x` and `y` are attending the same party
- `different`, if `x` and `y` are attending different parties
- `unknown`, if we don't have enough information to determine whether they are attending the same or different parties

Assume that `x` and `y` in the above commands are between `1` and `n`, where `n` is the number of students.

Let's go through an example. Here is a sample call of `solve_party`:

```
solve_party(
    ['tell 1 3',
     'add 1 3',
     'tell 1 3',
     'add 3 4',
     'tell 1 4'
    ])
```

This is what happens on each step:

- `tell 1 3`: what do we know about the parties that students 1 and 3 are attending? Nothing! So we append `unknown` to our list.
- `add 1 3`: we learn that students 1 and 3 are attending different parties.
- `tell 1 3`: this one again, except that we know now that students 1 and 3 are attending different parties, so we append `different` to our list.
- `add 3 4`: we learn that students 3 and 4 are attending different parties.

- `tell 1 4`: what do we know here? Well, we know from earlier `add` commands that students 1 and 3 are attending different parties, and that students 3 and 4 are attending different parties. This lets us conclude that students 1 and 4 are attending the same party — so we append `same` to our list. (If you're not convinced: because students 1 and 3 are attending different parties, let's say that student 1 is attending the CS party and student 3 is attending the math party. We also know that student 3 and 4 are attending different parties, so student 4 must be attending the CS party. Now we see that students 1 and 4 are attending the same party.)

For this example, `solve_party` returns `['unknown', 'different', 'same']`.

Your goal is to design `add` and `tell` to run as quickly as possible. Include in your `ps4.pdf/ps4.tex` a clear description of your algorithm, justification that your algorithm is correct, and the running time of your algorithm. In question 4, you'll be asked to implement your algorithm.

Solution:

```
class NodeQueue():

    def __init__(self):
        self.head = None
        self.end = None
        self.length = 0



    def enqueue(self, node):
        if self.head == None:
            self.head = node
            self.end = node
            self.length = 1
        else:
            self.end.next = node
            self.end = node
            self.length += 1


    def dequeue(self):
        if self.length == 1:
            result = self.head
            self.head = None
            self.end = None
            self.length = 0
            return result
        else:
            result = self.head
            self.head = self.head.next
            self.length -= 1
            return result



class Node():

    def __init__(self, data):
        self.data =  data
        self.next = None
        self.distance = 0
        self.checked = False
        self.connected = []



def generate_graph(node_num1, node_num2, diction):
    if node_num1 in diction.keys():
        if node_num2 in diction.keys():
            diction[node_num1].connected.append(diction[node_num2])
            diction[node_num2].connected.append(diction[node_num1])
        else:
            new_node = Node(node_num2)
            new_node.connected.append(diction[node_num1])
            diction[node_num2] = new_node
            diction[node_num1].connected.append(new_node)
    else:
        if node_num2 in diction.keys():
            new_node = Node(node_num1)
            new_node.connected.append(diction[node_num2])
            diction[node_num1] = new_node
            diction[node_num2].connected.append(new_node)
        else:
            new_node1 = Node(node_num1)
            new_node2 = Node(node_num2)
            new_node1.connected.append(new_node2)
            new_node2.connected.append(new_node1)
            diction[node_num1] = new_node1
            diction[node_num2] = new_node2



def get_path(node1, node2):
    if node1 == node2:
        return -1
    Node_queue = NodeQueue()
    Node_queue.enqueue(node1)
    node1.checked = True
    while Node_queue.length != 0:
        node = Node_queue.dequeue()
        for nodes in node.connected:
            if nodes.checked == False:
                nodes.checked = True
                nodes.distance = node.distance + 1
                if nodes.data == node2.data:
                    return nodes.distance
                Node_queue.enqueue(nodes)
    return 0



def solve_party(commands):
    ’’’
    Pre: commands is a list of commands
    Post: return list of ’tell’ results
    ’’’
    graph = {}
    result = []

    for command in commands:
        if command[0] == 't':
            node_num1 = command.split()[-2]
            node_num2 = command.split()[-1]
            if not (node_num1 in graph.keys() and node_num2 in graph.keys()):
                result.append('unknown')

            else:
                num = get_path(graph[node_num1], graph[node_num2])
                if num == 0:
                    result.append('unknown')
                elif num == -1 or num%2 == 0:
                    result.append('same')
                elif num%2 == 1:
                    result.append('different')
                for key in graph:
                    graph[key].checked = False
                    graph[key].next = None
                    graph[key].distance = 0
        if command[0] == 'a':
            node_num1 = command.split()[-2]
            node_num2 = command.split()[-1]
            generate_graph(node_num1, node_num2, graph)
    return result
```

- `Algorithm`:

  First, we build a graph according to the given information, where each vertex represents a student, and the connection between two vertexes represents that they are not in the same party. Secondly, using BFS algorithm to find the distance value between any two given vertexes. Finally, determine whether the distance is odd or even. If the value is even, then these two students are in the same party, if the value is odd, then these two students are not in the same party, else if the distance path does not exist, then their relationship is unknown.

  We use adjacency list to represent the graph. Each vertex(student) is a node. For efficiency, we use dictionary instead of list, where every nodes is the value of a key, the key has a same name with the node's data(self.data).

  When the command is asking to `add X Y`, the function `generate_graph()` will modify the dictionary. According to the given relationship between X and Y, `generate_graph()` will add node Y to X.connected, and add X to Y.connected. If X or Y is not a key in the dictionary, the function will create new key first, and then add them respectively.

  When the command is asking to `tell X Y`, the function `get_path()` will return the shortest distance value from X to Y. `get_path()` uses BFS algorithm to traverse the dictionary, until it finds the node Y or all the nodes is being checked. The function will check the node X first, after checking node X(X.checked = True), the function `enqueue()` will enqueue all the nodes in X.connected into `NodeQueue`(if X.connected is not empty). The next checking will start from the first element in `NodeQueue`, after this checking, the self.distance of this node will be the distance of the node who finds the current node + 1. Then this element will be dequeue from `NodeQueue`, and its child nodes(nodes in self.connected) will be `enqueue()` into the `NodeQueue`(if self.connected is not empty). We will follow this searching pattern until the `NodeQueue` is empty or the node Y is being found. If the function `get_path()` does not find node Y, it will return 0, otherwise, it will return Y.distance. If X Y are the same node, it will return -1.

After the function `get_path()` returns, the function `solve_party()`: will check the return value of `get_path()`, and then append information into the list `result` accordingly.Then, `solve_party()` will reset the information(self.checked, self.next and self.distance) for the next `tell` command. After all the commands have been executed, it will return `result`.

- `Algorithm correctness`:

  By the given information, there are only two parties, so we can consider this question as a bipartite graph question. The main idea of this algorithm is to find the distance value between the given nodes to determine their relationship, that is, in bipartite graph, to find how many links are there between two given nodes. In a bipartite graph, a single link from one group set must point to the other group set. According to this logic, we can conclude that the even number of links will finally point to the group set itself, and odd number of links will finally point to the other group set. Similarly, if the distance value between two nodes is even, then they must be in the same party, if the distance value is odd, then they must be in different parties.

- `Runtime analysis`:
  1.`generate_graph(node_num1 , node_num2 , diction)`:
  The worst case for this function is when both `node_num1` and `node_num2` are not a key in diction. Where to determine whether `node_num1` and `node_num2` are keys takes run time O(1). And in this case, we will create two new nodes, which takes run time O(1), and then append each node into their self.connected list respectively, this action also has run time O(1). So the total run time in worst case for this function is O(1).

  2.`get_path(node1 , node2)`:
  The worst case for this function is when `node2` is not in the dictionary, so the function has to traverse the whole dictionary. In this case, assume we have V nodes, and each nodes has a self.connected list with length E. We will visit each node once, and each visit takes constant work, so the total runtime for visit each node is O(V). For each node, we need to loop through to check all the nodes in self.connected. this will take runtime O(E). So the total run time for this function in worst case is O(V+E).

  3.`solve_party()`:
  Assume that the length of commands is n,the worst case for this function will be use n/2 commands to build a dictionary, and use the other n/2 commands to call function to ask for an unknown relationship between two nodes. When we use n/2 commands to build the dictionary, because the worst case run time for the function `generate_graph()` is O(1), and we will call the function n/2 times, so the total run time to build the dictionary is O(n/2). When we use n/2 commands to call function `get_path()`, because the worst case run time for this function is O(V+E), and the maximum value of V+E after n/2 times of building is (n/2)*2*2 = 2n.(assume each `add` commands provides two new nodes and each nodes has 1 connected node) Therefore, the total run time to execute `tell` command will be (n/2) * 2n = n^2. So the total worst case run time for the whole function is $n/2 + n^2 = O(n^2)$.

3. **[12]** Dan and Sushant are playing a board game on an m by n board. The board has at least 4 rows and 4 columns. The bottom row is row 0 and the top row is row $m-1$; the left-most column is column 0 and the right-most column is column $n-1$.

   Dan moves first, then Sushant, then Dan, then Sushant, etc. until the game is over. The game is over when one of two things happens: Sushant wins or Dan wins.

   - Sushant wins if he lands on the same square as Dan before Dan reaches the top row. Note that this winning condition is checked **only after Sushant moves**; Sushant can never win right after Dan moves, even if Dan lands on the same square as Sushant.
   - Dan wins (go Dan go!) if Dan reaches the top row before Sushant wins, i.e., Dan reaches the top row without Sushant ever landing on the same square as Dan. As soon as Dan reaches the top row, Dan wins (Sushant cannot move anymore).

   Dan has no choice on his move: he always moves up one square. Sushant, by contrast, has eight choices of move to make on his turn:

- 1 up, 2 right
- 1 up, 2 left
- 1 down, 2 right
- 1 down, 2 left
- 2 up, 1 right
- 2 up, 1 left
- 2 down, 1 right
- 2 down, 1 left

That is, if `S` is the location of Sushant, then his valid moves are a-h in the following table:

		f		e		
	b				a	
			S			
	d				c	
		h		g		

Note that some moves may be unavailable depending on Sushant's location; for example, if Sushant is already in column $n - 1$, then any move that tries to go to the right is not allowed.

Given the starting locations of Dan and Sushant, design an algorithm that determines the result of the game, as follows:

- If it is possible for Sushant to win, then report that Sushant can win and give the minimum number of Sushant moves required for him to win.
- Otherwise, Dan wins; report the number of Sushant moves that occur before Dan wins.

You may assume the following regarding Dan and Sushant's starting locations:

- Dan's starting location is never in the top row.
- Sushant's starting location is never the same as Dan's.

Include in your `ps4.pdf/ps4.tex` a clear description of your algorithm, justification that your algorithm is correct, and the running time of your algorithm. In question 5, you'll be asked to implement your algorithm.

Solution:

```python
moves = [(-1, 2), (-1, -2), (1, 2), (1, -2), (-2, 1), (-2, -1), (2, 1), (2, -1)]
def get_path_len(rows, cols, start, end):
  board = [[0] * cols for i in range(rows)]
  paths = {}
  tries = []
  p1 = end
  while p1 != start:
    for d in moves:
      p2 = (p1[0]+d[0], p1[1]+d[1])
      if p2[0] in range(rows) and p2[1] in range(cols) and
      board[p2[0]][p2[1]] == 0 and p2 not in paths:
        board[p2[0]][p2[1]] = 1
        tries.append(p2)
        paths[p2] = p1
    try:
      p1 = tries.pop(0)
    except IndexError:
      return -1
  count = 0
  p = start
  while p != end:
    p = paths[p]
    count += 1
  return count
def game_outcome(rows, cols, dan_row, dan_col, sushant_row, sushant_col):
  target_positions = [(i, dan_col) for i in range(dan_row+1, rows)]
  start = (sushant_row, sushant_col)
  x = 1
  while x < len(target_positions):
    steps = get_path_len(rows, cols, start, target_positions[x-1])
    if steps < 0:
      continue
    elif ((steps + x) ^ 2) & 1:
      break
    elif steps - x > 2:
      x += (steps - x) // 2
      continue
    elif steps <= x:
      return "Sushant wins in {0} moves".format(x)
    x += 1
  return "Dan wins in {0} moves".format(rows - dan_row - 2)
```

- Algorithm:
  For every position of Dan's path before reaching the top row, calculate the minimum steps that Sushant need to take to reach that position.
  - If Sushant is not able to reach that position, continue the loop;
  - If the steps that Dan need and the steps that Sushant need are in different parity, break the loop, Dan definitely wins the game, return the fixed steps;
  - Otherwise (At this time, two kinds of steps always are in same parity), when Dan needs more steps to reach that position (before reaching the top row), then Sushant will win at that steps.
- Correctness:
  Suppose the start positions of Dan and Sushant are $(a_0,\ b_0)$, $(p_0,\ q_0)$ respectively. Dan will reach the position $(a,\ b_0)$ after s_1 steps and the minimum steps that Sushant reaches $(a,\ b_0)$ is s_2.
  - If s_1 and s_2 are in the same parity, then Sushant is able to catch Dan after a finite even number of steps since he moves faster than Dan and lands on the same column, i.e. by taking movements a then b or e then f (shows in question descriptions) repeatedly, Sushant can catch up Dan.
  - Otherwise, Sushant can never catch up Dan, since two numbers with different parities adding a same even number are still in different parities.
- Runtime:
  In worst case, the function will go through all points on Dan's path, and for every point, we use BFS to find the shortest path of Sushant. Therefore, the runtime is $nO(BFS)$, which is $O(mn^2)$, m, n are the number of colums and rows of the board respectively.

Programming Question

The best way to learn a data structure or an algorithm is to code it up. In each problem set, we will have a programming exercise for which you will be asked to write some code and submit it. You may also be asked to include a write-up about your code in the PDF/TEXfile that you submit. Make sure to **maintain your academic integrity** carefully, and protect your own work. The code you submit will be checked for plagiarism. It is much better to take the hit on a lower mark than risking much worse consequences by committing an academic offence.

4. [**4**] Your first programming exercise is to write function `solve_party` that implements your algorithm for the Two Party Problem. Remember that you are to return a list giving the result, in order, of each `tell` command.
   - Your code must be written in Python 3, and the filename must be `party.py`.
   - We will grade only the `solve_party` function; please do not change its signature in the starter code. include as many helper functions as you wish.
   - `solve_party` should **not** have any `print` calls. Instead, please `return` the correct list of strings.
5. [**4**] Your second programming exercise is to write function `game_outcome` that implements your algorithm for the board game.

   There are two valid types of strings to return from this function, as follows (`xxx` is an integer):

   ```
   Sushant wins in xxx moves
   Dan wins in xxx moves
   ```

   - Your code must be written in Python 3, and the filename must be `game.py`.
   - We will grade only the `game_outcome` function; please do not change its signature in the starter code. include as many helper functions as you wish.
   - `game_outcome` should **not** have any `print` calls. Instead, please `return` the correct string.