National Institute of Statistics and Applied Economics

Master of Information Systems and Intelligent Systems

INSEA

Single-Machine Jobs scheduling for $1||\sum U_i$ and $1||\sum \alpha E_i + \beta T_i$

Realised By :

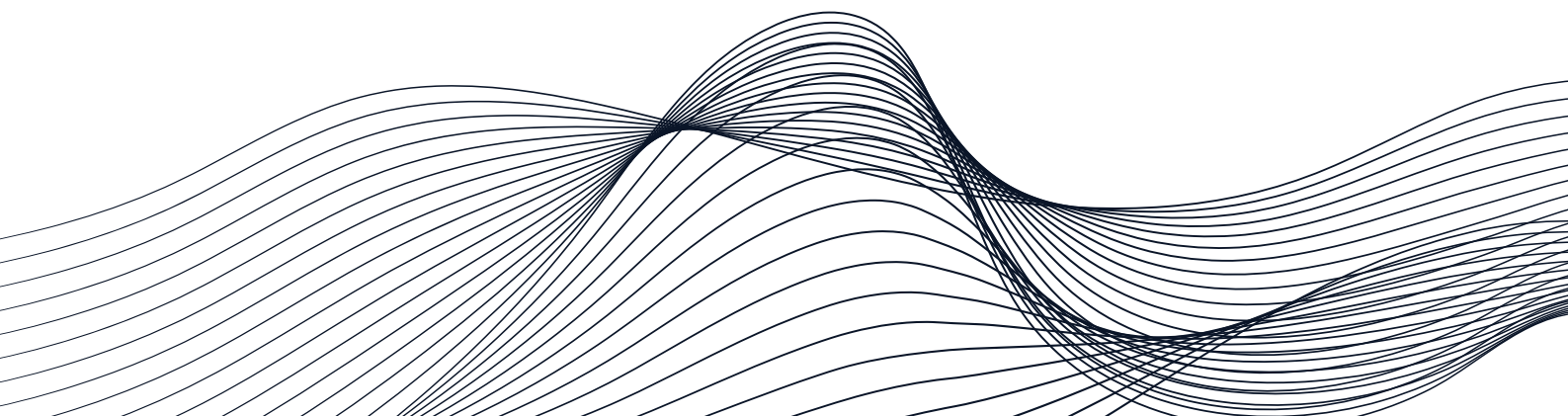
ABDELLATIF AHAMMAD

MOHAMED OURAHOU

Supervised by :

Prof. RACHID BENMANSOUR

Academic Year : **2021/2022**



Abstract

Given a set of n Jobs, each of which is assigned a processing time and a due-date, all these jobs are simultaneously available to be processed on a single machine, This project aims at presenting heuristics to a single-machine scheduling environment for two problems; the first is to find the optimal job scheduling order that minimizes the total number of latey jobs and the seconde is to minimizes the total tardiness and earliness penalties sum. Since these problems have been shown to be NP-complete, we present DP (dynamic programming) formulation of them and apply the principle of its optimality to find the optimal solution by implicit enumeration, we present also a polynomial approach known as Moore-Hodgson algorithm and a recent metaheuristic of General Variable Neighborhood Search (GVNS).
Finally we discuss the benchmark and the accuracy (perfermance) of these problems and give some explanations about that.

Key words: Scheduling problems, Single machine, EDD, Metaheuristics, VND, GVNS

Acknowledgements

We would like to express our special thanks of gratitud to our beloved teacher **Rachid BEN-MANSOUR** who gaves us the golden oppurtunity to do this wonderful project on the topic **Single-Machine Jobs scheduling for $1||\sum U_i$ and $1||\sum \alpha_i E_i + \beta_i T_i$**, which also helped us in doing a lot of research and we came to know about so many new things.

Contents

List of Figures

List of Algorithms

Listings

Chapter 1 Overview

Scheduling problems are one of the clasical combinatorial optimisation problems concerned with the optimal allocation of scarce resources to activities over time.

The practice of this field dates to the first time two humans contended for a shared resource and developed a plan to share it without bloodshed,after that these problems have been studied and mainstreamed to cover a variety of settings such as flexible manufacturing systems, production planning, airline industry, etc.

More generally, scheduling problems involve jobs that must be scheduled on machines subject to certain constraints to specify a schedule that specifies when and on which machine each job is to be executed and also optimize some objective functions based on some criterions, like the criterion of giving penalties to jobs which are completed before the due date as well as those completed after the due date.

Even the simplest models that deal with these problems have been studied, a number of algorithms has been proposed and many heuristics have been developed, they are still known to be **NP-complet.**

1.1 The Framework of Basic Problems

A scheduling problem is defined by three separate elements by Graham, Lawler, Lenstra and Rinnooy Kan **(1979)** and it is denoted by $\alpha|\beta|\gamma$ where α denotes **the machine environment**, β denotes a **set of side constraints and characteristics** and γ denotes an **the optimality criterion**.

1.1.1 The machine environment (α)

1.1.1.1 The single machine scheduling ($\alpha = 1$)

In all of scheduling problems we begin with a set J of n jobs, numbered 1; : : : ; n. In the single machine environment we have one machine that can process at most one job at a time. Each job j has a processing requirement pj ; namely, it requires processing for a total of P_j units of time on the machine. If each job must be processed in an uninterrupted fashion, we have a nonpreemptive scheduling environment, whereas if a job may be processed for a period of time, interrupted and continued at a later point in time, we have a preemptive environment.
A schedule S for the set J specifies, for each job j, which P_j units of time the machine uses to process job j. Given a schedule S, we denote the completion time of job j in schedule S by C_j^S. The goal of a scheduling algorithm is to produce a "good" schedule, but the definition of "good" will vary depending on the application:may be processing the entire batch of jobs as quickly as possible, or, in other words, to minimize the completion time of the last job

finished in the schedule. Therefore, given a set of jobs and a machine environment, we must specify an optimality criterion; the goal of a scheduling algorithm will be to construct a schedule that optimizes this criterion.
Scheduling problems arise for a variety of variants, as is illustrated by the following examples:

- The problem $1||\sum C_j$ aims to minimize the sum of completion times. It can be solved optimally by the Shortest Processing Time First rule (SPT): the jobs are scheduled by ascending order of their processing time .

- The problem $1||\sum w_j C_j$ aims to minimize the weighted sum of completion times.

- The problem $1|chains|\sum w_j C_j$ is a generalization of the above problem for jobs with dependencies in the form of chains.

- The problem $1||L_{max}$ aims to minimize the maximum lateness. For each job j, there is a due date d_j. If it is completed after its due date, it suffers lateness defined as $L_j = C_j - d_j$

- The problem $1|prec|h_{max}$ generalizes the $1||L_{max}$ in two ways: first, it allows arbitrary precedence constraints on the jobs like that a job is available at time t if all its predecessors have completed processing by time t; second, it allows each job to have an arbitrary cost function h_j, which is a function of its completion time (lateness is a special case of a cost function).

- The problem $1|r_j|L_{max}$ generalizes the $1||L_{max}$ by allowing each job to have a different release time by which it becomes available for processing. The presence of release times means that, in some cases, it may be optimal to leave the machine idle, in order to wait for an important job that is not released yet. Minimizing maximum lateness in this Variants setting is NP-hard.

- The problem $1||\sum U_j$ aims to minimize the number of late jobs which we will discuss in first problem.
- The problem $1||\sum w_j U_j$ aims to minimize the weight of late jobs. It is NP-hard.

1.1.1.2 More Complex Machine Environments : ($\alpha > 1$)

1. **shop environments**

   In the shop environment, which primarily models various sorts of production environments, In this setting we have m machines and a job j is made up of operations, with each operation requiring processing on a specific one of the m machines. Diferent operations may take diferent amounts of time (possibly 0).

   On shop environemnt we there is three types:

   - **O: Open-shop environment:** the operations of a job can be processed in any order, as long as no two operations are processed on diferent machines simultaneously.
   - **J: Job-shop environment:** In the job-shop environment, there is a total order on the operations of a job, and one operation cannot be started until its predecessor in the total order is completed.
   - **F: Flow-shop environment:** is a special case of the job-shop which the order of the operations is the same—each job requires processing on the same machines and in the same order, but different jobs may require different amounts of processing on the same machine. Typically in the flow shop and open shop environment, each job is processed exactly once on each machine.

2. **parallel machines**

   A parallel machines problem is a generalisation of a job-shop problem to the case when there are identical machines of the same type. Job-shop problems encountered in a flexible manufacturing system, train timetabling, production planning and in other real-life scheduling systems a job-shop problem is modelled via a weighted mixed graph with a conflict resolution strategy used for finding an appropriate schedule Having introduced all of the optimality criteria.

   In these environments we are given m machines. A job j with processing requirement p_j can be processed on any one of the machines, or, if preemption is allowed, started on one machine, and when preempted potentially continued on another machine. A machine can process at most one job at a time and a job can be processed by at most one machine at a time. In the unrelated parallel machines environment we model machines that have different capabilities and thus their relative performance on a job is unrelated. In other words, the speed of machine i on job j, s_{ij} , depends on both the machine and the job; job j requires p_j/s_{ij} processing time on machine i. We define $p_{ij} = p_j/s_{ij}$.

- **P: Identical-machines scheduling:** In the identical parallel machine environment the machines are identical, and a job j requires p_j units of processing time when processed on any machine. In the uniformly related machines environment each machine i has a speed $S_i > 0$, and thus job j, if processed entirely on machine i, would take a total of p_j/S_i time to process.
- **Q:Uniform-machines scheduling:** Each machine i has a speed $s_i > 0$, and thus job j, if processed entirely on machine i, would take a total of p_j/s_i time to process.
- **R:Unrelated-machines scheduling:** In the unrelated parallel machines environment we model machines that have different capabilities and thus their relative performance on a job is unrelated. In other words, the speed of machine i on job j, s_{ij}, depends on both the machine and the job; job j requires p_j/s_{ij} processing time onmachine i. We define $p_{ij} = p_j/s_{ij}$.

These letters might be followed by the number of machines, which is then fixed. For example, P2 indicates that there are two parallel identical machines. Pm indicates that there are m parallel identical machines, where m is a fixed parameter.

1.1.2 Constraints and characteristics β

β is a subset of r_j, prec, and pmtn (job preemption allowed.), where these denote:

1.1.2.1 Job characteristics

All processing times are assumed to be integers. In some older research papers however they are assumed to be rationals.

- $p_i = p$, or $p_{ij} = p$: the processing time is equal for all jobs.
- $p_i = 1$, or $p_{ij} = 1$: the processing time is equal to 1 time-unit for all jobs.
- r_j: for each job a release time is given before which it cannot be scheduled, default is 0.
- $online - r_j$: an online problem. Jobs are revealed at their release times. In this context the performance of an algorithm is measured by its competitive ratio.
- d_j: for each job a due date is given. The idea is that every job should complete before its due date and there is some penalty for jobs that complete late. This penalty is denoted in the objective value. The presence of the job characteristic is implicitly assumed and not denoted in the problem name, unless there are some restrictions as for example , assuming that all due dates are equal to some given date.
- $\bar{d}_j$: for each job a strict deadline is given. Every job must complete before its deadline.

- **pmtn**: Jobs can be preempted and resumed possibly on another machine. Sometimes also denoted by 'prmp'.
- $size_j$: Each job comes with a number of machines on which it must be scheduled at the same time. The default is 1. This is an important parameter in the variant called parallel task scheduling.

1.1.2.2 Precedence relations

:might be given for the jobs, in form of a partial order, meaning that if i is a predecessor of i' in that order, i' can start only when i is completed.

- **Prec:** Given general precedence relation. If $i < j$ then starting time of should be not earlier than completion time of .
- **Chains:** Given precedence relation in form of chains (indegrees and outdegrees are at most 1).
- **Tree:** Given general precedence relation in form of a tree, either intree or outtree.
- **intree:** Given general precedence relation in form of an intree (outdegrees are at most 1).
- **outtree:** Given general precedence relation in form of an outtree (indegrees are at most 1).
- **opposing forest:** Given general precedence relation in form of a collection of intrees and outtrees.
- **sp-graph:** Given precedence relation in form of a series parallel graph.
- **bounded height:** Given precedence relation where the longest directed path is bounded by a constant.
- **level order:** Given precedence relation where each vertex of a given level l (i.e. the length of the longest directed path starting from this vertex is l) is a predecessor of all the vertices of level l-1.
- **interval order:** Given precedence relation for which one can associate to each vertex an interval in the real line, and there is a precedence between x and y if and only if the half open intervals $x = [s_x, e_x)$ and $y = [s_y, e_y)$ are such that e_x is smaller than or equal to s_y.
- **quasi-interval order:** Quasi-interval orders are a superclass of interval orders defined in Moukrim: Optimal scheduling on parallel machines for a new order class, Operations Research Letters, 24(1):91-95, 1999.

- **over-interval order:** Over-interval orders are a superclass of quasi-interval orders defined in Chardon and Moukrim: The Coffman-Graham algorithm optimally solves UET task systems with overinterval orders, SIAM Journal on Discrete Mathematics, 19(1):109-121, 2005.
- **Am-order:** Am orders are a superclass of over-interval orders defined in Moukrim and
- **Quilliot:** A relation between multiprocessor scheduling and linear programming. Order, 14(3):269-278, 1997.
- **DC-graph:** A divide-and-conquer graph is a subclass of series-parallel graphs defined in
- **Kubiak et al.:** Optimality of HLF for scheduling divide-and-conquer UET task graphs on identical parallel processors. Discrete Optimization, 6:79-91, 2009.
- **2-dim partial order:** A 2-dimensional partial order is a k-dimensional partial order for k=2.
- **k-dim partial order:** A poset is a k-dimensional partial order iff it can be embedded into the k-dimensional Euclidean space in such a way that each node is represented by a kdimensional point and there is a precedence between two nodes i and j iff for any dimension the coordinate of i is smaller than or equal to the one of j.

1.1.2.3 Various constraints

- **Rcrc:** Recirculation, also called Flexible job shop. The promise on μ is lifted and for some pairs $k \neq k'$ we might have $\mu_{kj} = \mu_{k'j}$.
- **No-wait**: The operation $O_{k+1,i}$ must start exactly when operation O_{ki} completes. Sometimes also denoted as 'nwt'.
- **No-idle:** No machine is ever idle between two executions.
- $size_j$: Multiprocessor tasks on identical parallel machines. The execution of job is done simultaneously on $size_j$ parallel machines.
- : fix_j Multiprocessor tasks. Every job is given with a set of machines $fix_j \subseteq 1, ..., m$, and needs simultaneously all these machines for execution. Sometimes also denoted by 'MPT'.
- M_j: Multipurpose machines. Every job needs to be scheduled on one machine out of a given set $M_j \subseteq 1..., m$. Sometimes also denoted by $'M'_j$.
- ...etc

1.1.3 Optimal criterions (γ)

Usually the goal is to minimize some objective value. One difference is the notation $\sum U_i$ where the goal is to maximize the number of jobs that complete before their deadline. This is also called the throughput. The objective value can be sum, possibly weighted by some given priority weights w_j per job.

- -: The absence of an objective value is denoted by a single dash. This means that the problem consists simply in producing a feasible scheduling, satisfying all given constraints.
- C_j: the completion time of job j .C_{max} is the maximum completion time; also known as the makespan. Sometimes we are interested in the mean completion time (the average of C_j over all j), which is sometimes denoted by mft (mean finish time).
- F_j: The flow time of a job is difference between its completion time and its release time, i.e. $F_j = C_j - r_j$.
- L_j: Lateness. Every job is given a due date d_j. The lateness of job j is defined as $F_j = C_j - d_j$. Sometimes L_{max} is used to denote feasibility for a problem with deadlines. Indeed using Various constraints Objective functions binary search, the complexity of the feasibility version is equivalent to the minimization of L_{max}.
- U_j: Throughput. Every job is given a due date d_j . There is a unit profit for jobs that complete on time, i.e. $U_j = 1$ if $C_j \leq d_j$ and $U_j = 0$ otherwise. Sometimes the meaning of U_j is inverted in the literature, which is equivalent when considering the decision version of the problem, but which makes a huge difference for approximations.
- T_j: Tardiness. Every job j is given a due date d_j . The tardiness of job j is defined as $T_j = max0, C_j - d_j$.
- E_j: Earliness. Every job is given a due date d_j . The earliness of job is defined as $E_j = max0, d_j - C_j$. This objective is important for just-in-time scheduling.

Chapter 2

Solving $1||\sum U_i$ scheduling problem

2.1 The problem formulation

A $1||\sum U_i$ scheduling problem consists of a set S of n jobs to be sequenced on a single disjunctive resource. The interval $[0, d_j]$ defines the execution window of each job j, where 0 is the release date of job j and d_j its due-date (all the jobs have the same release date). The processing time p_j of j is known and preemption is not allowed. A job sequence σ is said feasible if, for any job j, $s_j \geq r_j$ and $s_j + p_j \geq d_j$, s_j being the earliest starting time of job j in σ.

From the complexity viewpoint, let us mention that determining whether it exists a feasible sequence (i.e., all jobs meet their due date) is NP-complete and NP-hard for the single resource problem. In this project, we take an interest in finding a job sequence that minimizes the number of late jobs by using three different algorithms as follow.

2.2 Implementation of different algorithms solution to a $1||\sum U_i$ scheduling problem

2.2.1 Polynomial algorithm (Moore-Hodgson)

2.2.1.1 Definition

In 1968, J. M. Moore [1] presented an algorithm and analysis for minimizing the number of late jobs on a single machine problem. Moore stated ‘ The algorithm developed in this paper, however, consists of only two sorting operations performed on the total set of jobs, . . . Consequently, this method will be computationally feasible for very large problems and can be performed manually on many smaller problems.’

The Moore-Hodgson Algorithm applies a number of iterations. Each iteration maintains a sequence σ of a subset of the jobs. Initially, $\sigma = 1, 2, ..., n$. Each iteration either rejects

one job from the sequence σ, or terminates with the guarantee that σ has no late jobs. The algorithm finishes by outputting the concatenated schedule σ,ζ , where σ is the sequence from the last iteration (that has no late jobs), and ζ is an arbitrary permutation of all the rejected (i.e., late) jobs. At the start of each iteration, σ is an EDD (Earliest Due Date) sequence of the non-rejected jobs. An iteration of the algorithm examines the sequence of jobs $\sigma_1, \sigma_2, \ldots, \sigma_l$ (where $l \leq n$), and finds the smallest index k such that the job σ_k is late (thus, $C_{\sigma_k} \succ d_{\sigma_k}$ and $C_{\sigma_j} \leq d_{\sigma_j}$, $\forall j < k$). The iteration terminates if there are no late jobs; otherwise, it examines the "prefix" subsequence $\sigma_1, \ldots, \sigma_k$, picks an index m such that p_{σ_m} is maximum among $p_{\sigma_1}, \ldots, p_{\sigma_k}$, and rejects the job p_{σ_m}.

2.2.1.2 Complexity and Pseudo-code

The time complexity of the Moore-Hodgson Algorithm is $O(n \log n)$ because the hardest part is to order jobs by increasing values of due dates.
The algorithm pseudo code is as follow:

Algorithm 1 Moore-Hodgson

Input: n jobs with processing times p_i and due dates d_i.
Output: a list S of non-tardy jobs and a list A of tardy jobs..

EDD: Sort jobs in order of increasing due date: d_j;
$S_0 = \{\emptyset\}$ ▷ non-tardy jobs
$A = \{\emptyset\}$ ▷ Tardy jobs
$C = 0$ ▷ completion time of all jobs in S
for $j = 1$ to n **do**
    if $C + p_j \leq d_j$ **then** ▷ if job j is late
        $S_j = S_{j-1} \cup \{j\}$
        $C = C + p_j$
    else
        $j_{max} \in S_{j-1} \cup \{j\}$
        $S_j = S_j \cup \{j\} \setminus \{j_{max}\}$
        $A = A \cup \{j_{max}\}$
        $C = C + P_j - p_{j_{max}}$
return $\{S, A\}$

2.2.1.3 Example

The following example illustrates the working of the algorithm;

j	1	2	3	4	5	6
p_j	3	5	2	7	10	5
d_j	15	8	10	12	4	6

Table 2.1: Jobs squence

- **Step 1:** Sort jobs in order of increasing due date: d_j;

j	5	6	2	3	4	1
p_j	10	6	5	2	7	3
d_j	4	6	8	10	12	15

Table 2.2: EDD sequence

- **Initialisation:** start iterations from the EDD sequence, $S_0 = \{\emptyset\}$, $A = \{\emptyset\}$ and $C = 0$
- *the* 1^{th} *iteration:* $j = J_5$; the selected job is job 5
  $S = \{J_5\}$, J_5 is late because $d_{J_5} < C + p_{J_5}$ $(5 < 10)$. so reject J_5 from S and set $S_0 = \{\emptyset\}$, $A = \{J_5\}$ and $C = 4 - 4 = 0$
- *the* 2^{nd} *iteration:* $j = J_6$; the selected job is job 6
  $S = \{J_6\}$, J_6 is not late because $d_{J_6} > C + p_{J_5}$
- *the* 3^{th} *iteration:* $j = J_2$; the selected job is job 2
  $S = \{J_6, J_2\}$, J_2 is late because $d_{J_2} < C + p_{J_2}\,(8 < 6 + 5)$.so reject amoung S the a job whith heigh time processing, it's the job 6, then $S = \{J_2\}$, $C = 11 - 6 = 5$ and $A = \{J_5, J_6\}$
- *the* 4^{th} *iteration:* $j = J_3$; the selected job is 3
  $S = \{J_2, J_3\}$, J_3 is not late because $d_{J_3} > C + p_{J_3}\,(10 > 5 + 2)$.
- *the* 5^{th} *iteration:* $j = J_4$; the selected job is 4
  $S = \{J_2, J_3, J_4\}$, J_4 is late because $d_{J_4} < C + p_{J_4}\,(12 < 7 + 7)$.so reject amoung S the a job whith heigh time processing, it's the job 4, then $S = \{J_2, J_3\}$, $C = 14 - 7 = 7$ and $A = \{J_5, J_6, J_4\}$
- *the last iteration:* $j = J_1$; the selected job is 1
  $S = \{J_2, J_3, J_1\}$, J_1 is not late because $d_{J_1} > C + p_{J_1}\,(15 > 7 + 3)$ .so no change about A and S.
- The algorithm stop whith two sub sequences ; $S = \{J_2, J_3, J_1\}$ which contain no late jobs, $A = \{J_5, J_6, J_4\}$. so the optimal solution is $S \cup A = \{J_2, J_3, J_1, J_5, J_6, J_4$ whith 3 late jobs.

2.2.1.4 Implementation in Python

```
1
2
3   import time
4   from operator import itemgetter
5
```

```
6
7     # function to check task delay , it takes two params a squence and
      currentTime wich represente total of procesing time for previous tasks
8     # exemple [task_name,processing_time,due_date]
9     def checkDelay(lst, currentTime):
10        return (currentTime) > lst[2]
11
12
13    #read file and get data as list of lines
14    f = open("P1_n10.txt", "r")
15    lines = f.readlines()
16
17
18    # reprsent data and format it to a valid structure
19    # example :
20    # process = [[1, 24, 57], [2, 1, 144], [3, 47, 46],[4, 48, 214]]
21
22    numberOfelements = int(lines[0])
23    firstLine = str(lines[1]).split('\t')
24    secondLine = str(lines[2]).split('\t')
25    results = {}
26    test={}
27    representation = []
28    for i in range(0,numberOfelements,1):
29      representation.append([i, int(firstLine[i]),int(secondLine[i])])
30
31
32    # delete an used Data
33    del firstLine
34    del secondLine
35
36
37    # the main function of the polynomial Algorithme
38    def polynomialAlgo(lst):
39      currentTime = 0
40      # liste where we store removed process
41      removedProcesss = []
42      # liste where we store Solution's process
43      solution = [None]*len(lst)
44
45      # here we start the first step of sorting the tasks based on the
        due_date
46      sortedNodes = sorted(lst,key=itemgetter(2))
47      #step:2 start looping over all the sequences
48
49      for i in range(0, len(sortedNodes)):
50      # in the begining we can add the task what ever it is
```

```
51          solution[i] = sortedNodes[i]
52          currentTime += sortedNodes[i][1]
53
54          # check if the task that we add to solution's sequence is before due
    Date or not
55          if not checkDelay(sortedNodes[i],currentTime):
56            pass
57          else:
58            # if task is late then ,we have to search for the task with max
    processing value to be removed ,and stored in removedProcesss table
59            max = 0
60            maxId = 0
61            for j in range(0, i+1, 1):
62              if solution[j] != None:
63                if solution[j][1] > max:
64                  max = solution[j][1]
65                  maxId = j
66            removedProcesss.append(solution[maxId])
67            currentTime -= solution[maxId][1]
68            solution[maxId] = None
69
70       # remove the nulled values during the algo
71       solution = list(filter(lambda a: a != None, solution))
72       #return object of optimal tasks sequence and number of tardy tasks
73       return {"solution":solution+removedProcesss,"lateTasks":len(
    removedProcesss)}
74
75
76     # call the algo function to be applied on data that we read from files
77
78     print(polynomialAlgo(representation))
79
80
```

Listing 2.1: Moore-Hodgson algorithm python code for $1||\sum U_i$ problem

2.2.2 Dynamic programming

2.2.2.1 Definition

Held and Karp (1962) are probably the first researchers to apply DP technique to sequencing problems. Their idea is based on a simple observation that if the objective function of a sequencing problem can be decomposed into a series of objective functions corresponding to a series of smaller problems,then solve these sub-problems until finding the solution of the original problem. DP technique determines the optimal solution of the sub-problem along with its contribution to the objective function and tries to improve the solution by a number of iterations. Boundary conditions, a recursive relation and an optimal value function characterize it.In scheduling, the

forward dynamic programming and backward dynamic programming approaches can be used.

2.2.2.2 Complexity and Pseudo-code

Though the effort required to solve the problem grows exponentially with the problem size, DP is more efficient than complete enumeration as computational demands for the latter increases as a factorial of the problem size. It is often described as implicit enumeration as it considers certain sequences only indirectly, without actually evaluating them. So the computational time required to obtain an exact result is $O(n2^n)$, since all 2^n subsets of N jobs need to be generated as states of this dynamic programming.

- ❑ **Defining terms:**
  Let $J \subseteq N = \{1, 2.....n\}$ be an arbitrary subset of jobs, and let f(J) be the minimum of latey jobs number function over J when the jobs in J are sequenced in the first $S = |J|$ positions of the whole sequence.So the objective function is:

$$f(S,J) = \min_{j \in J} \left\{ f(S-1, J \setminus \{j\}) + \sum_{k \in J} U_k \right\} \tag{2.1}$$

  with:

  - The initial condition is $f(0, \varnothing) = 0$;
  - $$U_k = \begin{cases} 1 & \text{if } d_k < c_k \\ 0 & \text{else} \end{cases}$$
  - $\sum_{k \in J} U_k$ is the nombre of latey jobs in J

- ❑ **Pseudo-code:**

2.2.2.3 Example

A long speech does not explain clearly like an example, so this example show how this algorithm works:

j	1	2	3
p_j	3	5	2
d_j	15	6	7

Table 2.3: Jobs squence

The steps of the algorithm are illustrated in the table below:

Algorithm 2 Dynamique programming for $1||\sum U_i$

Input: N jobs with processing times p_i and due dates d_i.
Output: an exact solution .

1: $f(0, \emptyset) = 0$ ▷ initial solution
2: $B(0, \emptyset) = \emptyset$
3: **for** $i = 1$ to n **do**
4: $M \longleftarrow$ **Generate all subsequences from N with the lenght** $|J| = i$ **in the increasing order.**
5: **for** J in M **do** ▷ iterate in M
6: $f(S,J) = \min_{j \in J} \left\{ f(S-1, J \setminus \{j\}) + \sum_{k \in J} U_k \right\}$ ▷ save the result in a table to use it in the following iterations
7: $B(S,J) \longleftarrow$ the sequence J except a job j which f(S,J) is minimum
8: go backward from f(n,N) to f(0,ø) recursively to construct the optimal job sequence.

J	j Last job	$\sum_{k \in J} U_k$	f(S,J)	B(S,J)
$\{1\}$	1	0	$f(0,\emptyset)+0=0*$	$\{\emptyset\}$
$\{2\}$	2	0	$f(0,\emptyset)+0=0*$	$\{\emptyset\}$
$\{3\}$	3	0	$f(0,\emptyset)+0=0*$	$\{\emptyset\}$
$\{1,2\}$	1	0	$f(1,\{2\})+0=0*$	$\{2\}$
	2	1	$f(1,\{1\})+1=1$	
$\{1,3\}$	1	0	$f(1,\{3\})+0=0$	
	3	0	$f(1,\{1\})+0=0*$	$\{1\}$
$\{2,3\}$	2	1	$f(1,\{3\})+1=1$	
	3	0	$f(1,\{2\})+0=0*$	$\{2\}$
$\{1,2,3\}$	1	0	$f(2,\{2,3\})+0=0*$	$\{2,3\}$
	2	1	$f(2,\{1,3\})+1=1$	
	3	1	$f(2,\{1,2\})+1=1$	

Table 2.4: Dynamique programming example for $\sum_{k \in J} U_k$

The algorithm start by generating all possible subsequences in the increasing order.
in the last row of the table we can see that the minimum number of latey jobs is zero (we can schedule all the jobs without any late ones). to get optimal sequence we start by $B(3,\{1,2,3\}) = \{2,3\}$ so the job 1 will be the last with the index 3. Then $B(2,\{2,3\}) = \{2\}$ so the job 3 will be the seconde in the optimal schedule, thus 2 will be the first.
Finaly the optimal schedule is $\{2,3,1\}$

2.2.2.4 Implementation in python

```
1  import itertools
2
3
```

```
# function to check task delay , it takes two params a squence and currentTime wich represente total of procesing time for previous tasks
# and return 0 if it's done before due Date and 1 for else

def checkDelay(lst, currentTime):
  if (lst[1] + currentTime) > lst[2]:
    return 1
  else:
    return 0


#read file and get data as list of lines
f = open("P1_n10.txt", "r")
lines = f.readlines()



# reprsent data and format it to a valid structure
# example :
# process = [[1, 24, 57], [2, 1, 144], [3, 47, 46],[4, 48, 214]]

nbOfEelements = int(lines[0])
processingTime = str(lines[1]).split('\t')
dueDate = str(lines[2]).split('\t')

data = []

for i in range(0,nbOfEelements):
  data.append([i,int(processingTime[i]),int(dueDate[i])])


# the begining of the dynamic programing algo
def dynamicAlgo(lst,results,sequence):
  # define a nunmber that will considered as max , like INT_MAX
  min = 177272

  # represente the key of the dictionary object
  # exemple we get [[1,1,4],[2,4,5]]
  # the key will be  the index of tasks splited with ',' ==> "1,2"
  identifier = ",".join("'%s'" % a[0] for a in lst)
  # check if list is not null if it is then return 0
  if len(lst) != 0:
    # looping throw list of  tasks
    for element in range(0, len(lst), 1):
      currentTime = 0
      # put all the given tasks list to modifiedList variable except the one that we are looping now
```

```
49        modifiedList = lst[:element]+lst[element+1:]
50        # claclulate currentTime which is the sum of processing time for all
    the other sequecnes  except the one that we are looping now
51        for index in range(0, len(modifiedList)):
52          currentTime += list(modifiedList)[index][1]
53        # represente the key of the dictionary object of the other sequence
54        identifierTemp = ",".join("'%s'" % a[0] for a in list(modifiedList))
55        # check if we already have this sequence min value in list of results
56        # else calculate it and memoizat it
57        if identifierTemp in results.keys():
58          res = checkDelay(lst[element], currentTime) + results[
    identifierTemp]
59        else:
60          res = checkDelay(lst[element], currentTime) + 0
61        # check if the current sequence order is the less than min then it
    will be the new min
62        if res < min:
63          min = res
64          if identifierTemp in sequence.keys():
65            # print(sequence[identifierTemp])
66            sequence[identifier] = list(sequence[identifierTemp])+[lst[
    element]]
67          else:
68            # print(modifiedList+lst[element])
69            sequence[identifier] = list(modifiedList)+[list(lst[element])]
70      return min
71     else:
72      return 0
73
74
75  # simple function to wrape the main dynamic algo and pass the possible
     combinations to it
76  def dynamicAlgoWrapper(data):
77     identifier = ""
78     results ={}
79     sequence = {}
80     for i in range(1, len(data) + 1, 1):
81       sample = list(itertools.combinations(data, i))
82       # overwite the old value and keep only the ones we need in the next
     calculations
83       results = {k: v for k, v in results.items() if len(k.split(',')) ==i-1}
84       sequence  = {k: v for k, v in sequence.items() if len(k.split(',')) ==i
    -1}
85       # pass combinations to dynamic algo to count it
86       for iteam in sample:
87         identifier = ",".join("'%s'" % a[0] for a in iteam)
88         results[identifier] = dynamicAlgo(list(iteam),results,sequence)
```

```
89      # return the min sequence which is clearlly the last one that will
    contain all the indexs splited by ','
90     return {"min":results[identifier],"sequence":sequence[identifier]}
91
92
93
94  # call the function to start processing data
95  print(dynamicAlgoWrapper(data))
96
97
98
99
```

Listing 2.2: Dynamic Programming python code for $1||\sum U_i$ problem

2.2.3 GVNS algorithm

2.2.3.1 Definition

General Variable Neighbourhood Search (GVNS) is a metaheuristic, or framework for building heuristics, aimed at solving combinatorial and global optimization problems. Its basic idea consists in a systematic change of neighbourhood combined with a local search. Since its inception, VNS has undergone many developments and been applied in numerous fields. We review below the basic rules of VNS and of its main extensions.
A deterministic optimization problem may be formulated as:

$$\min \{f(x)|x \in X, X \subseteq S\} \tag{2.2}$$

where S, X, x and f respectively denote the solution space and feasible set, a feasible solution and a real-valued objective function.

VNS is a metaheuristic which systematically exploits the idea of neighbourhood change, both in descent to local minima and in escape from the valleys which contain them. VNS heavily relies upon the following observations:

- **Fact 1** A local minimum with respect to one neighbourhood structure is not necessarily a local minimum for another neighbourhood structure.
- **Fact 2** A global minimum is a local minimum with respect to all possible neighbourhood structures.
- **Fact 3** For many problems local minima with respect to one or several neighbourhoods are relatively close to each other.

2.2.3.2 Neighborhood structures

- **Swap operator**

  swap the position of two elements in sequence

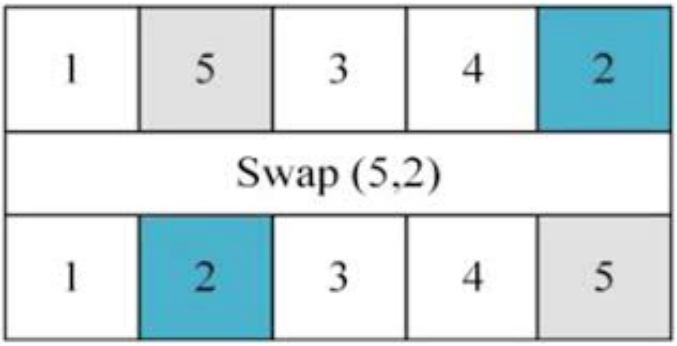


Figure 2.1: Swap two elements

Algorithm 3 Swap(X, i, j)

Input: A current sequence and a pair of jobs to swap.
Output: a new sequence

1: $tmp = 0$
2: $tmp = X[i]$
3: $X[i] = X[j]$
4: $X[j] = tmp$
5: **return** X

```
1       def swap(li):
2         I=0
3         J = 0
4
5         while I==J :
6           I = random.randint(0, len(li) -1)
7           J = random.randint(0, len(li) - 1)
8         i=min(I,J)
9         j=max(I,J)
10
11        t=li.copy()
12        tmp=t[i]
13        t[i]=t[j]
14        t[j]=tmp
15        return t
16
```

Listing 2.3: Swap oprator

- **Insertion operator**

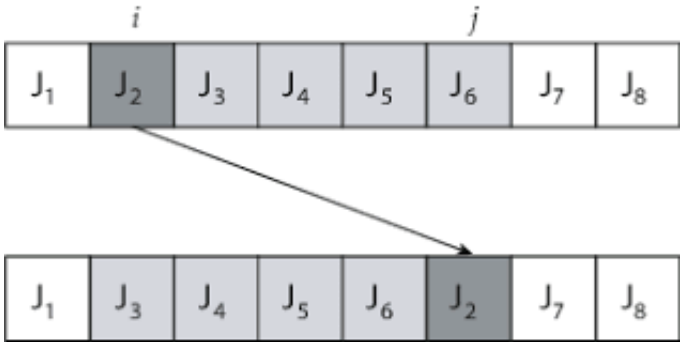


Figure 2.2: Insertion

Algorithm 4 insertion(X, v, p)

Input: A current sequence,a job j and a position p to insert in .
Output: a new sequence

1: Remove the element with the index v .
2: Shift the right sub sequence one index back .
3: Insert the removed element into X in the position p.
4: **return** X

```
1       def insertion(li):
2         I=0
3         J = 0
4
5         while I==J :
6           I = random.randint(0, len(li) - 1)
7           J = random.randint(0, len(li) - 1)
8
9         #the jobs selected to insert it in the position j
10        i=min(I,J)
11
12        #insertion position
13        j=max(I,J)
14
15        l=li.copy()
16
17        b=l.pop(i-1)
18        l.insert(j,b)
19        return l
20
```

Listing 2.4: Insertion operator

- **2-opt operator**

A **2-Opt** move deletes two edges, thus breaking the tour into two parts, and then reconnects

those paths in the other possible way (see Figure 2.1). This is equivalent to reversing the order of the cities between the two edges, thus a 2-Opt move can be seen as a segment reversal.

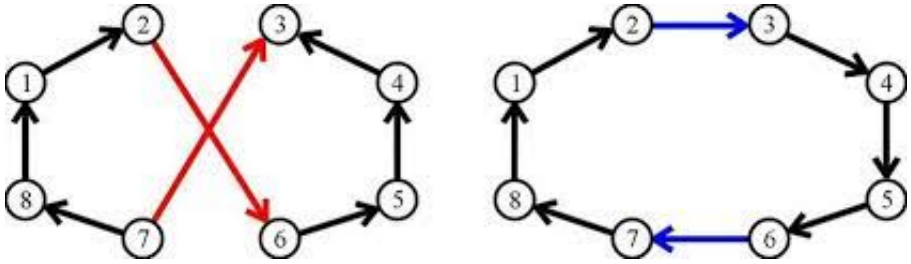


Figure 2.3: 2-opt ilustrator

Algorithm 5 twoOpt(X, i, j)

Input: A current sequence and two indexes .
Output: a new sequence

1: $L = 0$
2: $i = \min(i, j)$
3: $j = \max(i, j)$
4: add the sub sequence from the indexes 0 to i-1 to L.
5: reverse the sub sequence from the indexes i to j and then add it to L.
6: add the rest sub sequence from the indexes j+1 to the end.
return L

```
    def twopt(li):
      I=0
      J = 0
      m=[]
      #while loop to avoid selecting the same edge
      while I==J :
        I = random.randint(0, len(li) - 1)
        J = random.randint(0, len(li) - 1)

      #the edge (i,i+1)
      i=min(I,J)

      #the edge (j,j+1)
      j=max(I,J)

      m = m + li[:i]
      n = li[i:j].copy()
      n.reverse()
      m = m + n
      m = m + li[j:]


      return m

```

Listing 2.5: 2-Opt operator

2.2.3.3 Shaking

the shake function simply generates a random neighbor from a given neighborhood structure $N_k(X)$, the related pseudo-code can be expressed as follow:

Algorithm 6 shake(X, N_k)

Input: A current sequence X,a neighborhood structure N_k .
Output: a random neighbor X'.

1: Choose $X' \in N_k(x)$ at random;
2: **return** X'

2.2.3.4 Local search

A local search heuristic consists in choosing an initial solution x, finding a direction of descent from x, within a neighbourhood N(x), and moving to the minimum of f(x) within N(x) in the same direction. If there is no direction of descent, the heuristic stops; otherwise, it is iterated. Usually the steepest direction of descent, also referred to as best improvement, is used. This set of rules is summarized in Algorithm 7 and 8, where we assume that an initial solution x is

given. The output consists of a local minimum, also denoted by x.
these are some examples of local searche:

- **Best-Improvement** It considers every single possible neighbor of our current solution X and tries to get the best out of the neighborhood, this technique is useful in smaller data-sets but it's a time consuming and an ineffective technique, its implementation is given below.

Algorithm 7 Best-Improvement(X, N_k)

Input: current solution (X) and neighborhood structure.
Output: New local optima X'.

1: **repeat**
2: $X' = X$
3: **for** $X'' \in N_k(X')$ **do**
4: **if** $f(X'') < f(X')$ **then**
5: $X' = X''$
6: **until** $f(X') < f(X)$
7: **return** X'

- **First-Improvement** First improvement is based on two other facts, firstly, local minima isn't that good, secondly, it takes too much time to get to that not good local minima.

Algorithm 8 First-Improvement(X, N_k)

Input: current solution (X) and neighborhood structure.
Output: New local optima X'.

1: **repeat**
2: (1) Find first solution $x' \in N_k(x)$
3: (2) If $f(x') > f(x)$, find the next solution $x'' \in N_k(x)$; set x'=x" and iterate
4: (2) otherwise, set x=x' and iterate(1);
5: (3) If all solutions of N(x) have been comsidered, stop.

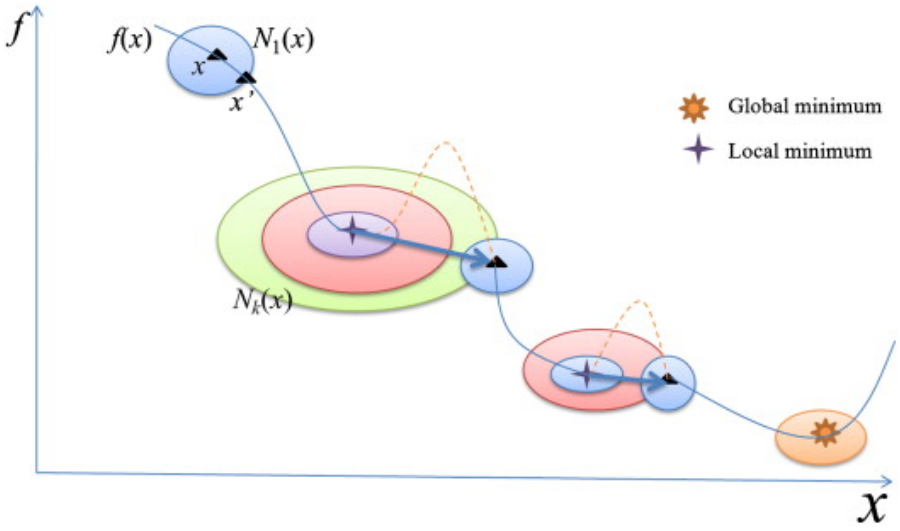


Figure 2.4: Local and global optimums

2.2.3.5 VND

The variable neighbourhood descent (VND) method is obtained if the change of neighbourhoods is performed in a deterministic way. Its steps are presented in Algorithm below. More precisely a VND procedure explores several neighborhood structures either in a sequential or nested fashion in order to possible improve a given solution. As a search strategy it may use either the first improvement or the best improvement search strategy.

Algorithm 9 VND

Input: current solution (X) and a k'_{max}
Output: New local minimum X'.

repeat
  $k = 1$
  repeat
    $x' = \arg\min_{y \in N'(x)_k} f(x)$
    if $f(x'') < f(x)$ **then**
      $x = x''$
      $k = 1$
    else
      $k = k + 1$
  until $k - k'_{max}$
until *no improvement is obtained*

```
1       N=[twopt,insertion,twopt]  #list of neighborhoosd structures
2
3
4       def VND(li,N):
```

```
        x0=li.copy()

        #time limit condition
        timelimit=1
        start=time.time()
        stop1=True
        while stop1:
          l=0
          #iterate in neighborhood structures
          while l<len(N):


            #testing if time ecouling
            if (time.time()-start)>timelimit :
                stop1=False
              break
            x1=N[l](x0)  #shaking
            if f(x1)<f(x0):  #move or not
              x0=x1.copy()
              l=0
            else:
                l+=1


        return x0

```

Listing 2.6: VND

2.2.3.6 VNS

The VNS algorithm is designed to enrich the search space by restarting a local search heuristic with randomly generated neighborhood solutions from an incumbent solution by a predetermined set of neighborhood structures. Systematic changes of the neighborhood solution within the local search is a key concept of VNS algorithms to improve a solution quality. Thus, the performance of VNS algorithm is mainly influenced by the local search heuristic and the neighborhood structure to meet problem characteristics.

Algorithm 10 VNS

Input: current solution (X), number of neighborhood structures k_{max} and time limit t_{max}.
Output: new jobsequence.

repeat
  $k = 1$;
  repeat
    $x' = Shake(x, k)$
    x"=VND(x')
    if $f(x'') < f(x)$ **then**
      $x = x''$
      $k = 1$
    else
      $k = k + 1$
  until $k = k_{max}$;
  t=CpuTime()
until $t > t_{max}$

```
N=[twopt,insertion,twopt]

def VNS(li,N,temps):
  vx=[]
  vx=li.copy()

  timelimit2=temps
  start2=time.time()
  x2=[]
  stop2=True
  while stop2:
    k=0
    while k<len(N):

      if (time.time()-start2)>timelimit2 :
        stop2=False
        break
       #shaking
      x2=N[k](li)

      #VND
      vndx=VND(x2,N)

       #move or not
      if f(vndx)<f(vx):
        vx=vndx.copy()
        k=0
      else:
```

```
            k=k+1

      return vx

    #call the VNS function
    v=VNS(process,N,1)
    print(f(v))

```

Listing 2.7: VNS

Chapter 3

Solving $1||\sum \alpha_i E_i + \beta_i T_i$ scheduling problem

3.1 The problem formulation

A set of jobs $J = 1, 2, ..., n$ has to be scheduled in a single machine.Each job $i \in J$ has a processing time denoted by p_i, which is assumed to be integer. A schedule is a sequence of n starting times S_1,S_2,...,S_n, subject to the constraints $S_i \geq S_j + p_j$ or $S_j \geq S_i + p_i$ (for all $i \ni j$). these constraints are called the disjunctive constraints. they mean that two jobs can not be processeddd at the same time by the machine . then a job i is secheduled -without preemption- from S_i to its completion time , denoted by C_i, equal to $S_i + p_i$.

A cost function f_i is attached to each job $i \in J$, the cost of job i in a schedule is $F_i(C_i)$. The total cost of the schedule is simply the sum of the costs of all jobs, that is $\sum_{i \in J} F_i(C_i)$. The problem is to find the minimum cost. The objective function or the cost function is equal to:

$$F_i(C_i) = \alpha_i E_i + \beta_i T_i = \alpha_i \max(0, d_i - C_i) + \beta_i \max(C_i - d_i, 0) \qquad (3.1)$$

d_i represents the due date of job i:when i completes at time d_i, its cost is null and i is *on-time*. If $C_i < d_i$, job i early and its cost is $\alpha_i(d_i - C_i)$. α_i is the *earliness penalty of job i per unit time*. Symmetrically, if $C_i > d_i$, job i is late, its cost is $\beta_i(C_i - d_i)$, β_i being its *tardiness penalty per unit time.*

3.2 Implementation of different algorithms solution to a $1||\sum \alpha_i E_i + \beta_i T_i$ scheduling problem

3.2.1 Dynamic programming

3.2.1.1 Definition

The dynamic programming algorithm described for the problem $1||\sum U_i$ can be adapted to solve the problem $1||\sum \alpha_i E_i + \beta_i T_i$ in a straightforward manner. Recall that the state-variable $U_i\,(i = 1, \ldots, n)$ was defined as the minimum total number of latey jobs over all feasible subsets of N consisting of exactly n jobs. When we change this such that $f\,(S, J)$ is equal to the minimum total cost (earliness and tardiness) over all feasible subsets of N. Then we can use it to solve the problem $1||\sum \alpha_i E_i + \beta_i T_i$, since the whole analysis of the problem 1 goes through the same situation. We only need to change the recurrence function to :

$$F_i(C_i) = \alpha_i E_i + \beta_i T_i = \alpha_i \max(0, d_i - C_i) + \beta_i \max(C_i - d_i, 0) \qquad (3.2)$$

3.2.1.2 Implementation in Python

```
1
2       import itertools
3
4
5       # function to check task delay , it takes two params a squence and
    currentTime wich represente total of procesing time for previous tasks
6       # and return alpha*max(0,dueDate - (currentTime + processingTime)) +
    beta*max(0,(currentTime + processingTime)-dueDate)
7
8       def checkDelay(lst, currentTime):
9         e = max(0,lst[2]-(currentTime+lst[1]))
10        t = max(0,(currentTime+lst[1]) - lst[2])
11        return lst[3]*e+lst[4]*t
12
13
14      #read file and get data as list of lines
15      f = open("P2_n10.txt", "r")
16      lines = f.readlines()
17
18
```

```
19
20      # reprsent data and format it to a valid structure
21      # example :
22      # process = [[1, 24, 57,2,3], [2, 1, 144,4,5], [3, 47, 46,1,0],[4, 48,
    214,0,4]]
23
24      nbOfEelements = int(lines[0])
25      processingTime = str(lines[1]).split('\t')
26      dueDate = str(lines[2]).split('\t')
27      alpha = str(lines[3]).split('\t')
28      beta = str(lines[4]).split('\t')
29      data = []
30
31      for i in range(0,nbOfEelements):
32        data.append([i,int(processingTime[i]),int(dueDate[i]),int(alpha[i]),
    int(beta[i])])
33
34
35      # the begining of the dynamic programing algo
36      def dynamicAlgo(lst,results,sequence):
37        # define a nunmber that will considered as max , like INT_MAX
38        min = 177272
39        # represente the key of the dictionary object
40        # exemple we get [[1,1,4],[2,4,5]]
41        # the key will be  the index of tasks splited with ',' ==> "1,2"
42        identifier = ",".join("'%s'" % a[0] for a in lst)
43        # check if list is not null if it is then return 0
44        if len(lst) != 0:
45          # looping throw list of  tasks
46          for element in range(0, len(lst), 1):
47            currentTime = 0
48            # put all the given tasks list to modifiedList variable except
    the one that we are looping now
49            modifiedList = lst[:element]+lst[element+1:]
50            # claclulate currentTime which is the sum of processing time for
    all the other sequecnes  except the one that we are looping now
51            for index in range(0, len(modifiedList)):
52              currentTime += list(modifiedList)[index][1]
53            # represente the key of the dictionary object of the other
    sequence
54            identifierTemp = ",".join("'%s'" % a[0] for a in list(
    modifiedList))
55            # check if we already have this sequence min value in list of
    results
56            # else calculate it and memoizat it
57            if identifierTemp in results.keys():
```

```
58              res = checkDelay(lst[element], currentTime) + results[
    identifierTemp]
59            else:
60              res = checkDelay(lst[element], currentTime) + 0
61            # check if the current sequence order is the less than min then
    it will be the new min
62            if res < min:
63              min = res
64              if identifierTemp in sequence.keys():
65                sequence[identifier] = list(sequence[identifierTemp])+[lst[
    element]]
66              else:
67                # print(modifiedList+lst[element])
68                sequence[identifier] = list(modifiedList)+[list(lst[element])
    ]
69            return min
70          else:
71            return 0
72
73
74      # simple function to wrape the main dynamic algo and pass the possible
    combinations to it
75      def dynamicAlgoWrapper(data):
76        identifier = ""
77        results ={}
78        sequence = {}
79        for i in range(1, len(data) + 1, 1):
80          sample = list(itertools.combinations(data, i))
81          # overwite the old value and keep only the ones we need in the next
     calculations
82          results = {k: v for k, v in results.items() if len(k.split(',')) ==
    i-1}
83          sequence  = {k: v for k, v in sequence.items() if len(k.split(','))
     ==i-1}
84          # pass combinations to dynamic algo to count it
85          for iteam in sample:
86            identifier = ",".join("'%s'" % a[0] for a in iteam)
87            results[identifier] = dynamicAlgo(list(iteam),results,sequence)
88        # return the min sequence which is clearlly the last one that will
    contain all the indexs splited by ','
89        return {"min":results[identifier],"sequence":sequence[identifier]}
90
91
92
93      # call the function to start processing data
94      print(dynamicAlgoWrapper(data))
95
```

```
96
```

Listing 3.1: Dynamic Programming python code for $1||\sum \alpha_i E_i + \beta_i T_i$ problem

3.2.2 GVNS

3.2.2.1 Definition

Owing to the fact that the criterion of this kind of scheduling problems is not depended to algorithms, thus in this kind of single machine scheduling we need just to change the objective function.

3.2.2.2 Implementation in Python

```
1   def VNS(li,N,temps):
2     vx=[]
3     vx=li.copy()
4     timelimit2=temps
5     start2=time.time()
6     x2=[]
7     #while loop for time limit condition
8     while (time.time()-start2)<timelimit2:
9       k=0
10      n=0
11      while k<len(N):
12
13        #shaking
14        x2=N[k](li)
15
16        vndx=VND(x2,N)
17        if f(vndx)<f(vx):
18          vx=vndx.copy()
19          k=0
20        else:
21          k=k+1
22
23    return vx
24
25  v=VNS(process,N,1)
26  print(f(v))
27
28
```

Listing 3.2: GVNS python code for $1||\sum U_i$ problem

Chapter 4 Benchmarking experiments

As declared before, most scheduling problems stay NP-hard and NP-complet, whatever machine enviroment, optimal criterion and constraints were. Each algorithm yields a solution whith some accuracy. The best one is which have heigh accuracy, less consumming the memory and less CPU time requiring to get the best result. So the easiest way to compare between them is some **benchmarking tests**.

4.1 Performance analysis

The curves below illustrate the performance of the three algorithms studied above:

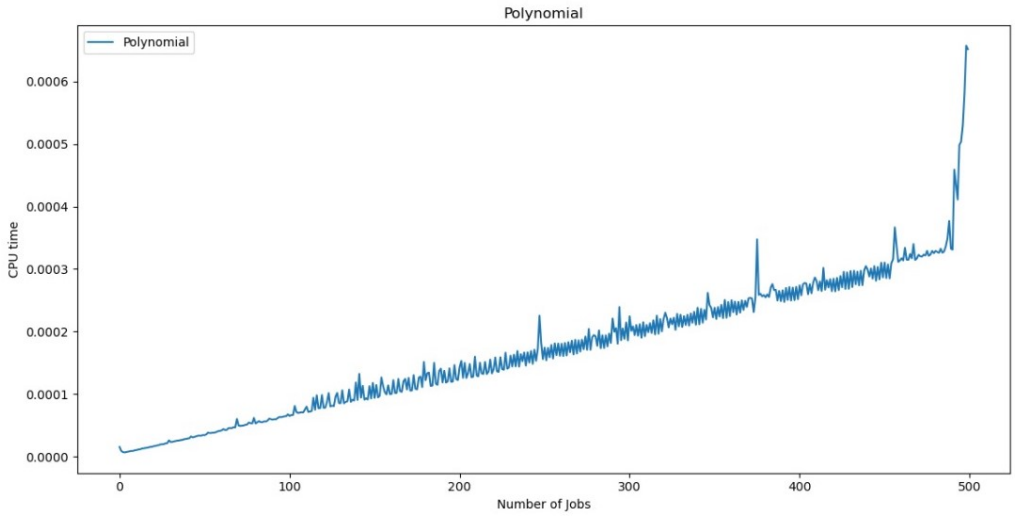


Figure 4.1: Moore-Hodgson performance

Based on the illustrations in these figures and a detailed comparison of the results for Moore-Hodgson algorithm, DP algorithm and GVNS anglorithm indicate that, on average:

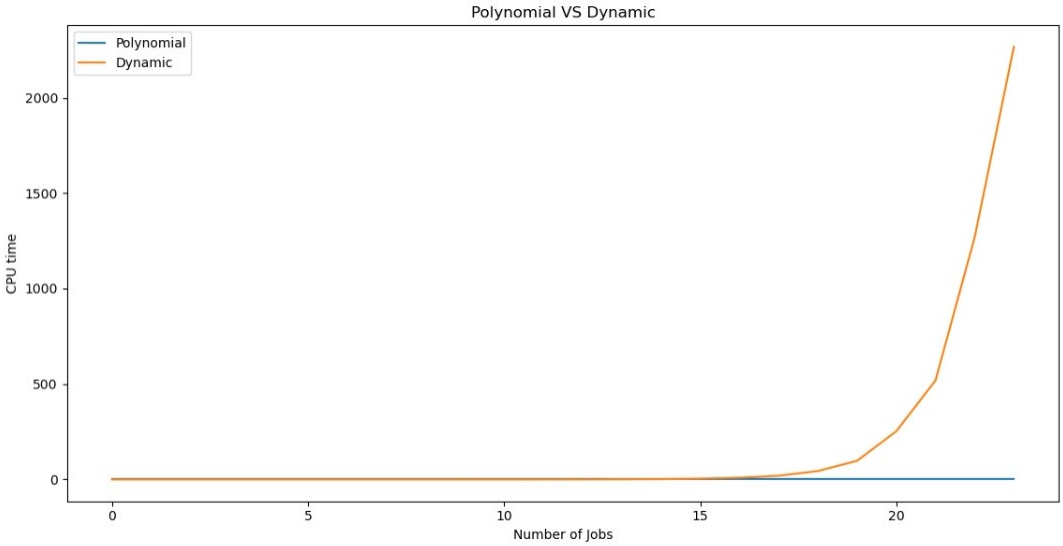


Figure 4.2: Moore-Hodgson and Dynamic programming performance

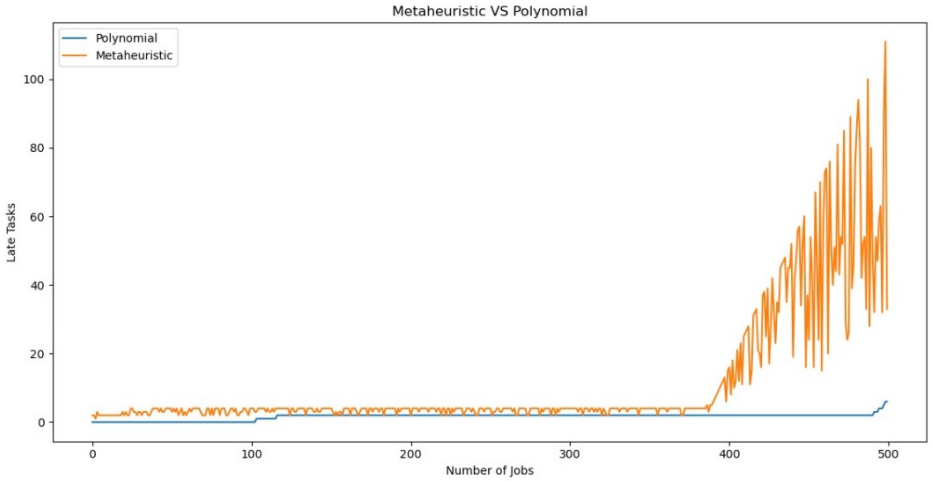


Figure 4.3: GVNS and Moore-Hodgson

- DP give an exact solution with excessive computation time and memory. The average number of jobs that can solve with our cumputer is approximately up to 25 jobs but from 20 to 25 jobs the CPU time increase exponentially, the same as all other algorithms.
- Moore-Hodgson algorithm give the result immediately for a small number of jobs even for big numbers.
- In the last figure we can see that the GVNS algotithm also yiels a very approximate or an exact optimal value for small numbers of jobs, however its accuracy decreases exponentially, proportionally to the number of jobs .

4.2 Performance explanation

An explanation of this difference in performance is that for DP algorithm, it considers all possible cases and stock the result of each subsequence schedule to use it directly later, thus the memory become full and CPU will be consumed for long time.
For GVNS algorithm exaclty in the shaking stage ,the sequence generated randomly may not be improvable for VND if we use first improvement or best improvement local searches.

Chapter 5 Conclusion

This project considers the problems of scheduling jobs on a single machine to minimize the number of latey jobs in the first problem, and total earliness and tardiness penalties in the seconde problem. We propose three different algorithms with their related informations (complexities,pseudo codes and implementations). We establish that these problems are NP-hard for the general case.
We present the algorithms' formulations of the problems and we apply the principle of their optimality to find the optimal solution by implicit enumeration.

In order to evaluate these effectiveness and efficiency of these algorithms, extensive computational experiments are carried out. The experimental results show that:

- DP algorithm is a far more efficient optimization method than complete enumeration for solving the two problems, but it stills limited for a small number of jobs.
- Moore-Hodgson algorithm is more promising: it yields very close-to-optimal solutions in short computational times and it's applicable for the first problem only.
- For the metaheuristic GVNS we found that the results are close-to-optimal in a few jobs only and less efficient inversely with the sequence size.

Bibliography

[1] J. Michael Moore, An n job machine sequencing algorithm for minimizing the number of late jobs.

[2] Hanane Krim1, Rachid Benmansour, David Duvivier and Abdelhakim Artiba A variable neighborhood search algorithm for solving the single machine scheduling problem with periodic maintenance.

[3] Mladenovic N, Hansen P (1997), Variable neighborhood search.

[4] Basar Ogun, Çigdem Alabas-Uslu (2018), Mathematical Models for a Batch Scheduling Problem to Minimize Earliness and Tardiness.

[5] Susanne Albers and Peter Brucker (1991) The complexity of one-machine batching problems.