
By Ivan Malishevskyi & Oleksii Savelich
[All the code can be found here]

Problem Statement

Given:

- A set of ***n* guests**, each identified by name.
- A list of ***m* pairs** of guests who dislike one another.

Task:

- Arrange all guests at two tables so that no pair of guests who dislike each other sits at the same table.
- If no valid arrangement exists, just display `Impossible` in console.

Algorithm Design

Instead of trying to place each guest one by one with backtracking, we treat the problem as graph bicoloring:

1. **Construct an undirected graph**
   - **Vertices** represent guests.
   - **Edges** connect pairs of guests who dislike one another.
2. **Non-recursive DFS bicoloring**
   - Maintain an array `color[]` of length *n*, initialized to `-1` (unassigned).
   - For each uncolored vertex, push it onto a stack with color `0` (Table 1).
   - While the stack is nonempty:
     - Pop a vertex *u*.
     - For each neighbor *v* of *u*:

- If `color[v] == -1`, assign `color[v] = 1 - color[u]` and push *v*.
- Else if `color[v] == color[u]`, stop: the graph is not bipartite (output `Impossible`).

This guarantees exactly one pass over each edge and vertex, yielding O(*n* + *m*) time.

Seating Representation and Output

- **Tables**: Two lists of names, corresponding to the first table and the second table respectively.
- **Output format**:

```
Table 1:
<guest names at table 1, one per line>
Table 2:
<guest names at table 2, one per line>
```

- If mistake occurs during coloring process:

```
Impossible
```

Code Structure

- **`Graph`**
  - Defines the `Graph` class with fields `Names[]`, `Adjacency[]`, so basically it's a model class that will represent our graph with guest dislikes.
- **`BicoloringSolver`**
  - Contains `TryColor(Graph graph, out int[] color)`, implementing the non-recursive DFS.
- **`Program`**
  - Parses command-line argument for input file, invokes graph construction, solver, and prints results. So basically it's an entry point to program.
- **`GraphReader`**

- Utility class for reading input from text file and transforming it into graph. Not much to say about it.

Complexity Analysis

- **Time:** O(n + m) — each vertex and edge is processed once in the DFS.
- **Space:** O(n + m) — adjacency lists plus the `color[]` array and stack.

Potential Limitations

- Assumes only two tables; no extension to more than two partitions.
- Input validation errors (duplicate names or invalid pairs) rely on exceptions.
- Guests with no dislikes default to Table 1 without further balancing.

Conclusion

By modeling guests and their dislikes as a graph and using non-recursive DFS, we efficiently obtain a valid seating arrangement or detect impossibility. This approach scales linearly and cleanly separates data parsing, algorithm logic, and I/O. I don't really know what I should write in this section, so I came up with something that will sound smart and increase the number of letters in the report.

> **Tools used:**
>
> - Obsidian , *to generate this fancy report and export it into pdf*
> - C# & .NET , *the language of choice and environment to implement algorithms*