A decomposition Approach for solving the JSP using multi-shot ASP*

Mohammed M. S. El-Kholany$^{1[0000-0002-1088-2081]}$, Martin Gebser$^{1,2[0000-0002-8010-4752]}$, and Konstantin Schekotihin$^{1[0000-0002-0286-0958]}$

Alpen-Adria-Universität Klagenfurt, Klagenfurt, Austria {mohammed.el-kholany, martin.gebser and konstantin.schekotihin}@aau.at
Technische Universität Graz, Graz, Austria
mgebser@ist.tugraz.at

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

Keywords: Job-shop Scheduling Problem · Answer Set Programming · Decomposition.

1 Introduction

In today's competitive markets, manufacturers have to respond quickly to orders and meet shipping dates committed to the customers. This requires the ability to schedule production activities to use the available scarce resources efficiently. Effective scheduling techniques are essential in complex manufacturing systems such as semiconductor manufacturing. In general, scheduling operations is one of the most critical issues in the planning and managing of the manufacturing processes[24].

One of the most challenging scheduling problems is the Jop-shop Scheduling Problem (JSP). A set of jobs needs to be processed on a set of machines while optimizing a performance indicator, minimizing makespan, the time needed to complete all the jobs, or tardiness, i.e., the summation of the delays in executing all jobs according to their deadlines. Each job has a set of consecutive operations; each operation requires exactly one machine; machines are continuously available and can process only one operation at a time. The main purpose is to determine the sequence of the operations on each machine to optimize a performance indicator. The JSP is an NP-complete combinatorial optimization problem where obtaining the optimal solution is challenging regardless of the problem scale[15, 19, 20]. For instance, benchmark (FT10) consists of 10 jobs and 10 machines took researchers roughly 20 years to find the optimal solution[3, 26].

In reality, the number of operations is reaching thousands of operations. Therefore, the exact methods Answer Set Programming (ASP), Constraint Programming and Branch and Bound cannot find the optimal solutions in a reasonable time[6, 22, 14]. Since the performance of the exact methods will be hardly satisfactory, several studies have presented one of the most effective methods for solving the JSP, which is decomposition [26]. The decomposition aims to split the problem into a series of subproblems

* Supported by organization x.

based on a particular decomposition policy and then solve each part separately and obtain the final solution by integrating all of these solutions. Several strategies have been proposed to decompose the JSP, and each varies according to the problem features. More specifically, there is no evidence that one of the introduced strategies is the best for solving all JSP(s) [21].

In this study, we aim to introduce a new decomposition strategy for solving the JSP. Our study proposes a new policy to efficiently divide the problem into subproblems, aiming to reach a near-optimal solution. Then each part is solved using an ASP scheduler-based, and the solutions are integrated to obtain the solution of the whole problem. We develop and implement our model using ASP, which is considered as one of the most popular paradigms for knowledge representation and reasoning, especially in combinatorial optimization problems [2].

The paper is organized as follows. The following section shows the most related work and which decomposition techniques have been introduced in the past, followed by the problem formulation section illustrates the problem in detail. Section 4 presents our proposed model, which shows the decomposition techniques we used and describes one in more detail. In addition, we present the schedule model using Answer Set Programming. Our model is tested by performing extensive experiments on a set of benchmark instances in Section 5. Section 6 summarizes this work and provides some ideas to extend the current work.

2 Literature Review

This section will review some of the work that has applied the decomposition approach for solving scheduling problems. As mentioned in the previous section, no particular decomposition method has proved its efficiency in solving all scheduling problems. However, many articles have introduced efficient decomposition methods for solving particular scheduling problems with specific features. The decomposition idea for solving JSP has been initially suggested by introducing a *Shifting bottleneck* (SB) procedure which decomposes the problem into parts; each part contains only one machine [3]. At each iteration, the bottleneck machine is determined from a set of unsequenced machines and then scheduled. Afterward, all the previously established sequences are locally reoptimized and iterate until all the machines have been sequenced. The idea of SB had been later improved by integrating an optimization algorithm for solving the one-machine problem with the delayed precedence constraints procedure [4].

Another study investigated the performance of a new decomposition procedure based on SB in the Job-shop and flow shop with different levels of bottleneck machines. The computational experiments showed that the proposed method obtained better solutions in a shorter time than for the problems in which the machine's workload is identical. Other different methods have been developed to decompose the JSP. For instance, a rolling horizon heuristic has been presented [23] to solve a large-scale job shop. The authors have decomposed the problem and solve each independently while minimizing the total weighted tardiness. They tested their model on a set of benchmark instances, and the results showed that the proposed model is superior to large instances.

The rolling horizon procedure has been extended by constructing a prediction model to obtain the scheduling characteristics values, including the information of the bottleneck jobs. The obtained information aided in decomposing the problem efficiently. In addition, they have proposed a genetic algorithm to solve each sub-problem. In order to evaluate the performance of the proposed model, they performed numerical computational experiments that showed the effectiveness of the model [20]. A decomposition-based hybrid optimization algorithm has been introduced for the JSP, where the total weighted tardiness is minimized. A Simulated Annealing is used to define subproblems iteratively and then is solved by a Genetic Algorithm. They have developed a fuzzy system that provides information about bottleneck jobs. This information is used to guide the process of subproblem-solving to promote the optimization efficiency [26]. The numerical computational results showed that the proposed algorithm is effective for large-scale scheduling problems.

A decomposition method based on bottleneck machines has been presented for solving the JSP. The proposed model decomposes the problems into subproblems and then detects the multi-bottleneck machines using a critical path method. The information of the bottleneck machines has been employed to improve the solution quality by splitting the operations into bottleneck operations and non-bottleneck operations, where the bottleneck operations are scheduled by the Genetic Algorithm and the non-bottleneck operations are scheduled by dispatching rules [25].

From the literature, we can find that most of the related work focused on applying the decomposition approach while minimizing the total tardiness, and the number of work tackled large-scale instances to minimize the makespan is relatively low. This paper proposes different decomposition strategies for solving the JSP with larger instances using the ASP model while minimizing the makespan.

3 Preliminaries

Answer Set Programming (ASP) is a declarative programming language for solving hard combinatorial optimization problems. ASP has become an established paradigm for knowledge representation and reasoning. ASP has proved its efficiency for solving the combinatorial optimization problem in different applications such as bioinformatics[12, 18], databses[5] and scheduling and industrial applications [10, 9, 13, 7].

A logic program is a finite set of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \tag{1}$$

where a_i is an atom for $0 \leq i \leq n$ and "$\sim$" is a default negation. If n quals to 0, then a rule (1) is a fact. If a_0 is omitted, the rule is an integrity constraint. An *atom* is an expression of the form $p(t_1, \ldots, t_l)$, where p is a predicate and $t_1, \ldots, t_l$ are *terms*. Each term can be a variable or a constant. A *literal* is either an atom or its negation. Given a rule r of the form (1), the set $H(r) = a_0$ denotes the *head* atom and the set $B(r) = B^+(r) \cup B^-(r) = \{a_1, \ldots, a_m\} \cup \{\sim a_{m+1}, \cdots \sim a_n\}$ is the *body* of r where $B^+(r)$ and $B^+(r)$ represent the positive and negative body literals, respectively.

Multi-shot Solving: Multi-shot solving ASP allows solving logic programs dynamically in sequential order. This can be manipulated using APIs implementation via an imperative programming language. Such programming language is used to control the grounding and solving processes and allows the usage of external variables that could be set to $True$ or $False$ to control some logic rules. The main advantage of *multi-shot* solving is to avoid the re-grounding and exploit conflicts learned over time.

ASP Difference Logic: *clingo*[DL] is an extension of the input language *clingo* by theory atoms representing difference constraints [16, 17]. Difference constraints are defined by specific constraint atoms of the form $\texttt{\&diff}\{u - v\} \leq k$ where u and v are terms which are interpreted as integer variables and k is an integer constant. *clingo*[DL] provides the following extension of the normal form (1):

$$\texttt{\&diff}\{u - v\} \leq k \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n.$$

This rule shows that whenever the body is $True$ the inequality constraints must be satisfied.

4 Problem Formulation

The Job-shop Scheduling Problem (JSP) has been a complex and combinatorial optimization problem since the 1950s, and it was shown to be NP-hard. In the JSP, there are a set of jobs to be processed on a set of machines. The number of jobs is n, and the number of machines is m. Each job J_i contains a chain $O_{i,1}, O_{i,2}, ..., O_{i,m}$ of operations must be executed on an known order. The routings of the operations are deterministic and known a priori, as are the processing times of each operation on each machine. We consider here the problem of minimizing the total completion time (makespan). The task of the scheduling is to determine the starting time for each operation while optimizing the makespan. The basic assumptions are as follows:

1. The processing time of each operation is fixed.
2. Machine breakdown preemption of the operations are not allowed.
3. Each machine can process only one operation at a time.
4. Each operation can be executed by only one machine.
5. A machine cannot process more than one operation at a time.
6. The jobs and the machines are available at time 0.

4.1 Modelling JSP with Hybrid ASP

ASP has been applied to solve different scheduling problems. For example, ASP is used to solve scheduling problems in the healthcare systems [11, 8]. One of these studies aimed to assign nurses to shifts according to some constraints[11]. In addition, ASP has been proposed to solve a train scheduling problem while minimizing the train delay[1]. One of the limitations of ASP mentioned in these studies is the grounding issues while solving a large number of instances. One of the recent works has presented a multi-shot solving for scheduling problem [14]. This paper aims to use different decomposition methods to solve the JSP using ASP with Difference Logic.

Listing 1. Problem instance

```
 1  operation(1,1).  operation(1,2).  operation(1,3).
 2  operation(2,1).  operation(2,2).  operation(2,3).
 3  operation(3,1).  operation(3,2).  operation(3,3).

 5  pro(1,1,9).  pro(1,2,3).  pro(1,3,12).
 6  pro(2,1,4).  pro(2,2,6).  pro(2,3,2).
 7  pro(3,1,4).  pro(3,2,3).  pro(3,3,5).

 9  assign(1,1,2).  assign(1,2,3).  assign(1,3,1).
10  assign(2,1,3).  assign(2,2,2).  assign(2,3,1).
11  assign(3,1,1).  assign(3,2,3).  assign(3,3,2).
```

Problem Instances. In order to encode the problem using ASP, we should define a set of predicates that represents the problem instances. For instance, the operations are encoded with the predicate `operation/2` where the first variable denotes the job number and the second is the operation number, see lines 1-3 in Listing 1. The time needed to finish an operation is represented by the predicate `pro/3` in which the operation is determined by the first two terms and the third represents the processing time in lines 5-7. For the machine assignment, the predicate `assign/3` shows that a particular operation is executed by a machine in lines 9-11 For instance, *operation(1,1)* is processed by machine 2.

4.2 Decomposition strategies

As we mentioned in the previous sections, it is hard to solve the JSP and reach a near-optimal solution in a reasonable time, especially when the number of operations to be processed is high. The decomposition approach has been widely used to solve the JSP by many researchers and proved its efficiency.

This section will mention 4 different decomposition strategies that we developed to split the problem into parts, describe one of them in detail, and show the encoding. The decomposition strategies can be classified into two categories; time-based and machine-based. The time-based decomposition aims to prioritize the operations based on the processing time. However, machine-based decomposition methods consider the processing time of the operations and the machine's workload. The main idea behind the decomposition procedure is to find a criterion that ranks the operations to assign in the proper time window to obtain higher-quality schedules without violating the precedence constraints. Earliest Starting Time (EST) and Most Total Work Remaining (MTWR) procedures have been applied to rank the order the operations. The decomposition procedures will be shown as follows:

EST Time-based: it ranks the operations based on calculating the earliest possible starting time for each operation. It is calculated by aggregating its predecessor(s) processing time. The operation with smaller EST will be assigned to a time window before the others with greater EST.

Listing 2. Pre-decomposition

```
1  #const numTW =2.
2  timeWindows(1..numTW).

4  job(Job) :- operation(Job, MachNum).
5  mach(MachNum) :- assign(Job, Step, MachNum).

7  numJobs(Job)   :- job(Job), not job(Job + 1).
8  numMach(MachNum) :- mach(MachNum), not mach(MachNum + 1).

10 numOper(M1) :- numJobs(Job), numMach(MachNum),
11                M1 = Job * MachNum.

13 numOperTW(M2) :- numOper(M1), M2 = (M1 + numTW - 1) / numTW.
```

***MTWR Time-based**:* the operation rank is determined based on the work remaining of a job to be completed. The operation belongs to a job with a high remaining processing time, will be assigned earlier to a time window.

***EST Machine-based**:* it applies the same idea as EST Time-based. However, the bottleneck machine is taken into account throughout the decomposition process. More specifically, the operation with smaller EST and executed by a bottleneck machine will be assigned earlier to a time window.

***MTWR Machine-based**:* we calculate the MTWR of each operation, and the operation with the Largest MTWR and processed by a bottleneck machine will be assigned earlier to a time window.

In this study, we will describe in detail the decomposition of the operations based on the **EST Time-based**. We have split the decomposition encoding into two parts; the first part is for the pre-decomposition phase, which is in Listing 2. The first two lines are to determine the number of time windows, which is two in this example. The lines 7-8 are to calculate the total number of jobs and machines of a particular instance, respectively. Therefore, the total number of operations to be scheduled is computed by the rule in lines 10-11. In line 13, the number of operations assigned to a time window is determined.

Listing 3 shows the decomposition using *EST Time-based* strategy. The first two rules in the lines 1-6 calculate the earliest possible start time of each operation, where the EST for the first operation of all jobs are 0 in line 1. The second rule calculates the other operations by aggregating the processing time of their predecessors.

We prioritize the operations based on the estimated starting time, calculating an index for each operation. The operation with a shorter estimated start time will get a smaller index than the others with a higher estimated starting time. If the estimated starting time between two operations or more is similar, we check the processing time; the higher the processing time, the smaller index, and, therefore, the higher priority to be assigned earlier. If the processing time is equal, we look at the operation number

Listing 3. EST-decomposition

```
1   est(Job, 1, 0) :- operation(Job, 1).

3   est(Job, Step + 1, ST + PT) :-
4                     est(Job, Step, ST),
5                     pro(Job, Step, PT),
6                     operation(Job, Step + 1).

8   index(Job, Step, N) :- est(Job, Step, ST),
9                          pro(Job, Step, PT),
10              N = #count{Job1, Step1 :
11                         est(Job1, Step1, ST1),
12                         pro(Job1, Step1, PT1),
13                        (ST1, PT1, Step1, Job1) <
14                        (ST, PT, Step, Job)}.

16  assignTW(Job, Step, (N+M) / M) :- index(Job, Step, N),
17                                    numOperTW(M).
```

Listing 4. Time Window Assignment

```
assignTW(1,1,1).      assignTW(1,2,2).
assignTW(2,1,1).      assignTW(1,3,2).
assignTW(2,2,1).      assignTW(2,3,2).
assignTW(3,1,1).      assignTW(3,3,2).
assignTW(3,2,1).
```

and then the job number. This strategy is encoded in lines 8-14. The fourth rule in lines 16-17 is to assign each operation to a time winodw.

After running the decomposition encoding, we described above; we get an assignment of each operation to a TW. This assignment is represented by a set of atoms shown in the Listing 4. We can see that the operations operations $\{O_{1,1}, O_{2,1}, O_{2,2}, O_{3,1}, O_{3,2}\}$ are assigned to the first TW and the rest to the second TW.

4.3 Multi-shot with JSP

This section will show how to solve the problem dynamically, where the output of each iteration is an input to the next and merging them to obtain the solution of the whole problem. In the first step, we consider the output of the decomposition phase, which is the assignment of the operations to TW as facts. For the multi-shot solving, the encoding is split into two parts; the base, which is run and grounded one time at the beginning of the optimization process presented in Listing 5. The first rule is to determine the logical sequence between operations of same job in lines 3-6 which is represented in predicate `seqL/3`. On the other hand, Lines 8-11 identify the sequence of the operations assigned to the same machine with predicate `sameMach/4`.

Listing 5. base-prog

```
#program base.

seqL((Job1, Step1), (Job1, Step2), PT1)  :-
                  pro(Job1, Step1, PT1),
                  pro(Job1, Step2, PT2),
                  Step2 = Step1 + 1.

sameMach(Job1, Step1, Job2, Step2) :-
                  assign(Job1, Step1, MachNum),
                  assign(Job2, Step2, MachNum),
                  Job1 < Job2.
```

The next iteration is to optimize each TW by solving the subproblem(t) where t refers to the current TW we aim to optimize. The model starts to schedule and optimize the first TW and after a fixed amount of time, the solver stops and obtain the makespan of the current TW and the execution time of each operation represented by an atom `startTime((Job, Step), ST, t)` where the first term is an operation, the second is the starting time and the last is the current time window. The solver moves to the next TW, where the starting time of the scheduled operations obtained from the first TW is sent entirely as an input to the next TW. Listing 6 handles the sequence between the operations in different cases. The first rule in lines 3-8 derives a new atom with a predicate `seq/4` represents the logical sequence between two operations of a same job. In this rule, the first term of the head is the predecessor of a particular operation followed by the successor, the third term defines the minimum waiting time between the starting of the both, and the last term defines the TW of the successor. The second and third rules handle the case of two operations assigned to the same machine and in the same TW. On the other hand, if two operations are assigned to the same machine and different TW, the fourth and fifth rules ensure that the operation assigned to the earlier TW will be executed before the other is assigned to the subsequent TW. For instance, the fourth rule assumes that the `operation (Job1, Step1)` is assigned to an earlier TW than `operation (Job2, Step2)`.

The third part of the scheduler is shown in Listing 7 that handles the difference constraints between the operations. The first rule in lines 1-2 ensures that all the operations will be processed starting from time 0. The second and the third rule fix the execution time of the operations scheduled in the previous TW. The fourth rule aimed to avoid overlapping the operations that either belongs to the same job or are assigned to the same machine. The fifth rule ensures that the ending time of all operations will not exceed the bound variable, which is the makespan in our case.

4.4 A Subsection Sample

Please note that the first paragraph of a section or subsection is not indented. The first paragraph that follows a table, figure, equation, etc., do not need, either.

Subsequent paragraphs, however, are indented.

Sample Heading (Third Level) Only two levels of headings should be numbered. Lower level headings remain unnumbered; they are formatted as run-in headings.

Sample Heading (Fourth Level) The contribution should contain no more than four levels of headings. Table 1 gives a summary of all heading levels.

Table 1. Table captions should be placed above the tables.

Heading level	Example	Font size and style
Title (centered)	**Lecture Notes**	14 point, bold
1st-level heading	**1 Introduction**	12 point, bold
2nd-level heading	**2.1 Printing Area**	10 point, bold
3rd-level heading	**Run-in Heading in Bold.** Text follows	10 point, bold
4th-level heading	*Lowest Level Heading.* Text follows	10 point, italic

Displayed equations are centered and set on a separate line.

$$x + y = z \tag{2}$$

Please try to avoid rasterized images for line-art diagrams and schemas. Whenever possible, use vector graphics instead (see Fig. 1).

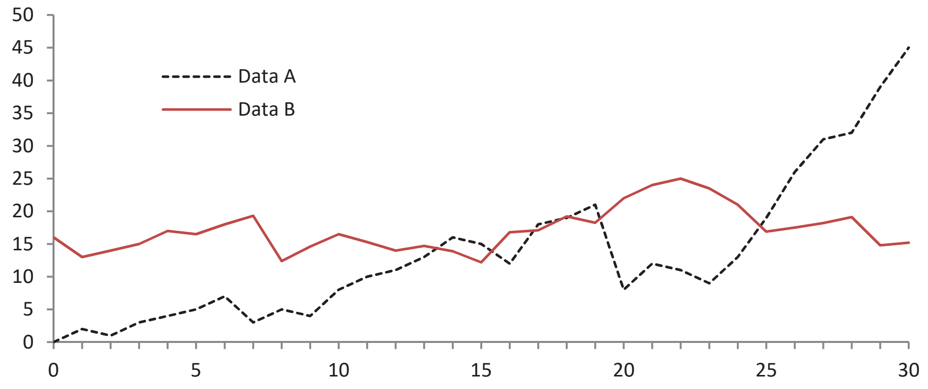

Fig. 1. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

Theorem 1. *This is a sample theorem. The run-in heading is set in bold, while the following text appears in italics. Definitions, lemmas, propositions, and corollaries are styled the same way.*

Proof. Proofs, examples, and remarks have the initial word in italics, while the following text appears in normal font.

For citations of references, we prefer the use of square brackets and consecutive numbers. Citations using labels or the author/year convention are also acceptable. The following bibliography provides a sample reference list with entries for journal articles [**?**], an LNCS chapter [**?**], a book [**?**], proceedings without editors [**?**], and a homepage [**?**]. Multiple citations are grouped [**?**,**?**,**?**], [**?**,**?**,**?**,**?**].

References

1. Abels, D., Jordi, J., Ostrowski, M., Schaub, T., Toletti, A., Wanko, P.: Train scheduling with hybrid asp. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 3–17. Springer (2019)
2. Abseher, M., Gebser, M., Musliu, N., Schaub, T., Woltran, S.: Shift design with answer set programming. Fundamenta Informaticae **147**(1), 1–25 (2016)
3. Adams, J., Balas, E., Zawack, D.: The shifting bottleneck procedure for job shop scheduling. Management science **34**(3), 391–401 (1988)
4. Balas, E., Lenstra, J.K., Vazacopoulos, A.: The one-machine problem with delayed precedence constraints and its use in job shop scheduling. Management Science **41**(1), 94–109 (1995)
5. Caniupán, M., Bertossi, L.: The consistency extractor system: Answer set programs for consistent query answering in databases. Data & Knowledge Engineering **69**(6), 545–572 (2010)
6. Daneshamooz, F., Fattahi, P., Hosseini, S.M.H.: Mathematical modeling and two efficient branch and bound algorithms for job shop scheduling problem followed by an assembly stage. Kybernetes (2021)
7. Dodaro, C., Galatà, G., Khan, M.K., Maratea, M., Porro, I.: Operating room (re) scheduling with bed management via asp. arXiv preprint arXiv:2105.02283 (2021)
8. Dodaro, C., Galatà, G., Maratea, M., Porro, I.: An asp-based framework for operating room scheduling. Intelligenza Artificiale **13**(1), 63–77 (2019)
9. Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F., Shchekotykhin, K.: Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). Theory and Practice of Logic Programming **16**(5-6), 653–669 (2016)
10. Dodaro, C., Leone, N., Nardi, B., Ricca, F.: Allotment problem in travel industry: A solution based on asp. In: International Conference on Web Reasoning and Rule Systems. pp. 77–92. Springer (2015)
11. Dodaro, C., Maratea, M.: Nurse scheduling via answer set programming. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 301–307. Springer (2017)
12. Erdem, E., Oztok, U.: Generating explanations for biomedical queries. Theory and Practice of Logic Programming **15**(1), 35–78 (2015)
13. Fabricius, F., De Bortoli, M., Selmair, M., Reip, M., Steinbauer, G., Gebser, M.: Towards asp-based scheduling for industrial transport vehicles. In: Joint Austrian Computer Vision and Robotics Workshop (2020)
14. Francescutto, G., Schekotihin, K., El-Kholany, M.M.: Solving a multi-resource partial-ordering flexible variant of the job-shop scheduling problem with hybrid asp. In: Logics in Artificial Intelligence: 17th European Conference, JELIA 2021, Virtual Event, May 17–20, 2021, Proceedings 17. pp. 313–328. Springer (2021)
15. Garey, M.R., Johnson, D.S., Sethi, R.: The complexity of flowshop and jobshop scheduling. Mathematics of operations research **1**(2), 117–129 (1976)

16. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
17. Janhunen, T., Kaminski, R., Ostrowski, M., Schellhorn, S., Wanko, P., Schaub, T.: Clingo goes linear constraints over reals and integers. Theory and Practice of Logic Programming **17**(5-6), 872–888 (2017)
18. Koponen, L., Oikarinen, E., Janhunen, T., Säilä, L.: Optimizing phylogenetic supertrees using answer set programming. Theory and Practice of Logic Programming **15**(4-5), 604–619 (2015)
19. Lenstra, J.K., Kan, A.R., Brucker, P.: Complexity of machine scheduling problems. In: Annals of discrete mathematics, vol. 1, pp. 343–362. Elsevier (1977)
20. Liu, M., Hao, J.H., Wu, C.: A prediction based iterative decomposition algorithm for scheduling large-scale job shops. Mathematical and Computer Modelling **47**(3-4), 411–421 (2008)
21. Ovacik, I.M., Uzsoy, R.: Decomposition methods for complex factory scheduling problems. Springer Science & Business Media (2012)
22. Shi, G., Yang, Z., Xu, Y., Quan, Y.: Solving the integrated process planning and scheduling problem using an enhanced constraint programming-based approach. International Journal of Production Research pp. 1–18 (2021)
23. Singer, M.: Decomposition methods for large job shops. Computers & Operations Research **28**(3), 193–207 (2001)
24. Uzsoy, R., Wang, C.S.: Performance of decomposition procedures for job shop scheduling problems with bottleneck machines. International Journal of Production Research **38**(6), 1271–1286 (2000)
25. Zhai, Y., Liu, C., Chu, W., Guo, R., Liu, C.: A decomposition heuristics based on multi-bottleneck machines for large-scale job shop scheduling problems. Journal of Industrial Engineering and Management (JIEM) **7**(5), 1397–1414 (2014)
26. Zhang, R., Wu, C.: A hybrid approach to large-scale job shop scheduling. Applied intelligence **32**(1), 47–59 (2010)

Listing 6. sub-prog

```
#program subproblem(t).

 seq(Oper1, (Job1, Step2), PT1, t) :-
                  seqL(Oper1, (Job1, Step2), PT1),
                  Oper1 = (Job1, Step1),
                  assignTW(Job1, Step2, t),
                  assignTW(Job1, Step1, T),
          T = #max{TT : assignTW(Job1, Step1, TT), TT <= t}.

{seq((Job1, Step1), (Job2, Step2), PT1, t)} :-
                  sameMach(Job1, Step1, Job2, Step2),
                  pro(Job1, Step1, PT1),
                  assignTW(Job1, Step1, t),
                  assignTW(Job2, Step2, t).

 seq((Job2, Step2), (Job1, Step1), PT2, t) :-
                  sameMach(Job1, Step1, Job2, Step2),
                  pro(Job1, Step1, PT1),
                  pro(Job2, Step2, PT2),
                  assignTW(Job1, Step1, t),
                  assignTW(Job2, Step2, t),
              not seq((Job1, Step1), (Job2, Step2), PT1, t).

 seq((Job1, Step1), (Job2, Step2), PT1, t) :-
                  sameMach(Job1, Step1, Job2, Step2),
                  pro(Job1, Step1, PT1),
                  assignTW(Job1, Step1, T),
                  assignTW(Job2, Step2, t),
              T = #max{TT : assignTW(Job, Step, TT), TT < t}.

 seq((Job2, Step2), (Job1, Step1), PT2, t) :-
                  sameMach(Job1, Step1, Job2, Step2),
                  pro(Job2, Step2, PT2),
                  assignTW(Job1, Step1, t),
                  assignTW(Job2, Step2, T),
              T = #max{TT : assignTW(Job, Step, TT), TT < t}.
```

Listing 7. diff-log

```
&diff{ 0 - (Job, Step) } <= 0 :- operation(Job, Step),
                                 assignTW(Job, Step, t).

&diff{ 0 - (Job, Step) } <= -ST :-
                            startTime((Job, Step), ST, t-1).

&diff{ (Job, Step) - 0 } <=  ST :-
                            startTime((Job, Step), ST, t-1).

&diff{ Oper1 - Oper2 }   <= -PT :- seq(Oper1, Oper2, PT, t).


&diff{ (Job, Step) - bound} <= -PT :- pro(Job, Step, PT),

assignTW(Job, Step, t).

#program opt(b).
#external bound(b).
&diff{ bound  - 0 }     <= b :- bound(b).
```