INFORMS JOURNAL ON COMPUTING
Vol. 00, No. 0, Xxxxx 0000, pp. 000–000
ISSN 0899-1499 | EISSN 1526-5528 | 00 | 0000 | 0001

INFORMS
DOI 10.1287/xxxx.0000.0000

Solving Talent Scheduling with Dynamic Programming

Maria Garcia de la Banda
Faculty of Information Technology, Monash University, 6800, Australia,
mbanda@infotech.monash.edu.au

Peter J. Stuckey
National ICT Australia, Victoria Laboratory, Department of Computer Science & Software Engineering, University of Melbourne, 3010, Australia,
pjs@cs.mu.oz.au

Geoffrey Chu
National ICT Australia, Victoria Laboratory, Department of Computer Science & Software Engineering, University of Melbourne, 3010, Australia,
gchu@csse.unimelb.edu.au

We give a dynamic programming solution to the problem of scheduling scenes to minimize the cost of the talent. Starting from a basic dynamic program, we show a number of ways to improve the dynamic programming solution, by preprocessing and restricting the search. We show how by considering a bounded version of the problem, and determining lower and upper bounds, we can improve the search. We then show how ordering the scenes from both ends can drastically reduce the search space. The final dynamic programming solution is, orders of magnitude faster than competing approaches, and finds optimal solutions to larger problems than were considered previously.

Key words: dynamic programming; optimization; scheduling
History:

1. Introduction.

The talent scheduling problem (Cheng et al. 1993) can be described as follows. A film producer needs to schedule the scenes of his/her movie on a given location. Each scene has a duration (the days it takes to shoot it) and requires some subset of the cast to be on location. The cast are paid for each day they are required to be on location from the day the first scene they are in is shot, to the day the last scene they are in is shot, even though some of those days they might not be required by the scene currently being shot (i.e., they will be on location waiting for the next scene they are in to be shot). Each cast member has a different daily salary. The aim of the film producer is to order the scenes in such a way as to minimize the salary cost of the shooting.

We can formalize the problem as follows. Let S be a set of scenes, A a set of actors, and $a(s)$ a function that returns the set of actors involved in scene $s \in S$. Let $d(s)$ be the duration in days of scene $s \in S$, and $c(a)$ be the cost per day for actor $a \in A$. We say that actor $a \in A$ is on location at the time the scene placed in position k, $1 \leq k \leq |S|$ in the schedule is being shot, if there is a scene requiring a scheduled before or at position k, and also there is a scene requiring a scheduled at or after position k. In other words, a is on location from the time the first scene a is in is shot, until the time the last scene a is in is shot. The talent scheduling problem aims at finding a schedule for scenes S (i.e., a permutation of the scenes) that minimizes the total salary cost.

The talent scheduling problem as described in the previous paragraph is certainly an idealised version of the real problem. Real shooting schedules must contend with actor availability, setup costs for scenes, and other constraints ignored in this paper. In addition, actors can be flown back from location mid shoot to avoid paying their holding costs for extended periods. However, the talent cost, in real situations, is a prominent feature of the movie budget (Cheng et al. 1993). Hence, concentrating on this core problem is worthwhile. Furthermore, the underlying mathematical problem has many other uses, including archaeological site ordering, concert scheduling, VLSI design and graph layout. See Section 6 for more discussion on this.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	$c(a)$
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
$d(s)$	1	1	2	1	3	1	1	2	1	2	1	1	

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	$c(a)$
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**
extra	0	20	28	34	84	13	24	10	0	10	0	0	**223**

Figure 1 **(a) An Example $a(s)$ Function: $a_i \in a(s_j)$ if the Row for a_i in Column s_j has an X. (b) An Example Schedule: a_i is on location when scene s_j is Scheduled if the Row for a_i in Column s_j has an X or a –.**

EXAMPLE 1. Consider the talent scheduling problem defined by the set of actors $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$, the set of scenes $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}\}$, and the $a(s)$ function determined by the matrix M shown in Figure 1(a), where an X at position M_{ij} indicates that actor a_i takes part in scene s_j. The daily cost per actor $c(a)$ is shown in the rightmost column, and the duration of each scene $d(s)$ is shown in the last row.

Consider the schedule obtained by shooting the scenes in order $s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11}, s_{12}$.The consequences of this schedule in terms of actor's presence and cost are illustrated by the matrix M shown in Figure 1(b), where actor a_i is on location at the j^{th} shot scene if the position M_{ij} contains either an X (a_i is in the scene) or an – (a_i is waiting). The cost of each scene is shown in the second last row, being the sum of the daily costs of all actors on location multiplied by the duration of the scene. The total cost for this schedule is 604. The *extra cost* for each scene is shown in the last row, being the sum of the daily costs of only those actors waiting on location, multiplied by the duration of the scene. The extra cost for this schedule is 223.

□

The scene scheduling problem was introduced by Cheng et al. (1993). In its original form each of the scenes is actually a shooting day and, hence, the duration of each of the scenes is 1. A variation of the problem, called concert scheduling (Adelson et al. 1976), considers the case where the cost for each player is identical. The scene scheduling problem is known to be NP-hard (Cheng et al. 1993) even if each actor appears in only two scenes, all actor costs are identical and all durations are identical.

The main contributions of this paper are:

- We define an effective dynamic programming solution to the problem
- We define and prove correct a number of optimizations for the dynamic programming solution, that increase the size of problems we can feasibly tackle
- We show how using bounded dynamic programming can substantially improve the solving of these problems
- We show how, by considering a more accurate notion of subproblem equivalence, we can substantially improve the solving

The final code can find optimal solutions to problems larger than previous methods.

In Section 2 we give our call-based best-first dynamic programming formulation for the talent scheduling problem, and consider ways it can be improved by preprocessing and modifying the search. Section 3 examines how to solve a bounded version of the problem, which can substantially improve performance, and how to compute upper and lower bounds for the problem. Section 4 investigates a better search strategy where we schedule scenes from both ends of the search and Section 5 presents an experimental evaluation of the different approaches. In Section 6 we discuss related work and in Section 7 we conclude.

2. Dynamic Programming Formulation

The talent scheduling problem is naturally expressible in a dynamic programming formulation. To do so we extend the function $a(s)$ which returns the set of actors in scene $s \in S$, to handle a set of scenes $Q \subseteq S$. That is, we define $a(Q) = \cup_{s \in Q} a(s)$ as a function that returns the set of actors appearing in any scene $Q \subseteq S$. Similarly, we extend the cost function $c(a)$ to sets of actors $G \subseteq A$ in the obvious way: $c(G) = \sum_{a \in G} c(a)$.

Let $l(s,Q)$ denote the set of actors on location at the time scene s is scheduled assuming that the set of scenes $Q \subset (S - \{s\})$ is scheduled after s, and the set $S - Q - \{s\}$ is scheduled before s. Then

$$l(s,Q) = a(s) \cup (a(Q) \cap a(S - Q - \{s\})),$$

i.e., the on locations actors are those who appear in scene s, plus those who appear in both a scene scheduled after s and one scheduled before s. The problem is amenable to dynamic programming because $l(s,Q)$ does not depend on any particular order of the scenes in Q or $S - Q - \{s\}$. Let $Q \subseteq S$ denote the set of scenes still to be scheduled, and let $schedule(Q)$ be the minimum cost required to schedule the scenes in Q. Dynamic programming can be used to define $schedule(Q)$ as:

$$schedule(Q) = \begin{cases} 0 & Q = \emptyset \\ \min_{s \in Q}((d(s) \times c(l(s, Q - \{s\}))) + schedule(Q - \{s\})), & \text{otherwise} \end{cases}$$

which computes, for each scene s, the cost of scheduling the scene s first $d(s) \times c(l(s, Q - \{s\}))$ plus the cost of scheduling the remaining scenes $Q - \{s\}$. Dynamic programming is effective for this problem because it reduces the raw search space from $|S|!$ to $2^{|S|}$, since we only need to investigate costs for each subset of S (rather than for each permutation of S).

2.1 Basic Best-first Algorithm

The code in Figure 2 illustrates our best-first call-based dynamic programming algorithm, which improves over a naïve formulation by pruning children that cannot yield a smaller cost.

The algorithm starts by checking whether Q is empty, in which case the cost is 0. Otherwise, it checks whether the minimum cost for Q has already been computed (and stored in $scost[Q]$), in which case it returns the previously stored result (code shown in light gray). We assume the $scost$ array is initialized with zero. If not, the algorithm selects the next scene s to be scheduled (using a simple heuristic that will be discussed later) and computes in sp the value $\mathsf{cost}(s, Q - \{s\}) + \mathsf{schedule}(Q - \{s\})$, where function $\mathsf{cost}(s, B)$ returns the cost of scheduling scene s before any scene in $B \subset S$ (and after any scene in $S - B - \{s\}$), calculated as:

$$\mathsf{cost}(s, B) = d(s) \times c(l(s, B))$$

Note, however, that the algorithm avoids (thanks to the **break**) considering scenes whose lower bound is greater than or equal to the current minimum min, since they cannot improve on the current solution. As a result, the order in which the Q scenes are selected can significantly affect the amount of work performed. In our algorithm, this order is determined by a simple heuristic that selects the scene s with the smallest calculated lower bound if scheduled immediately $\mathsf{cost}(s, Q - \{s\}) + \mathsf{lower}(Q - \{s\})$, where function

$\mathsf{lower}(B)$ returns a lower bound on the cost of scheduling the scenes in $B \subseteq S$, and it is simply calculated as:

$$\mathsf{lower}(B) = \sum_{s \in B} d(s) \times c(a(s))$$

which is the sum of the costs for actors that appear in each scene. The index construct index $\min_{s \in Q} e(s)$ returns the s in Q that causes the expression $e(s)$ to take its minimum value.

A call to function $\mathsf{schedule}(S)$ returns the minimum cost required to schedule the scenes in S. Extracting the optimal schedule found from the array of stored answers $scost[]$ is straightforward, and standard for dynamic programming.

EXAMPLE 2. Consider the problem of Example 1. An optimal solution is shown in Figure 3. The total cost is 434, and the extra cost 53. □

2.2 Preprocessing

We can simplify the problem in the following two ways:

- Eliminating single scene actors: Any actor a' that appears only in one scene s can be removed from s (i.e., we can redefine set $a(s)$ as $a(s) - \{a'\}$) and add its fixed cost $d(s) \times c(a')$ to the overall cost. This is correct because the cost of a' is the same independently of where s is scheduled (since a' will never have to wait while on location).
- Concatenating duplicate scenes: Any two scenes s_1 and s_2 such that $a(s_1) = a(s_2)$ can be replaced by a single scene s with duration $d(s) = d(s_1) + d(s_2)$. This is correct because there is always an optimal schedule in which s_1 and s_2 are scheduled together.

$\mathsf{schedule}(Q)$
  if $(Q = \emptyset)$ **return** 0
  if $(scost[Q])$ **return** $scost[Q]$
  $min := +\infty$
  $T := Q$
  while $(T \neq \emptyset)$
    $s :=$ index $\min_{s \in T} \mathsf{cost}(s, Q - \{s\}) + \mathsf{lower}(Q - \{s\})$
    $T := T - \{s\}$
    if $(\mathsf{cost}(s, Q - \{s\}) + \mathsf{lower}(Q - \{s\}) \geq min)$ **break**
    $sp := \mathsf{cost}(s, Q - \{s\}) + \mathsf{schedule}(Q - \{s\})$
    if $(sp < min)$ $min := sp$
  $scost[Q] := min$
  return min

Figure 2 **Pseudo-Code for Best-first Call-based Dynamic Programming Algorithm. schedule(Q) Returns the Minimum Cost Required for Scheduling the Set of Scenes Q**

	s_5	s_2	s_7	s_1	s_6	s_8	s_4	s_9	s_3	s_{11}	s_{10}	s_{12}	$c(a)$
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	45	19	19	39	39	66	29	29	50	25	54	20	**434**
extra	0	0	10	4	9	10	20	0	0	0	0	0	**53**

Figure 3 **An Optimal Order for the Problem of Example 1**

Since each simplification can generate new candidates for the other kind of simplification, we need to repeatedly apply them until no new simplification is possible.

The simplification that concatenates duplicate scenes has been applied before, but not formally proved to be correct. For example, the real scene scheduling data from Cheng et al. (1993) was used in Smith (2005) with this simplification applied.

LEMMA 1. *If there exists s_1 and s_2 in S where $a(s_1) = a(s_2)$, then there is an optimal order with s_1 and s_2 scheduled together.*

Proof. Let Π denote a possibly empty sequence of scenes. In an abuse of notation, and when clear from context, we will sometimes use sequences as if they were sets. Without loss of generality, take the order $\Pi_1 s_1 \Pi_2 s' \Pi_3 s_2 \Pi_4$ of the scenes in S, and consider the actors on location for scene s_1 to be $l(s_1, \Pi_2 s' \Pi_3 s_2 \Pi_4) = A_1$ and for scene s_2 to be $l(s_2, \Pi_4) = A_2$. Now, either $c(A_1) \leq c(A_2)$ (which, since $a(s_1) = a(s_2)$ means that the cost of the actors *waiting* in A_1 is smaller or equal than that of the actors *waiting* in A_2) or $c(A_1) > c(A_2)$. We will show how, in the first case, choosing the new order (a) $\Pi_1 s_1 s_2 \Pi_2 s' \Pi_3 \Pi_4$ can only decrease the cost for each scene. It is symmetric to show that, in the second case, choosing the new order (b) $\Pi_1 \Pi_2 s' \Pi_3 s_1 s_2 \Pi_4$ can only decrease the cost for each scene.

Let's examine the costs of s_1 and s_2. The cost of s_1 does not change from the original order to that of (a) since the set of scenes before and after s_1 remains unchanged (i.e., since by definition $l(s_1, s_2 \Pi_2 s' \Pi_3 \Pi_4) = l(s_1, \Pi_2 s' \Pi_3 s_2 \Pi_4)$). The cost of s_2 in (a) is the cost of the actors in

$$\begin{aligned}
l(s_2, \Pi_2 s' \Pi_3 \Pi_4) &= a(s_2) \cup (a(\Pi_2 s' \Pi_3 \Pi_4) \cap a(\Pi_1 s_1)) \\
&\quad \text{By definition of } l(s, Q) \\
&= a(s_1) \cup (a(\Pi_2 s' \Pi_3 \Pi_4) \cap a(\Pi_1 s_1)) \\
&\quad \text{By hypothesis of } a(s_1) = a(s_2) \\
&= a(s_1) \cup (a(\Pi_2 s' \Pi_3 \Pi_4) \cap a(\Pi_1)) \\
&\quad \text{By definition of } a(Q) \\
&= a(s_1) \cup (a(\Pi_2 s' \Pi_3 s_2 \Pi_4) \cap a(\Pi_1)) \\
&\quad \text{By definition of } a(Q) \text{and by hypothesis of } a(s_1) = a(s_2) \\
&= l(s_1, \Pi_2 s' \Pi_3 s_2 \Pi_4) \\
&\quad \text{By definition of } l(s, Q)
\end{aligned}$$

which is known to be A_1. Hence, the cost of s_2 can only decrease, since $c(A_1) \leq c(A_2)$.

Let's consider the other scenes. First, it is clear that the products in Π_1 and Π_4 have the same on location actors since the set of scenes before and after remain unchanged. Second, let us consider the changes in the on locations actors for s', which can be seen as a general representative of scenes scheduled in between s_1 and s_2 in the original order. While in the original order the set of on location actors at the time s' is scheduled is $l(s', \Pi_3 s_2 \Pi_4) = a(s') \cup (a(\Pi_1 s_1 \Pi_2) \cap a(\Pi_3 s_2 \Pi_4))$, in the new order the set of on location actors is $l(s', \Pi_3 \Pi_4) = a(s') \cup (a(\Pi_1 s_1 s_2 \Pi_2) \cap a(\Pi_3 \Pi_4))$. Clearly (a) $a(\Pi_3 \Pi_4) \subseteq a(\Pi_3 s_2 \Pi_4)$ and (b) since $a(s_1) = a(s_2)$, we have that $a(\Pi_1 s_1 s_2 \Pi_2) = a(\Pi_1 s_1 \Pi_2)$. Hence, by (a) and (b) we have that $l(s', \Pi_3 \Pi_4) \subseteq l(s', \Pi_3 s_2 \Pi_4)$, which means the set of on location actors when s' is scheduled can only decrease and, hence, the cost of scheduling it does not increase. □

EXAMPLE 3. Consider the scene scheduling problem from Example 1. Since actor a_6 only appears in one task, we can remove this actor and add a total of $2 \times 7 = 14$ to the cost of the resulting problem to get the cost of the original problem. We also have that $a(s_3) = a(s_{11})$ and, after the simplification above, $a(s_{10}) = a(s_{12})$. Hence, we can replace these pairs by single new scenes of the combined duration. The resulting preprocessed problem is show in Figure 4.□

	s_1	s_2	s'_3	s_4	s_5	s_6	s_7	s_8	s_9	s'_{10}	$c(a)$
a_1	X	.	X	.	.	X	.	X	X	X	20
a_2	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	.	.	.	.	.	.	.	.	.	.	7
$d(s)$	1	1	3	1	3	1	1	2	1	3	

Figure 4 The Problem of Example 1 after Preprocessing.

2.3 Scheduling Actor Equivalent Scenes First

Let $o(Q) = a(S - Q) \cap a(Q)$ be the set of on location actors just before an element of Q is scheduled, i.e., those for whom some of their scenes have already been scheduled (appear in $S - Q$), and some have not (appear in Q). We can reduce the amount of search performed by the code shown in Figure 2 (and thus improve its efficiency) by noticing that any scene whose actors are exactly the same as those on location now can always be scheduled first without affecting the optimality of the solution. In other words, for every $s \in Q$ for which $a(s) = o(Q)$, there must be an optimal solution to $schedule(Q)$ that starts with s.

EXAMPLE 4. Consider the scene scheduling problem of Example 1. Let us assume that the set of scenes $Q = \{s_1, s_2, s_4, s_7, s_8, s_9\}$ is scheduled after those in $S - Q = \{s_3, s_5, s_6, s_{10}, s_{11}, s_{12}\}$ have been scheduled. Then, the set of on location actors after $S - Q$ is scheduled is $o(Q) = \{a_1, a_2, a_4\}$ and an optimal schedule can begin with s_1 since $a(s_1) = o(Q)$. An optimal schedule of this form is shown in Figure 5.□

LEMMA 2. *If there exists $s \in Q$ where $a(s) = o(Q)$, then there is an optimal order for $schedule(Q)$ beginning with s.*

Proof. Let Π denote a possibly empty sequence of scenes. As before, we will sometimes use sequences as if they were sets. Without loss of generality, take the order $\Pi_1 \Pi_2 s' \Pi_3 s \Pi_4$ of the scenes in S where $\Pi_2 s' \Pi_3 s \Pi_4$ is the sequence of scenes in Q, and consider altering the order to $\Pi_1 s \Pi_2 s' \Pi_3 \Pi_4$. We show that the cost for each scene in Q can only decrease.

First, it is clear that the scenes in Π_4 have the same on location actors since the set of scenes before and after it remains unchanged. Second, let us consider the changes in the on locations actors for s', which can be seen as a general representative of scenes scheduled before s in the original order. While in the original order the set of on location actors at the time s' is scheduled is $l(s', \Pi_3 s \Pi_4) = a(s') \cup (a(\Pi_1 \Pi_2) \cap a(\Pi_3 s \Pi_4))$, in the new order the set of on location actors is $l(s', \Pi_3 \Pi_4) = a(s') \cup (a(\Pi_1 s \Pi_2) \cap a(\Pi_3 \Pi_4))$. Now, for every set of scenes $Q'' \subseteq Q'$ we know that $a(Q'') \subseteq a(Q')$, i.e., increasing the number of scenes can only increase the number of actors involved. Thus, we have that (a) $a(\Pi_3 \Pi_4) \subseteq a(\Pi_3 s \Pi_4)$, and (b) since $a(s) = o(Q) = (a(Q) \cap a(\Pi_1))$ we have that $a(s) \subseteq a(\Pi_1)$, and thus that $a(\Pi_1 s \Pi_2) = a(\Pi_1 \Pi_2)$. Hence, by (a) and (b) we have that $l(s', \Pi_3 \Pi_4) \subseteq l(s', \Pi_3 s \Pi_4)$, which means the set of on location actors for s' in the altered schedule can only decrease and, thus, its cost cannot increase. Finally, we also have to examine the cost for s. Since

	s_{12}	s_{10}	s_{11}	s_3	s_5	s_6	$o(Q)$	s_1	s_2	s_9	s_8	s_7	s_4	$c(a)$
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
$d(s)$	1	2	1	2	3	1		1	1	1	2	1	1	

Figure 5 An Optimal Schedule for the scenes $Q = \{s_1, s_2, s_4, s_7, s_8, s_9\}$ Assuming $\{s_3, s_5, s_6, s_{10}, s_{11}, s_{12}\}$ Have Already been Scheduled

$a(s) = o(Q)$ we have that $l(s, Q - \{s\}) = a(s)$. That means there is no actor waiting if we schedule s now, which is the cheapest possible way to schedule s. Hence, the costs of scheduling this scene here is no more expensive than in the original position. □
We can modify the pseudo code of Figure 2 to take advantage of Lemma 2 by adding the line

if $(\exists s \in Q.a(s) = o(Q))$ **return** $d(s) \times c(l(s, Q - \{s\})) + \mathsf{schedule}(Q - \{s\})$

before the line min := $+\infty$.

2.4 Pairwise Subsumption

When we have two scenes s_1 and s_2 where the actors in one scene (s_1) are a subset of the actors in the other (s_2) and the extra actors $a(s_2) - a(s_1)$ are already on location then we can guarantee a better schedule if we always schedule s_2 before s_1. Intuitively, this is because if s_1 is shot first the missing actors would be waiting on location for scene s_2 to be shot, while if s_2 is shot first some of those missing actors might not be needed on location anymore.

LEMMA 3. *If there exists $\{s_1, s_2\} \subseteq Q$, such that $a(s_1) \subseteq a(s_2)$, $a(S - Q) \cup a(s_1) \supseteq a(s_2)$, then for any order of Q where s_1 appears before s_2, there is a permutation of that order where s_2 appears before s_1 with equal or lower cost.*

Proof. Let Π denote a possibly empty sequence of scenes. As before, we will sometimes use sequences as if they were sets. Without loss of generality, take the order $\Pi_1\Pi_2 s_1 \Pi_3 s' \Pi_4 s_2 \Pi_5$ of scenes in S where $\Pi_2 s_1 \Pi_3 s' \Pi_4 s_2 \Pi_5$ is the sequence of scenes in Q, and consider the actors on location for scene s_1 to be $l(s_1, \Pi_3 s' \Pi_4 s_2 \Pi_5) = A_1$ and for s_2 to be $l(s_2, \Pi_5) = A_2$. Now either $c(A_1) \leq c(A_2)$ or $c(A_1) > c(A_2)$.

Case $c(A_1) \leq c(A_2)$: We show that choosing $\Pi_1\Pi_2 s_2 s_1 \Pi_3 s' \Pi_4 \Pi_5$ as new order can only decrease the cost for each scene. The cost of s_1 in the original schedule is the cost of the actors in $l(s_1, \Pi_3 s' \Pi_4 s_2 \Pi_5)$ which is computed as $a(s_1) \cup (a(\Pi_3 s' \Pi_4 s_2 \Pi_5) \cap a(\Pi_1\Pi_2))$, while for the second schedule is the cost of the actors in $l(s_1, \Pi_3 s' \Pi_4 \Pi_5)$ which is computed as $a(s_1) \cup (a(\Pi_3 s' \Pi_4 \Pi_5) \cap a(\Pi_1 \Pi_2 s_2))$. Since by hypothesis $a(\Pi_1) \cup a(s_1) \supseteq a(s_2)$ and by definition $a(\Pi_3 s' \Pi_4 \Pi_5) \subseteq a(\Pi_3 s' \Pi_4 s_2 \Pi_5)$, we have that $l(s_1, \Pi_3 s' \Pi_4 \Pi_5) \subseteq l(s_1, \Pi_3 s' \Pi_4 s_2 \Pi_5)$ and, hence, the cost of s_1 can only decrease.

Regarding s_2, the set of actors in the new order is

$$\begin{aligned}
l(s_2, s_1\Pi_3 s'\Pi_4\Pi_5) &= a(s_2) \cup (a(s_1\Pi_3 s'\Pi_4\Pi_5) \cap a(\Pi_1\Pi_2)) \\
&\quad \text{By definition of } l(s, Q) \\
&= (a(s_2) \cup a(s_1\Pi_3 s'\Pi_4\Pi_5)) \cap (a(s_2) \cup a(\Pi_1\Pi_2)) \\
&\quad \text{Distributing } \cup \text{ over } \cap \\
&= (a(s_1) \cup a(s_2\Pi_3 s'\Pi_4\Pi_5)) \cap (a(s_2) \cup a(\Pi_1\Pi_2)) \\
&\quad \text{By definition of } a(Q) \\
&\subseteq (a(s_1) \cup a(s_2\Pi_3 s'\Pi_4\Pi_5)) \cap (a(s_1) \cup a(\Pi_1\Pi_2)) \\
&\quad \text{By hypothesis of } a(\Pi_1) \cup a(s_1) \supseteq a(s_2) \\
&= l(s_1, \Pi_3 s'\Pi_4 s_2\Pi_5) \\
&\quad \text{By definition of } l(s, Q)
\end{aligned}$$

which is known to be A_1. Hence, the cost of s_2 can only decrease in the new schedule.

Let's now consider the other scenes. First, it is clear that the products in Π_1 Π_2, and Π_5 have the same on location actors since the set of scenes before and after remain unchanged. Second, let us consider the changes in the on locations actors for s', which can be seen as a general representative of scenes scheduled in between s_1 and s_2 in the original order. While in the original order the set of on location actors at the time s' is scheduled is $l(s', \Pi_4 s_2 \Pi_5) = a(s') \cup (a(\Pi_1\Pi_2 s_1 \Pi_3) \cap a(\Pi_4 s_2 \Pi_5))$, in the new order the set of on location actors is $l(s', \Pi_4\Pi_5) = a(s') \cup (a(\Pi_1\Pi_2 s_2 s_1 \Pi_3) \cap a(\Pi_4\Pi_5))$, Clearly (a) $a(\Pi_4\Pi_5) \subseteq a(\Pi_4 s_2 \Pi_5)$

and (b) since by hypothesis $a(\Pi_1) \cup a(s_1) \supseteq a(s_2)$, we have that $a(\Pi_1\Pi_2 s_2 s_1 \Pi_3) \subseteq a(\Pi_1\Pi_2 s_1 \Pi_3)$. Hence, by (a) and (b) we have that $l(s', \Pi_4\Pi_5) \subseteq l(s', \Pi_4 s_2 \Pi_5)$ and, hence, the cost of scheduling it cannot increase.

Case $c(A_1) > c(A_2)$:

We show that choosing $\Pi_1\Pi_2\Pi_3 s'\Pi_4 s_2 s_1 \Pi_5$ as new order can only decrease the cost for each scene. Regarding s_1, the set of actors in the new order is

$$\begin{aligned}
l(s_1, \Pi_5) &= a(s_1) \cup (a(\Pi_5) \cap a(\Pi_1\Pi_2\Pi_3 s'\Pi_4 s_2)) \\
&\quad \text{By definition of } l(s,Q) \\
&= (a(s_1) \cup a(\Pi_5)) \cap (a(s_1) \cup a(\Pi_1\Pi_2\Pi_3 s'\Pi_4 s_2)) \\
&\quad \text{Distributing} \cup \text{over} \cap \\
&= (a(s_1) \cup a(\Pi_5)) \cap (a(s_2) \cup a(\Pi_1\Pi_2 s_1 \Pi_3 s'\Pi_4)) \\
&\quad \text{By definition of } a(Q) \\
&\subseteq (a(s_2) \cup a(\Pi_5)) \cap (a(s_2) \cup a(\Pi_1\Pi_2 s_1 \Pi_3 s'\Pi_4)) \\
&\quad \text{By hypothesis of } a(s_1) \subseteq a(s_2) \\
&= l(s_2, \Pi_5) \\
&\quad \text{By definition of } l(s,Q)
\end{aligned}$$

which is known to be A_2. Hence, the cost of s_1 can only decrease in the new schedule.

Now since $a(s_1) \subseteq a(s_2)$ we have that $a(s_2 s_1 \Pi_5) = a(s_2\Pi_5)$ and, since adding scenes can only increase cost, we have that $a(\Pi_1\Pi_2\Pi_3 s'\Pi_4) \subseteq a(\Pi_1 s_1 \Pi_2\Pi_3 s'\Pi_4)$. Thus, $l(s_2, s_1\Pi_5) \subseteq l(s_2, \Pi_5)$, which means the cost of s_2 can only decrease.

Let's consider the other scenes. As before, it is clear that the products in Π_1 Π_2, and Π_5 have the same on location actors since the set of scenes before and after remain unchanged. Let us then consider the changes in the on locations actors for s', which can be seen as a general representative of scenes scheduled in between s_1 and s_2 in the original order. While in the original order the set of on location actors at the time s' is scheduled is $l(s', \Pi_4 s_2 \Pi_5) = a(s') \cup (a(\Pi_1\Pi_2 s_1 \Pi_3) \cap a(\Pi_4 s_2 \Pi_5))$, in the new order the set of on location actors is $l(s', \Pi_4 s_2 s_1 \Pi_5) = a(s') \cup (a(\Pi_1\Pi_2\Pi_3) \cap a(\Pi_4 s_2 s_1 \Pi_5))$, Clearly (a) by definition $a(\Pi_1\Pi_2 s_1 \Pi_3) \supseteq a(\Pi_1\Pi_2\Pi_3)$ and (b) by hypothesis of $a(s_1) \subseteq a(s_2)$ we have that $a(\Pi_4 s_2 s_1 \Pi_5) = a(\Pi_4 s_2 \Pi_5)$. Hence, by (a) and (b) we have that $l(s', \Pi_4 s_2 s_1 \Pi_5) \subseteq l(s', \Pi_4 s_2 \Pi_5)$, which means the set of on location actors when s' is scheduled can only decrease and, hence, the cost of scheduling cannot increase. □

EXAMPLE 5. Consider the scene scheduling problem of Example 1. Let us assume that the set of scenes $Q = S - \{s_5\}$ is scheduled after s_5. Then, the on location actors after $\{s_5\}$ are $o(Q) = \{a_2, a_4\}$. Consider s_1 and s_6. Since $a(s_6) \subseteq a(s_1)$ and $o(Q) \cup a(s_6) \supseteq a(s_1)$, s_1 should be scheduled before s_6. This means we should never consider scheduling s_6 next! □

We can modify the pseudo code of Figure 2 to take advantage of Lemma 3 by adding the line

forall $(s_1 \in T)$
  if $(\exists s_2 \in T. a(s_1) \subseteq a(s_2) \wedge a(S-Q) \cup a(s_1) \supseteq a(s_2))$ $T := T - \{s_1\}$

after the line $T := Q$ and before the **while** loop. However, this is too expensive in practice. To make this efficient enough we need to precalculate the pairs P of the form (s, s') where $a(s) \subseteq a(s')$ and just check that $s' \in T$, $s \in T$ and $a(S-Q) \cup a(s) \supseteq a(s')$ for each pair in P.

Pairwise subsumption, was first used in the solution of Smith (2005, 2003), although restricted to cases where the difference in the sets is one or two elements. Although no formal proof is given, there is an extensive example in Smith (2003) explaining the reasoning for the case where the scenes differ by one element.

2.5 Optimizing Extra Cost

The *base cost* of a scene scheduling problem is given by $\sum_{s \in S} d(s) \times c(a(s))$. This is the cost for just paying for the time of the actors of the scenes they actually appear in. Instead of minimizing the total cost, we can minimize the *extra cost* which is the total cost minus the base cost (i.e., the cost of paying for actors that are waiting, rather than playing). We can recover the minimal cost by simply adding the base cost to the minimal extra cost.

To do so we simply need to change the cost and lower functions used in Figure 2 as follows:

$$\mathsf{cost}(s,Q) = d(s) \times c(l(s,Q) - a(s))$$

$$\mathsf{lower}(Q) = 0.$$

The main benefit of this optimization is simply that the cost of computing the lower bounds becomes free.

3. Bounded Dynamic Programming

We can modify our problem to be a bounded problem. Let $bnd\_schedule(Q,U)$ be the minimal cost required to schedule scenes Q if this is less than or equal to the bound U, and otherwise some number k where $U < k \leq schedule(Q)$. We can change the dynamic program to take into account upper bounds U on a solution of interest. The recurrence equation becomes

$$bnd\_schedule(Q,U) = \begin{cases} 0 & Q = \emptyset \vee U < 0 \\ \min_{s \in Q} d(s) \times c(a(s, Q - \{s\})) + \\ \quad bnd\_schedule(Q - \{s\}, U - d(s) \times c(a(s, Q - \{s\}))), & \text{otherwise} \end{cases}$$

The only complexity here is that the upper bound is reduced in the recursive relation to take into account the cost of scene s.

Using bounding can have two effects, one positive and one negative. On the positive side, we may be able to determine without much search that a subproblem cannot provide a better solution for the original problem, thus restricting the search. On the negative side, it may increase the search space since we have now multiplied the potential number of subproblems by the upper bound U.

3.1 Bounded Best-first Algorithm

Some of the potential subproblem explosion of adding bounds can be ameliorated since if $schedule(Q) \leq U$ then $bnd\_schedule(Q,U) = schedule(Q)$ and, otherwise, $bnd\_schedule(Q,U) \leq schedule(Q)$ (i.e., $bnd\_schedule(Q,U)$ is a lower bound for $schedule(Q)$). Therefore, we only need to store one answer in the hash table for problem Q (rather than one per U): either the value $OPT(v)$ indicating we have computed the optimal answer v, or the value $LB(v)$ indicating we have determined a lower bound v on the answer. We assume the hash table is initialized with entries $NONE$ indicating no result has been stored. The only time we have to reevaluate a subproblem Q is if the stored lower bound v is less than or equal than the current U.

The code for the bounded dynamic program is shown in Figure 6. Note that the hash table handling is slightly more complex, since we can immediately return a lower bound $v > U$ if that is stored in the hash table already. The key advantage w.r.t. efficiency is that the break in the **while** loop uses the value U rather than min, since clearly no schedule beginning with s will be able to give a schedule costing less than U in this case. This requires us to update the bound U if we find a new minimum. When the search completes we have either discovered the optimal (if it is less than U), in which case we store it as optimal, or we have discovered a lower bound ($> U$) which we store in the hash table.

This means we can prune more subproblems. Note that this kind of addition of bounds can be automated (Puchinger and Stuckey 2008).

```
bnd_schedule(Q, U)
  if (Q = ∅) return 0
  if (scost[Q] = OPT(v)) return v
  if (scost[Q] = LB(v) ∧ v > U) return v
  min := +∞
  T := Q
  while (T ≠ ∅)
    s := index min_{s∈T} cost(s, Q − {s}) + lower(Q − {s})
    T := T − {s}
    if (cost(s, Q − {s}) + lower(Q − {s}) ≥ U) break
    sp := cost(s, Q − {s}) + bnd_schedule(Q − {s}, U − cost(s, Q − {s}))
    if (sp < min) min := sp
    if (min ≤ U) U := min
  if (min ≤ U) scost[Q] := OPT(min)
  else scost[Q] := LB(min)
  return min
```

Figure 6 Pseudo-Code for bounded Best-first Call-based Dynamic Programming Algorithm. bnd_schedule(Q,U) Returns the Minimum Cost Required for Scheduling the Set of Scenes Q if it is Less than or Equal to U, Otherwise it Returns a Lower Bound on the Minimal Cost

3.2 Upper Bounds

Now that we are running a bounded dynamic program, we need an initial upper bound for the original problem. A trivial upper bound is the maximum possible cost, i.e., if all actors are on location at all times:

$$(\sum_{s\in S} d(s)) \times (\sum_{a\in A} c(a))$$

To generate a better upper bound, we use a heuristic based on the idea that keeping expensive actors waiting around is bad. Thus, it prioritises expensive actors by attempting to keep their scenes together as much as possible (i.e., as long as this does not imply separating scenes of more expensive actors). To do this, the algorithm maintains a sequence of disjoint sets of scenes (each set corresponding to the scenes kept together for some actors) which provides a partial schedule, i.e, the scenes in a set are known to be scheduled after the scenes in any set to the left and before the scenes in any set to the right. The idea is to (a) only partition sets into smaller sets when this benefits the next actor to be processed, and (b) never to insert new scenes into the middle of the schedule (i.e., scenes are never added to an already formed set, and sets are only added at the beginning or the end of the partial schedule).

Initially the schedule is empty. And the remaining actors are $R = A$. Then, we select the remaining actor $a \in R$ with greatest fixed cost $c(a) \times (\sum_{s\in S, a\in a(s)} d(s))$, and we determine the first and last sets in the schedule involving a. If the actor is currently not involved in any set, then we simply add a new set at the end of the schedule with all the scenes in which a appears. If all the scenes involving a in are in a single set, we break the set into those involving a and those not, arbitrary placing the second set afterwards. If all the scenes involving a are already scheduled, we split the first set that involves the actor into two: first those not involving the actor, and then those involving the actor. We do the same for the last set involving the actor, except that the set involving the actor goes first. This ensures that the scenes involving a are placed as close as possible without disturbing the scheduling of the previous actors.

If not all the scenes involving a are already scheduled, we first need to decide whether to put the set of remaining scenes at the beginning or the end of the current schedule. To do this we calculate the total duration for which actor a will be on location if the scenes were scheduled before or after, and place their

	s_{10}	s_{12}	s_8	s_3	s_9	s_{11}	s_1	s_6	s_2	s_5	s_7	s_4	$c(a)$
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
$d(s)$	2	1	2	2	1	1	1	1	1	3	1	1	

Figure 7 The Schedule Defined by the Heuristic Upper Bound Algorithm.

remaining scenes wherever it leads to the smallest duration. Then, if we place the scenes afterwards, we split the group where the actor a first appears into two: first those not involving the actor, and then those involving the actor. Similarly, if the remaining scenes are scheduled at the beginning we split the last group where a appears into two: first those involving the actor, and those not involving the actor.

This process continues until all actors are considered. We may have some groups which are still not singletons after this process. We order them in any way, since it cannot make a difference to the cost.

Note that this algorithm ensures that the two most expensive actors will never be waiting.

EXAMPLE 6. Consider the scene scheduling problem from Example 1. The fixed cost of the actors $a_1, a_2, a_3, a_4, a_5, a_6$ are respectively $220, 55, 16, 60, 16, 14$. Thus, we first schedule all scenes involving a_1 in one group $\{s_1, s_3, s_6, s_8, s_9, s_{10}, s_{11}, s_{12}\}$. We next consider a_4, which has some scenes scheduled (s_1 and s_6) and some not (s_2 and s_5). Thus, we first need to decide whether to place the set $\{s_2, s_5\}$ after or before the current schedule. Since the duration for which a_4 will be waiting on location is 0 in both cases, we follow the default (place it after) and split the already scheduled group into those not involving a_4 and those involving a_4, resulting in partial schedule $\{s_3, s_8, s_9, s_{10}, s_{11}, s_{12}\}$ $\{s_1, s_6\}$ $\{s_2, s_5\}$. The total durations of the groups are 9, 2, and 4 respectively

We next consider a_2, whose scenes $\{s_4, s_7\}$ are not scheduled. The total duration for a_2 placing these at the beginning is 2 + 9 + 2 + 4 = 17, while placing them at the end is 4 + 2 + 4 + 2 = 10. Thus, again we place them at the end, and split the first group, obtaining the partial schedule $\{s_8, s_{10}, s_{12}\}$ $\{s_3, s_9, s_{11}\}$ $\{s_1, s_6\}$ $\{s_2, s_5\}$ $\{s_4, s_7\}$.

We next consider a_3, whose scenes are all scheduled and some appear in the first and the last group. We thus split these two groups to obtain $\{s_{10}, s_{12}\}$ $\{s_8\}$ $\{s_3, s_9, s_{11}\}$ $\{s_1, s_6\}$ $\{s_2, s_5\}$ $\{s_7\}$ $\{s_4\}$. Then we consider a_5, whose scenes are also all scheduled and appear first in the second group and last in the last group. Splitting these groups has no effect since a_5 appears in all scenes in the group so the partial schedule is unchanged. Similarly a_6 only appears in one group (the first) so this is split into those containing a_6 and those not to obtain $\{s_{10}\}$ $\{s_{12}\}$ $\{s_8\}$ $\{s_3, s_9, s_{11}\}$ $\{s_1, s_6\}$ $\{s_2, s_5\}$ $\{s_7\}$ $\{s_4\}$. The final resulting schedule is shown in Figure 7.□

Note that we can easily improve a heuristic solution of a scene scheduling problem by considering swapping the positions of any two pairs of scenes, and making the swap if it lowers the total cost. This heuristic method is explored in Cheng et al. (1993). We also tried a heuristic that attempted to build the schedule from the middle by first choosing the most expensive scene and then choosing the next scene that minimizes cost to left or right. However, our experiments indicate that the upper bounds provided by any heuristic have very little effect on the overall computation of the optimal order, probably because the bnd_schedule function overwrites the upper bound as soon as it finds a better solution. Hence, we did not explore many options for better heuristic solutions. Instead, we focused on devising better search strategies.

3.3 Looking Ahead

We can further reduce the search performed in bnd_schedule (and schedule) by looking ahead. That is, we examine each of the subproblems we are about to visit, and if we have already calculated an optimal value or correct bound for them, we can use this to get a better estimate of the lower bound cost. Furthermore, we can use this opportunity to change the lower bound function so that it memos any lower bound calculated in the hash table $scost$. The only modification required is to change the definition of the lower function to:

lower(Q)
 if ($scost[Q] = OPT(v)$) **return** v
 if ($scost[Q] = LB(v)$) **return** v
 lb := $\sum_{s \in Q} d(s) \times c(a(s))$ %% if we are using normal costs
 lb := 0 %% if we are using extra costs
 $scost[Q]$:= $LB(lb)$
 return lb

This has the effect of giving a much better lower bound estimate and, hence, reducing search. Lookahead is somewhat related to the lower bounding technique used in Russian Doll Search (Verfaillie et al. 1996), but in that case all smaller problems are forced to be solved before the larger problem is tackled, while lookahead is opportunistic, just using results that are already there.

3.4 Better Lower Bounds

If we are storing the lower bound computations, as described in the previous subsection, it may be worthwhile spending more time to derive a better lower bound. Here we describe a rather complex lower bound which is strong enough to reduce the number of subproblems examined by 1-2 orders of magnitude. We use the following straightforward result

LEMMA 4. *Let $a_i, b_i, i = 1, \ldots, n$ be positive real numbers. Let π be a permutation of the indices. Define $f(\pi) = \sum_{i=1}^{n}[a_{\pi(i)} * \sum_{j=1}^{i} b_{\pi(j)}]$. The permutation π which minimizes $f(\pi)$ satisfies $b_{\pi(1)}/a_{\pi(1)} \leq b_{\pi(2)}/a_{\pi(2)} \leq \ldots \leq b_{\pi(n)}/a_{\pi(n)}$.* □

This lemma allows us to solve certain special cases of the Talent Scheduling Problem with a simple sort. Consider the following special case. We have a set of actors $a_1, ..., a_n$ already on location, and a set of scenes $s_1, \ldots, s_n$ where s_i only involves the actor a_i for each i. Then given a schedule $s_{\pi(1)} s_{\pi(2)} \ldots s_{\pi(n)}$ where π is some permutation, the cost is given by $f(\pi) = \sum_{i=1}^{n} c(a_{\pi(i)}) * \sum_{j=1}^{i} d(s_{\pi(j)})$. This is of the form required for Lemma 4, and we can find the optimal scene permutation π_{opt} simply by sorting the numbers $d(s_i)/c(a_i)$ in ascending order. The minimum cost can then be calculated by a simple summation. Unfortunately, in general, the subproblems for which we wish to calculate lower bounds do not fall under the special case, as scenes generally involve multiple actors. To take advantage of the lemma then, we need to do much more work.

THEOREM 1. *Let Q be a set of scenes remaining to be scheduled. Let $A' = o(Q)$, the actors currently on location. Without loss of generality, let $A' = \{a_1, \ldots, a_n\}$. Let $Q' \subseteq Q$ be the set of unscheduled scenes that involve at least one actor from A'. Let $sc(s) = \sum_{a \in A' \cap a(s)} c(a)$. Let $x(a, s) = 1$ if $a \in a(s)$, and 0 otherwise. Let $w(a, s) = x(a, s) * c(a)/sc(s)$. Let $e(a) = \sum_{s \in Q'} w(a, s) * d(s)$. Let $f(\pi) = \sum_{k=1}^{n} c(a_{\pi(k)}) * \sum_{i=1}^{k} e(a_{\pi(i)})$. A correct lower bound on the extra cost for actors A' for scenes Q' is given by $f(\pi_{opt}) - \sum_{s \in Q'} d(s) * [sc(s) + \sum_{a \in A' \cap a(s)} c(a)^2/sc(s)]/2$, where π_{opt} is the permutation of the indices given by sorting $r(a_i) = e(a_i)/c(a_i)$ in ascending order.*

Proof: First we describe what each of the defined quantities mean. $sc(s)$ gives the sum of the cost of the actors for scene s, but only counting the actors which are currently on location. $w(a, s)$ is a measure of how much actor a is contributing to the cost of scene s. We have $0 \leq w(a, s) \leq 1$, and $\sum_{a \in A' \cap a(s)} w(a, s) = 1$. $e(a)$ is a weighted sum of the duration of the scenes that a is involved in, weighted by $w(a, s)$. $f(\pi)$ is constructed so that it follows the form required for Lemma 4 to apply, which we will take advantage of. The actual lower bound is given by the minimum value of $f(\pi)$, minus a certain constant.

Given any complete schedule that extends the current partial schedule, there is an order in which the on location actors $a_1, \ldots, a_n$ may finish. Without loss of generality, label the actors so that they finish in the order $a_1, a_2, ..., a_n$ (break ties randomly). We have the following inequalities for the cost of the remaining schedule $t(a)$ for each of these actors:

$$\begin{aligned} t(a_k) &\geq c(a_k) * \sum_{\{s \in Q' | \exists i, i \leq k, a_i \in a(s)\}} d(s) \\ &\geq c(a_k) * [\sum_{i=1}^{k} e(a_i) + \sum_{\{s \in Q' | a_k \in a(s)\}} [d(s) * (1 - \sum_{i=1}^{k} w(a_i, s))]] \end{aligned}$$

These inequalities hold for the following reasons. Consider a_k. Any scene which involves any of $a_1, \ldots, a_k$ must be scheduled before a_k can leave, since by definition $a_1, \ldots, a_{k-1}$ leave no later than a_k. So for such scenes s, we must pay $c(a_k) * d(s)$ for actor a_k, which gives rise to the first inequality. Now, in the second line, the scene durations from the first line are split up and summed together in a different way, with some terms thrown away. The second line consists of two sums within the outer set of square brackets. A scene which does not involve any of $a_1, \ldots, a_k$ will not be counted in any $e(a)$ in the first sum, and is not counted by the second sum which only counts scenes involving a_k. So as required, such durations do not appear in the second line. A scene which involves some of $a_1, \ldots, a_k$ will have part of its duration counted in the first sum. To be exact, a proportion $\sum_{i=1}^{k} w(a_i, s) \leq 1$ of it is counted in the first sum. The second sum counts the bits that were not counted in the first sum for scenes that involve a_k. Since the second line never counts more than $d(s)$ for any scene appearing in the first line, the inequality is valid.

Now, we split the last line of the inequality into its two parts and sum over the actors. Define U and V as follows:

$$U = \sum_{k=1}^{n} c(a_k) * \sum_{i=1}^{k} e(a_i)$$

$$V = \sum_{k=1}^{n} c(a_k) * \sum_{\{s \in Q' | a_k \in a(s)\}} [d(s) * (1 - \sum_{i=1}^{k} w(a_i, s))]$$

Then $U + V$ is a lower bound on the cost for the actor finish order $a_1, \ldots, a_n$. As can be seen, U corresponds to $f(\pi)$ in the theorem. Different permutations of actor finish order π will give rise to different values of U equal to $f(\pi)$. By applying Lemma 4, we can quickly find a lower bound on U over all possible actor finish orders. That is, for each actor a, we calculate $r(a) = e(a)/c(a)$. We then sort the actors based on $r(a)$ from smallest to largest and label them from a'_1 to a'_n. We then calculate U using finish order $a'_1, \ldots, a'_n$, which will give us a lower bound on U over all possible actor finish orders.

V on the other hand, although it looks like it depends on the actor finish order, actually evaluates to a constant.

$$\begin{aligned} V &= \sum_{k=1}^{n} c(a_k) * \sum_{\{s \in Q' | a_k \in a(s)\}} [d(s) * (1 - \sum_{i=1}^{k} w(a_i, s))] \\ &= \sum_{k=1}^{n} \sum_{\{s \in Q' | a_k \in a(s)\}} c(a_k) * [d(s) * (1 - \sum_{i=1}^{k} w(a_i, s))] \\ &= \sum_{s \in Q'} \sum_{k=1, a_k \in a(s)}^{n} c(a_k) * [d(s) * (1 - \sum_{i=1}^{k} w(a_i, s))] \\ &= \sum_{s \in Q'} \sum_{k=1, a_k \in a(s)}^{n} c(a_k) * d(s) - \sum_{s \in Q'} \sum_{k=1, a_k \in a(s)}^{n} c(a_k) * d(s) * \sum_{i=1}^{k} w(a_i, s) \\ &= \sum_{s \in Q'} \sum_{k=1, a_k \in a(s)}^{n} c(a_k) * d(s) - \sum_{s \in Q'} \sum_{k=1, a_k \in a(s)}^{n} c(a_k) * d(s) * \sum_{i=1, a_i \in a(s)}^{k} c(a_i)/sc(s) \\ &= \sum_{s \in Q'} \sum_{k=1, a_k \in a(s)}^{n} c(a_k) * d(s) - \sum_{s \in Q'} d(s)/sc(s) * \sum_{k=1, a_k \in a(s)}^{n} \sum_{i=1, a_i \in a(s)}^{k} c(a_k) * c(a_i) \end{aligned}$$

The first double sum is simply the base cost needed to pay each actor for each scene they appear in, and is clearly a constant. Of the second term, only the innermost double sum may be dependent on the actor finish order. Let $W(s) = \sum_{k=1, a_k \in a(s)}^{n} \sum_{i=1, a_i \in a(s)}^{k} c(a_k) * c(a_i)$.

$$\begin{aligned} 2 * W(s) &= 2 * \sum_{k=1, a_k \in a(s)}^{n} \sum_{i=1, a_i \in a(s)}^{k} c(a_k) * c(a_i) \\ &= \sum_{k=1, a_k \in a(s)}^{n} \sum_{i=1, a_i \in a(s)}^{k} c(a_k) * c(a_i) + \sum_{i=1, a_i \in a(s)}^{n} \sum_{k=i, a_k \in a(s)}^{n} c(a_k) * c(a_i) \\ &= \sum_{k=1, a_k \in a(s)}^{n} \sum_{i=1, a_i \in a(s)}^{k} c(a_k) * c(a_i) + \sum_{k=1, a_k \in a(s)}^{n} \sum_{i=k, a_i \in a(s)}^{n} c(a_i) * c(a_k) \\ &= \sum_{k=1, a_k \in a(s)}^{n} \sum_{i=1, a_i \in a(s)}^{n} c(a_k) * c(a_i) + \sum_{k=1, a_k \in a(s)}^{n} c(a_k) * c(a_k) \\ &= sc(s)^2 + \sum_{k=1, a_k \in a(s)}^{n} c(a_k)^2 \\ W(s) &= [sc(s)^2 + \sum_{k=1, a_k \in a(s)}^{n} c(a_k)^2]/2 \end{aligned}$$

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	
a_1	X	.	.	.	X	.	X	1
a_2	.	X	.	.	X	X	.	1
a_3	.	.	X	.	.	X	X	1
a_4	.	.	.	X	.	.	X	1
	1	1	1	1	1	1	1	

Figure 8 An Original Set of Remaining Scenes, Assuming a_1, a_2, a_3, a_4 are on Location.

Which is constant. Now, $U + V$ gives a lower bound for the total cost. A lower bound for the extra cost is simply $U + V$ minus the base cost of the actors A' for the scenes Q'. Luckily, this term already appears as the first term in V. Thus the lower bound for the extra cost is $f(\pi_{opt}) - \sum_{s\in Q'} d(s)/sc(s) * W(s) = f(\pi_{opt}) - \sum_{s\in Q'} d(s) * [sc(s) + \sum_{a\in A'\cap a(s)} c(a)^2/sc(s)]/2$ as claimed. □

EXAMPLE 7. Consider the scenes shown in Figure 8, where the cost/duration of each actor/scene is 1 for simplicity. To calculate $f(\pi_{opt})$, we need to calculate $r(a)$ and sort them. Since the costs are all 1, we have $r(a_1) = e(a_1) = 11/6$, $r(a_2) = e(a_2) = 2$, $r(a_3) = e(a_3) = 11/6$, $r(a_4) = e(a_4) = 4/3$. So we reorder the actors as $a'_1 = a_4, a'_2 = a_1, a'_3 = a_3, a'_4 = a_2$ and calculate $f(\pi)$ using finish order $a'_1, \ldots, a'_4$, which gives $f(\pi_{opt}) = 1 * 4/3 + 1 * (4/3 + 11/6) + 1 * (4/3 + 11/6 + 11/6) + 1 * (4/3 + 11/6 + 11/6 + 2) = 16.5$, which is a lower bound on U over all actor finish orders. Next, we calculate $\sum_{s\in Q} d(s) * [sc(s) + \sum_{a\in A'\cap a(s)} c(a)^2/sc(s)]/2 = 1 + 1 + 1 + 1 + 3/2 + 3/2 + 2 = 9$. Thus the lower bound for the extra cost at this node is $16.5 - 9 = 7.5$. □

If we are optimizing the extra cost (Section 2.5), then to implement this lower bound, we simply need to add the following code into the code for **lower** before the saving of the lower bound in $scost$.

```
A' := o(Q)
for (a ∈ A')
    r[a] := 0
for (s ∈ Q')
    a'(s) = a(s) ∩ A'
    total_cost := Σ_{i∈a'(s)} c(i)
    total_cost_sq := Σ_{i∈a'(s)} c(i)^2
    for (a ∈ a'(s))
        r[a] = r[a] + d(s)/total_cost
        lb = lb − d(s) * (total_cost + total_cost_sq/total_cost)/2
Sort A' based on r[a] in ascending order
c := Σ_{i∈A'} c(i)
for (a ∈ A')
    lb = lb + c * r[a] * c(a)
    c = c − c(a)
```

Clearly this is quite an expensive calculation

4. Double Ended Search

We will say an actor is *fixed* if we know the first and last scene where the actor appears. Knowing that an actor is fixed is useful, because the cost for that actor is fixed (thus the name) regardless of the schedule of the remaining intervening scenes, if any. For this reason it is beneficial to search for a solution by alternatively placing the next scene in the first remaining unfilled slot and the last remaining unfilled slot, since this will increase the number of fixed actors. Let B denote the set of scenes scheduled at the beginning, and E the set of scenes scheduled at the end. We know the cost of any actor appearing both in scenes of B and scenes

```
bnd_de_schedule(B, E, U)
   Q := S − B − E
   if (a(Q) ⊆ a(B) ∩ a(E)) return 0
   hv := hash_lookup(B, E)
   if (hv = OPT(v)) return v
   if (hv = LB(v) ∧ v > U) return v
   min := +∞
   T := Q
   while (T ≠ ∅)
      s := index min_{s∈T} cost(s, B, E) + lower(B ∪ {s}, E)
      T := T − {s}
      if (cost(s, B, E) + lower(B ∪ {s}, E) ≥ U) break
      sp := cost(s, B, E) + bnd_de_schedule(E, B ∪ {s}, U − cost(s, B, E))
      if (sp < min) min := sp
      if (min ≤ U) U := min
   if (min ≤ U) hash_set(B, E, OPT(min))
   else hash_set(B, E, LB(min))
   return min
```

Figure 9 Pseudo-Code for bounded Best-first Call-based Dynamic Programming Algorithm. bnd_de_schedule(Q, B, E) Returns the Minimum Cost Required for Scheduling the Set of Scenes Q if it is Less than or Equal to U, Otherwise it Returns a Lower Bound on the Minimal Cost

of E, since we know the duration of the remaining set of scenes $Q = S - B - E$. This strategy was used in the branch-and-bound solution of Cheng et al. (1993). A priori this might appear to be a bad strategy since the search space has increased: there are more subproblems of the form "schedule remaining scenes Q given scenes in B are scheduled before and scenes in E are scheduled after (where $B \cup Q \cup E = S$)", than there are "schedule remaining scenes Q given scenes in $S - Q$ are scheduled before". However, as we will see in the experiments, this is compensated by the fact that we will get much more accurate estimates on the cost of the remaining schedule.

The change in search strategy causes considerable changes to the algorithm. The sub problems are now defined by B the set of scenes scheduled at the beginning, and E the set of scenes scheduled at the end. The search tries to schedule each remaining scene s at the beginning of the remaining scenes, just after B, and then swaps the role of B and E to continue building the schedule. We can thus modify the cost function to ignore the cost of actors already fixed by B and E (i.e., those in $a(B) \cap a(E)$), and take only into account the cost of actors *newly* fixed by the scene. This can be done as follows:

$$\begin{aligned}\mathsf{cost}(s, B, E) = {} & d(s) \times c(l(s, S - B - E - \{s\}) - (a(B) \cap a(E))) \\ & + \textstyle\sum_{a \in ((a(s) - a(B)) \cap a(E))} d(S - B - E - \{s\}) \times c(a)\end{aligned}$$

where the first part adds the cost for scheduling scene s excluding the fixed actors $(a(B) \cap a(E))$, and the second part adds the cost of each actor a which is newly scheduled by s (appears in $(a(s) - a(B))$) and already scheduled at the end (appears in $a(E)$).

The lower bound cost function also has to change to ignore the actors fixed by B and E:

$$\mathsf{lower}(B, E) = \sum_{s \in S - B - E} d(s) \times c(a(s) - (a(B) \cap a(E)))$$

The code for the new algorithm is shown in Figure 9. The algorithm first tests whether there are any remaining actors to be scheduled: If $a(Q) \subseteq a(B) \cap a(E)$ then all actors playing in scenes of Q

are fixed (must be on location for the entire period regardless of Q schedule), and we simply return 0 (since their cost has already been taken into account). Otherwise, the algorithm checks the hash table to find whether the subproblem has been examined before. Note that we replaced the array of subproblems $scost[Q]$ by two functions hash_lookup(B, E), which returns the value stored for subproblem B, E, and hash_lookup(B, E, ov), which sets the stored value to ov. The remainder of the code is effectively identical to bnd_schedule using the new definitions of cost and lower. The only important thing to note is that the recursive call swaps the positions of beginning and end sets, thus forcing the next scene to be scheduled at the other end.

Note that any solution to the scene scheduling problem has an equivalent solution where the order of the scenes is reversed (a fact that has been noticed by many authors). We are implicitly using this fact in the definition of bnd_de_schedule when we reverse the order of the B and E arguments to make the search double ended, since we treat the problem starting with B and ending in E as equivalent to the problem starting with E and ending in B. We can also take advantage of this symmetry when detecting equivalent subproblems (i.e. when looking up whether we have seen the problem before). A simple way of achieving this is to store and lookup problems assuming that $B \leq E$ (that is, in lexicographic order).

$$\begin{aligned}\text{hash\_lookup}(B,E) &= \textbf{if } (B \leq E)\ scost[B,E]\ \textbf{else}\ scost[E,B]\\ \text{hash\_set}(B,E,ov) &= \textbf{if } (B \leq E)\ scost[B,E] := ov\ \textbf{else}\ scost[E,B] := ov\end{aligned}$$

4.1 Better Equivalent Subproblem Detection

While taking into account symmetries helps, we can further help the detection of equivalent subproblems by noticing that the cost of scheduling the scenes in $Q = S - B - E$ does not really depend on B and E. Rather, it depends on $o(B)$ and $o(E)$, i.e., on the set of actors that will always be on location at the beginning and at the end of Q, respectively.

EXAMPLE 8. Consider the partial schedule of the problem of Example 1 where $B = \{s_1, s_9, s_{12}\}$ and $E = \{s_3, s_5, s_6, s_{11}\}$. The remaining scenes to schedule are $Q = \{s_2, s_4, s_7, s_8, s_9, s_{10}\}$. An optimal schedule of Q (given B and E) is shown at the top of in Figure 10. The total cost ignoring the fixed actors a_1, a_2 and a_4 is 16 + 8 + 14 = 48.

Consider the subproblem where $B' = \{s_3, s_{11}, s_5, s_1\}$ and $E' = \{s_9, s_6, s_{12}\}$. The remaining scenes to schedule are still $Q = \{s_2, s_4, s_7, s_8, s_9, s_{10}\}$. Now $o(B') = o(E)$ and $o(E') = o(B)$ and hence any optimal order for the first subproblem can provide an optimal schedule for this subproblem, by reversing the order of the schedule. This is illustrated at the bottom of Figure 10. □

We can modify the hash function to take advantage of these subproblem equivalences. We will store the subproblem value on $o(B), o(E)$, and Q under the assumption that $o(B) \leq o(E)$.

hash_lookup(B, E)
  $Q := S - B - E$
  if $(o(E) < o(B))$ **return** $scost[o(E), o(B), Q]$
  return $scost[o(B), o(E), Q]$

hash_set(B, E, ov)
  $Q := S - B - E$
  if $(o(E) < o(B))$ $scost[o(E), o(B), Q] := ov$
  else $scost[o(B), o(E), Q] := ov$

In order to prove the correctness of the equivalence we need the following intermediate result.

LEMMA 5. *For every $Q, Q' \subseteq S$ such that $Q \cap Q' = \emptyset$ (which is the same as saying $Q' \subseteq S - Q$), we have that $a(Q) \cap a(Q') = o(Q) \cap a(Q') = a(Q') \cap o(Q') = o(Q) \cap o(Q')$.*

	s_{12}	s_1	s_9	$o(B)$	s_4	s_8	s_2	s_7	s_{10}	$o(E)$	s_5	s_6	s_{11}	s_3	$c(a)$
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
$d(s)$	1	1	1		1	2	1	1	2		3	1	1	2	

	s_3	s_{11}	s_5	s_1	$o(B')$	s_{10}	s_7	s_2	s_8	s_4	$o(E')$	s_9	s_6	s_{12}	$c(a)$
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
$d(s)$	2	1	3	1		2	1	1	2	1		1	1	1	

Figure 10 Two Equivalent Subproblems.

Proof: Let us first prove that $o(Q) \cap a(Q') = a(Q) \cap a(Q')$. We have that:

$$\begin{aligned} o(Q) \cap a(Q') &= (a(Q) \cap a(S-Q)) \cap a(Q') \\ &\quad \text{By definition of } o(Q) \\ &= a(Q) \cap (a(S-Q) \cap a(Q')) \\ &\quad \text{By associativity of } \cap \\ &= a(Q) \cap a(Q') \\ &\quad \text{By hypothesis of } Q' \subseteq S-Q \end{aligned}$$

A symmetric reasoning can be done to prove that $a(Q) \cap o(Q') = a(Q) \cap a(Q')$. To prove that $o(Q) \cap o(Q') = a(Q) \cap a(Q')$ we follow a similar reasoning:

$$\begin{aligned} o(Q) \cap o(Q') &= (a(Q) \cap a(S-Q)) \cap (a(Q') \cap a(S-Q')) \\ &\quad \text{By definition of } o(Q) \\ &= (a(Q) \cap a(S-Q')) \cap (a(S-Q) \cap a(Q')) \\ &\quad \text{By associativity of } \cap \\ &= a(Q) \cap a(Q') \\ &\quad \text{By hypothesis of } Q' \subseteq S-Q \text{ and } Q \subseteq S-Q' \end{aligned}$$

□

Given the above result, one could decide to hash on $a(B)$ and $a(E)$ (rather than on $o(B)$ and $o(E)$). This is also correct but it would miss some equivalences since: while $o(B) \cap o(E) = a(B) \cap a(E)$, $a(B)$ might contain more actors than $o(B)$, those who start and finish within B and will thus never be on location during the scenes in Q. Therefore, these actors are not relevant for Q. The same can be said for $a(E)$ and $o(E)$.

THEOREM 2. *Let* $\Pi_1\Pi_2\Pi_3$ *and* $\Pi_4\Pi_2\Pi_5$ *be two permutations of* S *such that* $o(\Pi_4) = o(\Pi_1)$, $o(\Pi_5) = o(\Pi_3)$. *Then, the cost of every scene of* Π_2 *is the same in* $\Pi_1\Pi_2\Pi_3$ *as in* $\Pi_4\Pi_2\Pi_5$.

Proof: Without loss of generality, let Π_2 be of the form $\Pi_2' s \Pi_2''$. We will show that for the cost of s is the same in $\Pi_1\Pi_2\Pi_3$ and $\Pi_4\Pi_2\Pi_5$. Now

$$\begin{aligned} l(s, \Pi_2''\Pi_3) &= a(s) \cup (a(\Pi_2''\Pi_3) \cap a(\Pi_1\Pi_2')) \\ &\quad \text{By definition of } l(s, Q) \end{aligned}$$

$= a(s) \cup ((a(\Pi_2'') \cup a(\Pi_3)) \cap (a(\Pi_1) \cup a(\Pi_2')))$
By definition of $a(Q)$
$= a(s) \cup ((a(\Pi_2'') \cap a(\Pi_1)) \cup (a(\Pi_2'') \cap a(\Pi_2')) \cup (a(\Pi_3) \cap a(\Pi_1)) \cup (a(\Pi_3) \cup a(\Pi_2')))$
Distributing $\cap$ over $\cup$
$= a(s) \cup ((a(\Pi_2'') \cap o(\Pi_1)) \cup (a(\Pi_2'') \cap a(\Pi_2')) \cup (o(\Pi_3) \cap o(\Pi_1)) \cup (o(\Pi_3) \cup a(\Pi_2')))$
by the Lemma 5
$= a(s) \cup ((a(\Pi_2'') \cap o(\Pi_4)) \cup (a(\Pi_2'') \cap a(\Pi_2')) \cup (o(\Pi_5) \cap o(\Pi_4)) \cup (o(\Pi_5) \cup a(\Pi_2')))$
Since $o(\Pi_4) = o(\Pi_1)$ and $o(\Pi_5) = o(\Pi_3)$
$= a(s) \cup ((a(\Pi_2'') \cap a(\Pi_4)) \cup (a(\Pi_2'') \cap a(\Pi_2')) \cup (a(\Pi_5) \cap a(\Pi_4)) \cup (a(\Pi_5) \cup a(\Pi_2')))$
$= a(s) \cup ((a(\Pi_2'') \cup a(\Pi_5)) \cap (a(\Pi_4) \cup a(\Pi_2')))$
$= l(s, \Pi_2''\Pi_5)$

4.2 Revisiting the Previous Optimizations

Once we are performing double ended search, we introduce fixed actors which are no longer of any importance to the remaining subproblem since their cost is fixed. We may be able to improve the previous optimizations by ignoring fixed actors whenever performing a double ended search.

4.2.1 Preprocessing The second preprocessing step (concatenating duplicate scenes) can now be applied during search. This is because, given fixed actors $F = a(B) \cap a(E)$, we can apply Lemma 1 if $a(s_1) \cup F = a(s_2) \cup F$, since the cost of the fixed actors is irrelevant. This means we should concatenate any scenes in $Q = S - B - E$ where $a(s_1) \cup F = a(s_2) \cup F$. We can modify the search strategy in bnd_de_schedule to break the scenes in Q into equivalent classes $Q_1, \ldots Q_n$ where $\forall s_1, s_2 \in Q_i . a(s_1) \cup F = a(s_2) \cup F$, and then consider scheduling each equivalence class. In many cases the equivalence class will be of size one!

4.2.2 Scheduling Actor Equivalent Scenes First Lemma 2 can be extended so that we can always schedule a scene s first where $o(B) = a(s) \cup F$ since the on location actors will include the fixed actors and the extra cost for them will be payed for scene s wherever it is scheduled.

4.2.3 Pairwise Subsumption The extension of Lemma 3 also holds if $a(s_1) \cup F \subseteq a(s_2) \cup F$ and $a(B) \cup a(s_1) \supseteq a(s_2)$ (since $F \subseteq a(B)$). But this means we need to do a full pairwise comparison of all scenes in $Q = S - B - E$, for each subproblem considered. We did implement this, and although it did cut down search substantially, the overhead of the extra comparison did not pay off. This is the only optimisation not used in the experimental evaluation.

4.2.4 Optimizing Extra Cost This is clearly applicable in the doubled ended case, but it complicates the computation of cost(s, B, E) since we now have to determine exactly which scenes a newly fixed actor occurs in, rather than just adding the cost of the actor for the entire duration of the remaining scenes.

4.2.5 Looking Ahead This is applicable as before. Note that lower takes the same arguments (B, E) (excluding the upper bound) as bnd_de_schedule. We have to modify the definition of lower(B, E) to make use of hash_lookup and hash_set.

4.2.6 Better Lower Bounds The same reasoning on better lower bounds can be applied to the set of actors $o(B) - F$, since the actors in F will always be on location in the remaining subproblem. Indeed, we sum the results of the better lower bounds calculated from both ends for $o(B) - F$ and $o(E) - F$, since the actors in these sets cannot overlap (by the definition of F).

4.2.7 Better Equivalent Subproblem Detection We could improve equivalent subproblem detection by noticing that the fixed actors play no part in determining the schedule of the remaining scenes $Q = S - B - E$. We could thus build a hash function based on the form of the remaining scenes after eliminating the fixed actors $F = a(B) \cap a(E)$. But the cost of determining this reduced form seems substantial since, in effect, we have to generate new scenes and hash on sets of them. We have not attempted to implement this approach.

Table 1 Arithmetic Mean Solving Time (ms) for Structured Problems of Size n, and Relative Slowdown if the Optimization is Turned Off

n	Time (ms)	2.3	2.4	3.3	4.2.1	3.2	3.4	2.5	4.1	4	3
18	78	1.09	1.29	1.24	1.04	1.01	2.75	1.21	0.98	0.44	31.94
19	157	1.15	1.32	1.23	1.04	0.99	3.06	1.20	1.05	0.53	33.18
20	190	1.12	1.39	1.18	1.03	0.99	3.29	1.19	1.02	0.45	47.05
21	317	1.17	1.35	1.17	1.04	1.02	4.13	1.21	1.06	0.51	61.12
22	702	1.18	1.37	1.24	1.06	0.99	3.75	1.19	1.14	0.69	52.16
23	870	1.19	1.48	1.23	1.05	1.01	4.30	1.20	1.09	0.63	—
24	1269	1.23	1.47	1.23	1.11	1.02	5.08	1.19	1.16	0.74	—
25	1701	1.26	1.55	1.25	1.08	1.00	5.32	1.20	1.20	0.86	—
26	2934	1.29	1.63	1.33	1.12	1.01	6.15	1.22	1.25	0.98	—
27	3699	1.31	1.79	1.36	1.14	1.02	7.12	1.23	1.25	1.07	—
28	5172	1.35	1.92	1.38	1.15	1.00	7.83	1.24	1.26	1.17	—

Table 2 Arithmetic Mean Subproblems Solved for Structured Problems of Size n, and Relative Increase if the Optimization is Turned Off

n	Subproblems	2.3	2.4	3.3	4.2.1	3.2	3.4	2.5	4.1	4	3
18	5091	1.08	1.18	1.02	1.06	1.06	10.21	1.00	1.07	0.93	73.29
19	9699	1.12	1.22	1.03	1.09	1.01	11.18	0.99	1.11	1.43	75.43
20	10467	1.11	1.26	1.02	1.07	1.01	12.36	1.00	1.09	0.95	114.63
21	16154	1.15	1.21	1.02	1.07	1.03	15.88	1.00	1.14	1.24	152.42
22	36531	1.17	1.23	1.03	1.12	1.00	13.38	0.99	1.21	1.97	121.31
23	42224	1.17	1.32	1.03	1.10	1.02	15.73	1.00	1.13	1.52	—
24	59349	1.22	1.33	1.02	1.21	1.03	18.50	0.98	1.16	1.98	—
25	77766	1.25	1.37	1.03	1.17	1.00	19.00	0.98	1.19	2.08	—
26	136778	1.28	1.40	1.03	1.23	1.01	20.49	1.00	1.20	2.51	—
27	167232	1.28	1.50	1.03	1.25	1.04	23.63	0.99	1.19	2.70	—
28	233328	1.31	1.56	1.03	1.27	1.01	25.04	0.99	1.19	2.88	—

5. Experimental Results

We tested our approach on the two sets of problem instances detailed below. All experiments were run on Xeon Pro 2.4GHz processors with 2GB RAM running Red Hat Linux 5.2. The dynamic programming code is written in C, with no great tuning or clever data structures, and many runtime flags to allow us to compare the different versions easily. The dynamic programming software was compiled with gcc 4.1.2 using -O3. Timings are calculated as the sum of user and system time given by `getrusage`, since it accords well with wall-clock times for these CPU-intensive programs. For the problems that take significant time we observed around 10% variation in timings across different runs of the same benchmark.

5.1 Structured Benchmarks

The first set of benchmarks are structured problems based on the realistic talent scheduling of Mob Story first used in Cheng et al. (1993). We use these problems to illustrate the effectiveness of the different optimizations.

We first extended the benchmarks film103, film105, film114, film116, film118, film119 used in Smith (2005), adding three new actors to each problem to bring it to 11, and bringing the number of scenes to 28 (the original problems each involve 8 actors and either 18 or 19 scenes). This gave us 6 *base* problems of size 11×28. These base problems were constructed in such a way that preprocessing did not simplify them (so that the number of "important" actors and scenes was known).

Then, from each base problem we generated smaller problems by removing in turn newly added scenes. In particular, for each base problem we obtained 10 problems ranging from 11×27 to 8×18, where each problem in the sequence is a subproblem of the larger ones, and the original problem from Smith (2005) was included.

From each base problem we also generated smaller problems by randomly removing k scenes where k varied from 1 to 10. In particular, for each base problem we obtained 10 problems ranging from 11×27 to 11×18, where the sets of removed scenes in differently sized problem are unrelated (as opposed to have a subset/superset relationship).

In total this created 126 core problems.

From each *core* problem we generated three new variants: equal duration where all durations are set to 1, equal cost where the cost of all actors are set to 1, and equal cost and duration where all durations and costs are set to 1.

We compared the executions of running the dynamic program with all optimizations enabled, and individually turning off each optimization. The optimizations are: scheduling actor equivalent scenes first (Section 2.3), pairwise subsumption (Section 2.4), looking ahead (Section 3.3), concatenating duplicate scenes (Section 4.2.1), upper bounds (Section 3.2), better lower bounds (Section 3.4), optimizing extra cost (Section 2.5), better equivalent subproblem detection (Section 4.1), double-ended search (Section 4), and bounded dynamic programming (Section 3). The average times in milliseconds obtained by running the dynamic program with all optimizations enabled for each size n, are shown in the second column of Table 1. The remaining columns show the relative average time when each of the optimizations is individually turned off. For the last column without bounding, only the problems up to size 22 are shown. Table 2 shows the same results in terms of the number of subproblems solved (that is, the number of pairs (B, E) appearing in calls to bnd_de_schedule or Q in earlier variants).

The tables clearly show that bounded dynamic programming (Section 3) is indispensable for solving these problems. Better lower bounding is clearly the next most important optimization, massively reducing the number of subproblems visited. Doubled-ended search (Section 4) is also very important except for the fact that better lower bounding (Section 3.4) improves the single-ended search much more than it does the double-ended search, so only on the larger examples does it begin to win. Without better lower bounding it completely dominates single-ended search. The next most effective optimization is pairwise subsumption (Section 2.4). Looking ahead (Section 3.3) and scheduling actor equivalent scenes first (Section 2.3) are quite beneficial, as are optimizing extra cost (Section 2.5) and better equivalent subproblem detection (Section 4.1). The upper bounds optimization (Section 3.2) is clearly unimportant, only reducing the number of problems slightly. Note that while some optimizations given more or less constant improvements with increasing size, most are better as size increases.

If we look at the different variants individually (in results not shown) we find that the equal duration variants are slightly (around 7-10%) harder than the core problems, while the equal cost and equal cost and duration variants are 3–4 times harder than the core problems, indicating that cost is very important for pruning.

5.2 Random Benchmarks

The second set of benchmarks is composed of randomly generated benchmarks. We use these problems to show the effect of number of actors and number of scenes on problem difficulty.

The problems were generated in a manner almost identical to that used in Cheng et al. (1993): for a given combination of m actors and n scenes we generate for each actor $i \in 1..m$ (a) a random number[1] $n_i \in 2..n$ indicating the number of scenes actor i is in, (b) n_i different random numbers between 1 and n indicating the set of scenes actor i is in, and (c) a random number between 1 and 100 indicating the cost of actor i. For each combination of actors $m \in \{8, 10, 12, 14, 16, 18, 20, 22\}$ and scenes $n \in \{16..64\}$, we generate 100 problems. Note that, given the above method, a scene might contain no actors while an actor must be involved in at least two scenes (and at most all). We ensured preprocessing could not simplify any instance.

We ran the instances with a memory bound of 2Gb. Table 3 shows the average time in milliseconds obtained for finding an optimal schedule for all random instances of each size which did not run out of

[1] In Cheng et al. (1993) they generate a number from $1..n$ but actors appearing in only 1 scene are uninteresting (see Section 2.2).

Table 3 Arithmetic Mean Solving Time (ms) for Random Problems with m Actors and n Scenes.

	Number of Scenes n												
m	16	18	20	22	24	26	28	30	32	34	36	38	40
8	7	20	39	94	141	323	362	685	1403	2291	2977	2408	7101
10	11	33	85	165	441	650	1981	2531	3179	8901	10690	13426	20907
12	21	47	149	319	829	2056	3830	6674	10082	13155	20903	—	—
14	25	75	255	759	1519	3700	8862	12705	17602	—	—	—	—
16	41	129	357	1012	2602	6284	14130	23270	—	—	—	—	—
18	53	221	533	1463	3708	11546	18797	—	—	—	—	—	—
20	87	248	757	2745	6680	15414	21194	—	—	—	—	—	—
22	119	338	997	2855	11090	18672	—	—	—	—	—	—	—

m	42	44	46	48	50	52	54	56	58	60	62	64
8	7697	15669	19004	21703	23939	25891	49547	42433	49406	61351	62089	—
10	25903	—	—	—	—	—	—	—	—	—	—	—

Table 4 Arithmetic Mean Subproblems Solved for Random Problems with m Actors and n Scenes.

	Number of Scenes n												
m	16	18	20	22	24	26	28	30	32	34	36	38	40
8	569	1477	2431	5440	6905	17825	20020	37803	81388	124579	153515	113402	296798
10	784	2377	5408	8747	23898	33692	108048	128041	149676	387515	420769	484846	663511
12	1780	3032	9261	15757	41955	113971	184998	309110	410699	510775	668273	—	—
14	1846	4880	15388	43122	74523	194726	403624	521340	626265	—	—	—	—
16	3071	8153	20366	50527	128969	301001	623235	889799	—	—	—	—	—
18	3218	16317	29785	71885	175373	510349	742264	—	—	—	—	—	—
20	4911	14612	41608	153560	340470	668144	768588	—	—	—	—	—	—
22	4929	17559	52531	138078	547756	782389	—	—	—	—	—	—	—

m	42	44	46	48	50	52	54	56	58	60	62	64
8	312200	575387	610651	501283	585939	578558	747825	788145	832748	924486	869846	—
10	736360	—	—	—	—	—	—	—	—	—	—	—

memory, while Table 4 shows the average number of subproblems solved. The entries — show where less than 80 of the 100 instances solved without running out of memory. The schedules were computed using all optimizations. The results show that while the number of scenes is clearly the most important factor in the difficulty of the problem, if the number of actors is small then the problem difficulty is limited. While increasing the number of actors increases difficulty, as it grows larger than the number of scenes, the incremental difficulty decreases. Note also that the random problems are significantly easier than the structured problems.

While we should be careful when reading these tables, since the difficulty of each 100 random benchmarks considered in each cell can vary remarkably (the standard deviation is usually larger than the average shown), the trend is clear enough.

6. Related Work

The talent scheduling problem (which appears as prob039 in CSPLIB (CSPLib 2008) where it is called the rehearsal problem) was introduced by Cheng et al. (1993). They consider the problem in terms of shooting days instead of scenes so, in effect, all scenes have the same duration. Note, however, that once we make use of Lemma 1 the requirement for different durations arises in any case. They give one example of a real scene scheduling problem, arising from the film Mob Story, containing 8 actors and 28 scenes. They show that the problem is NP-hard even in the very restricted case of each actor appearing in exactly two scenes and all costs and durations being one, by reduction to the optimal linear arrangement (OLA) problem (Garey et al. 1976)

In their paper they consider two methods to solve the scene scheduling problem. The first method is a branch and bound search, where they search for a schedule by filling in scenes from both ends in an

	25	26	24	27	22	23	19	20	21	5	28	8	11	9	6	7	9	2	16	17	18	3	15	13	14	1	12	3	
Luce	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	X	X	X	–	–	–	X	X	X	X	X	X	X	10
Tom	.	.	.	.	.	X	X	X	X	–	X	X	–	X	–	–	–	X	–	X	X	X	–	X	X	X	X	.	4
Mindy	.	.	.	.	.	.	X	X	X	X	–	X	–	X	–	–	–	X	–	X	X	X	.	.	.	.	.	.	5
Maria	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	X	–	X	X	X	.	.	.	5
Gianni	.	X	X	X	–	–	X	–	–	–	X	X	–	X	–	–	–	X	X	.	.	.	.	.	.	.	.	.	5
Dolores	.	.	X	X	X	X	X	X	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	40
Lance	.	.	.	.	.	.	.	X	X	X	–	X	–	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	4
Sam	.	.	.	.	.	.	.	.	.	.	.	X	X	X	X	X	X	.	.	.	.	.	.	.	.	.	.	.	20
$d(s)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

Figure 11 An Optimal Schedule for the Film "Mob Story"

alternating fashion (double ended search). They optimize on extra cost, and the lower bounds they use are simply the result of fixed costs (so equivalent to the definition of lower in Section 4 minus the fixed costs). They do not store equivalent solutions and, hence, are very limited in the size of the problem they can tackle. Their experiments go up to 14 scenes and 14 actors.

The second method is a simple greedy hill climbing search. Given a starting schedule they consider all possible swaps of pairs of scenes, and move to the schedule after a swap if the resulting cost is less. They continue doing this until they reach a local minimum. On their randomly generated problems the heuristic approach gives answers around 10-12% off optimal regardless of size. They use this algorithm to re-schedule Mob Story with an extra cost of $16,100 as opposed to the hand solution of $36,400. This solution required 1.05 second on their AMDAHL mainframe. In comparison, our best algorithm finds an optimal answer with extra cost $14,600 in 0.1 seconds on a Xeon Pro 2.4GHz processor (which is admittedly very much more powerful). The search only considers 6,605 different subproblems. Note that after preprocessing, it only involves 20 scenes. The optimal solution found is shown in Figure 11 (costs are divided by 100).

Adelson et al. (1976) define a restricted version of the talent scheduling problem for rehearsal scheduling where the costs of all actors are uniform, and also note how it can be used for an application in archeology. They give a dynamic programming formulation as a recurrence relation, more or less identical to that shown at the beginning of Section 2. They report solving an instance (from a real archaeological problem) with 26 "actors" and 16 "scenes" in 84 seconds on a CDC 7600 computer. We were not able to locate this benchmark.

Smith (2005, 2003) uses the talent scheduling problem as an example of a permutation problem. These papers solved the problem using constraint programming search with caching of search states, which is very similar to dynamic programming with bounds. The paper considers both scheduling from one end, or from both ends. This paper was the first to use a form of pairwise subsumption, restricted to the case where the scenes differ by at most 2 actors. It also used the preprocessing of merging identical scenes (without proof). This was the first approach (we are aware of) to calculate the optimal solution to the Mob Story problem.

A comparison of the approaches is shown in Table 5. The table shows the sizes (after preprocessing). Note that the timing results for Smith (2005) are for a 1.7GHz Pentium M PC running ILOG Solver 6.0, whereas our results are for Xeon Pro 2.4GHz processors running gcc on Red Hat Linux. However, note also that there is around 3 orders of magnitude difference between our times and those of Smith (2005). Also the number of cached states in the approach of Smith (2005) is around two orders of magnitude bigger than the number of subproblems (which is the equivalent measure). This is probably a combination of our better lower bounds, better detection of equivalent states and better search strategy. A web page with the problems and solutions can be found at `www.csse.unimelb.edu.au/~pjs/talent/`.

The talent scheduling problem is a generalization of the Optimal Linear Arrangement (OLA) problem (see Cheng et al. (1993)). This is a very well investigated graph problem, with applications including VLSI design (Hur and Lillis 1999), computational biology (Karp 1993), and linear algebra (Rose 1970). The OLA is known to be very hard to solve, it has no polynomial time approximation scheme unless NP-complete problems can be solved in randomized sub-exponential time (Ambühl et al. 2007). Unfortunately, the problem size in this domain is in the thousands, which means methods that find exact linear arrangements

Table 5 Comparison with the Approach of Smith (2005) on the Examples from that Paper.

	Size		Smith		this paper	
Problem	Actors	Scenes	Time	Cached states	Time	Subproblems
MobStory	8	20	64.71s	136,765	108ms	6,605
film105	8	18	16.07s	40,511	20ms	1,108
film116	8	19	125.8s	225,314	156ms	13,576
film119	8	18	70.80s	144,226	84ms	7,105
film118	8	19	93.10s	205,190	40ms	1,980
film114	8	19	127.0s	267,526	84ms	4,957
film103	8	19	76.69s	180,133	64ms	4,103
film117	8	19	76.86s	174,100	96ms	7,227

(as dynamic programming does), cannot be applied. Interestingly, there are heuristic methods (Koren and Harel 2002) that use exact methods as part of the entire process, and our algorithm could potentially be applied here.

The talent scheduling problem is highly related to the problem of minimizing the maximum number of open stacks. In this problem there are no durations, or costs and the aim is to minimize the maximum number of actors on location at any time. The problem has applications in cutting, packing and VLSI design problems. Compared to the talent scheduling problem, the open stacks problem has been well studied (see e.g.Yuen (1991, 1995), Yuen and Richardson (1995), Yannasse (1997), Faggioli and Bentivoglio (1998), Becceneri et al. (2004)). The best current solution is our dynamic programming approach (Chu and Stuckey 2009), but surprisingly almost none of the methods used there to improve the base dynamic programming approach are applicable to the talent scheduling problem. In the end the solutions are quite different, probably because the open stacks problem, while also NP-hard, is fixed parameter tractable (Yuen and Richardson 1995), as opposed to the talent scheduling problem.

7. Conclusion

The talent scheduling problem is a very challenging combinatorial problem, because it is very hard to compute accurate bounds estimates from partial schedules. In this paper we have shown how to construct an efficient dynamic programming solution by carefully reasoning about the problem to reduce search, as well as adding bounding and searching in the right manner. The resulting algorithm is orders of magnitude faster than other complete algorithms for this problem, and solves significantly larger problems than other methods.

There is still scope to improve the dynamic programming solution, by determining better heuristic orders in which to try scheduling scenes, and possibly determining better dynamic lower bounds by reasoning on the graph of actors that share scenes. One very surprising thing for us, was how much harder the talent scheduling problem is than the highly related problem of minimizing the maximum number of open stacks.

Acknowledgments

The first two authors of this paper would like to thank Professor Manuel Hermenegildo at IMDEA Software, Universidad Polytecnica de Madrid, Spain, who they were visiting while this work was undertaken. We would like to thank Barbara Smith for many interesting discussions on the talent scheduling problem, and giving us her example data files. We would also like to thank the reviewers for their careful reviewing which improved the paper substantially. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

Adelson, R.M., J.J. Norman, G. Laporte. 1976. A dynamic programming formulation with diverse applications. *Operational Research Quarterly* **27** 119–121.

Ambühl, C., M. Mastrolilli, O. Svensson. 2007. Inapproximability results for sparsest cut, optimal linear arrangement, and precedence constrained scheduling. *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 329–337.

Becceneri, J.C., H.H. Yannasse, N.Y. Soma. 2004. A method for solving the minimization of the maximum number of open stacks problem within a cutting process. *Computers & Operations Research* **31** 2315–2332.

Cheng, T. C. E., J. E. Diamond, B. M. T. Lin. 1993. Optimal scheduling in film production to minimize talent hold cost. *Journal of Optimization Theory and Applications* **79** 479–482.

Chu, G., P.J. Stuckey. 2009. Minimizing the maximum number of open stacks by customer search. I. Gent, ed., *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, *LNCS*, vol. 5732. Springer-Verlag, 242–257.

CSPLib. 2008. Csplib. www.csplib.org.

Faggioli, E., C.A. Bentivoglio. 1998. Heuristic and exact methods for a cutting sequencing problem. *European Journal of Operational Research* **110** 564–575.

Garey, M.R., D.D. Johnson, L. Stockmeyer. 1976. Some simplified NP-complete graph problems. *Theoretical Computer Science* **1** 237–267.

Hur, Sung-Woo, John Lillis. 1999. Relaxation and clustering in a local search framework: application to linear placement. *Proceedings of the 36th ACM/IEEE conference on Design Automation Conference,*. 360–366.

Karp, Richard M. 1993. Mapping the genome: some combinatorial problems arising in molecular biology. *Proceedings of the twenty-fifth annual ACM Symposium on Theory of Computing*. 278–285.

Koren, Yehuda, David Harel. 2002. A multi-scale algorithm for the linear arrangement problem. *Graph-Theoretic Concepts in Computer Science, 28th International Workshop*. LNCS, Springer, 296–309.

Puchinger, J., P.J. Stuckey. 2008. Automating branch-and-bound for dynamic programs. R. Glück, O. de Moor, eds., *Proceedings of the ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation (PEPM '08)*. ACM, 81–89. doi:http://doi.acm.org/10.1145/1328408.1328421.

Rose, D. J. 1970. Triangulated graphs and the elimination process. *J. Math. Appl.* **32** 597–609.

Smith, B.M. 2003. Constraint programming in practice: Scheduling a rehearsal. Tech. rep., StAndrews University. Technical Report APES-67-2003, APES Research Group.

Smith, B.M. 2005. Caching search states in permutation problems. P. Van Beek, ed., *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming - CP 2005,*, *LNCS*, vol. 3709. Springer, 637–651.

Verfaillie, G., M. Lemaitre, T. Schiex. 1996. Russian doll search for solving constraint optimization problems. *Proceedings of the National Conference on Artificial Intelligence (AAAI96)*. 181–187.

Yannasse, H.H. 1997. On a pattern sequencing problem to minimize the maximum number of open stacks. *European Journal of Operational Research* **100** 454–463.

Yuen, B.J. 1991. Heuristics for sequencing cutting patterns. *European Journal of Operational Research* **55** 183–190.

Yuen, B.J. 1995. Improved heuristics for sequencing cutting patterns. *European Journal of Operational Research* **87** 57–64.

Yuen, B.J., K.V. Richardson. 1995. Establishing the optimality of sequencing heuristics for cutting stock problems. *European Journal of Operational Research* **84** 590–598.