European Journal of Operational Research 250 (2016) 412–426

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete Optimization

An enhanced branch-and-bound algorithm for the talent scheduling problem

Hu Qin[a], Zizhen Zhang[b,*], Andrew Lim[c], Xiaocong Liang[d]

[a] School of Management, Huazhong University of Science and Technology, No. 1037, Luoyu Road, Wuhan, China
[b] School of Mobile Information Engineering, Sun Yat-sen University, Guangzhou, China
[c] Department of Industrial & Systems Engineering National University of Singapore 1, Engineering Drive 2, Singapore 117576
[d] Department of Computer Science, Sun Yat-Sen University, Guangzhou, China

ARTICLE INFO

Article history:
Received 8 November 2014
Accepted 1 October 2015
Available online 22 October 2015

Keywords:
scheduling
Talent scheduling
Branch-and-bound
Dynamic programming
Dominance rules

ABSTRACT

The talent scheduling problem is a simplified version of the real-world film shooting problem, which aims to determine a shooting sequence so as to minimize the total cost of the actors involved. In this article, we first formulate the problem as an integer linear programming model. Next, we devise a branch-and-bound algorithm to solve the problem. The branch-and-bound algorithm is enhanced by several accelerating techniques, including preprocessing, dominance rules and caching search states. Extensive experiments over two sets of benchmark instances suggest that our algorithm is superior to the current best exact algorithm. Finally, the impacts of different parameter settings, algorithm components and instance generation distributions are disclosed by some additional experiments.

1. Introduction

The scenes of a film are not generally shot in the same sequence as they appear in the final version. Finding an optimal sequence in which the scenes are shot motivates the investigation of the talent scheduling problem, which is formally described as follows. Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of n scenes and $A = \{a_1, a_2, \ldots, a_m\}$ be a set of m actors. All scenes are assumed to be shot on a given location. Each scene $s_j \in S$ requires a subset $a(s_j) \subseteq A$ of actors and has a duration $d(s_j)$ that commonly consists of one or several days. Each actor a_i is required by a subset $s(a_i) \subseteq S$ of scenes. We denote by Π the permutation set of the n scenes and define $e_i(\pi)$ (respectively, $l_i(\pi)$) as the earliest day (respectively, the latest day) in which actor i is required to be present on location in the permutation $\pi \in \Pi$. Each actor $a_i \in A$ has a daily wage $c(a_i)$ and is paid for each day from $e_i(\pi)$ to $l_i(\pi)$ regardless of whether he (or she) is required in the scenes. The objective of the talent scheduling problem is to find a shooting sequence (i.e., a permutation $\pi \in \Pi$) of all scenes that minimizes the total paid wages.

Table 1 presents an example of the talent scheduling problem, which is reproduced from de la Banda, Stuckey, and Chu (2011). The information of $a(s_j)$ and $s(a_i)$ is determined by the $m \times n$ matrix M shown in Table 1(a), where cell $M_{i,j}$ is filled with an "X" if actor a_i participates in scene s_j and with a " · " otherwise. Obviously, we can obtain $a(s_j)$ and $s(a_i)$ by $a(s_j) = \{a_i | M_{i,j} = X\}$ and $s(a_i) = \{s_j | M_{i,j} = X\}$, respectively. The last row gives the duration of each scene and the rightmost column gives the daily cost of each actor. If the shooting sequence is $\pi = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}\}$, we can get a matrix $M(\pi)$ shown in Table 1(b), where in cell $M_{i,j}(\pi)$ a sign "X" indicates that actor a_i participates in scene s_j and a sign "–" indicates that actor a_i is waiting at the filming location. The cost of each scene is presented in the second-to-last row and the total cost is 604. The cost incurred by the waiting status of the actors is called *holding cost*, which is shown in the last row of Table 1(b). The optimal solution of this instance is $\pi^* = \{s_5, s_2, s_7, s_1, s_6, s_8, s_4, s_9, s_3, s_{11}, s_{10}, s_{12}\}$ whose total cost and holding cost are 434 and 53, respectively.

The talent scheduling problem was originated from Adelson, Norman, and Laporte (1976) and Cheng, Diamond, and Lin (1993). Adelson et al. (1976) introduced an orchestra rehearsal scheduling problem, which can be viewed as a restricted version of the talent scheduling problem with all actors having the same daily wage. They proposed a simple dynamic programming algorithm to solve their problem. Cheng et al. (1993) studied a film scheduling problem in which all scenes have identical duration. They first showed that the problem is NP-hard even if each actor is required by two scenes and the daily wage of each actor is one. Next, they devised a branch-and-bound algorithm and a simple greedy hill climbing heuristic to solve

* Corresponding author. Tel.: +86 13826411848; fax: +852 34420189.
E-mail addresses: tigerqin1980@gmail.com, tigerqin@hotmail.com (H. Qin), zhangzizhen@gmail.com (Z. Zhang), lim.andrew@cityu.edu.hk (A. Lim), yurilxc@gmail.com (X. Liang).

http://dx.doi.org/10.1016/j.ejor.2015.10.002
0377-2217/© 2015 Elsevier B.V. and Association of European Operational Research Societies (EURO) within the International Federation of Operational Research Societies (IFORS). All rights reserved.

Table 1
An example of the talent scheduling problem reproduced from de la Banda et al. (2011).

(a) The matrix M for an instance of the talent scheduling problem.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	$c(a_i)$
a_1	X	·	X	·	·	X	·	X	X	X	X	X	20
a_2	X	X	X	X	X	·	X	·	X	·	X	·	5
a_3	·	X	·	·	·	·	X	X	·	·	·	·	4
a_4	X	X	·	·	X	X	·	·	·	·	·	·	10
a_5	·	·	·	X	·	·	·	X	X	·	·	·	4
a_6	·	·	·	·	·	·	·	·	·	X	·	·	7
$d(s_j)$	1	1	2	1	3	1	1	2	1	2	1	1	

(b) The matrix $M(\pi)$ corresponding to a solution π of the instance.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	$c(a_i)$
a_1	X	–	X	–	–	X	–	X	X	X	X	X	20
a_2	X	X	X	X	X	–	X	–	X	–	X	·	5
a_3	·	X	–	–	–	–	X	X	·	·	·	·	4
a_4	X	X	–	–	X	X	·	·	·	·	·	·	10
a_5	·	·	·	X	–	–	–	X	X	·	·	·	4
a_6	·	·	·	·	·	·	·	·	·	X	·	·	7
Cost	35	39	78	43	129	43	33	66	29	64	25	20	604
Holding cost	0	20	28	34	84	13	24	10	0	10	0	0	223

their problem. Later, Smith (2003) applied constraint programming to solve both the problems introduced by Adelson et al. (1976) and Cheng et al. (1993). In her subsequent work, namely Smith (2005), she accelerated her constraint programming approach by caching search states.

The talent scheduling problem we study in this article was first formally described by de la Banda et al. (2011). This problem is a generalization of the problems introduced by Adelson et al. (1976) and Cheng et al. (1993), where scenes may have different durations and actors may have different wages. However, it is a simplified version of the movie shoot scheduling problem (MSSP) introduced by Bomsdorf and Derigs (2008). In the MSSP, we need to deal with a couple of practical constraints, such as the precedence relations among scenes, the time windows of each scene, the resource availability, and the working time windows of actors and other film crew members. Recently, Liang, Zhang, Qin, Guo, and Lim (2014) proposed a branch-and-bound algorithm to solve the talent scheduling problem and achieved better results than de la Banda et al. (2011).

In literature, there exist several meta-heuristics developed for the problem introduced by Cheng et al. (1993). Nordström and Tufekci (1994) provided several hybrid genetic algorithms for this problem and showed that their algorithms outperform the heuristic approach in Cheng et al. (1993) in terms of both solution quality and computation speed. Fink and Voß (1999) treated this problem as a special application of the general pattern sequencing problem, and implemented a simulated annealing algorithm and several tabu search heuristics to solve it.

The talent scheduling problem is a very challenging combinatorial optimization problem. The current best exact approach by de la Banda et al. (2011) can only optimally solve small- and medium-size instances. In this paper, we propose an enhanced branch-and-bound algorithm for the talent scheduling problem, which uses the following two main techniques:

- *Dominance rules.* When a partial solution represented by a node in the search tree can be dominated by another partial solution, this node need not be further explored and can be safely discarded.
- *Caching search states.* The talent scheduling problem can be solved by dynamic programming algorithm (see de la Banda et al. (2011)). It is beneficial to incorporate the dynamic programming states into the branch-and-bound framework by a memoization technique. In the branch-and-bound tree, each node is related to a dynamic programming state. If the search process explores a certain node whose already confirmed cost is not smaller than the value of its corresponding cached state, this node can be pruned.

There are three main contributions in this paper. Firstly, we formulate the talent scheduling problem as a mixed integer linear programming model so that commercial mathematical programming solvers can be applied to the problem. Secondly, we propose an enhanced branch-and-bound algorithm whose novelties include a new lower bound, caching search states and two problem-specific dominance rules. Thirdly, we achieved the optimal solutions for more benchmark instances by our algorithm. The experimental results show that our branch-and-bound algorithm is superior to the current best exact approach by de la Banda et al. (2011).

The remainder of this paper is organized as follows. In Section 2, we present the mixed integer linear programming model for the talent scheduling problem. Next, we describe our branch-and-bound algorithm in Section 3, including the details on a double-ended search strategy, the computation of the lower bound, a preprocessing step, the state caching process and the dominance rules. The computational results are reported in Section 4, where we used our algorithm to solve over 200,000 benchmark instances. Finally, we conclude our study in Section 5 with some closing remarks.

2. Mathematical formulation

The talent scheduling problem is essentially a permutation problem. It tries to find a permutation (i.e., a schedule) $\pi = (\pi(1), \ldots, \pi(n)) \in \Pi$, where $\pi(k)$ is the kth scene in permutation π, such that the total cost $C(\pi)$ is minimized. The value of $C(\pi)$ is computed as:

$$C(\pi) = \sum_{i=1}^{m} c(a_i) \times \big(l_i(\pi) - e_i(\pi) + 1\big)$$

We set the parameter $m_{i,j} = 1$ if $M_{i,j} = X$ and $m_{i,j} = 0$ otherwise. The total holding cost can be easily derived as:

$$H(\pi) = \sum_{i=1}^{m} c(a_i) \times \left(l_i(\pi) - e_i(\pi) + 1 - \sum_{j=1}^{n} m_{i,j} d(s_j)\right)$$

Apparently, for this problem minimizing the total cost is equivalent to minimizing the total holding cost (de la Banda et al., 2011).

We create two dummy scenes s_0 and s_{n+1} to represent the first and the last scenes to be shot, namely, $\pi(0) = s_0$ and $\pi(n+1) = s_{n+1}$. The starting days for shooting s_0 and s_{n+1} are equal to zero and $\sum_{j=1}^{n} d(s_j) + 1$, respectively. The durations of s_0 and s_{n+1} are both equal to zero. The talent scheduling problem can be formulated into an integer linear programming model using the following decision variables:

$x_{k,j}$: a binary variable that equals 1 if scene s_j is scheduled immediately after scene s_k, and 0 otherwise.
t_j: the starting day for shooting scene s_j.
e_i: the earliest shooting day that requires actor a_i.
l_i: the latest shooting day that requires actor a_i.

The integer programming model is given by:

$$\text{(IP)} \quad \min \sum_{i=1}^{m} c(a_i)(l_i - e_i + 1) \tag{1}$$

$$\text{s.t.} \sum_{j=1}^{n} x_{0,j} = 1 \tag{2}$$

$$\sum_{k=1}^{n} x_{k,n+1} = 1 \tag{3}$$

$$\sum_{j=1,k\neq j}^{n+1} x_{k,j} = 1, \ \forall\, 1 \le k \le n \tag{4}$$

$$\sum_{k=0,k\neq j}^{n} x_{k,j} = 1, \ \forall\, 1 \le j \le n \tag{5}$$

$$\sum_{j=1,k\neq j}^{n} t_j x_{k,j} = t_k + d(s_k), \ \forall\, 0 \le k \le n \tag{6}$$

$$t_0 = 0, t_{n+1} = \sum_{j=1}^{n} d(s_j) + 1 \tag{7}$$

$$e_i \le t_j, \ \forall\, 1 \le i \le m, \ 1 \le j \le n, \ m_{i,j} = 1 \tag{8}$$

$$t_j + d(s_j) - 1 \le l_i, \ \forall\, 1 \le i \le m, \ 1 \le j \le n, \ m_{i,j} = 1 \tag{9}$$

$$x_{k,j} \in \{0, 1\}, \ \forall\, 0 \le k \le n, \ 1 \le j \le n+1 \tag{10}$$

$$e_i, l_i, t_j \ge 0 \text{ and integer}, \ \forall\, 1 \le i \le m, \ 0 \le j \le n+1 \tag{11}$$

The objective (1) is to minimize the total cost, where $l_i - e_i + 1$ is the number of days in which actor a_i is present on location. Constraints (2) and (3) ensure that the first and the last scenes are s_0 and s_{n+1}, respectively. Constraints (4) and (5) guarantee that every scene has exactly one immediate successor and one immediate predecessor, respectively. Constraints (6) state that the starting day of scene s_j is determined by the starting day of its predecessor scene s_k. Moreover, these constraints prevent sub-tours from occurring. Constraints (8) and (9) ensure that the earliest and the latest shooting days that require actor a_i are determined by the starting days of scenes in which he (or she) is involved.

Observe that Constraints (6) are nonlinear. To linearize them, we introduce a set of additional variables $z_{k,j}(0 \le k \le n, 1 \le j \le n, k \neq j)$, and set $z_{k,j} = t_j x_{k,j}$. We know that $z_{k,j} = t_j$ if $x_{k,j} = 1$ and $z_{k,j} = 0$ otherwise. Thus, $z_{k,j}$ can be restricted by the following four linear constraints:

$$z_{k,j} \ge 0 \tag{12}$$

$$z_{k,j} \le t_j \tag{13}$$

$$z_{k,j} \ge t_j + L(x_{k,j} - 1) \tag{14}$$

$$z_{k,j} \le L x_{k,j} \tag{15}$$

where L is a sufficiently large positive number. Accordingly, Constraints (6) can be rewritten as:

$$\sum_{j=1,k\neq j}^{n} z_{k,j} = t_k + d(s_k), \ \forall\, 0 \le k \le n \tag{16}$$

The objective (1) and Constraints (2)–(5), (7)–(16) constitute an integer linear programming model (ILP) for the talent scheduling problem. This ILP is quite difficult to be optimally solved by commercial integer programming solvers, e.g., ILOG CPLEX. Preliminary experiments revealed that only very small-scale instances, e.g., $n = 10$ and $m = 5$, can be optimally solved by CPLEX 12.1. This is mainly because the linear relaxation of the ILP model cannot provide a high-quality lower bound for the problem.

3. An enhanced branch-and-bound approach

Branch-and-bound is a general technique for optimally solving various combinatorial optimization problems. The basic idea of the branch-and-bound algorithm is to systematically and implicitly enumerate all candidate solutions, where large subsets of fruitless candidates are discarded by using upper and lower bounds, and dominance rules. In this section, we describe the main components of our proposed branch-and-bound algorithm, including a double-ended search strategy, a novel lower bound, the preprocessing stage, the state caching strategy and two dominance rules. For the rest of this discussion, we choose minimizing the total holding cost as the objective of the talent scheduling problem.

3.1. Double-ended search

The solutions of the talent scheduling problem can be easily presented in a branch-and-bound search tree. Suppose we aim to find an optimal permutation $\pi^* = (\pi^*(1), \pi^*(2), \ldots, \pi^*(n))$. A typical branch-and-bound process first determines the first k scenes to be shot, denoted by a partial permutation $(\hat{\pi}(1), \ldots, \hat{\pi}(k))$, at level k of the search tree. Then, it generates $n - k$ branches, each trying to explore a node by assigning a scene to $\pi(k+1)$. At some tree node at level $k+1$, there is a known partial permutation $(\hat{\pi}(1), \hat{\pi}(2), \ldots, \hat{\pi}(k+1))$ and a set of $n - k - 1$ unscheduled scenes. If the lower bound LB to the value of the solutions that contain the partial permutation $(\hat{\pi}(1), \hat{\pi}(2), \ldots, \hat{\pi}(k+1))$ is not less than the current best solution value (i.e., an upper bound UB), then the branch to the node associated with $\hat{\pi}(k+1)$ can be safely discarded. Once the search process reaches a node at level n of the tree, a feasible solution is obtained and the current best solution may be updated accordingly.

The above search methodology can be called the *single-ended search strategy*. As did by Cheng et al. (1993) and de la Banda et al. (2011), we can employ a *double-ended search strategy* that alternatively fixes the first and the last undetermined positions in the permutation. That is to say, the double-ended search determines a scene permutation following the order $\pi(1), \pi(n), \pi(2), \pi(n-1)$ and so on. When using the double-ended search strategy, a node in some level of the search tree corresponds to a partially determined permutation with the form $(\hat{\pi}(1), \ldots, \hat{\pi}(k-1), \pi(k), \ldots, \pi(l), \hat{\pi}(l+1), \ldots, \hat{\pi}(n))$, where $1 \le k \le l \le n$ and the value of $\pi(h)$ $(k \le h \le l)$ is undetermined. We denote by B the set of scenes scheduled at the beginning of the permutation, namely $B = \{\hat{\pi}(1), \hat{\pi}(2), \ldots, \hat{\pi}(k-1)\}$, and by E the set of scenes scheduled at the end, namely $E = \{\hat{\pi}(l+1), \hat{\pi}(l+2), \ldots, \hat{\pi}(n)\}$. The remaining scenes are put in a set Q, namely $Q = S - B - E$. Moreover, for convenience, we denote by $\vec{B}$ and $\vec{E}$ the partially determined scene sequences at the beginning and at the end of a permutation, i.e., $\vec{B} = (\hat{\pi}(1), \ldots, \hat{\pi}(k-1))$ and $\vec{E} = (\hat{\pi}(l+1), \ldots, \hat{\pi}(n))$.

The double-ended search strategy is beneficial to solving the talent scheduling problem. As pointed out by de la Banda et al. (2011), more accurate lower bounds can be obtained by increasing the number of fixed actors. The actor required by the scenes in both B and E is labeled *fixed* since the total number of his/her on-location days is fixed and his/her cost in the final schedule already becomes known (Cheng et al., 1993). We do not need to consider any fixed actor in the later stages of the search process, which certainly reduces the size of the problem. Let $a(Q) = \cup_{s \in Q} a(s)$ be the set of actors required by at least one scene in $Q \subseteq S$. The set of all fixed actors can be defined by $F = a(B) \cap a(E)$.

A generic double-ended branch-and-bound framework is given in Algorithm 1. The operator "$\circ$" in lines 3, 11 and 15 indicates

Algorithm 1 A generic double-ended branch-and-bound search framework.

```
1:  Function: search($\vec{B}, Q, \vec{E}$)
2:  if $Q = \emptyset$ then
3:    $current\_solution = \vec{B} \circ \vec{E}$;
4:    $z =$ evaluate($current\_solution$);
5:    if $z < UB$ then
6:      $UB = z$;
7:      $best\_solution = current\_solution$;
8:    end if
9:  else
10:   for all $s \in Q$ do
11:     $LB =$ lower_bound($\vec{B} \circ s, Q - \{s\}, \vec{E}$);
12:     if $LB \geq UB$ then
13:       continue;
14:     end if
15:     Invoke search($R(\vec{E}), Q - \{s\}, s \circ R(\vec{B})$).
16:   end for
17: end if
```

concatenating two partially determined scene sequences. The function **search**$(\vec{B}, Q, \vec{E})$ returns the optimal solution to the talent scheduling problem with known $\vec{B}$ and $\vec{E}$; we denote this problem by $P(\vec{B}, Q, \vec{E})$. The optimal solution of the talent scheduling problem can be achieved by invoking **search**$(\vec{B}, Q, \vec{E})$ with $B = E = \emptyset$ and $Q = S$. The function **evaluate**(*solution*) returns the objective value of *solution*. The function **lower_bound**$(\vec{B} \circ s, Q - \{s\}, \vec{E})$ provides a valid lower bound to problem $P(\vec{B} \circ s, Q - \{s\}, \vec{E})$, where the set B of scenes is scheduled before scene s and the set $S - B - \{s\}$ of scenes is scheduled after scene s. If the lower bound (LB) at some branch-and-bound tree node is greater than the upper bound (UB), we discard this node (see lines 12–14, Algorithm 1). Note that UB is a global variable. The branch-and-bound search tries to schedule each remaining scene s immediately after $\vec{B}$, and then swaps the roles of $\vec{B}$ and $\vec{E}$ to continue building the search tree (see line 15, Algorithm 1). Note that we use $R(\vec{B})$ to denote the reverse sequence of $\vec{B}$. For example, if $P(\vec{B}, Q, \vec{E}) = P((1, 2), \{3, 4\}, (5, 6))$, then $P(R(\vec{E}), Q, R(\vec{B})) = P((6, 5), \{3, 4\}, (2, 1))$.

3.2. Lower bound to $P(\vec{B}, Q, \vec{E})$

The problem $P(\vec{B}, Q, \vec{E})$ corresponds to a node in the search tree. Its lower bound **lower_bound**$(\vec{B}, Q, \vec{E})$ can be expressed as:

$$\textbf{lower\_bound}(\vec{B}, Q, \vec{E}) = cost(\vec{B}, \vec{E}) + \textbf{lower}(B, Q, E),$$

where $cost(\vec{B}, \vec{E})$, called *past cost*, is the cost incurred by the path from the root node to the current node, and **lower**(B, Q, E) provides a lower bound to *future cost*, i.e., the holding cost to be incurred by scheduling the scenes in Q. We discuss the past cost $cost(\vec{B}, \vec{E})$ in this section and leave the description of **lower**(B, Q, E) in Section 3.4.

When $\vec{B}$ and $\vec{E}$ have been fixed, a portion of holding cost, namely $cost(\vec{B}, \vec{E})$, is determined regardless of the schedule of the scenes in Q. The past cost $cost(\vec{B}, \vec{E})$ is incurred by the holding days that can be confirmed by the following three ways:

1. For the actor $a_i \in a(B) \cap a(E)$, the number of his/her holding days in any complete schedule can be fixed (Cheng et al., 1993).
2. For the actor $a_i \in a(B) \cap a(Q) - a(E)$, the number of his/her holding days in the time period for completing scenes in B can be fixed.
3. For the actor $a_i \in a(E) \cap a(Q) - a(B)$, the number of his/her holding days in the time period for completing scenes in E can be fixed.

Furthermore, we use $cost(s, B, E)$ to represent the newly confirmed holding cost incurred by placing scene $s \in Q$ at the first unscheduled position, namely the position after any scene in B and before any scene in $S - B - \{s\}$. Note that $cost(s, B, E)$ is irrelevant to the orders of scenes in B and E. Obviously, we have $cost(\vec{B} \circ \{s\}, \vec{E}) = cost(\vec{B}, \vec{E}) + cost(s, B, E)$, which implies that the past cost of a tree node is the sum of the past cost of its father node and the newly confirmed holding cost incurred by branching. As a result, the lower bound function can be rewritten as:

$$\textbf{lower\_bound}(\vec{B} \circ s, Q - \{s\}, \vec{E}) = cost(\vec{B}, \vec{E}) + cost(s, B, E) + \textbf{lower}(B \cup \{s\}, Q - \{s\}, E).$$

The value of $cost(s, B, E)$ is incurred by the following two types of actors:

Type 1. If actor a_i is included in neither $a(B) \cap a(E)$ nor $a(s)$ but is still present on location during the days of shooting scene s (i.e., $a_i \notin a(B) \cap a(E)$, $a_i \notin a(s)$ and $a_i \in a(B) \cap a(Q - \{s\})$), he/she must be held during the shooting days of scene s.

Type 2. If actor a_i is not included in $a(B) \cap a(E)$ but is included in $a(E)$, and scene s is his/her first involved scene (i.e., $a_i \notin a(B)$ and $a_i \in a(s)$ and $a_i \in a(E)$), the shooting days of those scenes in $Q - \{s\}$ that do not require actor a_i can be confirmed as his/her holding days.

To demonstrate the computation of $cost(\vec{B}, \vec{E})$ and $cost(s, B, E)$, let us consider a partial schedule presented in Table 2, where $\vec{B} = (s_1, s_2)$, $\vec{E} = (s_5, s_6)$ and $Q = S - B - E = \{s_3, s_4\}$. In the columns "$cost(\vec{B}, \vec{E})$", "$cost(s_3, B, E)$" and "$cost(s_4, B, E)$", we present the corresponding holding cost associated with each actor. For example, the value of $cost(\vec{B}, \vec{E})$ can be obtained by summing up the values in all cells of the column "$cost(\vec{B}, \vec{E})$". Since actor a_1 is a fixed actor, his/her holding cost must be $c(a_1)(d(s_2) + d(s_4))$ no matter how the scenes in Q are scheduled. Actor a_2 is involved in B and Q but is not involved in E, so we can only say that the holding cost of this actor is at least $c(a_2)d(s_2)$. Similarly, actor a_3 has an already incurred holding cost $c(a_3)d(s_5)$. For actors a_4 and a_5, we cannot get any clue on their holding costs from this partial schedule and thus we say their already confirmed holding costs are both zero. Suppose scene s_4 is placed at the first unscheduled position. Since actors a_2 and a_4 must be present on location during the period of shooting scene s_4, the newly confirmed holding cost is $cost(s_4, B, E) = (c(a_2) + c(a_4))d(s_4)$. If we suppose scene s_3 is placed at the first unscheduled position,

Table 2
An example for computing $cost(\vec{B}, \vec{E})$ and $cost(s, B, E)$.

	$\vec{B}$		Q	$\vec{E}$		$cost(\vec{B}, \vec{E})$	$cost(s_3, B, E)$	$cost(s_4, B, E)$
	s_1	s_2	$\{s_3, s_4\}$	s_5	s_6			
a_1	X	.	{X, ·}	X	X	$c(a_1)(d(s_2) + d(s_4))$	0	0
a_2	X	.	{X, ·}	.	.	$c(a_2)d(s_2)$	0	$c(a_2)d(s_4)$
a_3	.	.	{X, ·}	.	X	$c(a_3)d(s_5)$	$c(a_3)d(s_4)$	0
a_4	.	X	{X, ·}	.	.	0	0	$c(a_4)d(s_4)$
a_5	.	.	{X, ·}	.	.	0	0	0

Table 3
The table for explaining Expression (17).

Actor pattern	B	$\{s\}$	$Q-\{s\}$	E	$o(B)$	$o(E)$	$a(s)$	$o(B)-o(E)$	$o(B)-o(E)-a(s)$	$a(s)-o(B)$	$(a(s)-o(B))\cap o(E)$
1	$\mathbb{X}$	$\mathbb{X}$	$\mathbb{X}$	$\mathbb{X}$	1	1	1	0	0	0	0
2	·	$\mathbb{X}$	$\mathbb{X}$	$\mathbb{X}$	0	1	1	0	0	1	1
3	$\mathbb{X}$	$\mathbb{X}$	$\mathbb{X}$	·	1	0	1	1	0	0	0
4	·	$\mathbb{X}$	$\mathbb{X}$	·	0	0	1	0	0	1	0
5	$\mathbb{X}$	$\mathbb{X}$	·	$\mathbb{X}$	1	1	1	0	0	0	0
6	·	$\mathbb{X}$	·	$\mathbb{X}$	0	1	1	0	0	1	1
7	$\mathbb{X}$	$\mathbb{X}$	·	·	1	0	1	1	0	0	0
8	·	$\mathbb{X}$	·	·	0	0	1	0	0	1	0
9	$\mathbb{X}$	·	$\mathbb{X}$	$\mathbb{X}$	1	1	0	0	0	0	0
10	·	·	$\mathbb{X}$	$\mathbb{X}$	0	1	0	0	0	0	0
11	$\mathbb{X}$	·	$\mathbb{X}$	·	1	0	0	1	1	0	0
12	·	·	$\mathbb{X}$	·	0	0	0	0	0	0	0
13	$\mathbb{X}$	·	·	$\mathbb{X}$	1	1	0	0	0	0	0
14	·	·	·	$\mathbb{X}$	0	0	0	0	0	0	0
15	$\mathbb{X}$	·	·	·	0	0	0	0	0	0	0
16	·	·	·	·	0	0	0	0	0	0	0

the newly confirmed holding cost is only related to actor a_3, namely, $cost(s_3, B, E) = c(a_3)d(s_4)$.

Define $o(Q) = a(S-Q)\cap a(Q)$ as the set of actors required by scenes in both Q and $S-Q$ (de la Banda et al., 2011). Then, $cost(s, B, E)$ can be mathematically computed by:

$$\begin{aligned} cost(s,B,E) &= d(s)\times c(o(B)-o(E)-a(s)) \\ &\quad + \sum_{s'\in Q-\{s\}} d(s')\times (c((a(s)-o(B))\cap o(E)) \\ &\quad - c((a(s)-o(B))\cap o(E)\cap a(s'))), \end{aligned} \tag{17}$$

where $c(G)$ is the total daily cost of all actors in $G\subseteq A$, i.e., $c(G) = \sum_{a\in G} c(a)$.

We use Table 3 to explain Expression (17). All actors can be classified into 16 patterns according to whether they are required by the scenes in sets $B, \{s\}, Q-\{s\}$ and E. If an actor is required by at least one scene in some set, the corresponding cell in columns 2–5 is filled with a sign "$\mathbb{X}$"; otherwise it is filled with a sign "$\cdot$". In columns 6–12, if an actor is included in some actor set, the corresponding cell is filled with "1"; otherwise, it is filled with "0". For example, for pattern 2 actors that has $(B, \{s\}, Q-\{s\}, E) = (\cdot, \mathbb{X}, \mathbb{X}, \mathbb{X})$, we can derive that all actors of this pattern must be included in sets $o(E), a(s), a(s)-o(B)$ and $(a(s)-o(B))\cap o(E)$ and cannot exist in sets $o(B), o(B)-o(E)$ and $o(B)-o(E)-a(s)$.

From Table 3, we can observe that set $o(B)-o(E)-a(s)$ only contains type 1 actors that have pattern $(B, \{s\}, Q-\{s\}, E) = (\mathbb{X}, \cdot, \mathbb{X}, \cdot)$. Thus, the first component of Expression (17) corresponds to type 1 actors. Set $\big(a(s)-o(B)\big)\cap o(E)$ contains type 2 actors that have either pattern $(B, \{s\}, Q-\{s\}, E) = (\cdot, \mathbb{X}, \mathbb{X}, \mathbb{X})$ or pattern $(B, \{s\}, Q-\{s\}, E) = (\cdot, \mathbb{X}, \cdot, \mathbb{X})$. The second component of Expression (17) is the holding cost of type 2 actors during the shooting days for the scenes in $Q-\{s\}$.

3.3. Preprocessing

The holding costs of all fixed actors will not change in the later stages of the search. We use set A_N to contain all *non-fixed actors*, namely $A_N = \{a_i \in A : a_i \notin a(B)\cap a(E)\}$. When solving problem $P(\vec{B}, Q, \vec{E})$, we only need to consider the actors in A_N. As did by de la Banda et al. (2011), the problem $P(\vec{B}, Q, \vec{E})$ can be further simplified as:

- We remove from A_N all actors that are required by only one scene. This is because such actors will not bring about extra holding cost.
- We exclude from A_N all non-fixed actors that are not required by the scenes in Q.

Table 4
An example to illustrate preprocessing steps.

(a) Before preprocessing.

	B	Q				E
		s_1	s_2	s_3	s_4	
a_1	$\mathbb{X}$	X	X	X	X	·
a_2	$\mathbb{X}$	·	X	X	X	·
a_3	·	X	X	X	·	$\mathbb{X}$
a_4	$\mathbb{X}$	·	X	·	·	$\mathbb{X}$
a_5	·	·	·	·	·	$\mathbb{X}$

(b) After preprocessing.

	B	Q			E
		s_1	s_2'	s_4	
a_1	$\mathbb{X}$	X	X	X	·
a_2	$\mathbb{X}$	·	X	X	·
a_3	·	X	X	·	$\mathbb{X}$

- If scenes s_1 and s_2 satisfy $a(s_1)\cap A_N = a(s_2)\cap A_N$, then we replace them with a single scene with duration $d(s) = d(s_1)+d(s_2)$ since they can be regarded as duplicate scenes. The correctness of merging duplicate scenes has been proved by de la Banda et al. (2011).

The example shown in Table 4 illustrates the preprocessing steps. In the problem given by Table 4(a), actor a_4 is fixed and actor a_5 is not required by the scenes in $Q = \{s_1, s_2, s_3, s_4\}$. Therefore, we can remove actors a_4 and a_5 to make $A_N = \{a_1, a_2, a_3\}$. Now since $a(s_2)\cap A_N = a(s_3)\cap A_N = \{a_1, a_2, a_3\}$, we can merge scenes s_2 and s_3. After these preprocessing steps, we get a simplified problem as shown in Table 4(b), where s_2' is the scene created by merging scenes s_2 and s_3.

3.4. Lower bound to future cost

In de la Banda et al. (2011), the authors proposed a lower bound to the future cost. They generated two lower bounds using $(o(B)-F, Q)$ and $(o(E)-F, Q)$ as input information, and claimed that the sum of these two lower bounds is still a lower bound (denoted by L_0) to the future cost. The reader is encouraged to refer to de la Banda et al. (2011) for the details of this lower bound.

In this section, we present a new implementation of **lower**(B, Q, E). Suppose σ is an arbitrary permutation of the scenes in Q. We define x_i as the holding cost of actor a_i during the period of shooting the scenes in Q with the order specified by permutation σ. If **lower**$(B, Q, E) = \min_\sigma\{\sum_{i\in A_N} x_i\}$, we get the minimum possible future cost. However, it is impossible to get the value of $\min_\sigma\{\sum_{i\in A_N} x_i\}$ unless all σ are checked. Instead, we propose a method to produce a lower bound to $\min_\sigma\{\sum_{i\in A_N} x_i\}$.

If an actor a_i satisfies $a_i \notin a(B)$, $a_i \notin a(E)$ and $a_i \in a(Q)$, the lowest possible holding cost of this actor during the period of shooting the scenes in Q may be zero. Therefore, we only consider the actors in set $A_N' = (o(B)-F)\cup(o(E)-F)\subseteq A_N$. For any two different actors $a_i, a_j \in A_N'$, we can derive a constraint $x_i + x_j \geq c_{i,j}$, where $c_{i,j}$ is a constant computed based on the following four cases:

Table 5
Two schedules in Case 1.

(a) The first schedule.

	B	Q								E
		s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	
a_i	$\mathbb{X}$	X	X	X	X	·	·	·	·	·
a_j	$\mathbb{X}$	X	X	·	·	X	X	·	·	·

(b) The second schedule.

	B	Q								E
		s_1	s_2	s_5	s_6	s_3	s_4	s_7	s_8	
a_i	$\mathbb{X}$	X	X	·	·	X	X	·	·	·
a_j	$\mathbb{X}$	X	X	X	X	·	·	·	·	·

Table 6
Two schedules in Case 3.

(a) The first schedule.

	B	Q								E
		s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	
a_i	$\mathbb{X}$	X	X	X	X	·	·	·	·	·
a_j	·	·	·	X	X	·	·	X	X	$\mathbb{X}$

(b) The second schedule.

	B	Q								E
		s_1	s_2	s_5	s_6	s_3	s_4	s_7	s_8	
a_i	$\mathbb{X}$	X	X	·	·	X	X	·	·	·
a_j	·	·	·	·	·	X	X	X	X	$\mathbb{X}$

Case 1: $a_i, a_j \in o(B) - F$. Let $a_i(s) =$ "X" if actor a_i is required by scene s and $a_i(s) =$ "·" otherwise. For any scene $s \in Q$, the tuple ($a_i(s)$, $a_j(s)$) must have one of the following four patterns: (X, X), (X, ·), (·, X), (·, ·). First, we schedule all scenes with pattern (X, X) immediately after the scenes in B and schedule all scenes with pattern (·, ·) immediately before the scenes in E. Second, we group the scenes with (X, ·) and the scenes with (·, X) into two sets. Third, we schedule these two set of scenes in the middle of the permutation, creating two schedules as shown in Table 5. If only actors a_i and a_j are considered, the optimal schedule must be either one of these two schedules. The value of $c_{i,j}$ is set to the holding cost of the optimal schedule related to only actors a_i and a_j. For the schedule in Table 5(a), if we define $S_1 = \{s \in Q | (a_i(s), a_j(s)) = (X, \cdot)\}$, then the holding cost is $c(a_j) \times d(S_1)$, where $d(S_1) = \sum_{s \in S_1} d(s)$. Similarly, for the schedule in Table 5(b), we have a holding cost $c(a_i) \times d(S_2)$, where $S_2 = \{s \in Q | (a_i(s), a_j(s)) = (\cdot, X)\}$. Accordingly, we set $c_{i,j} = \min\{c(a_j) \times d(S_1), c(a_i) \times d(S_2)\}$.

Case 2: $a_i, a_j \in o(E) - F$. We schedule all scenes with pattern (X, X) immediately before the scenes in E and schedule all scenes with pattern (·, ·) immediately after the scenes in B. The remaining analysis is similar to that in Case 1.

Case 3: $a_i \in o(B) - F$ and $a_j \in o(E) - F$. We schedule all scenes with pattern (X, ·) immediately after the scenes in B and schedule all scenes with pattern (·, X) immediately before the scenes in E. If there does not exist a scene with pattern (X, X), the holding cost may be zero and thus $c_{i,j}$ is set to zero; otherwise $c_{i,j}$ is set to $\min\{c(a_i), c(a_j)\} \times d(S_0)$, where $S_0 = \{s \in Q | (a_i(s), a_j(s)) = (\cdot, \cdot)\}$, which can be observed from Table 6.

Case 4: $a_i \in o(E) - F$ and $a_j \in o(B) - F$. This case is the same as Case 3.

A valid lower bound to the future cost (i.e., the value of **lower**(B, Q, E)) can be obtained by solving the following linear programming model:

$$(LB)\ z^{LB} = \min \sum_{a_i \in A'_N} x_i \tag{18}$$

$$\text{s.t. } x_i + x_j \geq c_{i,j},\ \forall a_i, a_j \in A'_N, i \neq j \tag{19}$$

$$x_i \geq 0,\ \forall a_i \in A'_N \tag{20}$$

The value of z^{LB} must be a valid lower bound to $\min_\sigma \{\sum_{i \in A_N} x_i\}$. If the daily holding cost of actor a_i is an integral number, decision variable x_i should be integer. When all variables x_i are integers, the model (LB) is an NP-hard problem since it can be easily reduced to the *minimum vertex cover problem* (Karp, 1972). If all variables x_i are treated as real numbers, this model can be solved by a liner programming solver. For some instances, the (LB) model needs to be solved a very large number of times. To save computation time, we apply the following two heuristic approaches to rapidly produce two lower bounds, i.e., L_1 and L_2, to z^{LB}. Obviously, L_1 and L_2 are also valid lower bounds to the future cost.

Approach 1: Sum up the left-hand-side and righ-hand-side of Eq. (19), generating $(|A'_N| - 1) \sum_{a_i \in A'_N} x_i \geq \sum_{a_i, a_j \in A'_N, i \neq j} c_{i,j}$. The valid lower bound L_1 is defined as:

$$L_1 = \sum_{a_i, a_j \in A'_N, i \neq j} c_{i,j} / (|A'_N| - 1).$$

Approach 2: Sort $c_{i,j}$ in descending order. If we select a $c_{i,j}$, we call the corresponding x_i and x_j *marked*. Beginning from the largest $c_{i,j}$, we select all $c_{i,j}$ whose x_i and x_j are not marked until all x_i are marked. The valid lower bound L_2 equals the sum of all selected $c_{i,j}$. This approach was termed the *greedy matching algorithm* (Drake & Hougardy, 2003). To demonstrate the process of computing L_2, we consider the following six constraints:

$$x_1 + x_2 \geq 2,\ x_1 + x_3 \geq 7,\ x_1 + x_4 \geq 6,$$
$$x_2 + x_3 \geq 12,\ x_2 + x_4 \geq 8,\ x_3 + x_4 \geq 5.$$

We first select $c_{2,3} = 12$ and mark x_2 and x_3. Then, we can only select $c_{1,4} = 6$ since x_1 and x_4 have not been marked. Now all x_i are marked and the value of L_2 equals 18.

In our algorithm, we set **lower**(B, Q, E) $= \max\{L_0, L_1, L_2\}$.

3.5. Caching search states

In de la Banda et al. (2011), the talent scheduling problem was solved by a double-ended dynamic programming (DP) algorithm, where a DP state is represented by $\langle B, E \rangle$. The DP algorithm stores the best value of each examined state, denoted by $\langle B, E \rangle.value$, which equals the minimum past cost of all search paths associated with sets B and E.

We embed this DP process in our branch-and-bound framework by use of *memoization* technique (Michie, 1968). More precisely, when the search process reaches a tree node $P(\vec{B}, Q, \vec{E})$, it first checks whether the value of $cost(\vec{B}, \vec{E})$ is less than the current $\langle B, E \rangle.value$. If so, it updates $\langle B, E \rangle.value$ by $cost(\vec{B}, \vec{E})$; otherwise, the current node must be dominated by some node and therefore can be safely discarded.

A better state representation for the DP algorithm is $\langle o(B), o(E), Q \rangle$, where $Q = S - B - E$; this was discussed by de la Banda et al. (2011) as follows. The cost of scheduling the scenes in $Q = S - B - E$ depends on $o(B)$ and $o(E)$ rather than B and E. Suppose $\vec{B}\vec{Q}\vec{E}$ and $\vec{B}'\vec{Q}\vec{E}'$ are two permutations of S, where B, Q, E, B' and E' are the corresponding sets of scenes. If $o(B) = o(B')$ and $o(E) = o(E')$, then the holding costs incurred by $\vec{Q}$ in these two permutations are equal. Moreover, if there are two states $\langle o(B), o(E), Q\rangle$ and $\langle o(B'), o(E'), Q\rangle$ that have $o(B) = o(E')$ and $o(E) = o(B')$, they are equivalent due to the symmetric property of the problem. Thus, we only need to memoize the state $\langle o(B), o(E), Q\rangle$ that satisfies $o(B) \le o(E)$. We compare $o(B)$ with $o(E)$ based on the lexicographical order of the actor indices. For example, given $o(B) = \{a_1, a_2, a_4, a_5\}$ and $o(E) = \{a_1, a_3, a_6, a_7\}$, we have $o(B) \le o(E)$ since the index of a_2 is less than that of a_3.

We also use the memoization technique to prune the search tree node. The process of checking whether a given node associated with problem $P(\vec{B}, Q, \vec{E})$ can be pruned is depicted in Algorithm 2. All states

Algorithm 2 The process of checking whether a given search node can be pruned.

```
1: Function: check($\vec{B}, Q, \vec{E}$);
2: $pc = cost(\vec{B}, \vec{E})$;
3: $index = $ hash$(o(B), o(E), Q)$;
4: if $hashTable[index].state \sim \langle o(B), o(E), Q\rangle$ and
   $hashTable[index].value \le pc$ then
5:     return true;
6: end if
7: for all $s \in Q$ do
8:     $index2 = hash(o(B), o(E), Q - \{s\})$;
9:     if $hashTable[index2].state \sim \langle o(B), o(E), Q - \{s\}\rangle$ and
       $hashTable[index2].value \le pc$ then
10:        return true;
11:    end if
12: end for
13: if replace$(index, pc)$ = true then
14:    $hashTable[index].state = \langle o(B), o(E), Q\rangle$; {// If we decide to replace the information in the storage place $index$, then store the state and its value.}
15:    $hashTable[index].value = pc$;
16: end if
17: return false.
```

are stored in a hash table *hashTable*. This algorithm first designates a storage slot in the hash table for state $\langle o(B), o(E), Q\rangle$ using function **hash(**$o(B), o(E), Q$**)**. This hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored. First, we transform our state to a search key. Let us consider an example with four actors and five scenes. If $(o(B), o(E), Q) = (\{1, 3, 4\}, \{2, 4\}, \{1, 2, 3, 4\})$, its binary code (i.e., search key) is 1011010111110, where $\{1, 3, 4\}$, $\{2, 4\}$ and $\{1, 2, 3, 4\}$ correspond to 1011, 0101 and 11110, respectively. Note that $o(B)$ and $o(E)$ are two sets of actors and Q is a set of scenes. Second, we get the corresponding decimal number of the binary code. Third, we calculate "the decimal number mod C", where C is the number of storage slots, to obtain the index of the storage space. In each storage space, a value is stored.

If the storage slot contains the state and the current value of the state is less than or equal to $cost(\vec{B}, \vec{E})$, the algorithm returns *true*, implying that the given node can be pruned (see lines 4–5, Algorithm 2). Next, it checks whether the state $\langle o(B), o(E), Q - \{s\}\rangle$ $(s \in Q)$ exists in the hash table and has a value less than or equal to $cost(\vec{B}, \vec{E})$ (see lines 7–12, Algorithm 2). If such state exists, the given node can also be pruned. The correctness of this pruning condition is guaranteed by Property 1, which was derived from the second theorem in de la Banda et al. (2011).

Property 1. Suppose $\vec{B}\vec{Q}\vec{E}$ and $\vec{B'}\vec{Q'}\vec{E'}$ are two permutations of S, where B, Q, E, B', Q' and E' are the corresponding sets of scenes. If $o(B) = o(B')$, $o(E) = o(E')$, $Q \subseteq Q'$ and the scenes in $\vec{Q}$ follow the order in which they appear in $\vec{Q'}$, then the holding cost incurred by $\vec{Q}$ is not greater than that incurred by $\vec{Q'}$.

Ideally, the hash function should assign each state to a unique storage slot, i.e., no hash collisions happen. However, this ideal situation is rarely achievable due to the huge number of states and the inadequate storage space. When solving the talent scheduling problem, we do not have sufficient storage space to store the exponential number of search states and therefore different states may be assigned by the hash function to the same storage slot, leading to hash collisions. To resolve this issue, we employ a mechanism called *direct mapped caching scheme*. Assume the direct mapped cache consists of C slots, each of which can only store one item. If an item is to be stored in a slot that already contains another item (i.e., a hash collision occurs), it may either replace the existing item or be discarded, which is decided by function **replace**$(index, pc)$. Several previous articles, such as Hilden (1976) and Pugh (1988), have discussed the replacement strategies implemented in **replace**$(index, pc)$. In this work, we tried *latest* and *greedy* caching strategies. The first strategy deals with the hash collisions by simply overwriting the cache slot while the second one stores in the cache slot the item that has smaller value. If we choose the latest caching strategy, **replace**$(index, pc)$ always returns true. If the greedy caching strategy is selected and the new state has a value less than that of the existing state, **replace**$(index, pc)$ returns true.

The direct mapped caching scheme can effectively prune the search nodes using limited storage space. When a state is revisited again but it has been removed from the cache during the previous stages, the search can still continue to explore its corresponding subtree. In Section 4, we experimentally analyze the impact of different values of C and the two replacement strategies on the performance of our branch-and-bound algorithm.

3.6. Dominance rules

Dominance rules were widely used in branch-and-bound algorithms (Braune, Zäpfel, & Affenzeller, 2012; Kellegöz & Toklu, 2012; Ranjbar, Davari, & Leus, 2012; Zhang, Qin, Zhu, & Lim, 2012) and dynamic programming algorithms (Dumas, Desrosiers, Gelinas, & Solomon, 1995; Mingozzi, Bianco, & Ricciardelli, 1997; Rong & Figueira, 2013) for eliminating search states. The purpose of dominance rules is to identify the partial solution represented by a node in the search tree that is dominated by another partial solution. The dominated partial solution need not be further explored and can be safely pruned. In our branch-and-bound algorithm, two new dominance rules are introduced to reduce the search space.

3.6.1. Dominance Rule 1

At a branch-and-bound tree node associated with problem $P(\vec{B}, Q, \vec{E})$, we suppose that scene s_1 is the scene to be scheduled immediately after B and scene s_2 belongs to $Q - \{s_1\}$. If $a(s_1) \cup o(B) \supseteq a(s_2) \cup o(B)$ and $a(s_1) \cup o(E) \subseteq a(s_2) \cup o(E)$, then the branch associated with scene s_1 can be ignored.

Tables 7 and 8 are used to explain this dominance rule. In Table 7, $Q = \{s_1, s_2\} \cup \Omega_1 \cup \Omega_2$, where Ω_1 and Ω_2 are two arbitrary subsets of $Q - \{s_1, s_2\}$ and $\Omega_1 \cap \Omega_2 = \emptyset$. Actors in A_N can be classified into twelve patterns according to whether they are required by the scenes in sets B, E, $\{s_1\}$ and $\{s_2\}$. Since we do not need the information related to Ω_1 and Ω_2, all cells in columns 4 and 6 remain empty. Similar to Table 3, the numbers 1 and 0 in the right part of Table 7 indicate whether an actor is included in the corresponding actor set.

In the absence of the information in columns 4 and 6, we cannot directly judge whether pattern 4 actors are included in $o(B)$ and whether pattern 8 actors are included in $o(E)$. However, we know that all remaining actors are non-fixed and must be required by the scenes in Q. In other words, if some pattern 4 and 8 actors are kept in A_N, then they must be required by some scene in $\Omega_1 \cup \Omega_2$. Therefore, we fill the corresponding cells with "1" (see the numbers in bold in Table 7).

Table 7
The table for explaining dominance rules.

Actor pattern	B	$\{s_1\}$	Ω_1	$\{s_2\}$	Ω_2	E	$o(B)$	$o(E)$	$a(s_1)\cup o(E)$	$a(s_1)\cup o(B)$	$a(s_2)\cup o(E)$	$a(s_2)\cup o(B)$	$(a(s_1)\cup o(B))\cap(a(s_2)\cup o(E))$
1	𝕏	𝕏		𝕏		·	1	0	1	1	1	1	1
2	𝕏	𝕏		·		·	1	0	1	1	0	1	0
3	𝕏	·		𝕏		·	1	0	0	1	1	1	1
4	𝕏	·		·		·	**1**	0	0	1	0	1	0
5	·	𝕏		𝕏		𝕏	0	1	1	1	1	1	1
6	·	𝕏		·		𝕏	0	1	1	1	1	0	1
7	·	·		𝕏		𝕏	0	1	1	0	1	1	0
8	·	·		·		𝕏	0	**1**	1	0	1	0	0
9	·	𝕏		𝕏		·	0	0	1	1	1	1	1
10	·	𝕏		·		·	0	0	1	1	0	0	0
11	·	·		𝕏		·	0	0	0	0	1	1	0
12	·	·		·		·	0	0	0	0	0	0	0

Table 8
Actor patterns before and after exchanging scenes s_1 and s_2.

Actor pattern	B	$\{s_1\}$	Ω_1	$\{s_2\}$	Ω_2	E	Actor pattern	B	$\{s_2\}$	Ω_1	$\{s_1\}$	Ω_2	E
1	𝕏	𝕏		𝕏		·	1	𝕏	𝕏		𝕏		·
$\underline{3}$	𝕏	·		𝕏		·	$\underline{3}$	𝕏	𝕏		·		·
4	𝕏	·		·		·	4	𝕏	·		·		·
5	·	𝕏		𝕏		𝕏	5	·	𝕏		𝕏		𝕏
$\underline{6}$	·	𝕏		·		𝕏	$\underline{6}$	·	·		𝕏		𝕏
8	·	·		·		𝕏	8	·	·		·		𝕏
9	·	𝕏		𝕏		·	9	·	𝕏		𝕏		·
12	·	·		·		·	12	·	·		·		·

Table 9
Actor patterns before and after shifting scene s_2 immediately before scene s_1.

Actor pattern	B	$\{s_1\}$	Ω_1	$\{s_2\}$	Ω_2	E	Actor pattern	B	$\{s_2\}$	$\{s_1\}$	Ω_1	Ω_2	E
1	𝕏	𝕏		𝕏		·	1	𝕏	𝕏	𝕏			·
$\underline{2}$	𝕏	𝕏		·		·	$\underline{2}$	𝕏	·	𝕏			·
3	𝕏	·		𝕏		·	3	𝕏	𝕏	·			·
$\underline{4}$	𝕏	·		·		·	$\underline{4}$	𝕏	·	·			·
5	·	𝕏		𝕏		𝕏	5	·	𝕏	𝕏			𝕏
$\underline{6}$	·	𝕏		·		𝕏	$\underline{6}$	·	·	𝕏			𝕏
8	·	·		·		𝕏	8	·	·	·			𝕏
9	·	𝕏		𝕏		·	9	·	𝕏	𝕏			·
10	·	𝕏		·		·	10	·	·	𝕏			·
12	·	·		·		·	12	·	·	·			·

We list in the left part of Table 8 all actor patterns that satisfy the conditions $a(s_1)\cup o(B)\supseteq a(s_2)\cup o(B)$ and $a(s_1)\cup o(E)\subseteq a(s_2)\cup o(E)$. Table 8 shows that branching to scene s_1 is dominated by branching to scene s_2. After exchanging the positions of scenes s_1 and s_2, the holding costs for pattern 1, 4–5, 8–9 and 12 actors remain unchanged while the holding costs for pattern 3 and 6 actors are probably reduced. Thus, scheduling scene s_2 immediately after B must result in less or equal holding cost than scheduling scene s_1 at that position.

3.6.2. *Dominance Rule 2*

At a branch-and-bound tree node associated with problem $P(\vec{B}, Q, \vec{E})$, we suppose that s_1 is the scene to be scheduled immediately after B and s_2 belongs to $Q - \{s_1\}$. If $a(s_1)\cup o(B)\supseteq a(s_2)\cup o(B)$ and $c((a(s_1)\cup o(B))\cap(a(s_2)\cup o(E))) - c(a(s_2)\cup o(B)) > 0$, then the branch associated with scene s_1 can be ignored.

We list in the left part of Table 9 all actor patterns that satisfy the conditions $a(s_1)\cup o(B)\supseteq a(s_2)\cup o(B)$. The right part of Table 9 is the result of shifting scene s_2 immediately before scene s_1 and immediately after B. From Table 9, we can get the following four observations: (1) the holding costs for pattern 5 actors remain unchanged; (2) the holding costs for pattern 1, 3, 8–10 and 12 actors are probably reduced; (3) the holding cost of each actor a_i with pattern 2 or 4 is probably increased by $c(a_i)d(s_2)$; (4) the holding cost of each pattern 6 actor a_i is definitely decreased by $c(a_i)d(s_2)$. If the decreased amount (related to pattern 6 actors) is greater than the increased amount (related to pattern 2 and 4 actors), then shifting scene s_2 immediately before scene s_1 must lead to a cost reduction. Given that $a(s_1)\cup o(B)\supseteq a(s_2)\cup o(B)$ is satisfied, the set $a(s_1)\cup o(B))\cap(a(s_2)\cup o(E)$ includes pattern 1, 3, 5–6 and 9 actors and the set $a(s_2)\cup o(B)$ includes patterns 1–5, and 9 actors. This means both $a(s_1)\cup o(B))\cap(a(s_2)\cup o(E)$ and $a(s_2)\cup o(B)$ include pattern 1, 3, 5, 9 actors. So we can derive that $c((a(s_1)\cup o(B))\cap(a(s_2)\cup o(E))) - c(a(s_2)\cup o(B))$ is equal to the cost of all pattern 6 actors minus the cost of all pattern 2 and 4 actors. Thus, if $a(s_1)\cup o(B)\supseteq a(s_2)\cup o(B)$ and $c((a(s_1)\cup o(B))\cap(a(s_2)\cup o(E))) - c(a(s_2)\cup o(B)) > 0$, scheduling scene s_2 immediately after B must result in less or equal holding cost than scheduling scene s_1 at that position.

3.7. *The enhanced branch-and-bound algorithm*

Our enhanced branch-and-bound algorithm for the talent scheduling problem is given by Algorithm 3, where the value of past cost z is initialized to zero at the root node. The preprocessing stage is realized by function **preprocess**(Q, A_N) (see line 9, Algorithm 3). The state caching technique is adopted through function **check**$(\vec{B}, Q, \vec{E})$ (see line 10, Algorithm 3), where the details of this function is described in Algorithm 2. The function **isDominated**$(\vec{B}, Q, \vec{E}, A_N, z, s)$ employs the proposed two dominance rules to check whether

Table 10
Computational results for Type 1 Data Set.

Instance	m	n	Smith (2005)		de la Banda et al. (2011)		Enhanced branch-and-bound		Total cost	Holding cost
			Time (s)	Subproblems	Time (s)	Subproblems	Time (s)	Subproblems		
MobStory	8	28	64.71	136,765	0.11	6605	0.05	849	871	146
film103	8	19	76.69	180,133	0.06	4103	0.02	828	1031	187
film105	8	18	16.07	40,511	0.02	1108	0.02	215	849	110
film114	8	19	127	267,526	0.08	4957	0.03	2027	867	143
film116	8	19	125.8	225,314	0.16	13,576	0.03	1937	541	110
film117	8	19	76.86	174,100	0.10	7227	0.02	987	913	197
film118	8	19	93.1	205,190	0.04	1980	0.02	537	853	156
film119	8	18	70.8	144,226	0.08	7105	0.02	580	790	159

Algorithm 3 The enhanced double-ended branch-and-bound algorithm for the talent scheduling problem.

```
1:  Function: search($\vec{B}, Q, \vec{E}, A_N, z$);
2:  if $Q = \emptyset$ then
3:    if $z < UB$ then
4:      $UB = z$;
5:      $best\_solution = \vec{B} \circ \vec{E}$;
6:    end if
7:    return ;
8:  end if
9:  $(Q, A_N) =$ preprocess$(Q, A_N)$;
10: if check($\vec{B}, Q, \vec{E}$) = true then
11:   return ; {// Terminate the algorithm since this state can be pruned.}
12: end if
13: for all $s \in Q$ do
14:   if isDominated($\vec{B}, Q, \vec{E}, A_N, z, s$) = true then
15:     continue; {// Terminate the current iteration and consider the next scene.}
16:   end if
17:   $LB = z + cost(s, B, E) +$ lower$(B \cup \{s\}, Q - \{s\}, E)$;
18:   if $LB \geq UB$ then
19:     continue; {// Terminate the current iteration and consider the next scene.}
20:   end if
21:   Invoke search$(R(\vec{E}), Q - \{s\}, \{s\} \circ R(\vec{B}), A_N, z + cost(s, B, E))$.
22: end for
```

branching to some scene s is dominated by other branches. In function **isDominated**$(\vec{B}, Q, \vec{E}, A_N, z, s)$, we check in turn whether s can be dominated by the scene in $Q - \{s\}$. If scene s can be dominated, then this function returns true and thus the corresponding node can be eliminated. The function **lower**$(B \cup \{s\}, Q - \{s\}, E)$ returns a valid lower bound to the future cost of the problem at some search node.

4. Computational experiments

Our algorithm was coded in C++ and compiled using the g++ compiler. All experiments were run on a Linux server equipped with an Intel Xeon E5430 CPU clocked at 2.66 gigahertz and 8 gigabytes RAM. The algorithm only has two parameters, namely the number (C) of cached states and the caching strategy used. After some preliminary experiments, we set $C = 2^{25}$ and chose the *greedy* caching strategy when solving the benchmark instances. In this section, we first present our results for the benchmark instances and then compare them with the results obtained by the best two existing approaches. Finally, we exhibit by experiments the impacts of the parameters on the overall performance of the algorithm. All computation times

Table 11
The number of optimally solved instances in each Type 2 instance group.

n	m							
	8	10	12	14	16	18	20	22
16	100	100	100	100	100	100	100	100
18	100	100	100	100	100	100	100	100
20	100	100	100	100	100	100	100	100
22	100	100	100	100	100	100	100	100
24	100	100	100	100	100	100	100	100
26	100	100	100	100	100	100	100	100
28	100	100	100	100	100	100	100	100^*
30	100	100	100	100	100	100^*	100^*	100^*
32	100	100	100	100	100^*	100^*	100^*	100^*
34	100	100	100	100^*	100^*	100^*	100^*	99^*
36	100	100	100	100^*	100^*	100^*	99^*	98^*
38	100	100	100^*	100^*	100^*	99*	99^*	98^*
40	100	100	100^*	99^*	100^*	98^*	97^*	98^*
42	100	100	100^*	100^*	100^*	96^*	97^*	94^*
44	100	100^*	100^*	100^*	97^*	99^*	93^*	$\underline{79}$
46	100	100^*	100^*	100^*	97^*	97^*	88^*	$\underline{71}$
48	100	100^*	100^*	100^*	99^*	96^*	88^*	$\underline{64}$
50	100	100^*	100^*	99^*	96^*	94^*	84^*	$\underline{59}$
52	100	100^*	100^*	98^*	99^*	87^*	$\underline{70}$	$\underline{52}$
54	100	100^*	98^*	96^*	89^*	$\underline{75}$	$\underline{63}$	$\underline{41}$
56	100	100^*	99^*	98^*	93^*	85^*	$\underline{61}$	$\underline{44}$
58	100	100^*	100^*	97^*	82^*	$\underline{77}$	$\underline{54}$	$\underline{37}$
60	100	100^*	99^*	89^*	87^*	$\underline{72}$	$\underline{53}$	$\underline{34}$
62	100	100^*	96^*	96^*	$\underline{75}$	$\underline{62}$	$\underline{50}$	$\underline{43}$
64	100^*	100^*	98^*	95^*	$\underline{74}$	$\underline{51}$	$\underline{43}$	$\underline{23}$

reported here are in CPU seconds on this server. All instances and detailed results are available in the online supplement to this paper at: http://www.tigerqin.com/publicatoins/talent-scheduling-problem.

4.1. Results for benchmark instances

In order to evaluate our algorithm, we conducted experiments using two benchmark data sets (Types 1 and 2), downloaded from http://people.eng.unimelb.edu.au/pstuckey/talent/. The Type 1 data set was introduced by Cheng et al. (1993) and Smith (2005), including seven instances, namely *MobStory*, *film103*, *film105*, *film114*, *film117*, *film118* and *film119*. Since these instances have small sizes, ranging from 18 × 8 (18 scenes by 8 actors) to 28 × 8, they were easily solved to optimality. Table 10 shows the results obtained by our branch-and-bound algorithm, the constraint programming approach in Smith (2005) and the dynamic programming algorithm in de la Banda et al. (2011). From this table, we can see that our algorithm reduced the number of subproblems significantly for each instance with much less computational efforts. In our branch-and-bound algorithm, a subproblem corresponds to a search tree node. Note that the results taken from Smith (2005) were produced on a PC with 1.7 gigahertz Pentium M processor, and the results from de la Banda et al. (2011) were produced on a machine with Xeon Pro 2.4 gigahertz processors and 2 gigabytes RAM.

The Type 2 data set was provided by de la Banda et al. (2011). Following a manner almost identical to that used by Cheng et al. (1993), de la Banda et al. (2011) randomly generated 100 instances for each combination of $n \in \{16, 18, 20, \ldots, 64\}$ and $m \in \{8, 10, 12, \ldots, 22\}$, for a total of 200 instance groups and 20,000 instances. They tried to solve these instances using their dynamic programming algorithm with a memory bound of 2 gigabytes. For each instance, if the execution did not run out of memory, they recorded the running time and the number of subproblems generated. They reported the average running time and the average number of subproblems for each Type 2 instance group with more than 80 optimally solved instances; these two average values were computed based on the optimally solved instances.

We tried to solve all Type 2 instances using our branch-and-bound algorithm with a time limit of 10 minutes and a memory of 2 gigabytes. Our algorithm requires some memory to store the information of the search tree and a limited number of states. The amount of memory available can fully satisfy this requirement and thus the out-of-memory exception did not occur. Table 11 gives the number of instances optimally solved in each Type 2 instance group, where an

Table 12
The average running time (in seconds) for each Type 2 instance group.

n	m							
	8	10	12	14	16	18	20	22
16	0.004	0.003	0.006	0.012	0.083	0.089	0.121	0.175
18	0.006	0.009	0.011	0.020	0.123	0.119	0.134	0.242
20	0.013	0.015	0.020	0.032	0.153	0.126	0.166	0.291
22	0.018	0.021	0.030	0.048	0.142	0.199	0.236	0.388
24	0.022	0.033	0.047	0.065	0.185	0.232	0.426	0.704
26	0.032	0.043	0.068	0.101	0.274	0.504	0.738	1.058
28	0.038	0.068	0.146	0.235	0.398	0.663	1.461	2.433
30	0.055	0.083	0.155	0.333	0.640	1.548	2.183	5.541
32	0.069	0.097	0.219	0.566	1.613	2.496	4.822	16.342
34	0.085	0.160	0.281	1.063	1.672	4.933	10.761	17.891
36	0.107	0.202	0.898	2.254	3.269	7.848	24.991	29.444
38	0.108	0.401	0.619	3.153	5.842	18.746	36.231	48.581
40	0.154	0.374	0.915	2.197	12.178	12.270	38.954	53.126
42	0.198	0.490	1.323	4.602	14.271	31.411	46.482	100.084
44	0.825	0.826	2.525	7.661	18.215	57.402	64.493	85.409
46	0.935	2.986	2.604	9.393	19.240	35.505	84.877	104.940
48	0.892	1.404	6.162	11.908	39.599	61.703	90.863	107.725
50	0.953	1.791	10.579	20.588	48.641	71.460	111.824	120.828
52	0.995	3.819	13.127	37.832	40.237	97.423	108.915	130.559
54	1.525	3.380	19.269	22.891	67.856	97.367	134.863	163.487
56	1.393	3.770	15.025	49.464	50.699	100.411	134.661	149.144
58	1.867	6.014	30.280	37.760	61.514	117.141	158.239	173.229
60	2.156	7.378	22.250	70.460	97.438	146.204	105.785	207.466
62	1.626	10.162	27.478	64.337	128.221	158.739	167.173	172.262
64	3.918	12.140	43.397	55.849	96.881	161.446	156.240	182.860

Table 13
The average number of search nodes for each Type 2 instance group.

n	m							
n	8	10	12	14	16	18	20	22
16	92	132	262	305	511	585	959	1044
18	173	288	381	579	1051	1650	1920	2490
20	278	464	797	1248	1559	2969	3763	5081
22	416	673	1116	2439	3214	5232	7791	10,174
24	480	1300	2104	3521	6002	8872	16,824	26,084
26	950	1685	3526	5860	10,500	23,521	33,464	42,133
28	1014	3504	8916	15,068	18,140	33,917	66,739	103,750
30	2052	4472	9943	21,104	34,806	79,595	105,243	226,899
32	3415	4779	15,232	36,176	93,686	130,977	235,314	665,416
34	4022	10,115	18,945	71,387	95,621	247,737	503,058	767,987
36	4974	12,923	61,748	140,081	183,606	392,267	1,150,811	1,266,911
38	3993	29,039	43,450	196,504	313,253	928,228	1,554,085	1,984,941
40	7703	25,478	64,107	141,726	618,351	616,794	1,728,150	2,221,610
42	11,143	32,937	89,721	286,598	754,305	1,356,438	2,120,414	4,059,816
44	16,423	59,929	135,760	448,782	947,224	2,620,935	2,794,299	3,453,335
46	22,400	147,524	145,857	529,693	1,031,659	1,684,446	3,729,016	4,316,409
48	25,743	74,511	364,220	659,308	1,953,064	2,823,159	3,983,337	4,556,976
50	29,874	85,712	567,577	1,088,311	2,384,690	3,159,260	4,686,411	4,891,403
52	31,840	210,741	716,255	1,979,313	1,973,444	4,318,443	4,747,867	5,172,142
54	64,838	192,700	1,130,890	1,208,212	3,341,742	4,504,014	5,936,410	6,539,519
56	58,341	218,177	867,317	2,644,071	2,493,294	4,467,560	5,734,008	5,948,378
58	87,367	346,222	1,711,089	1,950,019	2,943,275	5,249,511	6,886,448	6,844,526
60	100,968	402,869	1,245,312	3,676,459	4,677,957	6,569,635	4,345,421	8,538,072
62	64,903	546,580	1,530,491	3,163,571	5,765,984	6,796,924	7,023,583	6,804,257
64	166,009	655,791	2,308,876	2,718,102	4,356,074	6,882,350	6,422,940	7,377,196

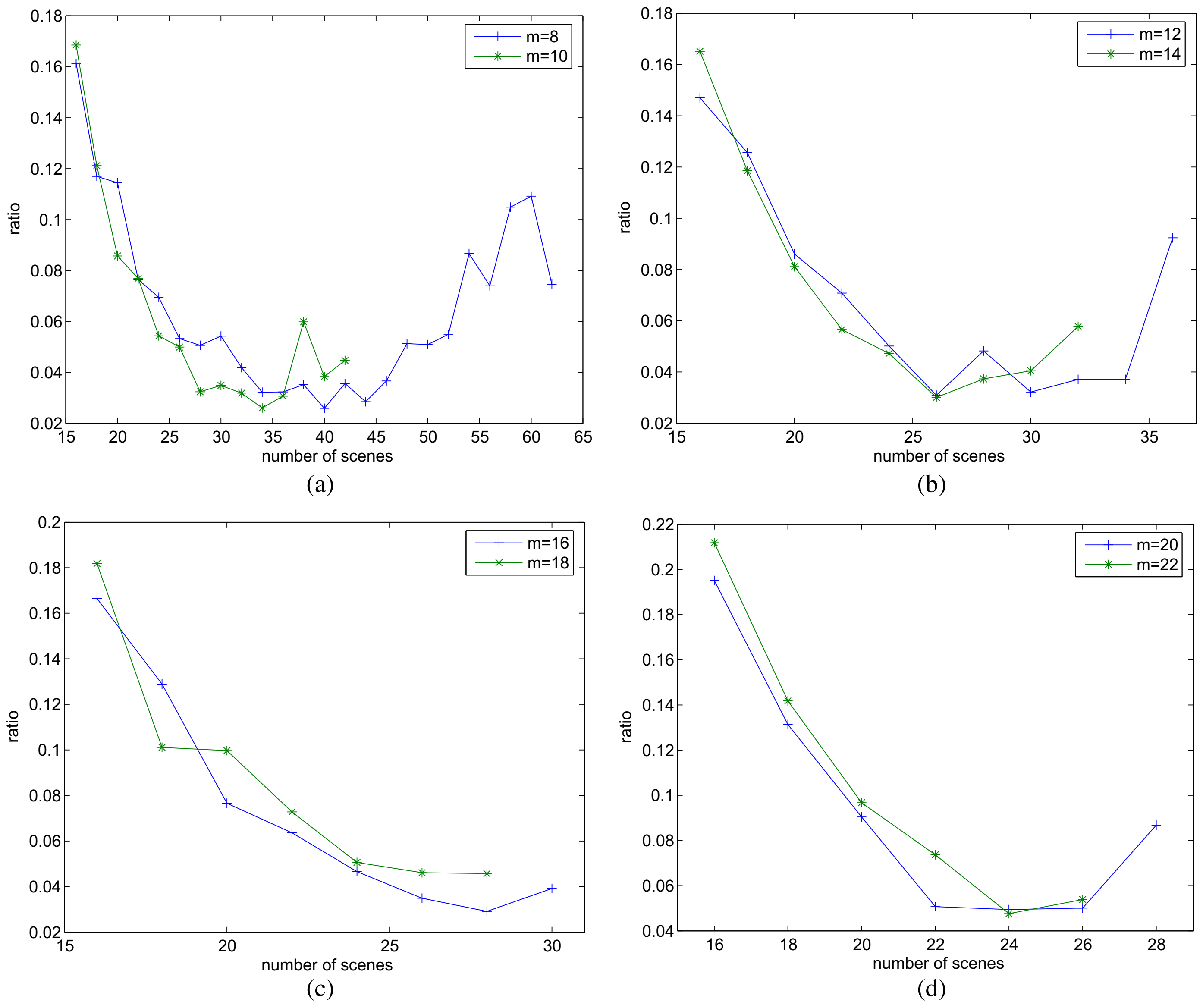

Fig. 1. (a) $m = \{8, 10\}$. (b) $m = \{12, 14\}$. (c) $m = \{16, 18\}$. (d) $m = \{20, 22\}$.

underline sign ("_") is added to the cell associated with the instance group with less than 80 optimally solved instances. For an instance group, if our algorithm optimally solved 80 or more instances while the dynamic programming algorithm failed to achieve so, the number in its corresponding cell is marked with an asterisk (*). From this table, we can see that our algorithm managed to optimally solve all instances with the number of scenes (n) not greater than 32 or the number of actors (m) not greater than 10. However, the dynamic programming algorithm by de la Banda et al. (2011) only optimally solved more than 80 out of 100 instances for the instance groups with $n \leq 26$. Their approach even did not optimally solve all instances with $m = 8$ and $n = 64$. In this table, 89 out of 200 instance groups are marked with asterisks, which clearly indicates that more Type 2 benchmark instances were successfully solved to optimality by our branch-and-bound algorithm. Although our machine is slightly more powerful, this cannot account for the dramatic difference in the number of optimally solved instances, it is reasonable to conclude that our branch-and-bound algorithm is more efficient than the dynamic programming algorithm.

Tables 12–13 show the average running time and the average number of search nodes, respectively, over all optimally solved instances for each instance group. Like in Table 11, the instance groups with less than 80 optimally solved instances are marked with "_". From Table 13, we can easily find that the average number of search nodes generated for each instance group with "_" exceeds 3,000,000.

To further compare our results with those reported by de la Banda et al. (2011), we pictorially show in Fig. 1 the ratio of the average number of subproblems (i.e., search nodes) generated by our algorithm to that generated by the dynamic programming algorithm. Each point in these curves corresponds to an instance group whose average number of subproblems was reported by de la Banda et al. (2011). On average, the number of subproblems generated by our algorithm is less than 22 percent of that generated by the dynamic programming algorithm, which should be attributed to the use of the new lower bound and domination rules. Moreover, we can observe some trends from these curves. The ratio first decreases as the number of scenes increases, which implies that our algorithm can eliminate more subproblems. Then, the ratio increases with the number of scenes. This is because hash collisions happened more frequently, reducing the opportunities of pruning search nodes and therefore increasing the number of subproblems.

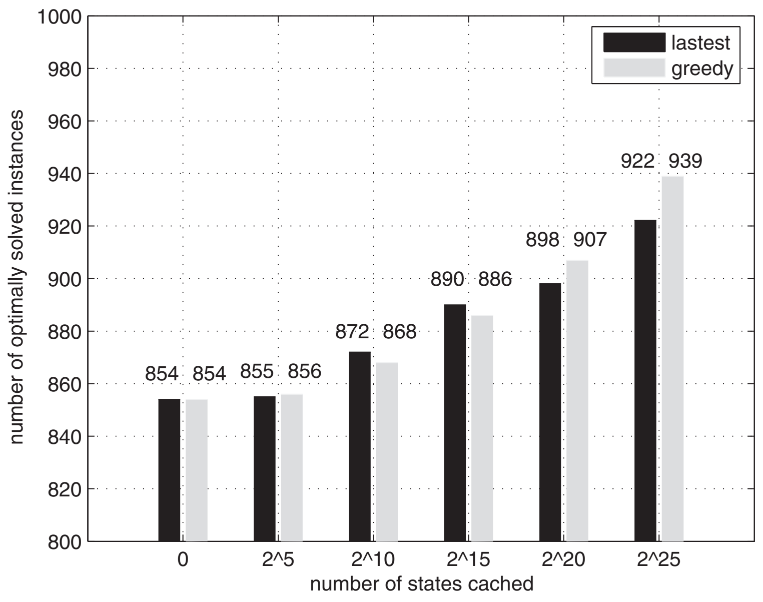

Fig. 2. The impact of different parameter settings on the number of optimally solved instances.

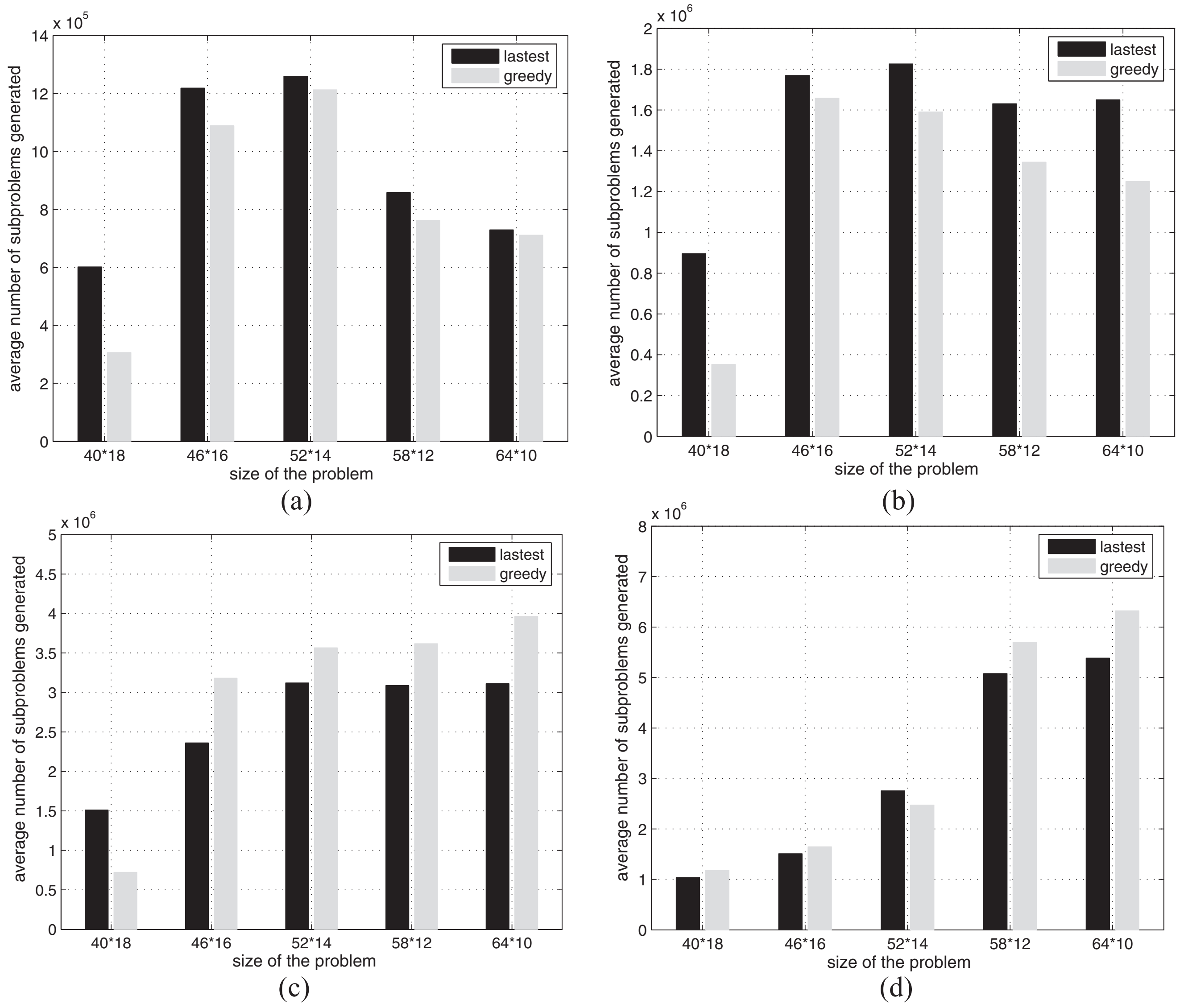

Fig. 3. (a) $C = 2^{10}$. (b) $C = 2^{15}$. (c) $C = 2^{20}$. (d) $C = 2^{25}$.

4.2. Impacts of parameter settings

We take the value of C from $\{0, 2^5, 2^{10}, 2^{15}, 2^{20}, 2^{25}\}$, where $C = 0$ means that cache is not used. Considering the two caching strategies, we have 12 parameter combinations in total. We tested these 12 parameter combinations using a portion of the Type 2 instances. Specifically, the first 5 instances were selected from each instance group, for a total of 1000 instances. We also imposed a time limit of 10 minutes on each execution of our algorithm. The results of those optimally solved instances were recorded for analysis.

Fig. 2 illustrates the number of optimally solved instances under each parameter setting. This figure shows that more caching states lead to more optimally solved instances under both caching strategies. Under the *latest* caching strategy, the number of instances

Table 14
The numbers of the selected instances optimally solved by B&B1 and B&B2.

B&B1									B&B2								
n	*m*								*n*	*m*							
	8	10	12	14	16	18	20	22		8	10	12	14	16	18	20	22
16	5	5	5	5	5	5	5	5	16	5	5	5	5	5	5	5	5
18	5	5	5	5	5	5	5	5	18	5	5	5	5	5	5	5	5
20	5	5	5	5	5	5	5	5	20	5	5	5	5	5	5	5	5
22	5	5	5	5	5	5	5	5	22	5	5	5	5	5	5	5	5
24	5	5	5	5	5	5	5	5	24	5	5	5	5	5	5	5	5
26	5	5	5	5	5	4	3	4	26	5	5	5	5	5	5	4	5
28	5	5	5	4	5	5	4	2	28	5	5	5	5	5	5	5	4
30	5	5	5	5	4	4	3	2	30	5	5	5	5	5	5	5	2
32	5	5	5	4	3	2	2	1	32	5	5	5	4	4	3	4	2
34	5	5	3	4	4	2	0	1	34	5	5	4	5	5	3	1	1
36	5	5	5	3	4	1	3	0	36	5	5	5	4	4	2	3	1
38	5	5	4	2	3	3	1	0	38	5	5	5	3	3	3	1	0
40	5	5	5	2	0	2	0	0	40	5	5	5	3	3	3	1	0
42	5	5	4	2	2	0	1	0	42	5	5	5	3	2	1	1	0
44	5	3	2	3	1	1	0	0	44	5	3	5	3	2	1	0	0
46	5	4	3	1	2	0	0	0	46	5	5	3	2	3	0	0	0
48	5	2	2	3	0	0	1	0	48	5	4	2	4	0	1	1	0
50	4	3	4	2	1	1	0	0	50	5	4	4	2	1	2	0	0
52	4	2	3	1	0	0	0	2	52	5	2	3	2	0	0	0	2
54	4	2	1	0	1	1	0	0	54	4	3	3	0	1	1	0	0
56	5	4	2	0	0	1	0	0	56	5	5	2	0	0	2	0	0
58	5	1	1	1	0	0	0	0	58	5	2	1	1	1	0	0	0
60	2	1	2	2	0	0	0	0	60	2	2	3	2	0	0	0	0
62	4	1	0	0	0	0	0	0	62	4	1	1	1	1	0	0	0
64	4	2	0	1	1	0	0	0	64	5	2	0	1	1	0	0	0
Sum	117	95	86	70	61	52	43	37	Sum	120	103	96	80	71	62	51	42

Table 15
The numbers of the selected instances optimally solved by B&B3 and B&B4.

B&B3									B&B4								
n	*m*								*n*	*m*							
	8	10	12	14	16	18	20	22		8	10	12	14	16	18	20	22
16	5	5	5	5	5	5	5	5	16	5	5	5	5	5	5	5	5
18	5	5	5	5	5	5	5	5	18	5	5	5	5	5	5	5	5
20	5	5	5	5	5	5	5	5	20	5	5	5	5	5	5	5	5
22	5	5	5	5	5	5	5	5	22	5	5	5	5	5	5	5	5
24	5	5	5	5	5	5	5	5	24	5	5	5	5	5	5	5	5
26	5	5	5	5	5	5	5	5	26	5	5	5	5	5	5	5	5
28	5	5	5	5	5	5	5	5	28	5	5	5	5	5	5	5	5
30	5	5	5	5	5	5	5	5	30	5	5	5	5	5	5	5	5
32	5	5	5	5	5	5	5	5	32	5	5	5	5	5	5	5	5
34	5	5	5	5	5	5	5	4	34	5	5	5	5	5	5	5	5
36	5	5	5	5	5	5	5	4	36	5	5	5	5	5	5	5	4
38	5	5	5	5	5	5	5	5	38	5	5	5	5	5	5	5	5
40	5	5	5	5	5	5	5	5	40	5	5	5	5	5	5	5	5
42	5	5	5	5	5	4	5	1	42	5	5	5	5	5	5	5	3
44	5	5	5	5	5	3	4	4	44	5	5	5	5	5	5	4	5
46	5	5	5	5	5	4	2	4	46	5	5	5	5	5	5	3	5
48	5	5	5	5	4	5	5	1	48	5	5	5	5	5	5	5	2
50	5	5	5	5	4	4	5	3	50	5	5	5	5	4	5	5	3
52	5	5	5	5	5	2	3	3	52	5	5	5	5	5	3	4	5
54	5	5	5	4	4	2	3	1	54	5	5	5	5	5	4	4	2
56	5	5	5	4	4	5	1	1	56	5	5	5	4	5	5	2	4
58	5	5	5	5	4	4	3	0	58	5	5	5	5	4	4	3	1
60	5	5	4	3	4	3	1	2	60	5	5	5	5	5	5	1	3
62	5	5	5	5	2	1	0	1	62	5	5	5	5	3	2	1	3
64	5	5	5	5	4	2	0	0	64	5	5	5	5	5	3	0	0
Sum	125	125	124	121	115	104	97	84	Sum	125	125	125	124	121	116	102	100

optimally solved increases from 854 ($C = 0$) to 922 ($C = 2^{25}$). Under the *greedy* caching strategy, this number increases from 854 to 939. When C is relatively small (e.g., $C \leq 2^{15}$), hash collisions occur frequently and the *latest* caching strategy leads to slightly better performance than the *greedy* caching strategy. The *greedy* caching strategy may store more states associated with the subproblems at the early level of the search tree, which cannot be used to effectively prune the nodes. We conjecture that since the *latest* caching strategy stores the newly encountered states and a certain state is revisited in short period with high probability, the pruning can occur with more opportunities and then the number of subproblems is reduced. When C is large (e.g., $C \geq 2^{20}$), the *greedy* caching strategy leads to more optimally solved instances than the *latest* caching strategy. This may be because a smaller state value in the caching slot is likely to eliminate more subproblems during the search process.

To further test the impacts of different parameter settings on the average number of subproblems generated, we selected five Type 2 instance groups, namely 40 × 18, 46 × 16, 52 × 14, 58 × 12 and 64 × 10. All instances in these five groups can be optimally solved using our branch-and-bound algorithm within 10 minutes of running time. We pictorially show the results associated with some parameter settings in Fig. 3. We can clearly observe that the average number of subproblems generated decreases as the number of cached states increases. This is in accordance with our intuition since more cache slots store more states, which helps prune more search nodes and therefore reduces the number of subproblems. This figure also reveals that the *greedy* caching strategy outperforms the *latest* caching strategy in terms of the average number of subproblems generated when $C = 2^{20}$ or $C = 2^{25}$, while the *latest* caching strategy generally generates fewer subproblems when C is small, i.e., $C = 2^{10}$ or $C = 2^{15}$. As a result, we adopted the *greedy* caching strategy and $C = 2^{25}$ in the final implementation of our branch-and-bound algorithm.

4.3. Impacts of algorithm components and instance generation distributions

We studied the impacts of the new lower bounds and the dominance rules by removing one type of component in turn and executing the resulting algorithm on the 1000 selected instances used in the previous section. Therefore, we have four versions of the branch-and-bound algorithm, which are:

1. B&B1 that uses only the lower bound from de la Banda et al. (2011) and does not include the dominance rules;
2. B&B2 that uses both the lower bound from de la Banda et al. (2011) and our newly proposed lower bounds, but does not include the dominance rules;
3. B&B3 that uses only the lower bound from de la Banda et al. (2011) and includes the dominance rules;
4. B&B4 that uses the lower bound from de la Banda et al. (2011), our newly proposed lower bounds and the dominance rules.

The computational results are presented in Tables 14–15, where the numbers of instances optimally solved by B&B1 – B&B4 are 561, 625, 895 and 938, respectively. These results imply that the introduction of the dominance rules and the newly proposed lower bounds increases the performance of the branch-and-bound algorithm significantly. Obviously, compared with the new lower bounds, the dominance rules contribute more for the better performance of the enhanced branch-and-bound algorithm.

Finally, we generated some new instances and evaluated the impacts of the distributions of "X". The instance generation procedure is described as follows:

1. The first 5 instances are selected from each instance group provided by de la Banda et al. (2011), for a total of 1000 instances.
2. The values of $d(s_j)$ and $c(a_i)$ and the number (n_i) of scenes that require actor a_i in each new instance are the same as those in its corresponding instance generated by de la Banda et al. (2011).
3. We generate n_i random numbers between 1 and n using Binomial ($p = 0.3$) and Poisson ($\lambda = \lceil n \rceil$) random number generators. These numbers indicate the set of scenes in which actor a_i is in. If the random number generator generates a number that has already been included in the set, we discard it and generate another one until n_i different numbers are generated. That is, we generate two new instances from each of the selected instance, for a total of 2000 new instances.

From these new instances, we observe that an actor usually participates in a set of consecutive scenes. In Tables 16–17, we present

Table 16
An instance with 8 actors and 16 scenes generated by de la Banda et al. (2011).

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	$c(a_i)$
a_1	·	X	·	X	·	·	·	X	X	·	·	X	X	X	X	X	48
a_2	X	X	X	X	X	·	·	X	X	·	X	0	X	·	·	·	23
a_3	·	X	X	X	·	·	X	·	·	X	X	X	·	X	X	·	64
a_4	X	X	·	X	X	X	·	X	X	X	·	X	·	·	·	X	41
a_5	·	·	X	·	·	X	·	X	·	X	·	·	·	·	·	X	23
a_6	X	X	X	X	·	X	·	·	X	·	X	X	X	·	X	X	73
a_7	X	X	X	X	X	·	X	X	X	X	X	X	X	X	X	X	25
a_8	·	·	·	X	·	·	·	·	·	·	·	X	·	X	X	·	25
$d(s_j)$	3	1	2	1	2	1	5	1	2	1	1	3	2	3	2	1	

Table 17
A new instance generated from the instance in Table 16.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	$c(a_i)$
a_1	·	X	X	X	X	X	X	X	X	X	·	·	·	·	·	·	48
a_2	X	X	X	X	X	X	X	X	X	·	·	·	·	·	·	·	23
a_3	·	X	X	X	X	X	X	X	X	X	·	·	·	·	·	·	64
a_4	X	·	X	X	X	X	X	X	X	X	X	·	·	·	·	·	41
a_5	·	·	X	X	X	X	·	X	·	·	·	·	·	·	·	·	23
a_6	X	X	X	X	X	X	X	X	X	X	X	·	·	·	·	·	73
a_7	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	·	25
a_8	·	·	·	X	X	X	0	X	·	·	·	·	·	·	·	·	25
$d(s_j)$	3	1	2	1	2	1	5	1	2	1	1	3	2	3	2	1	

Table 18
Average computation times of the instances generated using Binomial and Poisson random number generators.

Binomial									Poisson								
n	m								n	m							
	8	10	12	14	16	18	20	22		8	10	12	14	16	18	20	22
16	0.67	0.89	0.83	0.99	0.77	0.98	1.08	1.12	16	0.97	0.89	0.86	0.98	0.89	0.83	1.04	1.03
18	0.68	0.90	0.91	0.76	0.87	0.81	0.94	0.90	18	0.92	0.89	1.00	0.91	0.99	1.02	1.02	0.95
20	1.01	0.91	0.85	0.93	0.89	0.90	0.91	1.00	20	0.90	0.97	1.04	0.84	1.04	1.00	1.07	1.09
22	0.92	0.96	0.90	0.91	0.88	0.97	0.93	1.05	22	0.99	0.94	0.92	0.91	0.98	1.05	1.04	1.16
24	0.99	0.98	0.95	0.98	0.90	0.89	0.91	0.94	24	0.89	0.93	0.96	0.90	0.93	1.00	0.90	1.18
26	0.99	1.01	0.99	0.86	0.98	1.01	0.95	0.97	26	0.88	0.86	0.94	0.93	0.98	1.09	1.11	1.15
28	0.85	0.88	0.84	1.02	0.96	0.97	0.93	1.07	28	1.04	1.02	0.96	1.02	0.98	1.19	1.02	0.99
30	0.93	0.94	0.98	0.98	0.98	0.95	1.04	1.05	30	0.90	1.01	0.87	0.87	1.06	1.02	1.00	1.29
32	0.97	0.94	1.01	0.90	0.84	1.07	0.94	0.96	32	0.94	1.06	1.01	0.95	0.90	1.14	1.28	1.14
34	0.88	1.02	0.83	1.01	0.96	1.07	0.84	1.05	34	0.95	0.91	0.89	0.99	0.94	1.10	1.41	1.26
36	1.00	1.04	1.01	0.95	1.03	1.10	1.10	1.05	36	0.96	0.98	1.00	0.92	1.13	1.20	0.95	3.83
38	1.04	0.86	0.94	0.96	1.04	1.07	1.06	1.01	38	0.94	1.11	1.03	0.98	1.05	1.12	1.20	1.27
40	0.95	1.16	0.93	1.03	0.94	1.13	1.18	1.02	40	0.98	0.96	1.04	1.07	1.02	0.97	1.56	1.02
42	1.06	1.04	0.98	0.91	0.88	0.95	1.11	1.09	42	1.00	1.03	0.98	0.93	1.04	1.68	1.01	1.72
44	1.01	1.02	0.98	0.96	1.08	1.06	0.94	1.04	44	1.01	1.02	1.07	1.05	1.09	1.63	1.15	1.89
46	0.93	1.00	0.94	1.03	1.04	1.06	1.04	0.90	46	0.99	1.16	0.94	1.07	1.14	1.41	1.46	2.09
48	0.99	1.13	1.00	1.00	1.09	1.09	1.11	1.02	48	1.04	0.98	1.01	0.99	1.17	1.01	1.24	3.00
50	1.04	1.03	1.03	0.98	0.91	1.01	0.94	1.30	50	0.92	1.12	1.12	1.02	1.10	1.16	1.28	8.53
52	1.03	1.00	0.99	1.09	1.12	1.02	1.06	1.20	52	1.06	0.92	0.97	1.16	1.18	1.41	1.27	1.19
54	0.94	1.08	1.07	0.93	1.12	1.12	1.14	1.24	54	1.17	1.11	1.16	0.99	1.25	1.44	1.19	1.69
56	1.07	0.99	1.05	1.07	1.12	1.06	1.16	1.27	56	0.93	1.03	1.11	1.11	1.25	1.16	1.92	1.31
58	1.06	1.15	1.08	1.11	1.00	1.12	1.06	1.33	58	1.03	1.10	1.12	1.14	0.99	1.39	1.38	2.54
60	0.97	0.95	1.09	1.02	1.13	1.21	1.14	1.17	60	1.09	1.04	1.13	1.07	1.25	1.38	1.64	2.72
62	1.11	1.15	1.10	1.17	1.25	1.18	1.10	1.21	62	1.01	1.07	1.10	1.22	1.36	1.51	1.86	1.50
64	1.13	1.10	1.09	1.23	1.30	1.22	1.26	1.36	64	1.23	1.30	1.23	1.18	1.28	1.22	4.14	2.72

an example generated by de la Banda et al. (2011) and a new instance generated using the Binomial ($p = 0.3$) random number generator. Experimental results shown in Table 18 reveal that our enhanced branch-and-bound algorithm can handle these new instances very efficiently. All these 2,000 instances have been optimally solved within several seconds.

5. Conclusions

In this paper, we proposed an enhanced branch-and-bound algorithm to solve the talent scheduling problem, which is a very challenging combinatorial optimization problem. This algorithm uses a new lower bound and two new dominance rules to prune the search nodes. In addition, it caches search states for the purpose of eliminating search nodes. The experimental results clearly show that our algorithm outperforms the current best approach and achieved the optimal solutions for considerably more benchmark instances.

We present a mixed integer linear programming model for the talent scheduling problem in Section 2. A possible future research direction is to design mathematical programming algorithms for the talent scheduling problem, such as branch-and-cut algorithm and branch-and-bound coupled with Lagrangian relaxation and subgradient methods.

Acknowledgments

The research was partially supported by the National Natural Science Foundation of China (Grant No. 71571077, 71320107001) and Guangdong Natural Science Funds (No. 2014A030310312). A preliminary version of this paper appeared in the Proceedings of the 27th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA-AIE 2014).

References

Adelson, R. M., Norman, J. M., & Laporte, G. (1976). A dynamic programming formulation with diverse applications. *Operational Research Quarterly (1970-1977), Vol. 27*(1), 119–121 Part 1.
Bomsdorf, F., & Derigs, U. (2008). A model, heuristic procedure and decision support system for solving the movie shoot scheduling problem. *OR Spectrum, 30*(4), 751–772.
Braune, R., Zäpfel, G., & Affenzeller, M. (2012). An exact approach for single machine subproblems in shifting bottleneck procedures for job shops with total weighted tardiness objective. *European Journal of Operational Research, 218*(1), 76–85.
Cheng, T. C. E., Diamond, J. E., & Lin, B. M. T. (1993). Optimal scheduling in film production to minimize talent hold cost. *Journal of Optimization Theory and Applications, 79*(3), 479–492.
de la Banda, M. G., Stuckey, P. J., & Chu, G. (2011). Solving talent scheduling with dynamic programming. *INFORMS Journal on Computing, 23*(1), 120–137.
Drake, D. E., & Hougardy, S. (2003). A simple approximation algorithm for the weighted matching problem. *Information Processing Letters, 85*(4), 211–213.
Dumas, Y., Desrosiers, J., Gelinas, E., & Solomon, M. M. (1995). An optimal algorithm for the traveling salesman problem with time windows. *Operations Research, 43*(2), 367–371.
Fink, A., & Voß, S. (1999). Applications of modern heuristic search methods to pattern sequencing problems. *Computers & Operations Research, 26*(1), 17–34.
Hilden, J. (1976). Elimination of recursive calls using a small table of "randomly" selected function values. *BIT Numerical Mathematics, 16*(1), 60–73.
Karp, R. M. (1972). *Reducibility among combinatorial problems*. Springer.
Kellegöz, T., & Toklu, B. (2012). An efficient branch and bound algorithm for assembly line balancing problems with parallel multi-manned workstations. *Computers & Operations Research, 39*(12), 3344–3360.
Liang, X., Zhang, Z., Qin, H., Guo, S., & Lim, A. (2014). A branch-and-bound algorithm for the talent scheduling problem. In *Lecture notes in computer science: 8481* (pp. 208–217). Springer International Publishing.
Michie, D. (1968). "memo" functions and machine learning. *Nature, 218*(5136), 19–22.
Mingozzi, A., Bianco, L., & Ricciardelli, S. (1997). Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. *Operations Research, 45*(3), 365–377.
Nordström, A.-L., & Tufekci, S. (1994). A genetic algorithm for the talent scheduling problem. *Computers & Operations Research, 21*(8), 927–940.
Pugh, W. (1988). An improved replacement strategy for function caching. In: *Proceedings of the 1988 ACM conference on LISP and functional programming* (pp. 269–276). ACM.
Ranjbar, M., Davari, M., & Leus, R. (2012). Two branch-and-bound algorithms for the robust parallel machine scheduling problem. *Computers & Operations Research, 39*(7), 1652–1660.
Rong, A., & Figueira, J. R. (2013). A reduction dynamic programming algorithm for the bi-objective integer knapsack problem. *European Journal of Operational Research, 231*(2), 299–313.
Smith, B. M. (2003). Constraint programming in practice: Scheduling a rehearsal. *Research report apes-67-2003*. APES group.
Smith, B. M. (2005). Caching search states in permutation problems. *Lecture Notes in Computer Science, 3709*, 637–651.
Zhang, Z. Z., Qin, H., Zhu, W. B., & Lim, A. (2012). The single vehicle routing problem with toll-by-weight scheme: A branch-and-bound approach. *European Journal of Operational Research, 220*(2), 295–304.