Solving the Talent Scheduling Problem by Parallel Constraint Programming

Ke Liu, Sven Löffler, Petra Hofstedt

Solving the Talent Scheduling Problem By Parallel Constraint Programming

Ke Liu, Sven Löffler, and Petra Hofstedt

Brandenburg University of Technology Cottbus-Senftenberg, Germany
Department of Mathematics and Computer Science, MINT
Konrad-Wachsmann-Allee 5, 03044 Cottbus
{liuke,sven.loeffler,hofstedt}@b-tu.de

Abstract. The Talent Scheduling problem (TS) is a practical problem entailed by devising a schedule for shooting a film, which is a typical constraint optimization problem. The current modeling approaches are limited and not efficient enough. We present a more concise and efficient modeling approach for the problem. Besides, we exploit TS as a case study to explore how to utilize parallel constraint solving to speedup this constraint optimization problem.

Keywords: Scheduling · Constraint programming · Parallel constraint solving · The Talent scheduling problem

1 Introduction

The talent scheduling problem (TS) is a NP-hard problem originally presented in [1], and it is problem No.039 in CSPLib [2]. The problem can be described as follows: the process of making a film is partitioned into n individual pieces, each of which may require a different subset of the resources such as actors, props and costumes, etc., which can be viewed as a set whose members are m independent resources. Besides, the duration of pieces varies according to the requirement of the film shoot; the cost of different resources is paid at different rates. For a given piece, the cost incurred by one resource is equal to the product of the duration of the piece and the cost of the resource. A feasible solution of a TS problem can be represented as a table (cf. Table 1), in which each column stands for a fixed set of resources required by a piece, while each row represents the demand for the resource for all the pieces of the film. One feasible solution of the TS problem differs from another only because of their different permutations of pieces (columns). By contrast, the order of resources (rows) can always be fixed. A cell of a feasible solution is assigned to one if the resource is required by the piece, otherwise zero. For example, in Table 1, piece 4 requires resource 1 and does not require resource 2; hence, the corresponding cells are 1 and 0 respectively.

The cost of a resource only depends on the interval between the first piece in which it is involved and the last piece in which it is involved, which implies that the idle times of the resource in the interval also need to be paid. In the

Piece	4	1	11	10	13	12	3	2	6	8	7	9	5	20	15	14	17	18	16	19	Cost/100
resource 1	1	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	10
resource 2	0	1	0	1	1	0	1	1	0	1	1	0	0	1	1	1	1	0	0	0	4
resource 3	0	0	0	0	1	0	1	1	0	1	1	0	1	0	1	1	0	0	0	0	5
resource 4	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
resource 5	0	0	0	0	0	1	0	1	0	1	1	0	0	1	0	1	0	1	0	1	5
resource 6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	40
resource 7	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	4
resource 8	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	20
Duration	1	2	1	2	2	1	1	1	3	1	1	1	1	1	2	1	1	2	1	1	

Table 1. A feasible solution for a given TS in [3]. The Cost and Duration in the last column and the last row stands for the cost per time unit and the duration of the pieces, respectively. The overall cost of this solution is 14,600.

presented paper, if a resource lies idle in such an interval, we call it *idle resource*. For instance, resource 1 required from piece 4 to piece 8 lies idle for pieces 13 and 12 in the feasible solution shown in Table 1. In this case, resource 1 still needs to be paid due to pieces 13 and 12. Thus, the additional expense incurred by resource 1 is calculated by $10 * 2 + 10 * 1$, where 10 is the cost of resource 1, and 2 and 1 are the durations of pieces 13 and 12 respectively. The objective of the TS problem is to find a feasible solution that has the lowest cost incurred by all the idle resources of the feasible solution. Given the above, the TS problem is a typical constraint optimization problem (CSOP).

Consider the making process of a film composed of n pieces and m kinds of resources, d_j ($j \in \{1..n\}$) denotes the duration for piece j, while c_i ($i \in \{1..m\}$) denotes the costs of resource i. Besides, we define a function $\tau(i, j)$ as follows:

$$\tau(i, j) = 1 - T(i, j) \tag{1}$$

where $T(i, j)$ denotes the value in row i and column j of the given feasible solution, and the domain of j is over the interval between the first occurrence of one (f_i) and the last occurrence of one (l_i) in row i. Therefore, the total cost function for all the idle resources in a feasible solution t is:

$$cost(t) = \sum_{i=1}^{m} \sum_{j=f_i}^{l_i} \tau(i, j) \cdot d_j \cdot c_i \tag{2}$$

where t determines f_i, l_i, and $\tau(i, j)$. Therefore, the objective of the TS problem can be stated as finding a feasible solution which has the minimum value of the *cost* function, which is given by:

$$\underset{t \in T}{Minimize}\ cost(t) \tag{3}$$

where T is the total solution space of the given TS problem.

The remaining part of the paper is structured as follows: In Section 2, we introduce the basic notions used in this paper. Then, in Section 3, we gradually

describe the modeling approach for the problem in detail. Next, in Section 4, we present an approach to solve the problem in parallel, experimental results are given in Section 5. Finally, we conclude in Section 6.

2 Preliminaries

Constraint programming (CP) is one of the most powerful techniques to tackle combinatorial problems, generally NP-complete or NP-hard. It employs constraint propagation interleaved with backtrack search. The problem to be solved is expressed through a formal model by using constraints from a rich set of modeling primitives. A constraint network $\mathcal{R}$ or constraint satisfaction problem (CSP) is a triple $\langle X, D, C\rangle$, which consists of:

- a finite set of variables $X = \{x_1, \ldots, x_k\}$, where k is the number of variables in $\mathcal{R}$,
- a set of respective finite domains $D = \{D(x_1), \ldots, D(x_k)\}$, where $D(x_i)$ is the domain of the variable x_i, and
- a set of constraints $C = \{c_1, ..., c_t\}$, where a constraint c_j is a relation defined on the domains of a subset S_j of X, $c_j \subseteq \prod_{x \in S_j} D(X)$.

A Constraint Satisfaction Optimization Problem $\langle X, D, C, f\rangle$ (CSOP) is defined as a CSP with an optimization function f that *maps each solution to a numerical value* [4,5]. Generally, a solution t of a CSOP is a solution of the corresponding CSP, where $f(t)$ is maximal or minimal.

3 A CSP Model

We are now going to introduce our model in detail. Any feasible solution for the TS problem is a permutation of $\{1, 2, ...n\}$, where n is the number of pieces involved in film shooting. Hence, the problem can be viewed as to assign n values to n slots. We define the decision variables as $X = \{x_1, x_2, ..., x_n\}$, each of which has domain $\{1, 2, ...n\}$, where $x_i = j$ if slot i is assigned to piece j in the sequence. Therefore, the basic constraint of the model can be described as:

$$\forall i, j,\ x_i \neq x_j \tag{4}$$

where $1 \leq i < j \leq n$. Constraint 4 can be realized by the **allDifferent** constraint [6], which is implemented in almost all constraint solvers.

Though a TS problem can simply be solved by the **allDifferent** constraint[1], the search space would be immense even for an small number of pieces in practice, and consequently the problem cannot be solved in a reasonable time frame. For example, the problem with 20 pieces, shown in Table 1, has 20! permutations on the sequence {1..20}, which leads to the fact that to iterate over all the

[1] The names of the constraints used in this paper are consistent with the names of constraints used in the Choco Solver [7].

possible permutations is impossible in a reasonable execution time. Therefore, the subsequent constraints we are going to present are used to reduce the search space.

Both a feasible solution and the solution in its reverse order have the same cost function value because the expenses incurred by the idle resources in both solutions are the same. Thus, it is unnecessary to re-explore such search regions for the resolution process. The static symmetry breaking constraint in our model is relatively simple and can be stated as:

$$x_1 < x_n \tag{5}$$

Obviously, the **arithm** constraint can be used to express Constraint 5. By imposing Constraint 5, the overall search space is halved.

We can also take advantage of the intrinsic characteristics of the data to shrink the search space further. Specifically, if there is a fixed pattern which an optimal solution must contain, a set of *optimality constraints* can restrict the search space to the solutions containing such fixed pattern. An optimal solution remains unchanged if we exchange two pieces which require the same set of resources. However, as we can see from Table 1, there are some pairs of pieces in which two pieces request almost the same set of resources except for one difference. For example, piece 1 and piece 3 require almost the same resources apart from resource 3. As a result, only resource 3 has an effect on the positional relation among pieces 1, 3 and other pieces in an optimal solution since piece 1 would be treated as the same as piece 3 if they require the same resources. Moreover, piece 3 must be closer to the pieces (e.g., piece 13) requiring resource 3 in an optimal solution, compared to piece 1 because this arrangement of pieces is bound to incur a lower overall cost.

Based on this observation, we can first find all pairs of pieces requiring the same resources but one, and then, find a benchmark resource containing that difference resource, which can be stated as:

$$\mid idx_i - idx_{bm} \mid \quad < \quad \mid idx_j - idx_{bm} \mid \tag{6}$$

where idx_i, idx_j, and idx_{bm} are set to the index of the value (piece) i, j, and bm. the set of all idx_i, idx_j, and idx_{bm} is:

$$\{(i,j,bm) \mid i \neq j \neq bm, |R_i \cup R_j| - |R_i \cap R_j| = 1, (R_i \cup R_j) \setminus (R_i \cap R_j) \in R_{bm}\} \tag{7}$$

Herein R_i, R_j, and R_{bm} represent the set of resources required by pieces i, j, and bm (e.g., $R_1 = \{1,2\}, R_3 = \{1,2,3\}, R_{13} = \{2,3,4\}$). We denote idx_i and idx_j as the indices of piece i and piece j, where resources required by them only have one difference, given by $|R_i \cup R_j| - |R_i \cap R_j| = 1$. The index of bm (idx_{bm}) is the benchmark for the indices i and j, in which piece bm entails the different resource between resources required by piece i and piece j, which can be expressed as $(R_i \cup R_j) \setminus (R_i \cap R_j) \in R_{bm}$. In other words, the criterion for a benchmark is to select the piece requiring the only different resource between resources required by piece i and piece j.

Take the example mentioned above, piece 13 is the benchmark of pieces 3 and 1, and piece 3 must be closer to the benchmark than piece 1 is. We have the following constraint: $\mid idx_3 - idx_{13} \mid < \mid idx_1 - idx_{13} \mid$. Please note that piece 13 is not the only choice of the benchmark. All the pieces requiring resource 3 (e.g., pieces 2, 8 etc.) can be the benchmark between piece 3 and piece 1.

Constraint 6 can be realized by the **inverseChanneling**, the **distance** constraint, and the **arithm** constraint. For a specific example of Constraint 6, we first use the **inverseChanneling** constraint to record the indices of the decision variables X by introducing new auxiliary variables IDX, given by:

$$X[i] = j \Leftrightarrow IDX[j] = i \tag{8}$$

where $i, j \in \{1..n\}$. Having auxiliary variables IDX, we can control the distance between given values in a output of a sequence easily. For a given value i, j, and bm, Constraint 6 can be converted to:

$$distance \mid IDX[i] - IDX[bm] \mid < distance \mid IDX[j] - IDX[bm] \mid \tag{9}$$

If one implements this model in the Choco solver, two extra auxiliary variables must be introduced in order to store the *resulting variables* of Constraints 10 and 11 for the **distance** constraints. Therefore, both sides of the less than sign ($<$) of the Constaint 9 can be replaced by:

$$distance \mid IDX[i] - IDX[bm] \mid = aux_1 \tag{10}$$

$$distance \mid IDX[j] - IDX[bm] \mid = aux_2 \tag{11}$$

Then, the **arithm** constraint can be used to restrict the relation between the two auxiliary variables, given by:

$$arithm(aux_1 \ < \ aux_2) \tag{12}$$

Apparently, one instance of Constraint 6 might reduce more than half of the entire search space because the solutions satisfying $\mid idx_i - idx_{bm} \mid \geq \mid idx_j - idx_{bm} \mid$ are ruled out. In the concrete implementation steps , we first find all the pairs of pieces that only have one different resource, then impose the instances of Constraint 6 for these pairs on the model.

Local search (LS), an incomplete search method for finding an optimal solution, is often the method of choice to solve CSOPs. Several ways to combine CP and local search have been proposed in the literature [8,9]. One way to utilize LS for CP is to freeze a fragment of the variables specified with fixed values and to solve the subproblem defined by the uninstantiated variables. Hence, we should carefully decide which variables should be frozen. For the data shown in Table 1, resources 6 and 8 cost much more than other resources and there is no intersection between pieces requiring resources 6 and the pieces requiring resources 8; therefore, we assume that pieces involving with resource 6 or resource 8 are more likely to arrange together within an optimal solution. Please note that this method does not guarantee that no optimal solution will be ruled

out, but the experimental result shows that the best solution we obtained is the same as that one provided in [3]. In our implementation, we treat piece 6, 7, 8, and 9 as an *entirety*, say piece 21; and also treat piece 14, 15, 16, 17, and 18 as another entirety, say piece 22. Consequently, the entire search space is reduced from 20! to $13! \cdot 5! \cdot 4!$ by this means. Though, the more freezing variables are, the more search space reduction is, there is a trade-off between search space reduction and the optimal solutions possibly contained in the reduced search space. In general, to apply LS to solve TS problem, we should consider to freeze the pieces entailing the highest cost of the resource, the second highest cost of the resources, etc. in turn, and then treat these pieces as entireties. Meanwhile, we also take into account the trade-off we mentioned above. Based on the above analysis, selecting variables for LS are automated:

1. Sort resource based on their cost in descending order.
2. Freeze pieces requiring the current highest cost resource.
3. If the number of frozen pieces can lead the computation ends in predefined execution time (e.g., 20 mins), then the selection procedure stops. Otherwise, we set the next resource in the sorted resources as the current resource and then repeat step 2.

In summary, Constraints 4 and 5, together with constraints entailed by Constraint 6 with LS form the model used to tackle the TS problem.

4 Solving the TS in Parallel

We would like to briefly introduce *Embarrassingly Parallel Search* (EPS) [10] in constraint programming. EPS indicates no communication requirement during the solving process. An embarrassingly parallel workload distribution also implies that the independent worker works on distinct data. EPS is well suited for solving the TS problem in parallel since disjoint partial solutions can be easily obtained before the solving process and then mapped to workers. Hence, we use EPS with static decomposition to accelerate the solving process of TS problem.

First, a subset of the decision variables of the model is selected. One viable way is to select the pieces that do not belong to any entirety. In other words, pieces frozen by LS are excluded. For the data in Table 1, we chose a set of pieces $PS = \{1, 2, 3, 4, 5, 10, 11, 12, 13, 19, 20\}$ to generate partial solutions.

Then, the model presented in Section 3 is applied to the decision variables whose domain is PS and the number of the decision variables is equivalent to the cardinality of the set PS. The model used to generate the partial solutions includes Constraints 4, 5, and constraints entailed by Constraint 6. Please recall that all the pairs of pieces that only have one different resource are found, and then we impose the instances of Constraint 6 for these pairs on the model in sequential version. In the parallel version, the instances of Constraint 6 would not be entailed by the pieces not contained in PS when generating the partial solutions. In doing so, there are two advantages: First, the partial solutions can

be generated in a reasonable time for the next step; second, solutions that cannot be extended to an optimal solution wouldn't be generated.

After generating the partial solutions, each worker receives the same number of partial solutions and works on its own independent partial solutions in parallel. Embarrassingly parallel execution works as follows on each worker:

1. The entireties are inserted into all possible positions of a partial solution so that all possible permutations with entireties for all pieces are generated.
2. The cost function (Equation 2) is evaluated for each permutation. The branch-and-bound algorithm is used here to avoid visiting the nodes that are impossible to be extended as an optimal solution.

It is worth noting that PS doesn't include all the pairs of pieces that only have one different resource. Thus, in order to shrink the search space for each worker, the instances of Constraints 6 are entailed by the remaining part of such pairs of pieces. Also note that there is no communication between workers, which means no communication cost; on the other hand, a new lower bound discovered by a worker cannot be used by other workers for improving their current resolution. Obviously, there is a trade-off between sharing lower bounds and communication. We decided not to use communication, because, for the sake of scalability of the parallel algorithm, more workers and smaller sub-problems might lead to more expensive communication cost and the uselessness of sharing lower bound since the resolution time of the sub-problems may be short.

The final step is to obtain the permutation with the smallest value of the cost function.

5 Numerical Results

In order to confirm our theoretical discussion, we implemented the model as described in Section 3 and Section 4 and its parallel version in the Choco Solver 4.0.6 [7] with JDK version 9.0.1. All experiments were performed on a computer with an Intel i7-3720QM CPU, 2.60GHz with 4 physical and 8 logical cores, and 8 GB DDR3 memory running Ubuntu 17.10.

All tricks of solving TS problem in a reasonable execution time is essentially to reduce search space. The constraints regarding search space reduction allowed us only to evaluate 1,027,403,520 out of 20! feasible solutions, reducing approximately 99.99% of the total search space. The following experiment was carried out to test the effectiveness of the search space reduction sequentially and its parallel version.

As can be seen from Table 2, the efficiency dropped rapidly as the number of workers increased from 4 to 8. Theoretically, we would not have expected this result since no communication is required. Thus, in order to eliminate the factors such as the limit number of physical cores and the limited size of the cache, we halved the partial solutions, and executed the first part of the partial solutions and the second part of partial solution by using 4 workers in turn.

Number of Workers	1	2	4	8
Execution time (s)	451.68	250.56	137.64	119.82
Speedup	1	1.8	3.28	3.77
Efficiency	1	0.9	0.82	0.47125

Table 2. Solving the TS on a multi-core computer

Four Workers	First Part	Second Part
Execution time (s)	83.34	81.42

Table 3. Use 4 workers to calculate the first part and second part in turn.

As a result, the speedup of 8 workers can be 5.42 ($\frac{7.528}{1.389}$), and its efficiency is 0.68.

The execution time of the sequential part for generating the partial solutions was 25 seconds in the parallel version; therefore, by using Amdahl's law [11] we can calculate that the theoretical speedup of our approach is around 18.1 for the given data. Additionally, one benefit of parallel solving for the TS is that we may obtain different optimal solutions in a shorter time. For the problem shown in Table 1, we obtained the following optimal solutions:

4	1	11	10	13	3	12	2	6	8	7	9	5	20	15	14	17	18	16	19
4	1	11	10	13	12	3	2	6	8	7	9	20	5	15	14	17	18	16	19
4	1	11	10	3	13	12	2	6	8	7	9	20	5	15	14	17	18	16	19
4	1	11	10	13	3	12	2	6	8	7	9	20	5	15	14	17	18	16	19

Table 4. Optimal solutions in order of pieces with cost 14,600

6 Conclusion

We have presented the model for TS problem, as well as utilizing data-level parallelism to speedup the execution. Besides, our approach also employs the local search to reduce the search space. The experimental results indicate that embarrassingly parallel search can be an appropriate choice to solve such constraint optimization problems.

Nevertheless, we believe there is still a potential to improve the performance of our approach. Although it has performed well, Constraint 6 and the dual variables could be replaced by a tailored constraint that can realize the functional requirements of Constraint 6 and use only one constraint instead of the **distance**, the **arithm** constraint, and the **ifThen** constraint for the dual variables. Besides, the theoretical speedup can still be improved for larger instances though

it is limited for the given data. Finally, we would like to explore the use of our parallel approach to accelerate the resolution of other CSPs and CSOPs in the future.

References

1. TCE Cheng, JE Diamond, and BMT Lin. Optimal scheduling in film production to minimize talent hold cost. *Journal of Optimization Theory and Applications*, 79(3):479–492, 1993.
2. Barbara Smith. CSPLib problem 039: The rehearsal problem. Accessed: 2018-03-08.
3. Barbara Smith. Constraint programming in practice: Scheduling a rehearsal. *Research Report APES-67-2003, APES group*, 2003.
4. Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1995.
5. Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.
6. Jena-Lonis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial intelligence*, 10(1):29–127, 1978.
7. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.
8. Filippo Focacci, François Laburthe, and Andrea Lodi. Local search and constraint programming. In *Handbook of metaheuristics*, pages 369–403. Springer, 2003.
9. Gilles Pesant. A constraint programming approach to the traveling tournament problem with predefined venues. *Practice and Theory of Automated Timetabling*, pages 303–316, 2012.
10. Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science*, volume 8124, pages 596–610. Springer, Berlin, Heidelberg, 2013.
11. Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.