DEPARTMENT OF APPLIED ECONOMICS

RESEARCH REPORT

PARTITIONING A PERMUTATION GRAPH: ALGORITHMS AND AN APPLICATION

LINDA S. MOONEN & FRITS C.R. SPIEKSMA

OR 0358

Partitioning a permutation graph: algorithms and an application

Linda S. Moonen • Frits C.R. Spieksma

Katholieke Universiteit Leuven, Department of Applied Economics, Naamsestraat 69, B-3000 Leuven, Belgium

linda.moonen@econ.kuleuven.ac.be • frits.spieksma@econ.kuleuven.ac.be

In this paper we discuss the problem of partitioning a permutation graph into cliques of bounded size, and describe a real-life application of this problem encountered at a manufacturing company. We formulate the problem as an integer program, and present two exact algorithms for solving it. The first algorithm is a branch-and-price algorithm based on the integer programming formulation; the second one is an algorithm based on the concept of bounded clique-width. The latter algorithm was motivated by the structure present in the real-life instances. Test results are given, both for real-life instances and randomly generated instances. As far as we are aware, this is the first implementation of an algorithm based on bounded clique-width.

(*Integer Programming, Analysis of Algorithms*)

1. Introduction

Consider the following situation. Given is a set S of distinct points in the plane, $S = \{1, 2, ..., n\}$. For any pair of points $i, j \in S$, we say that i is *smaller* than j ($i \prec j$) iff

$$(x_i \leq x_j) \wedge (y_i \leq y_j).$$

Here, x_i and y_i denote the x- and y-coordinate of i, $i \in S$. Further, we call $R \subseteq S$ a *stack* iff for any two points $i, j \in R$ either $i \prec j$ or $j \prec i$.

The problem we consider is as follows: given the set S and an integer B, partition S into as few stacks as possible such that each stack contains no more than B points. We refer to this problem as problem P.

Problem P is intimately related to problems in graph theory. Indeed, when we build a graph $G = (V, E)$ as follows: for each point i in S there is a node in V, and two nodes are

adjacent iff $i \prec j$ or $j \prec i$, a so-called permutation graph arises (see for instance Golumbic 1980). Observe that a stack in S corresponds to a clique in G. Thus, in graph-theoretic terms, problem P boils down to finding a partition of the permutation graph G into a minimum number of cliques such that each clique has no more than B vertices. In fact, as we describe at the end of this section, the two problems are equivalent.

Of course, if B is not present in the input of our problem, the resulting problem is solvable in polynomial time since it is a special case of Dilworth's chain decomposition theorem (Dilworth 1950). However, Jansen (2003) proves that for each fixed $B \geq 6$, problem P is $\mathcal{NP}$-hard.

Apart from the application sketched in section 2, problem P occurs in the field of mutual exclusion scheduling problems (Jansen 2003, Baker and Coffman 1996). In this scheduling problem a graph is given such that each vertex corresponds to a job, and an edge between two vertices indicates that the two corresponding jobs are incompatible, i.e., cannot be processed at the same time. Assuming that we have B processors available, and that each job needs a single time-unit, computing a schedule such that the latest job finishes as soon as possible is an instance of problem P (provided that the conflict graph is a permutation graph). Another related problem, described in Felsner and Wernisch (1998) involves covering as many points in a planar point set as possible, using a given number of chains.

The goal of this work is

- to describe a real-life application of problem P,
- to propose two exact algorithms for solving problem P: a branch-and-price algorithm and an enumerative algorithm based on the concept of bounded clique-width,
- to assess the quality of these algorithms by performing computational experiments on instances from practice as well as on randomly generated instances.

The paper is organized as follows. In Section 2 we describe a setting we encountered at a manufacturer of storage systems. Section 3 proposes a branch-and-price approach based on a set-partitioning formulation of P (see Barnhart et al. (1998) for a description of branch-and-price algorithms). We show that the pricing problem is solvable in polynomial time, and that we can generalize this approach to partial orders. Section 4 is devoted to an exact enumeration algorithm for a special case of problem P. This algorithm is based on the concept of bounded clique-width. In Section 5, we show computational results from

the branch-and-price algorithm and from the algorithm based on bounded clique-width. Section 6 contains the conclusions.

Finally, let us argue that a permutation graph can be represented in the plane, implying the equivalence of problem P and the problem of partitioning a permutation graph into bounded-size cliques.

The following can be found in Golumbic (1980). For each permutation graph $G = (V, E)$, a permutation $\pi = (\pi_1, \pi_2, ..., \pi_{|V|})$ can be computed such that the presence of an edge between two nodes i and j is equivalent with $(i - j)(\pi_i^{-1} - \pi_j^{-1}) < 0$, where π_i^{-1} is the position of i in the permutation π. This means that, for $i > j$, there is an edge (i, j) iff i precedes j in the sequence π.

Consider now the following point set S. For every $i \in V$:

$$x_i = \pi_i^{-1}$$

$$y_i = |V| - i$$

It follows that an edge (i, j) in the permutation graph implies $i \prec j$ or $j \prec i$ in the set S. Hence, a clique in G corresponds to a stack in S, which implies that problem P and partitioning a permutation graph into bounded-size cliques are equivalent.

2. Problem Description

Bruynzeel Storage Systems (BSS), a manufacturing company in the Netherlands, produces storage systems. These storage systems are delivered worldwide. To construct such a system, BSS produces many rectangular shaped boxes, each with a specific length and a specific width. We refer to such a rectangular shaped box as an item. A single storage system may consist of up to 200 items. Further, there are no standard sizes, so each customer specifies its own requirements. The height of an item, however, is identical for all items. The items have to be loaded onto pallets for transportation to the clients. It is allowed to place items on top of each other in layers; however, the number of items per layer is restricted to one. Since the items all have identical heights, it follows that the height of the trucks that transport the pallets determines the maximum number of layers of each pallet. We denote this number by B (in the case of BSS, B is equal to 12). A crucial feature involves the *stability* of the pallets (see for example Bischoff (1991)). BSS stipulated that no larger item could be placed on top of a smaller item. More precisely, both the length and the width of an item placed

in some layer must be smaller than or equal to the length and the width of the item placed in the layer directly under it. This restriction ensures that pallets arrive in good shape at their final destination (Moonen 2001). In order to achieve an efficient usage of the trucks it is important to minimize the total number of pallets used.

This problem can be seen as a pallet loading problem (PLP). Pallet or container loading problems concern the optimal packing of small items into large containers or pallets. The terms *pallets* and *containers* are used interchangeably in most studies, although there is an important difference between them. When loading goods on a pallet, the notion of the *stability* of the loading schemes is far more important than when the goods are to be loaded into a container. When loading items on a pallet, we cannot make use of the upstanding walls that we have when loading items into a container, so the stability of the loading schemes must be guaranteed (Bischoff 1991).

Most of the research on PLPs has concentrated on the case where a set of identical items has to be loaded onto a single pallet. Dyckhoff (1990) gives a detailed overview of the different types of PLPs and proposes a number of solution approaches for solving them. In more recent work, Morabito and Morales (1998) developed a heuristic based on a recursive procedure to solve the problem, and G and Kang (2001) propose a heuristic that can be applied to relatively large instances (more than 5000 items). Letchford and Amaral (2001) give a detailed analysis of upper bounds for the PLP. Also, some heuristics have been suggested for solving the PLP with non-identical items. Scheithauer and Terno (1996b) developed a heuristic combining a general branch-and-bound framework with optimal two-dimensional loading patterns. More recently, Terno et al. (2000) proposed an algorithm that uses the G4-heuristic introduced in Scheithauer and Terno (1996a), and combine this with a branch-and-bound procedure.

Notice that the application described here allows for identical items, whereas we assume in problem P that all items are pairwise distinct. It is not difficult to see, however, that all results presented later are valid for the case where identical items are allowed.

3. A Branch-and-Price Algorithm

In this section we formulate problem P as an integer program and we describe a branch-and-price algorithm for solving it (see eg. Barnhart et al. (1998)).

3.1 Problem Formulation

For the mathematical formulation of our problem, we define a *stable stack* $R \subseteq S$ as a stack with $|R| \leq B$. Further, we introduce a decision variable x_k for every possible stable stack k, such that:

$$x_k = \begin{cases} 1 & \text{if stable stack } k \text{ is in the solution} \\ 0 & \text{otherwise.} \end{cases}$$

Using a set-partitioning formulation, we get the following model:

$$\text{Min} \sum_k x_k \tag{1}$$

subject to

$$\sum_{k:i \in k} x_k = 1 \qquad \forall i \tag{2}$$

$$x_k \in \{0, 1\} \qquad \forall k \tag{3}$$

The objective (1) is to minimize the total number of stacks needed to pack all items. Constraints (2) state that each item has to be in exactly one stack, and constraints (3) are the zero-one constraints on the x_k variables.

3.2 Column Generation

Since the number of variables in formulation (1)-(3) is exponentially large, we employ column generation to find the LP-relaxation of (1)-(3) without having to enumerate all variables. In the column generation process, we start with a small subset of the variables that contains a feasible solution. All other variables are implicitly assigned the value zero. The subproblem constructed in this way is called the restricted master problem (RMP). We solve the LP-relaxation of RMP, and then we have to determine whether the solution found is optimal for the master problem. To do this, we have to answer the question: do there exist variables with negative reduced costs? Let u_i $(i = 1, ..., n)$ be the dual variables corresponding to constraints (2) from our formulation. We can now formulate an expression for the reduced costs of a variable x_k:

$$1 - \sum_{i:i \in k} u_i$$

Thus, given a feasible solution to the LP-relaxation and the corresponding dual variables, the pricing problem boils down to the following question:

$$\exists k \quad \text{such that} \sum_{i:i\in k} u_i > 1?$$

Lemma 1 *The pricing problem can be solved in polynomial time.*

Proof: We construct a directed graph $D = (V, A)$. There is a node in V for each item, and there is a source s in V. We draw an arc from node i to node j if for the corresponding items $i \prec j$ holds; this arc has length u_j. Also, there is an arc from s to each node $i \in V$ with length u_i. Observe that the constructed graph is acyclic. We now define $d^p(j)$ to be the length of a longest path from s to j using at most p arcs ($j = 1, ..., n$). We claim that these longest paths can be calculated in polynomial time using the following dynamic programming recursion:

$$\begin{aligned} d^p(j) &= max(max_{i:(i,j)\in A} d^{p-1}(i) + u_j, d^{p-1}(j)) \\ &\text{with } d^1(j) = u_j \quad \forall j \neq s \quad (p = 2, ..., B) \end{aligned} \tag{4}$$

Let us show by induction that the values $d^p(j)$ computed by the dynamic programming recursion (4) satisfy their definition. The case $p = 1$ is trivial, so let us assume that it holds for $p = l - 1$. Consider now a longest path from s to j using at most l arcs. If this path contains exactly l arcs, there is a predecessor of j in this path, say j', such that the longest path from s to j' using at most $l - 1$ arcs consists of the first $l - 1$ arcs in the longest path from s to j. By induction the latter value (i.e., the length of a longest path from s to j' using at most $l-1$ arcs) is recorded in $d^{l-1}(j')$. If this path contains less than l arcs, it follows that $d^l(j) = d^{l-1}(j)$. It follows that (4) computes $d^l(j)$ correctly. Thus, testing whether a node j exists such that $d^B(j) > 1$ amounts to answering the pricing problem. □

A consequence of Lemma 1 is that the LP-relaxation of (1)-(3) can be solved in polynomial time.

Remark. One could consider a situation where a weight p_k is given for each possible stack k, and next minimize total weight. For instance, in terms of the application, it would be quite natural to define the weight of stack k as the area of its largest item. Indeed, it is easy to exhibit examples where minimizing total area is not equivalent to minimizing the number of stacks needed. Notice that in this case the efficient solvability of the pricing problem is preserved since by computing $d^B(j)$ using (4), and next comparing each value

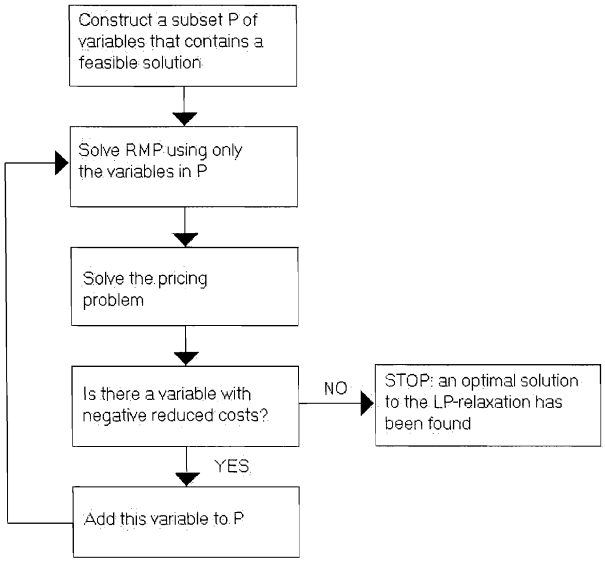

Figure 1: Procedure for column generation

with the corresponding area of item j determines whether a variable with negative reduced costs exists.

The solution found by applying the column generation procedure described in Figure 1 will in general be a fractional solution. We now sketch a branching structure in order to find the integer optimum.

3.3 Branching procedure

The branching rule we use to partition the solution space is based on the order in which items are packed into a stack. Two items are called *successors* if they are packed in the same stack such that one item lies directly above the other.

Lemma 2 *If a given LP-solution x is fractional, there exists a pair of items i and j which are successors in a certain stack k with $1 > x_k > 0$, such that*

$$0 < \sum_{k:i,j \ successors \ in \ k} x_k < 1$$

Proof: Suppose that the lemma is false. Consider a fractional stack k (i.e., a stack whose corresponding variable has a fractional value) and suppose that it contains m items, $\{1, 2, ..., m\}, m \geq 2$ (notice that such a stack always exists). For the lemma to be false, it must be true that

$$\sum_{k:l,l+1 \ successors \ in \ k} x_k = 1, \text{for } l = 1, ..., m-1.$$

Thus all fractional stacks that contain item l must also contain item $l+1$ as its successor ($l = 1, ..., m-1$). Further, since the LP-solution x satisfies constraints (2) for each item $l = 1, ..., m$, it follows that $x_k = 1$. Thus, the LP-solution is integral, which is in contradiction with our assumption of a fractional solution, and proves the correctness of the lemma. □

When an optimal, fractional LP-solution has been found, we identify two items i and j for which the sum of all stacks where i and j are successors lies between 0 and 1. We know from Lemma 2 that two such items exist. In the integer optimum, these two items will either be successors in a stack, or they will not. So, given two items i and j, we branch as follows. In one branch we modify the directed graph D in such a way that items i and j have to be successors. We can do this by deleting all arcs (i, p) for $p \neq j$ and all arcs (p, j) for $p \neq i$. Observe that, when solving the pricing problem in case j is a successor of i, the value of $d^B(i)$ is no longer relevant since a stack with item i not followed by item j is not allowed in this branch. Therefore, we record in each node of the tree which items cannot serve as a last item in a stack, and for these items j we do not consider $d^B(j)$. In the second branch, we make sure that items i and j can never be successors in a solution, by deleting arc (i, j) from D. In our algorithm we employ this branching step repeatedly to find an integer solution to our problem. Notice that this branching scheme keeps the problem structure intact, which allows us to use column generation throughout the branch-and-bound tree.

3.4 Partial Orders

To what extent can we generalize the branch-and-price approach? In this subsection we show how the approach sketched in subsections 3.2 and 3.3 remains valid for so-called partial orders.

In Trotter (1992), a partial order is defined as a pair $(X, \leq)$, where X is a set and $\leq$ is a relation on X satisfying:

1. $x \leq x$ for each $x \in X$;
2. if $x \leq y$ and $y \leq x$ then $x = y$;
3. if $x \leq y$ and $y \leq z$ then $x \leq z$.

A *chain* C is a subset of X such that, for all $x, y \in C$, one has $x \leq y$ or $y \leq x$. An *antichain* AC is a subset of X such that, for all $x, y \in AC$, one has $x \not\leq y$ and $y \not\leq x$. Now, consider the problem of decomposing a partial order into a minimum number of chains, such that

each chain contains no more than B elements. We will refer to such a chain as a B-chain. We claim that this problem can be tackled using the approach sketched here. First, one easily verifies that the formulation (1)-(3) goes through by substituting the word "B-chain" for "stable stack" in the definition of the x_k-variables. Second, the efficient solvability of the pricing problem (Lemma 1) depends on the fact that the digraph contains no directed cycles. This property is preserved when we consider partial orders. Finally, notice that also Lemma 2 holds in this more general setting, and it follows that the branching strategy remains valid.

4. An algorithm based on bounded clique-width

In this section we propose an enumeration algorithm that is based on a property of some of the instances encountered at BSS. It turns out that, in some instances, many items have a same length. We exploit this property in this section by assuming that the number of different lengths in an instance is bounded by a given parameter K. In other words, we assume that in the input of the problem an additional parameter K is present; we refer to this variant of problem P as problem $P(K)$.

As a motivating example we first explore the case $K = 2$. We define n_j as the number of items of length j, and we assume that $L_1 < L_2$, where L_i is the i-th length. Further, let $p = n_1 \ mod \ B$ and $q = n_2 \ mod \ B$. Consider now the items with length L_1, and find the width that corresponds to the p^{th} smallest item. Call this width w_1. Then consider the items of length 2, and find the width that corresponds to the q^{th} largest item, and call this width w_2. Notice that the optimal solution of problem $P(2)$ has value $\lceil \frac{n}{B} \rceil$ or $\lceil \frac{n}{B} \rceil + 1$ since $\lceil \frac{n_1}{B} \rceil + \lceil \frac{n_2}{B} \rceil \leq \lceil \frac{n}{B} \rceil + 1$. We now state, without proof, the following proposition:

Proposition 1 *The optimal solution of problem $P(2)$ has value $\lceil \frac{n}{B} \rceil$ iff $w_1 \leq w_2$.*

We now consider problem $P(K)$ in case of an arbitrary value of K. We assume that the lengths are ordered such that $L_1 < L_2 < ... < L_K$. In section 4.1 we focus on the concept of (bounded) clique-width. Section 4.2 describes an exact algorithm for problem $P(K)$.

4.1 Clique-width

A property of graphs that has received wide attention recently is clique-width. This property was first introduced by Courcelle et al. (1993); a related concept called NLC-width has been

introduced by Wanke (1994). The reason for this attention is the fact that important graph theoretic problems (like maximum clique or independent set) can be solved in polynomial time for graphs with bounded clique-width.

Informally, the notion of clique-width of a graph G can be described using the following operations (see Courcelle and Olariu (2001) or Brandstädt et al. (2003) for formal definitions):

- creation of a vertex labelled with some integer i (the vertex is said to have label i); we refer to this as operation 1.

- disjoint union of two vertex-labelled graphs (given $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, let $G = (V_1 \cup V_2, E_1 \cup E_2)$); we refer to this as operation 2.

- adding an edge between each vertex with label i and each vertex with label j, $i \neq j$; we refer to this as operation 3.

- relabel each vertex with label i by label j; we refer to this as operation 4.

The minimum number of labels needed to construct G using these operations is the clique-width of G. Notice that permutation graphs in general have unbounded clique-width (Brandstädt and Lozin 2003). However, in case of $P(K)$ we have the following:

Lemma 3 *A graph associated to an instance of $P(K)$ has clique-width at most $K + 1$.*

Proof: We prove the lemma by exhibiting a sequence of operations. First, we order the vertices according to the width of the associated item in decreasing order. In case of a tie, the vertex with the highest length goes first.

Let vertex i correspond to an item with length L_j; for each vertex $i = 1, \ldots, n$ we perform the following operations:

- create vertex i and label it $K + 1$, using operation 1.

- add vertex i to the graph, using operation 2, i.e., $G := (V \cup i, E)$.

- connect the vertex with label $K+1$ to all vertices with label $j, j+1, \ldots, K$, using operation 3 repeatedly.

- relabel the vertex with label $K + 1$ by label j using operation 4.

Observe that this construction guarantees that each vertex that corresponds to an item with length L_j is connected to all vertices that correspond to items that have length L_j or larger. Thus, the resulting permutation graph corresponds to an instance of $P(K)$. □

Remark: It is easy to verify that the graphs corresponding to instances of $P(K)$ do not have bounded tree-width.

We can now state the following theorem:

Theorem 1 *Problem $P(K)$ is solvable in polynomial time.*

Proof: This result follows from Lemma 3 above and Theorem 2 in Espelage et al. (2001), which states that the problem of partitioning a graph into cliques of bounded size is solvable in polynomial time for graphs with bounded clique-width. □

4.2 An algorithm for $P(K)$

We describe an exact algorithm for problem $P(K)$ that, given B, runs in polynomial time. We now state some preliminaries.

Definition 1 A stack is called *mixed* when it contains items of at least two different lengths. A stack that only contains items of the same length is called *pure*.

Definition 2 The length of an item i is denoted by l_i, and its width by w_i.

Property 1 A solution of problem $P(K)$ is said to have property 1 if each pure stack of a length whose items also occur in mixed stacks of that solution, has size B.

Property 2 A solution of problem $P(K)$ is said to have property 2 if it contains no more than 2^K mixed stacks.

Property 3 A solution of problem $P(K)$ is said to have property 3 if no item r in a mixed stack can be replaced by an item s from a pure stack, with $l_s = l_r$ and $w_s < w_r$.

Definition 3 We call a solution to problem $P(K)$ *minimal* if it simultaneously satisfies properties 1, 2 and 3.

Lemma 4 *There exists an optimal solution to problem $P(K)$ that is minimal.*

Proof: Consider some optimal solution to problem $P(K)$. By interchanging and transferring items, we show that there is an optimal solution that is minimal. If there is an item occurring in a mixed stack that has a length for which there exists a pure stack that is not of size B (i.e., the number of items in the stack is smaller than B), we can transfer this item to

the pure stack. In this way, property 1 is satisfied. If there exists an item r occurring in a mixed stack that can be replaced by an item s from a pure stack with $l_s = l_r$ and $w_s < w_r$, we interchange these items so that property 3 is also satisfied. To see that there exists an optimal solution that satisfies property 2, observe that the maximum number of mixed stacks with different length sets is equal to 2^K. Therefore, if we have found a solution containing more than 2^K mixed stacks, there exist at least two stacks with identical length sets. We now show that, by interchanging some items between these stacks, we can alter the solution such that no stacks with identical length sets are present in the solution. For this, we first define

$$p_i^A : \text{the smallest width of an item of length } i \text{ from stack } A$$

$$q_i^A : \text{the largest width of an item of length } i \text{ from stack } A$$

Observe that, when we discard the size-requirement of a stack, all items of length i can be transferred from a stack A to a stack C if the following two conditions hold:

$$q_{L_i}^A \leq p_{L_{i+1}}^C \tag{5}$$

$$p_{L_i}^A \geq q_{L_{i-1}}^C \tag{6}$$

Now, consider an optimal solution that contains more than 2^K mixed stacks. Assume, without loss of generality, that there exist two stacks A and C with identical length sets $L_1, L_2, ..., L_m$. We claim that there exist two lengths L_i and L_j such that either all items of L_i can be transferred from stack A to stack C, or all items of L_j can be transferred from C to A. This implies that we can construct an alternative optimal solution by interchanging items between A and C such that these stacks no longer have identical length sets.

Without loss of generality we assume that

$$p_{L_m}^A \geq q_{L_{m-1}}^C. \tag{7}$$

(If this would not be the case, we have $p_{L_m}^A < q_{L_{m-1}}^C$; we know, by feasibility of stacks A and C, that $q_{L_{m-1}}^A \leq p_{L_m}^A$ and $q_{L_{m-1}}^C \leq p_{L_m}^C$, and it follows from this that $p_{L_m}^C \geq q_{L_{m-1}}^A$, and we can simply change the order of the two stacks to arrive at our assumption that $p_{L_m}^A \geq q_{L_{m-1}}^C$.)

Since $p_{L_m}^A \geq q_{L_{m-1}}^C$, the items of length L_m from stack A can be transferred to stack C. Now, we have to find a length such that items from stack C can be transferred to stack A. In order to do so, we have to find a length for which conditions (5) and (6) hold. Assume that we cannot find such a length; we then show that we will ultimately arrive at a contradiction, proving that such a length does exist.

Claim 1 *If items of* $L_1, ..., L_j$ *cannot be transferred from* C *to* A *it follows that* $q^C_{L_j} > p^A_{L_{j+1}}, j = 1, ..., m-1$.

Proof: we use induction to prove this claim. Consider the case $j = 1$. We can transfer the items of L_1 from stack C to stack A if $q^C_{L_1} \leq p^A_{L_2}$. Since the items of L_1 are the smallest items, condition (6) does not apply, since there is no length smaller than L_1. We assumed that we could not transfer items from stack C to stack A, so it must hold that $q^C_{L_1} > p^A_{L_2}$. Next, suppose the claim is true for $j = l-1$, is it true for $j = l$? Since we are not able to transfer the items of L_l from C to A, at least one of the inequalities $q^C_{L_l} \leq p^A_{L_{l+1}}$ and $p^C_{L_l} \geq q^A_{L_{l-1}}$ must be violated. But we know by induction that $q^C_{L_{l-1}} > p^A_{L_l}$ which, together with $p^C_{L_l} \geq q^C_{L_{l-1}}$ and $p^A_{L_l} \geq q^A_{L_{l-1}}$, implies $p^C_{L_l} \geq q^A_{L_{l-1}}$. Hence, it follows that $q^A_{L_l} > p^A_{L_{l+1}}$, and this contradicts (7). □

Notice that we could actually replace the upper bound of 2^K on the number of mixed stacks by $2^K - K - 1$, since items of at least two different lengths must be present in a mixed stack. Lemma 4 implies that there exists an optimal solution such that for each $j = 1, ..., K$ the number of items of length L_j present in mixed stacks (denoted by s_j) equals

$$s_j = n_j - \alpha_j * B, \text{for some } \alpha_j \in \{0, 1, \ldots, \lceil \frac{n_j}{B} \rceil\}$$

Now, given a set of possible s_j-values, we enumerate all possible minimal solutions. We do this using the concept of a *partial* solution.

Definition 4 A *partial* solution is a family of 2^K sets of items such that each set corresponds to a feasible stack and such that each item occurs at most once in the family.

To each partial solution we associate a length. That is, the minimum length L_j for which less than s_j items are present in the current partial solution. Further, we can associate to each stack in the partial solution with less than B items, the minimal item of that length L_j that can be feasibly added to that stack.

We now give an algorithm that finds an optimal minimal solution to problem $P(K)$, assuming that a set of s_j-values is given. First, we deal exclusively with constructing the mixed stacks. For this, we start with a partial solution that has 2^K empty stacks, and we gradually fill - in many different ways - these stacks.

Algorithm ENUM:

Step 1. Start with the initial partial solution that consists of 2^K empty stacks. We associate length L_1 to this solution (assuming $s_1 > 0$), and set as minimal item for each stack the smallest item of L_1. Go to step 2.

Step 2. Generate (at most) 2^K new partial solutions by adding for each stack in the old partial solution its minimal item. Notice we get 2^K new partial solutions, since there are 2^K stacks in the old partial solution. Go to step 3.

Step 3. Associate to each partial solution the new minimum length L_j for which less than s_j items are present, and associate to each stack in all solutions its new minimal item. If $\sum_{j=1}^{K} s_j$ items are present in the new partial solution, go to step 4. Otherwise, go to step 2.

Step 4. For each final partial solution, i.e. for each partial solution where $\sum_{j=1}^{K} s_j$ items are assigned, verify whether each stack in the solution is a mixed stack. If not, we simply discard the solution. Else, go to step 5.

Step 5. Complete each final partial solution to a feasible solution by adding the remaining items in pure stacks in a straightforward way. Stop.

By associating a node to each partial solution and connecting two nodes if one partial solution is constructed by adding a single item to the other, a tree results. We refer to this as the tree of partial solutions.

Lemma 5 *A solution generated by algorithm ENUM is minimal.*

Proof: We verify whether a solution found by ENUM satisfies properties 1, 2 and 3. The choice of the s_j-values and step 5 of the algorithm guarantee that each solution found satisfies property 1. Obviously, it satisfies property 2. Now suppose that the solution found does not satisfy property 3, that is, there exists at least one item r that is present in a mixed stack, that could be interchanged with an item s satisfying $l_s = l_r$ and $w_s < w_r$. Let r be the smallest interchangeable item and consider the step in the algorithm when we added item r to a stack. Apparently, we could have added item s at that time. But that implies that item r was not a minimal item for that stack. Hence, such a solution can not have been generated by the algorithm. □

Lemma 6 *Any minimal solution is generated by algorithm ENUM.*

Proof: Consider a minimal solution S that is not generated by ENUM. So each generated solution differs from S. Consider the tree of partial solutions. Let us find a set of paths in this tree: starting with the initial solution, follow a branch to a next partial solution if it puts an item in a stack if in S the same item is in the same stack. Notice that no path makes it till the end (since S was not generated by ENUM). So let us consider a partial solution that has no outgoing branches and which is not final. To this partial solution a length is associated, say the current length.

Consider now the minimal item of the current length of that partial solution that is used in S, and that has not been considered when we followed branches. Say that this item is called item d and that it is in stack j in solution S. This stack j has another item, say item c, serving as minimal item when we look at the branch from our current partial solution to the partial solution where stack j receives an item (if $c = d$, we would have followed that branch). Thus, $c \prec d$. Now, since S is minimal it must use item c somewhere else (if S would not use c at all, we could replace d by c in S, contradicting the minimality of S (property 3)). Say item c is used in stack j' ($j' \neq j$). If we look at the branch from our current partial solution to the partial solution that gives j' another item, we know there is a minimal item that cannot be item c (otherwise we would have followed that branch) Thus, there is another item present in that partial solution, say item b, $b \prec c$. Again, b must be somewhere in S, say in stack j''. Notice that $j'' \neq j'$ (for obvious reasons) but also $j'' \neq j$ (since c is minimal for j and $b \prec c$). Let us look at the stack j'' and its minimal item given our current partial solution. It cannot be b (else we would have followed this branch), so it must be less than b, say a. Thus, a must be in S (otherwise we can interchange contradicting the minimality of S), say in j'''. Again, this stack j''' is different from the previously considered stacks j'', j' and j (otherwise each of the wouldn't have the minimal item they have). Continuing in this way, it leads to the conclusion that S has more than 2^K stacks, contradicting property 2 and hence S is not minimal. □

Theorem 2 *The running time of algorithm ENUM is bounded by* $(2^K)^{B*2^K} * n^K$.

Proof: The complexity of ENUM depends on the number of solutions generated. This number depends on the number of items that are present in mixed stacks. Property 2 implies that

$$\sum_{j=1}^{K} s_j \leq B * 2^K$$

Hence, ENUM cannot generate more than $(2^K)^{B*2^K}$ different solutions. Further, ENUM has to be executed for each possible set of s_j-values. Observe that for each s_j there are $O(\frac{n_j}{B})$ possible values, $j = 1, \ldots, K$, leading to $O(n^K)$ possible sets of s_j-values for a fixed B. The result follows. Notice that, for a fixed B and a fixed K, this is a polynomial time algorithm. □

5. Computational Experiments

In this section we discuss some issues concerning the implementation of algorithms described in this paper, and we show some computational results.

5.1 Implementation Issues

Both algorithms described in this paper are implemented on a 733 MHz computer with 128Mb of intern memory. The algorithms are coded in C++, and in the branch-and-price algorithm, we use LINDO to solve the restricted master problems.

We used two data sets for the computational experiments. The first data set contains x- and y-coordinates from 50 real-life instances provided to us by BSS, and the second data set contains 50 randomly generated instances. In both data sets the number of items ranges from 0 to 200 (see Table 1). The items from the random instances all have lengths and widths uniformly distributed between 0 and 3000, and the number of items also follows a uniform distribution between 0 and 200 items per instance. We use different values for B, ranging from 3 to 15. In the real-life setting from BSS, $B = 12$.

Table 1: Characteristics of the data sets.

#Items	**#Instances (data set 1)**	**#Instances (data set 2)**
0-40	10	9
40-80	15	13
80-120	10	7
120-160	7	12
160-200	8	9

For data set 1, the value of K ranges from 2 to 9, and in most instances (approximately 85%) K equals 2, 3 or 4. For the second data set however, the value of K is very close to n. Thus, the clique-width of the instances of data set 1 is small; this is not guaranteed for the instances of data set 2. Since the running time of the enumeration algorithm is exponential in K, the instances from data set 2 are very hard to solve for ENUM. In fact, none of the

instances could be solved by ENUM in less than one hour of computation time, so for ENUM we present only the results of the first data set.

In the branch-and-price algorithm, we use a heuristic to find a good starting solution, before starting the actual branch-and-price procedure. This starting solution is computed in a very straightforward way: all items are ordered, first according to their length (increasing) and second according to their width (also increasing). We start with the first item and put it in a stack. Then we simply go down the list, and if an item can be added to the current stack, we add it, and otherwise we continue with the next item. If a stack contains B items, or if we are at the end of the list, we start a new stack with the first available item and start this procedure over. To determine whether a solution generated by this heuristic is optimal, we use the lower bound $\lceil \frac{n}{B} \rceil$.

In the pricing problem, when trying to find new variables with negative reduced costs, we add one variable in each iteration of the longest path procedure. This is the variable with reduced costs that are the most negative.

In the enumeration algorithm, we first compute a lower and an upper bound. The lower bound equals $\lceil \frac{n}{B} \rceil$, and the upper bound is equal to $\lceil \frac{n_1}{B} \rceil + \ldots + \lceil \frac{n_K}{B} \rceil$. If these bounds coincide, there exists an optimal solution consisting of only pure stacks, and we do not need to run ENUM to find a solution.

Apart from computing the LP-relaxation and $\lceil \frac{n}{B} \rceil$, we computed a third lower bound, AC. AC stands for the size of a maximum antichain (see section 3.4). In other words, AC is the optimal value of problem P in case there is no restriction on B (i.e. $B = n$).

5.2 Results

The results for the first data set are shown in Table 2 and the results for the second data set are shown in Table 3. In the first two columns we give the value of B and a range for the number of items. The following three columns give the values of three lower bounds, namely $\lceil \frac{n}{B} \rceil$, the size of a maximum antichain and the solution of the LP-relaxation. The column labelled 'OPT' denotes the optimal integer solution. In the last columns we give the number of branching nodes visited in the search tree, and the computation time in seconds. In Table 2 these last two values are given both for the branch-and-price algorithm as for ENUM. Table 3 only shows the results of the branch-and-price algorithm. Notice that all values are average values over all test instances in the specific range.

Table 2: Results for real-life instances

B	n	$\lceil \frac{n}{B} \rceil$	AC	LP	OPT	Branch&Price		ENUM	
						Nodes	Time	Nodes	Time
3	≤ 40	11,90	2,50	11,47	11,90	0,00	0,00	3,70	0,00
	≤ 80	19,13	3,33	18,91	19,13	29,07	0,69	33,93	2,52
	≤ 120	36,90	2,90	36,40	36,90	0,00	0,01	2,90	0,01
	≤ 160	45,86	3,00	45,43	45,86	0,00	0,02	3,14	2,04
	≤ 200	60,50	2,38	60,08	60,50	0,00	0,05	0,63	0,00
6	≤ 40	6,20	2,50	5,72	6,20	0,00	0,00	7,40	0,00
	≤ 80	9,73	3,33	9,48	9,80	0,07	0,03	23,87	0,00
	≤ 120	18,70	2,90	18,20	18,70	0,00	0,01	11,80	0,00
	≤ 160	23,29	3,00	22,71	23,29	0,00	0,02	4,29	0,01
	≤ 200	30,38	2,38	30,04	30,38	0,00	0,05	4,63	0,00
9	≤ 40	4,40	2,50	3,97	4,40	0,10	0,02	11,00	0,00
	≤ 80	6,67	3,33	6,35	6,67	10,08	4,87	24,27	0,00
	≤ 120	12,80	2,90	12,18	12,80	5,40	0,23	5,80	0,00
	≤ 160	15,57	3,00	15,16	15,57	1,43	2,26	20,57	0,00
	≤ 200	20,38	2,38	20,04	20,38	7,75	0,65	11,00	0,00
12	≤ 40	3,50	2,50	3,23	3,70	4,80	0,56	14,00	0,00
	≤ 80	5,13	3,33	4,82	5,27	9,00	6,64	25,94	0,00
	≤ 120	9,60	2,90	9,10	9,60	0,00	0,01	25,90	0,00
	≤ 160	11,86	3,00	11,48	11,86	7,57	0,96	31,43	0,00
	≤ 200	15,50	2,38	15,02	15,50	8,75	2,60	51,88	0,01
15	≤ 40	2,70	2,50	2,81	3,10	8,10	5,72	22,10	0,00
	≤ 80	4,33	3,33	3,96	4,40	0,07	1,35	29,07	0,00
	≤ 120	7,60	2,90	7,29	7,60	4,90	5,13	28,00	0,00
	≤ 160	9,43	3,00	9,10	9,43	10,71	6,89	61,14	0,00
	≤ 200	12,63	2,38	12,19	12,96	15,50	7,78	55,88	0,01

In the tables with the results we see that, in a number of cases the number of branching nodes is equal to 0. For the branch-and-price algorithm, this means that the solution found by the heuristic equals $\lceil \frac{n}{B} \rceil$ (this happened 217 out of 250 times in Table 2 and 12 out of 250 times in Table 3). For the ENUM algorithm it means that the lower- and upper bound computed at the start of the algorithm are the same, which means that there exists an optimal solution with zero items in mixed packages (this happened 90 out of 250 times in Table 2).

From Table 2 we conclude that the instances from data set 1 can be solved optimally very fast by both algorithms; 90% of the instances is solved in less than one second for the branch-and-price algorithm, and for ENUM even 99% of all instances is solved in less than one second. One reason for the good performance of the branch-and-price algorithm is that

Table 3: Results for random instances

B	n	$\lceil \frac{n}{B} \rceil$	AC	LP	OPT	Nodes	Time
3	≤ 40	9,89	8,67	10,11	10,44	15,11	0,10
	≤ 80	18,77	11,69	18,54	18,85	46,08	7,23
	≤ 120	33,57	16,14	33,38	33,57	101,14	12,92
	≤ 160	47,92	20,83	47,47	47,92	165,75	41,43
	≤ 200	61,11	23,22	60,85	61,11	305,56	120,10
6	≤ 40	5,22	8,67	8,67	8,67	1,00	0,02
	≤ 80	9,62	11,69	11,72	11,77	12,31	4,50
	≤ 120	17,00	16,14	17,10	17,29	69,14	82,53
	≤ 160	24,17	20,83	23,90	24,25	51,83	65,36
	≤ 200	30,67	23,22	30,50	30,67	62,67	112,78
9	≤ 40	3,78	8,67	8,67	8,67	1,00	0,02
	≤ 80	6,62	11,69	11,69	11,69	1,00	0,14
	≤ 120	11,57	16,14	16,14	16,14	9,57	9,25
	≤ 160	16,33	20,83	21,42	21,42	10,67	25,40
	≤ 200	20,67	23,22	26,00	26,00	15,22	97,62
12	≤ 40	2,78	8,67	8,67	8,67	1,00	0,03
	≤ 80	5,08	11,69	11,69	11,69	1,00	0,37
	≤ 120	8,71	16,14	16,14	16,14	1,00	3,68
	≤ 160	12,42	20,83	20,83	20,83	13,17	43,54
	≤ 200	15,56	23,22	23,22	23,22	1,00	78,54
15	≤ 40	2,44	8,67	8,67	8,67	1,00	0,03
	≤ 80	4,23	11,69	11,69	11,69	1,00	0,37
	≤ 120	7,29	16,14	16,14	16,14	1,00	3,98
	≤ 160	10,00	20,83	20,83	20,83	1,00	16,20
	≤ 200	12,67	23,22	23,22	23,22	1,00	85,93

in 86,8% of the instances, the heuristic for finding an initial solution in the branch-and-price algorithm provides us with an optimal solution which equals $\lceil \frac{n}{B} \rceil$. Indeed, the quality of the lower bound $\lceil \frac{n}{B} \rceil$ for data set 1 is striking. For the ENUM algorithm, an optimal solution is found without having to branch in 36,0% of the instances. Thus, in most cases ENUM has to be executed, and then it finds an optimal solution very fast, i.e., usually faster than the branch-and-price algorithm. As described before, from the results it is also clear that both $\lceil \frac{n}{B} \rceil$ and LP are good lower bounds for the integer optimum; the value of AC however, is in many cases far from the optimum.

When we look at the results from the random instances in Table 3, we see that the computation times of the branch-and-price algorithm are slower than those from the real-life instances. Further, the lower bound from the LP-relaxation and the value of AC are very close to the integer optimum; for large B ($B \geq 9$) they even coincide. Not surprisingly, the

lower bound $\lceil \frac{n}{B} \rceil$ performs here much worse, especially for large B. The heuristic for finding an initial solution performs much worse compared to the results from the first data set: only for 4.8% of the instances the heuristic finds an optimal solution equalling $\lceil \frac{n}{B} \rceil$.

From the results we conclude that there is a clear difference between the real-life and the random instances. For the real-life instances, the computation times are much faster than for the random instances. Also, when we look at the results from the random instances, we see that as the number of items increases, the computation times generally increase for each value of B, while this is not so clear from the results of the real-life instances. These differences in performance can be explained by the fact that, in the real-life instances the number of different lengths is small; a property that is not present in the randomly generated instances. Because of this structure, the performance of our heuristic to find an initial solution performs much better for the real-life instances.

Finally, the results show that the solution to the LP-relaxation provides us for all instances with a very good lower bound on the integer optimum. In all instances, the value of the integer optimum (V_{IP}) is smaller than or equal to the value of the LP-solution (V_{LP}) rounded up: $V_{IP} \leq \lceil V_{LP} \rceil$. This is not true in general: one can easily construct instances such that $V_{IP} = \frac{4}{3} V_{LP}$.

6. Conclusion

In this paper we described two exact algorithms for partitioning a permutation graph into cliques of bounded size. The first algorithm is a branch-and-price algorithm, based on an integer programming formulation. The pricing problem can be formulated as a longest path problem and can be solved efficiently by dynamic programming. The second algorithm is an enumeration algorithm based on the concept of bounded clique-width. This algorithm was motivated by a special structure that is present in the real-life instances that were used for computational experiments. From the computational results we conclude that both the real-life and the random instances can be solved satisfactorily by the branch-and-price algorithm. The enumeration algorithm performs really well in case of the real-life instances (99% of the instances are solved within a second), but the random instances cannot be solved efficiently, due to the large number of different lengths in the input. From these results we also see that the LP-relaxation provides us with a good lower bound on the integer optimum.

Acknowledgements

This research was partially supported by EU Thematic Network APPOL II, IST-2001-32007. We would like to thank Eric Stinges from Bruynzeel Storage Systems for providing us with the data for the test instances.

References

Baker, B.S., E.G. Coffman, Jr. 1996. Mutual exclusion scheduling. *Theoret. Comput. Sci.*, **162**, 225-245.

Barnhart, C., E.D. Johnson, G.L. Nemhauser, M.W.P. Salvelsbergh, P.H. Vance. 1998. Branch-and-Price: column generation for solving huge integer programs. *Oper. Res.*, **46**, 316-329.

Bischoff, E.E. 1991. Stability aspects of pallet loading. *OR Spektrum*, **13**, 189-197.

Brandstädt, A., F.F. Dragan, H.-O. Le and R. Mosca. 2003. New graph classes of bounded clique-width. To appear in *Theory of Comput. Systems.*

Brandstädt, A., V.V. Lozin. 2003. On the linear structure and clique-width of bipartite permutation graphs. *Ars Combinatoria*, **LXVII**, 273-281.

Courcelle, B., J. Engelfriet, G. Rozenberg. 1993. Handle-rewriting hypergraph grammars. *J. of Comput. and System Sci.*, **46**, 218-270.

Courcelle, B., S. Olariu. 2001. Upper bounds to the clique-width of graphs. *Discrete Appl. Math.*, **101**, 77-114.

Dilworth, R.P. 1950. A decomposition theorem for partially ordered sets. *Ann. of Math.*, **51**, 161-166.

Dyckhoff, H. 1990. A typology of cutting and packing problems. *Eur. J. of Oper. Res.*, **44**, 145-159.

Espelage, W., F. Gurski, E. Wanke. 2001. How to solve NP-hard graph problems on clique-width bounded graphs in polynomial time. In *Proceedings of WG '01*, LNCS, **2204**, 117-128, Springer Verlag.

Felsner, S., L. Wernish. 1998. Maximum k-chains in planar point sets: combinatorial structure and algorithms. *SIAM J. of Comput.*, **28**, 192-209.

G, Y.-G., M.-K. Kang. 2001. A fast algorithm for two-dimensional pallet loading problems of large size. *Eur. J. of Oper. Res.*, **134**, 193-202.

Golumbic, M.C. 1980. *Algorithmic graph theory and perfect graphs.* Academic Press, New York.

Jansen, K. 2003. The mutual exclusion scheduling problem for permutation and comparability graphs. *Inform. and Comput.*, **180**, 71-81.

Letchford, A.N., A. Amaral. 2001. Analysis of upper bounds for the pallet loading problem. *Eur. J. of Oper. Res.*, **132**, 582-593.

Moonen, L.S. 2001. *Optimaliseren van een productieproces.* Master's thesis of Maastricht University (in Dutch).

Morabito, R., S. Morales. 1998. A simple and effective procedure for the manufacturer's pallet loading problem. *J. of the Oper. Res. Soc.*, **49**, 819-828.

Scheithauer, G., J. Terno. 1996a. The G4-heuristic for the pallet loading problem. *J. of the Oper. Res. Soc.*, **47**, 511-522.

Scheithauer, G., J. Terno. 1996b. A heuristic approach for solving the multi-pallet packing problem. Working Paper, Dresden University of Technology.

Terno, J., G. Scheithauer, U. Sommerweiß, J. Riehme. 2000. An efficient approach for the multi-pallet loading problem. *Eur. J. of Oper. Res.*, **123**, 372-381.

Trotter, W.T. 1992. *Combinatorics and partially ordered sets: dimension theory.* The John Hopkins University Press, Baltimore.

Wanke, E. 1994. k-NLC graphs under polynomial algorithms. *Discrete Appl. Math.*, **54**, 251-266.