**algorithm.py :**

"""

    This function implements an evolutionary algorithm to generate an optimal timetable for a bachelor defense scheduler.

    It generates a solution by creating the first timetable randomly

    then changes the new solution by calling the neighbor function from neighboring file.

    It calculates the cost for the solution

    if the cost for the new solution is less than or equal to the cost of the current solution

    Then it changes the value of the solution to the new solution.

    It checks for hard constraints such as the number of examiners, supervisors, and classrooms per slot and continuity of the schedule.

    """

Here is a step-by-step explanation of the function:

1. The function starts by setting some parameters, such as the maximum number of generations and the number of runs (iterations).
2. It loads input data from a JSON file using a function called **load\_data** and extracts relevant information about examiners, supervisors, rooms, dates, and slots.
3. The **neighbor** function is imported from the neighboring file, which is responsible for making small changes (neighborhood exploration) to the current solution.
4. The function then enters a loop to run the evolutionary algorithm for the specified number of runs.
5. For each run, it generates an initial solution randomly by calling the **generate\_solution** function with the loaded data.
6. The algorithm proceeds with iterations, each time exploring the neighborhood of the current solution by calling the **neighbor** function.
7. It calculates the cost of the current solution (a measure of how well it satisfies the constraints) using the **cost\_function** and stores it in the variable **ft**.
8. It then calculates the cost of the new solution generated by the **neighbor** function, stores it in the variable **ftn**.
9. If the cost of the new solution (**ftn**) is less than or equal to the cost of the current solution (**ft**), the current solution is updated to be the new solution.
10. The algorithm continues to explore the neighborhood until either the maximum number of generations is reached or an optimal solution is found (where the cost is 0, indicating that all constraints are satisfied).
11. The best timetable found among all runs is stored in the variable **best\_timetable**.
12. The function prints information about the best solution, including its cost and the number of examiners, supervisors, and rooms assigned per slot. It also checks if hard constraints for examiners, supervisors, rooms, and continuity are satisfied.
13. After finding the best solution, the function creates a schedule for each examiner, assigning rooms for their respective slots and storing it in a variable called **solution**.
14. Finally, the function converts the time format in the **solution** to a more readable format, saves the solution as a JSON file, and creates an output file using the **Outputcreation.Create\_output()** function.

**neighboring.py :**

The function takes several parameters: **solution**, **flag1**, **days**, and **slots**. The **solution** is a data structure containing information about the current state of the scheduling, and the other parameters define constraints and properties of the scheduling problem.

Let's break down the function step-by-step:

1. **candidates = []**: Initialize an empty list to store potential candidates for making changes to the current schedule.
2. The function starts with three main loops, each examining different aspects of the scheduling:

a. The first loop checks for examiners who are assigned to too many or too few time slots in a day and appends the corresponding entries to the **candidates** list.

b. The second loop calculates the number of working days for each examiner and adds their entries to **candidates** if they exceed a certain limit (more than 2 days).

c. The third loop iterates through all the time slots and checks if the number of assigned rooms in a slot is greater than the total number of rooms available. If so, it appends those slots to **candidates**.

1. The function then checks various conditions for each entry in the **solution** data structure to identify potential issues with the current schedule and adds the problematic entries to **candidates** along with the reasons for their inclusion (e.g., "continuity," "examinerRepeat," "supervisorRepeat," etc.).
2. The **reason** list is used to keep track of the reasons why a candidate is being considered.
3. If there are no candidates or the **flag1** parameter is **True**, a random index is selected from the **solution** data structure. Otherwise, a random entry is chosen from the **candidates** list, and its index is stored in **i**.
4. The function then retrieves information about the selected entry (time, examiner, supervisor, and room) from the **solution** data structure.
5. The assignment for the selected entry is removed from the corresponding time slot, examiner, and supervisor.
6. The function proceeds to find a suitable replacement time slot (**r**) for the selected entry by checking various conditions and constraints. It aims to find a slot that satisfies examiner constraints and maintains continuity in the schedule.
7. The function then checks if the selected examiner and room can be swapped to another slot while satisfying constraints. If this is possible, the examiner is reassigned to that slot, and the function returns the updated **solution**.
8. If the above swap is not possible, the function proceeds to find a random available slot (**r1**) that satisfies the constraints and assigns the selected examiner and room to that slot.
9. Finally, the updated **solution** is returned.