IMPLEMENTATION OF TABU SEARCH FOR JOB SCHEDULING

Summary

Introduction

This report outlines the implementation of the Tabu Search algorithm for solving a job scheduling problem. The problem involves scheduling a set of jobs to minimize total tardiness, where tardiness is defined as the amount of time by which a job's completion time exceeds its due date.
The objective is to determine the optimal schedule of 6 jobs on a machine that minimizes global job tardiness. Each job has a processing time p_j, a due date d_j, and a penalty for tardiness equal to $w_j, j = 1, \dots, 6$. The objective function (to be minimized) is computed as:

$$T = \sum_{j=1}^{6} w_j [C_j - d_j]^+$$

where C_j is the completion time of job j, while for each value v, $[v]^+ = max(0, v)$. Completion time C_j is computed as the sum of the processing time p_j of job j, and the processing times of all the jobs that have been scheduled before j.
Each job has the following data:

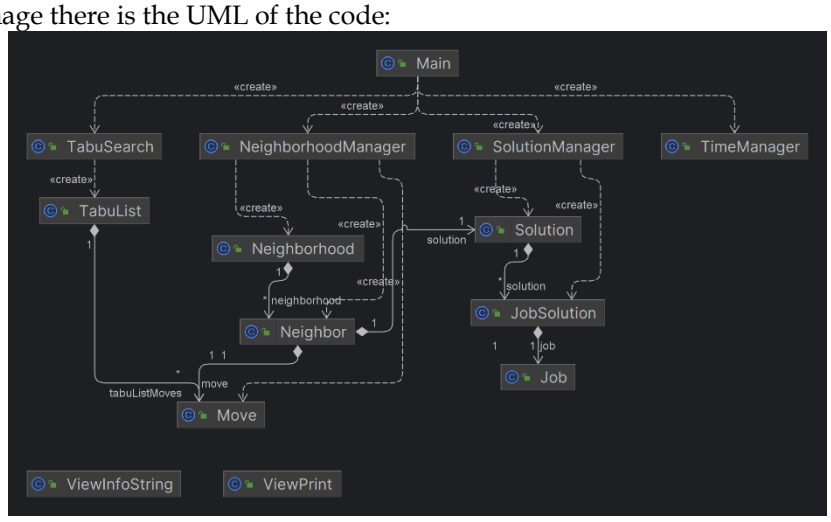
Job	w_j	p_j	d_j
1	1	6	9
2	1	4	12
3	1	8	15
4	1	2	8
5	1	10	20
6	1	3	22

Algorithm overview

The algorithm proceeds through the following steps:

1. **Initialization:**
   - Instantiate objects for `SolutionManager`, `NeighborhoodManager`, and `TabuSearch`;
   - Define algorithm parameters such as tabu list size and maximum number of iterations.
2. **Initial solution creation:**
   - Generate an initial feasible solution using `SolutionManager`;
   - The initial solution can be generated *randomly* or by *ordering* jobs based on their p-values.
3. **Tabu Search execution:**
   - Execute the main loop until a stopping criterion is met;
   - At each iteration, generate the neighborhood of the current solution using `NeighborhoodManager`;
   - Select a new neighbor solution that does not violate the tabu list rules, as determined by `NeighborhoodManager`;
   - Update the current solution with the new neighbor solution;
   - Print information about the newly found neighbor solution and the current tabu list.
4. **Information printing:**
   - Print important information during the algorithm execution, such as the initial solution, found neighbors, total time used, and times the algorithm reaches the optimal solution.
5. **Termination:**
   - Terminate the algorithm when the maximum number of iterations is reached.

In the following image there is the UML of the code:



Technical choices

An **initial feasible solution** can be determined by the `SolutionManager` class using the `createInitialFeasibleSolution` method. This method generates an initial solution by either randomly shuffling the jobs or ordering them based on their processing times (`p-values`). The choice between random shuffling and ordering by `p-values` is determined by the `choice` parameter passed to the method.

The **move** that determines the neighborhood is defined by the `Move` class. In this implementation, a move involves swapping the positions of two jobs in the job sequence. This swap operation is implemented in the `moves` method of the `Move` class.

In this implementation, the **attribute that becomes tabu** is the move itself. Once a move is performed, it is added to the tabu list to prevent the algorithm from revisiting the same move in subsequent iterations. The tabu list is managed by the `TabuList` class, which ensures that moves violating the tabu tenure are not selected as neighbors.

The **tabu tenure**, which determines how long a move remains tabu, is defined by the parameter `tabuListSize` passed to the `TabuList` constructor. This parameter, in my test equals to 3, specifies the maximum size of the tabu list. When the tabu list reaches its maximum size, the oldest move is removed from the list to make room for new moves. This mechanism ensures that moves remain tabu for a limited duration, preventing premature convergence to suboptimal solutions.

Performance Testing

Performance testing was conducted to evaluate the effectiveness of the Tabu Search algorithm in finding optimal solutions for the job scheduling problem.

Two scenarios were tested: one with a randomly generated initial solution and the other with a deterministic initial solution based on job processing times.

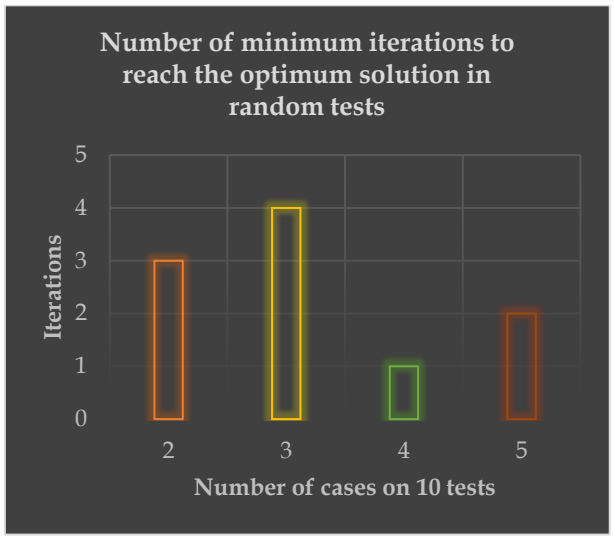
Hardware Specifications

The tests were performed on a machine with the following specifications:

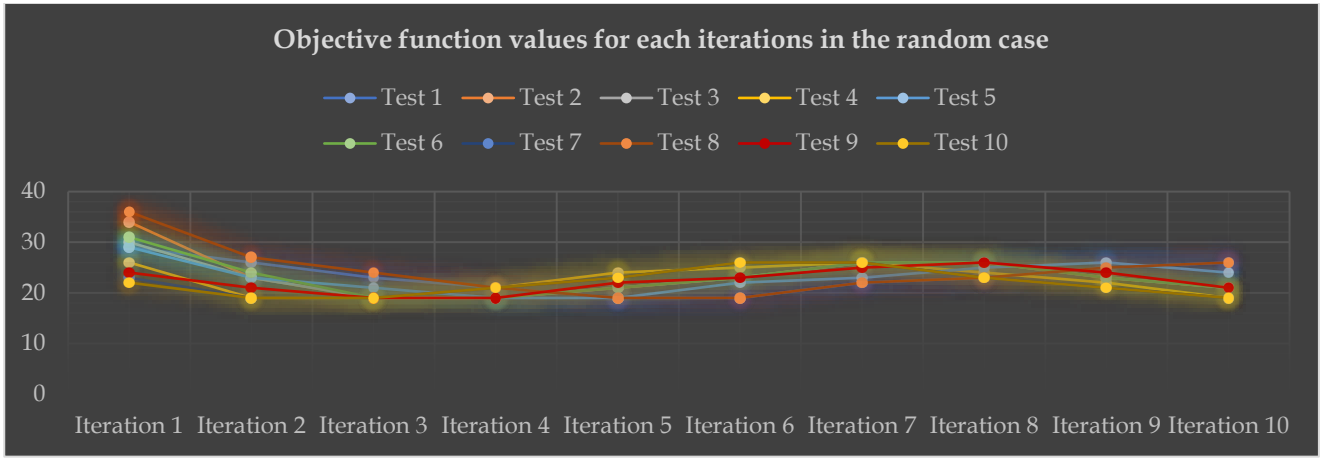
- Processor: AMD Ryzen 5 5600H with Radeon Graphics, 3.30 GHz
- RAM: 16,0 GB
- Operating System: Windows 11 Home

Test Results

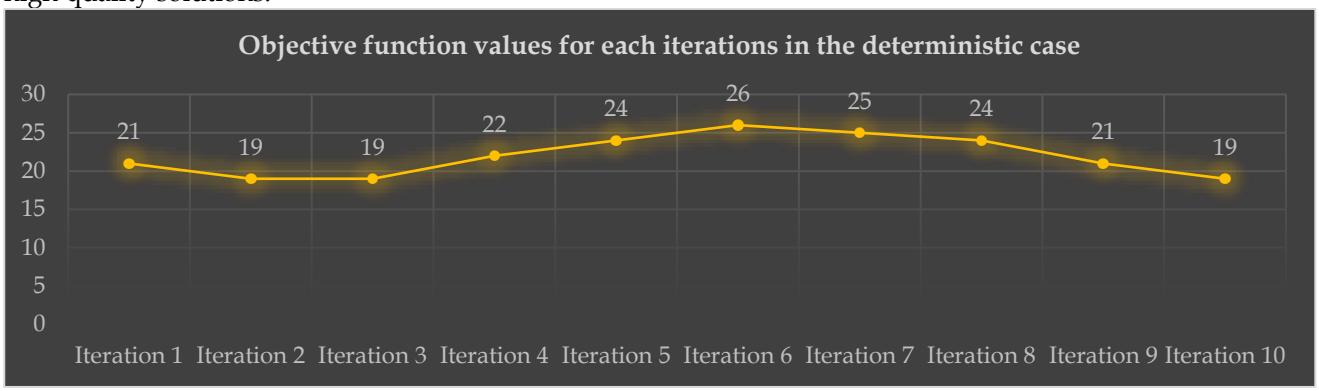
For the **random initial solution**, it was observed that the algorithm efficiently discovered high-quality solutions within a reasonable time frame, ranging between 706800E-9 and 6196200E-9 seconds, with iterations typically ranging from 2 to 5.



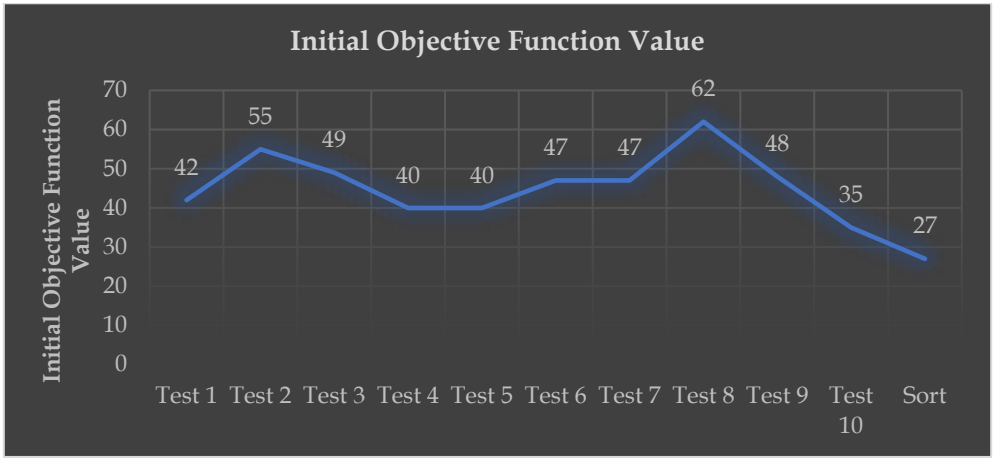
The objective function exhibited a notable decrease over these iterations, indicating a gradual enhancement in the solution's quality, aligning with the intended behaviour of the algorithm.

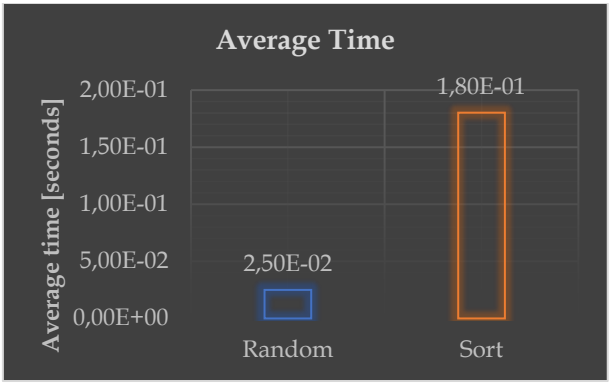


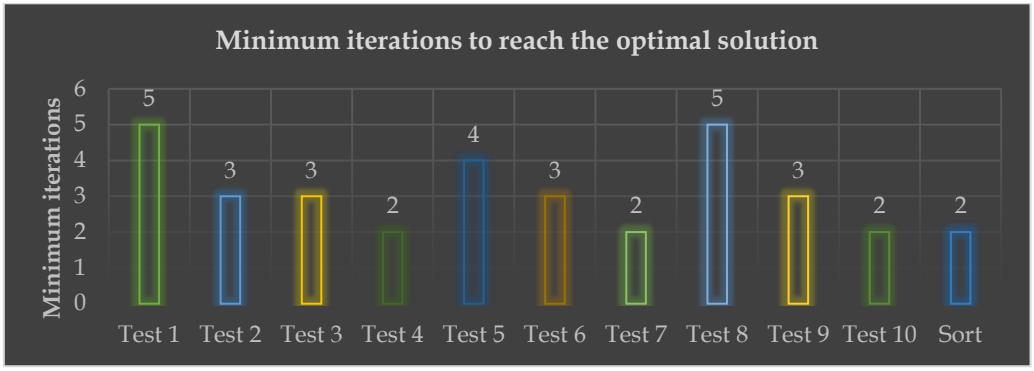
In contrast, the **deterministic initial solution**, obtained by sorting the jobs based on their processing times, led to optimal or near-optimal solutions in as few as 2 iterations, surpassing the performance of the random solution. This underscores the notion that a structured initial solution can accelerate the convergence towards high-quality solutions.

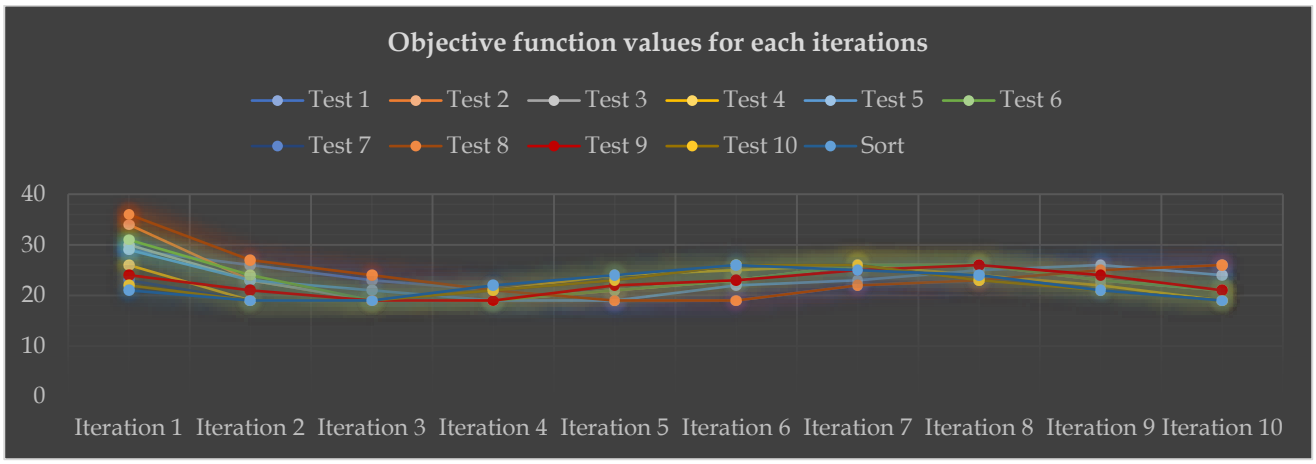


In both scenarios, the algorithm adeptly managed the constraints imposed by the tabu list and effectively explored the solution space. However, it's worth noting that performance might vary depending on the specific configuration of initial jobs.
After comparing the results of the random and deterministic scenarios, the following values were obtained for the initial objective function value, average time, minimum iterations to reach the optimum, and objective function values over the 10 iterations.









Finally, below is a table showing the time, in seconds, corresponding to the iteration in which, for each test, the optimal solution was found:

	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7	Iteration 8	Iteration 9	Iteration 10
Test 1	N/A	N/A	N/A	N/A	**1,86E-01**	**1,88E-01**	N/A	N/A	N/A	N/A
Test 2	N/A	N/A	**2,2018E-03**	**3,32E-03**	N/A	N/A	N/A	N/A	N/A	N/A
Test 3	N/A	N/A	**1,85E-03**	**2,57E-03**	N/A	N/A	N/A	N/A	N/A	N/A
Test 4	N/A	**1,17E-03**	**1,62E-03**	N/A	N/A	N/A	N/A	N/A	N/A	**4,77E-03**
Test 5	N/A	N/A	N/A	**1,86E-03**	**2,27E-03**	N/A	N/A	N/A	N/A	N/A
Test 6	N/A	N/A	**1,69E-03**	**2,11E-03**	N/A	N/A	N/A	N/A	N/A	N/A
Test 7	N/A	**8,64E-04**	**1,42E-03**	N/A	N/A	N/A	N/A	N/A	N/A	**6,20E-03**
Test 8	N/A	N/A	N/A	N/A	**3,79E-03**	**4,76E-03**	N/A	N/A	N/A	N/A
Test 9	N/A	N/A	**1,62E-03**	**2,04E-03**	N/A	N/A	N/A	N/A	N/A	N/A
Test 10	N/A	**7,07E-04**	**1,29E-03**	N/A	N/A	N/A	N/A	N/A	N/A	**4,72E-03**
Sort	N/A	**1,67E-01**	**1,69E-01**	N/A	N/A	N/A	N/A	N/A	N/A	**1,79E-01**

N/A means that in that specific test at that iteration the optimal solution was not achieved.

Additional notes

As evident from the results, often immediately after reaching an optimal solution, another optimal solution is achieved in the subsequent iteration. This occurs because the problem has two optimal solutions, and it is possible to obtain one by making only one swap from the other.

The two optimal solutions in question are: (1 4 2 3 6 5) and (4 1 2 3 6 5), both with an optimal value of 19.

In the appendix, it's possible to find the excel file containing all the data related to the conducted tests.