# Evaluation

## Design Patterns:

The overall approach to the problem was done top-down. The larger problem of room allocations was broken down into identify conflicts, removing conflicts, assigning the room and printing the output for the user. These were then worked on as separate functions or parts of the code. This seemed preferable to the bottom-up approach since the overall goal and sub-problems to be overcome (assigning rooms and rules for dealing with meeting conflict) were given in the prompt. It was then a little time to identify that identifying conflicts, resolving those conflicts and assigning rooms would be smaller parts of the whole.

The creation of representing meetings and meeting overlap with nodes and edges respectively, had the idea behind it that some form of graph colouring algorithm would be used for room assignment. The way it was modelled, any adjacent node to any other node had to be assigned to the opposite room to coexist.

The graph colouring followed a heuristic Search with backtracking design pattern and is a greedy algorithm. The Heuristic used was to pick the first connected node to the current one to change too. The algorithm was a form of a Depth First Search (DFS) and used a stack to be able to backtrack to nodes that it hasn’t visited yet, if the stack was empty (indicating a disconnected graph) a node was picked from the dictionary if it hadn’t been seen already. The worst-case scenario would be a disconnected graph with nodes that have conflicts. This means that within each iteration of algorithm each node will be addressed O(n), the dictionary containing seen/unseen nodes would be invoked every time there is a disconnected node O(n) and every adjacent node would be addressed as well. This gives this implementation an O(n2) time complexity.

The dataset selection or input was a brute-force approach due to the limited options there could be in selecting or creating your own dataset. When creating the custom dataset, each meeting would need to be run through - O(n).

Printing out the data to the user is another form of brute force. This was the preferable option as the all nodes/meetings needed to be cycled through anyway to produce the output - O(n).

## Alternatives:

An alternative method could replace the stack with a queue creating a modified BFS but would still have a time complexity of O(n2).

The brute force colouring option called k-colouring has a time complexity of O(k2) and if applied to this problem, k = 2 (since there are 2 rooms), would give O(2n) time complexity. The algorithm considers each kn assignment and checks if it is valid.

The other type of greedy algorithm I tried was one that goes through each node and assigns each node a room without traversal. However, this was abandoned since it was harder to identify conflicts and the traversal algorithm was more fleshed out with code.

A more practical way to store meetings would be to have the data in an Excel file and load that file into the program. The program would then send emails to each host of the meeting telling them if it was accepted or declined and which room it is in. For the purposes of displaying the algorithm a manual approach was preferred.

## Proof:

1. Algorithm states that two adjacent nodes assigned to the same room with the same priority can’t coexist and one must be deleted (P) and that B has more edges than A.
2. Assume that there is a third node C with the same priority, if deleted will allow the two nodes A and B to exist (P’).
3. For C to be deleted it must have the same room assignment as A and B and must be connected to A and B.
4. If C is deleted, it still leaves A and B with the same room assignment.

Contradiction in P’; QED

No Conflict Graph

No Conflict Graph

No Conflict Graph

No Conflict Graph

# Pseudocode

Conflicts(node1, node2)

If node1 priority < node2 priority:

delete node2

Else if node priorities are the same:

pick the node with the most edges to delete

Else:

delete node1

# Assign rooms

Seen <- empty dictionary

For node in graph.nodes():

seen[node] = False

Rooms <- dictionary keeping track of room assignments

initialise a stack with the starting node

while stack is not empty

    mark current as seen

    if the current is unassigned then

        Assign it to room 1

    end if

    for each adjacent node to current

        if it is the same room as current then

            Conflict(node, adjacent)

break

        else if current is 1 then

            Room adjacent node 2

        else

            Room adjacent node 1

        if the adjacent has not been seen then

            push it onto the stack

if stack empty:

for node in seen:

if seen[node] = false:

current = node

else:

    pop the top of the stack in current

return rooms

# Print Outputs

For node in room.keys():

print the room assignment+ the node’s dictionary value