Decision Making with Constraint Programming

Exercise 4

Dessì Leonardo [0001141270]

Spini Riccardo [0001084256]

Corso di laurea magistrale in Informatica

Alma Mater Studiorum
Università di Bologna

December, 2023

1 | Resource Constrained Project Scheduling Problem (RCPSP)

	Default search		Smallest search	
	Time	**Makespan**	**Time**	**Makespan**
data1	0.34 s	90	0.33 s	90
data2	0.76 s	53	300 s	54
data3	300 s	82	300 s	75

Table above reports the objective value and the execution time of three instances for the Resource Constrained Project Scheduling Problem.
The RCPSP problem consists of: a set of cumulative resources, a set of tasks with duration and resource requirements, and precedence constraints between some tasks. We want to execute each task in a way that minimizes the makespan (the line that defines the end of the scheduling), subject to precedences and cumulative resource constraints.
The constraints of this problem are:

- process p_i must start before process p_j with a preemptive right defined by the graph in figure 1.1.
- the resources are cumulative.

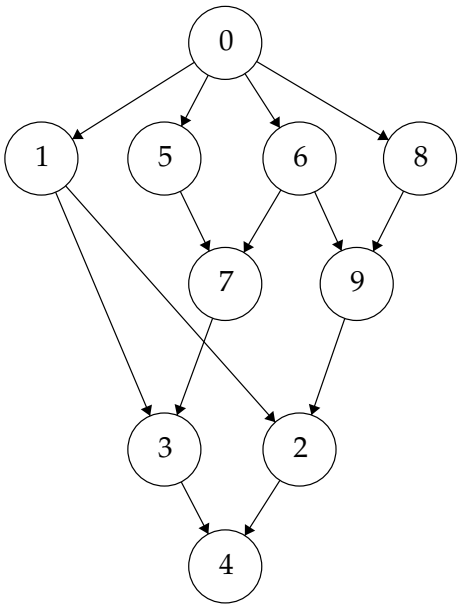

Figure 1.1: Precedence graph for *rcpspData1*

The goal of RCPSP is to minimize makespan. In our case, the makespan is defined as:

$$\max S_i + D_i$$

where i is between 1 and n_{task}.

- *smallest* is a heuristic that chooses the variables with the smallest value in their domain first. In our case, those with the smallest value are those that have the time window starting earlier and therefore the Earliest Start Time smaller.

As we can see from the results table, we can see that there are no sensitive differences between the two search methods for *data1*. This is probably caused by the small size of the instance. Therefore, we cannot say which of the two methods is better than the other. Calculating the failures we notice that *default search* has more than *smallest search*, but the latter has no perceptible improvement given the small instance.
In *data2*, on the other hand, we notice that there is a substantial difference in terms of time although the value of the objective function is very similar. Indeed, the *default search* immediately finds the optimal solution.
With the default search, Geocode 6.3.0 chooses independently which combination of search methods to use. So we do not know which methods are chosen because Geocode 6.3.0 does not say which combination of search methods it uses to find the best solution in less than a second. Perhaps the reason that the default search is faster is related to the problem instance, or the fact that it somehow succeeds faster in proving optimality
Regarding the *smallest search*, when the solver finds a solution and wants to improve it, it has to go back to the root node completely to improve it and reach the objective value. This process of backtracking and thus new reassignments to variables can lead to a large increase in time while still finding (if there were no time limit at 5 minutes) the smallest makespan value.
Regarding the results obtained in *data3*, for the same amount of time we obtained a better makespan with the smallest search.

In conclusion, in our opinion having such discordant results we cannot say that smallest search is better than default search and vice versa, so we think it may depend on how the instances are structured.
Finally, a graphical representation was created with python of the optimal solution of the *rcpspData1* problem. In addition, we noticed that task 0 does not consume any resources however it has its own duration and precedence constraints (Figure 1.1) do not allow other tasks to overlap with task 0, that is why all drawn tasks start after 10

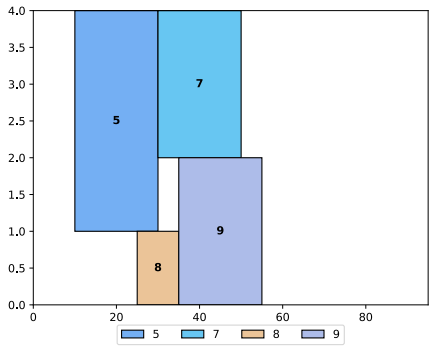

Figure 1.2: Resource 1 Cumulative chart for *rcpspData1*

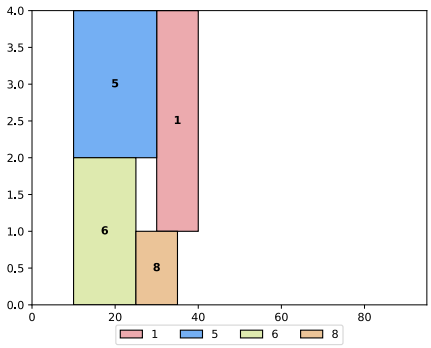

Figure 1.3: Resource 2 Cumulative chart for *rcpspData1*

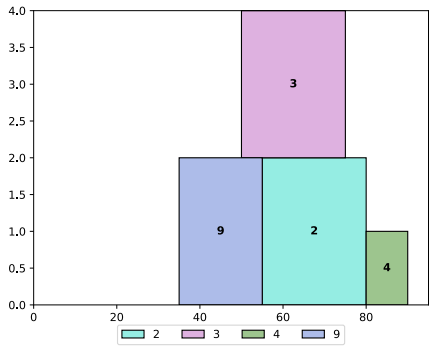

Figure 1.4: Resource 3 Cumulative chart for *rcpspData1*

2 | Job Shop Scheduling Problem (JSP)

	Default search		Smallest search	
	Time	**Makespan**	**Time**	**Makespan**
jobshop1	0.36 s	663	2.37 s	663
jobshop2	300 s	846	300 s	921

Table above reports the objective value and the execution time of two instances for the Job Shop Scheduling Problem.

In JSP problem we have a set of machines and a set of jobs, each composed of a sequence of tasks/orders, each requiring a different machine and machine requirements and machine-dependent durations of the job tasks. In this problem, unlike the RCPSP, we use a disjunctive constraint. This requires that the tasks do not overlap in time.

As in the previous exercise, our goal is to minimize the makespan.
As we can see from the table, the *default search* has better performance in terms of time in the first instance and in terms of reaching the best objective value in the second instance, than the *smallest search* method.
Looking at the results, **we realized that the smallest search is a good way to find a solution, but not the optimal solution**. This is because the smallest search follows a more greedy approach. To improve the results we think it would be useful to use a postponing strategy. The *smallest search* may be problematic for finding the optimal solution due to the following reasons:

- Greedy decision making: Prioritizing tasks with the smallest starting times may lead to a greedy approach where short-term gains are prioritized without considering the long-term consequences. This can result in not optimal scheduling decisions.

A postponing strategy, on the other hand, aims to delay certain tasks' scheduling to explore alternative sequences and resource allocations. This strategy is often beneficial for finding optimal solutions because:

- we suppose that postponing decisions allows for a more global consideration of the project schedule. It could enable the algorithm to explore different combinations of task sequences, resource allocations, and start times, potentially identifying more efficient schedules.
- it could give the algorithm the flexibility to explore different branches of the solution space, increasing the probability of finding better solutions. This exploration is essential to avoid local optima and achieve a more complete search.

Finally, we implemented a graphical visualization in Python to display the result of *jobshop1* (Figure 2.1). Each color corresponds to a job and each row to a machine. Each job has a task in each machine, and the various tasks must be executed sequentially

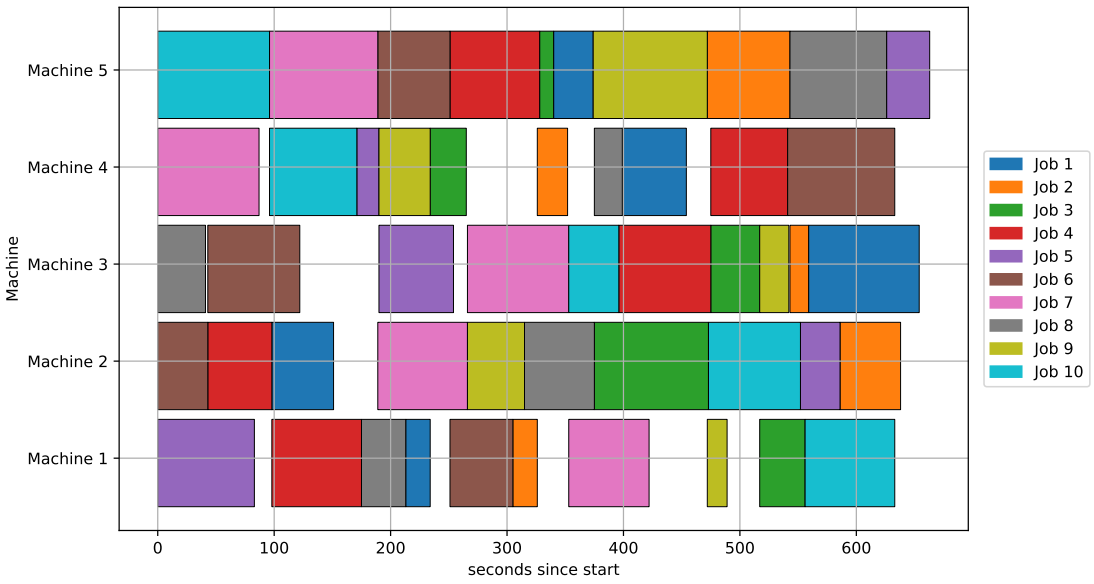

Figure 2.1: Best solution Gantt diagram of *jobshop1*

Is searching on the smallest (earliest) start times is always a good idea?

No, it is not always a good idea!
This is because the choice of task execution orders and start times can affect overall performance and the search for optimal objective value. While minimizing start times can help reduce makespan, there are other factors to consider, such as resource constraints in this particular problem. Indeed, some resources are more difficult to "schedule" so trying to schedule them as early as possible could lead to increased competition for resources, generating delays and increasing the probability of violating resource constraints. In conclusion, the choice of execution orders and start times depends on the specific nature of the problem, resource constraints, and project objectives.