CSC263 – Problem Set 4

Si Tong Liu(1004339628), Jing Huang(1003490705), Yifei Gao(1004152640)

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.

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.

   Solution:

   [1] Since the T given in the question is an MST, it is a connected and acyclic subgraph of G which covers all vertices in G. Now if we add a new edge (u, v) to G, this edge will also in T and it will create a cycle in T (by theorem). Then we start with an empty tree T', and keeps adding edges in T that keep the hope of T' growing into an MST. For the runtime, before adding new edge, T has exactly $|V| - 1$ edges. After adding one new edge, total number of edges is $|V|$ now. Therefore, in worst-case, we will go through every edges in T to check, which makes the worst-case runtime $\mathcal{O}(|V|)$.

   [2] After deleting an edge (u, v) from G, there should be two cases:

   Case 1: The deleting edge is not in T. Since after deleting edge (u, v), the new graph G' is still connected. And if edge is not in T. The new MST T' will be T which does not change. In this case, we will just return T and the runtime will be $\mathcal{O}(1)$.

   Case 2: The deleting edge (u, v) was in T. After deleting edge, T will be separated into two disconnected trees. So to find the new MST T', we need to go through one of the disconnected trees, then in G' check every edge that contains vertices in this tree and vertices in the other tree. Finally, find the lightest edge and connect the two disconnected tree with the lightest edge. Since the number of edges will be at most E which is given in the question. In the worst case, we need to check every edge and the runtime will be $\mathcal{O}(|E|)$.

   In conclusion, the total runtime is $\mathcal{O}(|E|)$

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.

Design algorithm:

We are considering using `List`, `Dictionary`, and `Set` to design algorithm for this question.

Step [1]: We use `Set` to represent the relationship of two students in every `add` command. And every time we are given an `add` command, we record it by adding it to a list. For example $\{x, y\}$, where x and y are two students. Runtime for creating set takes $\mathcal{O}(1)$.

Step [2]: Whenever we find there are students in the same party for all given `add` command, we put them into the same `Set` by call `union` method. And we choose one element as the representative of the set. Then we make the representative as the key and the `Set` the value inside a dictionary. For example, $\{x : \{x, z\}\}$. In this example, x is the representative of the `Set` $\{x, z\}$. And the `Set` $\{x, z\}$ means student x and student z are in the same party. Lastly, add it into a dictionary to record it. Runtime for finding student in a dictionary takes $\mathcal{O}(n)$, if we assume the length of the dictionary is n. Runtime for assigning representative is $\mathcal{O}(1)$. Runtime for unioning students in the same party takes $\mathcal{O}(n^2)$. So the runtime for this step takes $\mathcal{O}(n^2)$

Step [3]: Every time there is a `tell` command, we firstly look into the `Dictionary` we created. If both students can be found inside the current `Dictionary`, we determine whether they are in same `set` by telling the key. If they are the same, add 'same' to result; if not, add 'different'. If one of the student can not be found in the `Dictionary`, we go through all the previous add commands to see if the two students were recorded as going to different parties already, if found, add 'different' to the result; if not, add 'unknown' to the result. Assume the length for dictionary and the list we created are n and m, then we need to go through either of them depending on the situation, so the runtime for this step takes $\mathcal{O}(n + m) \leq \mathcal{O}(n)$

Step [4]: In the `solve_party` function, every time we get a new `add` command, we do step [1] and step [2], ;if we get `tell` command, we do step [3]. Then repeat... Finally, we will get a list that contains the different results of all `tell` commands. Assume there are n `add` command and m `tell` command in the given commands. Runtime for the entire algorithm is: $\mathcal{O}(n^3) + \mathcal{O}(m^2) \leq \mathcal{O}(n^3)$

In conclusion, the runtime for the algorithm would be $\mathcal{O}(n^3)$

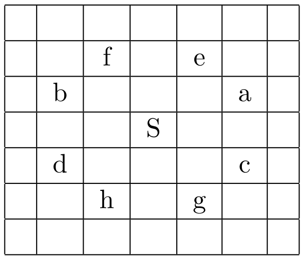
3. **[12]** Dan and Sushant are playing a board game on an m by n board. The board has at least 4 rows and 4 columns. The bottom row is row 0 and the top row is row $m - 1$; the left-most column is column 0 and the right-most column is column $n - 1$. q Dan moves first, then Sushant, then Dan, then Sushant,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



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.

Design algorithm:

Because the only way for Dan to move is one step up, we can split the algorithm into two parts: the first scenario is when Dan wins, the second is when Sushant wins.

[1] We use s to represent the row-difference between Dans original location and the top. We then calculate the row-distance between Dan and Sushant (similar as the difference in Y coordinate in algebra). Since Sushant moves up/down 1 or 2 rows a time while moving left/right 2 or 1 columns a time, the distance sushant moves is a distance of 2 and 1. Then we set up two equations to calculate the exact 2-1 steps and 1-2 steps Sushant needs to reach Dan. We assume Sushant takes x amount of 2-1 steps and y amount of 1-2 steps to reach Dan. The row-difference between Dan and Sushant takes 2x and y steps to be reduced and the column-difference between Dan and Sushant takes x and 2y steps to be reduced. If the sum of these steps is a number that is greater than or equal to s (which is the steps Dan needs to win the game), then we conclude that Sushant will lose the game. Since Dan is the first to move, when Dan wins at step s, the game is over and Sushant had moved s-1 steps. And because we only used linear equation group of math to solve the problem, the runtime for this algorithm will be $\mathcal{O}(1)$.

[2] When Dan cannot win, we should find how many steps it takes for Sushant to win. Our main idea is to reduce the size of the board when dynamically. Firstly, we check the direction of Dan according to Sushant. We assume the board is a square with Dan and Sushant on the end of the diagonal of the square. We cut the left, rightdown area of the board according to this square, leaving only this square and the upper area of it. We repeat this every time Dan moves up, until the board has only 4 columns. Runtime for cutting the board will take O(1), after cutting the board, we find all possible path from Sushant to Dans location, which takes around $\mathcal{O}(4^n)$

So the total runtime for this algorithm will take $\mathcal{O}(1) + \mathcal{O}(4^n) \leq \mathcal{O}(4^n)$

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.

   Write up of `solve_party`:

   (1) for `find_student` function: it finds whether a specific student is in a given dictionary, and return the key(represent) of the set that contains the student in the given dictionary. If not found, return -100, since students are represented from 0 to n. If the return result is -100, the student can not be found in the dictionary.

   (2) `get_first_element` function returns the first element of a set and `get_second_element` function returns the second element of a set.

   (3) In `union_info` function: given a list of sets and a dictionary which contains sets as values and representative as keys. This function check every set and union all students who attend to the same party.

   (4) In `solve_party` function: go through every command in commands, if `add` command, record the two students as set do `union_info`. if `tell` command, firstly look into the dictionary, if found, compare the representative; if not, check all add command.

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.

Write up for `game_outcome`:

(1) Use class `Square` to represent every square in the board game

(2) Create a class `Game` which represents the board game.

(3) Inside class `Game`, `sushant_next_moves` is about the rule of how Sushant can move inside the board game. Function `check_if_dan_win_directly` checks whether Dan can win directly by doing math equation. Inside `play_Game`, we check every possible path from Sushant to Dan.

(4) In side `game_outcome` function, we just do step (3) and get the final result.