May 20, 2020

Moscow Institute of Physics and Technology
Phystech-school of applied mathematics and informatics

Single machine preemptive scheduling

In search of an exact solution

Prepared by:
Fomin Artem, group 799

Supervisor:
Goldengorin Boris Isaakovich

Abstract

We propose two novel modeling approaches to solve single machine preemptive scheduling problems with numerous objective functions. Moving away from widely used linear programming formulations, we relax the scheduling problem to the linear assignment problem, and then employ two different branch and bound algorithms to solve the problem to optimality. For the second branching strategy we also prove a few theoretical insights, which can be used outside this particular relaxation. We also propose numerous modifications to the model, which allow to take into account many particularities real-life schedules may have. A set of thorough computational experiments have been conducted to establish algorithms' performance.

1 Introduction

In scheduling problems one must appoint a number of jobs to be processed to the specific time, such that a certain objective function is minimized. Scheduling problems widely appear in lots of theoretical and practical problems such as manufacturing, management science and computer science. *Single machine* means that at each point of time only one job can be processed. *Preemptive* means that preemptions are allowed, i.e. a job can be interrupted and finished later.

Such scheduling problems are known to be NP-hard in the strong sense[6]. Due to the complexity of the problem, most research focuses on development of heuristics and bounds. A few papers focusing on exact methods are considering only non-preemptive cases. A recent 2020 study by Jaramillo, Keles and Erkoc[1] tried to fill the gap, but could only solve relatively small problems.

2 Problem formulations

We will employ discretization of the problem. The time is separated into discrete and equal intervals. A job, assigned to process at a specific interval, will be processed for the whole interval. The jobs can be started or preempted only between the intervals.

That is fairly realistic, since people don't use perfectly continuous timeframes anyway: in terms of big constructions, it may be reasonable to measure time in months; for complex productions, timeframe of days may be suitable; for everyday jobs the hours would be suitable; CPU clock cycles for computing. Also, in real life it's at least somewhat costly to interrupt current job and start another one. For both people and computers. So, it is reasonable to believe that interruptions don't happen at just any time and certainly do not happen all the time. Hence, with tight enough discretization we will be able to model necessary problems efficiently.

For the most simple and most popular formulation jobs have the following characteristics:

r_j	release date
d_j	due date
w_j	weight
p_j	processing time

All these values are positive integers. Processing time p_j stands for how many time intervals it takes to finish the job. Release date r_j denotes the time interval, at which the job becomes available. Due date d_j signifies the time period, after which we will be fined, if we haven't completed the job yet, according to the objective function. Weight w_j signifies a priority of the task, which can be used in objective function to impose greater penalties for failing to succeed on a more important job.

Following variables will be used for measuring quality of a solution:

c_j	completion time
t_j	tardiness
e_j	earliness
u_j	tardiness indicator

Completion time c_j signifies a time interval, at which the job was completed. Tardiness $t_j = max\{0, c_j - d_j\}$ stands for how many time intervals late are we behind the due date, if any. Earliness $e_j = max\{0, d_j - c_j\}$, similarly, stands for how many time intervals early are we, if any. Tardiness indicator $u_j = I\{l_j > 0\}$ tells, if we are late, with no regard to how late.

Let n denote the number of jobs and $T = \sum_{i=1}^{n} p_i$ denote a time horizon.

Then, the objectives can be formulated as:

1. Total weighted tardiness (TWT): $\sum_{j=1}^{n} w_j t_j$
2. Total weighted completion time (TWCT): $\sum_{j=1}^{n} w_j c_j$
3. Total weighted earliness and tardiness (TWET): $\sum_{j=1}^{n} w_j (e_j + t_j)$
4. Total weighted number of tardy jobs (TWNTJ): $\sum_{j=1}^{n} w_j u_j$

Our task is to find a schedule, which would assign parts of jobs to exactly one of T time intervals, such that every job is fully processed according to its release date and processing time, no two jobs are processed at the same time and the objective function is minimized.

3 Linear assignment problem relaxation

We will relax our problem to the assignment problem (AP) by dividing each job into p_i separate parts, each of which takes exactly one time interval to be executed. Then, we will assign these parts to T time intervals. It is easy to see, that every solution of the original problem is also a solution of the corresponding

AP. But not every solution of the AP is feasible for the original problem, as it may assign a job's first part before its release date or to be made after the second part, because AP doesn't take into consideration the order of the parts.

Now we will demonstrate our approach for the TWT objective function by working through a numerical example with following specifications:

	1	2	3
r_j	1	2	4
d_j	1	1	3
w_j	1	2	3
p_j	2	3	5

To start with, we need to reduce it to AP. The reduction is similar to one presented in [3], generalized for non-equal processing times. A common AP representation takes form of a matrix, where each row represents a job (one part of a job in our case) and each column represent an agent (time interval in our case). Each cell of the matrix has a value in it, which refers to how costly it is to assign a particular job to a particular agent. Here we have $T = \sum_{i=1}^{n} p_i = 2 + 3 + 5 = 10$, hence 10 time intervals (agents, columns) and also 10 parts of the jobs in total, hence the matrix will be of 10×10 size.

First, we will assign a cost of ∞ to each cell, that is impossible for assignment. The first impossibility is produced by release dates: for the first part of the job, we mark all columns prior to its release date by ∞.

	1	2	3	4	5	6	7	8	9	10
1										
2										
3	∞									
4										
5										
6	∞	∞	∞							
7										
8										
9										
10										

Next, we can also tell that second and third part of second job can't be done in the first time interval. But also, as executing the second part requires executing the first part beforehand, we can also tell that the second part of the second job (row 4) can't be done in the second time interval. Similarly, the third part can't be done in the first, second and third time intervals. We fill the table ∞ in these cells analogously for each job.

	1	2	3	4	5	6	7	8	9	10
1										
2	∞									
3	∞									
4	∞	∞								
5	∞	∞	∞							
6	∞	∞	∞							
7	∞	∞	∞	∞						
8	∞	∞	∞	∞	∞					
9	∞	∞	∞	∞	∞	∞				
10	∞	∞	∞	∞	∞	∞	∞			

Secondly, we can also tell that it's to late to start doing third job at the 10th interval, as we won't finish it in time. Actually, the third job should be started not later than the sixth period to make it in time, so every sell in the 6th row starting with the 7th column can be marked as ∞. But starting in time is not enough: to finish a job before reaching the time horizon, every part of time must be done in appropriate time. That is, the second part of third job must be done not later than 7th interval, third part — not later than 8th, ans so on.

	1	2	3	4	5	6	7	8	9	10
1										∞
2	∞									
3	∞								∞	∞
4	∞	∞								∞
5	∞	∞	∞							
6	∞	∞	∞				∞	∞	∞	∞
7	∞	∞	∞	∞				∞	∞	∞
8	∞	∞	∞	∞	∞				∞	∞
9	∞	∞	∞	∞	∞	∞				∞
10	∞	∞	∞	∞	∞	∞	∞			

Technically, instead of ∞ a sufficiently big enough number can be used. For example, a sum of every non-∞ number in the table. Then, optimal solution will not include such a cell, if only there exists a solution that doesn't include such cell.

Lastly, we need to put values in every other cell, which will represent the cost of each job according to our objective function. As the only important points for the any objective function are completion times of each job, we can fill all cells, except for the cells in each job's last row, with zeroes.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞									
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	0	0	∞
5	∞	∞	∞							
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞			

In last rows we will put the value of our penalties. For example, the first job's due date is 1 and the weight is 1 as well. So, if we finish it at the 1st interval of prior, we won't suffer any penalties. But that is impossible. If we finish it at 2nd period, the penalty will be 1, at 3rd period — 2, at 4th period — 3, and so on.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	1	2	3	4	5	6	7	8	9
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	0	0	∞
5	∞	∞	∞							
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞			

For the second job with due date 1 and weight 2, it is possible to finish at the 4th time period, suffering a penalty of $2 \cdot (4 - 1) = 6$. Finishing at the 5th interval will result in a penalty of 8, 6th — 10, and so on. And analogously for the third job.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	1	2	3	4	5	6	7	8	9
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	0	0	∞
5	∞	∞	∞	6	8	10	12	14	16	18
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	15	18	21

Finally, we have formulated our problem as an assignment problem, reflected by a matrix.

4 Binary branch and bound

We will use the an optimal AP solution as a lower bound to any feasible solution. For solving the AP we will use a modification of Jonker-Volgenant algorithm [4], which produces solution in $O(n^3)$ time in the worst case. In examples, for simplicity, we will instead use the hungarian algorithm (HA)[5], which serves as the base of Konker-Volgenant algorithm and is almost identical is steps mentioned. Using that mechanism, we will employ branch and bound algorithm to solve our original problem to optimality. We will demonstrate how that is done using previous numerical example for which we will clarify each step.

To begin with, we will use the hungarian algorithm to find an optimal solution of the AP. First, we need to reduce by rows and columns — i.e. to subtract a biggest possible number from each row such that the numbers in the row stay non-negative. Now every row has at least one zero. If zeroes form an *independent set* — a set of zeroes such, that there is exactly one zero in each row and each column — then we have found the optimal solution in terms of AP. But it might not be so, as some of the zeroes can be in the same column. Then, we also reduce by columns. Now every row and every column has a zero. If zeroes form an independent set, we have found an optimal solution. Notice that the sum of subtracted numbers is a lower bound for the optimal solution.

	1	2	3	4	5	6	7	8	9	10	
1	0	0	0	0	0	0	0	0	0	∞	
2	∞	0	1	2	3	4	5	6	7	2	-1
3	∞	0	0	0	0	0	0	0	∞	∞	
4	∞	∞	0	0	0	0	0	0	0	∞	
5	∞	∞	∞	0	2	4	6	8	10	6	-6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞	
7	∞	∞	∞	∞	0	0	0	∞	∞	∞	
8	∞	∞	∞	∞	∞	0	0	0	∞	∞	
9	∞	∞	∞	∞	∞	∞	0	0	0	∞	
10	∞	∞	∞	∞	∞	∞	∞	0	3	0	-15
										-6	

Next, we need to find, if possible, an independent set of zeroes in the table. Remember that current lower bound for the optimal solution is 1+6+15+6 = 28.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	0	0	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

There is an independent set of zeroes in this matrix. It is a feasible solution for the LAP relaxation, but it is not feasible for our scheduling problem, because we can't first finish 2nd job at 4th period (cell $(5, 4)$) and only then do its second part (cell $(4, 9)$). Could it be that there is another independent set of zeroes, corresponding to the feasible solution?

Let's analyze. If there's only one zero in a row or a column, then we must select that zero for the solution. Such zeroes are $(1, 1)$ for column 1, $(2, 2)$ for row 2, $(5, 4)$ for row 5 and $(10, 10)$ for row 10. These cells are marked bright-red in the following table. These zeroes are a must-have, but there is more than one way to choose an independent set from the rest of the zeroes.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	0	0	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

Now, notice that we are working on job 1 in 1st and 2nd periods (cells $(1,1)$ and $(2,2)$) and then somehow finish job 2, requiring 3 periods, at 4th period (cell $(5,4)$). So these required cells by themselves produce an infeasibility.

Now, we will use branch and bound algorithm to search for the feasible solution. Remember, that the hungarian algorithm's solution is always a lower bound to our problem. Current lower bound is 28. Let's now see, how the answer changes if we *forbid* sell $(5,4)$. To do so, we can simply change its value to inf. Then, we again solve the LAP problem with hungarian algorithm.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	0	0	∞
5	∞	∞	∞	∞	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

Now we use hungarian algorithm once again to determine optimal solution. Notice, also, that the 5th row is reduced by 2, hence the lower bound is increased to $28 + 2 = 30$.

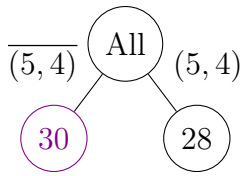
	1	2	3	4	5	6	7	8	9	10	
1	0	0	0	0	0	0	0	0	0	∞	
2	∞	0	1	2	3	4	5	6	7	2	
3	∞	0	0	0	0	0	0	0	∞	∞	
4	∞	∞	0	0	0	0	0	0	0	∞	
5	∞	∞	∞	∞	0	2	4	6	8	4	-2
6	∞	∞	∞	0	0	0	∞	∞	∞	∞	
7	∞	∞	∞	∞	0	0	0	∞	∞	∞	
8	∞	∞	∞	∞	∞	0	0	0	∞	∞	
9	∞	∞	∞	∞	∞	∞	0	0	0	∞	
10	∞	∞	∞	∞	∞	∞	∞	0	3	0	

Now, that is obviously a feasible solution. But overall lower bound is 28, and this solution gives us 30. So we haven't solved the problem to optimality yet. We still need to work through another option regarding cell $(5, 4)$: if it's not included in the solution, then the answer is 30; but if it is included, the answer is 28. Let's develop the second option to see, if we can find a better solution, when cell $(5, 4)$ is included in the solution. To do so, we will *forcibly include* that cell. Let's return to the solution, which included $(5, 4)$.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	0	0	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

Note that if $(5, 4)$ is included, then we needn't to look at its row as we already know, at what time will the 5th job be done, and we needn't look at its column, as we know, what job will be done at 4th period of time.

Our current branching:

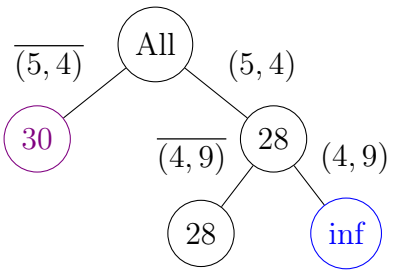


$\overline{(5,4)}$ stands for the case, where we forbid cell $(5,4)$. The other branch stands for the case, where $(5,4)$ is forcibly included. The violet color means that there exists a feasible solution with corresponding answer. Right now, we know that the best solution in the left branch gives us answer 30, but there might be other solutions with answers 28 or 29 in the right branch. To find out, we will branch further. But we already have an assumption, that there is no feasible solutions with answer 28. To capture that idea, let's start branching with cell $(4,9)$, which is used in current optimal, but infeasible solution. Notice that if it's included, then any such solution is infeasible, as we have already forcibly included $(5,4)$ — and including a pair $(4,9),(5,4)$ by itself leads to an infeasible solution. Such infeasibility-seeking branching will reduce the depth significantly.

We still need to see, what happens if we forbid $(4,9)$. It can easily be seen, that there exists another solution with answer 28, even if $(4,9)$ is forbidden:

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	0	∞	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

Current branching tree, where "inf" and blue color stand for certain infeasibility of the whole branch:



Branching further, let's forbid all cells in row 4, which are later than 4th period, because all these solutions are infeasible, but optimal for relaxed problem. See below, how it's done.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	∞	∞	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	∞	∞	∞	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

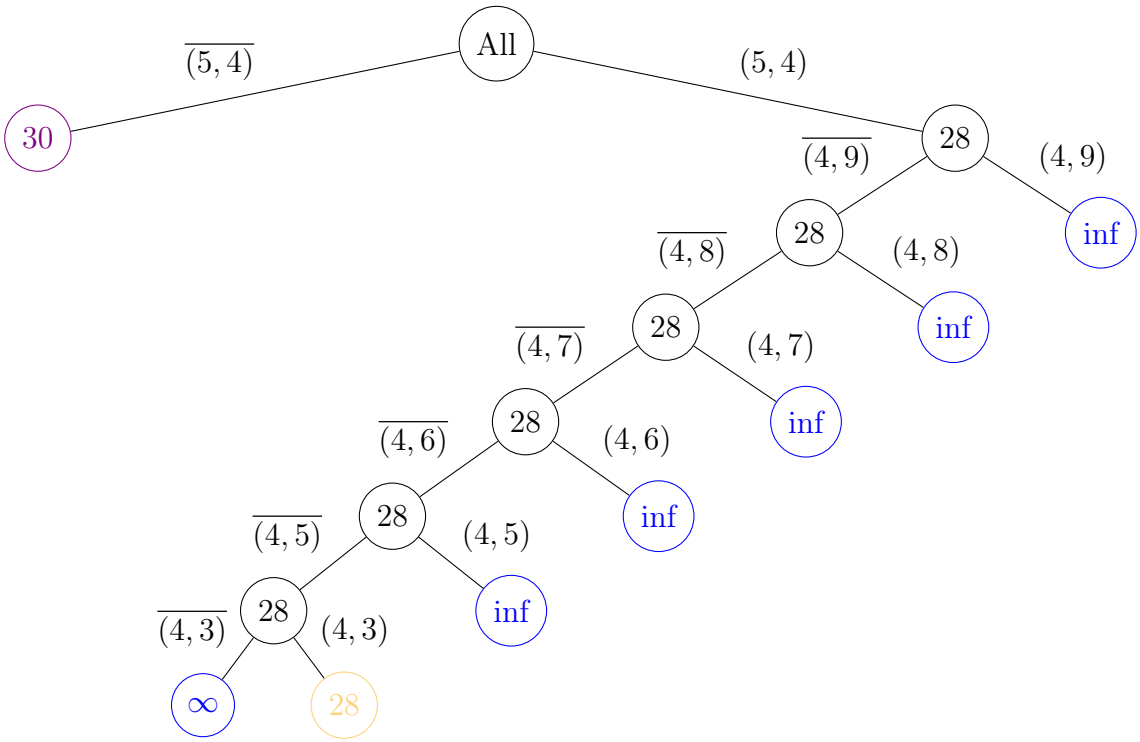
	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	∞	∞	∞	∞	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

At this point, we can't pick $(4,4)$ already and if we forbid cell $(4,3)$, we will have a solution with ∞ answer, as $(4,3)$ is the only available non-∞ cell left. So, basically considering two branching options, just doing it in text, rather than in tables, we are now on the branch, where cell $(4,3)$ is forcibly included. Then we also fix the 4th row and the 3rd column. Current table:

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

Current branching, where the last table refers to the yellow node:



A quick-witted reader could have already developed a question: "Why wouldn't we just forbid all the cells in the 4th row, which come after the 4th, because selecting them would anyway result in an infeasible solution?" Indeed, we will do just that from now on. A proper proof for that idea will be given later. Now, when we have selected $(5,4)$ and $(4,3)$ — second and third parts of 2nd job — we know, that the first part must be done beforehand, i.e. before 3rd period. So we can just ban every period after the 3rd:

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	∞	∞	∞	∞	∞	∞
4	∞	∞	0	0	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

And now we see that in this case, the only non-∞ cell left if $(3, 2)$, which interferes with the $(2, 2)$ cell, which we can move, but paying the cost of at least 3 answer increase. But let's also see, how the hungarian algorithm would have solved this problem. First of all, current matrix, dropping all allocations, except the selected by branching, as we don't know the optimal solution yet:

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	0	1	2	3	4	5	6	7	2
3	∞	0	0	0	∞	∞	∞	∞	∞	∞
4	∞	∞	0	0	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	0	2	4	6	8	10	6
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	0	3	0

Now, as there's nothing to allocate in the green rows and columns, we can delete them and work with a reduced matrix:

	1	5	6	7	8	9	10
1	0	0	0	0	0	0	∞
2	∞	3	4	5	6	7	2
6	∞	0	0	∞	∞	∞	∞
7	∞	0	0	0	∞	∞	∞
8	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	0	3	0

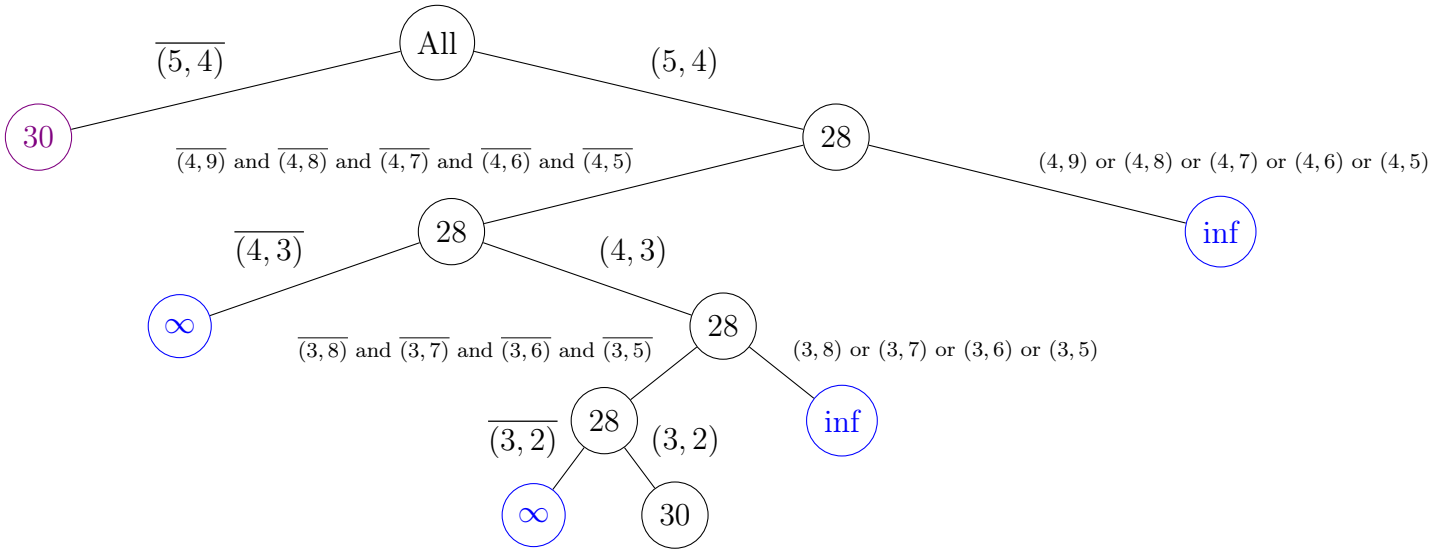
Now, there are no zeroes in the 2nd row, so we reduce it by 2, getting a lower bound of $28 + 2 = 30$. And there will exist an independent set of zeroes:

	1	5	6	7	8	9	10	
1	0	0	0	0	0	0	∞	
2	∞	1	2	3	4	5	0	-2
6	∞	0	0	∞	∞	∞	∞	
7	∞	0	0	0	∞	∞	∞	
8	∞	∞	0	0	0	∞	∞	
9	∞	∞	∞	0	0	0	∞	
10	∞	∞	∞	∞	0	3	0	

The set is not feasible, but it doesn't matter, as the lower bound is already 30 — and we have a feasible solution with that answer. So, we needn't search

for feasible solutions in this branch, as they all will have an answer of at least 30. And we already have a solution with such answer.

Then, our full branching tree:



Notice, that each leaf has a lower bound of at least 30 or is certainly infeasible, meaning that there are no feasible solutions with answer less than 30. And we've found a feasible solution with answer 30, so the problem is solved. Let's now just check our solution on the original matrix to see, if the answer really if 30:

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	∞
2	∞	1	2	3	4	5	6	7	8	9
3	∞	0	0	0	0	0	0	0	∞	∞
4	∞	∞	0	0	0	0	0	0	0	∞
5	∞	∞	∞	6	8	10	12	14	16	18
6	∞	∞	∞	0	0	0	∞	∞	∞	∞
7	∞	∞	∞	∞	0	0	0	∞	∞	∞
8	∞	∞	∞	∞	∞	0	0	0	∞	∞
9	∞	∞	∞	∞	∞	∞	0	0	0	∞
10	∞	∞	∞	∞	∞	∞	∞	15	18	21

And the answer is $21 + 8 + 1 = 30$.

In the proceedings of the hungarian algorithms one case wasn't analyzed. Even after reducing by rows and by columns, zeroed in the matrix may not

form an independent set. Following example illustrates that:

	1	2	3
p_i	2	3	1
r_i	1	1	2
d_i	0	0	0
w_i	3	2	5

Notice also, that is every job's due date is 0, then TWT objective is equivalent to the TWC objective. Lets now build the corresponding AP matrix.

	1	2	3	4	5	6
1	0	0	0	0	0	∞
2	∞	6	9	12	15	18
3	0	0	0	0	∞	∞
4	∞	0	0	0	0	∞
5	∞	∞	6	8	10	12
6	∞	10	15	20	25	30

Now to reduce rows and columns.

	1	2	3	4	5	6	
1	0	0	0	0	0	∞	
2	∞	0	3	6	9	6	-6
3	0	0	0	0	∞	∞	
4	∞	0	0	0	0	∞	
5	∞	∞	0	2	4	0	-6
6	∞	0	5	10	15	14	-10
						-6	

There is no independent set of zeroes in this matrix, because the only zeroes in the 2nd and 6th row are both in the 2nd column. In this case, the hungarian algorithm covers every zero with minimal possible number of rows or columns. If more than one cover exists, any one of them is sufficient. For example:

	1	2	3	4	5	6
~~1~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~∞~~
2	∞	0	3	6	9	6
~~3~~	~~0~~	~~0~~	~~0~~	~~0~~	~~∞~~	~~∞~~
~~4~~	~~∞~~	~~0~~	~~0~~	~~0~~	~~0~~	~~∞~~
~~5~~	~~∞~~	~~∞~~	~~0~~	~~2~~	~~4~~	~~0~~
6	∞	0	5	10	15	14

Notice, that a minimal cover covers every cell iff and independent set exist. For that reason, many implementations of hungarian algorithm find a cover instead of looking for an independent set of zeroes. Here, some cells are not covered. And if this is the case, the algorithm reduces not covered cell by the minimal value, thus creating additional zero. Here, the minimal not covered value is 3.

	1	2	3	4	5	6
1	0	0	0	0	0	∞
2	∞	0	3	6	9	6
3	0	0	0	0	∞	∞
4	∞	0	0	0	0	∞
5	∞	∞	0	2	4	0
6	∞	0	5	10	15	14

Now, reduce all not covered cells by that value, adding it to a lower bound. For why that will be a lower bound, go to original paper[5].

	1	2	3	4	5	6
1	0	0	0	0	0	∞
2	∞	0	0	3	6	3
3	0	0	0	0	∞	∞
4	∞	0	0	0	0	∞
5	∞	∞	0	2	4	0
6	∞	0	2	7	12	11

Now we have one more zero and an independent set exists.

	1	2	3	4	5	6
1	0	0	0	0	0	∞
2	∞	0	0	3	6	3
3	0	0	0	0	∞	∞
4	∞	0	0	0	0	∞
5	∞	∞	0	2	4	0
6	∞	0	2	7	12	11

And it actually corresponds to a feasible solution. The answer, as a sum of values reduced by, should be $6 + 6 + 10 + 6 + 3 = 31$.

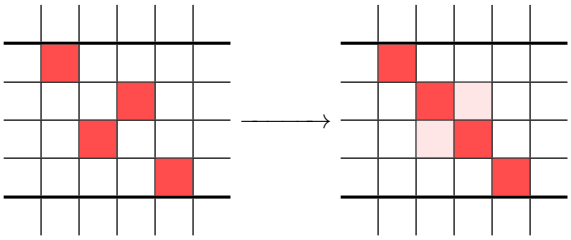
	1	2	3	4	5	6
1	0	0	0	0	0	∞
2	∞	6	9	12	15	18
3	0	0	0	0	∞	∞
4	∞	0	0	0	0	∞
5	∞	∞	6	8	10	12
6	∞	10	15	20	25	30

On the initial matrix the answer is $9 + 12 + 10 = 31$, as expected.

5 Wide completion time branching

5.1 Reordering

Let's once more consider what we deem infeasible.



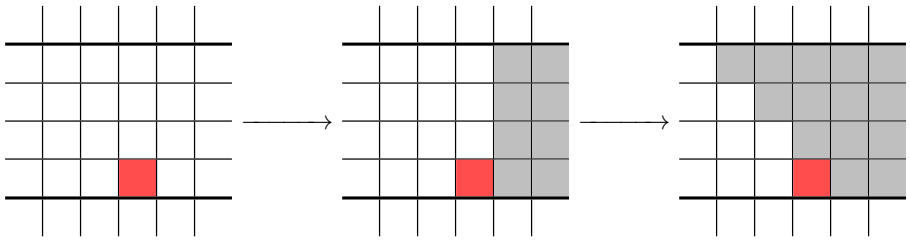
Consider a schedule on the left. Technically, it is infeasible, because second and third operations are done in a wrong order. But they can actually be reordered to a feasible schedule without change in the objective function, because non-infinity cell, except for the cells in completing operations rows, cost 0. An example of such reordering can be seen in a table on the right.

Theorem 1. (Reordering) For a job j with processing time p_j and operations $o_1, o_2, \ldots, o_{p_j}$ and a schedule S, which assigns o_i to a time interval t_i, if $\forall i < p_j t_i < t_{p_j}$(last operation is processed last in time), then there exists a schedule S' with the same cost, in which other jobs' assignment is unchanged and job j's assignment is feasible.

Proof. We sort available times $\{t_i\}_{i=1}^{p_j}$ in ascending order $t'_1 < t'_2 < \ldots < t'_{p_j}$. Then we assign o_i to t'_i and other jobs as in the schedule S to produce schedule S'. Now, every operations are produced in a right order and, as the last operation was produced last as-is, it is hasn't changed its time interval. The completion times are the same, hence the cost is unchanged. ■

Therefore, we we will consider a job unfeasible if it's last operation is not processed last. Also, we formally introduce a *shortcutting* procedure. Imagine that we fixed a certain cell in the table. Then we can impose a restriction on which cells could be used in that job's previous operations. Now look at the example below. Grey cells can be forbidden, since choosing them will result in

an infeasible solution.



Proposition 1 (shortcutting). If for job j it's i-th operation o_i is fixed at a time interval t_0, then assigning k-th($k < i$) operation o_k to a time interval $t > t_0 - (i - k)$ will result in an infeasible solution.

Proof. Imagine that $t > t_0 - (i - k)$. Then, even if every other operation is processed right away, o_i will be processed at a $t + (i - k) > t_0$ interval, which results in a contradiction. ■

To get most from the shortcutting procedure, it makes sense to branch only on the last operations to shortcut every previous operation. It also makes sense, because they're the only cells, which have a price on them. While other cells only indirectly affect the schedule cost through afflicting the last operation cells with infeasibility constraints.

We will now switch to a *wide branching on completion times*. At the root, we have the relaxed to assignment problem. At each node, we will solve LAP to produce a lower bound on a scheduling solution. If the solution cost is higher than an already found solution, we prune the branch by bound. If that solution is feasible, we prune the branch by optimality. Otherwise, we choose a job, which is infeasible, and generate a child for each possible completion time, where that time is fixed for the last operation.

Another positive is that we can get rid of working through infeasible nodes and omit infeasibility checks and pruning whatsoever. To do that, we will have to employ a bit of more complicated ideas.

Corollary 1. If a job's completion time is fixed, other operations' cells are accordingly shortcutted and if a LAP solution exists, then there always exists LAP solution with the same cost, in which other jobs' assignment is unchanged and that job's assignment is feasible.

Proof. With shortcutting we know that the last operation is processed last. Hence, from Theorem 1 immediately follows that such a schedule exists. ■

But this way we're still stuck with the problem of LAP solution existence, since we obviously can fix completion times in faulty ways. To solve that problem, we introduce constraints on which completion times can be chosen for a job based on previously fixed completion times. That will also reduce number of each node's children, since not every completion time is possible.

5.2 Completion times feasibility

Now we will **relax release constraints**. That is, we will completely disregard release times and will allow for jobs to be processed even before their release date. That obviously only widens number of feasible solutions and will provide us with good lower bounds on objective function.

Theorem 2. (With relaxation of release constraints) For any set of jobs A with fixed completion times a schedule exists if and only if $\forall A' \subseteq A : \sum_{j \in A'} p_j \leq \max_{j \in A'} c_j$.

That is, if you select any subset of jobs, the whole subset is finished at $\max_{j \in A'} c_j$ and the sun of processing times $\sum_{j \in A'} p_j$ is not larger than this time.

Proof. It is obvious that if the condition is wrong, then a certain subset can't be physically processed in time by itself, which can not be alleviated by adding other jobs.

If the condition is satisfied, then there must exist a schedule. We will prove its existence by constructing such schedule. Before we start, we will renumber all jobs in A such that $c_1 < c_2 < \ldots < c_m$.

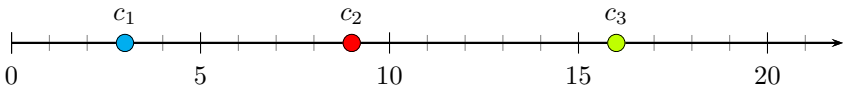
First, we will place job 1 completion time at c_1 and schedule all its operations before c_1 — it doesn't matter to which intervals specifically. That is possible, because $p_1 < c_1$ — which is a constraint satisfied for $A' = \{1\}$.

Next, we will schedule jobs consequentially. If k jobs has already been scheduled, then we schedule job $(k+1)$ last operation at c_{k+1} — it's available, since the whole scheduling was happening in time intervals $\leq c_k < c_{k+1}$. And we will schedule all the other operations at available intervals before c_{k+1}. There are enough intervals, because for $A' = \{1, \ldots, k+1\}$ the constraint is $\sum_{j=1}^{k+1} p_j \leq c_{k+1}$. ■

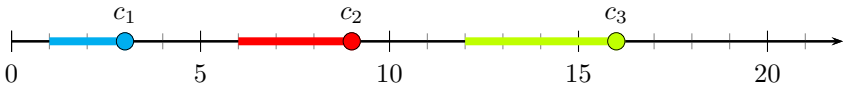
If we can efficiently employ Theorem 2, we can always be sure that the fixed completion times induce a feasible schedule. Hence, all infeasibilities will be coming from LAP solution and branched upon.

Let's look at an example, which would also explain how to easily find completion time bounds. Here we will schedule first 4 jobs with processing and release times as in the table on the right. Imagine that we've already fixed following completion time for jobs 1, 2 and 3:

	1	2	3	4	...
p_j	2	3	4	?	
r_j	0	2	5	0	

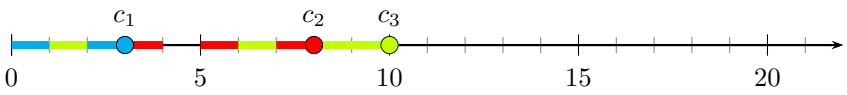


Notice that there is a lot of possible schedules for these completion times such as



That example is the simplest scheduling, because schedules might look like

the following:



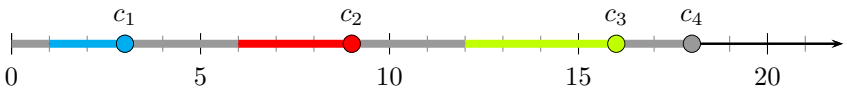
But, because only completion operations change the objective function it doesn't matter how specific operations are distributed among themselves. Instead of thinking about other operation, we will simply make use of Theorem 2. It will tell us whether a schedule with specific completion times exist. Let's see how exactly it works. While proving the theorem, we already saw that we don't actually have to check all 2^A subsets of fixed jobs. Let's properly prove that.

Lemma 1. (With relaxation of release constraints) For any set of jobs A with fixed completion times number all jobs in A such that $c_1 < c_2 < \ldots < c_m$. Then a schedule exists if and only if $\forall 1 \leq k \leq m : \sum_{j=1}^{k} p_j \leq c_k$.

Proof. Notice that $c_k = \max_{\{1 \leq j \leq k\}} c_j$. Then, as before, it is obvious that if the condition is wrong, then a certain subset can't be processed in time.

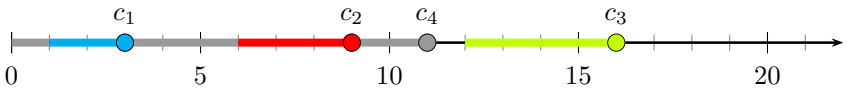
To proof that if the condition is satisfied, then a schedule exists, we will prove that lemma's condition is equivalent to the Theorem 2 condition. $\forall A' \subseteq A$ let k be the largest job index in A'. Then, let $A_k = \{1, 2, \ldots, k\}$. Clearly, $A' \subseteq A_k$. Then, if A_k can be scheduled with such fixed processing times, then A' can be scheduled as well by simple removing some jobs if necessary. Hence $\forall A' \subseteq A$ can be scheduled if every A_k can be scheduled. The reverse is obviously true as $A_k \subseteq A$. Therefore the condition of Lemma 1 and Theorem 2 are equivalent. ■

Using that lemma, we can efficiently produce bound on completion times. Let's return to the example. First, we calculate how much empty time intervals are there before the rightmost completion point c_3. In this example, there are 7 of them. Hence, if p_4 is greater than 7, then $c_4 > c_3$. More specifically, $c_4 \geq c_3 + p_4 - 7$.



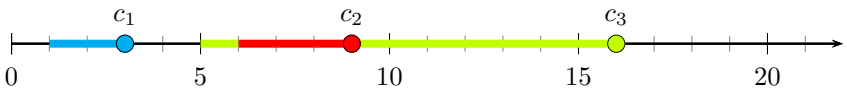
The logic behind it is simple: even if we fill in every blank and take all leftmost intervals, even then we will complete the job only at moment $p_1 + p_2 + p_3 + p_4 = c3 - 7 + p_4$.

But if p_4 is not greater than 7, then it 4th job can be completed before c_4. Then, if $7 \geq p_4 > 4$, it can be sompleted before c_3, but not before c_2 — there are only 4 empty intervals before c_2. More specifically, $c_4 \geq c_2 + p_4 - 4$.



And so on. In example such as that, it is easy to find a smallest possible c_4. Just move c from right to left, checking each time if there are enough free intervals to complete at time c. But how to efficiently check it?

Turns out, just calculating free space naively as $c_k - \sum_{j=1}^{k} p_k$ is correct, when continuously considering time from right to left. Checking each job consequentially from right to left is important with that approach. Consider a following example:

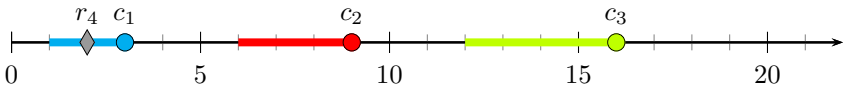


Here, for the second job that calculation would give us $c_2 - (p_1 + p_2) = 9 - (2 + 3) = 4$ empty intervals. While in reality we only have 3. And a job with $p_4 = 4$ would seem fitting, when it actually isn't. But if we checked the third job before, we would see that $c_3 - (p_1 + p_2 + p_3) = 16 - (2 + 3 + 8) = 3$ and we would see that there's only 3 empty intervals.

That approach is correct because of Lemma 1. Let's say we are trying to complete job at interval c_0. Then, we would have checked for every $c_k > c_0$ that $p_0 + \sum_{i=1}^{k} p_k \leq c_k$ — Lemma 1 condition. And the condition for every $c_k < c_0$ hasn't changed. Hence, we only need to consider that the condition for c_0 itself is correct and check it each time we are moving from $c_0 > c_k$ to $c_0 < c_k$.

Using appropriate data structures it is possible to find smallest possible c_0 in linear time. It might be possible to improve that result by keeping track of empty intervals rather than $\sum p_j$ and order of c_j.

The other thing we still need to consider is release times. Imagine that $r_4 = 2$. Then there is an empty interval before c_1, but it can not be used by job 4.



Therefore there still are ways to improve width of branching, especially for jobs with large release dates. Technically, algorithm will shortcut and determine right away that there is no feasible solution possible. But that improvement might hold theoretical insight into the optimal schedule.

Lastly, as it is very important for any branch and bound algorithm, we use a WSRPT heuristic [2] to produce initial upper bound on the cost of an optimal solution.

6 Competition

"Modeling single machine preemptive scheduling problems for computational efficiency" [1]

The authors are solving preemptive scheduling problems with TWT, TWCT, TWET, TWNTJ objectives. They present two ILP models: binary preemptive scheduling (BPS) and aggregate preemptive scheduling (APS). Then, they relax them to the LP problems, which are used to produce lower bounds on feasible solutions to the original problem. The models are explained in detail and an

analytical comparison is given. The authors introduce capacity α and amount of work p_i, from which we can derive more conventional processing time by diving these values ($\frac{p_i}{\alpha}$). Computational experiments are performed on up to 160 jobs. Processing times are uniformly distributed in $[0.6, 1.4]$, capacity α is uniformly distributed in $[0.15, 0.5]$. Hence, the maximum processing time is $max\{\frac{p_i}{\alpha}\} = \frac{max\{p_i\}}{min\{\alpha\}} = \frac{1.4}{0.15} \approx 9.33$ and the mean processing time is $\frac{1}{0.325} \approx 3$. The models don't always solve the problem to optimality even with 20 jobs and never solve to optimality starting with 80 jobs.

7 Computational experiments

We perform computational experiments to evaluate wide completion time branching algorithm performance. The simplest TWCT objective function is used. The performance is investigated depending on problem size in two ways: number of jobs and jobs' processing times. All computations are performed on a PC with i5-7300HQ processor with 3.5 GHz and 8 GB of RAM.

Data not mentioned in the table is distributed as $w_j \sim U[1, n]; r_j \sim U[1, 5]$.

Number of jobs	Number of tests	p_j	min	max	mean
9	30	U[1, 9]	0.427	3352	490
8	30	U[1, 9]	0.04	67	11
7	100	U[1, 9]	0.02	90	2.2
6	100	U[1, 9]	0	0.677	0.07
5	100	U[1, 9]	0	0.78	0.005
11	30	2	0.2	1137	113
10	30	2	0.216	158	14
9	100	2	0.009	9.5	0.9
5	30	U[20,40]	1.5	85	23
4	100	U[20,40]	0.005	1.6	0.25

The other set of experiments, more similar to [1], with $r_j \sim U[1, 2n]$. Problems with such characteristics are not always idle-free, i.e. there exists intervals, when no job is processed. As we work with idle-free scheduling, we will skip such examples and redo generation phase. If a problem requires idleness, it can be fairly simply modified to become idle-free.

Number of jobs	Number of tests	p_j	min	max	mean
9	30	U[1, 9]	0	285	37
8	30	U[1, 9]	0.001	368	18.7
7	30	U[1, 9]	0	10	0.95
11	30	3	0.04	698	87
10	30	3	0	468	24
9	100	3	0	64	1.6
8	100	3	0	1.6	0.1

These results are comparable with recent state-of-art results[1], as the problem's characteristics are mostly distributed the same way. Such small problems are solved somewhat faster than by APS model and an order of magnitude faster than BPS model. Without a way to directly compare the algorithms, it is fair to say that they are equal.

Larger problems are not always solved to optimality in reasonable time. Hence, to establish solution's quality, it is required to perform bounds and integrality gap calculations, which our algorithm is not adapted for.

Also, peculiarly, almost 95% of instances are solved optimally by WSRPT heuristic [2].

8 Model's flexibility

We considered one of the most popular objectives in single machine preemptive scheduling, but can we use this approach to compute other objectives or add new constraints? Everything that can be precomputed into AP matrix can then be solved by our algorithm. Which favorably distinguishes our model from other models. This includes:

1. Other objective functions. If we can calculate, how costly it will be to finish a certain job at a certain point, then we can put that cost into the corresponding cell in the matrix and then solve the problem. Including, but not limited to, total weighted completion time, total weighted earliness, total weighted number of tardy jobs. And non-linear objectives too. For example, total weighted squared tardiness, which is formulated as $\sum_{j=1}^{n} w_j t_j^2$, as it can be computed and put in the matrix. Any function, which depends on initially known problem's parameters will suffice. The approach might be inapplicable, if, for example, the objective functions depends on the order of the job, i.e. the penalty is increased, if a certain job is done after some other job.

2. Deadlines. If a job must be finished by a certain point in time, we can fill its finishing operation's row with ∞ after the deadline date, forcing every solution to finish this job before the deadline.

3. Weights changing in time. For some tasks, a 1 or 2 interval tardiness may be unwanted, but bearable; 2 or 3 interval tardiness is troublesome;

and more than 3 intervals tardiness is unacceptable. For example, when building roads, being late on schedule is bad, but generally acceptable. Not finishing a road by the time the machinery used for the construction is needed in some other place is worse. And finishing it in snowy winter is unacceptable, as the quality decreases drastically. Which can be modeled as a weight function of time $w_j(t)$. Then, whichever objective you choose, we still can precompute each cell's cost and fill AP matrix with corresponding values.

4. Sub-tasks costs. Job may consists of several subtasks, each of which has a certain cost. For example, a construction company may have several objects to work — these will be our jobs. Each object's construction consists of several smaller parts, such as foundation placement, carcass building and facing. Notice that these sub-tasks must be performed in a specific order, thus breaking them into different jobs may not produce correct results. And each sub-task may have a specific due date of its own with a penalty for failing the requirement. We can reflect it in out model by placing costs on non-finishing operations, instead of always putting zeroes in corresponding cells.

5. Forbidden cells. For example, if all carpenters go on a special training on the 6th interval, we can place ∞ in each row representing their objectives in the 6th column. That will also work for holidays, vacations and more people's rigid schedules.

And any other specification you can think of, if only it can be fit into an AP matrix.

Good examples of not fitting models are:

1. Non-preemptive scheduling or objective functions, which penalize for preemptions.

2. Makespan($max_j c_j$) or maximum lateness($max_j(c_j - d_j)$) objective functions.

9 Conclusion and further work

We have presented two novel Branch-and-Bound-algorithms for single machine preempted scheduling problem. Experimental results are close to the existing state-of-the-art results, but are not exactly superior. On the other hand, our model is much more flexible and allows for numerous modifications. It would be interesting to develop each of them and compare to the existing results — if there are any. Second approach to further research is improvements of preprocessing. There are scheduling insights lurking, which were left undeveloped in the paper. For example, we could probably fix a disproportionately important job's processing unpreempted even before starting the algorithm. That would boost the performance significantly, especially on specific distributions. Other

possible approach is to find tighter lower bounds for branching algorithms, as LAP relaxation is a very strong one, often resulting in a substantially lower costs. Lastly, it might be possible to modify algorithms to produce bounds if stopped prematurely, although there are difficult complications on that path, such as memory limitations.

References

[1] Fernando Jaramillo, Busra Keles, Murat Erkoc.
Modeling single machine preemptive scheduling problems for computational efficiency.
Annals of Operations Research (2020) 285:197–222
Link

[2] Mikhail Batsyn, Boris Goldengorin, Panos M. Pardalos, Pavel Sukhov.
Online heuristic for the preemptive single machine scheduling problem of minimizing the total weighted completion time
Link

[3] Mikhail Batsyn, Boris Goldengorin, Pavel Sukhov, and Panos M. Pardalos.
Lower and Upper Bounds for the Preemptive Single Machine Scheduling Problem with Equal Processing Times.
Springer Proceedings in Mathematics and Statistics. 59. 11-27. 10.1007/978-1-4614-8588-9_2.
Link

[4] Jonker, R., Volgenant, A.
A shortest augmenting path algorithm for dense and sparse linear assignment problems.
Computing 38, 325–340 (1987).
Link

[5] H.W. Kuhn, Bryn Yaw.
The Hungarian method for the assignment problem.
Naval Res. Logist. Quart, 1955
Link

[6] Lenstra, J. K., Kan, A., Brucker, P.
Complexity of machine scheduling problems.
Annals of Discrete Mathematics(1977), 1, 343–362.