To appear in the *International Journal of Production Research*
Vol. 00, No. 00, 00 Month 20XX, 1–22

A branch and bound algorithm for scheduling unit size jobs on parallel batching machines to minimize makespan

Onur Ozturk[a], Mehmet A. Begen[b*] and Gregory S. Zaric[b]

[a]*Université Paris-Est, ESIEE Paris, Département Ingénierie des Systèmes, 2, boulevard Blaise Pascal Cité Descartes BP 99 93162 Noisy le Grand Cedex, France;* [b]*Ivey Business School, Western University, London ON Canada*

(*Received 00 Month 20XX; accepted 00 Month 20XX*)

In this paper we present a branch and bound algorithm for the parallel batch scheduling of jobs having different processing times, release dates and unit sizes. There are identical machines with a fixed capacity and the number of jobs in a batch cannot exceed the machine capacity. All batched jobs are processed together and the processing time of a batch is given by the greatest processing time of jobs in that batch. We compare our method to a mixed integer program as well as a method from the literature that is capable of optimally solving instances with a single machine. Computational experiments show that our method is much more efficient than the other two methods in terms of solution time for finding the optimal solution.

Keywords: scheduling, batch scheduling, parallel machines, makespan, branch and bound, heuristics, mathematical programming

1. Introduction

In this study, we investigate the parallel batch scheduling of unit size jobs with different processing times and release dates on parallel identical machines. Each machine can process multiple jobs simultaneously as long as the capacity constraint is not violated. All jobs processed at the same time constitute a single batch. The processing time of a batch is given by the longest processing time of any job contained in that batch. The objective is to minimize makespan. Batch processing is encountered in casting, metallurgy, aircraft manufacturing, burn-in operations of integrated circuit and sterilization services of hospitals (Mathirajan and Sivakumar (2006); Ozturk, Begen, and Zaric (2014)).

Using Graham's notation (Graham et al. (1979)), this is a $P|p-batch,\ r_j, p_j, B < n|C_{max}$ scheduling problem. In this notation, P stands for identical parallel machines and $p-batch$ for parallel batching, i.e., more than one job can be processed at the same time in a machine. Job release dates and processing times are denoted by r_j and p_j, respectively. The number of jobs is given by n and B is the batch/machine capacity. Finally, C_{max} refers to the objective function, i.e., minimization of makespan.

Our problem is NP hard by the following argument. A special case of our problem is the case in which each batch can process a single job at a time, i.e., a classical scheduling problem in the presence of jobs with release dates, different processing times and parallel machines. This special case was shown to be NP-hard (Pinedo (2012)); thus, our problem is also NP-hard. Although there are exact solution procedures for the case of a single batching machine, there are only heuristics,

*Corresponding author. Email: mbegen@ivey.uwo.ca meta-heuristics and approximation algorithms in the literature for the case of parallel machines. In this paper, we develop a branch and bound algorithm ($B\&B$) which is able to find the optimum solution much faster than the state of art branch and bound algorithm proposed by Sung and Choung (2000) for single machine problems. Moreover, our method is able to solve parallel machine instances in a reasonable amount of time when the number of machines in problem instances is small.

The remainder of this paper is organized as follows. In section 2, we provide a literature review of batch scheduling problems. Section 3 is dedicated to the presentation of the branch and bound algorithm. Lower bounds and a heuristic method are also presented in section 3. Section 4 presents computational tests. In section 5, we conclude and propose some further research directions.

2. Literature Review

We review here only parallel batch scheduling literature considering different processing times, release dates and unit sizes for jobs. We classify the literature according to the solution methods. The literature review presented in this section is not exhaustive. We refer the reader to review articles by Potts and Kovalyov (2000) and Mathirajan and Sivakumar (2006) for further information about batch scheduling.

2.1 *Exact methods for NP-hard batching problems*

Sung and Choung (2000) studied the minimization of makespan on a single machine and presented a branch and bound algorithm which is based on enumerating every possible sequencing of jobs and applying lower bound cuts to minimize the search space. To the best of our knowledge, the algorithm they developed is the only exact method for our problem. In a second paper, Sung et al. (2002) took into account job families and they constructed their branch and bound tree as in Sung and Choung (2000). In both papers, dynamic programming algorithms were used to evaluate the optimum makespan value of a given job sequence at each node. Tangudu and Kurz (2006) also considered job families and proposed a branch and bound algorithm for minimizing total weighted tardiness. Yao, Jiang, and Li (2012) studied the minimization of the sum of job completion times in the presence of job families and a single machine, and also proposed a branch and bound algorithm.

While we are focused on the batch processing of unit size jobs in this paper, it is worth mentioning that the only exact methods for problems considering different job sizes apply to the case of equal release dates, i.e., $r_j = r \; \forall \; j$ (Uzsoy (1994); Dupont and Dhaenens-Flipo (2002); Malapert, Guéret, and Rousseau (2012); Parsa, Karimi, and Kashan (2010)).

2.2 *Heuristics and approximation methods*

Most solution methods for parallel batch scheduling in the literature are heuristics and approximation algorithms. Lee, Uzsoy, and Martin-Vega (1992) proposed heuristics based on list scheduling for the problem of minimizing makespan on parallel identical burn-in ovens. Lee and Uzsoy (1999) studied the minimization of makespan in the presence of a single machine and proposed heuristics based on longest processing time algorithm. Uzsoy (1995) considered the same problem taking into account job families. Deng et al. (2005) and Liu and Cheng (2005) proposed polynomial time approximation schemes for minimizing the sum of completion times and weighted sum of completion times, respectively, in the presence of a single machine.

Cheraghi, Vishwaram, and K.K. (2003) and Gupta and Sivakumar (2006) developed genetic algorithms for minimizing maximum lateness and maximum tardiness, respectively, on a single machine. When parallel machines are considered, Mönch et al. (2005) used genetic algorithms for minimizing total weighted tardiness. Beside heuristic approaches for parallel machines, we encounter also polynomial time approximation schemes in Li, Li, and Zhang (2005), Liu, Ng, and Cheng (2009) and Li and Yuan (2010).

Ozturk, Begen, and Zaric (2014) studied the batch scheduling of jobs having different release dates and sizes, but equal processing times. They presented a branch and bound heuristic based on a binary tree. Note that the difficulty of our problem is due to different job processing times and thus a binary tree as presented in Ozturk, Begen, and Zaric (2014) is not sufficient to cover all possible time moments to compute the optimal makespan value. The difficulty of their problem is due to different job sizes and even the special case with equal release dates is NP-hard. Parsa, Karimi, and Husseini (2016) also studied the case with different job sizes to minimize total flow time. They developed an ant colony method in which the optimal total tardiness of each job sequence is found with a dynamic programming algorithm. Zarook et al. (2015) took into account the aging effect of machines (which results in a decrease in the processing speed) and minimized makespan in an application of a maintenance planning scheduling. Another application in batch scheduling comes from integrated scheduling of a system of supplier, manufacturer and customers. Cheng et al. (2015), in addition to batch scheduling, considered the delivery of batches using a fixed capacity vehicle following a production process. Cheng, Li, and Hu (2015) and Cheng, Yang, and Hu (2016) also tackled the delivery issue after the processing of batches for different configurations (e.g., single customer or multiple customers, vehicle with different capacities). Agnetis, Aloulou, and Fu (2014) studied the coordination of batch production and interstage batch delivery using a third-party logistic provider. In a later paper, Agnetis et al. (2015) improved the dynamic programming algorithms developed in Agnetis, Aloulou, and Fu (2014).

Although several authors have studied similar problems, the work of Sung and Choung (2000) remains the state of the art for our problem and we use their algorithm for benchmarking.

2.3 *Contribution of this paper*

The batch scheduling literature is dominated by heuristic and meta heuristic methods. Sung and Choung (2000) proposed the only exact method for the parallel batching problem of jobs with different processing times, release dates and unit sizes in the presence of a single machine. We propose here a new branch and bound ($B\&B$) algorithm and test its quality on a wide range of instances. Numerical results show that our $B\&B$ can find the optimum solution much faster than the one proposed by Sung and Choung (2000). Moreover, the structure of our $B\&B$ allows generalizing the solution procedure for the case of parallel machines.

3. Branch and Bound Algorithm: $B\&B$

The branch and bound method that we propose consists of two phases. The aim of the first phase is to decompose the problem into two parts by determining an instant that we call the "critical instant (cI)". If all the jobs released earlier than cI can be processed before cI, then the rest of the jobs can be treated separately by the enumeration scheme at the second phase. To calculate the instant cI, we obtain first a modified subproblem by changing the processing times by setting them equal to the largest processing time among those jobs whose release dates are smaller than cI ($p_{max} \leftarrow max(p_j)\ \forall\ j$ such that $r_j < cI$). Then, the modified subproblem with equal processing times is optimally solved with the greedy algorithm of Ozturk et al. (2012). If all jobs can be batched and processed before cI, then those jobs are excluded from the original problem instance and hence we obtain a smaller problem to deal with. The pseudo code of the *critical instant* procedure is given in Appendix A.

The second phase of the branch and bound algorithm constructs a search tree by intelligently exploring instants that can be reached through different batch assignments. Each node v of the search tree is characterized by an instant t, list of machines and two job lists: $joblist_1$ representing the list of jobs which have been released but not processed, and $joblist_2$ representing the list of jobs which have not been released yet. Jobs in $joblist_1$ are thus ready for processing. Without loss of generality, jobs in each list are sorted in non-decreasing order of release dates. At each node, the decision is whether jobs in $joblist_1$ should be batched and processed immediately or if processing should be delayed until the next job (i.e., the first job in $joblist_2$) is released. (Note that deferring the processing of a job to an instant before the next job is released does not make sense.) Delaying the processing of a batch results in having child nodes for each parent node. In the branch and bound tree, the leftmost branch of a parent node represents processing no batch and waiting for a new job to be released, and all right branches represent the creation and processing of batches.

We give an overview of the branching scheme. Suppose there are M machines and k jobs are available for processing at node v.

1- Branch left by processing no job in $joblist_1$ and wait for the release of first job in $joblist_2$. This creates a child node representing all elements of $joblist_1$ plus the first job(s) of $joblist_2$.

2- Branch right by creating k different batches each having a processing time equal to processing time of k jobs in $joblist_1$. Apply a modified version of the longest processing time rule to jobs present at node v to fill each batch without changing the processing time.

3-Assign a new created batch to each of M machines. This results in M child nodes.

4- Use lower bound methods to make necessary cuts.

We describe these steps more in detail in the following sections. The pseudo code of the branch and bound algorithm is given in Appendix A.

3.1 ***Branching step***

As mentioned earlier, there are two types of branches at each node of the branch and bound tree:

- a branch representing processing no job,
- at least one branch representing the processing of a batch.

3.1.1 Left branching

The leftmost branch of a node represents processing no job. Let v be a node and t be an instant corresponding to node v. t is either the greatest job release date in $joblist_1$ or an instant when a machine becomes idle. By definition of $joblist_1$ and $joblist_2$, the first job in $joblist_2$, say job j, has a release date greater than t. The next instant to be explored by left branching is then r_j. Moreover, when a child node is created by left branching, the job lists are updated as the following: let r_{first} be the release date of the first job in $joblist_2$, $joblist_1 \leftarrow joblist_1 \cup$ job(s) j and $joblist_2 \leftarrow joblist_2$ - job(s) j such that $r_j = r_{first}$. The idea of left branching is to delay a job (or jobs) to put it (or them) in a batch with some other jobs which become available later.

3.1.2 Right branching

The right branches of a node represent the processing of batches. Suppose there is just a single job, say job j with processing time p_j, available at an instant t. Then the right branch at instant t represents the processing of job j. If, however, there is more than one job available at t, then we need to explore every possible instant that can be reached from t. Let k be the number of jobs available at t and let $p_{t_1}, p_{t_2}, ..., p_{t_k}$ be the processing times of these jobs. We thus create k batches each having a processing time equal to $p_{t_1}, p_{t_2}, ..., p_{t_k}$. This way, each batch has its first element that determines the batch processing time. Once the processing time of a batch is determined, a modified version of the longest processing time (LPT) rule, called LPT_{batch}, is applied to fill the remaining available space in each batch. In the LPT rule, jobs are first sorted in non-increasing order of processing times. Then batches are filled in that order. The first $\lceil n/B \rceil - 1$ batches are full and the last batch contains $n - B * (\lceil n/B \rceil - 1)$ jobs. On the other hand, LPT_{batch} creates a single batch respecting the processing time p_{t_l} of the branch. This algorithm sorts jobs in non-increasing order of processing times. Then, starting from the first element of the sorted list, a job is put in the batch if its processing time is smaller or equal to the processing time of the batch, i.e., p_{t_l}. The algorithm stops if the batch capacity is reached or if all jobs are batched. This process is repeated for all different processing times $p_{t_1}, p_{t_2}, ..., p_{t_k}$. If there are several jobs having the same processing time p_{t_l}, tie is arbitrarily broken in the sorting process. Moreover, as will be discussed below, a single branch is formed for the processing time p_{t_l}.

To illustrate the batch creation procedure, suppose there are 3 jobs j_1, j_2 and j_3 available at instant t representing a node v. Let their processing times be p_1, p_2, and p_3 in non-decreasing order (i.e., $p_1 \leq p_2 \leq p_3$). Since all jobs are available at t, their release dates are smaller or equal to t. Suppose that there is just one machine with capacity 2 which is idle at t' (t' can be smaller, greater or equal to t). Then, because we have three jobs with different processing times present simultaneously at t, we create three batches, i.e., 3 child nodes for node v. The first batch has a processing time equal to p_1, the second batch has a processing time equal to p_2 and p_3 is the processing time for the third batch. Because LPT_{batch} is applied to jobs present at node v, the first batch contains only j_1, the second batch contains j_2 and j_1 and the third batch contains jobs j_3 and j_2. Moreover, the processing of these batches gives the following instants for child nodes: $max(t, t') + p_1$, $max(t, t') + p_2$ and $max(t, t') + p_3$. An illustration is given in Figure 1. Note that we do not have a batch with all 3 jobs since the machine capacity is 2.

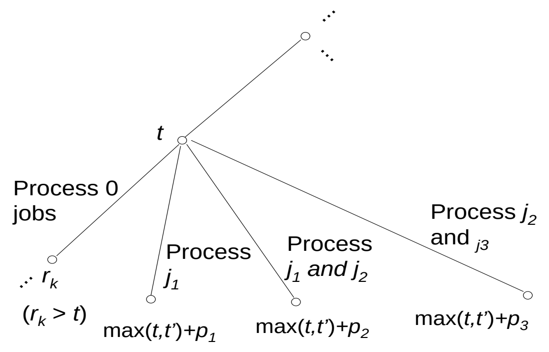


Figure 1. Exploration of instants for a single machine problem with capacity equals to 2

Proposition 1: LPT_{batch} creates optimum batches.

Proof. Because for every available job at instant t a batch is created, a processing time, p_j, is assigned to each batch in which job j becomes the first element of the batch. The remaining space in each batch must be filled with jobs whose processing times are smaller or equal to p_j for two reasons: 1-not to repeat any other right branch(es) by putting in the same batch a job k such that $p_k > p_j$, 2-not to have a longer processing time for a following batch.

Let J_b be the set of jobs put in batch b with LPT_{batch}. Consider the case that the remaining available space in batch b is not filled using LPT_{batch} and let J'_b be the set of jobs in the new batch configuration denoted by b'. Since the first element of each batch is the same in both batches, $p_j \geq p_{j'} \ \forall \ j' \in J'$. Now, consider a job k which is in batch b but not in b'. Because there is no guarantee that jobs with greater processing times are prioritized in batch b' (unlike LPT_{batch}, in which jobs are prioritized), there is an element k' of J'_b put in batch b' such that either $p_{k'} = p_k$ or $p_k > p_{k'}$. If $p_{k'} = p_k$, then subsequent batches are not affected by the alternation of jobs k and k'. However when $p_k > p_{k'}$, the processing time of the subsequent batch b_s containing job k is affected by the alternation of jobs k and k' if job k is the first job of batch b_s. Let p_{s_2} be the second greatest processing time in batch b_s. Then, the length of batch b_s is increased by $p_k - max(p_{k'}, p_{s_2})$. If job k is not the first job in batch b_s, alternating jobs k and k' does not affect the processing time of batch b_s. Hence in each case LPT_{batch} yields a smaller or equal batch completion time compared to any other batch creation procedure and thus provides optimum batches.□

Next we provide a condition that avoids having multiple right branches, i.e., batches having the same processing time.

Proposition 2: If at instant t two or more jobs have the same processing time p_t, then it is sufficient to create a single right branch corresponding to the batch having processing time p_t.

Proof. Creating two or more right branches/batches having equal processing times is redundant since LPT_{batch} would assign the same jobs to those batches. □

If all jobs are released but some jobs are not processed yet, then there is no need for delaying the processing of these jobs to a greater instant. Since delaying is done by left branching, if all jobs are released, only right branching is done to explore new instants. Let r_n denote the greatest job release date and t an instant.

Proposition 3: For all nodes representing an instant t such that $t \geq r_n$, only right branching is done using the LPT rule.

Proof. Since all jobs are available by instant t, there is no more need for delaying jobs. Thus, the LPT rule can be applied to create batches. □

Whenever jobs are delayed, we show by the following proposition that it is not optimal to process them solely with other delayed jobs.

Proposition 4: Let t be an instant reached through left branching following instant t'. If a job j could have been processed at instant t' but was delayed until t, then it is not optimal to process job j alone or with other delayed jobs at t.

Proof. If a job j is delayed that could be processed earlier at instant t', then batching and processing j all alone or with some other delayed jobs at instant t increases the processing ending time of the batch by $t - t'$ compared to processing the same batch at instant t'. □

3.1.3 Updating and using the critical instant cI for further pruning

Recall that phase 1 calculates an initial critical instant cI before which all jobs can be processed without interacting with jobs released later or at cI. Since the same kind of instants (i.e., an instant t at which $joblist_1$ is empty and the first job release date in $joblist_2$ is greater or equal to t) can be encountered more than once during the enumeration process, the critical instant can be updated during the exploration of the branch and bound tree. Thus, if there is an instant t greater than cI such that $jobList_1 = \emptyset$ and $jobList_2 \neq \emptyset$ $(r_j \geq t \ \forall \ j \in jobList_2)$, then $cI \leftarrow t$.

Proposition 5: If during the solution process, a node v' represents an instant t' smaller or equal to cI, then node v' is pruned.

Proof. It is sufficient to explore only a single time child nodes of the node representing instant cI to reach the optimal makespan. Thus if the same instant cI is encountered in another node, there is no need to go further and the node is pruned. Similarly, since exploring only a single time child nodes of the node representing instant cI gives the optimal makespan, if another instant t' smaller than cI is encountered recursively, it is also pruned. □

3.1.4 Generalization to parallel machines

The above branching schemes let us generalize the $B\&B$ to the case of parallel machines. Machine assignment of batches is done in the following way. Once batches are formed according to the above branching rules, we try all possible configurations by assigning the batches to every machine (such an assignment procedure is necessary since jobs have different processing times). More precisely, if a batch b is to be processed and if we have M machines available, batch b is processed on machine 1, then on machine 2, and so on until M. Thus, processing batch b results having M child nodes. We illustrate the assignment procedure in Figure 2. Let v be a node representing instant t at which two non-delayed jobs j_1 and j_2 are ready for processing. Let p_1 and p_2 be the processing time of these jobs ($p_1 < p_2$). There are two machines with capacity two and their idle times are t_m with $m = 1, 2$ (t_m can be smaller, greater or equal to t). The creation of batches and machine assignments are shown as child nodes of v in Figure 2.

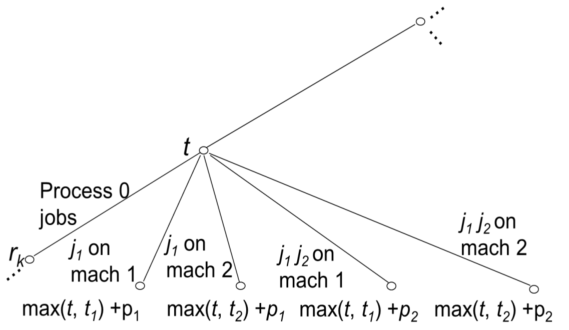


Figure 2. Illustration of batch creation and machine assignment

Proposition 6: If more than one machine becomes idle at the same time, then it is sufficient to assign the batch to a single machine among those becoming idle at the same time.

Proof. Since machines are identical, assigning the batch to only one machine in the case of equal machine idle times prevents equivalent solutions and decreases the search space of the branch and bound tree. □

Because of proposition 6, we implement a tie-breaking rule that always chooses the biggest indexed machine among those machines idle at the same time.

A numerical example and its solution with the proposed branching scheme is presented in Appendix B.

3.2 *Lower bounds*

We develop three approaches for finding a lower bound (LB) which are calculated at each node of the search tree.

$LB1$: The first lower bound calculation uses the minimum machine idle time and the job j that has the greatest completion time value among all unprocessed jobs. Let $disp_{min}$ be the minimum machine idle time ($disp_{min} = min(t_m)$ where t_m is the instant machine m becomes idle). Job j is identified by $argmax(max(disp_{min}, r_j) + p_j)$. We have thus $LB_1 = max(disp_{min}, r_j) + p_j$ which is a lower bound value.

$LB2$: The second lower bound is based on the LPT rule. If all jobs are released at the same time (i.e., $r_j = r \ \forall \ j$), then LPT is optimal for the makespan criterion in the case of a single machine (Pinedo (2012)). If, however, there are several identical machines, then a lower bound is obtained by first applying the LPT rule to all unprocessed jobs, making the sum of all batch processing times, and then dividing the found value by the number of machines.

Let $value_{LPT}$ be the value from applying LPT to all unprocessed jobs as if they were all present at the same moment and were processed on a single machine. Let r_{min} be the smallest release date of jobs among unprocessed jobs. Then, the second lower bound is calculated by the following formula: $LB_2 = max(disp_{min}, r_{min}) + value_{LPT}/M$.

$LB3$: Ozturk et al. (2012) proposed an optimal algorithm for the problem of minimizing C_{max} in the presence of divisible jobs with different sizes, release dates and equal processing times. This problem is a special case of ours since we consider unit job sizes and different processing times. Then a lower bound for our problem can be obtained by first modifying the input of the original problem such that $p_{min} \leftarrow min(p_j) \ \forall \ j$, i.e., selecting the minimum processing time as the processing time of all jobs, and then applying the greedy algorithm proposed by Ozturk et al. (2012).

However, setting all job processing times equal to the smallest one leads to a poor relaxation of the original problem when there is a huge gap between the smallest and biggest processing times. Thus instead of setting a global smallest processing time with $p_{min} \leftarrow min(p_j) \ \forall \ j$, we determine a time window $[r_k, r_l[$, and set $p_{min} \leftarrow min(p_j) \ \forall \ j$ for $r_j \in [r_k, r_l[$. If all the jobs j can be processed with the greedy algorithm of Ozturk et al. (2012) before the release of job l, then the time window is updated as $[r_l, r_{l+1}[$ and a new p_{min} is calculated. Otherwise, the time window is updated as $[r_k, r_{l+1}[$. This way, each time window has its own smallest processing time that leads to a better relaxation. Let LB_3 represent the numerical value of the third LB.

The maximum among previous values is chosen as the lower bound; i.e., $LB = max\{LB_1, LB_2, LB_3\}$. The running time of the algorithms is $O(n)$ for LB_1 and LB_2. The LB_3 is computed in $O(n^2)$ time. At each node of the search tree, all unscheduled jobs are taken into account to calculate a lower bound. Then, if the obtained lower bound value is greater than the best C_{max} value obtained so far, the branch is pruned. The performance of the three lower bound methods is discussed in the numerical tests section.

3.3 *Initialization heuristic*

Initializing the branch and bound procedure with a C_{max} value close to optimal is important to reduce the solution time. In this section, we propose a moving interval heuristic that is used to find an initial solution at the root node of $B\&B$.

The heuristic starts by defining a time window [0, t], then proceeds by creating a single batch with jobs whose release dates are in that time window applying the LPT_{batch} procedure of $B\&B$. Considering the determination of the time window, it begins at instant 0 and its length t is determined by $max(r_k$, first machine idle time) where r_k is the release date of job k and k is a parameter varying between 1 to n. We provide the pseudo code of the initialization heuristic and its complexity in Appendix A.

4. Numerical Tests

In this section, we test our branch and bound procedure on a wide range of instances. For benchmarking, we use the branch and bound algorithm of Sung and Choung (2000) for instances containing only a single machine. Sung and Choung (2000) have also proposed a mixed integer linear model (MILP) which can easily be generalized to parallel machine problems. Thus, a second benchmarking is done implementing the MILP model in CPLEX 12.6.

Job release dates and processing times are generated as indicated in Sung and Choung (2000).

Job processing times are generated using a U[1, 20] distribution. There are three types of ranges for generating job release dates which are U[0, 5], U[0, 20] and U[0, 5*n] where n is the number of jobs in the instance. We test instances containing 10, 15, 20, 25, 30, 50, 75 and 100 jobs. Instances containing 30 jobs or less are called *small instances* while others are called *big instances.* Machine capacity is varied as 3 or 5. For each combination of different release date type, machine capacity and number of jobs, 10 problem instances are generated (thus more than 1200 instances are tested throughout this section). Algorithms are coded in Java and an Intel i5-4570 CPU with 3.20 Ghz. machine is used for all numerical tests. The solution time limit is set to 900 seconds.

4.1 *Benchmarking between branch and bound methods (on a single machine instances)*

Throughout the rest of this section, we denote our method as $B\&B$ and the method of Sung and Choung (2000) as $B\&B_{Lit}$. Tables 1 and 2 show for both methods average number of nodes, average solution time in seconds and number of instances solved within 900 seconds. For instances which are not solved within 900 seconds, we report the gap between $B\&B$ and $B\&B_{Lit}$ using the following formula: $[(Makespan_{B\&B_{Lit}} - Makespan_{B\&B})/Makespan_{B\&B}] * 100$. For the average number of generated nodes, M denotes a number bigger than 100 million. We report also some results considering the performance of the heuristic in the last columns of tables. Column 9 shows the number of times the heuristic finds the optimal or best solution (*best* in case an instance is not optimally solved within 900 seconds by $B\&B$). Column 10 reports the average gap of the heuristic solution for instances that the heuristic cannot find the optimal or best solution. We do not report any results for the MILP model since it is not efficient for single machine instances. As an example, even the simplest instances containing 20 jobs are not optimally solved within 60 seconds with the MILP model and the optimality gap is around 30%.

Both branch and bound methods are sensitive to job release dates. When job release dates are generated from a U[0,5] or U[0,20] distribution, both methods can easily find the optimum solution in less than 1 second for small sizes instances in the presence of a machine with capacity 3. However, $B\&B_{Lit}$ is more sensitive to number of jobs and the machine capacity than $B\&B$. Their algorithm is based on enumerating different job sequences. The optimal makespan value of a job sequence is found with a $O(n * Cap)$ pseudo polynomial dynamic programming algorithm where n is the number of jobs in the sequence and Cap is the machine capacity. Thus, increasing n and Cap decreases the performance of $B\&B_{Lit}$. However, increasing the machine capacity has a positive impact on our method. When the machine has a larger capacity, branches representing batch processing have the possibility to put larger numbers of jobs in a batch. Thus the number of unprocessed jobs decreases more quickly which results having a smaller problem to deal with.

When the number of jobs is greater than 50, the solution time with $B\&B_{Lit}$ increases while $B\&B$ can easily solve the same instances within milliseconds. Table 1 also shows how $B\&B_{Lit}$ is sensitive to machine capacity. When the machine capacity is equal to 5, only instances that $B\&B_{Lit}$ can solve in less than 1 second contain less than or equal to 15 jobs for $r_j \in [0, 5]$ or $r_j \in [0, 20]$. $B\&B$ can solve any instance in less than 1 second for any machine capacity and job release date except for $r_j \in [0, 5 * n]$.

For instances with $r_j \in [0, 5 * n]$, $B\&B$ is again able to find the optimal solution faster than $B\&B_{Lit}$. Nevertheless, there is a decrease in its performance compared to other instances. This is due to having a large range for job release dates in which case left branching, i.e., delaying the processing of jobs, occurs more often in the search tree. Moreover, delaying jobs also results having more right branches which increases the search space to be explored.

The optimality gap of the initialization heuristic is approximately 10% on instances for which it cannot find the optimal or best solution. We also observed that for a small range of release dates, the heuristic performs quite good and finds most of the time the same solution as given by

B&*B*. Otherwise, when the release date are generated on a wide range such as $r_j \in U[0, 5*n]$, the performance of the heuristic decreases due to determining only few instants (i.e., job release date or machine idle time) for batch creation.

To have a better understanding about the efficiency of the cI, same instances were also tested without the cI phase. Numerical results showed that the solution time of optimally solved instances increased by almost 50% when cI is not included in the branch and bound algorithm.

Table 1. Comparison between branch and bound methods for capacity = 5

	B&*B*			$B\&B_{Lit}$			Avg.	Heuristic		LB.
No. of of jobs	Avg. nodes	Avg. time	# solved	Avg. nodes	Avg. time	# solved	gap (in %)	# best	Avg. gap	Avg. gap (root node)
$r_j \in U[0,5]$										
10	6	<1	10	53701	<1	10	0	9	2%	13 %
15	11	<1	10	40106	<1	10	0	9	1%	7 %
20	14	<1	10	135167	5	10	0	5	2%	4%
25	21	<1	10	403493	32	10	0	7	2%	3 %
30	24	<1	10	644455	81	10	0	8	3%	2 %
50	42	<1	10	1952170	>900	0	0	6	5%	1 %
75	70	<1	10	1262440	>900	0	6.3	5	4%	0.2 %
100	99	<1	10	1130567	>900	0	15.9	7	2%	0.1 %
$r_j \in U[0,20]$	10									
10	25	<1	10	80613	<1	10	0	6	7%	17%
15	42	<1	10	62601	<1	10	0	7	6%	21%
20	99	<1	10	507205	9	10	0	3	7%	14%
25	97	<1	10	1685376	58	10	0	2	8%	13 %
30	160	<1	10	4357453	213	8	0	2	4%	9%
50	302	<1	10	3915114	894	1	0	2	5%	4%
75	258	<1	10	3365860	>900	0	8.7	3	12%	1%
100	401	<1	10	2580183	>900	0	21.6	2	13%	1%
$r_j \in U[0,5*n]$										
10	270	<1	10	154885	<1	10	0	3	15%	4%
15	1246	<1	10	M	596	4	0	5	14%	3%
20	3695449	2.5	10	89737022	630	3	0	1	15%	3%
25	M	222	8	76627890	723	2	0	-	9%	1.8%
30	M	360	6	53683209	646	3	0	-	12%	2%
50	M	270	7	17522289	>900	0	0	4	8%	0.6%
75	M	180	8	14114639	>900	0	0.15	1	7%	0.9%
100	M	274	7	12182870	>900	0	0.15	2	8%	0.9%

4.2 *Performances of B&B on parallel machine instances*

To the best of our knowledge, other than the MILP model we use, there is no other exact method to solve the case of parallel machines. As mentioned previously, $B\&B_{Lit}$ of Sung and Choung (2000) is based on evaluating different job sequences with a pseudo polynomial time algorithm such that optimal batches are ordered on a single machine. Their method cannot be generalized to the case of parallel machines using the branching scheme we presented in section 3.1.4 since our *B*&*B* is based on exploring time moments and theirs is based on evaluating job sequences at each node.

We test here the performance of *B*&*B* on instances containing 2, 3 and 4 machines and 10, 20

Table 2. Comparison between branch and bound methods for capacity = 3

	$B\&B$			$B\&B_{Lit}$			Avg.	Heuristic		LB.
No. of of jobs	Avg. nodes	Avg. time	# solved	Avg. nodes	Avg. time	# solved	gap (in %)	# best	Avg. gap	Avg. gap (root node)
$r_j \in U[0,5]$										
10	17	<1	10	144	<1	10	0	7	5%	2%
15	22	<1	10	370	<1	10	0	8	8%	2%
20	24	<1	10	686	<1	10	0	4	2%	1.6%
25	40	<1	10	1365	<1	10	0	6	4%	0.8%
30	39	<1	10	1762	<1	10	0	5	3%	0.8%
50	68	<1	10	6338	1	10	0	4	1%	0.1%
75	104	<1	10	14891	12	10	0	5	4%	0%
100	148	<1	10	33668	64	10	0	3	2%	0%
$r_j \in U[0,20]$										
10	33	<1	10	244	<1	10	0	5	5%	16%
15	44	<1	10	335	<1	10	0	4	4%	10%
20	85	<1	10	988	<1	10	0	2	3%	7%
25	124	<1	10	1436	<1	10	0	6	10%	3%
30	146	<1	10	3165	<1	10	0	7	9%	3%
50	230	<1	10	9522	<1	10	0	4	4%	0.9%
75	432	<1	10	27902	9	10	0	2	7%	0.3%
100	604	<1	10	72079	49	10	0	4	5%	0.2%
$r_j \in U[0,5*n]$										
10	96	<1	10	4180	<1	10	0	3	15%	12 %
15	11130	<1	10	5971092	29	10	0	2	12%	6 %
20	3503878	3.2	10	50900448	533	5	0	3	12%	5 %
25	M	180	9	49412478	>900	2	0	4	9%	6 %
30	M	469	5	M	812	2	0	1	10%	7%
50	M	810	1	M	>900	1	0.7	1	11%	3.5 %
75	M	>900	2	M	>900	2	1.9	-	12%	4 %
100	M	>900	1	M	>900	1	2.8	2	9%	3.8%

and 30 jobs. Batch capacity is varied as 3 and 5. MILP performance is also reported. We observed that first unsolved instances within 900 seconds contain 20 or 30 jobs depending on the job release date type. Thus, we limited the number of jobs to 30. For each release date type, machine capacity, number of machines and number of jobs, we tested 10 instances.

While there is a decrease in the performance of $B\&B$ due to the machine assignment procedure, the results are promising especially compared to the performance of MILP. Figure 3 shows the percentage of times that one method is faster the other one for finding the optimal solution. If classified according to job release dates, $B\&B$ has a better performance for instances of $r_j \in U[0,5]$, i.e., when job release dates are generated in a small range compared to job processing times. This is because when the last job release date is small enough, there are mostly right branches which results having a smaller search tree.

Another performance measure we tested is the number of instances solved within 900 seconds. We observed that for instances of $r_j \in U[0,5]$ $B\&B$ largely dominates MILP. Only instances that cannot be optimally solved contain 30 jobs. For $r_j \in U[0,20]$, both methods perform similarly and 35% of instances are not optimally solved. For $r_j \in U[0,5*n]$, MILP dominates $B\&B$ when the machine capacity is 3. 10% of instances cannot be optimally solved by $B\&B$ while this percentage is 3% for MILP. When the capacity is 5, however, both methods find the optimal solution for every single instance and $B\&B$ is much faster than MILP. The increase in the performance of $B\&B$ is

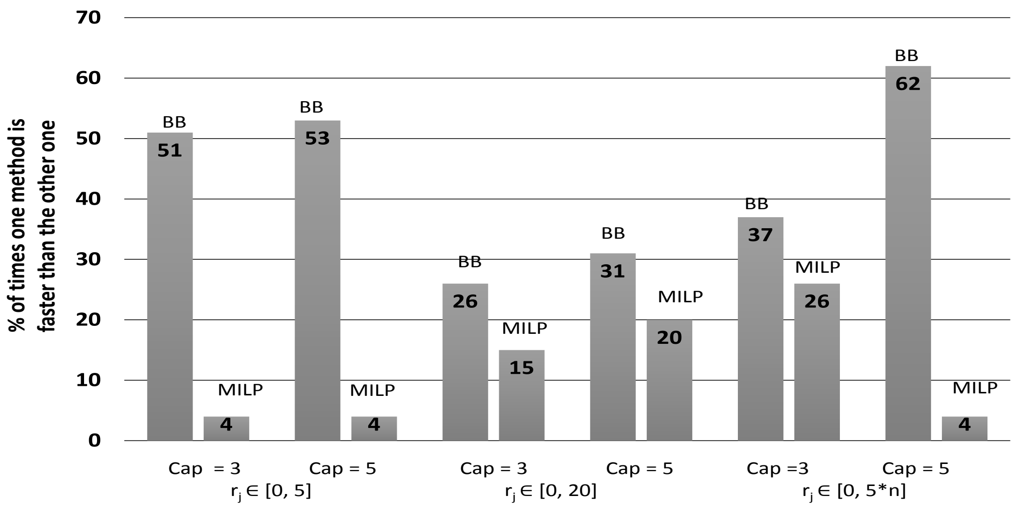


Figure 3. Percentage of time *B&B* dominates MILP model for solution times

due to increasing batch capacity and the use of critical instant, cI. When job release dates are generated in a large range and if there is more than one machine, it is possible to find an initial cI to decrease the size of the problem, and hence the size of the search tree.

4.3 *Quality of the lower bound*

To have a better insight on the quality of lower bound cuts in the solution process, we tested some of the same instances omitting the lower bound method from the algorithm and observed a huge increase in the solution time. For example in single machine instances, the solution time is around 10 seconds for a 35 job instance and 45 seconds on a 40 job instance when $r_j \in U[0, 20]$ or $r_j \in U[0, 5]$ and capacity = 5. The last columns of Tables 1 and 2 represent the average gap obtained with the lower bound algorithm at the root node. Recall that there are three methods that apply to calculate a lower bound. A detailed analyses showed that the individual lower bound quality of those methods vary according to the instance type. When job release dates are in a small range, i.e., $r_j \in U[0, 5]$, method two (based on the longest processing time algorithm) is the dominant one for all tested instances. For $r_j \in U[0, 20]$ LB_1 provides the best lower bound values and for $r_j \in U[0, 5 * n]$ LB_1 and LB_3 are in competition.

A similar observation was done for parallel machine instances. Table 3 shows the average gap between the lower bound and optimal values for the root node. We report also the method that returns the best lower bound value among all (if a method finds the best LB value more than the others, it is reported in the table). Similar to the previous results, LB_2 performs better than the other methods when job release dates are generated in a small range. However, its performance decreases compared to the single machine case. Besides, increasing the number of machines improves the performance of LB_1 and LB_3 since it becomes easier to find an idle machine for batch processing.

An interesting result is an observation on the use of LB_3 for $r_j \in U[0, 5]$ and $r_j \in U[0, 20]$ instances. LB_3 has a complexity of $O(n^2)$ while LB_1 and LB_2 are $O(n)$ and numerical results showed that LB_3 never gave the best lower bound value at the root node unless $r_j \in U[0, 5 * n]$. Since the complexity of LB_3 is higher than the other methods, we omitted LB_3 and tested the same instances with $r_j \in U[0, 5]$ and $r_j \in U[0, 20]$. As expected when the problem has a single machine, the difference in the solution time is not significant since all instances are solved within one second

Table 3. Lower bound performance for the case of parallel machines

	Cap = 3			Cap = 5		
	10 jobs	20 jobs	30 jobs	10 jobs	20 jobs	30 jobs
$r_j \in U[0,5]$						
M = 2	14% - LB2	4% - LB2	5 % - LB2	2 % - LB1	5 % - LB2	7 % - LB2
M = 3	1.2 % - LB1	12 % - LB2	10 % - LB2	0 % - LB1	0 % - LB2	14 % - LB2
M = 4	0 % - LB1	7.5 % - LB1	17 % - LB2	0 % - LB1	0 % - LB1	1 % - LB1
$r_j \in U[0,20]$						
M = 2	3 % - LB1	17 % - LB2	23 % - LB2	1.3% - LB1	6% - LB1	21% - LB1/LB2
M = 3	0 % - LB1	4.2 % - LB1	23 % - LB1	0.4 % - LB1	0.3 % - LB1	3 % - LB1
M = 4	0 % - LB1	0 % - LB1	10 % - LB1	0% - LB1	0 % - LB1	0 % - LB1
$r_j \in U[0,5*n]$						
M = 2	0.5 % - LB1/LB3	0 % - LB1/LB3	0.6 % - LB1	0.3 % - LB1/LB3	0.1 % - LB1	0.2 % - LB1
M = 3	0 % - LB1/LB3	0 % - LB1/LB3	0.1 % - LB1/LB3	0 % - LB1/LB3	0 % - LB1/LB3	0 % - LB1
M = 4	0 % - LB1	0 % - LB1	0% - LB1/LB3	0 % - LB1	0 % - LB1/LB3	0 % - LB1

of computation. Thus, we generated bigger instances containing up to 500 jobs and observed that omitting LB_3 from the algorithm decreases slightly the resolution time (e.g., around 10 seconds for 500 job instances). However, it is the complete opposite for the case of parallel machines. Although LB_3 never gave the best lower bound value at the root node for $r_j \in U[0,5]$ and $r_j \in U[0,20]$ instances, testing the same instances only with LB_1 and LB_2 increased enormously the solution time for the case parallel machines. For example, while 20 job and 2 machine instances are solved very quickly (within a few seconds), when LB_3 is omitted, the solution time is increased more than 100% for all tested instances. The increase in the solution time is due to the number of machines. LB_3 procedure is sensitive to the number of machines and job release dates more than LB_1 and LB_2. When, there is more than one machine, LB_3 assigns a batch always to the machine having the smallest idle time (i.e., time instant when a machine is available to process a batch) and the processing of the batches starts without violating job release dates. The weakness of LB_3 at the root node when job release dates are generated in a small range comes from making all job processing times equal to the smallest one. However, during the solution process, the structure of problem instances changes and better relaxations are obtained in case jobs with small processing times are processed first.

5. Conclusion

In this paper, we studied the parallel batching problem for jobs with different processing times, release dates and unit sizes. The objective is to minimize makespan. Most previous studies focus on heuristic and approximation methods. For our problem, there is a single exact method in the literature which is applied to the case of a single machine. We proposed a branch and bound algorithm (*B&B*) which can handle parallel machine problem instances. We tested its performance on a wide range of instances and compared it to the state of art branch and bound method from the literature for instances containing only a single machine. We found that our method dominates the other one in terms solution time. For instances containing parallel machines, our *B&B* is able to solve moderate size instances within reasonable amount of time.

For future work, this study can be extended by considering jobs with different capacity requirements. Studying other objective functions such as minimization of total job completion times to reduce the amount of inventory would also be a challenging task that could be tackled in future research.

References

Agnetis, A., M.A. Aloulou, L.-L. Fu, and M.Y. Kovalyov. 2015. "Two faster algorithms for coordination of production and batch delivery: A note." *European Journal of Operational Research* 241 (3): 927–930.

Agnetis, A., M. A. Aloulou, and L.-L. Fu. 2014. "Coordination of production and interstage batch delivery with outsourced distribution." *European Journal of Operational Research* 238 (1): 130–142.

Cheng, B., K. Li, and X. Hu. 2015. "Approximation algorithms for two-stage supply chain scheduling of production and distribution." *International Journal of Systems Science: Operations & Logistics* 2 (2): 78–89.

Cheng, B., Y. Yang, and X. Hu. 2016. "Supply chain scheduling with batching, production and distribution." *International Journal of Computer Integrated Manufacturing* 29 (3): 251–262.

Cheng, B.-Y., J.Y.-T. Leung, K. Li, and S.-L. Yang. 2015. "Single batch machine scheduling with deliveries." *Naval Research Logistics (NRL)* 62 (6): 470–482.

Cheraghi, S. H., V. Vishwaram, and Krishnan. K.K. 2003. "Scheduling asingle batch processing machine with disagreeable ready times and due dates." *International Journal of Industrial Engineering* 10 (2): 175–187.

Deng, X., H. Feng, G. Li, and B. Shi. 2005. "A PTAS for Semiconductor Burn-in Scheduling." *J. Comb. Optim.* 9 (1): 5–17.

Dupont, L., and C. Dhaenens-Flipo. 2002. "Minimizing the makespan on a batch processing machine with non-identical job sizes: an exact procedure." *Computers and Operations Research* 29 (7): 807–819.

Graham, R.L., E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. 1979. "Optimization and approximation in deterministic sequencing and scheduling: a survey." *Annals of Discrete Mathematics* 5: 287–326.

Gupta, A.K., and A.I. Sivakumar. 2006. "Optimization of due-date objectives in scheduling semiconductor batch manufacturing." *International Journal of Machine Tools and Manufacture* 46 (12): 1671–1679.

Lee, C.Y., and R. Uzsoy. 1999. "Minimizing Makespan on a Single Batch Processing Machine with Dynamic Job Arrivals." *International Journal of Production Research* 37 (1): 219–236.

Lee, C.-Y., R. Uzsoy, and L. A. Martin-Vega. 1992. "Efficient algorithms for scheduling semiconductor burn-in operations." *Oper. Res.* 40 (4): 764–775.

Li, S., G. Li, and S. Zhang. 2005. "Minimizing makespan with release times on identical parallel batching machines." *Discrete Applied Mathematics* 148 (1): 127 – 134.

Li, S., and J. Yuan. 2010. "Parallel-machine parallel-batching scheduling with family jobs and release dates to minimize makespan." *J. Comb. Optim.* 19 (1): 84–93.

Liu, L. L., C. T. Ng, and T. C. E. Cheng. 2009. "Scheduling jobs with release dates on parallel batch processing machines." *Discrete Appl. Math.* 157 (8): 1825–1830.

Liu, Z., and T. C. E. Cheng. 2005. "Approximation schemes for minimizing total (weighted) completion time with release dates on a batch machine." *Theor. Comput. Sci.* 347 (1-2): 288–298.

Malapert, A., C. Guéret, and L.-M. Rousseau. 2012. "A constraint programming approach for a batch processing problem with non-identical job sizes." *European Journal of Operational Research* 221 (3): 533–545.

Mathirajan, M., and A.I. Sivakumar. 2006. "A literature review, classification and simple meta-analysis on scheduling of batch processors in semiconductor." *International Journal of Advance Manufacturing Technology* 29: 990–1001.

Mönch, L., H. Balasubramanian, J. W. Fowler, and M. E. Pfund. 2005. "Heuristic scheduling of jobs on parallel batch machines with incompatible job families and unequal ready times." *Computers & OR* 32: 2731–2750.

Ozturk, O., M. A. Begen, and G. S. Zaric. 2014. "A branch and bound based heuristic for makespan minimization of washing operations in hospital sterilization services." *European Journal of Operational Research* 239 (1): 214226.

Ozturk, O., M-L. Espinouse, M. Di Mascolo, and A. Gouin. 2012. "Makespan minimisation on parallel batch processing machines with non-identical job sizes and release dates." *International Journal of Production Research* 50 (20).

Parsa, N.R., B. Karimi, and A.H. Kashan. 2010. "A branch and price algorithm to minimize makespan on a single batch processing machine with non-identical job sizes." *Computers and Operations Research* 37 (10): 1720–730.

Parsa, N. R., B. Karimi, and S.M.M. Husseini. 2016. "A neural network based approach to minimize total completion time on a single batch processing machine." *Computers & Industrial Engineering* .

Pinedo, M. 2012. *Scheduling: theory, algorithms, and systems.* Springer.

Potts, C.N., and M.Y. Kovalyov. 2000. "Scheduling with batching : A review.." *European Journal of Operational Research* 120: 228–249.

Sung, C. S., and Y. I. Choung. 2000. "Minimizing makespan on a single burn-in oven in semiconductor manufacturing." *European Journal of Operational Research* 120 (3): 559–574.

Sung, C. S., Y. I. Choung, J. M. Hong, and Y. H. Kim. 2002. "Minimizing makespan on a single burn-in oven with job families and dynamic job arrivals." *Computers & OR* 29 (8): 995–1007.

Tangudu, S.K., and M.E. Kurz. 2006. "A branch and bound algorithm to minimise total weighted tardiness on a single batch processing machine with ready times and incompatible job families." *Production Planning and Control* 17 (10): 728–741.

Uzsoy, R. 1994. "Scheduling a single batch processing machine with non-identical job sizes." *International Journal of Production Research* 32 (7): 1615–1635.

Uzsoy, R. 1995. "Scheduling batch processing machines with incompatible job families." *International Journal of Production Research* 33 (10): 2685–2708.

Yao, S., Z. Jiang, and N. Li. 2012. "A branch and bound algorithm for minimizing total completion time on a single batch machine with incompatible job families and dynamic arrivals." *Computers & Operations Research* 39 (5): 939–951.

Zarook, Y., J. Rezaeian, R. Tavakkoli-Moghaddam, I. Mahdavi, and N. Javadian. 2015. "Minimization of makespan for the single batch-processing machine scheduling problem with considering aging effect and multi-maintenance activities." *The International Journal of Advanced Manufacturing Technology* 76 (9-12): 1879–1892.

Appendix A. Critical Instant Procedure, Branch and Bound Algorithm and Moving Interval Heuristic

Table A1. Notation used in the algorithms

Cap	machine capacity
M	number of machines
m	index of machines ($m = 1, ..., M$)
n	number of jobs
j	index of jobs
r_j	release date of job j
$jobList$	list of jobs
$jobList_j$	j^{th} element of $jobList$
$r_{jobList_j}$	release date of job j in $jobList$
$p_{jobList_j}$	processing time of job j in $jobList$
$dispMach$	list containing machine idle times
$disp_m$	idle time of machine m
$disp_{min}$	smallest machine availability
$disp_{max}$	greatest machine availability
t	an instant in the problem (i.e., a job release date or processing ending time of a batch)
$batch$	set of jobs ready for processing
$pTime$	processing duration of the batch
$rTime$	ready time for processing of the batch
$nbrBatch$	number of batches
$LB(.)$	procedure that returns a numerical value as the lower bound
//	comment line
Procedure1(.)	procedure responsible for right branching for instants smaller than the last job release date
Procedure2(.)	procedure responsible for right branching for instants greater or equal to the last job release date
assign(.)	procedure that returns false if two machines having consecutive indexes have the same idle time
LPT1(.)	a procedure that applies LPT_{batch} to jobs of $jobsList1$ in order to create a single batch such that the processing time is equal to $pTime$. If there are only delayed jobs in that batch, then no job is put into the batch
assign(.)	procedure that helps to find the greatest machine index in case some machines have the same idle time
LPT2(.)	procedure that applies the longest processing time rule to all unscheduled jobs in $jobsList1$
LB(.)	Lower bound calculation
greedy(.)	greedy algorithm of Ozturk et al. (2012) (takes a job list, a processing time and machine list as input, returns minimal makespan value as output)
cI	critical instant as a global variable (initially calculated with Algorithm 2 below)
BB	abbreviation of Branch and Bound (cf. Algorithm 4)

Algorithm 1: Critical instant procedure

```
Input: $jobsList, dispMach$
Output: $integer$
Set critical instant $cI \leftarrow 0$
forall the jobs $j$ in $jobsList$ from 1 to $N-2$ do
    Set $cI \leftarrow r_j$, $p_{max} \leftarrow p_j$
    Generate $newJobList \leftarrow \emptyset$
    forall the jobs $k$ in $jobsList$ from $j+1$ to $N-1$ do
        $newJobList \leftarrow newJobList \cup job\ k-1$
        if $p_k > p_{max}$ then
            $p_{max} \leftarrow p_k$
        end
        if greedy($p_{max}, newJobList, dispMach$) $\leq r_k$ then
            Set $cI \leftarrow r_k$ and $j \leftarrow k-1$
            break;
        end
    end
    if $k == N-1$ and $cI == 0$ then
        return $cI$
    end
end
return $cI$
```

Algorithm 2: Procedure1

```
Input: $jobsListA, jobsListB, dispMach$
Output: void
forall the jobs $j$ in $jobsListA$ do
    $jobsList1 \leftarrow jobsListA$
    $jobsList2 \leftarrow jobsListB$
    $pTime \leftarrow p_{jobList1_j}$
    $rTime \leftarrow r_{jobList1_j}$
    $jobsList1 \leftarrow jobsList1 - batch$
    forall the $m$ from 1 to $M$ do
        if assign($m$, dispMach) = true then
            $dispmachNew \leftarrow dispMach$
            $rTime \leftarrow Max(rTime, dispmachNew_m)$
            $dispmachNew_m \leftarrow rTime + pTime$
            if $LB(jobsList1, jobsList2, dispmachNew < C_{max})$ then
                BB($dispmachNew_{min}$, $jobsList1, jobsList2, dispmachNew$)
            end
        end
    end
end
```

Algorithm 3: Procedure2

```
Algorithm: Procedure2
Input: $jobsListA, jobsListB, dispMach$
Output: $void$
$jobsList1 \leftarrow jobsListA \bigcup jobsListB$
$nbrBatch \leftarrow \lceil$ number of jobs in $jobsList1/Cap\rceil$
while $nbrBatch > 0$ do
    $batch \leftarrow LPT2(jobsList1)$
    $jobsList1 \leftarrow jobsList1 - batch$
    $nbrBatch \leftarrow nbrBatch - 1$
    forall the $m$ from 1 to $M$ do
        if $assign(m, dispmach) = true$ then
            $dispmachNew \leftarrow dispmach$
            $rTime = Max(rTime, dispmachNew_m)$
            $dispmachNew_m \leftarrow rTime + pTime$
            if $LB(jobsList1, jobsList2, dispmachNew < C_{max}$ then
                BB($dispmachNew_{min}$, $jobsList1, jobsList2, dispmachNew$)
            end
        end
    end
end
```

Algorithm 4: Branch and Bound

```
Input: $t, jobsListA, jobsListB, dispMach$;
Output: $void$;
boolean $goFurther \leftarrow true$
if $t < cI$ then
    $goFurther \leftarrow false$
end
else
    if $jobsListA == \emptyset$ and $jobsListB \neq \emptyset$ and $t > cI$ then
        $cI \leftarrow t$
        $goFurther \leftarrow true$
    end
end
if $jobsListA$ and $jobsListB$ are empty then
    if $C_{max} > disp_{max}$ then
        $C_{max} \leftarrow disp_{max}$
    end
end
if $goFurther == true$ and $LB(jobsListA, jobsListB, dispMach) < C_{max}$ then
    $jobsListA \leftarrow jobsListA \cup$ job(s) $j$ //job(s) $j \in jobsListB$ st. $r_j \leq t$// ;
    $jobsListB \leftarrow jobsListB$ - job(s) $j$ //job(s) $j \in jobsListB$ st. $r_j \leq t$// ;
    //Left branching// ;
    if $t < r_n$ then
        BB($r_{jobList_1}$, $jobsListA, jobsListB, dispmach$);
    end
    //Right branching in case there are machines for which idle times are smaller than the last job release date// ;
    if $t < r_n$ then
        Procedure1($jobsListA, jobsListB, dispMach$);
    end
    else
        //Right branching in case no machine idle time is smaller than the last job release date// ;
        Procedure2($jobsListA, jobsListB, dispMach$);
    end
end
```

Algorithm 5: Moving interval heuristic

```
Input: $L_1$list of jobs sorted in increasing order of $r_j$, list of machines
Output: $C_{max}$
initialization
Copy $L_1$ into $L_{temp}$
Set $C_{max} \leftarrow \infty$
forall the  k from 1 to n do
    while $L_1 \neq \emptyset$ do
        if number of jobs in $L_1 < k$ then
            $l \leftarrow$ number of jobs in $L_1$
        end
        else
            $l \leftarrow k$
        end
        Set the length of the time window $t \leftarrow max(r_l$, first machine idle time)
        Apply $LPT_{batch}$ rule to all the jobs $j$ such that $r_j \leq t$ to create a single batch
        Erase from $L_1$ the batched jobs
        Set $r_{max} \leftarrow$ greatest job release date in the batch
        Set $t' \leftarrow max(t, r_{max})$
        Execute the batch on the first idle machine at instant $t'$
    end
    $C_{max} \leftarrow min(C_{max}$, latest machine idle time)
    $L_1 \leftarrow L_{temp}$
end
```

The time complexity of the heuristic is $O(n^3 logn)$ since index k varies from 1 to n and for each different value of k a while loop is executed at most n times. Finally, the dominant step in the while loop is the application of LPT rule having a complexity of $O(nlogn)$.

Appendix B. Numerical example

In this section, we give a numerical example and its solution with the branch and bound algorithm. The instance contains two identical machines idle at instant 0 and three jobs with $r_1 = 0$, $r_2 = 10$, $r_3 = 15$, $p_1 = p_2 = 10$ and $p_3 = 20$. Machine capacity is equal to two jobs.

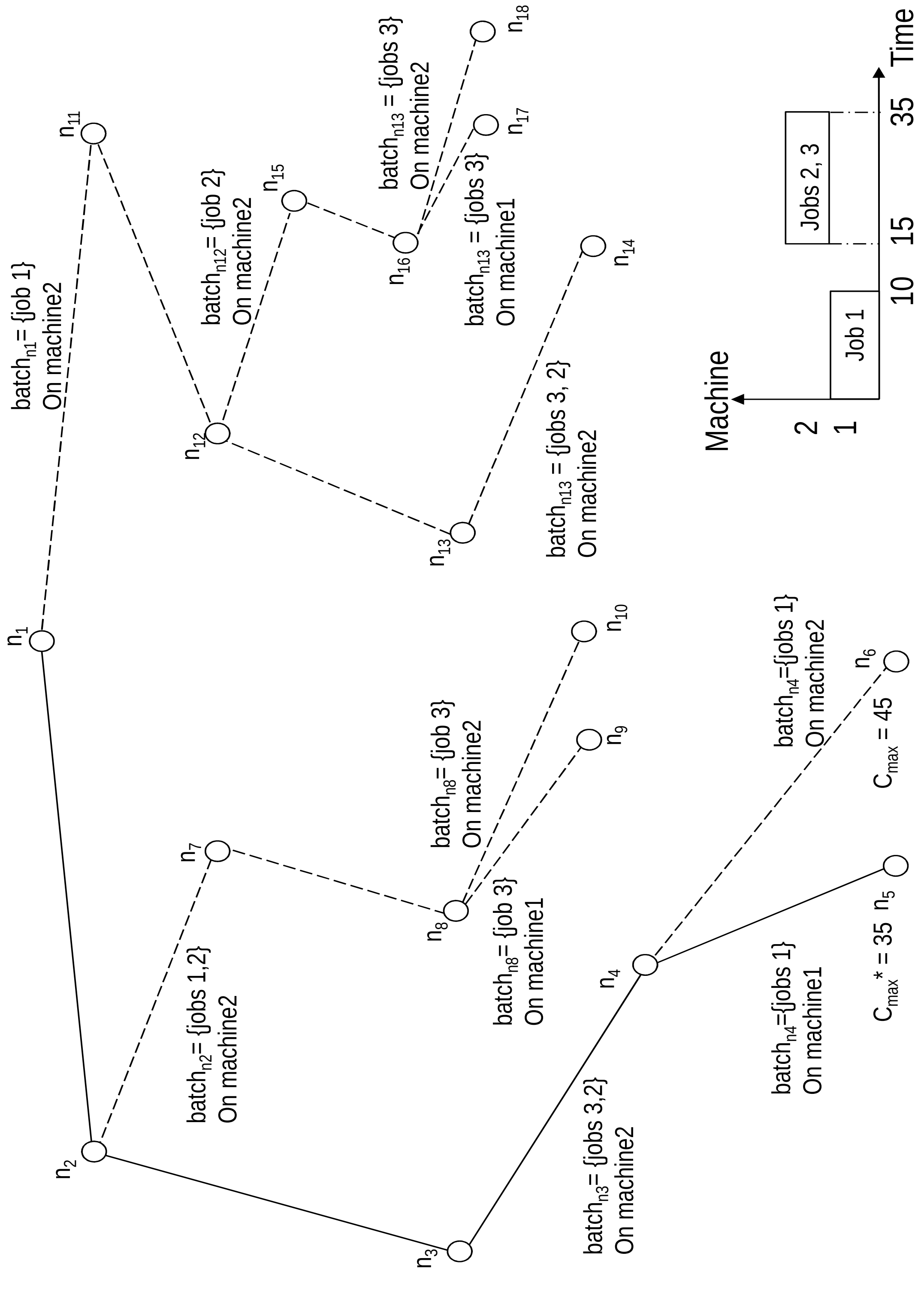
B.0.1 Solution steps for the numerical example

The exploration of the search tree is presented in Figure B1. The dotted branches are normally not created in the tree due to lower bound cuts. However, for demonstration purposes we give the whole tree as if no lower bound cuts are applied.

Nodes n_1, n_2, n_3: The value of the lower bound is equal to 35 at each of these nodes (cf. *LB1*). Thus non of these nodes are pruned. Besides, the algorithm applies depth first search to explore the tree. Thus, jobs are initially delayed until the last job is released. This way, *jobsListB* is emptied and the *LPT* rule is applied to create batches. More precisely, at node n_3 LPT_{batch} is applied and a single batch is created with jobs 3 and 2. Since both machines are idle at instant 0, the batch is assigned to machine 2.

Nodes n_4, n_5, n_6: A single batch containing job 1 is created with LPT_{batch}. Since machine 1 is idle at instant 0 and machine 2 is idle at instant 35, $batch_{n4}$ is assigned to both machines. Node n_4 gives the best makespan value.

Backtracking at node n_2 and right branching: A single right branch is created since both



n_1
n_2
n_3
n_4
n_5
n_6
n_7
n_8
n_9
n_{10}
n_{11}
n_{12}
n_{13}
n_{14}
n_{15}
n_{16}
n_{17}
n_{18}
$batch_{n1}$= {job 1} On machine2
$batch_{n2}$= {jobs 1,2} On machine2
$batch_{n3}$= {jobs 3,2} On machine2
$batch_{n4}$={jobs 1} On machine1
$batch_{n4}$={jobs 1} On machine2
C_{max}* = 35
C_{max} = 45
$batch_{n8}$= {job 3} On machine1
$batch_{n8}$= {job 3} On machine2
$batch_{n12}$= {job 2} On machine2
$batch_{n13}$ = {jobs 3, 2} On machine2
$batch_{n13}$ = {jobs 3} On machine1
$batch_{n13}$ = {jobs 3} On machine2
Machine
Time
2
1
Job 1
Jobs 2, 3
10
15
35

Figure B1. Execution of the branch and bound on the numerical example and the Gantt chart of the optimal solution

machines are idle at 0. The branch stands for the processing of the batch containing jobs 1 and 2.

Node n_7: *jobsListB* contains job 3 and machine 1 is idle at 0. Thus a left branch is generated.

Nodes n_8, n_9, n_{10}: Since machine 1 is idle 0 and machine 2 is idle at 20, batch containing job 3 is assigned to both of them generating nodes n_9 and n_{10}.

Backtracking at node n_1 and right branching: Job 1 is released at instant 0 where both machines are also idle. A batch containing job 1 is assigned to machine 2.

Nodes n_{11}, n_{12}, n_{13}:At node n_{11}, *jobsListB* contains jobs 2 and 3, machine 1 is idle at 0 and machine 2 is idle at 10. Thus job 2 is delayed with depth first search until the release of job 3, i.e., node n_{13}.

Node n_{13}: Job 3 is released at instant 15 and both machines are idle at that instant. A single batch containing jobs 2 and 3 is created and assigned to machine 2.

Backtracking at node n_{12} and right branching: Job 1 is released at instant 10 and both machines are idle at the same instant. A single batch containing job 2 is created and assigned to machine 2.

Node n_{15}: There is only job 3 left in *jobsListB* and machine 1 is idle at 10 which is grater than the release date of job 3. Thus a left branch is generated.

Node n_{16}: Job 3 is released at instant 15, machine 1 is idle at 10 and machine 2 is idle at 20. Thus the batch containing job 3 is assigned to first machine 1 reaching node n_{17} and then to machine 2 reaching node n_{18}.

While the optimal solution is found multiple times in different nodes, node n_5 is the first node that gives the optimal C_{max}. Besides, all other nodes have a lower bound value greater or equal to 35. Thus, when lower bound cuts are included in the algorithm, the dotted lines would not have been explored which decreases substantially the size of the tree.

Role of the critical instant procedure

Job 1 can be processed before the release of job 2. Hence the critical is instant is determined as 10 meaning that jobs 2 and 3 can be treated separately than job 1. Thus, when the cI procedure is included in the solution procedure for the numerical example, the root node becomes n_2 and only nodes n_2, n_3, n_4 and n_5 are explored.

Role of the heuristic procedure

The initialization heuristic finds the makespan value as 35 (job 1 processed at instant 0 on machine 1, job 2 processed at instant 10 on machine 1 and job 3 processed at instant 15 on machine 2). If the heuristic procedure is included in the solution process of the numerical example, non of the branches nor nodes are generated since the lower value is not smaller than the makespan value given by the heuristic.