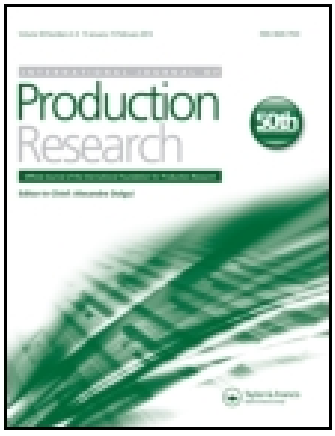

International Journal of Production Research
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tprs20

Iterated beam search for the combined car sequencing and level scheduling problem

Mesut Yavuz [a]

[a] Harry F. Byrd, Jr. School of Business , Shenandoah University , 1460 University Drive, Winchester , VA , 22655 , USA

Published online: 27 Mar 2013.

To cite this article: Mesut Yavuz (2013) Iterated beam search for the combined car sequencing and level scheduling problem, International Journal of Production Research, 51:12, 3698-3718, DOI: 10.1080/00207543.2013.765068

To link to this article: http://dx.doi.org/10.1080/00207543.2013.765068

International Journal of Production Research, 2013
Vol. 51, No. 12, 3698–3718, http://dx.doi.org/10.1080/00207543.2013.765068

Taylor & Francis
Taylor & Francis Group

Iterated beam search for the combined car sequencing and level scheduling problem

Mesut Yavuz*

Harry F. Byrd, Jr. School of Business, Shenandoah University, 1460 University Drive, Winchester, VA 22655, USA

(*Received 5 December 2011; final version received 27 December 2012*)

The level scheduling problem is concerned with the final stage of a multi-stage just-in-time production system so that different models of a product are evenly distributed in a discrete production sequence, thereby making the problem practically an unconstrained optimisation problem. The car sequencing problem, on the other hand, is a constraint satisfaction problem based on a number of options constricting the final assembly schedule. The combined car sequencing and level scheduling problem aims to find the optimal production schedule that evenly distributes different models over the planning horizon and satisfies all option constraints. This paper proposes a parametric iterated beam search algorithm for the combined problem that can be used either as a heuristic or as an exact optimisation method. The paper includes a computational study based on a 54-instance test bed that proves the effectiveness of the proposed algorithm.

Keywords: JIT; heuristics

1. Introduction

Car manufacturing is typically performed on mixed-model assembly lines that produce a variety of cars in any given planning period, such as a week, a day or even a shift. The variability of the cars produced on a line is kept to a manageable size by dedicating separate production lines to different model lines. However, the existence of optional features such as a sunroof, leather seats, navigation system, etc. gives rise to an enormous number of variants within a model line. Relatively low demand volumes of variants form a low-volume high-mix production environment that has its own set of operational challenges. For example, the production of different variants should be spaced out as evenly as possible over the planning horizon to increase responsiveness to consumers' demands, which, in turn, creates a need to almost constantly switch over from one variant to another, and setup times should be reduced, if not eliminated, to accommodate these frequent switch overs.

One of the biggest concerns with a car assembly line is to prevent line stoppages. The constant need to switch over from one variant to another can be handled very efficiently in a perfectly synchronised assembly line where each variant takes a unit time in each workstation, and a car leaves the assembly line every time unit (the cycle time). In practice, some options do not increase the processing time requirement in the workstation where they are installed, and, hence, lend themselves to synchronisation. For example, the time required to install leather seats may be the same as that of cloth seats. Other options, e.g. a navigation system, may increase the processing time moderately. In an extreme case, an option occupies an entire workstation and its time requirement is excessive. For example, a car with the sunroof option has to go through the whole sunroof installation process, whereas a car without the option does not take any time in the sunroof workstation. Varying task requirements and processing times of product variants prohibit perfect synchronisation and render the assembly line balancing problem challenging (Boysen, Fliedner, and Scholl 2009). After the balancing of a mixed-model assembly line, the remaining variability is handled in sequencing.

A common approach in sequencing to prevent line stoppages is spacing. That is, subsequent occurrences of an option are separated by at least a certain number of time units, or the number of variants requiring a certain option in every subsequence of the overall sequence is limited. For example, cars requiring the sunroof option may be limited to at most one in any subsequence of three cars. Factors affecting the option constraints include the time requirement of the option, the cycle time, the length of the station, the number of operators or operator teams working in the station and the demand rate of the option. The interested reader is referred to Lesert et al. (2011) for a detailed account of determining for which types of options a spacing constraint should be defined and how much space is necessary for an option. The *Car Sequencing Problem* (CSP), first proposed by Parello, Kabat, and Wos (1986), is concerned with finding feasible car sequences in the

*Email: myavuz@su.edu

presence of multiple option constraints. The CSP has attracted significant interest as a constraint satisfaction problem since the 1980s (Dincbas, Simonis, and van Hentenryck 1988; van Hentenryck Simonis, and Dincbas 1992; Tsang 1993). A test-bed for the CSP is included in CSPLib: a library for constraints (Gent and Walsh 1999).

Just-in-time (JIT) manufacturing is another crucial consideration in car manufacturing. Created at Toyota, the JIT philosophy is widely practiced in a gamut of industries around the world and it is shown to have a positive impact on company performance (Mackelprang and Nair 2010). Following Monden's (1983) seminal book, the JIT manufacturing philosophy has received tremendous interest both from practitioners and researchers in the past few decades. One of the key characteristics of JIT systems is pull production. In order to make pull production viable, production sequences must be smoothed. More specifically, different variants of a model produced on a shared production line should be distributed over the sequence as evenly as possible. This problem, namely the Production Smoothing, or the *Level Scheduling Problem* (LSP), has been extensively studied in the operations research literature (Yavuz and Akcali 2007). In a great majority of the work in the literature, the LSP is formulated as a sequencing problem in discrete time with only assignment constraints, i.e. each variant copy is assigned to exactly one position and exactly one variant copy is scheduled in each position.

A recent line of research (Drexl and Kimms 2001, Drexl, Kimms, and Matthiessen 2006) focuses on the *Combined Car Sequencing and Level Scheduling Problem* (CCSLSP). Basically, the CCSLSP is concerned with obtaining a level schedule of cars that also satisfies option constraints. Existing solution approaches for the CCSLSP include time-consuming exact algorithms. The aim of the present paper is to develop a new search algorithm that finds good heuristic solutions in a short time and optimal solutions if enough time is given.

The remainder of the paper is organised as follows. Section 2 presents a brief literature review and defines the problem. Section 3 examines the structural properties of the CCSLSP. Section 4 proposes an iterated beam search algorithm for the problem's solution. Section 5 presents a computational study and discusses the results thereof. Finally, Section 6 concludes the paper by highlighting the results obtained from the study and pointing out some future research directions.

2. Problem definition

2.1 *Literature review*

Let there be V car variants produced on a mixed-model production line. For each variant $v = 1, \ldots, V$, demand (d_v) is known *a priori* for the planning horizon. The production sequence is $T = \sum_{v=1}^{V} d_v$ units long (index t), with d_v copies (index$i$) of variant v, $v = 1, \ldots, V$.

Let O denote the number of options (index o), and $b_{v,o}$ the option o requirement of variant v. In this paper, $b_{v,o} \in \{0, 1\}$ is assumed. While this binary option requirement assumption is very common, the CSP literature also contains examples of non-binary option requirements (Bautista, Pereira, and Adenso-Diaz 2008). The total requirement of option o is $R_o = \sum_{v=1}^{V} b_{v,o} d_v$.

The constraint for option o is denoted by $H_o : N_o$, where no more than H_o cars requiring option o are allowed in any subsequence of N_o cars. Let $w_o(t) = \{\max\{1, t + 1 - N_o\}, \ldots, t\}$ be the set of positions constituting the constraint window for option o ending with position t.

Let $x_{v,i,t} \in \{0, 1\}$ be a binary variable denoting whether or not the ith copy of variant v is assigned to position t of the sequence. Table 1 summarises the mathematical notation used in the formulations and also defines variables related to algorithm development for future reference. The CSP consists of three constraints. First, one, and only one, variant copy must be assigned to each position. Second, each variant copy (v, i) is assigned to one, and only one, position in the sequence. Note that this also ensures that the sequence contains exactly d_v copies of each variant v. Third, option o occurs at most H_o times in each constraint window throughout the sequence. A formal presentation of the constraints is provided by the equations

$$\sum_{v=1}^{V} \sum_{i=1}^{d_v} x_{v,i,t} = 1, \quad \forall t = 1, \ldots, T, \tag{1}$$

$$\sum_{t=1}^{T} x_{v,i,t} = 1, \quad \forall v = 1, \ldots, V, \forall i = 1, \ldots, d_v, \tag{2}$$

Table 1. Notation.

Model building	
V	Number of distinct variants
v	Variant index
d_v	Demand for variant v
$T = \sum_{v=1}^{V} d_v$	Total demand and the length of the production sequence
i	Variant copy index
O	Number of options
o	Option index
$b_{v,o} \in \{0,1\}$	Whether variant v requires option o or not
$R_o = \sum_{v=1}^{V} b_{v,o} d_v$	Total requirement of option o
H_o	Maximum number of option o occurrences in a constraint window
N_o	Constraint window length for option o
$w_o(t) = \{\max\{1, t+1-N_o\}, \ldots, t\}$	Set of positions in the constraint window for option o ending with position t
$x_{v,i,t} \in \{0,1\}$	Binary decision variable denoting whether the ith copy of variant v is assigned to position t
$r_{v,i} = (i-(1/2))T/d_v$	Ideal position of the ith copy of variant v
Analysis and algorithm development	
$v(t)$	The variant assigned to position t
$possible_{v,t}$	Whether a copy of variant v can be assigned to position t or not
$M_{o,t} \in \{-1,0,1\}$	Whether option o must (1), may (0) or must not (–1) occur in position t
$range_{o,j,t}$	Whether the jth occurrence of option o can be in position t or not
$e_{o,j}$	The earliest position for the jth occurrence of option o
$l_{o,j}$	The latest position for the jth occurrence of option o
$\lambda_{v,i,t}$	Reduced cost of assigning the ith copy of variant v to position t
$\lambda_{v,t} = \min\{\lambda_{v,i,t} \mid i = 1, \ldots, d_v\}$	Minimum reduced cost among all copies of variant v in position t
$c_{v,i,t} = \begin{cases} \lvert t - r_{v,i} \rvert, & \text{if } possible_{v,t} = true, \\ \infty, & o/w \end{cases}$	Objective function coefficient for the assignment problem formulation
$\lvert v^{-1}(v,k) \rvert$	Number of copies of variant v scheduled in the first k stages of the sequence
$u(o,t) = b_{v(t),o}$	Whether option o is required by the variant assigned to stage t
Pseudocodes	
RecentUpdates	Information storage for the updates made in variables making up a partial solution
β	Beam width
ϵ	Lower bound difference tolerance
MaxTime	Maximum allowed runtime
Time	Actual runtime from the beginning of the algorithm
Root	Null solution
Parent	Partial solution used in branching
$Child_v$	Partial solution obtained by extending *Parent* via variant v
$List_t$	Partial solution list at level $t = 0, 1, \ldots, T$
$Head(List_t)$	Head of $List_t$, the partial solution in the list with the smallest lower bound value
$LowerBound(\cdot)$	Lower bound of a partial solution
LowestLowerBound	The lowest lower bound of all remaining unexplored partial solutions
UpperBound	Objective function value of the best complete solution obtained during the search process
BestSolution	The best complete solution obtained during the search process

$$\sum_{t' \in w_o(t)} \sum_{v=1}^{V} \sum_{i=1}^{d_v} b_{v,o} x_{v,i,t'} \leq H_o, \quad \forall o = 1, \ldots, O, \forall t = N_o, \ldots, T. \tag{3}$$

The CSP is shown to be NP-Hard in the strong sense (Kis 2004). Along with the computational complexity proof, Kis proposes a dynamic programming (DP) method for the CSP, and notes that it can solve problems with up to 20 positions in practical computation times. Given the complexity of the problem, and the fact that we are unlikely to find a polynomial-time exact algorithm to solve the problem, the goal in this paper is to develop a fast heuristic algorithm.

In addition to the interest from the constraint satisfaction community, the CSP has received increasing interest from operations researchers. A common transformation mechanism adopted by operations researchers is to take option constraints as soft constraints and formulate an objective function of minimising total soft constraint violations (see Gagne, Gravel, and Price (2006); Bautista, Pereira, and Adenso-Diaz (2008) and Fliedner and Boysen (2008) for examples). Operations researchers' interest in the CSP peaked in 2005, when ROADEF (the French Society of Operations Research and Decision Aid) chose the CSP as the subject of its bi-annual challenge. Data was provided by Renault and the problem was further enhanced by incorporating paint shop considerations into the problem. A great majority of the algorithms developed for the challenge were heuristics, ranging from the greedy method to large neighbourhood search, from simulated annealing to ant colony optimisation. The methods developed for the challenge were also published by the *European Journal of Operational Research* in 2008 as a feature cluster. See Solnon et al. (2008) and references therein for a variety of solution approaches.

The focus of this paper is on the combined optimisation problem arising at the intersection of car manufacturing and JIT production. Production schedules necessary to realise the one-piece-flow goal of JIT are often called smooth, or level schedules. Production smoothing, or level scheduling, has been studied in different contexts with different objective functions.

Production smoothing at Toyota focuses on reducing the variability of the consumption rates of the subassemblies used at the final stage (Monden 1983). Miltenburg and Sinnamon (1989) extend the Toyota approach by considering smoothing both end product production rates and subassembly consumption rates at the preceding stages of the manufacturing system. Miltenburg (1989) is concerned with reducing the variability of the production rate for end products at the final stage only. Kubiak (1993) refers to the former approach as the output rate variation (ORV) and the latter as the product rate variation (PRV), and shows that the ORV is NP-Hard. The PRV, on the other hand, is known to be polynomially solvable through a transformation into an assignment problem (Kubiak and Sethi 1991).

In the majority of studies addressing the level scheduling problem, the objective function is chosen as the minimisation of the sum of squared deviations between actual and ideal cumulative production amounts in each stage of the production sequence. An alternative approach is due to Inman and Bulfin (1991), who define the ideal position for each copy of each end product (variant) by $r_{v,i} = (i - (1/2))T/d_v$. Note that, if we divide the production sequence into d_v equal parts, the ideal position of the ith copy is the midpoint of the ith part.

The PRV with the objective function of minimising the sum squared or absolute deviations of actual positions from the ideal is then trivial, as the earliest due-date rule yields the optimal solution. See Yavuz and Akcali (2007) for more on the practical and modelling aspects of level scheduling as well as a survey of the state-of-the-art solution algorithms.

Drexl and Kimms (2001) propose a formulation for the CCSLSP. In their formulation the level scheduling objective is the minimisation of the sum of the squared deviations between the actual and ideal positions of variant copies. The ideal position of a variant copy is based on Inman and Bulfin's (1991) approach mentioned above. The authors also propose a set partitioning, column generation approach to solve the emerging problem.

Later, Drexl, Kimms, and Matthiessen (2006) studied the CCSLSP with an objective function of minimising the sum of absolute deviations between the actual and ideal positions of variant copies. The authors develop a branch-and-bound approach for the exact optimal solution of the problem. On a 36-instance test bed the authors report that the branch-and-bound approach is able to solve 34 of the test instances within three hours of computation time. The present paper uses the same problem formulation, aims to develop a novel algorithm to solve the problem and evaluates the performance of the algorithm on an expanded version of the same test bed.

2.2 *A mathematical formulation of the CCSLSP*

$$\text{CCSLSP}: \qquad \text{Minimise} \quad \sum_{v=1}^{V} \sum_{i=1}^{d_v} \sum_{t=1}^{T} |t - r_{v,i}| x_{v,i,t}, \tag{4}$$

subject to

$$\sum_{v=1}^{V} \sum_{i=1}^{d_v} x_{v,i,t} = 1, \quad \forall t = 1, \ldots, T, \tag{5}$$

$$\sum_{t=1}^{T} x_{v,i,t} = 1, \quad \forall v = 1, \ldots, V, \forall i = 1, \ldots, d_v, \tag{6}$$

$$\sum_{t' \in w_o(t)} \sum_{v=1}^{V} \sum_{i=1}^{d_v} b_{v,o} x_{v,i,t'} \leq H_o, \quad \forall o = 1, \ldots, O, \forall t = N_o, \ldots, T, \tag{7}$$

$$x_{v,i,t} \in \{0, 1\}, \quad \forall v = 1, \ldots, V, \forall i = 1, \ldots, d_v, \forall t = 1, \ldots, T. \tag{8}$$

Objective function (4) is the sum of the absolute differences between the actual and ideal positions of variant copies. Constraints (5) and (6) are regular assignment constraints ensuring that exactly one variant copy is assigned to each position, and each variant copy is assigned to exactly one position, respectively. Constraint (7) guarantees that no more than H_o cars requiring an option o appear in any subsequence of N_o cars. Finally, (8) defines the decision variables $x_{v,i,t}$ as binary integers (0–1).

2.3 *Research goals*

The primary research goal of this paper is to develop an intelligent search algorithm that runs fast and finds very good solutions. The computational complexity of the problem renders exact optimisation algorithms impractical for large size instances, and, hence, motivates the development of a heuristic algorithm. However, traditional heuristic algorithms have major shortcomings that draw criticism from the operations research community and prompt researchers to create hybrid methods combining the strengths of exact and heuristic methods (Ball 2011).

One of the biggest weaknesses of heuristics is that they cannot report how good their solution is. More specifically, a heuristic algorithm may find the optimal solution and keep searching for a better solution without being able to prove optimality, possibly coming back to the same best-known solution over and over again. The algorithm developed in this paper is designed to systematically search the solution space to return a lower and an upper bound, and, hence, the optimality gap. Other desired properties of the algorithm are listed below.

- Continually search to improve the upper bound.
- Keep track of a lower bound and continually improve it.
- If enough time is given, find an optimal solution and prove its optimality.
- If the computation time given is too short, report the optimality gap at termination.
- Avoid creating the same (partial) solution repeatedly.
- Do not get trapped at local optima.

The algorithm chosen in this paper to achieve these goals is *Iterated Beam Search* (IBS), a modification of the Beam Search (BS) heuristic. As is the case with any algorithm, IBS benefits from exploiting the structural properties of the problem. In the following section, such structural properties are analysed and a partial solution specification mechanism is developed to narrow the search space.

The secondary research goal is to explore the effectiveness of using a dynamic programming formulation as a filtering mechanism in a heuristic algorithm. As will be discussed later in this paper, beam search is a tree-search method that spans a wide spectrum from the greedy method to an exhaustive breadth-first search. A successful incorporation of dynamic programming into IBS can yield valuable insights for further development of the generic beam-search method as well as the integration of exact and heuristic optimisation methods.

3. Structural properties of the CCSLSP

Recall that the problem is to find a sequence of $T = \sum_v d_v$ copies of V distinct variants. Let $v(t)$ be the variant scheduled in the tth stage (i.e. position) of the sequence for $t = 1, 2, \ldots, T$. Note that $v(t)$ can be directly obtained from decision variables $x_{v,i,t}$ defined in the CCSLSP model above ($v(t) = \sum_v \sum_i \sum_t v x_{v,i,t}$).

Let $possible_{v,t}$ be a Boolean variable denoting whether or not variant v can be scheduled in an empty stage t. For a scheduled stage t, since the model scheduled in that stage is already known ($v(t)$) it is not necessary to denote the possibility of assigning a copy of variant $v(t)$ to stage t. The number of stages t with $possible_{v,t} = true$ tells us how much

flexibility we have in distributing the unassigned copies of variant v. If there are just enough stages where the remaining copies of v are possible, then they must be assigned to those stages, eliminating part of the decision problem. Similarly, if the number of possible stages is smaller than the number of remaining copies, then no feasible assignment can be made.

The mapping of option requirements to stages is ensured by $M_{o,t} \in \{-1, 0, 1\}$. $M_{o,t} = 1$ whenever a variant requiring o is scheduled in t, -1 whenever a variant not requiring o is scheduled in t, and 0 whenever no determination is made. $M_{o,t}$ can be set to 1 or –1 without a variant being scheduled in t, depending on the rest of the sequence. In that case, $M_{o,t} = 1$ ($M_{o,t} = -1$) means a copy of a variant v with $b_{v,o} = 1$ ($b_{v,o} = 0$) must be scheduled in t.

Let $range_{o,j,t}$ be a Boolean variable denoting whether or not the jth occurrence of o can be in t. Finally, let $e_{o,j}$ ($l_{o,j}$) denote the earliest (latest) stage for the jth occurrence of o, i.e. whether or not t is in the range for the jth occurrence of option o.

One starts solving the problem with a null solution, i.e. a blank sequence. As variants are assigned in stages a solution is gradually specified. From a fully specified solution an objective function value is calculated and the upper bound may need to be updated. On a partially specified solution a lower bound can be obtained and compared with the upper bound. Partial solutions with a lower bound that exceeds the upper bound can be eliminated. Also, specification of a partial solution can reveal if it cannot lead to any feasible solutions. Again, in that case, a partial solution can be eliminated early in the process. Therefore, the specification of partial solutions is critical to the success of the search process.

Drexl et al.'s specification routine takes advantage of the following structural properties of the CCSLSP.

- If there is one and only one t with $range_{o,j,t} = true$ for the jth occurrence of o, $M_{o,t}$ must be 1.
- If there is no j with $range_{o,j,t} = true$ for an (o, t) pair, $M_{o,t}$ must be –1.
- If $possible_{v,t} = true$ for one and only one variant v in stage t, v must be scheduled in t.
- If all of the variants with $possible_{v,t} = true$ in a stage t require an option o, $M_{o,t}$ must be set to 1.
- If none of the variants with $possible_{v,t} = true$ in a stage t require an option o, $M_{o,t}$ must be set to –1.
- If the number of unassigned copies of variant v equals the number of blank stages t in which $possible_{v,t} = true$, unassigned copies of v must be assigned to those stages in which it is possible

The following subsection describes a new specification routine that uses these and some newly developed structural properties.

3.1 *Partial solution specification*

A partial solution consists of the decision variables summarised in the second part of Table 1. $v(t)$ is a T vector containing the schedule information. $possible_{v,t}$ is a $V \times T$ matrix showing the possibility of assigning each variant to each position in the sequence. Similarly, $M_{o,t}$ is a $O \times T$ matrix showing the possibility of each option occurring in each position in the sequence. $range_{o,j,t}$ is a $\Theta \times T$ matrix (where $\Theta = \sum_o Ro$) detailing which occurrence(s) of which option(s) may take place in each position in the sequence. Finally, $e_{o,j}$ and $l_{o,j}$ are Θ vectors showing the earliest and latest occurrences of each option occurrence, respectively. A partial solution is a collection of all these variables.

Option constraints can partially or even fully define a sequence. That is, they may specify where option occurrences *must*, *may* or *must not* take place as well as where variant copies *may* or *must not* be assigned. Consider the instance given in Table 2. The problem is to find a sequence of 10 copies of four variants while meeting three option constraints.

Table 2. Example problem 1.

	Variant					
Option	1	2	3	4	$H_o : N_o$ constraint	Total requirement
1	•	•			1 : 2	3
2		•			1 : 5	2
3	•		•		2 : 8	4
Demand	1	2	3	4		

Option 1 is required by variants 1 and 2, has a total requirement of three and its constraint is 1 : 2 Three occurrences of a 1 : 2 option could be packed into a sequence of as few as five variants. Thus, in a sequence of 10, this constraint contains a lot of flexibility.

The second option is required by variant 2 only, has a total requirement of two and its constraint is 1 : 5. This information tells us that the first occurrence of option 2 must take place somewhere in the first five positions and the second in the final five.

The third and final option is required by variants 1 and 3, has a total requirement of four and its constraint is 2 : 8. This option constraint is much stricter than the first two. In a sequence of 10 variants, the first two occurrences of option 3 must be in the first two stages and the last two in the final two. This is a full specification of the sequence for this option. That is, the first two stages must contain it, the next six must not and the final two must. This option specification in fact triggers other option and variant specifications.

Variants 2 and 4 are not possible in stages 1, 2, 9 and 10. Since variant 2 is the only one requiring option 2, option 2 must not occur in stages 1, 2, 9 and 10. The earliest position for the first occurrence of option 2 is stage 3, and the latest position for its second occurrence is stage 8. Since the option 2 constraint is 1 : 5, the two occurrences must occur in stages 3 and 8, and thus the two copies of variant 2 must be assigned to stages 3 and 8. No other variant is possible in stages 3 and 8, which further narrows the possible stages for variant 4 to stages 4, 5, 6 and 7. Since there are four copies of variant 4 and only four stages where they are possible, they must be assigned to those stages.

Variant 2 also requires option 1. Since the option has a 1 : 2 constraint, it must not occur in stages next to where variant 2 is scheduled. This mandates that option 1 must not occur in stages 2, 4, 7 and 9, and as the other variant requiring option 1, variant 1 is not possible in those stages. This leaves variant 3 as the only one assignable to stages 2 and 9. In stages 1 and 10, either variant 1 or variant 3 may be scheduled. In terms of its constituting variables, this partial solution is represented by $v(t) = (\cdot, 3, 2, 4, 4, 4, 4, 2, 3, \cdot)$, a *possible* matrix containing *true* for variants 1 and 3 in positions 1 and 10 and *false* everywhere else, an M matrix containing a 0 for option 1 in positions 1 and 10 and either a 1 or -1 elsewhere, depending on whether the variant assigned to that position requires that option, and so on. Even with this small example, we can readily see how much information is stored in a partial solution and omit discussion of the *range*, e and l variables for sake of simplicity.

A partial solution specification yields the following two feasible sequences for the instance in the example: 1–3–2–4–4–4–4–2–3–3 and 3–3–2–4–4–4–4–2–3–1. Considering the 10!/(4!3!2!) = 12,600 possible permutations for this instance, the solution specification does a great job. Also note that the two sequences the specification routine yields are the mirror image of one another.

The discussion presented above is an example of solution specification working on a null solution, not triggered by a change in a decision variable's value. In practice, many instances will not result in full specification as in the above example. In the following subsection, several rules are developed in addition to Drexl et al.'s rules in order to advance the specification of a partial (or null) solution. Specification of a partial solution is triggered by the assignment of a variant to the first empty stage in the sequence. The rules are given in the order they are invoked in solution specification following the assignment of a variant copy to an empty stage.

3.1.1 *From $v(t)$ to $possible_{v,t}$*

- (*Rule #1*) When $v(t)$ is updated from 0 to a $v \in \{1, \ldots, V\}$, $possible_{v,t}$ is set to *false* for all v for the given t. That is, all models are made impossible in the assigned stage.

3.1.2 *From $possible_{v,t}$ to $M_{o,t}$*

- (*Rule #2*) In a stage t for an option o, if for all $v \,|\, possible_{v,t} = true$, $b_{v,o} = 1$, then $M_{o,t}$ must be set to 1.
- (*Rule #3*) In a stage t for an option o, if for all $v|\, possible_{v,t} = true$, $b_{v,o} = 0$, then $M_{o,t}$ must be set to -1.

These two rules jointly ensure that M is updated to reflect the option requirement of the remaining possible variants in a position.

3.1.3 *From $M_{o,t}$ to $range_{o,j,t}$*

- (*Rule #4*) If $M_{o,t} = 1$ for some o and t and $range_{o,j,t} = true$ for one and only one $j \in \{1, \ldots, R_o\}$, then $range_{o,j,t'}$ must be set to *false* for all $t' \neq t$. With this rule, the occurrence number of a known option occurrence is specified.
- (*Rule #5*) If $M_{o,t} = -1$ for some o and t, then $range_{o,j,t}$ must be set to *false* for all $j \in \{1, \ldots, R_o\}$.

3.1.4 *From $range_{o,j,t}$ to $e_{o,j}$ and $l_{o,j}$*

- (*Rule #6*) If $range_{o,j,e_{o,j}}$ is set to *false*, $e_{o,j}$ must be restored so that $e_{o,j} = \min\{t \mid range_{o,j,t} = true\}$.
- (*Rule #7*) If $range_{o,j,l_{o,j}}$ is set to *false*, $l_{o,j}$ must be restored so that $l_{o,j} = \max\{t \mid range_{o,j,t} = true\}$.

These two rules guarantee that the earliest and latest positions of option occurrences are updated to reflect changes in ranges.

3.1.5 *Consistency of $e_{o,j}$ and $l_{o,j}$*

Every time an $e_{o,j}$ or $l_{o,j}$ is updated there is an opportunity to update other earliest or latest position variables in a domino effect. That is, if $e_{o,j}$ is updated to t, assigning the jth through $j + H_o$th occurrences of option j to their earliest positions may violate the option constraint $(H_o : N_o)$. Thus, to prevent such constraint violations, $e_{o,j'}$, $j' > j$, would need to be updated. Similarly, if $l_{o,j}$ is updated to t, $l_{o,j'}$, $j' < j$, may need to be updated.

- (*Rule #8*) For an option occurrence $(o,j) \mid j \leq R_o - 1$, if $e_{o,j} > e_{o,j+1} - 1$, then $e_{o,j+1}$ must be updated to $\min\{t \mid range_{o,j,t} = true \text{ and } t \geq e_{o,j} + 1\}$.
- (*Rule #9*) For an option occurrence $(o,j) \mid j \leq R_o - H_o$, if $e_{o,j} > e_{o,j+H_o} - N_o$, then $e_{o,j+H_o}$ must be updated to $\min\{t \mid range_{o,j,t} = true \text{ and } t \geq e_{o,j} + N_o\}$.
- (*Rule #10*) For an option occurrence $(o,j) \mid j > 1$, if $l_{o,j} < l_{o,j-1} + 1$, then $l_{o,j-1}$ must be updated to $\max\{t \mid range_{o,j,t} = true \text{ and } t \leq l_{o,j} - 1\}$.
- (*Rule #11*) For an option occurrence $(o,j) \mid j > H_o$, if $l_{o,j} < l_{o,j-H_o} + N_o$, then $l_{o,j-H_o}$ must be updated to $\max\{t \mid range_{o,j,t} = true \text{ and } t \leq e_{o,j} - N_o\}$.

3.1.6 *From $e_{o,j}$ and $l_{o,j}$ to $range_{o,j,t}$*

- (*Rule #12*) When $e_{o,j}$ is updated, $range_{o,j,t}$ must be set to *false* for all $t < e_{o,j}$.
- (*Rule #13*) When $l_{o,j}$ is updated, $range_{o,j,t}$ must be set to *false* for all $t > l_{o,j}$.
- (*Rule #14*) Following directly from the two rules above, if $e_{o,j} = l_{o,j}$, then $range_{o,j,t} = true$ for $t = e_{o,j} = l_{o,j}$ and *false* for all other t.

3.1.7 *From $range_{o,j,t}$ to $M_{o,t}$*

- (*Rule #15*) If, for some o and j, $range_{o,j,t} = true$ for one and only one t, then $M_{o,t}$ must be set to 1.
- (*Rule #16*) If, for an option o in a stage t, $range_{o,j,t} = false$ for all $j \in \{1, \ldots, R_o\}$, then $M_{o,t}$ must be set to -1.

3.1.8 *From $M_{o,t}$ to $possible_{v,t}$*

- (Rule #17) If $M_{o,t}$ is set to 1 for an option o in a stage t, then $possible_{v,t}$ must be set to *false* for all v with $b_{v,o} = 0$.
- (Rule #18) If $M_{o,t}$ is set to -1 for an option o in a stage t, then $possible_{v,t}$ must be set to *false* for all v with $b_{v,o} = 1$.

3.1.9 *From* $possible_{v,t}$ *to* $v(t)$

- (*Rule #19*) If in an empty stage t there is one and only one v with $possible_{v,t} = true$, $v(t)$ must be updated. That is, the only possible variant must be assigned to the stage.
- (*Rule #20*) If, for a variant v, the number of stages t in which $possible_{v,t} = true$ equals the number of unassigned copies of v, then the remaining copies of the variant must be assigned to the possible stages, i.e. $v(t)$ must be set to v in all $t \,|\, possible_{v,t} = true$.

Following the rules given above, the specification routine can be organised as a linear sequence of variable updates as shown in Figure 1. Furthermore, the rules do not have to be checked for all v, o, j, and t every time. Information on the updates performed recently can be stored and used to check for further updates. Let us define a new variable, *RecentUpdates*, as storage of this information. Details of this information storage are omitted in this paper for sake of keeping the paper short.

Figure 2 presents a pseudo-code for the partial solution specification routine. The routine depends on *RecentUpdates* as it takes it both as input variable and revises it throughout execution. The solution specification is not completed as long as updates are made. Steps 1–9 and 13 represent the rules presented in Sections 3.1.1 through 3.1.9. Steps 10–12 relate to the lower bounding procedure explained in the next subsection.

In steps 1–9, 12 and 13, certain rules are checked based on *RecentUpdates*. Each recent update is deleted after being examined and new updates may be made, in which case they are entered into *RecentUpdates*. In this way, the specifica-

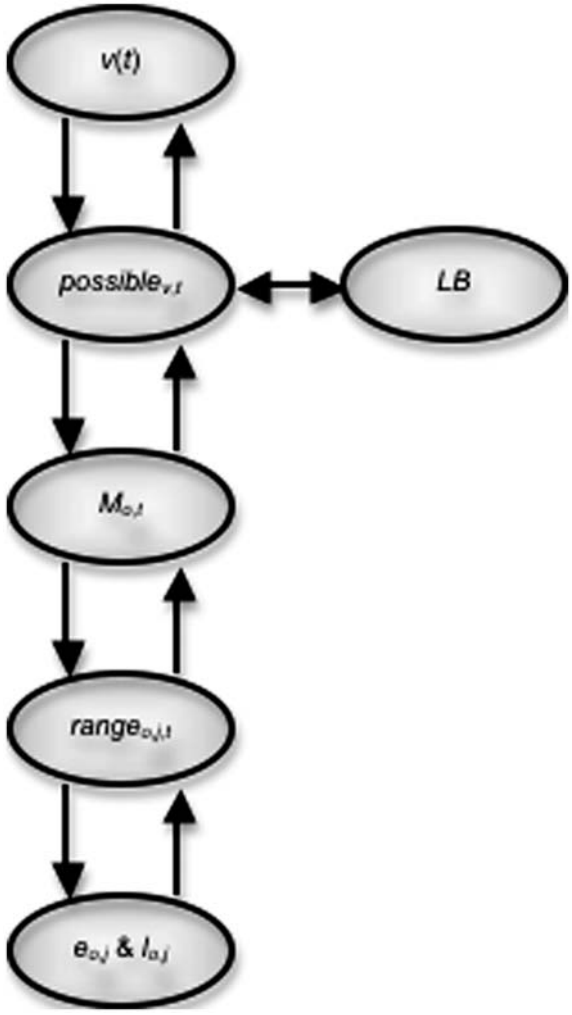

Figure 1. Specification routine.

Partial Solution Specification(*RecentUpdates*)

1. While $|RecentUpdates| > 0$
2. Check rule #1 and update $possible_{v,t}$ as necessary.
3. Check rules #2-3 and update $M_{o,t}$ as necessary.
4. Check rules #4-5 and update $range_{o,j,t}$ as necessary.
5. Check rules #6-7 and update $e_{o,j}$ and $l_{o,j}$ as necessary.
6. Check rules #8-11 and update $e_{o,j}$ and $l_{o,j}$ as necessary.
7. Check rules #12-14 and update $range_{o,j,t}$ as necessary.
8. Check rules #15-16 and update $M_{o,t}$ as necessary.
9. Check rules #17-18 and update $possible_{v,t}$ as necessary.
10. If no $possible_{v,t}$ update was made in Step 9, then
    11. Build the assignment problem formulation and solve it to obtain a lower bound and $\lambda_{v,t}$
    12. Check rule #21 and update $possible_{v,t}$ as necessary.

    End if
13. Check rules #19-20 and update $v(t)$ as necessary.

End while

End.

Figure 2. Pseudo-code for the specification routine.

tion routine works in a linear fashion going back and forth between different variables. Using an analogy, it can be described as an elevator going up and down between levels of decision variables, the sequence variable $v(t)$ representing both the entry and exit level.

Variable updates do not always flow smoothly. It is possible and frequently observed that an assignment may cause a conflict, which is a sign of infeasibility. In each of the steps (1–9, 12 and 13) an infeasibility may be detected. In Step 11, it may be observed that the lower bound of a partial solution exceeds the upper bound of the problem, thereby proving the partial solution suboptimal. In either case the specification routine is terminated and the infeasibility or suboptimality conclusion is returned. Since the specification routine is typically invoked after making a variant assignment to an empty stage of the sequence, infeasibility or suboptimality proves that variant assignment must not be made, therefore no complete solution that can be obtained from this partial solution needs to be considered. The more successful the specification routine, the more partial solutions are eliminated in the search process and the faster the search runs.

3.2 *Assignment problem formulation for improved lower bounds*

Since the objective function is the minimisation of the sum of the absolute differences between the actual and ideal position of each variant copy, a sequence minimising this objective can easily be obtained by ordering variant copies in non-decreasing order of their ideal positions. This can be applied to a null or partial solution to obtain a lower bound for the remaining part of the sequence. The worst case complexity of finding a lower bound this way is $O(VT)$. However, this approach fails to take advantage of all the effort that goes into specifying a solution. Most importantly, since the $possible_{v,t}$ variables are not taken into account in populating the sequence, the result is most likely an infeasible solution.

The cost of assigning each copy of each variant to each stage in the sequence can be calculated in $O(T^2)$ time in a pre-processing step and stored in a $T \times T$ matrix. Then, whenever a $possible_{v,t}$ variable is set to *false*, the related assignment costs can be set to infinity. The resulting assignment problem formulation is as follows:

$$\text{AP}: \qquad \text{Minimise} \quad \sum_{v=1}^{V} \sum_{i=1}^{d_v} \sum_{t=1}^{T} c_{v,i,t} x_{v,i,t}, \tag{9}$$

subject to

$$\sum_{v=1}^{V} \sum_{i=1}^{d_v} x_{v,i,t} = 1, \quad \forall t = 1, \ldots, T, \tag{10}$$

$$\sum_{t=1}^{T} x_{v,i,t} = 1, \quad \forall v = 1, \ldots, V, \forall i = 1, \ldots, d_v, \tag{11}$$

$$x_{v,i,t} \in \{0, 1\}, \quad \forall v = 1, \ldots, V, \forall i = 1, \ldots, d_v, \forall t = 1, \ldots, T. \tag{12}$$

Here, the objective function coefficients are

$$c_{v,i,t} = \begin{cases} |t - r_{v,i}|, & \text{if } possible_{v,t} = true, \\ \infty, & \text{o/w.} \end{cases}$$

In the specification routine, after there are no more updates to be made and the routine is ready to terminate, an assignment problem of at most T variant copies to T stages can be solved in $O(T^3)$ time using one of the well-known solution algorithms from the literature. In this paper, the Hungarian Algorithm (Kuhn 1955) is used.

The assignment problem approach not only yields an improved lower bound, but it is also more likely to yield feasible sequences since it avoids the known infeasible assignments. Furthermore, when the assignment problem is solved, one obtains reduced costs. Let the reduced cost of assigning a particular variant copy (v, i) to a particular stage t be $\lambda_{v,i,t}$, which tells us the additional cost of insisting on assigning that variant copy to that stage. Also, let $\lambda_{v,t} = \min\{\lambda_{v,i,t} \mid i = 1, \ldots, d_v\}$ be the minimum reduced cost of any copy of variant v in stage t, and $\Delta = UpperBound - LowerBound$ be the gap between the lower bound of the current partial solution and the upper bound.

If $\lambda_{v,t} \geq \Delta$ for some variant v in some empty stage t, then $possible_{v,t}$ can be set to *false*. This gives rise to the following specification rule, which is used in addition to the rules given above, so that the specification routine does not terminate immediately, but performs at least one more round of variable updating (Figure 1).

3.2.1 *From $\lambda_{v,t}$ to $possible_{v,t}$*

- (*Rule #21*) If $\lambda_{v,t} \geq \Delta$ for some v and t, $possible_{v,t}$ must be set to *false*.

It also follows directly from this rule that eliminating all but one of the possible variants in a stage is the same as assigning the only variant (with $\lambda_{v,t} < \Delta$) to the position.

3.3 *A dynamic programming approach*

Dynamic programming (DP) is known as an implicit enumeration method exploiting the problem structure to reduce the size of the search tree. Consider a partial solution of length k where $v(t)$, $t = 1, 2, \ldots, k$, are scheduled. In order to extend this partial solution, the objective function value of the partial sequence and actual cumulative production quantities of the variants must be known. Plus, the option requirements of the last $N_o - 1$ stages must be known for each option o.

Table 3. Example problem 2.

Option	Variant 1	2	3	4	$H_o : N_o$ constraint	Total requirement
1	•	•			2 : 3	23
2	•		•		1 : 2	22
Demand	10	12	13	15		

As an example, consider an instance of the problem with four variants and two options, summarised in Table 3. Option 1 is required by variants 1 and 2 only and option 2 by variants 1 and 3 only. The problem is to find a sequence of 50 copies of four variants.

Consider the following two partial sequences with the first four stages scheduled with two copies of variant 2, one of 3 and one of 4: 2–4–2–3 and 4–2–2–3. These two partial solutions are identical in terms of building on them. When one tries to complete either sequence, (s)he will have the exact same number of variant copies to schedule and be subject to the same exact constraints.

The goal in developing a DP structure is to identify partial solutions that are identical early in the process and eliminate all but one of them to minimise the computational effort needed to evaluate them and the subtrees starting with them.

The IBS method developed in this paper is based on a DP network of T stages with (possibly) multiple states in each stage and arcs pointing from stage $k-1$ to k only, for $k = 1, 2, \ldots, T$. A state at the T th stage is a leaf node, i.e. a complete solution. A state is denoted by the following:

$$(k, |v^{-1}(1,k)|, \ldots, |v^{-1}(V,k)|, u(1, k-N_1+2), \ldots, u(1,k), u(2, k-N_2+2), \ldots, u(O,k)). \quad (13)$$

Here, k is the stage index, $|v^{-1}(v,k)|$ is the number of copies of variant v scheduled in the first k stages of the sequence, and $u(o,t) = b_{v(t),o}$ is a binary function representing whether or not option o is required by the variant assigned to stage t. Note that, for each option o, only the last $N_o - 1$ stages' requirements are recorded. In the example given above, both partial solutions 2–4–2–3 and 4–2–2–3 would be represented by the DP state $(4, 0, 2, 1, 1, 1, 0, 1)$.

The DP structure formulated here is fundamentally different from that proposed by Kis (2004) in that this formulation is based on option requirements in the last $N_o - 1$ stages for each option, whereas Kis's formulation is based on the actual variants assigned to the last $\max\{N_1, \ldots, N_O\}$ stages. The formulation proposed here is tighter and can be used (as a stand-alone exact method) to solve the CSP more efficiently. However, as that is not the goal of this paper, the DP structure will be used to eliminate some of the emerging partial solutions only.

4. Iterated beam search algorithm

In this section, first a brief introduction to beam search is made and then a novel iterated beam search (IBS) algorithm is developed to solve the CCSLSP problem defined in Section 2 and discussed in Section 3.

4.1 *Beam search*

Beam search (BS) is an enhanced constructive heuristic method. It starts with a null solution and constructs complete solutions by fixing one variable at a time. At each stage, partial solutions are extended using a branching strategy. All newly created partial solutions are evaluated and only the best β (i.e. beam width) are kept for further branching. In other words, the search tree is pruned at each stage in order to keep it at a manageable size. An optimal solution may not be reached if all partial solutions leading to it are pruned at intermediate stages, which makes BS a heuristic method.

The strengths of the BS method are its speed and intelligence which can be incorporated into the evaluation of partial solutions. The fastest implementation of BS can be obtained by letting $\beta = 1$ and evaluating partial solutions based on local information only. This is the well-known greedy method. BS is a broad family of methods ranging from an extremely fast greedy heuristic to an enumerative exact method, characterised by the beam width and the partial solution evaluation strategy.

BS has been implemented on a variety of problems. Ow and Morton (1988) implement BS on single machine and flow shop scheduling problems. They also propose a filtered beam search (FBS) method. FBS uses two evaluation strategies. First, a simple evaluation strategy is used on all partial solutions, the best β (i.e. filter width) partial solutions are kept and others are pruned. Then, the remaining β partial solutions are evaluated via a more complicated and accurate strategy. Another common implementation of BS is recovering beam search (RBS). RBS uses local search on partial solutions in order to recover from previously made poor choices. The interested reader is referred to Ghirardi and Potts (2005); Valente and Alves (2005); Esteve et al. (2006); Shi-Jin, Bing-Hai, and Li-Feng (2008); Kang and Choi (2010) and references therein for examples of recent beam search implementations.

Bautista, Pereira, and Adenso-Diaz (2008) propose a beam search solution to the optimisation version of the CSP. In their formulation an option constraint of the CSP is referred to as upper over-assignment of that option. Measured as the positive difference between the number of times an option o appears and the maximum number of times it may appear in a subsequence of N_o cars, upper over-assignment measures the number of times the corresponding option constraint in the CSP is violated. The authors also define upper under-assignment of an option as the positive difference between the maximum number of times that option o may appear and the number of times it actually appears in a subsequence. Upper under-assignment is defined by the authors to promote repetition of option usage subsequences (patterns) throughout the sequence. To give upper over-assignment priority over upper under-assignment, the authors combine them in the objective function with weights 10^6 and 10^3, respectively. An implementation of the basic beam search method is reported to be effective on the CSP instances in the CSPLib (Gent and Walsh 1999).

A few papers in the literature implement BS on different versions of the level scheduling problem. Leu, Huang, and Russell (1998) discuss two BS implementations. The authors note that when the node evaluation strategy involves two consecutive stages, the computational time requirement of BS significantly increases, and, hence, omit that approach from further consideration. McMullen and Tarasewich (2005) consider two objective functions. The first is the product rate variation objective, whereas the second is the number of changeovers. The authors propose a BS method that starts pruning after a designated number of stages. Erel, Gocgun, and Sabuncuoglu (2007) propose six distinct FBS implementations on two separate versions of the production smoothing problem. First, they consider the output rate variation objective for subassemblies. Second, the station workload variation objective for the final assembly level is adopted. The authors enhance their FBS implementation in three directions, namely the use of independent vs. dependent beams, backtracking, and exchange of information (EOI). The results from their study show that FBS with independent beams, backtracking and EOI, and FBS with dependent beams and EOI, are the best of the tested methods. Both these methods significantly outperform the twostage heuristic (2SH). Sabuncuoglu, Gocgun, and Erel (2008) further investigate the use of dependent beams, backtracking and EOI improvements.

4.2 *Iterated beam search*

Existing BS implementations mentioned above are all single-pass methods. That is, partial solutions created at a stage of the tree are either further evaluated immediately, or discarded. With a single pass, some good partial solutions may be truncated due to poor performance of the evaluation strategy employed. Yavuz (2010) proposes an iterated beam search algorithm to overcome this deficiency via multiple passes (iterations) and implements it on the level scheduling problem with both the usage and workload goals at multiple levels. The author adopts the simple heuristics, namely the earliest due date, the one-stage heuristic and the two-stage heuristic, developed for special cases of the LSP, to this generalised version and makes a comparison with his IBS algorithm. Using a computational study, IBS is shown to outperform the simple heuristics and have a comparable computation time requirement. In the present paper, an iterated beam search algorithm is developed with the same goal in mind, however the presence of the CSP's option constraints makes it more difficult in implementation.

At the core of BS is a tree, a graphical representation of which is given in Figure 3. An empty sequence is the root of the tree, and each position in the sequence adds a level to the tree, therefore the search tree consists of T levels or stages, with leaves (complete solutions) at the final (Tth) stage. In the CCSLSP, there are V different variants, and, hence, at most V branches emanating from any non-leaf node in the tree. The number of leaf nodes is $T!/\prod_v d_v!$. The large number of leaf nodes, i.e. the number of potentially feasible solutions, renders explicit enumeration impractical for larger-sized instances of the problem.

The IBS algorithm proposed here incorporates some key mechanisms to speed up the search process. Through the *possible* variable presented in Section 3, at each branching point a majority of the potential branches may be eliminated depending on how many of the V variants are possible in that stage. Also, the DP structure explained in Section 3.3 helps eliminate some of the leaf nodes in advance.

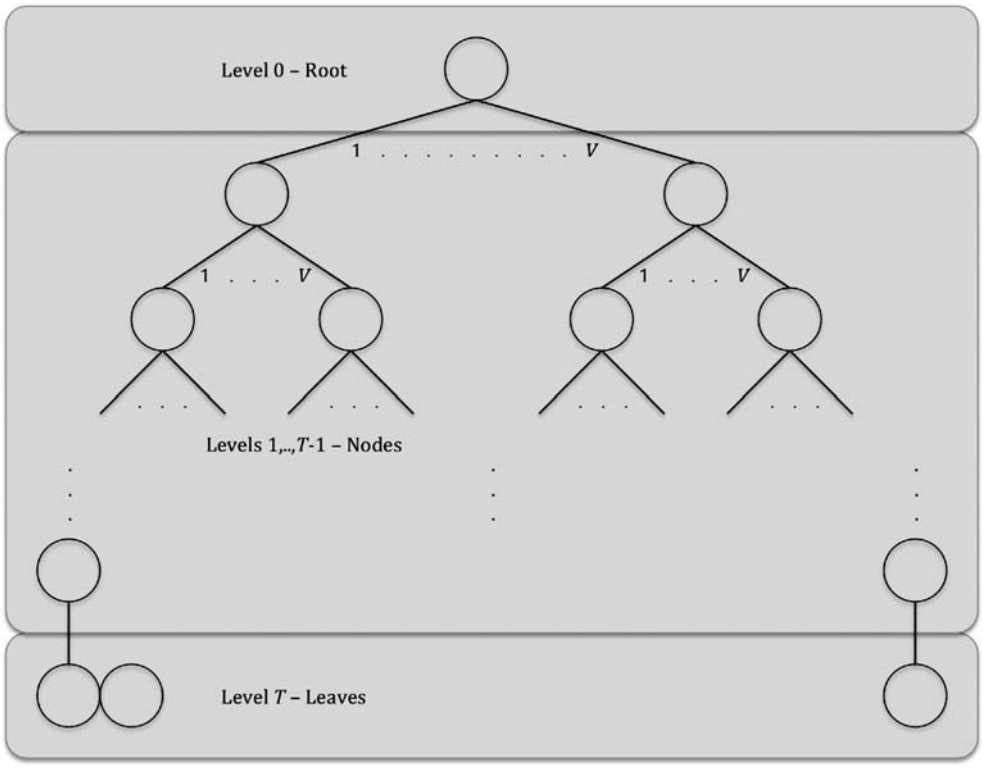

Figure 3. Graphical representation of the search tree.

In the single-pass application of the traditional beam search heuristic, one does not have to store evaluated nodes for future reference. Only the ones kept at hand are stored until the end of the pass. To implement beam search as a multiple-pass method, a linked list of nodes is stored at each stage.

The list of each stage holds partial solutions completed up to and including that stage, that is a partial solution stored in the list of the kth stage has a variant assigned to each of the first k stages of the sequence. It may also happen that some future stages $t > k + 1$ are also scheduled as a result of the specification routine. Stage $k + 1$ must be empty, otherwise the partial solution cannot be inserted into the list of the kth stage. In practice, the $v(t)$ variable is checked to determine which stage that partial solution belongs to before attempting to insert it somewhere.

The lists are ordered in non-decreasing order of the solutions' lower bound values. In this way, at any point in time, if one needs to check the lowest lower bound of the solutions stored in a list, checking the lower bound of the head node in the list is sufficient. Furthermore, when nodes are evaluated, they are taken from the head of the list to make sure nodes with smaller lower bounds (they are also more promising in terms of leading to the optimal solution) are evaluated first.

When a new node is created, it is first specified and when it cannot be further specified and a lower bound value is obtained it is considered for insertion into the list in the appropriate stage. The candidate node is compared with the nodes in the list. If there is no solution identical to the candidate (in regards to the DP structure) it is inserted into the list according to its lower bound value. Otherwise, the better (the one that has the smaller lower bound value) of the two identical solutions is kept in the list in the appropriate position and the other is simply eliminated.

Nodes are explored in forward passes. When a node is explored, all the branches emanating from that node are considered, i.e. at most V nodes at the next stage can be created. In order to prevent the lists from growing too fast and too big, only the variants with zero reduced cost in the next stage are used for branching. The parent node is then modified, $possible_{v,t}$ for variants v used for branching are set to *false* and the solution is specified again. If it is not eliminated by the specification routine (for being infeasible or having a lower bound greater than the upper bound), it is called a stem. Then the stem is inserted back into the list. When a stem is inserted back into the same level's list, it is not subject to the DP structure check. Because of the eliminated possibilities stored in the stems, elimination of a stem could potentially eliminate reaching an optimal solution. On the other hand, revising the DP structure to store the extra information would increase the computation time requirement and reduce the efficiency of the DP structure.

A forward pass starts at the earliest stage with a non-empty list of nodes, and explores β nodes at each stage. The nodes to explore are taken from the front of the list, i.e. are chosen according to their lower bound values. From each parent, all necessary children are created and specified. If the assignment problem solution that gives the lower bound is feasible, the upper bound is updated immediately. When the upper bound is updated, a pass is made over all lists and the nodes with lower bound values higher than the upper bound in each list are eliminated.

It may also happen that the assignment problem (AP) solution does not yield a feasible solution, but a simple heuristic construction mechanism may be able to repair the AP solution. The heuristic construction mechanism employed in this study goes over the reduced costs in the AP solution and counts the number of variants with zero reduced cost in stage t, for each t. The stages with only one zero reduced cost are fixed and then the solution specification routine is repeated. This iterative procedure is repeated until either (i) a feasible complete solution is obtained, (ii) an infeasibility occurs, or (iii) the solution cannot be completed because of lack of positions with only one zero reduced cost. If the first outcome is obtained, the upper bound is updated. If the second outcome is obtained, the constructed solution is simply terminated. When the third outcome is obtained, the construction effort continues by assigning the variant that requires the most options among all possible variants with zero reduced cost at the first unassigned stage and iterations continue

Iterated Beam Search (β,ε,*MaxTime*)

1. Set *BestSolution* = *null* and *UpperBound* = ∞,.
2. Create *Root*, specify it and add it to $List_0$.
3. Set *LowestLowerBound* = *LowerBound*(*Root*).
4. While *Time* ≤ *MaxTime* and there is at least one non-empty $List_t$
5. For each level t (= 0, 1, .., T) with non-empty $List_t$
6. For each of the first β nodes in $List_t$ or
   While *LowerBound*(*Head*($List_t$)) ≤ *LowestLowerBound* + ε
7. Remove the head node from $List_t$ and make it *Parent*.
8. For each v (=1, 2, .., V) with $\lambda_{v,t+1} = 0$
9. Create $Child_v$ by branching extending *Parent* via assigning v to t
10. Specify $Child_v$ and solve its AP to obtain *LowerBound*($Child_v$)
11. If AP solution is feasible, then update *BestSolution* and *UpperBound*.
12. Else if $Child_v$ is feasible and *LowerBound*($Child_v$) < *UpperBound*
13. Add $Child_v$ to the appropriate $List_k$, $k>t$
14. Heuristically complete $Child_v$
15. If $Child_v$ is feasible, then update *BestSolution* and *UpperBound*.
    End if
    End for
16. Set $possible_{v,t+1}$=*false* in *Parent* for all v considered in Step 6.
17. Parent is now a stem, insert it back to $List_t$
    End for / while
18. Update *LowestLowerBound*

End while

19. Return *BestSolution* and *UpperBound*.
20. If all lists are empty, then *BestSolution* is optimal; else, return *LowestLowerBound*.

End.

Figure 4. Pseudo-code for the IBS algorithm.

until one of the first two outcomes is reached. The benefit from this upper bounding mechanism is that if a new upper bound is obtained, it is used to prune parts of the search tree earlier.

During a forward pass, if after evaluating β nodes at a level it is observed that the lower bound of the head node of the list is very close to the lowest lower bound, the exploration of nodes at that level can continue. The allowed maximum difference from the lowest lower bound for evaluation is captured in a parameter of the algorithm (ϵ). When used, this feature of the algorithm makes sure that the lowest lower bound of all unexplored nodes, which is also the best-known lower bound of the instance, is improved by at least ϵ in each iteration.

The algorithm terminates when there are no more nodes to explore, i.e. all lists are empty, or when the maximum allowed computation time is reached. In the former case, the algorithm has found the optimal solution, whereas in the latter the algorithm reports the lowest lower bound, which lets us calculate the optimality gap.

A brief pseudo-code for the iterated beam search algorithm developed in this section is given in Figure 4.

5. Computational experiment and results

5.1 *Experiment plan*

In this study the 36-instance test bed developed by Drexl, Kimms, and Matthiessen (2006) is used as the primary set of test instances for performance evaluation. The limitation of this set is that it contains relatively small (up to 50 cars) instances. A second test set of 18 instances of up to 100 cars was generated using Drexl et al.'s generator and the same option constraints. This new set is summarised in Table 4. Data files of the instances are available from the author upon request.

The IBS algorithm is coded in Microsoft Visual C# 2010 Express, which is a freely available software development platform as part of the Microsoft Visual Studio 2010 package. The experiments are run on a personal computer with an Intel Core i7 870 CPU and 16 GB RAM.

After preliminary experiments, it was found that setting $\beta = 30$, forcing a lower bound improvement in the proximity of $\epsilon = 0.001$ in each iteration, works well.

5.2 *Results*

Table 4 shows a summary of the outcomes from the experiments on both sets of test instances. The first column is the instance number, and the next four columns show the size and characteristics of the instance; T is the sequence length, O is the number of options, the option constraint sets are chosen from either an easy or a hard group, and V is the number of variants. Instance numbers printed in bold represent the instances that were solved to optimality within an hour of computation time. The last four columns show some important time information, all in seconds. The 'First' column shows the time of finding the first feasible solution. Similarly, the 'Best' column shows the time of finding the best solution obtained for the instance, which is the optimal solution for those instances printed bold. The 'Finish' column shows the completion time for optimally solved instances and 'n/c' (not completed) for the rest. Finally, [DKM06] is the time reported by Drexl, Kimms, and Matthiessen (2006) for the first 36 instances.

The first observation made from Table 4 is the negligibly short computation time taken by some of the instances. Instances #1–23, 25–27, 29, 31–33, 35, 37–39, 43 and 49 are solved within one minute. Among these instances #1–3, 5, 14 and 26 were solved in practically no time because the root solution yielded a feasible solution. More specifically, the lower bound obtained by solving an assignment problem on the specification of the root was feasible. This is an exceptional situation that eliminates the need to branch on the root solution.

The second observation made from Table 4 is that, in 35 of the 36 instances, the computation time taken by the IBS method is either negligible (under one minute) or shorter than the best-known computation time in the literature. Part of the credit must be given to the speed of the computer used in the current study. To account for the speed difference, the IBS method is given only one hour compared with the earlier three-hour limit. In instances #24, 28 and 34 it is seen that the IBS method has reduced the computation time requirement from hours to minutes. For instance #36, which was previously unsolved, the optimal solution is obtained within 1401 seconds, i.e. approximately 24 minutes.

The third observation from Table 4 is the time required to find a feasible solution. In 50 of the 54 instances, including some with 100 cars, finding a feasible solution took up to 2 seconds. Note that the objective function does not measure feasibility and the search does not focus on finding a feasible solution. The focus on finding a level sequence takes attention away from finding a feasible solution, which is expected to delay finding feasible solutions. Nevertheless, the algorithm was able to find a feasible solution for all test instances and took about 7 minutes in the worst case. Considering that a measure of (in)feasibility is not the guide of the search, this result is significant and also demonstrates the effectiveness of the IBS method in finding feasible solutions.

Table 4. Summary of computational outcomes.

Instance #	T	O	Easy/ hard	V	Time (seconds) First	Best	Finish	[DKM06]
1	10	3	Easy	5	0	0	0	0
2	10	3	Hard	4	0	0	0	0
3	10	5	Easy	7	0	0	0	0
4	10	5	Hard	6	0	0	0	0
5	10	7	Easy	9	0	0	0	0
6	10	7	Hard	9	0	0	0	0
7	15	3	Easy	5	0	1	1	0
8	15	3	Hard	4	0	0	0	0
9	15	5	Easy	8	0	1	1	1
10	15	5	Hard	7	0	1	1	0
11	15	7	Easy	12	0	0	0	0
12	15	7	Hard	10	0	1	1	0
13	20	3	Easy	5	0	0	1	0
14	20	3	Hard	4	0	0	0	0
15	20	5	Easy	9	0	1	1	1
16	20	5	Hard	8	0	2	2	1
17	20	7	Easy	14	0	1	1	2
18	20	7	Hard	12	0	1	2	58
19	30	3	Easy	5	0	1	1	1
20	30	3	Hard	4	0	0	1	0
21	30	5	Easy	10	0	1	1	1
22	30	5	Hard	9	0	2	4	242
23	30	7	Easy	17	0	1	2	3
24	30	7	Hard	16	0	249	446	9474
25	40	3	Easy	6	0	4	4	2
26	40	3	Hard	4	0	0	0	0
27	40	5	Easy	12	0	3	3	24
28	40	5	Hard	10	0	57	596	7095
29	40	7	Easy	19	0	2	3	11
30	40	7	Hard	19	0	1523	n/c	n/c
31	50	3	Easy	6	2	7	7	11
32	50	3	Hard	5	0	0	2	233
33	50	5	Easy	12	0	8	8	87
34	50	5	Hard	12	0	18	231	10,364
35	50	7	Easy	20	0	14	47	195
36	50	7	Hard	22	1	507	1401	n/c
37	60	3	Easy	6	0	5	6	–
38	60	3	Hard	6	0	7	36	–
39	60	5	Easy	13	1	11	16	–
40	60	5	Hard	13	0	3173	n/c	–
41	60	7	Easy	22	0	21	142	–
42	60	7	Hard	23	1	139	n/c	–
43	80	3	Easy	6	1	40	40	–
44	80	3	Hard	6	0	27	306	–
45	80	5	Easy	14	0	126	192	–
46	80	5	Hard	13	19	2804	n/c	–
47	80	7	Easy	23	1	167	n/c	–
48	80	7	Hard	23	94	2529	n/c	–
49	100	3	Easy	6	1	44	45	–
50	100	3	Hard	6	0	43	642	–
51	100	5	Easy	14	2	89	451	–
52	100	5	Hard	13	60	1058	n/c	–
53	100	7	Easy	24	1	150	n/c	–
54	100	7	Hard	26	438	1794	n/c	–

The fourth observation from Table 4 is the success of finding optimal solutions. In 45 of the 54 instances the IBS algorithm proposed here successfully found an optimal solution within an hour of computation time. Of those 45 solutions, six are $T = 10$ car instances, six $T = 15$, six $T = 20$, six $T = 30$, five $T = 40$, six $T = 50$, four $T = 60$, three

$T = 80$ and three $T = 100$. Here, it is seen that the number of cars, i.e. the sequence length, is an important indicator of the problem's solvability, with the larger instances being less likely to be solved within an hour. Similarly, all 18 of the instances with $O = 3$, 15 of the 18 with $O = 5$ and only 12 of the 18 with $O = 7$ were solved. The number of options, O, significantly increases the complexity of the problem, with larger O values requiring more computational effort. Finally, 25 of the 27 instances with easy constraints were solved, whereas only 20 of those with hard constraints were solved. Overall, the results show that larger instances with seven hard constraints are the most challenging.

The fifth and final observation from Table 4 is the difference between the time an optimal solution is found and the time the algorithm terminates. When the algorithm finds an optimal solution, it only notes that it is better than the best previously know solution, and updates the upper bound. When the upper bound is updated a pass is made over all lists of unexplored nodes and those with a lower bound exceeding the new upper bound are eliminated.

In some cases, the last upper bound update eliminates a great majority of the remaining unexplored solutions, thereby letting the algorithm successfully complete the search in a short time thereafter. In instances #43 and 49 we see an example of this, with the search terminating within one second after finding an optimal solution.

In other cases, there are many unexplored nodes with a lower bound below the new upper bound that still need to be explored. The best solution found by the algorithm may already be optimal, and the upper bound may not be updated again. The algorithm continues the search by exploring and eliminating more solutions and continually increasing the lower bound in the process. If the given one hour of computation time suffices, the algorithm concludes that it found the optimal solution, such as in instance #51. In this instance the optimal solution was found in 89 seconds, however it was not proven until after all the remaining nodes were explored, in 451 seconds. Sometimes the one-hour limit is reached before the algorithm can explore all remaining nodes. For example, in instance #54, the first feasible solution is found in 438 seconds, and the upper bound is likely updated a few times, the last one being in the 1794th second and in the remaining half hour no better solution is found. Whether the best solution found so far is optimal or not remains unknown. For those nine instances that were not solved to optimality within the allotted hour, the algorithm returns lower and upper bounds, which are summarised in Table 5.

The first column of Table 5 shows the instance number. The next two show the lower and upper bound values at termination, respectively. The following two columns show the optimality gap as a value and percentage. The final two columns show the first obtained bounds. The obtained lower bound is obtained from the root solution at the very beginning and the first upper bound is obtained from the first feasible complete solution of the problem. The table shows that the nine unsolved instances have optimality gaps ranging from 12 to 120.6%. These instances can be used as a benchmark in future research. The upper bounds can be used for direct comparison of the solution quality with any other method and the lower bounds can be helpful in measuring optimality gaps.

The first lower bounds show that, for some problems, the method largely improved the best-known lower bound. For example, in instance #53 the first lower bound was 258.758 and it increased to 457.182 during the search. Similarly, the first upper bounds show that the search process was successful at finding much better solutions than the first-found feasible complete solution.

Our final results concern the contribution of using the DP structure. As explained earlier in the paper, when a node is created it is considered for insertion into an ordered list. If a node with the same critical characteristics already exists in the list, a comparison of the lower bounds is made and only the better of the two nodes is kept in the list. This can lead to significant time savings, as with each eliminated node a subtree is eliminated. Table 6 highlights the node elimination performance of the method on instances that took at least 60 seconds of computation time. The DP filter column shows the percentage of nodes eliminated by the DP structure and the UB filter column shows the percentage of the

Table 5. Optimality gap for unsolved instances.

Instance #	Bounds at termination		Optimality gap		First bounds	
	Lower	Upper	Value	%	Lower	Upper
30	127.939	157.879	29.940	23.4	117.576	207.333
40	132.444	152.667	20.223	15.3	96.444	236.222
42	209.667	300.000	90.333	43.1	190.000	335.333
46	123.143	225.365	102.222	83.0	92.698	379.333
47	326.222	365.333	39.111	12.0	214.889	546.889
48	222.533	440.400	217.867	97.9	213.333	473.600
52	139.460	272.253	132.793	95.2	113.429	430.828
53	457.182	623.273	166.091	36.3	258.758	737.818
54	302.786	668.048	365.262	120.6	289.881	850.857

Table 6. Summary of node eliminations.

#	Created	DP filter (%)	UB filter (%)	Explored	
24	183,407	14.8	0.3	155,666	84.9%
28	204,280	31.8	0.1	139,129	68.1%
30	629,263	29.5	0.0	227,719	36.2%
34	49,598	22.3	0.2	38,455	77.5%
36	264,648	15.8	0.1	222,650	84.1%
40	467,950	41.4	0.1	203,101	43.4%
41	9872	2.3	0.5	9596	97.2%
42	302,288	18.0	0.1	103,801	34.3%
44	24,202	39.9	0.5	14,427	59.6%
45	7115	8.7	3.2	6272	88.2%
46	310,627	43.8	0.2	135,366	43.6%
47	132,534	6.9	0.0	106,870	80.6%
48	192,068	30.7	0.0	71,185	37.1%
50	35,584	42.0	0.5	20,478	57.5%
51	9520	9.4	5.5	8102	85.1%
52	164,258	40.1	0.4	72,178	43.9%
53	107,699	5.7	0.0	83,355	77.4%
54	114,684	20.7	0.2	49,366	43.0%

nodes eliminated by an upper bound update. The last two columns show the number and percentage of the nodes explored since they were not eliminated by any filter. The rows in bold represent the solved instances. Note that, for those solved instances, the three percentages add up to 100%, whereas for those unsolved instances they do not add up to 100% due to the remaining unexplored nodes.

From the percentage of explored nodes it is seen that a significant number (up to 43.8%) of the nodes are eliminated. This observation proves that significant time savings are obtained by using a DP structure and frequent upper bound updates. A comparison between the two reveals that the DP structure is the more effective of the two.

Odd-numbered instances contain easy option constraints and even-numbered instances contain difficult option constraints (see Table 4). For the instances with easy constraints, the percentage of nodes eliminated by the DP filter ranges from 2.3 to 9.4%, whereas the range is from 14.8 to 43.8% with hard constraints. This result suggests that the DP filter is more effective on instances with hard constraints. Also, the percentages do not appear to change with problem size.

6. Conclusions and future research

The results discussed in the previous section prove that the primary research goal of developing an efficient and effective iterated beam search solution to the CCSLSP problem has been achieved. For 35 of the 36 test instances from the existing literature, the algorithm found the optimal solution in under one hour. This includes one instance that was previously unsolved. Additionally, for all previously solved instances, IBS found an optimal solution within only one minute, or in much less time than reported in the existing literature. Also, for 10 of the 18 newly created test instances, the algorithm successfully found an optimal solution in an hour, altogether bringing the number of solved instances to 45 out of 54.

The computational results show that the problem is less likely to be solved for large instances with seven hard option constraints. The results also show that using the DP structure developed in Section 3.3 as a filtering mechanism filters out a significant percentage of nodes, and thus speeds up the IBS algorithm. Moreover, the DP filter is more effective on instances with hard option constraints, regardless of problem size. This result gives hope for the development of a new, more efficient algorithm exploiting the DP structure more aggressively, which is a future research direction for the author.

The IBS algorithm proposed here is an intelligent algorithm that incorporates intelligence developed by analysing the structural properties of the CCSLSP in an iterative implementation of the beam search heuristic. In the core of the algorithm is a branching mechanism that prioritises possible branches. The algorithm also takes advantage of a DP formulation in filtering out a good percentage of suboptimal partial solutions and thereby avoiding the creation of multiple solutions with the same key characteristics. For each partial solution, an AP formulation is built and solved to obtain a tight lower bound. The AP formulation may yield a feasible solution directly, which makes the lower bound tight and gives an upper bound immediately. Or, a constructive heuristic routine is invoked to complete the partial solution to a

feasible complete solution with a low objective function value. In either case the upper bound may be updated and parts of the search tree can be eliminated with each upper bound update.

The AP formulation is critical for the algorithm. First, its solution gives a tight lower bound and unexplored partial solutions are sorted and retrieved in order of their lower bound values. The algorithm keeps track of the lowest lower bound, improves it by each iteration and if it terminates with some unexplored nodes at hand, it reports the lowest lower bound to be used in the optimality gap calculation. Second, the largest AP to be solved requires assigning T cars to T positions, which is solvable in $O(T^3)$ time. This computational complexity is low and makes the algorithm more time-effective. It should also be stated that the key to the AP formulation is the level scheduling objective adopted in the model. The current *minimum summation of absolute deviations from ideal positions of variant copies* objective lends itself to an AP formulation very easily.

The CSP is a very important constraint satisfaction problem. Efficient solution methods to find feasible solutions of the CSP with several hundred cars is desired by the constraint satisfaction as well as optimisation communities. The optimisation approach to the CSP requires defining an objective function that measures soft constraint violations and then focuses on finding a solution with zero objective function value. Tailoring the proposed IBS method to solve the CSP is a future research direction. The reason is that all straightforward approaches to an objective function formulation for the CSP (Gagne, Gravel, and Price 2006; Bautista, Pereira, and Adenso-Diaz 2008; Fliedner and Boysen 2008) consider variants occupying positions in each option's constraint windows. The adoption of that kind of objective function, instead of the current objective that matches variant copies to positions one-to-one, cannot be used in an assignment problem formulation, and, thus, would lose the benefit of obtaining tight lower bounds quickly. Developing a completely different objective function for the CSP would be a required first step to tailor the IBS to the CSP.

Another arising optimisation problem that is related to just-in-time manufacturing is the response time variability problem (RTVP). In the RTVP, one assumes that the production sequence will be executed not only once, but infinitely many times. Then, the focus on the ideal positions in a sequence is replaced by a focus on the distances between different copies of the same variant. The modelling of the RTVP is more complicated due to the need to tie the end of the sequence to the beginning. However, RTVP is an important problem not only in manufacturing systems, but also various service systems. Therefore, solution of the RTVP in conjunction with the option constraints, which can also be generalised to various real-life systems, is an important future research direction.

Acknowledgements

The author gratefully thanks Drs. Andreas Drexl, Alf Kimms and Lars Mathießen for providing the test instance generator of their study. Their collaboration is greatly appreciated. The contributions of two anonymous reviewers are also deeply appreciated. The reviewers' questions and comments triggered a substantial revision of the paper and improved the paper greatly.

References

Ball, M. O. 2011. "Heuristics Based on Mathematical Programming." *Surveys in Operations Research and Management Science* 16 (1): 21–38.

Bautista, J., J. Pereira, and B. Adenso-Diaz. 2008. "A Beam Search Procedure Approach for the Optimization Version of the Car Sequencing Problem." *Annals of Operations Research* 159: 233–244.

Boysen, N., M. Fliedner, and A. Scholl. 2009. "Sequencing Mixed-Model Assembly Lines: Survey, Classification and Model Critique." *European Journal of Operational Research* 192 (2): 349–373.

Dincbas, M., H. Simonis, and P. van Hentenryck. 1988. "Solving the Car Sequencing Problem in Constraint Logic Programming." In: *Proceedings of the 8th European Conference on Artificial Intelligence, ECAI-88*, Munich, 290–295.

Drexl, A., and A. Kimms. 2001. "Sequencing JIT Mixed-Model Assembly Lines under Station-Load and Part-Usage Constraints." *Management Science* 47: 480–491.

Drexl, A., A. Kimms, and L. Matthiessen. 2006. "Algorithms for the Car Sequencing and the Level Scheduling Problem." *Journal of Scheduling* 9: 153–176.

Erel, E., Y. Gocgun, and I. Sabuncuoglu. 2007. "Mixed-Model Assembly Line Sequencing Using Beam Search." *International Journal of Production Research* 45 (22): 5265–5284.

Esteve, B., et al. 2006. "A Recovering Beam Search Algorithm for the Single Machine Just-in-Time Scheduling Problem." *European Journal of Operational Research* 172 (3): 798–813.

Fliedner, M., and N. Boysen. 2008. "Solving the Car Sequencing Problem via Branch & Bound." *European Journal of Operational Research* 191 (3): 1023–1042.

Gagne, C., M. Gravel, and W. L. Price. 2006. "Solving Real Car Sequencing Problems with Ant Colony Optimization." *European Journal of Operational Research* 174 (3): 1427–1448.

Gent, I. P. and T. Walsh 1999. "CSPLib: A Benchmark Library for Constraints [online]." Technical Report APES-09-1999. Available from: http://csplib.cs.strath.ac.uk/.

Ghirardi, M., and C. N. Potts. 2005. "Makespan Minimization for Scheduling Unrelated Parallel Machines: A Recovering Beam Search Approach." *European Journal of Operational Research* 165 (2): 457–467.

van Hentenryck, P., H. Simonis, and M. Dincbas. 1992. "Constraint Satisfaction Using Constraint Logic Programming." *Artificial Intelligence* 58: 113–159.

Inman, R. R., and R. L. Bulfin. 1991. "Sequencing JIT Mixed Model Assembly Lines." *Management Science* 37: 901–904.

Kang, S. G., and S. H. Choi. 2010. "Multi-Agent Based Beam Search for Intelligent Production Planning and Scheduling." *International Journal of Production Research* 48 (11): 3319–3353.

Kis, T. 2004. "On the Complexity of the Car Sequencing Problem." *Operations Research Letters* 32: 331–335.

Kubiak, W. 1993. "Minimizing Variation of Production Rates in Just-in-Time Systems: A Survey." *European Journal of Operational Research* 66: 259–271.

Kubiak, W., and S. Sethi. 1991. "A Note on Level Schedules for Mixed-Model Assembly Lines for Just-in-Time Production Systems." *Management Science* 37: 121–122.

Kuhn, H. W. 1955. "The Hungarian Method for the Assignment Problem." *Naval Research Logistics Quarterly* 2: 83–97.

Lesert, A., et al. 2011. "Definition of Spacing Constraints for the Car Sequencing Problem." *International Journal of Production Research* 49 (4): 963–994.

Leu, Y.-Y., P. Y. Huang, and R. S. Russell. 1998. "Using Beam Search Techniques for Sequencing Mixed-Model Assembly Lines." *Annals of Operations Research* 70: 379–397.

Mackelprang, A. W., and A. Nair. 2010. "Relationship between Just-in-Time Manufacturing Practices and Performance. A Meta-Analytic Investigation." *Journal of Operations Management* 28 (4): 283–302.

McMullen, P. R., and P. Tarasewich. 2005. "A Beam Search Heuristic Method for Mixed-Model Scheduling with Setups." *International Journal of Production Economics* 96 (2): 273–283.

Miltenburg, J. 1989. "Level Schedules for Mixed-Model Assembly Lines for Just-in-Time Production Systems." *Management Science* 35: 192–207.

Miltenburg, J., and G. Sinnamon. 1989. "Sequencing Mixed-Model Just-in-Time Production Systems." *International Journal of Production Research* 27: 1487–1509.

Monden, Y. 1983. Toyota Production System. Industrial Engineering and Management Press.

Ow, P. S., and T. E. Morton. 1988. "Filtered Beam Search in Scheduling." *International Journal of Production Research* 26 (1): 35–62.

Parello, B. D., W. C. Kabat, and L. Wos. 1986. "Job-Shop Scheduling Using Automated Reasoning: A Case Study of the Car-Sequencing Problem." *Journal of Automated Reasoning* 2 (1): 1–42.

Sabuncuoglu, I., Y. Gocgun, and E. Erel. 2008. "Backtracking and Exchange of Information: Methods to Enhance a Beam Search Algorithm for Assembly Line Scheduling." *European Journal of Operational Research* 186 (3): 915–930.

Shi-Jin, W., Z. Bing-Hai, and X. Li-Feng. 2008. "A Filtered-Beam-Search-Based Heuristic Algorithm for Flexible Job-Shop Scheduling Problem." *International Journal of Production Research* 46 (11): 3027–3058.

Solnon, C., et al. 2008. "The Car Sequencing Problem: Overview of the State-of-the-Art Methods and Industrial Case-Study of the ROADEF'2005 Challenge Problem." *European Journal of Operational Research* 191 (3): 912–927.

Tsang, E. P. K. 1993. *Foundations of Constraint Satisfaction*. London: Academic Press.

Valente, J. M. S., and R. A. F. S. Alves. 2005. "Filtered and Recovering Beam Search Algorithms for the Early/Tardy Scheduling Problem with No Idle Time." *Computers & Industrial Engineering* 48 (2): 363–375.

Yavuz, M. 2010. "An Iterated Beam Search Algorithm for the Multi-Level Production Smoothing Problem with Workload Smoothing Goal." *International Journal of Production Research* 48 (20): 6189–6202.

Yavuz, M., and E. Akcali. 2007. "Production Smoothing in Just-in-Time Manufacturing Systems: A Review of the Models and Solution Approaches." *International Journal of Production Research* 45 (16): 3579–3597.