A branch and bound based heuristic for makespan minimization of washing operations in hospital sterilization services

Onur Ozturk, Mehmet A. Begen*, Gregory S. Zaric

Ivey Business School, Western University, 1255 Western Road London Ontario, Canada N6G 0N1

Abstract

In this paper, we address the problem of parallel batching of jobs on identical machines to minimize makespan. The problem is motivated from the washing step of hospital sterilization services where jobs have different sizes, different release dates and equal processing times. Machines can process more than one job at the same time as long as the total size of jobs in a batch does not exceed the machine capacity. We present a branch and bound based heuristic method and compare it to a linear model and two other heuristics from the literature. Computational experiments show that our method can find high quality solutions within short computation time.

Keywords: OR in health services, parallel batch scheduling, makespan, branch and bound heuristic

1. Introduction

Sterilization services are hospital departments where medical devices (MDs) are sterilized. There are two types of MDs: single use MDs and reusable MDs. Reusable MDs (RMDs) are used in surgeries, sterilized, and then reused in other surgeries. We consider the sterilization process of RMDs in this study.

*corresponding author (tel: +1 519 661 4146)

Email addresses: oozturk@uwo.ca (Onur Ozturk), mbegen@ivey.uwo.ca (Mehmet A. Begen), gzaric@ivey.uwo.ca (Gregory S. Zaric)

All RMDs used in a surgery constitute the RMD set of the surgery. After a surgery, all RMDs used are sent to the sterilization service. Due to surgery characteristics and surgeons needs, RMD sets may contain different numbers and types of instruments. Hence, they may have different sizes (or volumes). Moreover, they are sent to the sterilization service at different times within a day since each surgery may have a different starting and ending time.

A typical sterilization service is composed of the following steps (Di Mascolo and Gouin (2013)): pre-disinfection, washing, packing and sterilization. Pre-disinfection is a manual step during which RMDs are submerged in a chemical substance. Then, they are washed in an automatic washer. Afterwards, they are packed and sterilized with steam in autoclaves.

We are interested in the washing step which is a bottleneck for sterilization services. More than one RMD set can be washed in an automatic washer at the same as long as the machine capacity is not exceeded. All RMD sets washed at the same time constitute a single batch. Depending on the organization between operating theatres and the sterilization service, RMD arrivals can be known in advance. For instance, RMD arrivals can be known accurately for operating theatres where ambulatory surgeries take place. Another example is sterilization services that accept RMD arrivals only at specific times within a day. However, although RMD arrival times and sizes are known in advance, the decision of how to load the machines, i.e., how to batch RMD sets and launch washing cycles is not trivial. We model this problem using a parallel batch scheduling approach. Jobs may have different sizes (or volumes), different release dates and equal processing times. All jobs processed at the same time constitute a single batch which is processed on a single machine. The processing time of batches are the same and equal to the processing time of jobs. Hence, our problem becomes a parallel batching problem where RMD sets are treated as jobs having different sizes, different release dates and equal processing times.

The remainder of this paper is organized as follows. In section 2, we provide a literature review about batch scheduling problems and summarize the contributions of this paper. In section 3, we give a formal description of our problem. Section 4 is dedicated to the solution methodology. Section 5 presents computational tests. Finally, we conclude the study and propose some further research directions.

2. Literature review

2.1. Batch scheduling

We review only batch scheduling literature regarding jobs with different sizes. For more information about batch scheduling, we refer the reader to Potts and Kovalyov (2000) and Mathirajan and Sivakumar (2006). There are two types of batch scheduling: serial and parallel. In serial batch scheduling, jobs in the same batch are processed sequentially on one or more machines. The processing of a batch is completed when the last job of the batch is processed. A typical example is confection workshops where many types of clothes are sewed. For instance, sewing of t-shirts constitutes a batch while shirts, trousers, etc. may constitute a second batch. In parallel batching however, all jobs are processed simultaneously in the same machine. In this paper, we study a parallel batch scheduling problem.

2.1.1. Exact methods

To the best of our knowledge, exact methods for parallel batching with jobs having different processing times are only applied to the case when all jobs are available at the same time. Uzsoy (1994) proposes a branch and bound algorithm to minimize the sum of job completion times on a single machine in which jobs have different processing times and sizes. For the same problem but with the objective of minimizing makespan, Dupont and Dhaenens-Flipo (2002) develop a branch and bound algorithm. Later on, Parsa et al. (2010) propose a branch and price method for the same problem. They report that their method is more efficient in terms of solution time than the one proposed by Dupont and Dhaenens-Flipo (2002). Malapert et al. (2012) study the minimization of maximum lateness on a single machine for which they propose a constraint programming approach. Other than these studies, there are many other studies where the case of unit size jobs is tackled. For instance, Yuan et al. (2004) study the case where jobs have unit sizes but different processing times and release dates in the presence of job families. They provide dynamic programming algorithms when the number of jobs, number of job families and number of release dates are bounded. For the general case, they propose a 2-approximation algorithm. Cheng et al. (2005) propose polynomial time dynamic programming algorithms for a set of regular objective functions when jobs have unit sizes, unit processing times, release dates and precedence constraints in the presence of a single machine.

Regardless of processing times, all problems considering different job sizes are in the class of NP-hard. The additional difficulty in our problem is due to different release dates.

2.1.2. Heuristic and approximation methods

Most studies on batch scheduling with different job sizes focus on heuristic, meta-heuristic methods and approximation algorithms. Zhang et al. (2001) consider the case where jobs are available at the same time while having different sizes and processing times. They develop an approximation algorithm with a worst case performance ratio equal to 7/4 for makespan minimization on a single machine. Cheng et al. (2012) propose an approximation algorithm with a worst case ratio of 2 and (8/3 - 2/3*m) for makespan and total completion time criteria, respectively, in the presence of m identical machines. Li et al. (2005) extend the problem studied in Zhang et al. (2001) by considering job release dates. They present a 2+ϵ approximation algorithm which is derived from a polynomial time approximation scheme that they propose for the case where jobs have unit sizes. Lu et al. (2010) use a similar approach and provide a 2+ϵ approximation algorithm for bi-objective minimization of makespan and penalization of unscheduled jobs. Liu et al. (2014) present heuristics and approximation algorithms for makespan minimization in the presence of unit size jobs with release dates and different processing times. Their work is later generalized to the case of different job sizes by Li (2012). Chou (2007) studies the same problem as in Li et al. (2005) and proposes a genetic algorithm using a dynamic programming procedure to find the makespan of a given chromosome.

Because in our problem we have release dates, different job sizes and parallel machines, the articles cited in this paragraph are more related to our problem. Li (2012) presents the only approximation algorithm with a worst case performance ratio equal to 2+ϵ when jobs have different sizes, different processing times, release dates. There are, however, mostly heuristic/meta-heuristic methods in the literature for the batch scheduling problem studied by Li (2012). For the same problem, Chung et al. (2009) propose a mixed integer linear programming model (MILP) and heuristics. Many other authors use the heuristics of Chung et al. (2009) for benchmarking. Wang and Chou (2010), Damodaran and Velez Gallego (2010) and Damodaran et al. (2011) consider the same problem for which they develop a genetic algorithm, a greedy randomized adaptive search procedure (GRASP) meta-heuristic and a constructive heuristic, respectively. All report that their approaches out-

perform the heuristics proposed in Chung et al. (2009). In another work, Damodaran and Velez-Gallego (2012) propose a simulated annealing algorithm which is able compete with the GRASP approach. Ozturk et al. (2012) develop a MILP model that runs faster than that proposed by Chung et al. (2009) for the case with equal job processing times. They also treat some special cases and provide optimal greedy algorithms. Recently, Pearn et al. (2013) enlarge the broad of the problem considering job families, due dates and set-up times between the processing of batches from different families.

2.2. Contribution of this paper

The method we propose exploits the structural properties of the problem under study. It is based on constructing a search tree where each node represents a job release date or the starting time of batch processing thanks to equal job processing time property. Numerical tests show that our branch bound based heuristic method ($B\&B_H$) can solve problem instances containing up to 40 jobs in short computational time and can solve larger instances in reasonable time. MILP model of Ozturk et al. (2012) can find the optimal solution for small and medium size instances but it requires too much computational time. Regarding other methods from the literature, benchmarking results show that our method's solution quality is higher than two others heuristics from the literature. Our method is applicable in sterilization services since it can quickly solve real size instances.

3. Problem description, notation and complexity

We begin with definitions and notation:

- There are m identical parallel machines with a limited capacity B.
- There are n jobs to be processed. A job is a task that is characterized by a release date, r_j, a size, w_j, and a processing time, p.
- The size of a job cannot be greater than the machine capacity.
- Since washing times are the same for all RMD sets, job processing times are the same for all jobs.
- A batch is composed of jobs processed at the same time on the same machine. Several jobs can be batched together, complying with the machine capacity constraint.

- Once the processing of a batch is started, it cannot be interrupted (i.e. pre-emption is not allowed). Jobs cannot be split into multiple batches.
- The objective is to minimize makespan.

Using the Graham's notation (Graham et al. (1979)), we have a $P|p-batch, r_j, p_j = p, w_j, B|C_{max}$ scheduling problem. In this notation, P stands for identical machines. $p-batch$ indicates that we have a parallel batching problem where all jobs in the same batch are processed at the same time. r_j and w_j stands for job release dates and sizes, respectively. $p_j = p$ indicates that all job processing times are equal to p. B is the machine capacity. Finally, C_{max} is the objective function.

It is straightforward to show that this problem is NP-hard. Consider the special case where all jobs are simultaneously available at instant 0 ($1|p-batch, r_j = 0, p_j = p, w_j, B|C_{max}$). Then, minimizing makespan is equivalent to minimizing the number of batches, which is a bin-packing problem. Since bin-packing is strongly NP-hard, our problem is also strongly NP-hard.

4. Solution methodology

In this section, we present first a lower bound algorithm and then a branch and bound based heuristic for the problem of makespan minimization. The lower bound algorithm will be used for pruning in the branch and bound method. Throughout this section, without loss of generality, we suppose that jobs are sorted in non-decreasing order of release dates.

4.1. Lower bound: LB

The idea of the lower bound algorithm consists in splitting jobs in size and creating batches with consecutive jobs. When a job, say job j, is split in size, two new jobs j_1 and j_2 are obtained such that sum of their sizes is equal to the size of job j. Moreover, release dates of jobs j_1 and j_2 are equal to the release date of job j. Obviously, a lower bound on the number of batches is also obtained when jobs are allowed to be split in size. If after assigning a job to a batch, the number of batches to be created with the remaining jobs decreases, then this batch can be processed immediately. Because there is at least a batch whose processing starting time is equal to the release date, r_j, of the last job, j, it contains, and a minimum number of batches is created after job j with the remaining jobs, therefore a lower bound on C_{max} is obtained.

The lower bound algorithm takes the following steps:
1- Calculate a lower bound on the number of batches by allowing jobs to be split. Select the first unbatched job (i.e., the unbatched job having the smallest release date), put it in a batch, then recalculate the minimum number of batches with the remaining unbatched jobs.
2- If the number of batches to create decreases, close the batch.
3- In case that job does not completely enter the open batch, split the job, put its first part to the batch in order to have a 100% full batch and close the batch. Treat the second part of the job as a new job having the same release date.
4- Execute the same steps with the remaining jobs.

Here closing a batch means that the batch is ready for processing and no other job is put in that batch. The notation used and the lower bound algorithm can be found in the appendix.

The LB algorithm finds the minimum number of batches by finding the sum of all unbatched job sizes and dividing this sum by the machine capacity. Then, this value (if fractional) is rounded up to the smallest integer. To illustrate with a numerical example, Figure 1 shows the release dates and sizes of 4 jobs. Let p be 60 and consider two machines whose capacities are equal to 12.

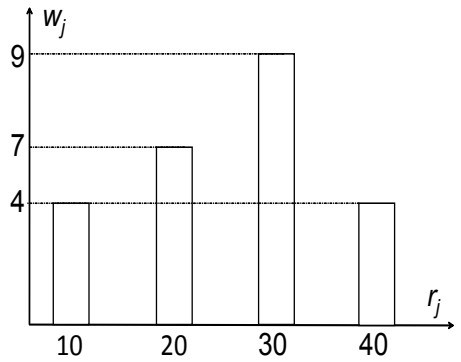

Figure 1: Numerical example

The minimum number of batches is equal to $\lceil (4+7+9+4)/12 \rceil = 2$. When the minimum number of batches is recalculated after placing the first job in a batch, we obtain $\lceil (7+9+4)/12 \rceil = 2$. Thus, the first batch is not

closed yet. The second job is also put to batch 1. The minimum number of batches with the remaining jobs is equal to $\lceil (9+4)/12 \rceil = 2$. Batch 1 stays open. The third job is put in batch 1 but because of the capacity limitation, it cannot completely be placed in batch 1. Thus, job 3 is split such that the size of the first split part is 1 and the second part's size is 8. The first part of job 3 is put in batch 1. The second part of job 3 is treated as new job having the same release date as job 3. Finally, the same procedure is applied to remaining jobs. Once all jobs are assigned to a batch, batch ready times are set equal to the greatest job release date they contain. Then, they are assigned consecutively on machines. The solution is shown on a Gantt diagram in Figure 2.

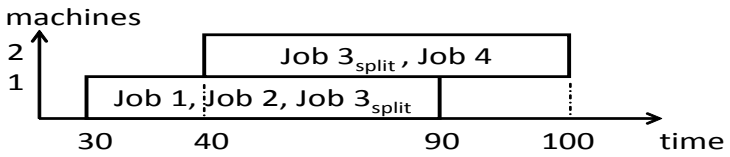

Figure 2: Solution of the numerical example with LB algorithm

4.2. Branch and Bound based heuristic algorithm: B&B$_H$

Because jobs have equal processing times, assignment of batches to machines is an easy task in the presence of identical machines. When a batch is to be processed, it is assigned to the machine having the smallest idle time. Here, *smallest idle time* (or *smallest machine idle time*) indicates the smallest instant when a machine becomes available to process new jobs. For instance consider the solution given in Figure 2. There are two machines such that machine M_1 terminates the processing of some jobs at instant 90 and machine M_2 terminates at instant 100. Then the smallest machine time for this example is instant 90 after which machine M_1 is available to process new jobs. Because jobs have equal processing times, we have a limited number of starting times for the processing of batches due to equal processing times (Baptiste (2000)). Let π be the set of all possible starting times for batches. Then, $\pi = \{r_i + k * p | i \in \{1, ..., n\}$ and $k \in \{0, ..., n\}\}$, $|\pi| = O(n^2)$. In our branch and bound tree, each node is characterized by an instant, say t, which is an element of set π, and by two sets of jobs representing present but unprocessed jobs at t and jobs which have not been released by t, respectively.

4.2.1. Enumerating batch processing instants

The algorithm explores all possible instants by creating a binary tree. Left branch of the tree represents delaying the processing of jobs until the release of next job. Right branch represents the processing of a batch (we talk about the batch creation procedure in the next section).

Left branching

Consider an instant t at which some jobs are available. A node, say v, in the search tree represents instant t as well as available (or released) but unprocessed and unavailable (or unreleased) jobs at t. Let us denote released but unprocessed jobs by $Jobs_A$ and unreleased jobs by $Jobs_{UA}$. Left branch leads to a child node, say v_l, by delaying the processing of jobs in $Jobs_A$ until the release of first job in $Jobs_{UA}$. Job j_{first} denoting the first job in $Jobs_{UA}$, v_l is characterized by an instant t_l such that $t_l \leftarrow r_{first}$ where r_{first} is the release date of job j_{first}, and sets $Jobs_{UA_l}$ and $Jobs_{A_l}$ such that $Jobs_{A_l} \leftarrow Jobs_A \cup$ job(s) j and $Jobs_{UA_l} \leftarrow Jobs_{UA}$ - job(s) j for $r_j = r_{first}$. For instance, let $Jobs_A = \{j_1\}$ available at t_1 and $Jobs_{UA} = \{j_2, j_3\}$ available at t_2 such that $t_1 < t_2$ and $r_2 = r_3 = t_2$. Then, $t_l = t_2$, $Jobs_{A_l} = \{j_1, j_2, j_3\}$ and $Jobs_{UA_l} = \{\}$.

Right branching

Regarding the right branch, a batch is created with jobs present in set $Jobs_A$. Let $jobs_{batch}$ represent the jobs put in the batch (we explain the batch creation procedure in section 4.2.4). After right branching, i.e., after processing jobs $jobs_{batch}$, $Jobs_A$ is updated as follows: $Jobs_A \leftarrow Jobs_A - jobs_{batch}$. For the processing starting time of the batch, batch ready time and the smallest machine idle time are taken into account. Instant r_{batch} representing the ready time of the batch and $disp_{min}$ the minimum machine idle time, the processing starting time of batch is $max(r_{batch}, disp_{min})$.

Exploring new instants after right branching

The idea of exploring new instants after right branching is based on finding the smallest instant t_s which will allow processing jobs. After right branching, if there are still unprocessed jobs in $Jobs_A$, then the next interesting instant is the smallest machine idle time, i.e., $t_s = disp_{min}$. If, however, $Jobs_A$ becomes empty after right branching, then the next interesting instant is the maximum between the smallest machine idle time and the first job release date r_{first} in $Jobs_{UA}$, i.e., $t_s = max(disp_{min}, r_{first})$. Once t_s is determined, left and right branchings reoccur to explore new instants.

Updating job lists after right branching

Let t_r be the next instant to be explored after right branching at t. $Jobs_A$ is updated by erasing jobs processed at t. Since new jobs may be released at instant t_r, $Jobs_A$ must contain these jobs. $jobs_{t_r}$ denoting the job(s) released earlier or at t_r but not processed by t_r, job lists are updated as follows: $Jobs_A \leftarrow Jobs_A \cup jobs_{t_r}$ and $Jobs_{UA} \leftarrow Jobs_{UA} - jobs_{t_r}$.

4.2.2. Branching scheme

A natural branching scheme would be depth first search by selecting always left branches first. However, this type of search may increase the solution time and space since the first right branching is done when all jobs are available, i.e., at the final node discovered by left branching. Instead, we develop a preprocessing method that calculates the lower bound value for each child node. More precisely, before any child node is visited, we calculate the value of lower bound for each child node reached by left and right branches. Then, child node having the smaller lower bound value is prioritized. In case lower bound values are equal, tie is broken by choosing the left branch.

4.2.3. Cuts

We add cuts to improve the solution time of the algorithm. The first cut is done using the lower bound algorithm.

Proposition 1. If at any node, the lower bound value is greater than the best makespan value obtained so far, that node is pruned.

Proposition 2. If a node represents an instant greater than r_n (release date of the last job), then left branching is no longer necessary.

Proof. Since r_n is the last job release date, there is no other job released after r_n. Then, unprocessed jobs by r_n (or at an instant greater than r_n) can be processed without being delayed, i.e., without left branching.□

Proposition 3. Let t be an instant associated with node v at which some jobs are available for processing. If t is smaller than r_n and if the next job release date is greater or equal to $t + p$, then there is no left branching at v.

Proof. Since the processing time of a batch is p, delaying the jobs available at t, until $t+p$ results in unnecessary waiting and hence, solely right branching

at node v is sufficient. A batch is thus created with jobs available at v. $\square$

4.2.4. Batch creation

We use a suboptimal procedure to create batches. The idea is to maximize the used batch capacity which reduces the batch creation step to a knapsack problem. Note that a similar approach is presented as an approximation algorithm in the bin-packing literature. While there are jobs to be put in a bin, the algorithm solves a knapsack problem until there is no job left. Gupta and Ho (1999) test this method on randomly generated instances and report that it performs much better than other heuristic methods in the bin packing literature. They also show that the algorithm guarantees the optimal solution if the sum of item sizes is at most equal to twice of bin capacity. Caprara and Pferschy (2004) show that the worst case performance ratio of the method is bounded by $4/3 + \ln 4/3 \approx 1.6210$. A detailed performance analysis of this approach and other bin-packing algorithms can be found in Vanderbeck (1999).

Although there is a tight relation between our problem and bin packing problem, we should point out that minimizing number of batches does not guarantee the optimality of makespan for our problem. Consider the following example for which job sizes and release dates are given in Table 1. Consider two machines with capacity B. Let p be the processing time and ϵ_j a number smaller than $B/2$ $\forall j$.

Table 1: Bin packing vs. Scheduling

Job	1	2	3	4	5	6
Release date	p	$2p$	$3p$	$4p$	$4p$	$4p$
Size	$B/2+\epsilon_1$	$B/2+\epsilon_2$	$B/2+\epsilon_3$	$B/2-\epsilon_1$	$B/2-\epsilon_2$	$B/2-\epsilon_3$

If the objective is minimizing the number of batches, 3 full batches can be created and the processing of batches starts at $4p$. Makespan value is thus $6p$. However, it is easy to see that the optimal makespan is $5p$ which can be achieved by creating 5 batches. Figure 3 shows both solutions. Hence, the structure of our problem allows to have optimal makespan without minimizing the number of batches. However, maximizing the used batch capacity is a natural approach when a batch is created. We strengthen this approach by applying a dominance criterion proposed by Martello and Toth (1990).

Dominance criterion: (Martello and Toth (1990)) Let f_1 and f_2 be two feasible partitions of jobs such that $\sum_{j\in f_1} v_j = \sum_{j\in f_2} v_j \leq Cap$. f_1 dominates f_2 if the cardinality of f_1 is smaller than the cardinality of f_2.

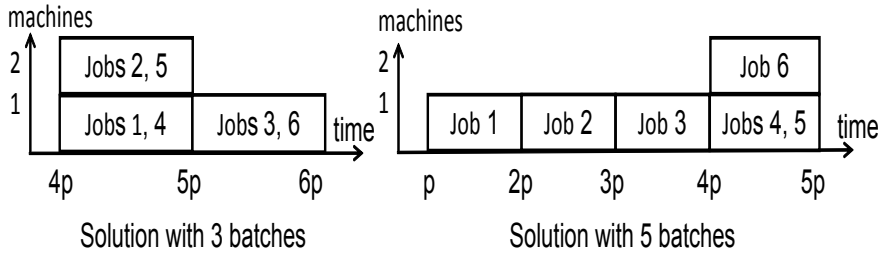

Figure 3: Two solutions for the example presented in Table 1

The dominance criterion above prioritizes big size jobs for batch creation. This way, if there are other big size jobs released later and if these jobs cannot be batched with earlier big size jobs, the idea of the dominance criterion is to leave smaller size jobs to later instants. For that purpose, a binary search procedure is used to create a single batch. If there are many batch configurations, binary search procedure chooses the one containing the least number of jobs. This procedure, named ($createBatch(.)$), is presented in the appendix.

4.2.5. Initialization: Upper bound heuristic

We use a heuristic to find an upper value on makespan at the root node. This heuristic creates batches with consecutive jobs. If a job cannot be placed in a batch because of capacity limitations, batch is closed and assigned to the machine having the smallest idle time.

4.2.6. Numerical example

Consider the example given in Figure 1. Figure 4 shows the search tree for the example. Numbers in nodes represent the order of visiting nodes in the search tree. Lower bound values associated with each child node is represented next to branches. We explain below the solution procedure step by step.

Solution steps for the numerical example

Heuristic finds an upper bound value equal to 140. The initial lower bound is 100.

Node 1 (root node): Left branch represents delaying the processing of job

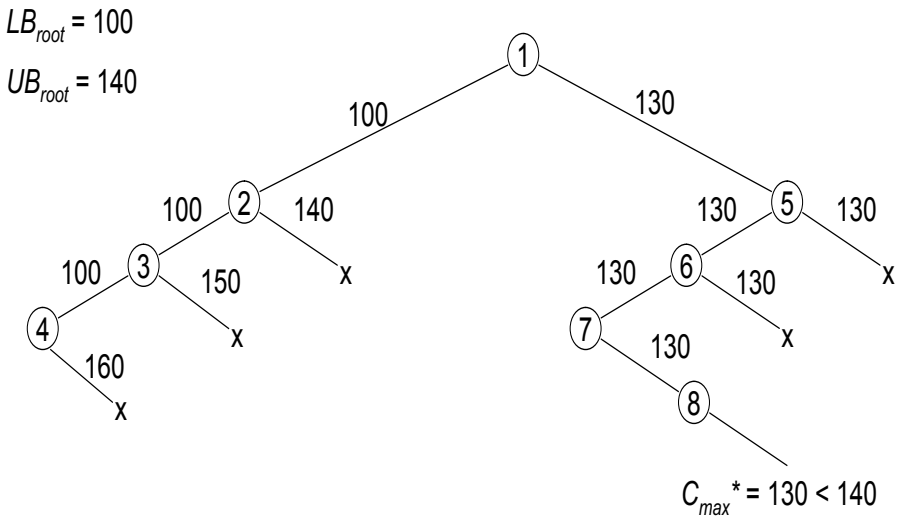

Figure 4: The search tree for the example

1 until the release of second job which provides a lower bound value smaller than processing job 1 immediately upon its release. Thus, left branching is prioritized.

Nodes 2, 3 and 4: Delaying the processing of jobs 1,2 and 3 until the release of job 4 gives a lower bound value smaller than right branching at nodes 2 and 3. Thus, left child nodes are visited first. At node 4, all jobs are available. Right branching at node 4 represents the processing of jobs 1 and 2 in the same batch at instant 40. But the lower bound value of the child node is 160. This branch is thus pruned.

Backtracking at Node 3: Left branch at node 3 processes jobs 1 and 2 in the same batch at instant 30. But with the remaining jobs, at least two other batches are created and thus the lower bound associated with that branch becomes 150. It is thus pruned.

Backtracking at Node 2: Left branch at node 2 processes jobs 1 and 2 in the same batch at instant 20. Since the lower bound associated with the right child node is equal to 140, i.e. current best makespan value, right branch is pruned.

Backtracking at Node 1 and right branching: Job 1 is processed on machine 1 at instant 10.

Node 5: Left branch has a lower bound value equal to that of right branch at node 5. Job 2, which is available at node 5, is thus delayed.

Node 6: Left and right branches has equal lower bound value. Tie is broken by choosing left branch.
Node 7: All unprocessed jobs are available at node 7. Job 1 has already been processed. Machine 1 is idle at 70 and machine two is idle at 0. Jobs 2 and 4 are put is the same batch and processed at instant 40 on machine 2.
Node 8: Finally, job 3 is processed on machine 1 which gives a makespan value equal to 130 (130 becomes the new best C_{max} value).
Backtracking at nodes 6 and 5: The lower bound values associated with the right branches of nodes 6 and 5 are greater or equal to 130. These branches are thus pruned.

Figure 5 shows the Gantt Diagram corresponding to the optimal solution.

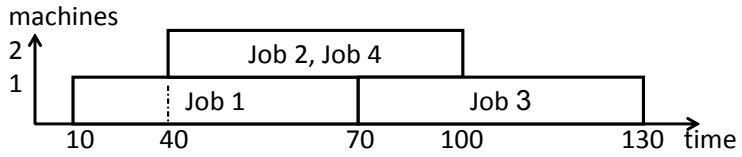

Figure 5: Solution of the numerical example with branch and bound

Pseudo-code of the algorithm is given in appendix.

4.2.7. Optimally solvable cases

Ikura and Gimple (1986) studied a special case of our problem where jobs have unit sizes and presented a polynomial time algorithm. It is straightforward to see that $B\&B_H$ guarantees the optimal solution if all jobs have the same size in a problem instance since batch creation becomes easy. We now present another special case that $B\&B_H$ can guarantee optimal solution. Suppose for any time interval of length of p/m, sum of job sizes is smaller or equal to machine capacity for jobs whose release dates are in that time interval. We first show that in this special case makespan value is equal to $r_n + p$ and then argue that $B\&B_H$ finds optimal solution.

Property 1. Suppose for any a time interval $[r_j, r_j + p/m]$ the sum of job sizes is smaller or equal to the machine capacity, i.e. $\sum w_k \leq B \ \forall k$ such that $r_k \in [r_j, r_j + p/m] \ \forall j$. Then, the optimum makespan value is equal to $r_n + p$ where r_n is the last job release date.

Proof. Let K be an integer such that $r_1 \in [r_n - K*p/m, r_n - (K-1)*p/m]$. Then, we have K intervals of length p/m which allows us to create K batches.

Considering the first batch is processed at instant $r_n-(K-1)*p/m$, a second batch can be created and processed at $r_n-(K-2)*p/m$ on machine 2. Then, the second batch on machine 1 is created with jobs whose release dates are in $[r_n-(K-m)*p/m, r_n-(K-(m+1))*p/m]$ and processed at most at instant $r_n-(K-(m+1))*p/m$. Observe that $r_n-(K-(m+1))*p/m$ is equal to the processing ending time of the first batch on machine 1, i.e. $r_n-(K-1)*p/m+p$. Similarly, jobs having release dates in the interval $[r_n-p/m, r_n]$ are processed at r_n which concludes the proof. □

In this special case, since $B\&B_H$ explores instants at which only a single batch can be created, batch creation is no longer a difficult task and thus $B\&B_H$ gives the optimal makespan thanks to exploring every possible instant in the problem.

5. Computational experiments

In this section, we test two types of problem instances which are inspired from the hospital sterilization context. Algorithms are coded in Java and implemented on an Intel Core i5 2.50 GHz machine. Solution time limit is set to one hour. We use the genetic algorithm of Wang and Chou (2010) noted GA_{Lit}, approximation algorithm of Li (2012) noted AA_{Lit} and the MILP model of Ozturk et al. (2012) noted $MILP_{Lit}$ for benchmarking.

$MILP_{Lit}$ can guarantee the optimal solution once a problem instance is completely solved. GA_{Lit} is a powerful meta-heuristic which provides slightly better results in terms of solution quality and solution time compared to other meta-heuristics from the literature. Briefly, GA_{Lit} generates chromosomes at each iteration by randomization and immigration, and then performs a two point cross over between chromosomes chosen with the roulette wheel technique. AA_{Lit} solves the problem initially by allowing jobs to be split. And then split jobs are scheduled by being assigned individually to a batch following the last batch in the lower bound solution. To the best of our knowledge, the performance of approximation algorithms in the batch scheduling literature, including the one proposed by Li (2012), have not been tested on numerical instances. It would be thus interesting to observe how AA_{Lit} performs on our instances. For all these reasons, we choose these three methods for benchmarking.

Before proceeding with the testing of instances inspired from the sterilization context, we tested $B\&B_H$ on many small instances which can be solved quickly by $MILP_{Lit}$. This way, we compared makespan values found

by $B\&B_H$ to optimal solutions given by $MILP_{Lit}$. For that purpose, we generated 20000 test instances containing 6 to 10 jobs in the presence of 1 to 4 machines (1000 problem instances are generated for each combination of number of jobs and number of machines). Job sizes are generated from a discrete uniform distribution: U[1, 120]. $p = 60$ minutes being the job processing time, $r_1 = 0$ being the first job release date in any instance, job release dates are generated using the following formula: $r_j = r_{j-1} + U[0, 30]$ $\forall j = 2, ..., n$. Among 20000 instances, only 34 of them could not be solved optimally by $B\&B_H$. This observation encouraged us about the solution quality of $B\&B_H$. Thus we proceeded with a detailed analysis of $B\&B_H$ by testing real life instances.

5.1. First instance type: Irregular job arrivals

In some hospitals, RMD sets are sent to the sterilization service just after the end of a surgery. Thus, RMD arrivals can happen at any time within a day. An example of this kind of organization can be found at the sterilization service of the ambulatory surgery department of University Hospital Gasthuisberg in Leuven, Belgium.

Supposing there are surgeries at operating blocks during 8 to 10 hours per day, job release dates are created according to the uniform distribution U[0, 600] (unit in minutes). The size of automatic washers can be between 6 and 12 din. (Din is a measurement unit for the volume of automatic washers which is about 0.003 m^3.) Fixing the machine capacity to 12 din, we create job sizes according to a continuous uniform distribution such that $w_j = U_c]0, 12]$ since any RMD set size smaller than machine capacity is possible. A washing cycle is 60 minutes. Number of machines is varied from 1 to 4. For each combination of number of jobs and number of machines, 20 problem instances are generated and solved through this section.

5.1.1. Medium size instances

Medium size instances contain 10 to 40 jobs. For all instances tested in Table 2, $B\&B_H$ is able to give the best solution each time except for one. Let us detail our analysis by providing more insights about the quality of solution provided by $B\&B_H$ and other methods. In Table 2, column $\#NB$ (not best) indicates the number of times a method does not provide the best solution. *Avg. gap* shows the average of gap for instances whose makespan value is not equal to the best one. The formula used for *Avg. gap* is the following: $(Solution_{method} - best\ solution)$ / *best solution*.

Table 2: Benchmarking results on medium size instances for irregular arrivals

		$B\&B_H$		$MILP_{Lit}$		GA_{Lit}		AA_{Lit}	
No. mach.	No. Jobs	# NB	$Avg.$ gap	# NB	$Avg.$ gap	# NB	$Avg.$ gap	# NB	$Avg.$ gap
	10	-	-	-	-	-	-	1	14%
1	20	-	-	-	-	-	-	2	10%
	30	1	≈ 0	-	-	4	5%	5	12%
	40	-	-	7	5%	5	6%	5	12%
	10	-	-	-	-	-	-	1	10%
2	20	-	-	-	-	-	-	1	4%
	30	-	-	-	-	4	2%	5	6%
	40	-	-	6	4%	3	4%	3	10%
	10	-	-	-	-	-	-	-	-
3	20	-	-	-	-	-	-	-	-
	30	-	-	-	-	-	-	1	≈ 0
	40	-	-	7	1%	-	-	1	≈ 0
	10	-	-	-	-	-	-	-	-
4	20	-	-	-	-	-	-	-	-
	30	-	-	-	-	-	-	1	≈ 0
	40	-	-	2	1%	-	-	2	≈ 0

For instances containing more than 10 jobs, MILP cannot find the optimal solution within one hour. Nevertheless, it can provide the best solution for all instances containing 10, 20 and 30 jobs (The optimality gap reported by CPLEX is around 10% for instances with 30 jobs at the end of one hour). Starting from 40 job instances, performance of $MILP_{Lit}$ decreases. Regarding GA_{Lit} and AA_{Lit}, their performances increase with the increasing number of machines since it becomes easier to find an idle machine for the processing of a batch. In the presence of one and two machines, solution quality of these heuristics are not satisfactory. However, their solution times are faster. AA_{Lit} can find a solution within some milliseconds. The maximum solution time with GA_{Lit} is 10 seconds. A detailed presentation of average solution times for all irregular type instances is given in Table 5.

$B\&B_H$ can quickly find a solution for instances containing up to 40 jobs in the presence of a single machine. The branching scheme and quality of upper and lower bound algorithms play an important role for the solution time. Table 3 shows the quality of the lower bound algorithm and the initialization heuristic as well as the number of nodes created by $B\&B_H$ and the average solution times in seconds. Columns 4 and 5 show the average and maximum gaps between lower bound and $B\&B_H$ which is calculated as ($Solution_{B\&B_H}-$

Table 3: Solution limits for $B\&B_H$ and quality of lower and upper bound algorithms for irregular arrivals

No. mach.	No. Jobs	Avg. No. Nodes	LB vs. $B\&B_H$ Avg.	LB vs. $B\&B_H$ Max	UB vs. $B\&B_H$ Avg.	UB vs. $B\&B_H$ Max.	Sol. time
	10	80	0.04	0.12	0.07	0.33	< 1
1	20	6387	0.07	0.22	0.14	0.3	< 1
	30	58705	0.09	0.18	0.18	0.41	4
	40	215389	0.04	0.09	0.19	0.31	228
	10	1641	0.01	0.11	0.01	0.09	< 1
2	20	11762	0.06	0.11	0.12	0.26	< 1
	30	4779125	0.06	0.1	0.16	0.21	700
	40	45423418	0.06	0.11	0.19	0.24	3374
	10	21	≈ 0	≈ 0	≈ 0	≈ 0	< 1
3	20	26935	0.01	0.09	0.03	0.11	< 1
	30	30126907	0.04	0.1	0.12	0.2	1443
	40	277753181	0.11	0.18	0.12	0.21	>3600
	10	16	≈ 0	≈ 0	≈ 0	≈ 0	< 1
4	20	1538	≈ 0	0.01	0.009	0.07	< 1
	30	34491940	0.02	0.08	0.07	0.13	1524
	40	262719518	0.1	0.17	0.1	0.15	>3600

$Solution_{LB}/Solution_{LB}$). Gap between initialization heuristic and $B\&B_H$ is reported in the same way in columns 6 and 7.

Solution time with $B\&B_H$ increases when the number of machines increases. In the presence of 3 or 4 machines and 40 jobs, no instance is completely solved within one hour. This is mainly because same instants are visited more than once in the branch and bound tree in the presence of parallel machines. For instance, if two batches can be created with jobs available at an instant and if there are two machines idle at the same time, algorithm processes the first batch on machine one and the second batch on machine two. If, moreover, new jobs are released after batch processing, left branching occurs more than once for the same job(s). Hence the number of nodes in the search tree increases.

We see that the lower bound algorithm performs quite well. The average lower bound value is around 5% which is close to the optimal/best C_{max} value. Regarding the quality of the initialization heuristic, we observe that the difference between the final value given by $B\&B_H$ and the upper bound value is around 15% which leaves room for improvement. For that purpose, we used the optimal/best makespan value as the initialization value and

tested some of the same problem instances. We observed that there is almost no improvement in the solution time. Then, for the same instances, we forced the initial makespan value to be equal to a very big number. We observed that the average solution time increased by an average of 1%. We can thus conclude that the performance of the initialization heuristic is good for decreasing the solution time.

5.1.2. Big size instances

We enlarge the broad of test instances and go beyond 40 jobs to test the behaviour of $B\&B_H$ on larger instances. Table 4 shows test results for 50, 75 and 100 jobs. Solution methods are stopped at the end of one hour since neither $B\&B_H$ nor $MILP_{Lit}$ can terminate within the time limit as also shown in Table 5.

$\triangle LB$ shows the average difference between the best solution and the lower bound solution. There is a slight decrease in the performance of the lower bound algorithm. This is due to the increase in the number of jobs while release dates are within the same interval. There is a slight decrease in the performances of $B\&B_H$ and GA_{Lit} in the presence of a single machine. When the number of machines increases, especially for the case 4 machines, GA_{Lit} performs better than $B\&B_H$ both for solution time and quality of makespan since the size of the branch and bound tree increases exponentially and more computational time is required to improve the solution quality. Performance

Table 4: Benchmarking results on big size instances for irregular arrivals

			$B\&B_H$		$MILP_{Lit}$		GA_{Lit}		AA_{Lit}	
No. mach.	No. Jobs	$\triangle LB$	# NB	$Avg.$ gap	# NB	$Avg.$ gap	# NB	$Avg.$ gap	# NB	$Avg.$ gap
	50	0.05	2	1%	10	8%	3	4%	15	14%
1	75	0.1	1	3%	18	42%	2	3%	7	14%
	100	0.11	2	4%	20	$> 50\%$	5	4%	9	16%
	50	0.11	1	≈ 0	8	7%	4	1%	10	8%
2	75	0.09	-	-	20	18%	2	≈ 0	8	7%
	100	0.07	-	-	20	$> 50\%$	4	≈ 0	12	8%
	50	0.06	-	-	4	1%	2	≈ 0	2	2%
3	75	0.03	2	2%	20	22%	3	≈ 0	2	≈ 0
	100	0.09	3	2%	20	$> 50\%$	1	≈ 0	3	1%
	50	0.06	-	-	5	1%	-	-	1	≈ 0
4	75	0.11	2	≈ 0	20	18%	-	-	2	≈ 0
	100	0.12	4	1%	20	$> 50\%$	-	-	4	≈ 0

Table 5: Summary of average solution times in seconds for irregular arrivals

		Number of jobs						
Method	No.mach	10	20	30	40	50	75	100
$MILP_{Lit}$		<1	>3600	>3600	>3600	>3600	>3600	>3600
BB_H	1	<1	<1	4	228	>3600	>3600	>3600
GA_{Lit}		<1	2	4	8	18	25	62
AA_{Lit}		<1	<1	<1	<1	<1	<1	<1
$MILP_{Lit}$		<1	>3600	>3600	>3600	>3600	>3600	>3600
BB_H	2	<1	<1	700	3374	>3600	>3600	>3600
GA_{Lit}		<1	<1	8	9	28	24	61
AA_{Lit}		<1	<1	<1	<1	<1	<1	<1
$MILP_{Lit}$		<1	>3600	>3600	>3600	>3600	>3600	>3600
BB_H	3	<1	<1	1443	>3600	>3600	>3600	>3600
GA_{Lit}		<1	<1	5	4	35	31	40
AA_{Lit}		<1	<1	<1	<1	<1	<1	<1
$MILP_{Lit}$		<1	>3600	>3600	>3600	>3600	>3600	>3600
BB_H	4	<1	<1	1524	>3600	>3600	>3600	>3600
GA_{Lit}		<1	<1	6	5	32	42	50
AA_{Lit}		<1	<1	<1	<1	<1	<1	<1

of AA_{Lit} also decreases in these big instances due to the increasing difference between the lower bound and optimal makespan values.

5.2. Second instance type: Regular arrivals

These test instances are inspired from the sterilization service of Grenoble University Hospital. RMD sets are sent to the sterilization service twice a day: early in the morning and in the afternoon. RMD sets arriving in the morning are those used the day before. Ones sent in the afternoon are those used in surgeries in the morning. Regarding these informations, we impose two different release dates for job arrivals: 0 and $r_{max}/2$ where r_{max} stands for the closing time of the service. Considering the sterilization service is open 10 hours per day, $r_{max} = 600$. We assume that half of the jobs are released at instant 0 and half of them at $r_{max}/2$.

Tables 6 and 7 summarize the test results for quality of makespan and solution time, respectively. There is a considerable decrease in the solution time of $B\&B_H$ due to having only two different release dates. Left branching is done a few times and thus number of nodes also decreases. Regarding the quality of the lower bound algorithm, we observe that it gives similar results compared to those in irregular arrivals in the presence of medium size jobs.

Table 6: Benchmarking results for regular arrival instances

			$B\&B_H$			$MILP_{Lit}$		GA_{Lit}		AA_{Lit}	
No. mach.	No. Jobs	$\triangle LB$	No. nodes	# NB	$Avg.$ gap	# NB	$Avg.$ gap	# NB	$Avg.$ gap	# NB	$Avg.$ gap
	10	0.02	9	-	-	-	-	-	-	2	4%
	20	0.03	27	2	12.5%	-	-	-	-	4	6%
	30	0.08	72	2	8.3%	-	-	2	8.2%	3	10.5%
1	40	0.06	92	2	4.7%	1	9%	3	5.6%	6	5.5%
	50	0.08	122	-	-	4	7%	2	7%	18	7.6%
	75	0.14	266	1	6%	12	11%	1	23%	20	14%
	100	0.18	526	1	13%	20	$> 50\%$	3	11%	20	11%
	10	0.05	4	-	-	-	-	-	-	-	-
	20	0.02	7	-	-	-	-	-	-	1	16.6%
	30	0.02	32	1	10%	-	-	3	11.1%	3	10%
2	40	0.08	91	1	12.5%	1	14.2%	2	14.2%	3	7.6%
	50	0.12	214	2	13.3%	2	13%	2	7%	10	11%
	75	0.10	229	3	12.5%	14	18%	6	14%	12	8.4%
	100	0.23	871	-	-	20	$> 50\%$	4	13%	18	11%
	10	0.06	30	-	-	-	-	-	-	-	-
	20	0.05	38	-	-	-	-	-	-	4	7%
	30	0.05	78	-	-	-	-	-	-	2	12.5%
3	40	0.04	95	-	-	-	-	2	9%	3	11%
	50	0.12	93	1	4.7%	-	-	-	-	6	13.3%
	75	0.14	257	-	-	12	9%	1	3%	8	12.5%
	100	0.16	589	-	-	20	$> 50\%$	1	7%	5	7%
	10	0.03	4	-	-	-	-	-	-	2	8%
	20	0.02	6	-	-	-	-	-	-	1	6%
	30	0.02	35	-	-	-	-	-	-	4	3%
4	40	0.04	20	-	-	-	-	-	-	1	7%
	50	0.08	36	-	-	-	-	-	-	3	14%
	75	0.11	114	-	-	9	7%	1	11%	8	16%
	100	0.07	228	-	-	20	$> 50\%$	2	3%	4	15%

This observation is consistent since lower bound algorithm takes into account only job sizes.

Although $B\&B_H$ and GA_{Lit} are able to find the best solution most of the time, there is an increase in the average gap for all methods. This is because of arrival of jobs in big quantities. When many jobs are simultaneously released, our problem becomes more like a bin-packing problem and hence even a small increase in the number of batches yields a bigger gap for C_{max}. We observed that our method has at most two more batches compared to

Table 7: Summary of average solution times in seconds for regular arrivals

		Number of jobs						
Method	No.mach	10	20	30	40	50	75	100
$MILP_{Lit}$		<1	>3600	>3600	>3600	>3600	>3600	>3600
BB_H	1	<1	<1	<1	<1	<1	62	1084
GA_{Lit}		<1	<1	2	3	18	22	61
AA_{Lit}		<1	<1	<1	<1	<1	<1	<1
$MILP_{Lit}$		<1	604	>3600	>3600	>3600	>3600	>3600
BB_H	2	<1	<1	<1	<1	<1	61	1300
GA_{Lit}		<1	<1	1	2	12	29	32
AA_{Lit}		<1	<1	<1	<1	<1	<1	<1
$MILP_{Lit}$		<1	422	>3600	>3600	>3600	>3600	>3600
BB_H	3	<1	<1	<1	<1	<1	73	1502
GA_{Lit}		<1	<1	1	3	9	31	45
AA_{Lit}		<1	<1	<1	<1	<1	<1	<1
$MILP_{Lit}$		<1	309	>3600	>3600	>3600	>3600	>3600
BB_H	4	<1	<1	<1	<1	<1	80	1480
GA_{Lit}		<1	<1	<1	5	15	25	37
AA_{Lit}		<1	<1	<1	<1	<1	<1	<1

the number of batches in the solution giving the best makespan value unless provided by $B\&B_H$. This observation is in line with the bin packing results given in Vanderbeck (1999). Regarding $MILP_{Lit}$, it gives the best solution for instances containing less or equal to 30 jobs. However, it requires too much computation. While $B\&B_H$ finds a solution within some seconds, the optimality gap is more than 50% with $MILP_{Lit}$ at the end of 300 seconds for the case of a single machine.

As in the case of irregular arrivals, the performance of the lower bound algorithm decreases when the number of jobs increases. However, this situation has almost no impact on the solution time with $B\&B_H$ since the size of the search tree is small due to having two different job release dates. While $MILP_{Lit}$ is not able to provide a better makespan value for instances containing more than 40 or 50 jobs depending on the number of machines, our method is able to compete with GA_{Lit} in the presence of a few machines. When the number of machines increases, $B\&B_H$ performs better than other methods. Regarding solution times of other methods, AA_{Lit} is very fast and it can find a solution within some miliseconds. GA_{Lit} on the other hand has an increase in its solution time. It can provide a solution in less than one minute for big size instances.

6. Conclusions

In this paper, we studied a parallel batch scheduling problem whose origin is hospital sterilization services. Jobs have different sizes, different release dates and equal processing times. Our objective is to minimize the makespan on parallel identical machines. MILP models in the literature require long computation time for real size instances. Heuristic methods are faster but do not guarantee the optimality for makespan. We presented a branch and bound based heuristic method which can solve instances containing up to 40 jobs within very short time. We tested this method on real life instances and compared the solution quality to other methods from the literature. Numerical results show that our method can provide high quality makespan values in reasonable computational time.

Many extensions of our problem can be considered for future work. Considering there is imperfect knowledge about job arrivals at the sterilization service, uncertain job release dates may be considered. Some dynamic stochastic approaches (e.g., rolling horizon method) can be applied to this new case instead of deterministic methods. Moreover, some other objective functions (e.g., $\sum C_j$) can be studied.

Appendix A. Lower bound algorithm: LB

Table A.8: Notations used in the lower bound algorithm

b	index for batches
$batch_b$	batch indexed b
$size_b$	total size of jobs in batch b
$ready_b$	ready time of batch b for processing
Cap	capacity of a batch
$jobList$	list of unprocessed jobs
j_{first}	first job in $jobList$
v_{first}	size of job j_{first}
r_{first}	release date of job j_{first}
$size_{jobList}$	total size of jobs in $jobList$
nb	minimum number of batches to be created with jobs in $jobList$ ($nb = \lceil size_{jobList}/Cap \rceil$)
$dispM$:	array of machine available times

Input: $jobList_1, jobList_2, dispM$;
Output: integer;
$jobList \leftarrow jobList_1 \cup jobList_2$;
b $\leftarrow 1$;
$size_b \leftarrow 0$;
$C_{max} \leftarrow 0$;
$nb \leftarrow \lceil size_{jobList}/Cap \rceil$;
$nb_{old} \leftarrow 0$;
while $jobList \neq empty$ **do**
  if $size_b + v_{first} \leq Cap$ **then**
    $batch_b \leftarrow batch_b \cup j_{first}$;
    $size_b \leftarrow size_b + v_{first}$;
    $ready_b \leftarrow r_{first}$;
    remove j_{first} from $jobList$;
    $nb_{old} \leftarrow nb$;
    $nb \leftarrow \lceil size_{jobList}/Cap \rceil$;
    if $nb_{old} > nb$ **then**
      $nb \leftarrow nb - 1$;
      $b \leftarrow b + 1$;
      $size_b \leftarrow 0$;
    end
  end
  if $size_b + v_{first} > Cap$ **then**
    $v_{first} \leftarrow v_{first}$ - $(Cap - size_b)$;
    $size_b \leftarrow Cap$;
    $ready_b \leftarrow r_{first}$;
    $nb \leftarrow nb - 1$;
    $b \leftarrow b + 1$;
    $size_b \leftarrow 0$;
  end
end
forall the *b from 1 to nb* **do**
  assign batch b to the machine having the smallest idle time in $dispM$;
end
return *greatest machine idle time in dispM*;

Algorithm 1: Lower Bound algorithm: LB

Appendix B. Branch and Bound heuristic: $B\&B_H$

Table B.9: Notation used in $B\&B_H$

t:	an instant in the problem
$jobList_A$:	set of available jobs by t
$jobList_{UA}$:	set of unreleased jobs by t
$dispM$:	array of machine available times
$best_{C_{max}}$:	best makespan value
C_{max}:	actual makespan value
C_{max_x}:	new makespan value after left and right branching for x=L and x=R, respectively
$valueLB$:	value of lower bound at instant t
r_{last}:	last job release date in the problem
$LB_{leftChildNode}$:	value of the lower bound after left branching
$LB_{rightChildNode}$:	value of the lower bound after right branching
t_x:	instant reached after left and right branching for x=L and x=R, respectively
$jobList_{A_x}$:	list of jobs that become available after left and right branching for x=L and x=R, respectively
$jobList_{UA_x}$:	list of unreleased jobs after left and right branching for x=L and x=R, respectively
$dispM_x$:	array of machine available times after left and right branching for x=L and x=R, respectively
$dispM_{R_{min}}$:	machine having the smallest idle time in array $dispM_R$
$dispM_{x_{max}}$:	machine having the greatest idle time in array $dispM_x$ for x=L and x=R

Input: $t, jobList_A, jobList_{UA}, dispM, C_{max}$;
if *$jobList_A$ and $jobList_{UA}$ are empty and $C_{max} < best_{C_{max}}$* **then**
  $best_{C_{max}} \leftarrow C_{max}$;
end
else
  $valueLB = LB(jobList_A, jobList_{UA}, dispM)$;
  if *($jobList_A$ or $jobList_{UA}$ is not empty) and $valueLB < best_{C_{max}}$*
  then
    create new $jobList_{A_L}$, $jobList_{UA_L}$, $jobList_{A_R}$, $jobList_{UA_R}$;
    create new $dispM_L$, $dispM_R$;
    if $t < r_{last}$ **then**
      $preprocessing_{leftBranch}(.)$;
    end
    $preprocessing_{rightBranch}(.)$;
    if $t < r_{last}$ **then**
      $LB_{leftChildNode} = LB(jobList_{A_L}, jobList_{UA_L}, dispM_L)$;
    end
    else
      $LB_{leftChildNode} \leftarrow \infty$;
    end
    $LB_{rightChildNode} = LB(jobList_{A_R}, jobList_{UA_R}, dispM_R)$;
    $C_{max_L} \leftarrow dispM_{L_{max}}$;
    $C_{max_R} \leftarrow dispM_{R_{max}}$;
    if $LB_{leftChildNode} <= LB_{rightChildNode}$ **then**
      if $t_L - t < p$ *and* $t < r_{last}$ **then**
        $B\&B(t_L, jobList_{A_L}, jobList_{UA_L}, dispM_L, C_{max_L})$;
      end
      $B\&B(t_R, jobList_{R_R}, jobList_{UA_R}, dispM_R, C_{max_R})$;
    end
    else
      $B\&B(t_R, jobList_{A_R}, jobList_{UA_R}, dispM_R, C_{max_R})$;
      if $t_L - t < p$ *and* $t < r_{last}$ **then**
        $B\&B(t_L, jobList_{A_L}, jobList_{UA_L}, dispM_L, C_{max_L})$;
      end
    end
  end
end

Algorithm 2: Branch and Bound Heuristic: $B\&B_H$

Input: $t_L, jobList_A, jobList_{UA}, dispM, jobList_{A_L}, jobList_{UA_L}, dispM_L$
;
$t_L \leftarrow$ release date of first job in $jobList_{UA}$;
$jobList_{A_L} \leftarrow jobList_A \cup$ job(s) j in $jobList_{UA}$ such that $r_j <= t_L$;
$jobList_{UA_L} \leftarrow jobList_{UA} -$ job(s) j in $jobList_{UA}$ such that $r_j <= t_L$;
$dispM_L \leftarrow dispM$;

Algorithm 3: $preprocessing_{leftBranch}()$

Input:
$t, t_R, jobList_A, jobList_{UA}, dispM, jobList_{A_R}, jobList_{UA_R}, dispM_R$;
$jobList_{A_R} \leftarrow jobList_A$;
$jobList_{UA_R} \leftarrow jobList_{UA}$;
create new $jobsInBatch$;
$jobsInBatch \leftarrow createBatch(jobList_{A_R}, 0, jobsInBatch, 0)$;
$jobList_{A_R} \leftarrow jobList_{A_R} - jobsInBatch$;
$dispM_R \leftarrow dispM$;
$dispM_{R_{min}} \leftarrow max(t, dispM_{R_{min}}) + p$;
if $jobList_{A_R}$ *is not empty* **then**
  $t_R \leftarrow dispM_{R_{min}}$;
end
else
  $t_R \leftarrow max(dispM_{R_{min}},$ first job release date in $jobList_{UA_R})$;
end
$jobList_{A_R} \leftarrow jobList_{A_R} \cup$ job(s) j in $jobList_{UR}$ such that $r_j <= t_R$;
$jobList_{UA_R} \leftarrow jobList_{UR} -$ job(s) j in $jobList_{UR}$ such that $r_j <= t_R$;

Algorithm 4: $preprocessing_{rightBranch}()$

Table B.10: Notations used in batch creation procedure

$jobList$:	list of available jobs for batching
$index$:	an integer representing job indexes
$usedCapacity$:	used capacity of the batch
$jobsInBatch$:	jobs in batch
$bestValue$:	value of the best capacity utilization in the current best solution
$jobList_k$:	k^{th} job in $jobList$
$solutionGlobal$:	jobs put in batch in the current best solution

Input: $jobList, index, jobsInBatch, usedCapacity$;
Output: jobs in batch
if $index + 1 \leq$ *last element index in* $jobList$ **then**
  if $usedCapacity +$ *size of* $jobList_{index+1} \leq$ *batch capacity* **then**
    $usedCapacity_{new} \leftarrow usedCapacity +$ size of $jobList_{index+1}$;
    $jobsInBatch_{new} \leftarrow jobsInBatch \cup jobList_{index+1}$;
    if $usedCapacity_{new} > bestValue$ **then**
      $bestValue \leftarrow usedCapacity_{new}$;
      $solutionGlobal \leftarrow jobsInBatch_{new}$;
    end
    if $usedCapacity_{new} = bestValue$ *and number of jobs in* $jobsInBatch_{new} <$ *number of jobs in* $solutionGlobal$ **then**
      $solutionGlobal \leftarrow jobsInBatch_{new}$;
    end
    $createBatch(jobList, index + 1, jobsInBatch_{new}, usedCapacity_{new})$;
  end
  $createBatch(jobList, index + 1, jobsInBatch, usedCapacity)$;
end
return $solutionGlobal$;

Algorithm 5: $createBatch()$

References

Baptiste, P., 2000. Batching identical jobs. Mathematical Methods of Operations Research 52, 355–367.

Caprara, A., Pferschy, U., 2004. Worst-case analysis of the subset sum algorithm for bin packing. Operations Research Letters 32 (2), 159–166.

Cheng, B., Yang, S., Hu, X., Chen, B., 2012. Minimizing makespan and total completion time for parallel batch processing machines with non-identical job sizes. Applied Mathematical Modelling 36 (7), 3161 – 3167.

Cheng, T., Yuan, J., Yang, A., 2005. Scheduling a batch-processing machine subject to precedence constraints, release dates and identical processing times. Computers and Operations Research 32 (4), 849 – 859.

Chou, F., 2007. A joint ga+dp approach for single burn-in oven scheduling problems with makespan criterion. Int J Adv Manuf Technol 35, 587–595.

Chung, S., Tai, Y., Pearn, W., 2009. Minimizing makespan on parallel batch processing machines with non-identical ready time and arbitrary job sizes. International Journal of Production Research 47 (18), 5109–5128.

Damodaran, P., Velez Gallego, M., 2010. Heuristics for makespan minimization on parallel batch processing machineswith unequal job ready times. International Journal of Advanced Manufacturing Technology 49 (9–12), 1119–1128.

Damodaran, P., Velez-Gallego, M., Maya, J., 2011. A grasp approach for makespan minimization on parallel batch processing machines. Journal of Intelligent Manufacturing 22 (5), 767–777.

Damodaran, P., Velez-Gallego, M. C., 2012. A simulated annealing algorithm to minimize makespan of parallel batch processing machines with unequal job ready times. Expert Systems with Applications 39 (1), 1451 – 1458.

Di Mascolo, M., Gouin, A., 2013. A generic simulation model to assess the performance of sterilization services in health establishments. Health care management science 16 (1), 45–61.

Dupont, L., Dhaenens-Flipo, C., 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., Lawler, E., Lenstra, J., Rinnooy Kan, A., 1979. Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of Discrete Mathematics 5, 287–326.

Gupta, J. N., Ho, J. C., 1999. A new heuristic algorithm for the one-dimensional bin-packing problem. Production planning & control 10 (6), 598–603.

Ikura, Y., Gimple, M., 1986. Efficient scheduling algorithms for a single batch processing machine. Operations Research Letters 5 (2), 61–65.

Li, S., 2012. Makespan minimization on parallel batch processing machines with release times and job sizes. Journal of Software 7 (6), 1203–1210.

Li, S., Li, G., Wang, X., Liu, Q., 2005. Minimizing makespan on a single batching machine with release times and non-identical job sizes. Operations Research Letters 33 (2), 157–164.

Liu, L., Ng, C., Cheng, T., 2014. Scheduling jobs with release dates on parallel batch processing machines to minimize the makespan. Optimization Letters 8 (1), 307–318.

Lu, S., Feng, H., Li, X., 2010. Minimizing the makespan on a single parallel batching machine. Theoretical Computer Science 411 (7–9), 1140–1145.

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

Martello, S., Toth, P., 1990. Knapsack Problems: Algorithms and Computer Implementation. John Wiley and Sons.

Mathirajan, M., Sivakumar, A., 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.

Ozturk, O., Espinouse, M.-L., Di Mascolo, M., Gouin, A., 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., Karimi, B., Kashan, A., 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), 1720–1730.

Pearn, W., Hong, J., Tai, Y., 2013. The burn-in test scheduling problem with batch dependent processing time and sequence dependent setup time. International Journal of Production Research 51 (6), 1694–1706.

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

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

Vanderbeck, F., 1999. Computational study of a column generation algorithm for bin packing and cutting stock problems. Mathematical Programming 86 (3), 565–594.

Wang, H., Chou, F., 2010. Solving the parallel batch-processing machines with diferent release times job sizes, and capacity limits by metaheuristics. Expert Systems with Applications: An International Journal 37 (2), 1510–1521.

Yuan, J., Liu, Z., Ng, C., Cheng, T., 2004. The unbounded single machine parallel batch scheduling problem with family jobs and release dates to minimize makespan. Theoretical Computer Science 320 (23), 199 – 212.

Zhang, G., Cai, X., Lee, C., Wong, C., 2001. Minimizing makespan on a single batch processing machine with nonidentical job sizes. Naval Research Logistics 48 (3), 226–240.