Les Cahiers du GERAD ISSN: 0711–2440

An exact dynamic programming algorithm for the precedence-constrained class sequencing problem

R. Bürgy,
P. Baptiste, A. Hertz

G–2017–82

October 2017
Revised: October 2019

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2019
– Bibliothèque et Archives Canada, 2019

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2019
– Library and Archives Canada, 2019

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

An exact dynamic programming algorithm for the precedence-constrained class sequencing problem

Reinhard Bürgy [a]

Pierre Baptiste [b,c]

Alain Hertz [b,c]

[a] *Département d'Informatique, Université de Fribourg, Fribourg, Switzerland*

[b] *Département de Mathématiques et de Génie Industriel, Polytechnique Montréal, Montréal (Québec), Canada*

[c] *GERAD, Montréal (Québec), Canada*

reinhard.buergy@unifr.ch
pierre.baptiste@polymtl.ca
alain.hertz@gerad.ca

October 2017
Revised: October 2019
Les Cahiers du GERAD
G–2017–82

Abstract: This article discusses the precedence-constrained class sequencing problem (PCCSP). In scheduling terms, this is a one-machine scheduling problem with precedence constraints and setups with the goal of minimizing the number of setups. From a practical perspective, PCCSP covers a wide range of applications such as, for example, scheduling problems in systems where multipurpose processors need retooling to switch from one operation to another. Previous research has shown that PCCSP is difficult to solve from both a theoretical and computational perspective, and only little research has been conducted on computational methods. This article bridges this gap by proposing a dynamic programming algorithm for solving PCCSP exactly. It comprises specialized dominance rules, lower bound computations, propagation algorithms, and heuristics that successfully exploit the problem's structure. Based on extensive numerical experiments, we analyze the algorithm in detail and show that it outperforms mixed-integer programming and constraints programming models.

Keywords: Sequencing, setup times, precedence constraints, one-machine scheduling, dynamic programming

Acknowledgments: R. Bürgy was partially supported by the Swiss National Science Foundation Grant P2FRP2_161720, which is gratefully acknowledged.

1 Introduction

Given are some operations, each belonging to some class, and precedence constraints among operations. The operations must be performed sequentially in a one-machine environment, and a setup is required between the executions of two consecutive operations if they belong to different classes. The precedence-constrained class sequencing problem (PCCSP) asks for sequencing all operations so as to minimize the number of setups while respecting all precedence constraints.

More formally, given are a set V of n operations, a set $\mathcal{C}$ of classes, and a set $A \subseteq V \times V$ of precedence constraints. Each operation $v \in V$ belongs to exactly one class $c_v \in \mathcal{C}$. Let $C \in \mathcal{C}^V$ be the class assignment vector. An instance of PCCSP will be denoted by $(V, A, \mathcal{C}, C)$. Consider the directed graph $G = (V, A)$, where each operation $v \in V$ is represented by a vertex v and each precedence constraint $(v, w) \in A$ is modeled by an arc (v, w). Graph G will be called *precedence graph*. A solution of PCCSP is a total ordering of the operations that satisfies all precedence constraints. Equivalently, a solution is a bijection $f : \{1, \ldots, n\} \to V$ such that $(f(j), f(i)) \notin A$ for all $1 \leq i < j \leq n$, and its objective value –the number of setups– is $\sum_{i=1}^{n-1} s_i$, where $s_i = 1$ if $c_{f(i)} \neq c_{f(i+1)}$ and 0 otherwise. Clearly, a solution exists if and only if there is no circuit in G. Hence, we assume that G has no circuit, which can efficiently be checked in a preprocessing step.

An illustrative example of a PCCSP instance is given in Figure 1. It consists of 17 operations, 3 classes (represented by the colors white, gray and black), and 24 precedence constraints. The sequence (9, 8, 12, 1, 2, 6, 10, 13, 3, 14, 4, 5, 11, 15, 17, 7, 16) is an optimal solution with six setups.

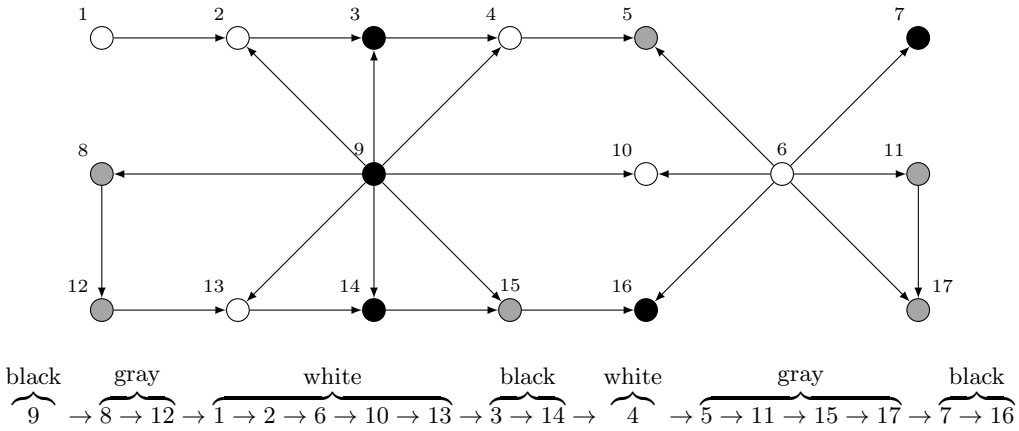

Figure 1: The precedence graph (above) and an optimal sequence (below) of a PCCSP instance.

Lofgren (1986) and Lofgren et al. (1991) introduce PCCSP. The authors address the problem of routing printed circuit boards through an assembly cell and define PCCSP to capture this practical problem with graph theory. They prove that PCCSP is NP-hard, show that two classes of heuristics have arbitrary bad worst case performance, develop a set of heuristics, and provide some computational results. Interestingly, the authors prove that the non-repetitive shortest common supersequence problem, which is a restricted but still NP-hard version of the well-known shortest common supersequence problem, is a special case of PCCSP. Altogether, their prolific contributions show that PCCSP is a challenging and practically relevant optimization problem. This view is further supported by Tovey (2004) and Correa et al. (2007) who prove that no polynomial-time algorithm with constant worst-case performance exists for PCCSP unless $\mathcal{P} = \mathcal{NP}$. We remark that PCCSP is trivial to solve if no precedence constraints exist or if there are at most two classes. However, as shown in (Lofgren et al., 1991; Darte, 2000), it becomes NP-hard in the strong sense if there are at least three classes, even if the precedence graph consists only of disjoint paths.

From a practical perspective, PCCSP models various recurring scheduling problems in systems where multipurpose processors, i.e., processors with the flexibility to perform multiple types of operations, need retooling or some other type of setup to switch from one operation to another (Tovey,

2004). Lofgren et al. (1991) describe an application in the semiconductor industry. Darte (2000) mentions a traveling salesman application where the salesman (or another type of worker) must perform a set of tasks in some partial order, each task must be executed in some given city, and the goal is to minimize the number of moves between cities. Camelot (2012) studies an aircraft disassembly process where reusable parts of an end-of-life aircraft have to be collected subject to precedence constraints, each part belongs to a specific zone of the aircraft, and the objective is to minimize the number of moves from one zone to another.

PCCSP is closely related to a loop fusion problem addressed by Darte (2000). Loop fusion is a standard compiler optimization and loop transformation process in computer science. The loop fusion problem of Darte (2000) can briefly be described as follows. Given is a set of loops, which is the same as the set of operations in PCCSP. Each loop belongs to some type (of loops) and a set of precedence-constraints between loops is specified. The goal is to find a total ordering of the loops so that the number of setups is minimized. A setup must be performed when switching from one type of loop to another, and –this is the difference with PCCSP– one can also specify the need for a setup between loops of the same type. These so-called fusion-preventing constraints make the loop fusion problem more general than PCCSP. Darte (2000) formulates the loop fusion problem with graph theory, establishes a link to the scheduling literature, and develops complexity results for some variants of the loop fusion problem. We refer to his publication for further details.

PCCSP is also related to the following mixed graph coloring problem. Given is a mixed graph $G^{\text{mix}} = (V, A, E)$ where V is a set of vertices, A a set of arcs and E a set of edges. A function ϕ is a coloring of G^{mix} if it assigns to each vertex $v \in V$ a positive integer $\phi(v)$ such that $\phi(v) \leq \phi(w)$ if $(v, w) \in A$ and $\phi(v) \neq \phi(w)$ if $\{v, w\} \in E$. The number of colors of a coloring ϕ is given by the largest integer $\phi(v)$ assigned to a vertex $v \in V$. The goal is to find a coloring of G with a minimum number of colors. An instance $(V, A, \mathcal{C}, C)$ of PCCSP can be formulated as a mixed graph coloring problem as follows. The mixed graph $G^{\text{mix}} = (V, A, E)$ is obtained by simply adding an edge $\{v, w\} \in E$ to the precedence graph $G = (V, A)$ for each pair of vertices v, w of distinct classes, i.e., $c_v \neq c_w$. Observe that the undirected part (V, E) of G^{mix} has a special structure: it is a complete k-partite graph where k equals the number $|\mathcal{C}|$ of classes. Given a coloring ϕ, a solution of PCCSP can be obtained as follows. According to coloring ϕ, we first execute all operations v with $\phi(v) = 1$, then all operations v with $\phi(v) = 2$, and continue this process until all operations are executed. Considering a group of operations with the same color, we execute them in an order that respects the precedence constraints A, which can easily be found as the precedence graph G has no circuit. The so-obtained total ordering of the operations is a solution of PCCSP, and the number of setups is exactly the number of colors of ϕ minus 1 as all operations with the same color $\phi(v)$ belong to the same class. As a result, a coloring of G^{mix} with a minimum number of colors corresponds to a solution of PCCSP with a minimum number of setups.

The discussed mixed graph coloring problem is introduced by Sotskov et al. (2002). The authors discuss some theoretical properties and devise an exact algorithm for solving this problem. A more efficient exact algorithm is subsequently proposed by Andreev et al. (2000), who develop a branch-and-bound algorithm based on some conflict resolution and arc adding strategies. Based on numerical experiments, the authors conclude that their approach is less suited for mixed graphs where the number of edges is larger than the number of arcs as their lower bounding procedures are not taking into account the edges. In PCCSP instances, the number of edges is typically much larger than the number of arcs. Hence, solving PCCSP instances via their mixed graph coloring approach seems to be unpromising. Meuwly et al. (2010) propose heuristic solution approaches for the mixed graph coloring problem. Their tabu and variable neighborhood search methods are based on particol (see Blöchliger and Zufferey, 2008), which is a robust tabu search heuristic for the standard graph coloring problem. Additional theoretical investigations and algorithms for slightly different mixed graph coloring problems can be found in (Hansen et al., 1997; Ries, 2007; Kouider et al., 2017).

This paper presents a dynamic programming (DP) algorithm enabling to solve to optimality large instances of PCCSP within reasonable computation times. The dynamic program is based on a refor-

mulation of the problem and is solved by applying specialized dominance rules, lower bound computations, propagation algorithms, and heuristics, which help to drastically reduce the state space. Such techniques have regularly been proposed to improve the effectiveness of DP approaches. Interested readers are referred to (Morin and Marsten, 1976; Easton, 1990; Mingozzi et al., 1997; Sewell and Jacobson, 2012).

The remainder of this article is organized as follows. The next section proposes a DP algorithm for PCCSP. After introducing the general DP structure, the specific sub-procedures incorporated into the method are discussed. Section 3 provides a computational study, which is based on 288 instances obtained from an instance generation scheme of Lofgren et al. (1991). The numerical results are used to analyze the sub-procedures of the DP algorithm, to discuss the difficulty of the generated instances, and to compare our algorithm with integer linear programming and constraint programming methods. Concluding remarks are given in Section 4 and the detailed computational results are provided in the appendix.

2 A dynamic programming algorithm for PCCSP

Given a PCCSP instance $(V, A, \mathcal{C}, C)$, a solution can be constructed by successively selecting a next class and executing all possible operations of this class so that the precedence constraints are satisfied, until all operations are executed. This scheme, introduced by Lofgren et al. (1991), gives an indirect representation of the solutions by class sequences, which makes it possible to specify the optimization problem on the solution space of the class sequences.

This can be formalized as follows. Let $\widehat{G} = (V, \widehat{A})$ be the transitive closure of precedence graph $G = (V, A)$, i.e., an arc (v, w) belongs to $\widehat{A}$ if and only if there exists a path from v to w in graph G. A vertex (or operation) v is *available* for its execution if there is no w in graph G with $(w, v) \in \widehat{A}$ and $c_w \neq c_v$, and a class $c \in \mathcal{C}$ is *available* to be executed if it contains at least one available vertex. We define the *execution* of an available class $c \in \mathcal{C}$ in graph $G = (V, A)$ to be the deletion of all available vertices $v \in V$ of class $c_v = c$ (and all incident arcs) resulting in a graph $G' = (V', A')$, and denote this operation with symbol $\odot$, i.e., $G' = G \odot c$. A class sequence $S = (c(1), c(2), ...c(r))$ is called *feasible* if every class $c(i)$, $i \in \{1, \ldots, r\}$, is available after the execution of $c(1), \ldots, c(i-1)$, and the successive execution of the classes in S results in the deletion of all vertices of G. Given any feasible class sequence S, a solution of PCCSP, which is a total ordering of the operations, can simply be obtained by additionally sequencing all operations that are deleted at the same time in S. This is a simple task as G has no circuit, and the objective value of the resulting solution (i.e., the number of setups) is the length of the class sequence S minus one. We call a feasible class sequence *optimal* if it is of shortest length among all feasible class sequences.

As shown next, we can solve PCCSP by searching for an optimal class sequence.

Proposition 1 *An optimal solution of PCCSP can be constructed from an optimal class sequence by the simple scheme introduced above.*

Proof. First, each feasible solution of PCCSP exactly matches to one class sequence. This sequence can be obtained by looking at the sequence of executed classes in the solution. However, the obtained class sequence may not be feasible w.r.t. the above definition. Indeed, one may choose to switch to another class in a solution of PCCSP without executing all available operations of the current class. However, when executing an operation of a certain class, it is profitable (or at least not disadvantageous) to perform all available operations of this class before switching to another. Consequently, there exists an optimal solution of PCCSP that is representable by a feasible class sequence. Second, each feasible class sequence corresponds to at least one feasible solution of PCCSP, and all corresponding solutions have the same objective value, namely the length of the class sequence minus one. Therefore, from an optimal class sequence, one can always construct an optimal solution of PCCSP. □

To find an optimal class sequence, we develop a DP algorithm with the decision tree structure depicted in Figure 2. Each state of DP is specified by a precedence graph $G = (V, A)$, where V are

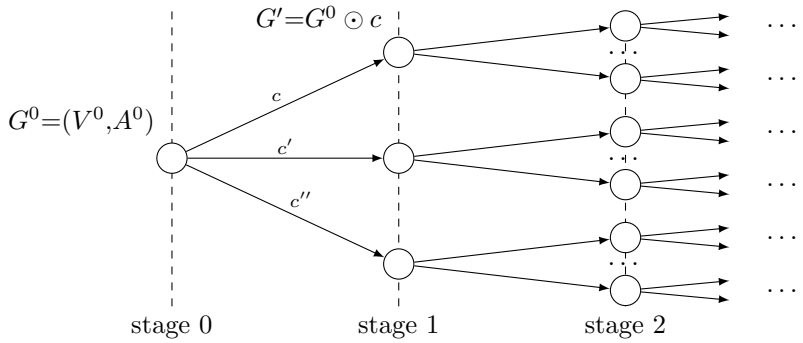

Figure 2: Sketch of the decision tree of the dynamic program.

the remaining operations to be executed and A the precedences that must be respected. In each stage of DP, we decide which available class to execute next. For each state $G = (V, A)$ of a stage n, the minimal number of class executions needed to remove all remaining operations V can then be expressed by the following recursive optimal value function

$$v_n(G) = \begin{cases} 1 + \min\limits_{\substack{c \in \mathcal{C}: \\ c \text{ avail.}}} \{v_{n+1}(G \odot c)\} & \text{if } G \text{ is not empty,} \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

where an empty graph is a graph without vertices. Looking at the structure of the optimal value function, one can observe that the number of class executions for reaching any state in stage n is n. Hence, we can define the minimal length of a class sequence for state G in stage n to be

$$f_n(G) = n + v_n(G), \tag{2}$$

which is attained when executing the first n classes according to the decisions on the path from the root of the decision tree to state G and removing the last V operations in an optimal fashion according to (1).

By construction of the decision tree, an optimal class sequence can be constructed as soon as a stage is reached with a state consisting of an empty graph. The DP algorithm relies on this observation and constructs the decision tree stage-wise until reaching a state $G = (V, A)$ with $V = \emptyset$.

A major problem of this algorithm is that the number of states quickly becomes unmanageably large. We can, however, drastically reduce the number of states that must be considered within any stage by the following two simple observations that rely on comparing the sets of remaining operations of two states $G = (V, A)$ and $G' = (V', A')$ of a stage n. First, if the two states have the same set of remaining operations i.e., $V = V'$, then, considering (2), we can delete one of these "duplicated" states from the decision tree. Second, if the set of remaining operations V is a proper subset of V' i.e., $V \subset V'$, then

$$v_n(G) \leq v_n(G') \tag{3}$$

holds by definition of operation $\odot$ and (1), and we can delete G' as it is dominated by G according to (2). In each stage, we compare each pair of its states and check if we can apply the above rules to delete some states.

While we typically delete a substantial amount of states with these rules, the decision tree still tends to grow rapidly. To further reduce the state space, we enrich the DP algorithm with specialized methods whose principles are inspired from branch-and-bound algorithms and constraint programming. We call

these methods *merging*, *bounding*, *reasoning*, *immediate selection*, and *heuristic*. More specifically, for any state $G = (V, A)$ of the decision tree, the *merging* procedure tries to merge same-class operations so that the graph G becomes smaller. The *bounding* method computes a lower bound for $f_n(G)$. The *reasoning* procedure tries to derive new precedence constraints. The *immediate selection* procedure tries to identify an optimal next class of the optimal value function (1) so that only this branch must be generated in the decision tree, and the *heuristic* method tries to find a new best feasible class sequence by heuristically generating a class sequence that removes the set of remaining operations V. The applied procedures are described in detail in the next subsections.

2.1 Merging

Given any state $G = (V, A)$ in the decision tree, we try to identify same-class operations that are executed at the same time in (at least) one optimal class sequence for the set of remaining operations V. If we can identify such a set of same-class operations, we can represent this set of operations by a single (pseudo-) operation. We only consider the case of merging pairs of operations. Technically, two operations v and w are merged into a single operation q in $G = (V, A)$ as follows. First, we add a vertex q to V. Then, for each vertex $p \in V \setminus \{v, w\}$, we add an arc (p, q) to A if $(p, v) \in A$ or $(p, w) \in A$, and an arc (q, p) if $(v, p) \in A$ or $(w, p) \in A$. Finally, we delete both v and w and all their incident arcs. The class c_q associated to operation q is $c_v = c_w$, and we keep track of this merging operation by storing the mapping of q to set $\{v, w\}$ to reconstruct the solution for the original instance at the end. Note that when checking for duplicated and dominated states, we compare the original non-executed operations.

For each operation $v \in V$, denote by $Pred(v)$ the set of different-class predecessors of v in graph G, i.e.,

$$Pred(v) = \{w \in V : c_w \neq c_v \text{ and } (w, v) \in \widehat{A}\},$$

recalling that $\widehat{G} = (V, \widehat{A})$ is the transitive closure of G. We establish the following first merging condition.

Proposition 2 *For any two distinct same-class vertices v and w, if the sets $Pred(v)$ and $Pred(w)$ contain exactly the same vertices, then they are executed at the same time in any feasible class sequence removing the remaining set of operations V.*

Proof. Given two distinct same-class vertices v and w with $Pred(v) = Pred(w)$, the two operations v and w become available for execution at the same time, independently of the class sequence. Hence, if we choose to execute class c_v at some point in time, we either execute both operations v and w or none of them. □

Considering our illustrative example of Figure 1, we observe that both white vertices 1 and 6 have no predecessors. Hence, they can be merged to a single vertex called 1; 6. The resulting graph is depicted in Figure 3 a). In this graph, we observe that vertices 2 and 10 can be merged as they have the same different-class predecessors, namely vertex 9. After this merging, we obtain the graph depicted in b). Similarly, we obtain c) after merging vertices 11 and 17 and d) after merging vertices 8 and 12.

The second merging condition is the following. If the sets of different-class predecessors and successors of a vertex include the corresponding sets of another same-class vertex, then they can be merged. More formally, for each vertex $v \in V$, denote by $Succ(v)$ the set of different-class successors of v in G, i.e.,

$$Succ(v) = \{w \in V : c_w \neq c_v \text{ and } (v, w) \in \widehat{A}\}.$$

Proposition 3 *For any two distinct same-class vertices v and w, if $Pred(v) \subseteq Pred(w)$ and $Succ(v) \subseteq Succ(w)$ then the two operations v and w are executed at the same time in some optimal class sequence for the remaining set of operations V.*

Proof. Given are two distinct same-class vertices v and w of G such that $Pred(v) \subseteq Pred(w)$ and $Succ(v) \subseteq Succ(w)$. Consider any feasible class sequence $S' = (c(1), \ldots, c(r))$ that removes all non-executed operations V of the given state $G = (V, A)$. Let i_v and i_w be such that v is removed by the execution of $c(i_v)$, and w is removed by the execution of $c(i_w)$. As $Pred(v) \subseteq Pred(w)$ holds, we have $i_v \leq i_w$. Suppose $i_v < i_w$ and let W be the set of successors of v of class c_v removed by the execution of a $c(k)$ with $k < i_w$. Also, let j be such that $i_v < j < i_w$ and $c(j) \neq c(i_v) = c(i_w)$. As $Succ(v) \subseteq Succ(w)$, there exists no path in G from a vertex in $\{v\} \cup W$ to any vertex removed by the execution of $c(j)$. The removal of the vertices in $\{v\} \cup W$ can therefore be delayed and performed together with the removal of w. The so-created class sequence S'' is feasible and its length is not larger than the length of S'. □

This type of merging is illustrated in our illustrative example of Figure 3 from d) to e), where operations 7 and 16 are merged. The only predecessor of vertex 7 in d) is vertex 1;6 which is one of the two predecessors of vertex 16, and both vertices 7 and 16 have no successors. Hence, we can merge these two vertices as $Pred(7) \subset Pred(16)$ and $Succ(7) = Succ(16)$ holds. Similarly, we observe in e) that vertices 5 and 11;17 can be merged. The resulting graph is depicted in f). The *merging* procedure is then stopped as there are no more merging opportunities detected.

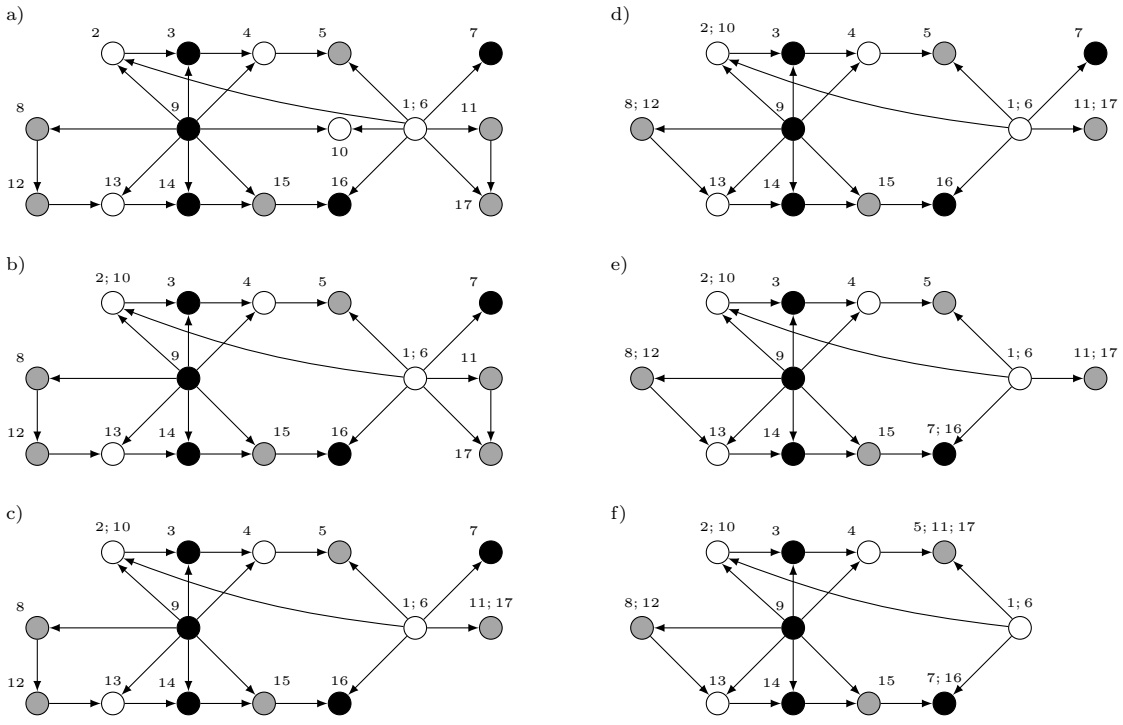

Figure 3: Six successive merging steps in our illustrative example.

2.2 Bounding

Given a state $G = (V, A)$ in some stage n in the decision tree, we compute a lower bound for this state according to (2). If we have a feasible class sequence at hand that is at least as good as a lower bound for state $G = (V, A)$, we can obviously delete this state in the decision tree.

Proposed by Lofgren et al. (1991), a bound can be obtained by the following *critical-path* bounding calculation. Let $G^+ = (V^+, A^+, d)$ be the graph obtained from $G = (V, A)$ by adding a source vertex σ, a sink vertex τ (i.e., $V^+ = V \cup \{\sigma, \tau\}$), an arc (σ, v) with length 1 and an arc (v, τ) with length 0 for all $v \in V$ (i.e., $A^+ = A \cup \{(\sigma, v), (v, \tau) : v \in V\}$). For each arc $(v, w) \in A$, we define its length d_{vw} as 1 if $c_v \neq c_w$, and 0 otherwise. The length d_{vw} reflects a class change between the execution of v and w, and the length of 1 for all arcs incident to the source accounts for the execution of the first class. We then calculate a longest path from σ to τ in G^+ with a topological sorting algorithm. Let $LP(G^+)$ be the length of such a longest path. This length can be used to compute a lower bound as the following holds.

Proposition 4 *$v_n(G) \geq LP(G^+)$ and $f_n(G) \geq n + LP(G^+)$.*

Proof. By construction of graph G^+, the length of any path from σ to τ is a lower bound on the number of remaining class executions to perform all operations of V, hence $v_n(G) \geq LP(G^+)$. If c is the last class executed before reaching state $G = (V, A)$, then no vertex of class c is available in G. Hence, given the partial class sequence of length n obtained for reaching state $G = (V, A)$ in the decision tree, a lower bound on the length of a feasible class sequence extending this state is $n + LP(G^+)$ and $f_n(G) \geq n + LP(G^+)$ follows. □

Consider again the graph depicted in Figure 3 f), which was obtained by applying the *merging* procedure to our illustrative example. The critical-path bounding calculation gives $\{\sigma, 9, 8; 12, 13, 14, 15, 7; 16, \tau\}$ as vertex set of a longest path from σ to τ in G^+ with length 6. Since we are in stage 0, the lower bound is 6.

We now propose a stronger lower bounding procedure obtained by determining a bound on the number r_c of remaining class executions needed to remove all vertices of each class $c \in \mathcal{C}$. We call this the *one-class* bounding procedure. Each value r_c is calculated in graph $G_c^+ = (V^+, A^+, d^c)$ obtained from G^+ by changing the arc lengths as follows. For each arc $(v, w) \in A$, we set d^c_{vw} to 1 if $c_v \neq c$ and $c_w = c$, and 0 otherwise. Also, for each vertex $v \in V$, we set $d^c_{\sigma v}$ to 1 if $c_v = c$ and 0 otherwise. These arc weights reflect when changing from any other class to class c in a class sequence. Then, for each class $c \in \mathcal{C}$, we calculate a longest path from σ to τ in G_c^+ using a topological sorting algorithm. Let $r_c(G)$ be the length of such a longest path. The obtained values can be used to compute a lower bound as the following holds.

Proposition 5 *$v_n(G) \geq \sum_{c \in \mathcal{C}} r_c(G)$ and $f_n(G) \geq n + \sum_{c \in \mathcal{C}} r_c(G)$.*

Proof. For any class $c \in \mathcal{C}$, the length of any path from σ to τ in G_c^+ is a lower bound on the number of remaining class executions for class c by construction of G_c^+. Hence, $v_n(G) \geq \sum_{c \in \mathcal{C}} r_c(G)$ and, as in Proposition 4, $f_n(G) \geq n + \sum_{c \in \mathcal{C}} r_c(G)$ follows. □

For the graph depicted in Figure 3 f), we obtain a one-class lower bound of 7. Indeed, we get a lower bound of 2 for the white vertices, 2 for the gray ones, and 3 for the black ones. Since we are in stage 0, $n = 0$ and the overall lower bound is $2 + 2 + 3 = 7$.

It can easily be proved that the one-class lower bound is at least as strong as the critical-path bound. However, the computational effort of the one-class lower bound is $|\mathcal{C}|$-times higher. We analyze this computational trade-off in Section 3.

Note that for any stage n in the decision tree, the bounding procedure also provides a global lower bound simply obtained by taking the lowest bound value over all states of this stage.

2.3 Reasoning

Given a state $G = (V, A)$ in some stage n in the decision tree, let S be the partial class sequence that corresponds to the decisions that led to $G = (V, A)$ in the decision tree. Suppose that we already

found a feasible class sequence with length UB. We try to find additional precedence constraints that must be satisfied by any feasible class sequence extending S that is of length at most $UB-1$. Such constraints can be added as we already have a solution with objective value UB at hand, so that we can restrict the search space to all solutions with objective values at most $UB-1$. We apply for this the *bounding* procedure introduced before as follows.

For each pair v, w of distinct vertices in G such that $(v,w) \notin \widehat{A}$ and $(w,v) \notin \widehat{A}$, we check with the *bounding* procedure if we can obtain a lower bound of at least UB after adding precedence constraint (v,w) to A. If the answer is yes, w is executed not later than v in any feasible class sequence with length at most $UB-1$. Therefore, (w,v) can be added to A, which forbids to perform w later than v. Clearly, operations v and w are still allowed to be performed at the same time if both are from the same class. Similarly, if we obtain a lower bound of at least UB after adding precedence constraint (w,v) to A, then we add (v,w) to A to avoid w being removed before v. Once no more arcs can be added to A, we update G to its transitive reduction (see, e.g., Aho et al., 1972). This helps to reduce the computational effort when applying topological sorting algorithms in G (see the *bounding* procedure).

Consider again the graph depicted in Figure 3 f), and assume that we already found a feasible class sequence of length 8 so that $UB = 8$. We check if we can add the precedence constraint $(1;6, 14)$. For this purpose we temporarily add the opposite arc $(14, 1;6)$ to A and calculate the one-class lower bound. We get 3 for the white vertices, 2 for the gray ones, and 3 for the black ones, so that the overall lower bound is 8. As this is not lower than UB, we add $(1;6, 14)$ to A. The *reasoning* procedure then sequentially adds arcs $(2;10, 14)$, $(3, 15)$, $(4, 15)$, $(5;11;17, 7;16)$, $(8;12, 3)$, $(9, 1;6)$, $(13, 3)$, $(14, 4)$, and $(8;12, 1;6)$ to A in our example, obtaining the graph depicted in Figure 4.

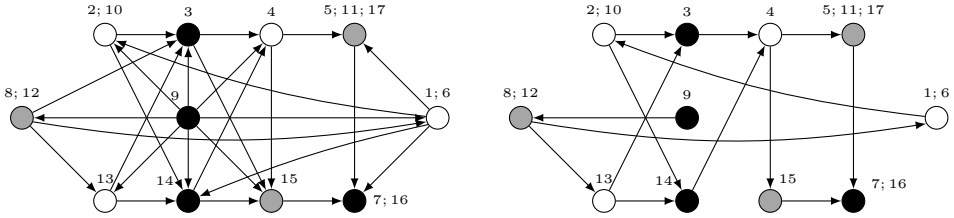

Figure 4: Output of the *reasoning* procedure when applied to the graph of Figure 3 f). Resulting graph before and after applying the transitive reduction on the left and right, respectively.

We consider two versions of the *reasoning* procedure. In the first version, called *one-pass reasoning*, we stop after checking each pair v, w of operations. If we added at least one additional arc in this pass, we might find new precedence constraints when checking all pairs in another pass. Hence, in the second version, called *deep reasoning*, we continue to check all pairs of operations until no further precedence is detected in a pass.

The reasoning procedure is quite time consuming as we run the *bounding* procedure for each pair of vertices $(v,w) \notin \widehat{A}$ and $(w,v) \notin \widehat{A}$. In Section 3, we analyze the effect of the reasoning procedure by comparing three versions: no reasoning, one-pass reasoning, and deep reasoning.

2.4 Immediate selection

Given a state $G = (V, A)$ in the decision tree, we try to identify an optimal next class of the optimal value function (1) so that only this branch must be generated in the decision tree.

In the following two cases, we can find an optimal next class. First, if there is only one available class c, then c is trivially an optimal next choice. Second, if for some available class c, its set of available operations cannot be extended by postponing the execution of c, then c is an optimal next class as there is no gain in further waiting for its execution. Such a situation can be detected as follows.

Let V_c^{av} be the set of available operations of an available class c, and let $V_c^{\text{nav}} = \{v \in V : c_v = c\} \setminus V_c^{\text{av}}$ be the set of non-available operations of class c. Consider any vertex $u \in V_c^{\text{nav}}$. We call u as being *blocked* if there exists a path P from a vertex $v \in V_c^{\text{av}}$ to u in G such that at least one vertex on P, say p, is of a different class $c_p \neq c$. Clearly, u cannot be removed together with the vertices of V_c^{av} in a class execution. Indeed, since there is a path from v to p and $c_p \neq c_v$, p will be removed after v and, similarly, u will be removed after p in any feasible class sequence. Hence, u will never be available when v will be removed together with all vertices of V_c^{av} (and possibly other vertices). We therefore conclude the following.

Proposition 6 *If all vertices u in V_c^{nav} are blocked for some class $c \in \mathcal{C}$, then the set of available operations of c cannot be extended.*

Consider again our illustrative example in Figure 4. Only the black class is available, and it contains only one available operation, namely vertex 9. Note that this class cannot be extended since the paths $(9, 8;12, 13, 3)$, $(9, 8;12, 13, 14)$ and $(9, 8;12, 13, 14, 4, 15, 7;16)$ connect vertex 9 to the three other black vertices 3, 14 and 7;16, respectively, and they all contains the gray vertex 8;12. Hence, there are two reasons why black is an optimal next choice: it is the only available class, and it is not extendable. After deleting vertex 9 from G, we observe that the gray class can be executed as it is the only available class. Subsequently, the *immediate selection* procedure sequentially detects the optimal classes white, black, white, gray, black, obtaining an optimal class sequence with length 7 for the given instance without branching in the decision tree.

2.5 Heuristics

To discard states with the *bounding* method and to successfully apply the *reasoning* procedure, it is important to determine good feasible solutions. This is the task of the *heuristic* procedure, which works as follows.

Given an upper bound UB on the length of an optimal class sequence and a state $G = (V, A)$ in some stage n in the decision tree, let S be the partial class sequence that corresponds to the decisions that led to $G = (V, A)$ in the decision tree. We try to extend S to a feasible class sequence of length at most $UB - 1$. For this purpose, we apply the *merging* procedure and then try to find a next class by the *immediate selection* procedure. If this procedure finds an optimal next class, we directly execute it. Otherwise, we determine the next class by a heuristic *rule*. We restart these steps until V contains no more operations. Algorithm 1 contains a pseudo-code of this generic constructive heuristic.

Algorithm 1: A generic heuristic for PCCSP.

```
Copy graph G = (V, A) from the current state and work on this copy; Set S' equal to S;
while S' is not feasible and |S'| < UB do
    apply merging to G = (V, A) state;
    apply immediate selection to G = (V, A) and update S';
    if no class was executed in this immediate selection procedure then
        select an available class with some rule, execute it, and update S';
end
if |S'| < UB then return S';
else return null;
```

Clearly, the performance of this heuristic is mainly determined by the *rule* that selects a class among all available ones. We consider the following simple rules, which are introduced by Lofgren et al. (1991):

Random Randomly choose an available class.

Greedy Choose an available class with a largest number of available operations. We hereby count the number of available operations with respect to the original problem. If, for example, some operation obtained through merging steps is available, we count the number of operations it corresponds to in the original instance.

Altruistic Choose an available class that would, if executed, make the largest number of new operations available. Again, we count the number of newly available operations with respect to the original instance.

Critical path level For each available operation $v \in V$, determine the length l_v of a longest path from v to τ in G^+ (see Section 2.2 for the details). An available vertex v is called critical if l_v is equal to $LP(G^+) - 1$, and a class is called critical if it contains at least one critical vertex. We then choose a critical class. Hereby, we apply one of the three rules described before, i.e., random, greedy, or altruistic, for breaking ties, which gives three different heuristics for this rule.

For the input graph of Figure 1 and an (unrestrictive) upper bound of $UB = |V| + 1$, the heuristic applied with the altruistic and critical path level rules finds the same optimal sequence of length 7 that we found at the end of Section 2.4, while it provides a class sequence of length 8 with the greedy rule.

In Section 3.2.1 we conduct computational tests to decide which version of the heuristic to include in the DP algorithm.

2.6 The dynamic programming algorithm for PCCSP

We now describe the technical implementation of the DP algorithm in more detail using the pseudocode given in Algorithm 2. Basically, we explore the state space of the dynamic program stage-wise, i.e., given a stage of the decision tree, we determine all states of the succeeding stage, and then continue with this stage. In each state $G = (V, A)$, we additionally store all decisions taken in the preceding stages, giving a partial class sequence S for each state.

As first step, given an instance $(V, A, \mathcal{C}, C)$, we generate the root state (at stage 0) of the decision tree consisting of the precedence graph $G = (V, A)$. Its corresponding partial class sequence S is empty. Variables *bestSolution* and *lowerBound* refer to the best class sequence found so far and to the current lower bound, respectively.

Before starting the actual decision tree exploration, we try to simplify the root state by sequentially applying the procedures *merging*, *heuristic*, *reasoning*, *bounding* and *immediate selection*, until the state is structurally not changed anymore, i.e., no further precedences or immediate selections are detected (lines 2 to 8). Note that if the procedure *immediate selection* finds a class, we directly execute it without generating a new state in the decision tree. In order to conform with (2), we can just think of increasing the stage count by 1.

If the lower bound obtained in this preprocessing step is not strictly smaller than the objective value of *bestSolution*, an optimal sequence is found and we can stop the search. Otherwise, we start the tree exploration. List *currentStates* contains all states of the current stage in the decision tree.

The next-stage states are generated as follows (lines 13 to 39). We first initialize list *nextStageStates* to empty. For each state in *currentStates*, we try to find an optimal next class with the *immediate selection* procedure. If we find one, we execute this class and add the state to the list *nextStageStates* (lines 15 to 17). Else, we branch on the available classes (lines 19 to 23). For each available class, we copy the current state, execute the class in the copied state, update sequence S and add the resulting state to list *nextStageStates*. We then delete duplicated and dominated states from this list (line 25).

To possibly find a new best solution, we then apply the *heuristic* procedure to each state in list *nextStageStates* and update *bestSolution* if some new best sequence is found (lines 26 to 27). For each state in list *nextStageStates*, after applying the *merging* and *reasoning* procedures (line 31), we compute a lower bound for the state with the *bounding* procedure (line 32). If the obtained bound is larger than or equal to the objective value of the best solution found so far, we delete the state from list *nextStageStates* (lines 33 to 34). The overall lower bound *lowerBound* is then set to the lowest lower bound of all states kept in list *nextStageStates* (lines 47 to 48). Finally, we update the list *currentStates* to *nextStageStates* (line 39).

Algorithm 2: The DP algorithm.

```
1  Generate the root state for stage 0 with the input (V, A, 𝒞, C), add an empty sequence S to this state, set
     bestSolution to empty and lowerBound to 0;
2  repeat
3  |  apply merging to state;
4  |  apply heuristic to state and update bestSolution;
5  |  apply reasoning to state;
6  |  apply bounding to state and update lowerBound;
7  |  apply immediate selection to state and update its sequence S;
8  until state did not change;
9  set currentStates to empty;
10 if lowerBound is strictly smaller than the objective value of bestSolution then
11 |  add root note to currentStates;
12 while currentStates is not empty do
13 |  set nextStageStates to empty;
14 |  foreach state in currentStates do
15 |  |  apply immediate selection to state and update its sequence S;
16 |  |  if a class was executed in this procedure then
17 |  |  |  add state to nextStageStates;
18 |  |  else
19 |  |  |  foreach available class of state do
20 |  |  |  |  copy the current state;
21 |  |  |  |  execute the class in the copied state and update its sequence S;
22 |  |  |  |  add the resulting state to nextStageStates;
23 |  |  |  end
24 |  end
25 |  delete duplicated and dominated states in nextStageStates;
26 |  foreach state in nextStageStates do
27 |  |  apply heuristic to state and update bestSolution;
28 |  end
29 |  initialize lowerBound to objective value of bestSolution;
30 |  foreach state in nextStageStates do
31 |  |  apply merging and reasoning to state;
32 |  |  apply bounding to state, obtaining lowerBoundOfState;
33 |  |  if lowerBoundOfState is not strictly smaller than the objective value of bestSolution then
34 |  |  |  delete state from nextStageStates;
35 |  |  else
36 |  |  |  if lowerBoundOfState is strictly smaller than lowerBound then
37 |  |  |  |  set lowerBound to lowerBoundOfState;
38 |  end
39 |  set currentStates to nextStageStates;
40 end
```

The search stops if the list *currentStates* becomes empty (line 12), which means that no more states need to be explored and *bestSolution* contains an optimal class sequence.

3 Computational results

The DP algorithm is implemented in Java using only its core class libraries and run on a PC with an Intel Core i5-7500 3.4 GHz processor (4 cores) and a maximum of 30 GB memory allocated to the Java Virtual Machine. To evaluate the performance of the developed DP algorithm and its sub-procedures, we test them on benchmark instances proposed by Lofgren et al. (1991) for routing printed circuit boards through an assembly cell.

In this section, we first describe the instance generation scheme, evaluate then the sub-procedures of the dynamic program, and finally compare the dynamic program with integer programming and constraints programming models.

3.1 Instance generation

The application and the benchmark instances we consider can briefly be described as follows. We refer to Lofgren et al. (1991) for the details. Given are a flexible assembly cell with W workstations and a rectangular board of size (X, Y) where components are placed at the integer points of an X times Y grid. At each workstation, a given set of components are available, and each component is present at exactly one workstation. In PCCSP, each workstation is represented by a class, each component on the board is an operation, and the class of an operation is given by the workstation where the component is available. The goal is to determine a complete ordering of the component assembly operations so that the number of times the board has to be moved from one workstation to another is minimized, which is the same as minimizing the number of setups in PCCSP.

There are three different sizes of components: large, medium, and small. Parameter L refers to the number of large components. Any non-large component is either small with a probability P, or medium otherwise. The small and medium components occupy only one grid point while large components take three consecutive grid points, either vertically or horizontally. Every grid point is used by exactly one component. Any component can be placed at any location, except that two large components cannot be adjacent to each other.

The precedence relations are derived as follows. Each large component has precedence over all components around its perimeter (i.e., left, right, above, and below), each medium component has precedence over any medium or small component to its right and over any medium component below it, and each small component has precedence over any medium or small component to its right.

Inspired by Lofgren et al. (1991), the parameters are set as follows in our benchmark set. We consider small boards of size $(20, 10)$ and large boards of size $(30, 15)$. The number of workstations W belongs to $\{3, 4, 5, 7\}$ and the components are randomly assigned to the workstations. The number L of large components is in $\{0, 4, 8, 12\}$ and the probability P in $\{0.5, 0.7, 0.9\}$. For every parameter setting, we randomly generate three instances, which gives a total of $4 \times 4 \times 3 \times 3 = 144$ small instances with board size $(20, 10)$, and 144 large instances with board size $(30, 15)$. For illustration purpose, we depict a small instance in Figure 5.

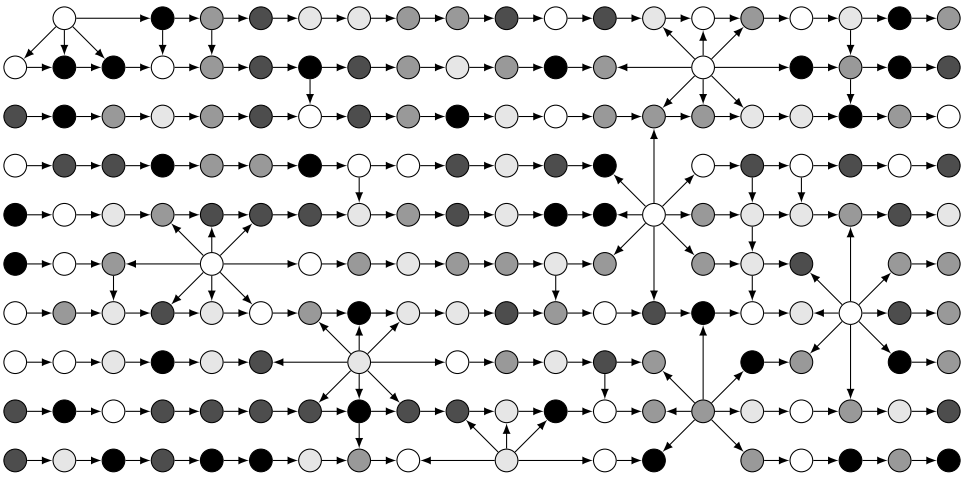

Figure 5: A small instance with $W = 5$, $L = 8$ and $P = 0.7$. Each workstation is represented by a color, and each component is depicted by a vertex having the color of the workstation to which it is assigned. Each arc represents a precedence relation.

3.2 Computational tests

We first consider the six different versions of the general heuristic described in Section 2.5 to decide which versions to include in the algorithm. For this purpose, we first run each version of the heuristic

on all 288 benchmark instances (using an unrestrictive upper bound). The analysis of the results is presented in Section 3.2.1.

We then consider different versions of the dynamic program. We define the standard DP version to be composed of the merging procedure, the one-pass reasoning method, the one-class bounding method, the immediate selection procedure, and the choice of the heuristics presented in Section 3.2.1. To analyze the importance of each component, we change the standard version as follows:

- no-merging: standard version without the merging method,
- no-reasoning: standard version without the reasoning method,
- deep-reasoning: standard version with the deep reasoning procedure instead of one-pass reasoning,
- critical-path bounding: standard version with critical-path bounding method instead of one-class bounding
- no-immediate-selection: standard version without immediate selection method.

We run each of these DP versions for all 288 benchmark instances using a computation time limit of 7200 seconds (2 hours) per run. We record the best solution found, the final lower bound as well as the number of states explored in the decision tree. Detailed results are provided in the electronic supplementary file. In what follows, we discuss these results.

3.2.1 Analysis of the heuristics

We begin with an analysis of the results obtained with the six different versions of the heuristic: random, greedy, altruistic, and critical-path-level with random (CPL-random), greedy (CPL-greedy), and altruistic (CPL-altruistic) tie breaking strategy. For each version and instance, we compute the relative percent deviation (RPD) of the attained objective value (res) to the lowest objective value ($best$) over all versions, i.e., RPD $= 100(res - best)/best$. The left part of Figure 6 illustrates the RPDs by showing the minimum, first quartile, median, third quartile, and maximum RPDs in box-and-whisker plots, using standard conventions for drawing such plots. Furthermore, for each version of the heuristic, we show the portion of instances for which a best solution is found with this heuristic on the right part of Figure 6. The following can be observed.

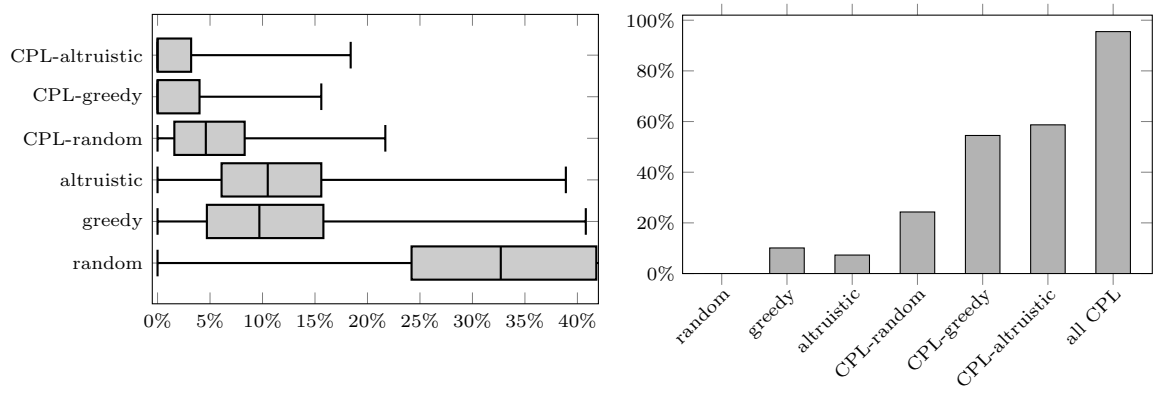

Figure 6: Left: box-and-whisker plots of the RPDs attained with the different versions of the generic heuristic. Right: Histogram showing the portion of the instances (in %) for which a best solution is found with the different versions. The last bar "all CPL" indicates the portion of the instances for which at least one of the CPL-based heuristics finds a best solution.

As expected, the random rule, which is mainly introduced for comparison purposes, is outperformed by all other rules. The performance of the greedy and altruistic rules are similar (i.e., their box-and-whisker plots have a similar shape). A substantially better performance is obtained with the three critical-path-level rules. It can be seen that CPL-greedy and CPL-altruistic find a best solution in

more than 50% of the instances. As expected, these two versions perform slightly better than CPL-random. While the CPL-random, CPL-greedy, and CPL-altruistic heuristics find a best solution in 24.3%, 54.5%, and 58.7% of the instances, respectively, at least one of the three versions finds a best solution in 95.5% of the instances. Since the heuristics only need milliseconds to be executed on the test instances, we decide to run all three CPL-versions when applying the *heuristic* procedure in the DP algorithm.

3.2.2 Analysis of the standard DP version

We now discuss the results obtained with the standard DP version. For this purpose, we illustrate some statistics in Figure 7 as follows. We present results for the small and large instances on the left and right, respectively. For the instances solved to proven optimality, called *solved instances* for short, we consider the computation time (in seconds), the number of explored states in the decision tree and the improvement of the upper bound (UB) and lower bound (LB) with respect to the corresponding bound obtained in the initial (root) state of the decision tree. These values are depicted in compact form with box-and-whisker plots. For the instances that are not solved to proven optimality, called *unsolved instances* for short, we similarly depict the relative optimality gaps at the end of the search, computed as (UB-LB) / LB, the numbers of explored states, and, as before, the improvements of the upper and lower bounds with respect to the initial bounds. To further analyze the behavior of the standard DP version with respect to the instance characteristics, we show the number of solved instances and their computation times for some interesting subsets of instances in Figure 8. For example, the first box-and-whisker plot on the left of this figure shows the computation times of the 27 small instances with 7 classes that could be solved to optimality. The following observations can be made.

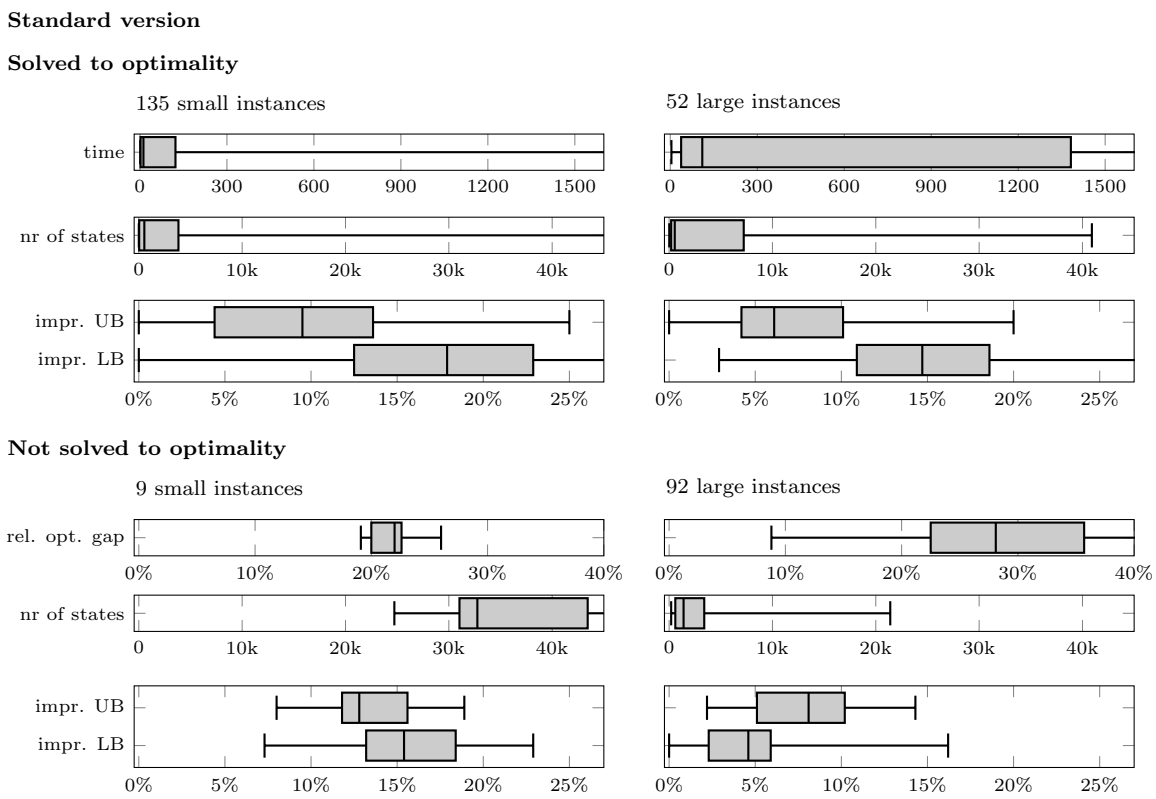

Figure 7: Analysis of the standard version of DP.

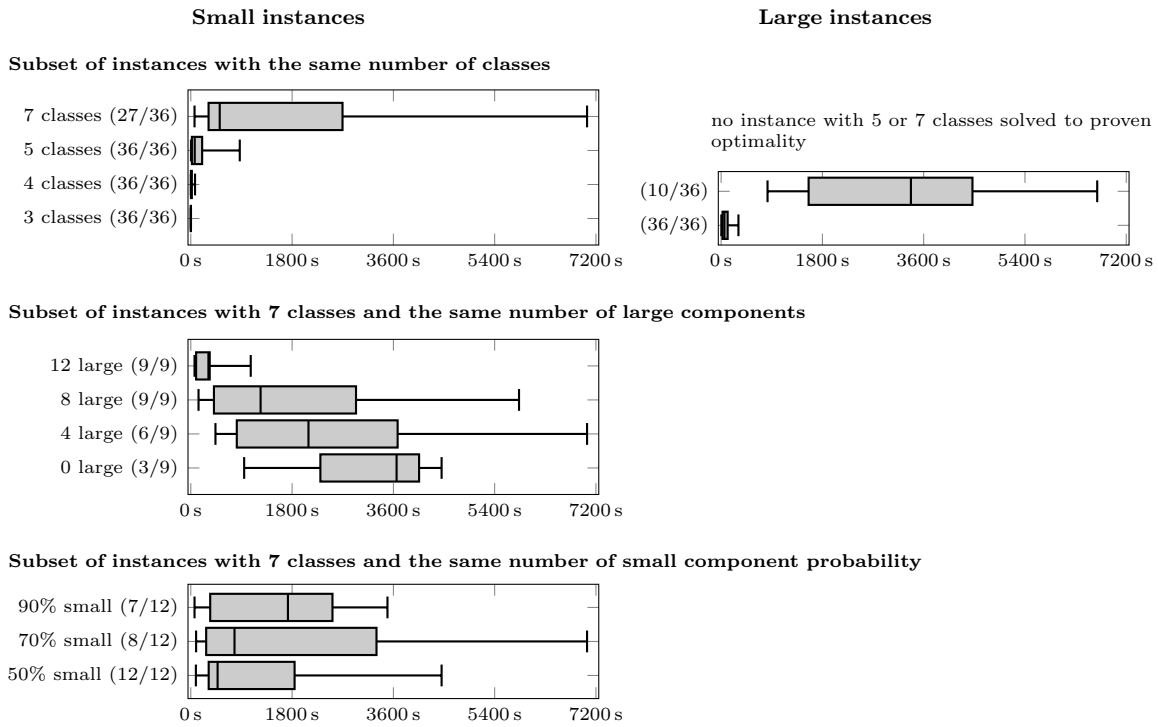

Figure 8: Box-and-whisker plots of the computation times attained for the solved instances of some subsets of instances. In round brackets, we indicate the number of solved instances and the total number of instances for each subset.

The large majority of the small instances could be solved to optimality while this is only the case for about 1/3 of the large instances. Moreover, all unsolved small instances have 7 classes, the largest amount among all instances. Hence, unsurprisingly, the number of operations and the number of classes are main determinants of the difficulty of an instance in our benchmark set.

Looking at the small solved instances, we see that the computation time is typically less than 150 s and the number of states less than 5000. We also see that the gap between the optimal value and the initial upper bound is about 4% to 15% in most instances while the gap between the optimal value and the initial lower bound is about 12% to 22%. All of these numbers are quite low meaning that the optimality gap at the initial state is relatively small and the algorithm only needs a short amount of time and a small number of states in the decision tree until the gap is closed. One can also see that the lower bound is typically slightly further away from the optimum than the upper bound. Similar observations can be made for the large solved instances. The main difference is the computation time, which varies considerably for these instances.

Considering the unsolved small instances, we observe that the relative optimality gaps are substantial, they are typically about 22%, although a large amount of states is explored (often more than 30 000). As already mentioned, all unsolved small instances have 7 classes. We further see in Figure 8 that the most difficult 7-class instances are those with few large components. The large components help in two ways to make the problem simpler. First, they imply many precedence constraints, and second, the total number of components is then smaller. Similarly, instances with more medium-sized components are simpler as medium-sized components also imply more precedence constraints than the small components.

When looking at the unsolved, large instances, one can see the effect of the number of classes. Indeed, no instance with 5 or 7 classes could be solved to proven optimality. Moreover, the relative optimality gaps of the unsolved instances are quite large: they are typically between 22% and 37%.

3.2.3 Analysis of merging

We now analyze the impact of the merging procedure by comparing the no-merging version with the standard version. Figure 9 shows the following descriptive statistics for the small and large instances on the left and right, respectively. For each instance solved to proven optimality with both versions, we compute the computation time ratio, i.e., comp. time no-merging / comp. time standard version, and similarly the ratio of the number of states. For each other instance, we consider the RPD of the upper bound, i.e., (UB no-merging − UB standard) / UB standard, and the RPD of the lower bound, i.e., (LB standard − LB no-merging) / UB standard. Note that no-merging is at an advantage given these definitions if the ratios are below 1 and the RPDs below 0. Figure 9 depicts the calculated numbers in compact form with box-and-whisker plots. In addition, we mention in this figure the number of small and large instances that are solved to proven optimality with both versions, only with one version, or remain unsolved. Considering the small instances, for example, 123 are solved to optimality with both versions, 12 only with the standard version, and 9 remain unsolved. Note that the structure of Figure 9 is used for the other comparisons in the sequel.

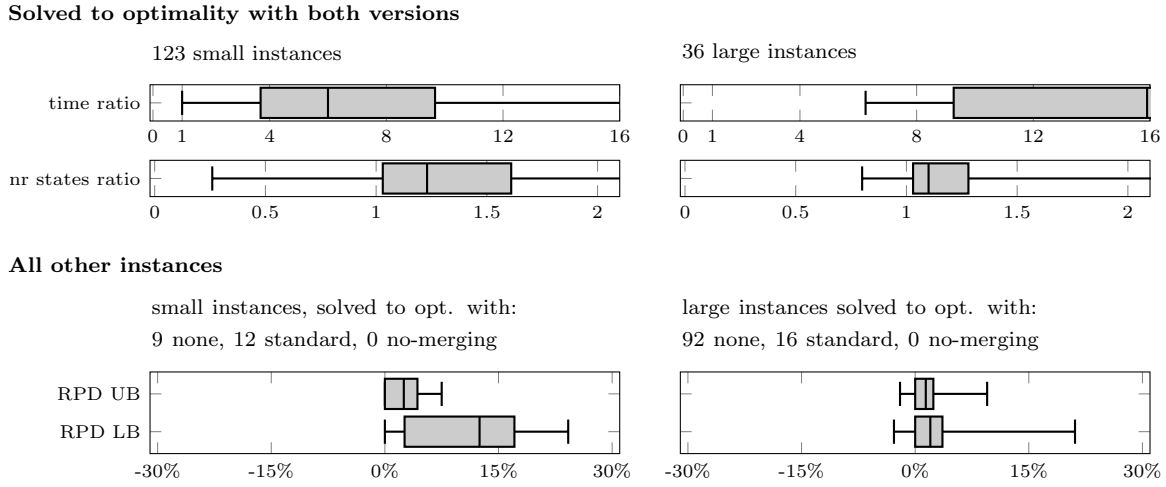

Figure 9: Comparison of the no-merging version with the standard DP version.

It is apparent that the merging procedure has a substantial positive effect on the performance of DP. When switching this procedure off, less instances are solved to optimality, the computation time is substantially higher for the instances that both versions solved to optimality, and the lower and upper bounds are worse in the other instances.

We remark that the positive effect of merging mainly stems from the additional precedence constraints that are introduced when merging on the one hand, leading to a lower number of states, and by the reduced size of the graphs on the other hand, leading to a faster processing of the states. Also, the computation time needed to execute the merging procedure is negligible (typically about one percent of the total computation time).

3.2.4 Analysis of bounding

We analyze the impact of the newly developed one-class bounding method. This is particularly interesting since the large majority of the computation time (often more than 90% according to our tests) is spent for deriving bounds. We compare the results of the standard DP version with those of the version where the lower bounds are obtained with critical-path bounding calculations (see Section 2.2 for the details). The results of this comparison are depicted in Figure 10.

Looking at the instances solved to optimality with both versions, we observe that the version with critical-path bounding needs slightly more computation time and explores considerably more states.

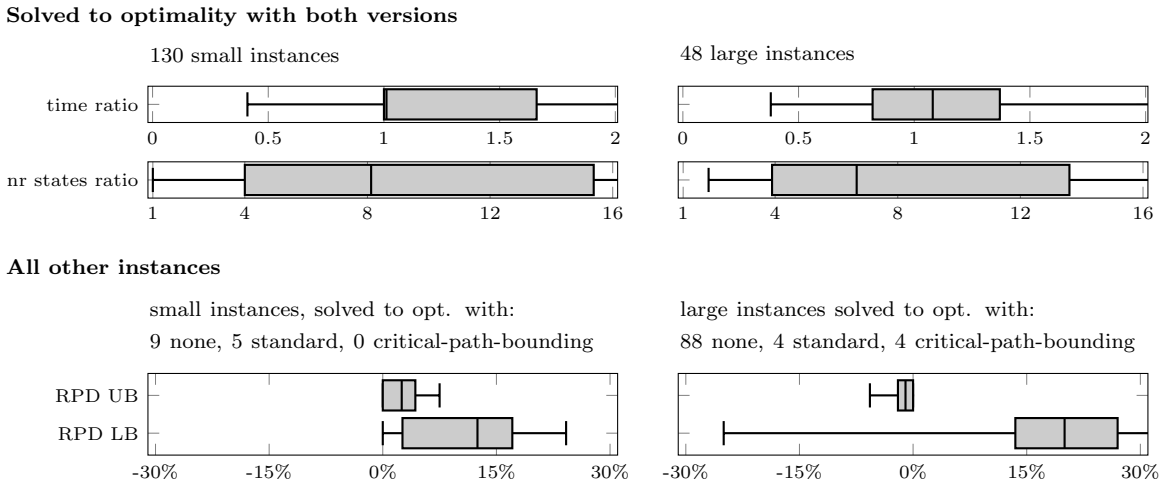

Figure 10: Comparison of the critical-path bounding version with the standard DP version.

This is consistent with our expectations. The critical-path bounds are weaker than the one-class bounds, which is a disadvantage when pruning the states. In contrast, it is faster as only one longest path calculation must be executed in the graph while the number of longest path calculations equals the number of classes in the one-class bounding method. As a results, for the solved instances, the standard DP version is slightly better in terms of computation time and considerably better in terms of explored states. The advantage of the new bounding method can further be seen in the unsolved instances. The lower bounds are substantially better with the one-class bounds. Therefore, we conclude that one-class bounding is an important ingredient of our DP.

3.2.5 Analysis of reasoning

We now analyze the impact of the reasoning procedure. For this purpose, we first compare the no-reasoning version with the standard DP version. The results of this comparison are depicted in Figure 11.

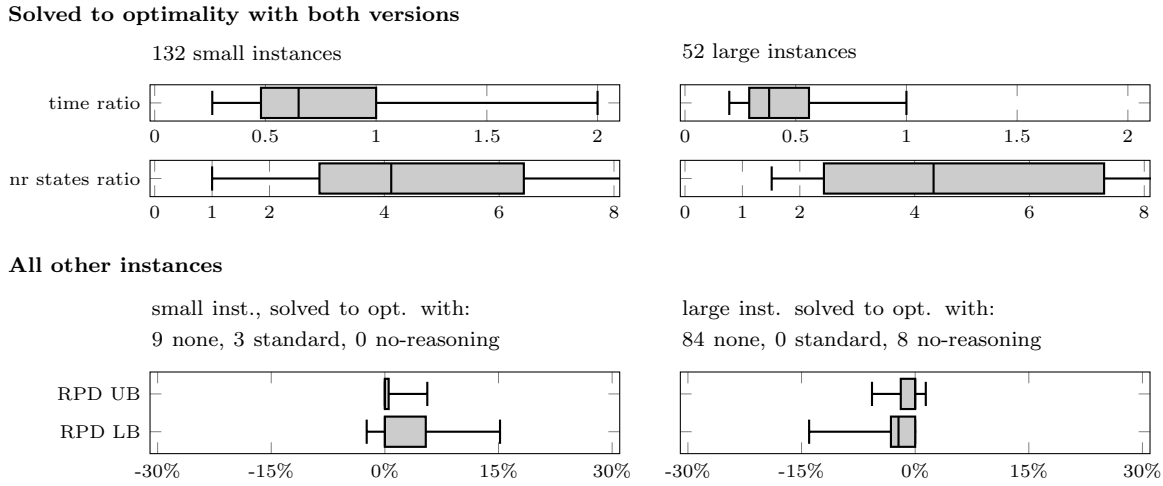

Figure 11: Comparison of the no-reasoning version with the standard DP version.

When looking at the time ratios, we observe that the no-reasoning version is definitively faster; In the median case, it is about 35% and 60% faster for the solved, small and large instances, respectively, and it is rarely slower. For the other, unsolved instances, both versions get similar lower and upper

bounds. The standard version is, however, at an advantage when looking at the number of states. Indeed, one-pass reasoning can reduce the number of explored states typically by a factor 2 to 6, which translates into huge savings with respect to memory usage. Briefly, when compared to no-reasoning, one-pass reasoning enables to keep the number of states considerably smaller while slightly increasing the overall computation time.

We compare the deep-reasoning version with the standard version in Figure 12. We observe that deep-reasoning reduces the number of explored states considerably, but the overall time needed to solve the instances increases and the lower bounds of the unsolved instances are substantially weaker.

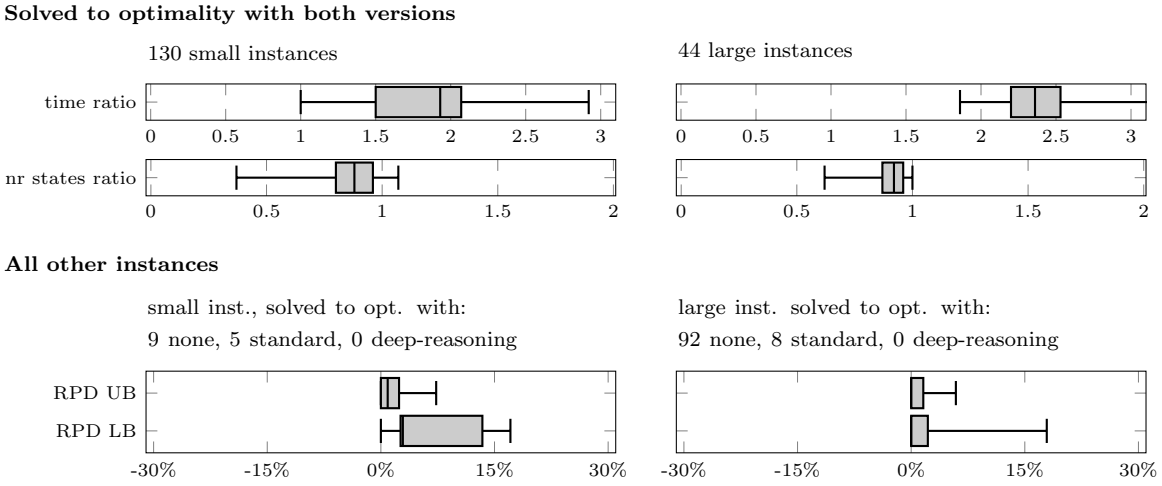

Figure 12: Comparison of the deep-reasoning version with the standard DP version.

Altogether we conclude that the reasoning method can be switched off without sacrificing the quality of the DP algorithm. However, one-pass reasoning might be a good choice in terms of time-memory trade-off. We therefore think that the standard DP version should be given a slight preference over the no-reasoning version.

3.2.6 Analysis of immediate selection

We look at the impact of the immediate selection procedure. For this purpose, we compare the results of the no-immediate-selection version with those of the standard DP version. The obtained results are illustrated in Figure 13.

We observe that, for the solved instances, the differences of the computation times and the number of explored states are small. More specifically, the standard version has a slight edge over the other version with respect to the computation time: it is typically faster by about 2% to 11%. The number of states is almost unchanged. For the unsolved instances, the standard version again has a slight edge over the no-immediate-selection version: without immediate selection, the lower and upper bounds often increase by about 1% to 7%. We conclude that immediate selection has a small but positive effect on the performance.

3.3 Comparison of the dynamic program with other methods

As no comprehensive computational results are available for PCCSP, we decide to compare our DP algorithm, for which we take the standard version, with two integer linear programming (IP) models of PCCSP and a constraint programming (CP) model. The first IP model is similar to standard continuous-time formulations of machine scheduling problems with sequence-dependent setup times, the second IP model is similar to discrete-time formulations of machine scheduling problems, and the

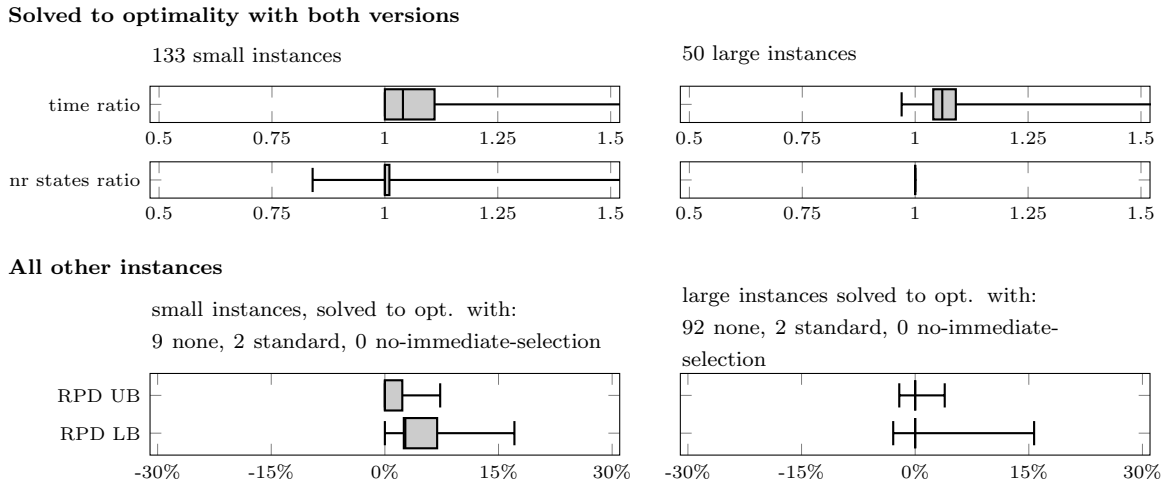

Figure 13: Comparison of the no-immediate-selection version with the standard DP version.

CP model is also a standard formulation for one-machine scheduling problems with sequence-dependent setup times. We use Gurobi 7.5 as an IP solver and IBM ILOG CP Optimizer 12.8 as a CP solver. The computation time is limited to 7200 s per run and we execute one run for each instance and IP / CP model. We record the final lower bound, upper bound and computation time for each run. Detailed results can be found in Table 1 and 2 of the Appendix. In this section, we first specify the IP and CP models and then analyze the obtained results.

3.3.1 Continuous-time IP model

The following continuous-time IP model is used to solve PCCSP. For each pair $(i, j) \in V \times V$ of distinct operations, introduce a binary variable y_{ij}, which takes value 1 if i is executed before j, and 0 otherwise. To count the number of setups, introduce an integer variable x_i for each operation $i \in V \cup \{\tau\}$ reflecting the number of setups necessary before starting the execution of operation i. Hereby, τ is a fictive end operation executed after all other operations. For each pair of operations $i, j \in V$, parameter s_{ij} indicates if a setup is needed when executing j directly after i. Therefore, set $s_{ij} = 1$ if $c_i \neq c_j$, and 0 otherwise. Also, let U be an upper bound on the optimal total number of setups. In our computational tests, we set U to the objective value of a solution obtained by the CP-greedy heuristic (see Section 2.5). PCCSP can then be described by the following integer linear program, called IP1:

$$\text{Minimize } x_\tau \tag{4a}$$

subject to

$$y_{ij} + y_{ji} = 1 \qquad \text{for all distinct } i, j \in V \tag{4b}$$

$$x_j - x_i - (s_{ij} + U)y_{ij} \geq -U \qquad \text{for all distinct } i, j \in V \tag{4c}$$

$$x_i - x_j - (s_{ji} + U)y_{ji} \geq -U \qquad \text{for all distinct } i, j \in V \tag{4d}$$

$$y_{ij} = 1 \qquad \text{for all } (i, j) \in A \tag{4e}$$

$$x_\tau \leq U \tag{4f}$$

$$x_\tau \geq x_i \qquad \text{for all } i \in V \tag{4g}$$

$$y_{ij} \in \{0, 1\} \qquad \text{for all distinct } i, j \in V \tag{4h}$$

$$x_i \in \mathbb{Z}_{\geq 0} \qquad \text{for all } i \in V \cup \{\tau\}. \tag{4i}$$

Constraints (4b) ensure that all pairs of operations are sequenced. Constraints (4c) and (4d) link the sequencing variables y with the setup counting variables x. Indeed, if i is executed before j, constraints (4c) impose $x_j \geq x_i + s_{ij}$ and constraints (4d) impose $x_j \leq x_i + U$ (which is not restrictive),

while $x_i \geq x_j + s_{ji}$ and $x_i \leq x_j + U$ are imposed if j is executed before i. Constraints (4e) specify that the precedences given in set A must be met. Constraints (4f) and (4g) make sure that the end operation τ is executed after all other operations and that there are at most U setups. Constraints (4h) tell that the variables y are binary and constraints (4i) specify that the variables x are non-negative integers. Clearly, x could also be specified as continuous variables. Finally, the objective is to minimize the total number of setups, which is reflected in (4a).

3.3.2 Discrete-time IP model

The second model is similar to typical discrete-time model of machine scheduling problems. For each operation $i \in V$ and each $p \in \{1, \ldots, U\}$, introduce a variable x_{ip}, which takes value 1 if there are exactly p setups performed before executing operation i, and 0 otherwise. Hereby, U is the same upper bound on the total number of setups as the bound used in the previous model. Introduce a variable z reflecting the total number of setups. Then the problem can be described by the following integer linear program, called IP2:

$$\text{Minimize } z \tag{5a}$$

subject to

$$\sum_{p=1}^{U} x_{ip} = 1 \qquad \text{for all } i \in V \tag{5b}$$

$$x_{ip} + x_{jp} \leq 1 \qquad \text{for all } p \in \{1, \ldots, U\} \text{ and } i, j \in V \text{ with } c_i \neq c_j \tag{5c}$$

$$x_{ip} + x_{jq} \leq 1 \qquad \text{for all } (i, j) \in A \text{ and } 1 \leq q < p \leq U \tag{5d}$$

$$z - p\, x_{ip} \geq 0 \qquad \text{for all } i \in V \text{ and } p \in \{1, \ldots, U\} \tag{5e}$$

$$x_{ip} \in \{0, 1\} \qquad \text{for all } i \in V \text{ and } p \in \{1, \ldots, U\} \tag{5f}$$

$$z \in \mathbb{Z}_{\geq 0}. \tag{5g}$$

Constraints (5b) ensure that there is exactly one value $p \in \{1, \ldots, U\}$ with $x_{ip} = 1$. Constraints (5c) specify that a setup must occur between two operations in distinct classes. Constraints (5d) ensure that all precedence constraints are met. Constraints (5f) state that variables x are binary, and constraint (5g) specifies that the number of setups is integer. Clearly, this last constraint could be dropped. Together with constraints (5e), the objective (5a) expresses the minimization of the total number of setups.

3.3.3 Constraint programming model

The third model is based on constraint programming and has the following structure. For each operation $i \in V$, introduce an interval variable (IloIntervalVar in CP Optimizer) x_i indicating the start time of operation i, where, as in the continuous-time IP model, the start time reflects the number of setups executed before processing operation i. Next, introduce an interval sequencing variable (IloIntervalSequenceVar) y representing a total ordering of the interval variables $x_i, i \in V$. Define a two-dimensional setup matrix s (IloTransitionDistance) with an entry for each pair of operations. The value of entry (i, j) is 1 if a setup is needed when executing operation j directly after operation i, and 0 otherwise. Then, add a no-overlap constraint (IloNoOverlap) on the sequencing variable y using the setup matrix s. Finally, add an end-before-start constraint for each precedence constraint in A and set the minimization of the maximum start time as objective.

3.3.4 Comparison of DP with the IP and CP models

Before comparing the results, the following two comments about the results of the IP models are in order. First, both IP models are not able to adequately tackle the large instances: almost no feasible solutions are obtained for the large instances and the lower bounds are very weak. Therefore, we do not report the detailed results of the IP models for the large instances as they are of little value.

Second, the continuous-time and discrete-time IP could not find a feasible solution for 59 and 82 small instances, respectively. These high numbers can partially be explained by the quite tight choice of parameter value U in the models: both integer linear programming models discard all solutions with objective value larger than U from the solution space.

For a comparison with our DP algorithm, we calculate the same statistics as done when evaluating the components of DP. There is only a minor change: we exclude the number of states in the comparison but include the computation time difference. Figure 14 illustrates these numbers using the same structure as in the previous figures.

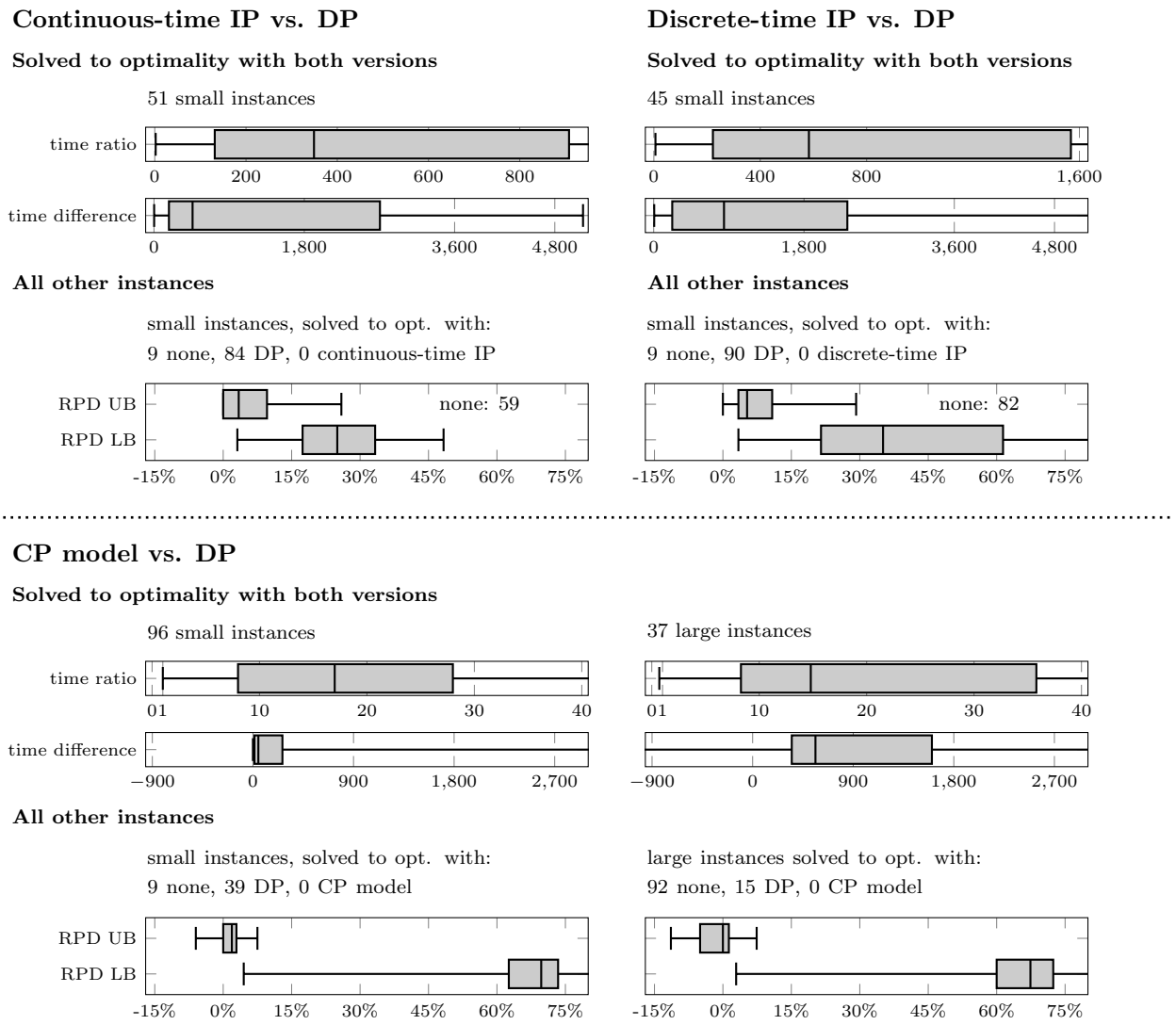

Figure 14: Comparison of the continuous-time IP model (top left), discrete-time IP model (top right), and CP model (bottom) with our DP (standard version).

Consider the results obtained with both IP models for the small instances. It is apparent that both IP models are not competitive with DP. Indeed, DP solves considerably more instances to optimality than both IP models (84 more than cont.-time IP and 90 more than disc.-time IP), and it needs substantially less computation time for the instances the IP models could also solve to optimality. This is confirmed by both the time ratio and the time difference box-and-whisker plots. Looking at the lower bounds, we observe that DP consistently provides better lower bounds than both IP models. Altogether, we conclude that DP clearly dominates both IP models.

Consider the results obtained with the CP model, which could handle the small as well as the large instances. We observe that whenever CP solves an instance to optimality, DP can solve it too. Additionally, DP can solve 39 small and 15 large instances to optimality that are unsolved with CP. Furthermore, the computation time spent for solving an instance is significantly smaller with DP than with CP, which is confirmed by both the time ratios and time differences. For example, the median time ratio is about 15 and the median time difference about 550 s for the large solved instances. For the instances that are not solved by both methods, interestingly, DP and CP find solutions of similar quality; The upper bound RPDs are often between -5% and +5%. However, CP provides very weak lower bounds; The lower bound RPDs are typically between 60% and 75%. We conclude that the low computation time and the good lower bounds are the main advantages of our DP algorithm over the CP model. Note that the value of having good lower bounds should not be underestimated. They do not only enable to prove optimality of a feasible solution and to finish the search earlier, they also generally provide a certificate of quality during the entire search process. Hence, when computation time is valuable, this certificate is important for deciding when to stop the search.

4 Concluding remarks

PCCSP captures various recurring decision problems in manufacturing and transportation systems. In scheduling terms, it models a one-machine problem with precedence constraints and setups where the goal is to minimize the number of setups. Previous research has shown that this problem is difficult to solve from both a theoretical and computational perspective. There has been, however, only little research on computational methods.

This article addresses this research gap by proposing a dynamic programming algorithm for PCCSP that incorporates sophisticated sub-procedures to successfully exploit the structure of the problem. As a result, the algorithm gives excellent computational results: it solves many printed circuit board routing instances with up to 350 operations to optimality and provides high-quality solutions and good lower bounds for the other instances.

In our view, these developments pave the way for interesting future research. The proposed dynamic programming algorithm could be made more efficient by improving its sub-procedures. A more complex heuristic, such as a local search approach similar to the one presented in Meuwly et al. (2010) for mixed graph coloring, could provide a better initial upper bound before starting the actual search. A promising avenue is also the development of better lower bound calculations. Similar as the one-class lower bound, one may calculate a lower bound for each pair of classes. The overall lower bound can then be obtained by choosing and combining some of these lower bounds. When considering the merging procedure, one may try to identify more complex structures to merge such as chains of vertices. One could also study PCCSP and our DP algorithm in the context of (multivalued) decision diagram for optimization, see (Cire and Hoeve, 2013; Bergman et al., 2016). In particular, it would be interesting to develop so-called relaxed and restricted decision diagrams to find better lower and upper bounds. We finally mention that, from an application perspective, it may be interesting to develop a similar solution method to exactly solve the shortest common supersequence problem, which is a standard problem in computer science and often used to model applications in genetics.

Appendix

Detailed computational results obtained for the small and large instances are given in Table 1 and 2, respectively. The tables are structured as follows. Each row presents the results for one instance. The first three columns describe the instance by providing its name and parameter values P (probability for being a small component) and L (number of large components). The next columns give the final lower bound (LB), upper bound (UB) and computation time (in seconds) obtained with the standard DP version, the continuous-time IP, the discrete-time IP, and the CP. The instances are grouped by the number of workstations, which is the same as the number of classes. No computation time is

indicated if the time limit of 7200 s is reached, and no lower bound is provided if the instance is solved to optimality, in which case LB equals UB. For each instance, the best lower and upper bounds are highlighted with a gray background. If multiple methods solved an instance to optimality, only the fasted is highlighted. No results of the IP models are listed for the large instances.

Table 1: Detailed results for the small instances.

Instance			*DP*			*Cont.-time IP*			*Discr.-time IP*			*CP*		
Name	***P***	***L***	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***
3 workstations														
PCB-S-001	0.5	0		29	3		29	345		29	842		29	13
PCB-S-002	0.5	0		31	1		31	192		31	405		31	20
PCB-S-003	0.5	0		28	2		28	468		28	891		28	12
PCB-S-004	0.5	4		31	1		31	128		31	323		31	14
PCB-S-005	0.5	4		24	1		24	123		24	229		24	8
PCB-S-006	0.5	4		26	1		26	261		26	583		26	8
PCB-S-007	0.5	8		27	1		27	24		27	31		27	10
PCB-S-008	0.5	8		23	1		23	349		23	1120		23	12
PCB-S-009	0.5	8		26	1		26	552		26	216		26	16
PCB-S-010	0.5	12		20	1		20	37		20	58		20	5
PCB-S-011	0.5	12		20	1		20	6		20	15		20	5
PCB-S-012	0.5	12		20	1		20	174		20	67		20	9
PCB-S-013	0.7	0		25	2		25	4496		25	2026		25	16
PCB-S-014	0.7	0		26	3		26	766		26	3525		26	19
PCB-S-015	0.7	0		27	2		27	292		27	576		27	39
PCB-S-016	0.7	4		26	1		26	349		26	2663		26	25
PCB-S-017	0.7	4		25	1		25	532		25	2057		25	20
PCB-S-018	0.7	4		23	1		23	135		23	1140		23	17
PCB-S-019	0.7	8		22	1		22	1007		22	279	21	22	
PCB-S-020	0.7	8		24	1		24	184		24	452		24	12
PCB-S-021	0.7	8		24	1		24	239		24	393		24	8
PCB-S-022	0.7	12		13	1		13	77		13	40		13	1
PCB-S-023	0.7	12		17	1		17	34		17	72		17	5
PCB-S-024	0.7	12		19	1		19	330		19	587		19	6
PCB-S-025	0.9	0		24	3		24	4024		24	2045		24	15
PCB-S-026	0.9	0		22	1		22	1526		22	2191		22	20
PCB-S-027	0.9	0		23	2		23	5138		23	2319		23	16
PCB-S-028	0.9	4		16	1		16	38		16	102		16	4
PCB-S-029	0.9	4		21	1		21	396		21	1314		21	20
PCB-S-030	0.9	4		22	1		22	551		22	2680		22	13
PCB-S-031	0.9	8		16	1		16	659		16	352		16	3
PCB-S-032	0.9	8		18	1		18	462		18	222		18	7
PCB-S-033	0.9	8		17	1		17	18		17	51		17	2
PCB-S-034	0.9	12		15	1		15	2		15	8		15	4
PCB-S-035	0.9	12		12	1		12	16		12	6		12	5
PCB-S-036	0.9	12		18	1		18	336		18	848		18	4
4 workstations														
PCB-S-037	0.5	0		41	25	36	42		17	-			41	298
PCB-S-038	0.5	0		37	16	35	37		33	-			37	103
PCB-S-039	0.5	0		35	46	30	36		16	-			35	3355
PCB-S-040	0.5	4		32	5	31	32		24	-			32	279
PCB-S-041	0.5	4		36	24	32	38		32	-			36	202
PCB-S-042	0.5	4		38	16	34	38		33	-			38	271
PCB-S-043	0.5	8		31	2		31	3096		31	6765		31	37
PCB-S-044	0.5	8		27	2		27	880	24	-			27	48
PCB-S-045	0.5	8		24	4	20	26		20	25			24	55
PCB-S-046	0.5	12		27	1	24	27			27	4158		27	31
PCB-S-047	0.5	12		21	1		21	1290		21	1470		21	31
PCB-S-048	0.5	12		31	2		31	3116		31	6156		31	42
PCB-S-049	0.7	0		32	48	24	-		11	-			32	1038
PCB-S-050	0.7	0		32	41	26	-		26	-			32	1065
PCB-S-051	0.7	0		36	75	32	37		28	-			36	291
PCB-S-052	0.7	4		29	9	25	31		28	30			29	42
PCB-S-053	0.7	4		29	12	25	30		25	-			29	165
PCB-S-054	0.7	4		31	12	26	31		27	-			31	133
PCB-S-055	0.7	8		25	2	21	25			25	4722		25	51
PCB-S-056	0.7	8		22	2		22	4681		22	3867		22	30

Table 1: continued

Instance			*DP*			*Cont.-time IP*			*Discr.-time IP*			*CP*		
Name	P	L	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***
PCB-S-057	0.7	8		25	3	21	28		23	26			25	58
PCB-S-058	0.7	12		19	2		19	4438	18	20			19	47
PCB-S-059	0.7	12		26	3		26	2316	25	26			26	51
PCB-S-060	0.7	12		23	2	19	23		20	23			23	37
PCB-S-061	0.9	0		30	36	22	-		12	-			30	1335
PCB-S-062	0.9	0		30	39	24	-		22	-			30	688
PCB-S-063	0.9	0		32	71	24	39		8	-			32	4034
PCB-S-064	0.9	4		23	6		23	3599	20	26			23	141
PCB-S-065	0.9	4		29	16	21	32		22	-			29	765
PCB-S-066	0.9	4		27	8	24	27		21	-			27	87
PCB-S-067	0.9	8		24	2		24	4288	19	26			24	116
PCB-S-068	0.9	8		21	2		21	3740	18	23			21	42
PCB-S-069	0.9	8		25	2	20	27		21	27			25	211
PCB-S-070	0.9	12		22	1		22	810		22	3407		22	35
PCB-S-071	0.9	12		23	2	20	23			23	3137		23	65
PCB-S-072	0.9	12		24	1		24	391		24	3596		24	34
5 workstations														
PCB-S-073	0.5	0		46	91	39	-		37	-			46	3633
PCB-S-074	0.5	0		45	108	37	-		19	-		24	45	
PCB-S-075	0.5	0		45	713	33	-		31	-		10	45	
PCB-S-076	0.5	4		39	102	31	-		9	-			39	2917
PCB-S-077	0.5	4		40	61	34	-		14	-			40	889
PCB-S-078	0.5	4		42	70	33	-		31	-		16	42	
PCB-S-079	0.5	8		36	20	28	-		12	-			36	1530
PCB-S-080	0.5	8		36	40	28	-		29	-			36	1264
PCB-S-081	0.5	8		35	69	27	39		25	-			35	4860
PCB-S-082	0.5	12		32	17	28	32		10	-			32	448
PCB-S-083	0.5	12		32	10	30	32		29	33			32	73
PCB-S-084	0.5	12		37	28	32	40		31	41			37	92
PCB-S-085	0.7	0		39	516	25	-		7	-		13	41	
PCB-S-086	0.7	0		38	839	23	-		6	-		12	39	
PCB-S-087	0.7	0		39	670	25	-		9	-		12	39	
PCB-S-088	0.7	4		35	214	26	-		19	-		14	36	
PCB-S-089	0.7	4		35	94	26	-		20	-		13	36	
PCB-S-090	0.7	4		37	319	27	-		11	-		16	37	
PCB-S-091	0.7	8		26	25	23	27		20	27			26	83
PCB-S-092	0.7	8		29	56		29	4722	22	-			29	467
PCB-S-093	0.7	8		29	9	21	30		21	-			29	6496
PCB-S-094	0.7	12		28	8	24	28		24	28			28	77
PCB-S-095	0.7	12		30	21		30	4990	23	35			30	214
PCB-S-096	0.7	12		24	3		24	4034	21	31			24	92
PCB-S-097	0.9	0		37	420	22	-		11	-		11	37	
PCB-S-098	0.9	0		35	372	22	-		7	-		12	36	
PCB-S-099	0.9	0		34	870	22	-		7	-		10	34	
PCB-S-100	0.9	4		30	96	21	-		21	-			30	2538
PCB-S-101	0.9	4		33	197	25	-		8	-			33	5701
PCB-S-102	0.9	4		32	86	24	-		9	-			32	4120
PCB-S-103	0.9	8		26	11	19	-		19	-			26	745
PCB-S-104	0.9	8		30	79	22	-		21	-		13	30	
PCB-S-105	0.9	8		27	41	22	34		22	-			27	121
PCB-S-106	0.9	12		24	5	19	27		18	-			24	139
PCB-S-107	0.9	12		23	7	19	24		19	-			23	268
PCB-S-108	0.9	12		20	6	16	22		15	23			20	102
7 workstations														
PCB-S-109	0.5	0		56	4457	37	-		18	-		17	57	
PCB-S-110	0.5	0		51	3657	30	-		23	-		11	52	
PCB-S-111	0.5	0		54	948	37	-		19	-		15	55	
PCB-S-112	0.5	4		47	4082	33	-		24	-		14	48	
PCB-S-113	0.5	4		51	438	33	-		21	-		20	51	
PCB-S-114	0.5	4		53	514	36	-		28	-		14	53	
PCB-S-115	0.5	8		45	1240	32	-		28	-		11	46	
PCB-S-116	0.5	8		38	262	28	-		9	-		7	40	
PCB-S-117	0.5	8		45	411	30	-		29	-		14	46	
PCB-S-118	0.5	12		36	90	28	-		27	-		14	37	

Table 1: continued

Instance			*DP*			*Cont.-time IP*			*Discr.-time IP*			*CP*		
Name	P	L	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***
PCB-S-119	0.5	12		45	336	34	-		32	-		12	45	
PCB-S-120	0.5	12		44	167	34	-		26	-		14	45	
PCB-S-121	0.7	0	37	50		25	-		6	-		11	47	
PCB-S-122	0.7	0	41	53		31	-		5	-		11	53	
PCB-S-123	0.7	0	40	51		25	-		21	-		12	49	
PCB-S-124	0.7	4	37	46		27	-		18	-		8	47	
PCB-S-125	0.7	4		41	7042	24	-		20	-		12	42	
PCB-S-126	0.7	4		41	2456	24	-		20	-		9	44	
PCB-S-127	0.7	8		36	5833	23	-		20	-		7	37	
PCB-S-128	0.7	8		34	137	25	-		24	-		15	35	
PCB-S-129	0.7	8		36	488	27	44		24	-		13	38	
PCB-S-130	0.7	12		30	94	24	30		23	-			30	1462
PCB-S-131	0.7	12		30	318	20	-		18	-		15	31	
PCB-S-132	0.7	12		37	1065	24	45		24	-		11	37	
PCB-S-133	0.9	0	39	50		23	-		4	-		13	49	
PCB-S-134	0.9	0	38	50		22	-		18	-		12	48	
PCB-S-135	0.9	0	38	47		25	-		6	-		12	46	
PCB-S-136	0.9	4	32	40		22	-		11	-		13	41	
PCB-S-137	0.9	4		33	1726	22	-		21	-		10	34	
PCB-S-138	0.9	4	35	45		23	-		13	-		10	44	
PCB-S-139	0.9	8		33	2936	18	-		19	-		7	35	
PCB-S-140	0.9	8		40	3495	23	-		19	-		7	43	
PCB-S-141	0.9	8		36	2099	23	-		8	-		9	38	
PCB-S-142	0.9	12		29	376	15	-		17	-		7	29	
PCB-S-143	0.9	12		31	66	22	-		21	-			31	4623
PCB-S-144	0.9	12		29	313	18	-		19	-		12	29	

Table 2: Detailed results for the large instances.

Instance			*DP*			*CP*		
Name	P	L	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***
3 workstations								
PCB-L-001	0.5	0		45	81		45	724
PCB-L-002	0.5	0		47	180		47	2041
PCB-L-003	0.5	0		45	86		45	648
PCB-L-004	0.5	4		47	231		47	5856
PCB-L-005	0.5	4		47	90		47	419
PCB-L-006	0.5	4		39	61		39	783
PCB-L-007	0.5	8		43	54		43	1038
PCB-L-008	0.5	8		48	63		48	241
PCB-L-009	0.5	8		46	26		46	385
PCB-L-010	0.5	12		45	36		45	402
PCB-L-011	0.5	12		40	40		40	327
PCB-L-012	0.5	12		40	35		40	332
PCB-L-013	0.7	0		38	146		38	598
PCB-L-014	0.7	0		38	134		38	623
PCB-L-015	0.7	0		42	123		42	1239
PCB-L-016	0.7	4		35	76		35	2718
PCB-L-017	0.7	4		34	119		34	6447
PCB-L-018	0.7	4		34	57		34	865
PCB-L-019	0.7	8		34	38		34	317
PCB-L-020	0.7	8		39	106		39	4027
PCB-L-021	0.7	8		36	115		36	1123
PCB-L-022	0.7	12		33	10		33	158
PCB-L-023	0.7	12		29	17		29	407
PCB-L-024	0.7	12		35	5		35	359
PCB-L-025	0.9	0		34	308		34	4614
PCB-L-026	0.9	0		34	306	16	35	
PCB-L-027	0.9	0		36	247	17	36	
PCB-L-028	0.9	4		33	86		33	6124
PCB-L-029	0.9	4		32	25		32	1629
PCB-L-030	0.9	4		33	38		33	669

Table 2: continued

Instance			*DP*			*CP*		
Name	P	L	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***
PCB-L-031	0.9	8		30	34		30	440
PCB-L-032	0.9	8		29	6		29	356
PCB-L-033	0.9	8		32	12		32	1421
PCB-L-034	0.9	12		30	4		30	158
PCB-L-035	0.9	12		28	9		28	181
PCB-L-036	0.9	12		27	8		27	369
4 workstations								
PCB-L-037	0.5	0	51	62		20	61	
PCB-L-038	0.5	0	52	61		22	60	
PCB-L-039	0.5	0		60	6133	22	62	
PCB-L-040	0.5	4		57	5176	18	58	
PCB-L-041	0.5	4	52	57		32	57	
PCB-L-042	0.5	4		54	4231	16	58	
PCB-L-043	0.5	8		54	3617	16	56	
PCB-L-044	0.5	8		51	1338	18	52	
PCB-L-045	0.5	8	47	57		16	58	
PCB-L-046	0.5	12		51	6684	12	51	
PCB-L-047	0.5	12	43	54		12	52	
PCB-L-048	0.5	12		50	1569		50	5668
PCB-L-049	0.7	0	38	49		13	49	
PCB-L-050	0.7	0	43	53		15	53	
PCB-L-051	0.7	0	39	51		18	53	
PCB-L-052	0.7	4	42	50		11	50	
PCB-L-053	0.7	4	41	51		13	50	
PCB-L-054	0.7	4	39	48		14	47	
PCB-L-055	0.7	8	37	44		14	45	
PCB-L-056	0.7	8	41	47		12	48	
PCB-L-057	0.7	8		40	2767	17	41	
PCB-L-058	0.7	12		41	1065	13	42	
PCB-L-059	0.7	12		43	2377	14	43	
PCB-L-060	0.7	12		42	3357		42	2272
PCB-L-061	0.9	0	37	48		16	48	
PCB-L-062	0.9	0	35	45		13	45	
PCB-L-063	0.9	0	37	46		14	47	
PCB-L-064	0.9	4	37	45		13	46	
PCB-L-065	0.9	4	35	43		12	43	
PCB-L-066	0.9	4	36	46		10	47	
PCB-L-067	0.9	8		39	5685	13	40	
PCB-L-068	0.9	8	35	42		12	44	
PCB-L-069	0.9	8		44	3389	11	46	
PCB-L-070	0.9	12		35	825	34	35	
PCB-L-071	0.9	12		37	1513		37	3636
PCB-L-072	0.9	12		38	3647	18	38	
5 workstations								
PCB-L-073	0.5	0	65	82		26	81	
PCB-L-074	0.5	0	58	80		28	77	
PCB-L-075	0.5	0	65	79		29	77	
PCB-L-076	0.5	4	60	79		13	75	
PCB-L-077	0.5	4	55	74		16	69	
PCB-L-078	0.5	4	55	71		22	68	
PCB-L-079	0.5	8	50	66		19	67	
PCB-L-080	0.5	8	52	69		14	67	
PCB-L-081	0.5	8	55	63		16	63	
PCB-L-082	0.5	12	45	60		12	59	
PCB-L-083	0.5	12	48	64		24	63	
PCB-L-084	0.5	12	53	70		14	66	
PCB-L-085	0.7	0	48	66		15	68	
PCB-L-086	0.7	0	45	64		17	64	
PCB-L-087	0.7	0	51	71		13	70	
PCB-L-088	0.7	4	46	67		10	65	
PCB-L-089	0.7	4	44	61		16	60	
PCB-L-090	0.7	4	43	60		11	59	
PCB-L-091	0.7	8	39	54		13	55	
PCB-L-092	0.7	8	41	57		12	55	

Table 2: continued

Instance			*DP*			*CP*		
Name	P	L	***LB***	***UB***	***Time***	***LB***	***UB***	***Time***
PCB-L-093	0.7	8	40	54		11	54	
PCB-L-094	0.7	12	39	50		14	51	
PCB-L-095	0.7	12	42	58		17	55	
PCB-L-096	0.7	12	38	52		12	52	
PCB-L-097	0.9	0	40	61		16	62	
PCB-L-098	0.9	0	40	61		13	61	
PCB-L-099	0.9	0	40	59		14	59	
PCB-L-100	0.9	4	41	57		13	57	
PCB-L-101	0.9	4	36	51		10	51	
PCB-L-102	0.9	4	36	54		12	54	
PCB-L-103	0.9	8	37	56		15	60	
PCB-L-104	0.9	8	41	55		13	55	
PCB-L-105	0.9	8	35	48		9	48	
PCB-L-106	0.9	12	35	49		10	51	
PCB-L-107	0.9	12	33	44		10	45	
PCB-L-108	0.9	12	38	48		10	47	
7 workstations								
PCB-L-109	0.5	0	66	103		30	98	
PCB-L-110	0.5	0	68	104		21	98	
PCB-L-111	0.5	0	70	103		34	102	
PCB-L-112	0.5	4	56	87		20	79	
PCB-L-113	0.5	4	64	100		12	92	
PCB-L-114	0.5	4	75	103		16	94	
PCB-L-115	0.5	8	62	98		30	90	
PCB-L-116	0.5	8	65	93		17	86	
PCB-L-117	0.5	8	67	95		21	96	
PCB-L-118	0.5	12	66	92		35	84	
PCB-L-119	0.5	12	62	89		15	82	
PCB-L-120	0.5	12	61	88		29	78	
PCB-L-121	0.7	0	57	91		11	92	
PCB-L-122	0.7	0	55	92		22	87	
PCB-L-123	0.7	0	52	87		11	82	
PCB-L-124	0.7	4	54	83		11	84	
PCB-L-125	0.7	4	51	82		14	77	
PCB-L-126	0.7	4	52	84		19	80	
PCB-L-127	0.7	8	47	80		9	72	
PCB-L-128	0.7	8	51	79		25	74	
PCB-L-129	0.7	8	48	75		11	76	
PCB-L-130	0.7	12	44	71		13	68	
PCB-L-131	0.7	12	50	72		11	69	
PCB-L-132	0.7	12	43	72		14	66	
PCB-L-133	0.9	0	44	85		13	79	
PCB-L-134	0.9	0	46	83		21	81	
PCB-L-135	0.9	0	47	85		19	80	
PCB-L-136	0.9	4	41	67		10	62	
PCB-L-137	0.9	4	46	75		17	75	
PCB-L-138	0.9	4	49	77		15	76	
PCB-L-139	0.9	8	45	76		13	70	
PCB-L-140	0.9	8	40	65		11	60	
PCB-L-141	0.9	8	40	75		13	69	
PCB-L-142	0.9	12	36	63		18	58	
PCB-L-143	0.9	12	39	57		17	56	
PCB-L-144	0.9	12	36	59		7	56	

References

A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. SIAM Journal on Computing, 1(2):131–137, 1972.

G. V. Andreev, Y. N. Sotskov, and F. Werner. A branch and bound method for mixed graph coloring and scheduling. In Proceedings of the 16th International Conference on CAD/CAM Robotics & Factories of the Future, volume 1, pages 1–8, Trinidad and Tobago, 2000.

D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker. Decision Diagrams for Optimization, volume 1. Springer, 2016.

I. Blöchliger and N. Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. Computers and Operations Research, 35(3):960–975, 2008.

A. Camelot. Modélisation des tâches pour la récupération de pièces réutilisables sur un avion en fin de vie. Master's thesis, Department of Mechanical Engineering, Polytechnique Montréal, 2012.

A. A. Cire and W.-J. V. Hoeve. Multivalued decision diagrams for sequencing problems. Operations Research, 61(6):1411–1428, 2013.

J. R. Correa, S. Fiorini, and N. E. Stier-Moses. A note on the precedence-constrained class sequencing problem. Discrete Applied Mathematics, 155(3):257–259, 2007.

A. Darte. On the complexity of loop fusion. Parallel Computing, 26(9):1175–1193, 2000.

F. F. Easton. A dynamic program with fathoming and dynamic upper bounds for the assembly line balancing problem. Computers & Operations Research, 17(2):163–175, 1990.

P. Hansen, J. Kuplinsky, and D. Werra. Mixed graph colorings. Mathematical Methods of Operations Research, 45(1):145–160, 1997.

A. Kouider, H. Ait Haddadène, S. Ourari, and A. Oulamara. Mixed graph colouring for unit-time scheduling. International Journal of Production Research, 55(6):1720–1729, 2017.

C. B. Lofgren. Machine configuration of flexible printed circuit board assembly systems. PhD thesis, School of ISyE, Georgia Institute of Technology, 1986.

C. B. Lofgren, F. M. Leon, and C. A. Tovey. Routing printed circuit cards through an assembly cell. Operations Research, 39(6):992–1004, 1991.

F.-X. Meuwly, B. Ries, and N. Zufferey. Solution methods for a scheduling problem with incompatibility and precedence constraints. Algorithmic Operations Research, 5(2):75–85, 2010.

A. Mingozzi, L. Bianco, and S. Ricciardelli. Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. Operations Research, 45(3):365–377, 1997.

T. L. Morin and R. E. Marsten. Branch-and-bound strategies for dynamic programming. Operations Research, 24(4):611–627, 1976.

B. Ries. Coloring some classes of mixed graphs. Discrete Applied Mathematics, 155(1):1–6, 2007.

E. C. Sewell and S. H. Jacobson. A branch, bound, and remember algorithm for the simple assembly line balancing problem. INFORMS Journal on Computing, 24(3):433–442, 2012.

Y. N. Sotskov, V. S. Tanaev, and F. Werner. Scheduling problems and mixed graph colorings. Optimization, 51(3):597–624, 2002.

C. A. Tovey. Non-approximability of precedence-constrained sequencing to minimize setups. Discrete Applied Mathematics, 134(1-3):351–360, 2004.