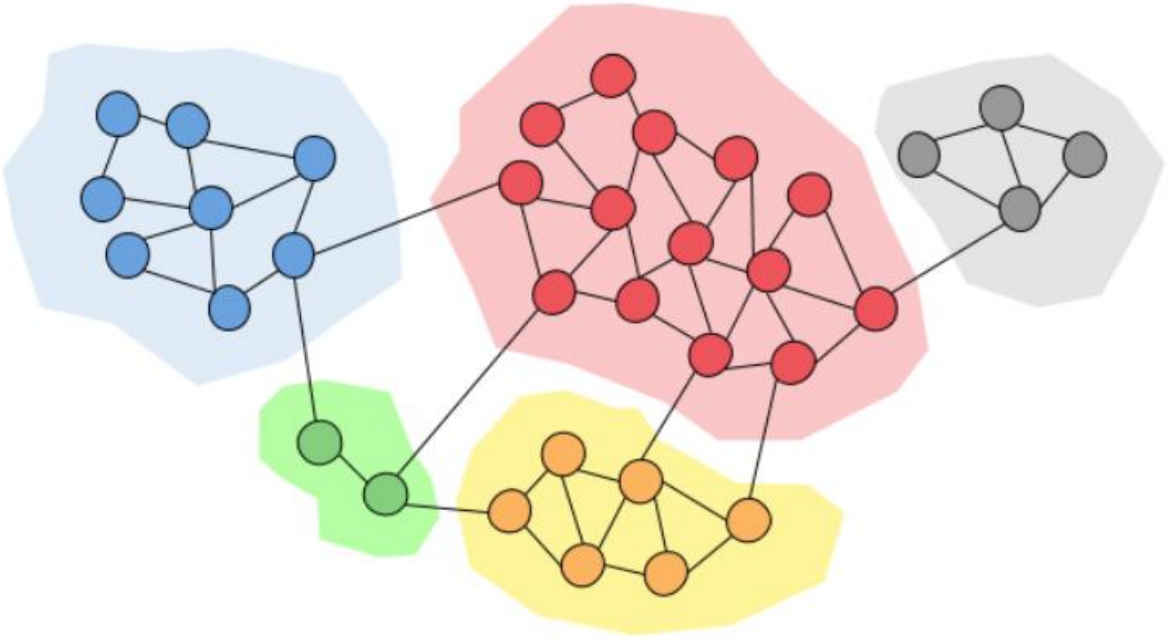
Data Structures and Algorithms Design

Assignment 1: PS06 – Community Detection



Data structure model

The data type employed is **set()** to store the participants in each group. The **set()** data type is a collection of unique elements, so it will automatically remove any duplicates. It also provides fast membership testing using the **in-operator**, which makes it efficient to check if a participant is already in a group.

Details of each operation

read_input():

```
def read_input():
    if os.path.exists("inputPS06.txt"):
        with open("inputPS06.txt", "r", encoding='utf-8-sig') as f:
            data = [line.strip() for line in f]

            if len(data) == 0:
                print("File is empty, please provide a valid input file")
            else:
                return data
    else:
        print("file doesn't exist, please provide the file")
```

This function reads the input data from the file **inputPS06.txt** and returns it as a list.

Time complexity: O(n), where n is the number of lines in the input file. This operation is efficient because it reads the input file line by line and processes each line in constant time.

process_data(data):

It has two parts

a) Split the entry into the two participants and check if the participants are already on the list of lists

```
def process_data(data):
    groups = []

    for entry in data:
        if '/' in entry:
            p1, p2 = entry.split("/")
        else:
            print("input file format is invalid")
            break

        found = False
        for group in groups:
            if p1 in group or p2 in group:
                group.add(p1)
                group.add(p2)
                found = True
                break
```

b) and Check if the participants are in another group

```
        if not found:
            for group in groups:
                if p1 in group or p2 in group:
                    if len(group) < len([p1, p2]):
                        group.update([p1, p2])
                    else:
                        [p1, p2].update(group)
                        groups.remove(group)
                        groups.append([p1, p2])
                    found = True
                    break

            if not found:
                groups.append(set([p1, p2]))

    return groups
```

This function takes the input data as a parameter and returns a list of sets, where each set represents a group and contains the participants.

Time complexity: $O(n^2)$, where n is the number of entries in the input data. This operation is efficient because it iterates through the input data and checks if the participants are already in the list of sets using the **in** operator, which has a time complexity of O(1).

create_output(groups):

```
def create_output(groups):
    output = ""
    for i, group in enumerate(groups):
        participants = ", ".join(group)
        output += f"Group {i + 1}: There are {len(group)} participants in the group and they are {participants}\n"
    return output
```

This function takes the list of sets as a parameter and creates the output string.

Time complexity: O(n), where n is the number of groups. This operation is efficient because it iterates through the list of sets and creates the output string for each group in constant time.

write_output(output):

```
def write_output(output):
    with open("outputPS06.txt", "w") as f:
        f.write(output)
```

This function takes the output string as a parameter and writes it to the output file **outputPS06.txt**.

Time complexity: O(n), where n is the length of the output string. This operation is efficient because it writes the output string to the file in a single step.

The overall time complexity of this program is **$O(n^2)$** as the core function of this program is process_data() which has two nested for loops running for the length of the data provided as the parameter.

An alternate way of modeling the problem

One alternate way of modeling the problem would be to use a dictionary to store the groups and their participants. The keys of the dictionary would be the group names, and the values would be lists of participants.

This approach would have a lower time complexity for the **process_data(data)** function because we could use the **setdefault()** method of the dictionary to add a participant to a group in constant time. However, it would have a higher time complexity for the **create_output(groups)** function, because we would have to iterate through the dictionary to create the output string.

The cost implications of this approach would be increased memory usage because dictionaries use more memory than lists. However, the difference in memory usage may not be significant for small input sizes.