APEM journal
Advances in Production Engineering & Management
Volume 13 | Number 4 | December 2018 | pp 373–388
https://doi.org/10.14743/apem2018.4.297
ISSN 1854-6250
Journal home: apem-journal.org
Original scientific paper

Flexible job shop scheduling using zero-suppressed binary decision diagrams

Meolic, R.[a], **Brezočnik, Z.**[a,*]

[a]University of Maribor, Faculty of Electrical Engineering and Computer Science, Maribor, Slovenia

ABSTRACT

A flexible job shop scheduling problem (FJSP) is a widely studied NP-hard combinatorial problem. Its goal is to optimise the production plans for simultaneously produced parts, where each part production consists of executing various operations. Each operation can be executed on several, or even all, available machines. A distinctive subproblem of FJSP is the identification of feasible solutions. A feasible solution is an allocation plan (i.e. assignment of a machine to a particular operation of a part to be produced) yielding an execution schedule satisfying the given resource constraints. FJSP is applied primarily in manufacturing systems, but it can be used to optimise Internet traffic, cloud computing, and other resource scheduling problems as well. So far, the exact methods for solving FJSP have not been considered attractive, since they seemed incapable of coping with real-size problems. This paper proposes a novel exact approach to solving FJSP which can find and count out all schedules of relatively large systems. The approach is successful due to the power of a special data structure called zero-suppressed binary decision diagrams to represent and manipulate the set of all feasible solutions efficiently. All the algorithms are implemented and tested by using our free Binary Decision Diagram package called Biddy.

ARTICLE INFO

Keywords:
Process planning;
Exact optimization;
Flexible job shop scheduling;
Unate cube set algebra;
Zero-suppressed binary decision diagram

**Corresponding author:*
zmago.brezocnik@um.si
(Brezočnik, Z.)

Article history:
Received 17 June 2018
Revised 13 November 2018
Accepted 15 November 2018

1. Introduction

Manufacturing sites have to plan their production to ensure the profitability of manufacturing products, resource utilization, and product delivery time. Production planning involves many attributes that can be categorised into different domains [1, 2]. Process planning and scheduling are the two most essential tasks in a manufacturing company [3]. From the computer science perspective, they are formulated together as a job shop scheduling problem. The job shop scheduling problem is about the optimization of the production of several parts which are produced simultaneously. Each part can be produced by one or more sequences of operations. Operations are executed using a given set of machines. In a flexible job shop scheduling problem (FJSP), all machines can perform all operations (total flexibility), or each machine can execute a subset of operations (partial flexibility). The processing time for operations varies on different machines. The following assumptions are also very common: parts are produced independently of each other, sequences of operations and processing times are fixed and known in advance, setup time and transport time are either negligible, or included in the processing time, all machines are available all the time, the operation execution cannot be interrupted.

Methods for FJSP can be categorised either as exact or approximation. Exact methods can obtain an exact optimal solution. However, they do not scale well for solving large FJSP prob-

lems. Therefore, most of the approaches to FJSP resort to approximation methods such as genetic and evolutionary algorithms. Such methods can solve large-scale problems, but may lack either local or global search ability. Our method of process planning and scheduling is oriented towards the generation of the exact solution, and is based on the efficient realization of unate cube set algebra with Zero-suppressed Binary Decision Diagrams (ZBDDs) [4, 5]. The previous work that has influenced the proposed method the most are [1] and [6]. Takahashi *et al.* [1] used ZBDDs to represent solution candidates that satisfy the constraints in process planning. Jensen *et al.* [6] implemented the task graph scheduling problem with uniform processors and arbitrary task execution times as a state space exploration problem, and solved it with Binary Decision Diagrams (BDDs) [7], but they did not use the unate cube set algebra and ZBDDs.

The remainder of this paper has the following structure. The framework for FJSP is specified in Section 2. Generating of feasible solutions is discussed in Section 3. The novel scheduling algorithm is introduced in Section 4. Section 5 gives the necessary background on ZBDDs, operations in unate cube set algebra, and gives complete algorithms for feasible solutions' generation and scheduling. Results of experimental studies are reported in Section 6. Section 7 describes the conclusions and challenges of future work.

2. Specification of a FJSP test case

For specifying the framework for FJSP we use the approach from [1]. The factory production programme that will be observed in this paper is given in Fig. 1. It contains eight different parts, denoted as P_s, $s = 1, \ldots, 8$. Each part can be produced in several different ways, called *process sequences.* An operation type O_w is performed at each step of a process sequence. The number of operations is 15. The order of operations within a process sequence is important. Operations are denoted with squared nodes, and the precedence relation between the nodes with an arrow. Moreover, the graphical representation includes two types of brackets. The vertical bars »||« enclosing nodes indicate alternative sequences of the enclosed nodes, and the square brackets »[]« mean the arbitrary order of the bracketed nodes. For example, in the manufacturing of P_1, there are three selectable alternative process sequences: $O_1 \rightarrow O_2 \rightarrow O_3$, $O_3 \rightarrow O_5 \rightarrow O_4$, or $O_3 \rightarrow O_1 \rightarrow O_4$. On the other hand, for example, any process sequence in P_3 begins with O_4 followed by O_7 or with O_7 followed by O_4. Thus, the process sequences for P_3 are $O_4 \rightarrow O_7 \rightarrow O_8$, $O_4 \rightarrow O_7 \rightarrow O_9$, $O_7 \rightarrow O_4 \rightarrow O_8$, or $O_7 \rightarrow O_4 \rightarrow O_9$. Finding process sequences for other parts is straightforward.

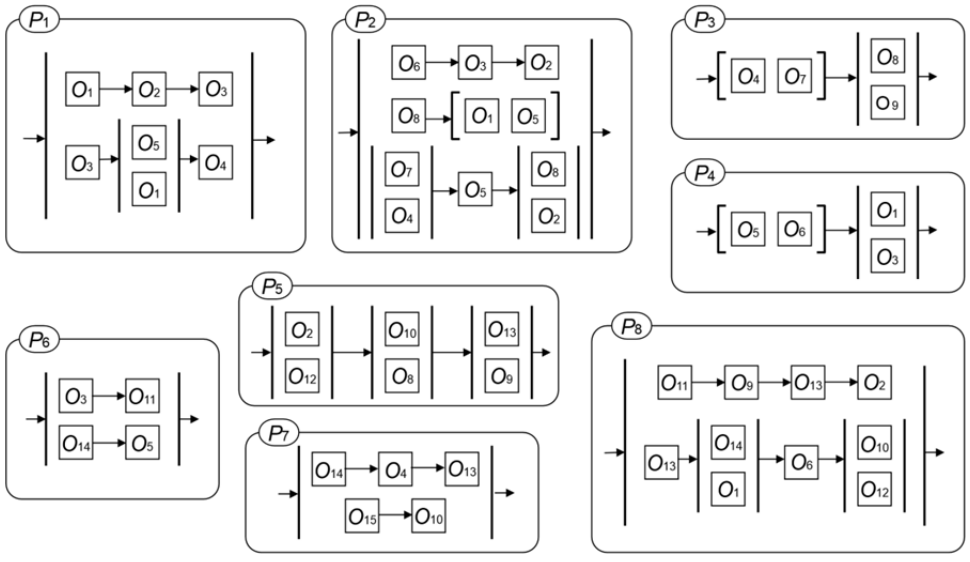

Fig. 1 Process sequences for a factory producing eight parts

In the factory, the production of parts is performed by a set of machine instances of different machine types $M_i, i = 1, \dots, 8$. Each machine type has a given number of instances with the same performance. Table 1 shows information on the available machine types, operations, number of machine instances of each machine type, and the processing time for operation O_w on machine type M_i. For example, O_1can be processed on an instance of the machine type M_1, M_2, or M_7. The processing time of O_1 on the machine type M_1, M_2, or M_7 is 6, 5, or 3 time units, respectively. All the values in Table 1 are taken from [2].

A set of machine instances installed in a factory is called the *factory configuration*. We assume that the maximum number of installable machine instances in the factory is equal to or less than the *factor capacity*. For example, if the factory capacity is 3, then the set of valid factory configurations includes the following ones: one instance of M_1, M_2, and M_3, two instances of M_1 and one instance of M_2 and also one instance of M_2 and one instance of M_3 as well, because the number of installed machines can be smaller than the factory capacity.

Table 1 Information on operations, machine types, number of machine type instances, and processing time

	Operations														
Machine type (#instances)	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}
M_1 (3)	6	-	-	6	9	-	2	-	-	-	-	-	3	-	-
M_2 (1)	5	8	-	-	-	5	4	-	-	-	-	4	-	-	-
M_3 (1)	-	-	10	-	8	-	-	6	-	-	8	-	-	-	-
M_4 (2)	-	-	8	4	-	6	-	-	3	-	-	-	-	10	-
M_5 (1)	-	-	-	-	7	5	-	-	-	-	5	-	-	-	8
M_6 (1)	-	4	-	-	-	-	1	-	-	5	-	2	2	-	-
M_7 (2)	3	-	-	-	-	-	-	-	-	2	-	1	-	-	12
M_8 (1)	-	-	-	-	-	-	-	-	5	-	3	-	4	14	-

3. Generating feasible solutions for process and resource planning with limits

A feasible solution is a combination of process sequences, machine types, and valid factory configuration which can be used to produce all required parts. For example, if a factory produces only part P_1 and factory capacity is 3, then one of the feasible solutions is a process sequence $O_3 \rightarrow O_1 \rightarrow O_4$, combined with the information that two instances of machine type M_1 and one instance of machine type M_3 are installed, and operation O_3 is processed on the machine type M_3, operation O_1 is processed on the first instance of the machine type M_1, and operation O_4 is processed on the second instance of the machine type M_1. Another feasible solution is a similar one, where both O_1 and O_4 are processed in the first instance of the machine type M_1. It is expected that the first mentioned feasible solution would be preferred, due to the fact that the part will be completed earlier, since operations O_1 and O_4 can be carried out in parallel.

3.1 Encoding the process planning problem with unate cube set algebra

Unate cube set algebra is a mathematical theory about manipulation of combination sets [4, 5]. For the given *universal set* of elements, a *combination set* (or a *cube set*) is a set of its subsets also called *cubes*. For example, if the universal set is {a,b,c,d}, then some of the possible combination sets are {}, {{}}, {{a},{b,d}}, and {{},{b,c},{b,c,d}}. The first one from this list is an *empty set*. The second one, which consists of the *empty cube* only, is called a *base set* (in some publications it is called a *unit set*). We shorten the notation of combination sets such that {{},{a},{b,c},{b,c,d}} is written as {1;a;bc;bcd}.

In encoding the process planning problem, the elements of a universal set are as follows:

- one $O^s_{w,r}$ element (called the O-variable) for each possible combination of values s, w and r denoting operation O_w, performed on the r-th place of a process sequence for part P_s.
- one $MX_{i,j}$ element (called the MX-variable) for each possible combination of values i and j, denoting an instance j for machine type M_i.

- one $M_i^{s,w,r}$ element (called the M-variable) for each possible combination of values s, w, r and i, denoting a machine type M_i for operation O_w, performed on the r-th place of a process sequence for part P_s.

For compatibility with [1] and [2], from now on, we write MX-variables without the letter X. Thus, M-variables and MX-variables differ only in the number of indices; the former ones have 4 and the latter 2 indices, respectively.

3.2 Algorithms for process and resource planning

There are several algorithms involved in the calculation of feasible solutions for process and resource planning, which are applied consecutively. To illustrate the ideas of the algorithms involved, we use a manageable small example, consisting only of part P_3 from Fig. 1 and machine types M_1, M_2, and M_3 from Table 1. For the sake of simplicity, let us set the factory capacity to 3.

The calculation starts by generating all possible *process sequences* for the production of P_3. Process sequences are encoded with cubes. According to the semantics of the graphical representation explained in Subsection 2.1, part P_3 is described with a set X^3 consisting of four cubes:

$$X^3 = \{O^3_{4,1}O^3_{7,2}O^3_{8,3}; O^3_{4,1}O^3_{7,2}O^3_{9,3}; O^3_{7,1}O^3_{4,2}O^3_{8,3}; O^3_{7,1}O^3_{4,2}O^3_{9,3}\} \tag{1}$$

In the second step, a process plan is generated for each part. A process plan is a set of all possible process sequences, extended with the information about the suitable machine types for processing each operation. Because there are many possibilities to complete an operation (e.g., operation O_7 can be completed either on machine type M_1 or M_2), we get more combinations than after the initial step. For our simple example, we get four combinations:

$$X^3 = \begin{Bmatrix} O^3_{4,1}M^{3,4,1}_1 O^3_{7,2}M^{3,7,2}_1 O^3_{8,3}M^{3,8,3}_3; O^3_{4,1}M^{3,4,1}_1 O^3_{7,2}M^{3,7,2}_2 O^3_{8,3}M^{3,8,3}_3; \\ O^3_{7,1}M^{3,7,1}_1 O^3_{4,2}M^{3,4,2}_1 O^3_{8,3}M^{3,8,3}_3; O^3_{7,1}M^{3,7,1}_2 O^3_{4,2}M^{3,4,2}_1 O^3_{8,3}M^{3,8,3}_3 \end{Bmatrix} \tag{2}$$

Please note that by assigning machine types to the process sequences, we also cut sequences that are not realisable. For example, operation O_9 is not used in any cube because it cannot be completed using the available set of machine types.

The third step is the construction of a *comprehensive process plan X* by computing a Cartesian product of all parts' process plans under the assumption that parts can be produced in parallel. They are not completely independent of each other, because they share the available machine types. However, in our small example, which illustrates the step-by-step evolving of the set of feasible solutions, the factory produces only one part, therefore, the comprehensive process plan is the same as the obtained process plan in the previous step ($X = X^3$). In general, the number of cubes in the comprehensive process plan is a product of cube numbers of all the produced parts.

In the fourth step, a set of MX-variables is added to each cube. The resulting set of feasible solutions is called a *process and resource plan*. All possible factory configurations should be considered up to three installed machine type instances. A process and resource plan is a huge set for any non-trivial problem. For our small example, we get a set of 12 cubes:

$$X = \begin{Bmatrix} O^3_{4,1}M^{3,4,1}_1 O^3_{7,2}M^{3,7,2}_1 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{3,1}M_{1,2}M_{1,3}; O^3_{4,1}M^{3,4,1}_1 O^3_{7,2}M^{3,7,2}_1 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{3,1}M_{1,2}; \\ O^3_{4,1}M^{3,4,1}_1 O^3_{7,2}M^{3,7,2}_1 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{3,1}; O^3_{4,1}M^{3,4,1}_1 O^3_{7,2}M^{3,7,2}_2 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{2,1}M_{3,1}M_{1,2}M_{1,3}; \\ O^3_{4,1}M^{3,4,1}_1 O^3_{7,2}M^{3,7,2}_2 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{2,1}M_{3,1}M_{1,2}; O^3_{4,1}M^{3,4,1}_1 O^3_{7,2}M^{3,7,2}_2 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{2,1}M_{3,1}; \\ O^3_{7,1}M^{3,7,1}_1 O^3_{4,2}M^{3,4,2}_1 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{3,1}M_{1,2}M_{1,3}; O^3_{7,1}M^{3,7,1}_1 O^3_{4,2}M^{3,4,2}_1 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{3,1}M_{1,2}; \\ O^3_{7,1}M^{3,7,1}_1 O^3_{4,2}M^{3,4,2}_1 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{3,1}; O^3_{7,1}M^{3,7,1}_2 O^3_{4,2}M^{3,4,2}_1 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{2,1}M_{3,1}M_{1,2}M_{1,3}; \\ O^3_{7,1}M^{3,7,1}_2 O^3_{4,2}M^{3,4,2}_1 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{2,1}M_{3,1}M_{1,2}; O^3_{7,1}M^{3,7,1}_2 O^3_{4,2}M^{3,4,2}_1 O^3_{8,3}M^{3,8,3}_3 M_{1,1}M_{2,1}M_{3,1} \end{Bmatrix} \tag{3}$$

In the presented result we have taken into account the fact that configurations of the same length consisting of the same machine types (e.g. $\{M_{1,1}, M_{1,2}\}, \{M_{1,1}, M_{1,3}\}$, and $\{M_{1,2}, M_{1,3}\}$), yield symmetric results and, thus, it is enough to keep only one of them. We keep a configuration

consisting of variables that are declared earlier (in this case $\{M_{1,1}, M_{1,2}\}$), and remove the others. A less effective approach is used in [1] and [2]. There, the authors first generate a process and resource plan without considering the mentioned redundancy, and then have to restrict the result by an extra algorithm.

In the last step, the result is limited to the factory capacity. Only six cubes meet the requirement that the maximum number of the installed machine types instances in the factory is three. This last step gives us the set of all feasible solutions:

$$X = \left\{ \begin{array}{c} O_{4,1}^{3} M_1^{3,4,1} O_{7,2}^{3} M_1^{3,7,2} O_{8,3}^{3} M_3^{3,8,3} M_{1,1} M_{3,1} M_{1,2};\ O_{4,1}^{3} M_1^{3,4,1} O_{7,2}^{3} M_1^{3,7,2} O_{8,3}^{3} M_3^{3,8,3} M_{1,1} M_{3,1}; \\ O_{4,1}^{3} M_1^{3,4,1} O_{7,2}^{3} M_2^{3,7,2} O_{8,3}^{3} M_3^{3,8,3} M_{1,1} M_{2,1} M_{3,1};\ O_{7,1}^{3} M_1^{3,7,1} O_{4,2}^{3} M_1^{3,4,2} O_{8,3}^{3} M_3^{3,8,3} M_{1,1} M_{3,1} M_{1,2}; \\ O_{7,1}^{3} M_1^{3,7,1} O_{4,2}^{3} M_1^{3,4,2} O_{8,3}^{3} M_3^{3,8,3} M_{1,1} M_{3,1};\ O_{7,1}^{3} M_2^{3,7,1} O_{4,2}^{3} M_1^{3,4,2} O_{8,3}^{3} M_3^{3,8,3} M_{1,1} M_{2,1} M_{3,1} \end{array} \right\} \quad (4)$$

4. Job shop scheduling algorithm

In general, the job shop scheduling problem consists of two subproblems: determining the best feasible solution, and determining the best schedule for this solution. After creating the set of all feasible solutions, we solve these subproblems jointly. The state space is examined step by step until we find the first set of feasible solutions (containing either one single solution or more solutions) that completes the production of all parts. Alternatively, the computation may continue until all feasible solutions are examined.

The straightforward criterion for job shop scheduling is the amount of time, called *makespan*, needed to produce all parts. Other criteria can be observed as well, such as the *total workload* (the sum of working times of all machines), and the *workload of the critical machine* (the working time of the most loaded machine).

4.1 Extending the universal set

The proposed scheduling algorithm is an extension of the approach to generate feasible solutions. A similar scheduling algorithm was introduced in [6], but it is not based on unate cube set algebra.

For scheduling, we extend the universal set with the following elements:

- one $W^{s,r}$ element (called the W-variable) for each possible combination of values s and r, denoting that processing of the part type P_s is waiting to start the r-th operation,
- one $R^{s,r}$ element (called the R-variable) for each possible combination of values s and r, denoting that production of the part type P_s is running the r-th operation,
- one $B_{i,j}$ element (called the B-variable) for each possible combination of values i and j, denoting that the j-th instance of machine type M_i is busy,
- several T_n^s elements (called T-variables) for each possible combination of values s and n, denoting that processing of the part type P_s needs n time units to complete,
- one $S_{i,j}^{s,r}$ element (called the S-variable) for each possible combination of values s, r, i, and j, denoting that the r-th operation of a process sequence of the part type P_s is scheduled to a j-th instance of machine type M_i,
- one FS_n element (called the FS variable) for each possible value of n, denoting that n different machines type instances (not n different machines types) are needed, and
- some $G_t^{s,r}$ elements (called G variables), denoting that the r-th operation of a process sequence of the part type P_s started after t time units from the beginning of the scheduling.

4.2 Scheduling algorithm

Let us continue the example from Subsection 3.2, where six feasible solutions were found. Before the scheduling begins, the initialization is carried out. It transforms the set of feasible solutions into a *working set of cubes* in the following way:

1. The size of factory configuration is added to each cube by appending the appropriate FS-variable. These variables help sorting the obtained solutions at the end, and also make scheduling algorithm more efficient.
2. Each part is marked with "waiting to start the first operation" by adding the appropriate W-variable.

In our small example that illustrates the production of a single part P_3, all cubes require a factory with either two or three machine instances. The working set of cubes X is obtained immediately after the initialization by transforming the set of feasible solution (Eq. 4):

$$X = \begin{Bmatrix} W^{3,1}O^{3}_{4,1}M^{3,4,1}_{1}O^{3}_{7,2}M^{3,7,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{3,1}M_{1,2}FS_3; \\ W^{3,1}O^{3}_{4,1}M^{3,4,1}_{1}O^{3}_{7,2}M^{3,7,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{3,1}FS_2; \\ W^{3,1}O^{3}_{4,1}M^{3,4,1}_{1}O^{3}_{7,2}M^{3,7,2}_{2}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{2,1}M_{3,1}FS_3; \\ W^{3,1}O^{3}_{7,1}M^{3,7,1}_{1}O^{3}_{4,2}M^{3,4,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{3,1}M_{1,2}FS_3; \\ W^{3,1}O^{3}_{7,1}M^{3,7,1}_{1}O^{3}_{4,2}M^{3,4,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{3,1}FS_2; \\ W^{3,1}O^{3}_{7,1}M^{3,7,1}_{2}O^{3}_{4,2}M^{3,4,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{2,1}M_{3,1}FS_3 \end{Bmatrix} \quad (5)$$

Scheduling consists of three phases. In the first phase, every cube in the working set is checked whether it includes a part waiting to start the next operation. If such a part exists, then it is scheduled for all appropriate free machine instances. Indeed, if there is more than one such instance, the number of cubes in the working set is increased. All of the waiting parts are scheduled simultaneously. All parts that can be scheduled are scheduled (see Subsection 5.4 for notes about omitting this requirement). In any case, two parts cannot both be scheduled to the same machine instance. More formally, the transformation is described by Rule TR1.

Rule TR1: If a cube includes $W^{s,r}$, $M_i^{s,w,r}$, $M_{i,j}$, and does not include $R^{s,r}$ and $B_{i,j}$, then remove $M_i^{s,w,r}$ and $W^{s,r}$ and add $R^{s,r}$, $B_{i,j}$, $S_{i,j}^{s,r}$, T_n^s, and $G_t^{s,r}$, where n is the number of time units needed to complete operation O_w on machine type M_i, and t is the number of time units from the beginning of the scheduling (determined by the number of repetitions of a scheduling loop).

Rule TR1 is applied maximally, i.e. each cube is transformed in the form for which the rule is no longer applicable. To get rid of different but equally efficient solutions if more instances of the same machine type are free, we create only a cube where the operation is scheduled for the instance with the smallest index j. We get the following working set of cubes (variables are ordered using the ordering that turns out to be the most efficient for the ZBDD representation, as explained in Subsection 5.6):

$$X = \begin{Bmatrix} G^{3,1}_{0}B_{1,1}T^{3}_{2}R^{3,1}S^{3,1}_{1,1}O^{3}_{7,1}O^{3}_{4,2}M^{3,4,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{3,1}M_{1,2}FS_3; \\ G^{3,1}_{0}B_{1,1}T^{3}_{2}R^{3,1}S^{3,1}_{1,1}O^{3}_{7,1}O^{3}_{4,2}M^{3,4,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{3,1}FS_2; \\ G^{3,1}_{0}B_{1,1}T^{3}_{6}R^{3,1}S^{3,1}_{1,1}O^{3}_{4,1}O^{3}_{7,2}M^{3,7,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{3,1}M_{1,2}FS_3; \\ G^{3,1}_{0}B_{1,1}T^{3}_{6}R^{3,1}S^{3,1}_{1,1}O^{3}_{4,1}O^{3}_{7,2}M^{3,7,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{3,1}FS_2; \\ G^{3,1}_{0}B_{1,1}T^{3}_{6}R^{3,1}S^{3,1}_{1,1}O^{3}_{4,1}O^{3}_{7,2}M^{3,7,2}_{2}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{2,1}M_{3,1}FS_3; \\ G^{3,1}_{0}B_{2,1}T^{3}_{4}R^{3,1}S^{3,1}_{2,1}O^{3}_{7,1}O^{3}_{4,2}M^{3,4,2}_{1}O^{3}_{8,3}M^{3,8,3}_{3}M_{1,1}M_{2,1}M_{3,1}FS_3 \end{Bmatrix} \quad (6)$$

As said above, we decided to explain the scheduling algorithm for a small example with a single part P_3 to keep the number of cubes in the evolving working sets as small as possible. Therefore, only one operation (the first one in each process sequence) is scheduled in every cube of Eq. 6. In general, if the factory produces several different part types, many operations are scheduled simultaneously.

In the second scheduling phase, each cube in the working set is checked whether it includes an active part. If such a part exists, its "time to finish" is decreased by one. This procedure is realised by adapting all T-variables and can be described formally by Rule TR2.

Rule TR2: If the cube includes T_n^s and $n > 0$, then remove T_n^s and add T_{n-1}^s.

As before, the rule is applied maximally. For our example, the result of the second scheduling phase is the same as given before, only all variables T_n^3 are replaced with variables T_{n-1}^3.

In the third scheduling phase, every cube in the working set is checked whether it includes a part that has completed the operation. If such a part exists, the busy machine instance is released, and the part is moved forward into the "waiting to start the next operation" state. This procedure is expressed formally by Rule TR3:

Rule TR3: If the cube includes $R^{s,r}$, $S_{i,j}^{s,r}$, and T_0^s, then remove $R^{s,r}$, T_0^s, and $B_{i,j}$, and add $W^{s,r+1}$.

In the obtained working set of cubes, none of the parts has completed an operation. Thus, the third scheduling phase does not change anything. All three scheduling phases are then repeated, and the following working set of cubes is obtained after applying Rule TR2 for the second time:

$$X = \begin{Bmatrix} G_0^{3,1}B_{1,1}T_0^3R^{3,1}S_{1,1}^{3,1}O_{7,1}^3O_{4,2}^3M_1^{3,4,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{3,1}M_{1,2}FS_3\ ; \\ G_0^{3,1}B_{1,1}T_0^3R^{3,1}S_{1,1}^{3,1}O_{7,1}^3O_{4,2}^3M_1^{3,4,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{3,1}FS_2\ ; \\ G_0^{3,1}B_{1,1}T_4^3R^{3,1}S_{1,1}^{3,1}O_{4,1}^3O_{7,2}^3M_1^{3,7,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{3,1}M_{1,2}FS_3\ ; \\ G_0^{3,1}B_{1,1}T_4^3R^{3,1}S_{1,1}^{3,1}O_{4,1}^3O_{7,2}^3M_1^{3,7,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{3,1}FS_2\ ; \\ G_0^{3,1}B_{1,1}T_4^3R^{3,1}S_{1,1}^{3,1}O_{4,1}^3O_{7,2}^3M_2^{3,7,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{2,1}M_{3,1}FS_3\ ; \\ G_0^{3,1}B_{2,1}T_2^3R^{3,1}S_{2,1}^{3,1}O_{7,1}^3O_{4,2}^3M_1^{3,4,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{2,1}M_{3,1}FS_3 \end{Bmatrix} \tag{7}$$

Now, the last two cubes include the completed operations. Thus, Rule TR3 is applied:

$$X = \begin{Bmatrix} G_0^{3,1}B_{1,1}T_4^3R^{3,1}S_{1,1}^{3,1}O_{4,1}^3O_{7,2}^3M_1^{3,7,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{3,1}M_{1,2}FS_3\ ; \\ G_0^{3,1}B_{1,1}T_4^3R^{3,1}S_{1,1}^{3,1}O_{4,1}^3O_{7,2}^3M_1^{3,7,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{3,1}FS_2\ ; \\ G_0^{3,1}B_{1,1}T_4^3R^{3,1}S_{1,1}^{3,1}O_{4,1}^3O_{7,2}^3M_2^{3,7,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{2,1}M_{3,1}FS_3\ ; \\ G_0^{3,1}B_{2,1}T_2^3R^{3,1}S_{2,1}^{3,1}O_{7,1}^3O_{4,2}^3M_1^{3,4,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{2,1}M_{3,1}FS_3\ ; \\ G_0^{3,1}S_{1,1}^{3,1}O_{7,1}^3W^{3,2}O_{4,2}^3M_1^{3,4,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{3,1}M_{1,2}FS_3\ ; \\ G_0^{3,1}S_{1,1}^{3,1}O_{7,1}^3W^{3,2}O_{4,2}^3M_1^{3,4,2}O_{8,3}^3M_3^{3,8,3}M_{1,1}M_{3,1}FS_2 \end{Bmatrix} \tag{8}$$

Scheduling is continued by repeating all three phases. If a cube without M-variables and R-variables appears in the working set, it represents a solution to the scheduling problem. For the example under consideration, this happens after 14 time units, when we get the following result:

$$X = \begin{Bmatrix} G_{10}^{3,3}G_6^{3,2}G_0^{3,1}B_{3,1}T_2^3S_{1,1}^{3,1}O_{4,1}^3S_{2,1}^{3,2}O_{7,2}^3R^{3,3}S_{3,1}^{3,3}O_{8,3}^3M_{1,1}M_{2,1}M_{3,1}FS_3\ ; \\ G_{10}^{3,3}G_4^{3,2}G_0^{3,1}B_{3,1}T_2^3S_{2,1}^{3,1}O_{7,1}^3S_{1,1}^{3,2}O_{4,2}^3R^{3,3}S_{3,1}^{3,3}O_{8,3}^3M_{1,1}M_{2,1}M_{3,1}FS_3\ ; \\ G_8^{3,3}G_6^{3,2}G_0^{3,1}S_{1,1}^{3,1}O_{4,1}^3S_{1,1}^{3,2}O_{7,2}^3S_{3,1}^{3,3}O_{8,3}^3W^{3,4}M_{1,1}M_{3,1}M_{1,2}FS_3\ ; \\ G_8^{3,3}G_6^{3,2}G_0^{3,1}S_{1,1}^{3,1}O_{4,1}^3S_{1,1}^{3,2}O_{7,2}^3S_{3,1}^{3,3}O_{8,3}^3W^{3,4}M_{1,1}M_{3,1}FS_2\ ; \\ G_8^{3,3}G_2^{3,2}G_0^{3,1}S_{1,1}^{3,1}O_{7,1}^3S_{1,1}^{3,2}O_{4,2}^3S_{3,1}^{3,3}O_{8,3}^3W^{3,4}M_{1,1}M_{3,1}M_{1,2}FS_3\ ; \\ G_8^{3,3}G_2^{3,2}G_0^{3,1}S_{1,1}^{3,1}O_{7,1}^3S_{1,1}^{3,2}O_{4,2}^3S_{3,1}^{3,3}O_{8,3}^3W^{3,4}M_{1,1}M_{3,1}FS_2 \end{Bmatrix} \tag{9}$$

The last four cubes are solutions. Now, the algorithm either stops and reports one solution, reports all solutions, or merely removes all solutions and continues with scheduling until the working set of cubes is not empty. The remaining W-variable in every solution is not significant, it was only added to mark that the process sequence is finished (no process sequence for part P_3 has the fourth operation).

In the observed small example, the factory is producing only one part (P_3), and there is no competition for resources. Thus, every feasible solution has one schedule. In general, there may be several different schedules for the same feasible solution. Let us observe another problem where parts P_1, P_2, and P_3 from Fig. 1 have to be produced by using machine types M_1, M_2, M_3, and M_4 from Table 1, while the factory capacity is 2. This problem has 56 feasible solutions.

After 27 time units, the scheduling algorithm generates a working set with 763 cubes, among which there are three solutions:

$$G_{24}^{3,3}G_{20}^{3,2}G_{16}^{2,3}G_{16}^{1,3}G_{9}^{1,2}G_{8}^{2,2}G_{5}^{3,1}G_{0}^{2,1}G_{0}^{1,1}S_{4,1}^{1,1}O_{3,1}^{1}S_{2,1}^{1,2}O_{1,2}^{1}S_{4,1}^{1,3}O_{4,3}^{1}S_{2,1}^{2,1}O_{6,1}^{2}S_{4,1}^{2,2}O_{3,2}^{2}S_{2,1}^{2,3}O_{2,3}^{2}S_{2,1}^{3,1}O_{7,1}^{3}S_{4,1}^{3,2}O_{4,2}^{3}S_{4,1}^{3,3}O_{9,3}^{3}M_{2,1}M_{4,1}FC_2;$$
$$G_{24}^{3,3}G_{20}^{1,3}G_{16}^{3,2}G_{16}^{2,3}G_{9}^{1,2}G_{8}^{2,2}G_{5}^{3,1}G_{0}^{2,1}G_{0}^{1,1}S_{4,1}^{1,1}O_{3,1}^{1}S_{2,1}^{1,2}O_{1,2}^{1}S_{4,1}^{1,3}O_{4,3}^{1}S_{2,1}^{2,1}O_{6,1}^{2}S_{4,1}^{2,2}O_{3,2}^{2}S_{2,1}^{2,3}O_{2,3}^{2}S_{2,1}^{3,1}O_{7,1}^{3}S_{4,1}^{3,2}O_{4,2}^{3}S_{4,1}^{3,3}O_{9,3}^{3}M_{2,1}M_{4,1}FC_2;$$
$$G_{23}^{1,3}G_{20}^{3,3}G_{16}^{3,2}G_{16}^{2,3}G_{9}^{1,2}G_{8}^{2,2}G_{5}^{3,1}G_{0}^{2,1}G_{0}^{1,1}S_{4,1}^{1,1}O_{3,1}^{1}S_{2,1}^{1,2}O_{1,2}^{1}S_{4,1}^{1,3}O_{4,3}^{1}S_{2,1}^{2,1}O_{6,1}^{2}S_{4,1}^{2,2}O_{3,2}^{2}S_{2,1}^{2,3}O_{2,3}^{2}S_{2,1}^{3,1}O_{7,1}^{3}S_{4,1}^{3,2}O_{4,2}^{3}S_{4,1}^{3,3}O_{9,3}^{3}M_{2,1}M_{4,1}FC_2$$

All three reported schedules are based on the same feasible solution, because they have the same O-variables, S-variables, and MX-variables.

5. Implementation using zero-suppressed binary decision diagrams

Unate cube set algebra can be realised very efficiently using ZBDDs. These are one of many variants of BDDs, a relatively new computer data structure studied intensively in the 1990s [4, 7]. A computer library that implements manipulation of BDDs is called a BDD package. Only some of the available BDD packages support ZBDDs. This section gives the basic idea, describes the necessary operations, and gives algorithms used in Section 4. The implementation details of a BDD package are out of the scope of this paper. The interested readers should see [4, 5, 7], and the source code of the Biddy BDD package (available from biddy.meolic.com), which is free software used to implement all the presented algorithms [8].

5.1 Zero-suppressed binary decision diagrams

A ZBDD is a rooted binary directed acyclic graph. Every node except the leaves is called an internal node and has two descendants, called 'else' and 'then' successors, respectively. The ZBDD has an edge to the root, which is called the top edge. Each node is associated with a label. The ZBDD evaluates so that labels in internal nodes are treated as elements, and combination sets are associated with the edges. The leaves are labelled with 0 or 1, and are called terminal nodes. The label in the root is called the top label. The efficiency of ZBDDs is enhanced by fixing the order of elements along every path from the root to a leaf and minimising the graph.

The combination set can be determined precisely from a ZBDD by finding all paths starting with the top edge and leading to a terminal node 1. Each such path represents a cube. An element is included in the cube if, and only if, such a path goes through its 'then' successor. The resulting combination set is a union of the obtained cubes. Examples of some simple combination sets are given in Fig. 2.

While in the worst case, the size of ZBDDs grows exponentially with the number of elements, in many practical examples they are a very efficient data structure for the representation of combination sets. They are especially suitable for representing sets of sparse cubes, i.e., if the universal set has many elements but the cubes include only a few of them. For example, a ZBDD representing an enormous combination set including about 32×10^{12} cubes is represented with only 195 ZBDD nodes (see Section 6). The element order has a huge impact on the size of the ZBDD. To represent the same combination set with less optimal element order could require millions of ZBDD nodes.

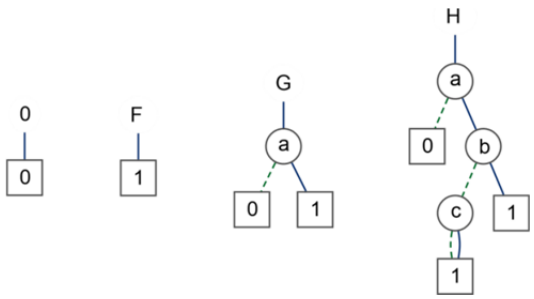

Fig. 2 ZBDDs representing different combination sets for a universal set {a,b,c}. From left to right, there are an empty set {}, a base set {{}}, set {{a}}, and set {{a},{a,b},{a,c}}

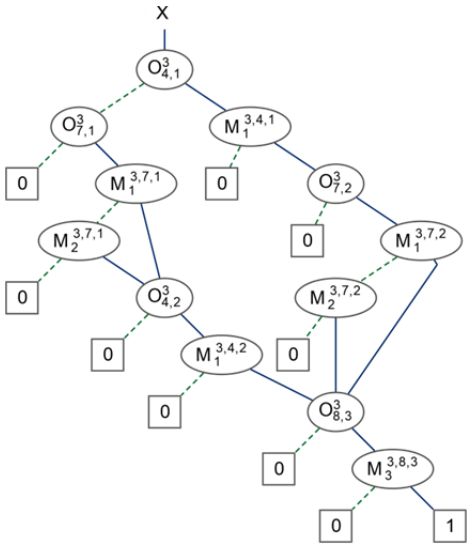

Fig. 3 A ZBDD representing a combination set (Eq. 2) with four cubes

We conclude this brief introduction of ZBDDs with Fig. 3, which gives a ZBDD representing a process plan for production of product P_3. The universal set includes five O-variables and seven M-variables. The combination set in Fig. 3 has four cubes, as described in Subsection 3.2.

5.2 Operations in unate cube set algebra

In unate cube set algebra there are all standard set operations and many special operations on cubes. We define here only those used further in the paper. In the rest of this section, a lowercase letter denotes an element, and an uppercase letter denotes a combination set. Moreover, we use the short notation of combination set introduced in Subsection 3.1.

- *Addition*, *Subtraction*, and *Intersection*: These are standard set operations. $F + G$ is the union of F and G. $F + v$ is the union of F and $\{v\}$. $F - G$ is the difference between F and G. $F \cdot G$ is the intersection between F and G. E.g., when $F = \{a; b; bc\}$, $G = \{b; c\}$, and $v = a$, $F + G = \{a; b; c; bc\}$ and $G + v = \{a; b; c\}$, $F - G = \{a; bc\}$, and $F \cdot G = \{b\}$.
- *Multiplication*: $F \times G$ is a result of all possible concatenations of two cubes in F and G. E.g., when $F = \{a; b; ac\}$ and $G = \{ab; acd\}$, $F \times G = \{a\} \times \{ab\} + \{b\} \times \{ab\} + \{ac\} \times \{ab\} + \{a\} \times \{acd\} + \{b\} \times \{acd\} + \{ac\} \times \{acd\} = \{ab\} + \{ab\} + \{abc\} + \{acd\} + \{abcd\} + \{acd\} = \{ab; abc; acd; abcd\}$.
- *Division* and *Modulo*: Quotient F/v is obtained by extraction of those cubes from F that include variable v, and removal of v from the extracted cubes. Remainder $F\%v$ is obtained by extraction of those cubes from F that do not include variable v. E.g., when $F = \{ab; bc; c\}$ and $v = b$, $F/v = \{a; c\}$ and $F\%v = \{c\}$.
- *Subset0* and *Subset1*: $Subset0(F, v)$ is exactly the same operation as *Modulo*. $Subset1(F, v)$ is obtained by extraction of those cubes from F that include variable v. E.g., when $F = \{ab; bc; c\}$ and $v = b$, $Subset0(F, v) = \{c\}$ and $Subset1(F, v) = \{ab, bc\}$.
- *Restriction*: $Restrict(F, G)$ extracts the cubes from F such that the cube is a superset of at least one cube in G. E.g., when $F = \{ab; abc; bcd; d\}$ and $G = \{abc; bc\}$, $Restrict(F, G) = \{abc; bcd\}$.
- *Change operation*: $Change(F, v)$ changes all the cubes from F such that it removes v from the cubes including it and adds v to the cubes not including it. E.g., when $F = \{ab; abc; bcd; d\}$ and $v = b$, $Change(F, v) = \{a; ac; cd; bd\}$.
- *Stretch operation*: $Stretch(F)$ extracts the cubes from F such that a proper superset of the cube is not in F. E.g., when $F = \{ab; abc; bcd; d\}$, $Stretch(F) = \{abc; bcd\}$.
- *Permitsym operation*: $Permitsym(F, n)$ extracts the cubes from F such that the cube consists of less than or equal to n elements. E.g., when $F = \{ab; abc; bcd; d\}$ and $n = 2$, $Permitsym(F, n) = \{ab; d\}$.

- *Selective multiplication*: $F \times_P^N G$ are all concatenations of two cubes in F and G such that at least one element from set P is included in a cube from G, all elements from set P are included in a cube from F if they are included in a cube from G, and also no element from N is included in a cube from F if it is included in a cube from G. E.g., when $F = \{a; b; ac\}$ and $G = \{ab; acd\}$, $F \times_{\{a\}}^{\{c\}} G = \{a\} \times_{\{a\}}^{\{c\}} \{ab\} + \{b\} \times_{\{a\}}^{\{c\}} \{ab\} + \{ac\} \times_{\{a\}}^{\{c\}} \{ab\} + \{a\} \times_{\{a\}}^{\{c\}} \{acd\} + \{b\} \times_{\{a\}}^{\{c\}} \{acd\} + \{ac\} \times_{\{a\}}^{\{c\}} \{acd\} = \{ab\} + \{\} + \{abc\} + \{acd\} + \{\} + \{\} = \{ab; abc; acd\}$.

The result of multiplication and selective multiplication with an empty set is an empty set for both cases when F or G is an empty set. Selective multiplication is not commutative.

5.3 Implementation of feasible solutions generation

To generate the set of all feasible solutions we follow mainly the approach from [1]. Algorithms are given in Fig. 4.

```
1  def createProcessPlans():
2    for s in 1..S:
3      X[s] = getSequences(s)
4      for w in 1..W:
5        for r in 1..max(s):
6          O = findO(s,w,r)
7          if O > 0:
8            z = 0
9            for i in 1..I:
10             M = findM(s,w,r,i)
11             if M > 0: z = z + M
12           z = Change(z,O)
13           X[s] = (X[s]/O)×z + (X[s]%O)
14
15 def createComprehensivePlan():
16   X = 1
17   for s in 1..S:
18     X = X × X[s]
19
20 def createMachineInstances():
21   for i in 1..I:
22     MI[i] = 0
23     for j in instances(i)..1:
24       M = findMX(i,j)
25       MI[i] = (MI[i]+1) × {M}
26
27 def createPRplan():
28   for s in 1..S:
29     for w in 1..W:
30       for r in 1..max():
31         for i in 1..I:
32           M = findM(s,w,r,i)
33           if M > 0:
34             f = Subset1(X,M)
35             g = Restrict(f,MI[i])
36             X = (f − g) × MI[i] + X%M + g
37
38 def restrictToFactoryCapacity(fc):
39   Mall = 1
40   for i in 1..I:
41     Mall = Mall × (MI[i]+1)
42   Mng = Mall − Permitsym(Mall,fc)
43   X = X − Restrict(X,Mng)
44
45 def createFeasibleSolutions():
46 # X, X[s], and MI[i] are global variables
47   createElements()
48   createProcessPlans()
49   createComprehensivePlan()
50   createMachineInstances()
51   createPRplan()
52   restrictToFactoryCapacity(FC)
```

Fig. 4 The functions involved in generating feasible solutions

In Fig. 4, S denotes the number of all part types, W denotes the number of all operation types, I denotes the number of all machine types, max(s) is the length of the longest sequence of operations to produce part P_s, and instances(i) is the number of instances of machine type M_i. We use Python-style pseudocode, where the line with a different indent than the previous one starts a new block. Function getSequences(s) returns a set of cubes such that every cube corresponds to a sequence of operations encoded with O-variables. Function createElements() creates only those elements that are included into a system. Functions findO(), findM(), and findMX() return an element with the given indices, or 0 if such an element does not exist. For specifying the set of serial numbers, we disobey the Python syntax and write $1..S$ for the set of numbers $\{1,2,\ldots,S\}$.

Function createProcessPlans() creates process plans for all parts (as Eq. 2 for P_3 in Subsection 3.2). The algorithm iterates over all values of s, w, and r. If O_w is not the r-th operation in part P_s, this combination of indices is skipped for the sake of efficiency (line 7). Othewise, a cube is added for each machine M_i that can be used to complete operation O_w. This is achieved by creating a union of adequate machines first (lines 9-11) and then calculating a product with the

subset of those cubes from the partial result z which include the particular operation (line 13). Cubes from z which do not include $O^s_{w,r}$ are kept unchanged. Please note that if operation *Subset1* were used instead of *Division*, line 12 would be removed. Because efficiency would not increase significantly, we use mathematically more elegant solution with *Division* and *Modulo* [1].

Function createComprehensivePlan() implements a simple product of process plans for all parts. This is the point where ZBDD shows its strength: while the number of cubes in the result is a product of cube numbers of the individual process plans, the size of resulting ZBDD (i.e. the number of nodes) is about a sum of sizes of involved ZBDDs (it can be even less than the sum!).

Function createMachineInstances() creates viable factory configurations for each machine as described in Subsection 3.2. For example, if machine M_1 has instances $M_{1,1}$, $M_{1,2}$, and $M_{1,3}$ then viable factory configurations are $\{M_{1,1}\}$, $\{M_{1,1}, M_{1,2}\}$, and $\{M_{1,1}, M_{1,2}, M_{1,3}\}$. Functions createPRplan() and restrictToFactoryCapacity() are the same as given and explained in [1].

5.4 Implementation of scheduling

As described in Section 4, the proposed scheduling approach is an extension of a method for generation of feasible solutions. Its implementation is described formally with the algorithms given in Fig. 5. There, functions findW(), findR(), findB(), findS(), and findT() return an element with the given indices, or 0 if such an element does not exist.

The input to the scheduling is a global variable X, which is a single ZBDD representing the set of all feasible solutions. Function schedulingInit() creates sets tr, setP, and setN (lines 9-17), which are stored in global variables and used later in schedulingPhase 1. Furthermore, it initialises scheduling procedure by adding "waiting to start the first operation" tag to all parts (lines 18-21).

Function schedulingPhase1() utilises the Selective multiplication operation for generating all possible schedules. The result of a single call to selective multiplication is a set of all cubes such that one of the parts included in the cube is scheduled to start an operation. Sometimes, more than one part can start an operation simultaneously, and, thus, there is a loop (lines 27-29). Please note that selective multiplication does not remove anything from cubes, elements are only added, and thus operation Stretch removes all schedules where Rule TR1 is not maximally applied elegantly. Searching the *full state space* where a part that can be scheduled is not required to be scheduled, can be achieved by merely omitting the Stretch operation in line 30. However, this yields a problem which is harder to calculate, while rarely bringing a better solution. The next action made in scheduling phase 1 is removing schedules where an instance with index j is occupied instead of a free instance with a lower index (lines 31-36). Finally, for cubes containing both R-variable and W-variable with the same indices (these denote processes which have just started an operation), the corresponding W-variable and M-variable are removed, and also an appropriate G-variable is added (lines 37-52). Adding G-variables is optional. By omitting line 45, the algorithm will be much more efficient, but the result of scheduling will be the set of different feasible solutions instead of a number of different schedules (if more than one instance of some machines are allowed, then the result may include several cubes for a single feasible solution).

Function schedulingPhase2() is simpler, since it only decrements the second index of every T-variable. Function schedulingPhase3() looks for cubes with T-variables that have the second index equal to zero (this denotes that an operation has been completed). For such cubes, it removes the appropriate T-variable, R-variable, and B-variable, and marks the corresponding part with the "waiting to start the next operation" tag (line 85). Function removeSolutions checks for cubes without any R-variable and M-variable. They are solutions, and the function reports and then removes them from the working set of cubes (lines 97-98). Scheduling is running until the working set of cubes is not empty (lines 103-108). Indeed, the algorithm can be modified in such a way that the scheduling is stopped after the first solution set is found.

```
def schedulingInit():
  tr = setP = setN = 0
  for s in 1..S:
    for w in 1..W:
      for r in 1..max(s):
        for i in 1..I:
          M = findM(s,w,r,i)
          if M > 0:
            for j in 1..instances(i):
              f = Change(1,M); setP = setP+{M}
              E = findW(s,r); f = Change(f,E); setP = setP+E
              E = findMX(i,j); f = Change(f,E); setP = setP+E
              E = findR(s,t); f = Change(f,E); setN = setN+E
              E = findB(i,j); f = Change(f,E); setN = setN+E
              E = findS(s,r,i,j); f = Change(f,E)
              E = findT(s,time(w,i)); f = Change(f,E)
              tr = tr + f
            f = Subset1(X,M)
            X = X − f
            f = Change(f,findW(s,r))
            X = X + f

def schedulingPhase1(step):
  Y = X ×^{setP}_{setN} tr
  X = X · Stretch(X+Y)
  f = Y
  while f != 0:
    f = f ×^{setP}_{setN} tr
    Y = Y + f
  Y = Stretch(Y)
  for i in 1..I:
    for j in 2..instances():
      f = Subset1(Y,findB(i,j))
      Y = Y − f
      f = Subset1(f,findB(i,j−1))
      X = X + f
  for s in 1..S:
    for r in 1..max(s):
      f = Subset1(Y,findR(s,r))
      Y = Y − f
      g = Subset1(f,findW(s,r))
      if g != 0:
        f = f − g
        g = Change(g,findW(s,r))
        g = Change(g,addGelement(s,r,step))
        for w in 1..W:
          for i in 1..I:
            h = Subset1(g,findM(s,w,r,i))
            g = g − h
            h = Change(h,findM(s,w,r,i))
            g = g + h
      Y = Y + f + g
  X = X + Y

def schedulingPhase2(step):
  k = 1
  complete = False
  while complete == False:
    complete = True
    for s in 1..S:
      T = findT(s,k)
      if T > 0:
        complete = False
        f = Subset1(X,T)
        X = X − f
        f = Change(f,T)
        f = Change(f,findT(s,k−1))
        X = X + f
    k += 1

def schedulingPhase3(step):
  for s in 1..S:
    for r in 1..max(s):
      g = 0
      for i in 1..I:
        for j in 1..instances(i):
          f = Subset1(X,findT(s,0))
          f = Subset1(f,findR(s,r))
          f = Subset1(f,findS(s,r,i,j))
          X = X − f
          f = Change(f,findT(s,0))
          f = Change(f,findR(s,r))
          f = Change(f,findB(i,j))
          g = g + f
      g = Change(g,findW(s,r+1))
      X = X + g

def removeSolutions(step):
  f = X
  for s in 1..S:
    for r in 1..max(s):
      f = f % findR(s,r)
      for w in 1..W:
        for i in 1..I:
          f = f % findM(s,w,r,i)
  if f != 0:
    reportSolutions(f,step)
    X = X − f

def scheduling():
  step = 1
  schedulingInit()
  while X != 0:
    schedulingPhase1(step)
    schedulingPhase2(step)
    schedulingPhase3(step)
    removeSolutions(step)
    step += 1
```

Fig. 5 Implementation of scheduling

5.5 Heuristic approach

Even if the resulting ZBDD representing a huge combination set is small, a problem may appear to create or manipulate it efficiently. Thus, a heuristic is needed to restrict the state space. Luckily, ZBDDs are very suitable for such an approach. Here, we outline two possible heuristics, the other ones, and especially the multi-objective optimization, are left for further work.

The heuristic which we can introduce the easiest is an upper bound for the *total workload* (i.e. total machine time). A weight is assigned to M-variables such that the weight of an element $M_i^{s,w,r}$ represents the time needed to finish the operation O_w on machine type M_i. All other variables have zero weights. Then, the sum of weights of all elements in a cube is the total workload for the solution corresponding to this cube. In the presented algorithms, a single ZBDD is used to represent the set of all viable solutions and, thus, a single pass over its nodes is enough to bound the total workload for all solutions simultaneously. The algorithm is given in Fig. 6. It extracts those cubes from the given set of solutions that have a total workload less than or equal to the given bound. The presented algorithm uses C-syntax, and is a typical recursive algorithm for manipulation of BDDs. Basic knowledge about BDDs is required in order to understand it. Please compare it with the algorithms given in [7] and [4]. The total workload bound is best used during a generation of feasible solutions. It should be added to the algorithm for construction of a comprehensive process plan (i.e. immediately after line 18 in Fig. 4).

```
1  ZBDD boundMachineTime(ZBDD f, unsigned int n)
2  {
3    ZBDD e,t,r;
4    unsigned int w;
5
6    if (n == 0) return emptySet;
7    if (isTerminalNode(f)) return f;
8    if (r = findInCache(f,n)) return r;
9    e = boundMachineTime(getElseSuccessor(f),n);
10   if (isMvariable(getTopLabel(f)) {
11     w = getWeight(getTopLabel(f));
12     if (n < w) n = 0; else n = n − w;
13   }
14   t = boundMachineTime(getThenSuccessor(f),n);
15   r = foaNode(getTopLabel(f),e,t);
16   storeInCache(f,n,r);
17   return r;
18 }
```

Fig. 6 A ZBDD-based algorithm for bounding total workload

Another interesting heuristic is a *makespan limit*. It can be implemented by using the same weights assigned to M-variables as already described. A sum of weights is calculated separately for each part and each machine type. If any part or machine type requires more time than the makespan limit to complete, the corresponding cube is removed from the set of solutions. Indeed, if more than one instance of a particular machine type is allowed, then the limit for this machine type should be multiplied with the number of instances. Again, a single pass recursive algorithm on the ZBDD representing the set of solutions is an efficient way to consider the given limit for makespan. We refer the interested reader to check the available source code. In contrast to the total workload bound, applying the makespan limit is not reasonable during the generation of feasible solutions. It is the most beneficial if this operation is taken before every scheduling step (i.e. immediately after line 103 in Fig. 5).

5.6 The element order in the ZBDD

The elements in ZBDD are ordered, and the order has a critical impact on its size. Therefore, we report here the order used in our implementation, which has been determined experimentally (the element in a node is considered smaller than the elements in its successors):

$B_{1,j} < B_{2,j} < ... < T_n^1 < W^{1,1} < R^{1,1} < S_{i,j}^{1,1} < O_{w,1}^1$ and $M_i^{1,w,1}$ (interleaved) $< W^{1,2} < R^{1,2} < S_{i,j}^{1,2} < O_{w,2}^1$ and $M_i^{1,w,2}$ (interleaved) $< ... < T_n^2 < W^{2,1} < R^{2,1} < S_{i,j}^{2,1} < O_{w,1}^2$ and $M_i^{2,w,1}$ (interleaved) $< ... < MX_{i,1} < MX_{i,2} < ... < FS_n < 1$

6. Results and discussion

First, the generation of all feasible solutions is observed. In Table 2 and Table 3, we report some details for the factory in Fig. 1 that produces eight different part types and has eight machine types specified. The production system involves 15 types of operations. The comprehensive process plan for this system contains 32,878,483,200,000 cubes, and is represented by our implementation with only 195 ZBDD nodes. These results can be compared with results from [1]. There, for the factory with capacity 8 and no restrictions to total workload, the authors report exactly the same number of cubes in a comprehensive process plan and the process and resource plan, but they needed 280 ZBDD nodes and 19,443 ZBDD nodes (our result is 13,072 ZBDD nodes) to represent them, respectively. Without restrictions, all the results are computed in less than a second, and, with a restriction on the maximal total workload, in just over two seconds. Thus, the exact computation times are irrelevant and not shown. When bounding the total workload to 100, the comprehensive process plan contains 245,837,448 cubes and is represented by our implementation with 1,744 ZBDD nodes. All the results have been obtained on a 3.40 GHz Intel Core i7-4770 processor with 32 GB of RAM memory.

Table 2 Size of a set of cubes representing the process and resource plans for the factory from Fig. 1

	Process and resource plan (without restrictions to total workload)		Process and resource plan (max. total workload = 100)	
Factory capacity	Number of cubes	ZBDD nodes	Number of cubes	ZBDD nodes
3	169,984	274	0	1
4	284,701,184	3,000	18488	695
5	41,207,077,120	8,193	1,526,572	5,965
6	1,365,249,188,224	11,729	26,001,900	16,898
7	14,411,349,910,656	12,925	180,702,952	24,620
8	65,501,043,610,240	13,072	642,479,776	26,190
9	164,241,617,343,104	13,041	1,398,613,308	26,181
10	272,777,626,896,512	13,011	2,158,556,924	26,162
11	343,383,824,875,136	12,994	2,620,121,648	26,145
12	364,877,105,061,888	12,985	2,747,814,784	26,136

With the help of the heuristic, we were able to count all the solutions for all the systems. The optimal lower bounds for total workload and makespan were determined experimentally by repeating the calculation with different parameters. Table 3 reports the number of all feasible solutions with the given maximal total workload, of feasible solutions yielding to the adequate schedules, and of adequate schedules for different combinations of parameters, respectively. For this system, there exist 65,501,043,610,240 feasible solutions if no restrictions on total workload and factory size are used. For scheduling, the algorithm without examining the full state space was used (see Subsection 5.4). If full state space is examined, the calculation takes much longer, and we are not able to complete it in most cases. However, in the case when the makespan is fixed to 15, and the total workload is fixed to 95, examining the full state space is feasible, and it brings an interesting result: two feasible solutions out of 9,262,892 possible ones yield to 704 adequate schedules, none of which can be found without examining the full state space.

Table 3 Results for the factory with eight parts and eight machine types (factory size is required to be exactly eight machine instances; the algorithm without examining full state space is used)

	Total workload					
	87	88	90	91	95	96
	Number of feasible solutions restricted to the total workload					
	24	672	28,112	116,376	9,262,892	22,474,516
Makespan	Number of feasible solutions yielding adequate schedules/Number of adequate schedules					
15	0/0	0/0	0/0	0/0	0/0	2/36
16	0/0	0/0	0/0	1/4	236/3,120	451/5,151
17	0/0	0/0	4/78	30/422	3,350/44,181	6,525/91,419
18	0/0	7/158	180/5,047	674/14,965	27,658/412,204	52,895/721,713
24	4/54	104/4,074	3,220/135,960	12,665/ 517,086	out of memory	out of memory

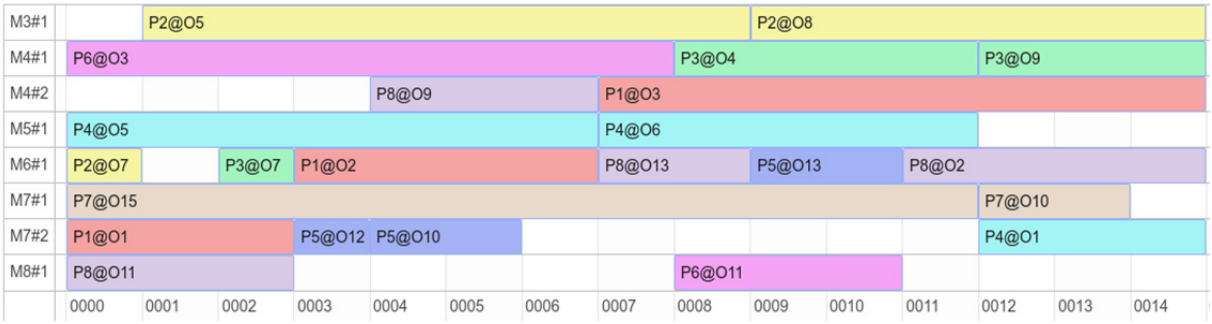

Fig. 7 Solution with makespan 15 and total workload 95

Determining the optimal solution depends on the selected goals, i.e. it is better to have a schedule with a shorter makespan, with a smaller total workload, or we want to minimise a function combining these and other objectives as well. Fig. 7 gives a Gantt chart for a solution with makespan 15 and total workload 95, which is quite hard to find and is optimal in many aspects.

To evaluate the potential of the presented method, we implemented benchmarks from [9] and [10] which have been used recently to compare different genetic and swarm algorithms [11, 12]. Without examining the full state space, we have been able to count adequate feasible solutions and schedules for all systems (see Table 4). We confirm that the most interesting solutions to these problems have been already found and reported. Still, we can present some original schedules. For example, the problem 4×5 can be solved equally well with only four machines (Fig. 8), and the problem 10×7 can be solved with makespan equal to 11 and total workload 61 (Fig. 9).

Table 4 Number of feasible solutions and schedules for benchmarks from [9] and [10] found without the full state space examination; to optimise the workload of the critical machine, maximal possible factory size was required

Problem	Makespan	Total workload	Number of feasible solutions	Number of schedules	Time to find all schedules (s)
4×5	11	32	4	4	0.36
8×8	14	77	3	4	3.14
	16	73	1	9	2.17
10×7	11	61	138	19,176	77.34
	11	62	441	44,480	205.38
	12	60	46	26,576	74.71
10×10	7	42	692	81,953	152.87
	8	41	704	818,568	630,35
15×10	11	91	60	907,570	9,003,50

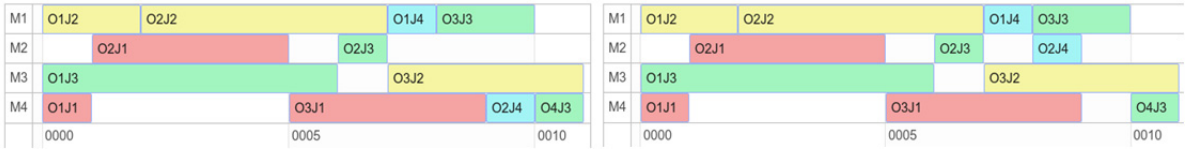

Fig. 8 There exist 4 schedules (18 in the case of the full state space exploration) based on two different feasible solutions to solve problem 4×5 with only 4 machines such that makespan is 11 and the total workload is 32

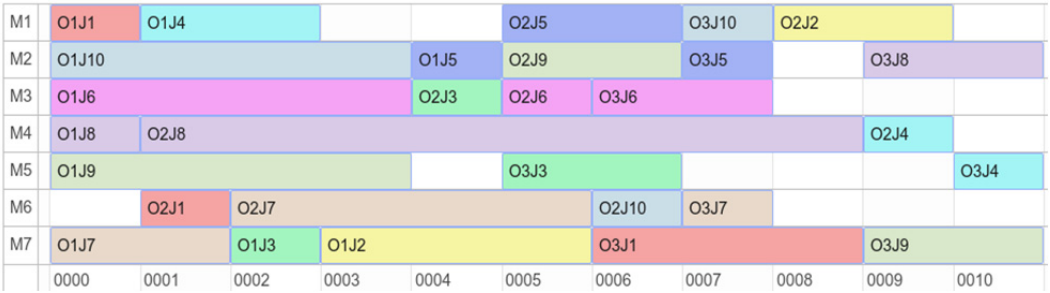

Fig. 9 Solution for problem 10×7 with makespan 11 and total workload 61

7. Conclusion

This paper proposes a novel method for generating and counting solutions of a flexible job shop scheduling problem. Feasible solutions are represented as cubes of a combination set, and the algorithm is implemented as a sequence of operations in unate cube set algebra. The efficiency of the approach is achieved by representing combination sets with ZBDDs, a data structure used typically in various methods for formal verification of systems.

In the proposed algorithm, process planning and scheduling are two sequential steps like in a typical non-linear process planning. However, in contrast to all other existing methods, the huge set of *all* feasible solutions is created in the first step. Although this looks like an unreasonable and awkward solution, ZBDDs somehow help to complete even the relatively large problems successfully. Moreover, in comparison to genetic algorithms, particle swarm optimization algorithms, and other evolutionary approaches, the obtained results are very comprehensive. For example, we have found out that the result reported in [13] is simply wrong, because for benchmark 8×8 no solution with makespan equal to 15 and total workload equal to 73 exists!

In further work, the presented algorithms can be adapted to handle additional constraints, such as the production of parts in a required order, production of several instances of each part type, and the last acceptable part delivery time. Moreover, dynamic rescheduling can be introduced to take care of eventual machine breakdowns or urgent new orders.

Acknowledgement

This work was supported in part by the Slovenian Research Agency through the Research Program Advanced Methods of Interaction in Telecommunication under Grant P2-0069.

References

[1] Takahashi, K., Onosato, M., Tanaka, F. (2014). Comprehensive representation of feasible combinations of alternatives for dynamic production planning using zero-suppressed binary decision diagram, *Journal of Advanced Mechanical Design, Systems, and Manufacturing*, Vol. 8, No. 4, JAMDSM0061, doi: 10.1299/jamdsm.2014jamdsm0061.

[2] Takahashi, K., Onosato, M., Tanaka, F. (2015). A solution method for comprehensive solution candidates in dynamic production planning by zero-suppressed binary decision diagrams, *Transactions of the Institute of Systems, Control and Information Engineers*, Vol. 28, No. 3, 107-115, doi: 10.5687/iscie.28.107.

[3] Phanden, R.K., Jain, A., Verma, R. (2011). Review on integration of process planning and scheduling, In: Katalinic, B., (ed.), *DAAAM International Scientific Book 2011*, DAAAM International, Vienna, Austria, 593-618, doi: 10.2507/daaam.scibook.2011.49.

[4] Minato, S.-I. (1993). Zero-suppressed BDDs for set manipulation in combinatorial problems, In: *Proceedings of 30th ACM/IEEE Design Automation Conference*, Dallas, Texas, USA, 272-277.

[5] Minato, S.-I. (2001). Zero-suppressed BDDs and their applications, *International Journal on Software Tools for Technology Transfer*, Vol. 3, No. 2, 156-170.

[6] Jensen, R.A., Lauritzen, B.L., Laursen, O. (2004). Optimal task graph scheduling with binary decision diagrams, from *https://pdfs.semanticscholar.org/0fe3/36b2e5e77df4ececfa749d28752694976636.pdf?_ga=2.231505503.177204796.1540581701-1029763941.1533058815*, accessed May 5 2018.

[7] Brace, K.S., Rudell, R.L., Bryant, R.E. (1990). Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '90)*, New York, USA, 40-45. doi: 10.1145/123186.123222.

[8] Meolic, R., (2012). Biddy – A multi-platform academic BDD package, *Journal of Software*, Vol. 7, No. 6, 1358-1366, doi: 10.4304/jsw.7.6.1358-1366.

[9] Kacem, I., Hammadi, S., Borne, P. (2002). Pareto-optimality approach for flexible job-shop scheduling problems: Hybridization of evolutionary algorithms and fuzzy logic, *Mathematics and Computers in Simulation,* Vol. 60, No. 3-5, 245-276, doi: 10.1016/S0378-4754(02)00019-8.

[10] Kacem, I., Hammadi, S., Borne, P. (2002). Approach by localization and multiobjective evolutionary optimization for flexible job-shop scheduling problems*, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews),* Vol. 32, No. 1, 1-13, doi: 10.1109/TSMCC.2002.1009117.

[11] Chaudry, I.A., Khan, A.M., Khan, A.A. (2013). A genetic algorithm for flexible job shop scheduling, In: *Proceedings of the World Congress on Engineering Vol. I*, London, U.K., 703-708.

[12] Yu, M.R., Yang, B., Chen, Y. (2018). Dynamic integration of process planning and scheduling using a discrete particle swarm optimization algorithm, *Advances in Production Engineering & Management*, Vol. 13, No. 3, 279-296, doi: 10.14743/apem2018.3.290.

[13] Zhang, H., Gen, M. (2005). Multistage-based genetic algorithm for flexible job-shop scheduling problem, *Journal of Complexity International*, Vol. 11, 223-232.