Report on AuntNamesday using python

Overview

The `AuntNamesday` class is designed to determine if guests at a party can be seated at two separate tables based on their mutual dislikes, ensuring that no two guests who dislike each other are seated at the same table.

Class Components

1. \_\_init\_\_ Method

def \_\_init\_\_(self):

self.dislikes = {}

self.color = {}

- Purpose: Initializes the `dislikes` and `color` dictionaries.

- `self.dislikes`: Stores the dislike relationships between guests.

- `self.color`: Tracks the color (table assignment) of each guest.

2. `is\_bipartite` Method

def is\_bipartite(self, start):

stack = [start]

self.color[start] = 0

while stack:

guest = stack.pop()

for disliked\_guest in self.dislikes.get(guest, []):

if disliked\_guest not in self.color:

self.color[disliked\_guest] = 1 - self.color[guest]

stack.append(disliked\_guest)

elif self.color[disliked\_guest] == self.color[guest]:

return False

return True

- Purpose: Determines if the subgraph starting from `start` is bipartite.

- Parameters: `start` - the starting guest for the DFS.

- Algorithm:

- Uses a stack to perform a depth-first search (DFS).

- Colors the starting guest with `0`.

- Iteratively processes each guest, coloring disliked guests with the opposite color.

- If a conflict is found (a disliked guest with the same color), returns `False`.

- If no conflicts are found, returns `True`.

3. `add\_dislike` Method

def add\_dislike(self, guest1, guest2):

if guest1 not in self.dislikes:

self.dislikes[guest1] = []

if guest2 not in self.dislikes:

self.dislikes[guest2] = []

self.dislikes[guest1].append(guest2)

self.dislikes[guest2].append(guest1)

- Purpose: Adds a mutual dislike relationship between `guest1` and `guest2`.

- Parameters: `guest1`, `guest2` - the guests who dislike each other.

- Algorithm:

- Ensures both guests are present in the `dislikes` dictionary.

- Adds each guest to the other's dislike list.

4. `assign\_tables` Method

def assign\_tables(self):

for guest in self.dislikes:

if guest not in self.color:

if not self.is\_bipartite(guest):

return False

return True

- Purpose: Determines if a valid seating arrangement is possible.

- Algorithm:

- Iterates over each guest in the `dislikes` dictionary.

- For each guest not already colored, checks if the subgraph is bipartite.

- If any subgraph is not bipartite, returns `False`.

- If all subgraphs are bipartite, returns `True`.

5. `print\_seating\_arrangement` Method

def print\_seating\_arrangement(self):

table1, table2 = [], []

for guest, col in self.color.items():

if col == 0:

table1.append(guest)

else:

table2.append(guest)

print("Table 1:", " ".join(table1))

print("Table 2:", " ".join(table2))

- Purpose: Prints the seating arrangement if a valid one is found.

- Algorithm:

- Separates guests into two lists based on their color.

- Prints the guests assigned to each table.

Conclusion

The `AuntNamesday` class effectively uses graph theory to determine if a valid seating arrangement is possible based on dislike relationships. It leverages depth-first search (DFS) to check for bipartiteness and assigns guests to two tables accordingly. The example provided demonstrates the functionality and the successful application of the algorithm to solve the problem.