Math. Prog. Comp. (2017) 9:39–59
DOI 10.1007/s12532-016-0106-x

FULL LENGTH PAPER

Fast separation for the three-index assignment problem

Trivikram Dokka[1] · **Ioannis Mourtos**[2] · **Frits C. R. Spieksma**[3]

Received: 16 December 2014 / Accepted: 21 April 2016 / Published online: 7 May 2016

Abstract A critical step in a cutting plane algorithm is separation, i.e., establishing whether a given vector x violates an inequality belonging to a specific class. It is customary to express the time complexity of a separation algorithm in the number of variables n. Here, we argue that a separation algorithm may instead process the vector containing the positive components of x, denoted as $supp(x)$, which offers a more compact representation, especially if x is sparse; we also propose to express the time complexity in terms of $|supp(x)|$. Although several well-known separation algorithms exploit the sparsity of x, we revisit this idea in order to take sparsity explicitly into account in the time-complexity of separation and also design faster algorithms. We apply this approach to two classes of facet-defining inequalities for the three-index assignment problem, and obtain separation algorithms whose time complexity is linear in $|supp(x)|$ instead of n. We indicate that this can be generalized to the axial k-index assignment problem and we show empirically how the separation algorithms

This paper is an improved version of an extended abstract that appeared as "Fast separation algorithms for three index assignment problems" in the proceedings of ISCO 2012, LNCS 7422, pp. 189–200, 2012. This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF)—Research Funding Program: Thales: Investing in knowledge society through the European Social Fund (MIS: 380232), and this research has been supported by the Interuniversity Attraction Poles Programme initiated by the Belgian Science Policy.

✉ Trivikram Dokka
t.dokka@lancaster.ac.uk

1 Department of Management Science, Lancaster University Management School, Lancaster LA1 14X, UK

2 Department of Management Science and Technology, Athens University of Economics and Business, 76 Patission Ave., 1043 34 Athens, Greece

3 ORSTAT, KU Leuven, Naamsestraat 69, 3000 Leuven, Belgium

exploiting sparsity improve on existing ones by running them on the largest instances reported in the literature.

Mathematics Subject Classification 90C10 · 90C27 · 90C57

1 Motivation

Cutting plane algorithms constitute a fundamental way of solving combinatorial optimization problems. Typically, in such an approach, a specific combinatorial optimization problem is formulated as an integer linear program (ILP) of the form $\min\{c^T x : Ax = b, x \in Z^n_+\}$, where x denotes an n-dimensional column vector of variables and A, b, and c are matrices of appropriate dimension. The convex hull of the feasible solutions is defined by the corresponding polyhedron $P_I = \text{conv}\{x \in Z^n_+ : Ax = b\}$. Then, there is interest in identifying classes of inequalities that are valid for P_I and, preferably, facet-defining (for related background see, for example, [23]). Although identifying such families provides a (partial) characterization of P_I, the computational benefit of these families, in terms of finding $z = \min\{c^T x : x \in P_I\}$, can be realized only if these inequalities can efficiently be added to the linear programming (LP) relaxation $P_L = \min\{c^T x : Ax = b, x \geq 0\}$.

Since there can be many inequalities within a family, their addition within a cutting plane scheme should be made 'on demand', i.e., an inequality should added to the current LP-relaxation only if violated by a specific vector $x \in \mathbb{R}^n$. The problem of determining whether such a vector violates an inequality of a specific family is called the *separation problem* for this family and an algorithm solving it is called a *separation algorithm*. Hence the importance of designing, typically family-specific, separation algorithms that should be of low computational complexity.

It is quite customary to express the complexity of a separation algorithm in terms of n, the dimension of the vector x. This seems reasonable since, at the very least, one would need to inspect every entry of the vector x to decide whether a violated inequality exists. In fact, separation algorithms with a complexity of $O(n)$ have been called "best-possible" for the 3-index assignment problem [4,21]. If the input to a separation algorithm is an optimal solution to the current LP-relaxation, i.e., a vector $x \in P_L \backslash P_I$, such an input is typically *sparse* in the sense that few entries of x are positive. Thus, replacing x by its support $supp(x)$ offers a more compact representation of the input. This leads to the following question: when the input to a separation algorithm is $supp(x)$, is it possible to obtain faster separation algorithms and express their time-complexity in terms of $|supp(x)|$?

Clearly, the input to a separation algorithm cannot be restricted to x-vectors that represent extreme vertices of the LP-relaxation at hand. Indeed, any x-vector is a potential input to the separation algorithm. For example, one might be interested in separating vectors that do not represent vertices, because the solution of the LP-relaxation is obtained via an interior point method, or because the vector to be separated is a vertex from a different relaxation. We point out that our results hold in such cases as well, since any vector x can be encoded by listing the $|supp(x)|$ positive components of x. However, it is true that the advantage of this more compact input in terms

of algorithm's running time disappears. Thus, representing a vector x by using the support leads to speedups when the vector is sparse. This typically happens when a separation algorithm is incorporated within a *Branch & Cut* or a cutting plane scheme, since then the separation algorithm receives as input not any $x \in R^n$, but a vertex $x \in P_L \setminus P_I$, which is sparse particularly for problems having much more variables than constraints.

We elaborate on the idea of using sparsity in Sect. 2, where we also discuss several known separation algorithms that exploit sparsity thus motivating our work. We apply this idea to the (axial) 3-index assignment problem (Sect. 3) and show that linear-time separation in terms of $supp(x)$ is plausible (Sect. 5). That is, we obtain algorithms of time $O(|supp(x)|)$, which are much faster than existing ones of $O(n)$ time [4] for separating not only vertices of the LP-relaxation but any sparse vector; the algorithms of [4] neither exploit sparsity nor assume that the input vector is a vertex. We also propose such algorithms for the k-index assignment problem (Sect. 6). Testing the performance of our separation algorithms on large literature instances for the 3-index assignment problem offers strong empirical support for our effort (Sect. 7).

2 Exploiting sparsity in separation algorithms

Traditionally, to describe the input needed for an algorithm separating a specific class of inequalities, the n entries of the vector $x = (x_1, \ldots, x_n)$ are provided. We propose here an alternative measure of the input: the cardinality of the support of the vector x, denoted by $T = |supp(x)|$. That is, we propose to express the running time of a separation algorithm in terms of T. This allows us to exploit the fact that the input received by a separation algorithm is not any vector x but, typically in cutting plane schemes, the vertex of a polyhedron having several zero entries.

It could be of help to consider the following (imaginary) setting. Suppose that several instances of some combinatorial optimization problem are being solved in parallel through a cutting-plane approach, using multiple computers. However, only a single computer is available for the separation routine. Then, this routine receives several fractional vectors corresponding to the instances being solved by the cutting-plane approach. In such a situation, one easily imagines that the sole input to the separation routine are the positive components of the vector x, i.e., $supp(x)$. In other words, we examine the decision problem with respect to a certain class of valid inequalities C, as follows:

- INPUT: $support(x)$
- QUESTION: Is there an inequality violated by x in C?

Let us emphasize that this idea is not new, since there are many examples in the literature where sparsity is implicitly used in the design of separation algorithms. The oldest such example concerns the traveling salesman problem (TSP) and the well-known class of subtour elimination constraints [3]. Assuming a binary variable x_e per edge e, these inequalities are formulated as

$$\sum_{e:|e\cap S|=1} x_e \geq 2 \quad \text{for any nonempty proper subset } S \text{ of cities.}$$

To solve the separation problem for these inequalities, the solution of the LP-relaxation of the traditional TSP formulation, say x^*, is used to build a support graph $G^* = (V^*, E^*)$ such that $e \in E^*$ if and only if $x_e^* > 0$. One particular separation heuristic ([3, see p. 159], called the *parametric connectivity* heuristic, uses an edge in $e \in E^*$ only if $x_e^* > \epsilon$ for a fixed $\epsilon > 0$, while the Padberg–Rinaldi exact separation algorithm computes a minimum cut in G^*. Another important class of inequalities for the TSP, namely the comb and blossom inequalities, are separated by the *odd component heuristic* [3] that uses a graph with vertex set is V and edge set is $\{e \in E^* : 0 < x_e^* < 1\}$.

Another indicative case is the separation of *odd-cycle* inequalities for the stable set problem. If $\tilde{C}$ is the set of all induced odd cycles in the underlying graph, such an inequality is

$$\sum_{e\in C} x_e \leq \frac{|C|-1}{2} \quad \text{for each } C \in \tilde{C}. \tag{1}$$

To separate odd-cycle inequalities for a given vector x^*, shortest paths are computed in an bipartite graph H [23, pp. 1186–1187]). The complexity of this separation algorithm is $O(|V| \cdot |E| \cdot \log|V|)$ [9,10], although zero-weighted edges may result in nodes and edges of H not required in the shortest path calculation [22].

Regarding the index selection problem, formulated as a set packing problem [8], the separation of lifted odd-hole inequalities is accomplished by determining a minimum-weight odd cycle in a graph having one node for each fractional variable, where the weight of each edge depends only on the positive components of x^*. The number of variables is much larger than the number of constraint, thus any x^* produced by the LP-relaxation is sparse because the number of its positive components is bounded by the number of constraints.

Further examples of separation procedures that use only the fractional components of the solution to the LP-relaxation have been proposed for the $0-1$ knapsack problem [16], the winner determination problem in combinatorial auctions [18], the sequence alignment problem [17], the time-indexed formulation of single-machine scheduling [12] and the pallet loading problem [1].

Overall, it is common that separation algorithms receive as input a vertex of the LP-relaxation and that such a vector is sparse, particularly when the number of variables is larger than the number of constraints.

3 The 3-index assignment polytope (3AP)

The 3-index assignment problem, defined on three disjoint n-sets I, J, K and a weight function $w : I \times J \times K \longrightarrow \mathbb{R}$, asks for a collection of triples $M \subseteq I \times J \times K$ such that each element of each set appears in exactly one triple, and the function w is minimized (over all possible such collections). Its formulation as an integer linear program is

$$\min \sum_{i\in I}\sum_{j\in J}\sum_{k\in K} w_{ijk}x_{ijk}$$

$$\text{s.t.} \quad \sum_{j\in J}\sum_{k\in K} x_{ijk} = 1 \quad \forall i \in I, \tag{2}$$

$$\sum_{i\in I}\sum_{k\in K} x_{ijk} = 1 \quad \forall j \in J, \tag{3}$$

$$\sum_{i\in I}\sum_{j\in J} x_{ijk} = 1 \quad \forall k \in K, \tag{4}$$

$$x_{ijk} \in \{0,1\} \qquad \forall i \in I, j \in J, k \in K. \tag{5}$$

Let A^n denote the (0, 1) matrix corresponding to the constraints (2)–(4), which has n^3 columns and $3n$ rows. Notice that hereafter n denotes the cardinality of each set being 'assigned', hence the number of variables is n^3. Then, the 3-index assignment polytope is $P_I = \text{conv}\{x \in \{0,1\}^{n^3} : A^n x = e\}$, while its LP-relaxation is $P^n = \{x \in R^{n^3} : A^n x = e, x \geq 0\}$. For a survey on the 3-index assignment problem, see [24].

The first investigation of the facial structure of P_I appears in [5]. Let us describe the two families of facet-defining inequalities presented in [5]. The *column intersection* graph of A^n, namely $G(V, E)$, has a node for each column of A^n and an edge for every pair of columns that have a $+1$ entry in the same row. Notice that a column contains three $+1$'s. We define the intersection of two columns c and d as the set of rows of A^n such that each row in the set has a +1 entry in column c and in column d; this is denoted by $|c \cap d|$. It is easy to see that $V = I \times J \times K$ and $E = \{(c, d) : \{c, d\} \subseteq V, |c \cap d| \geq 1\}$, i.e., a node in V corresponds to a triple, and two nodes are connected if the corresponding triples share some index. A *clique* is a *maximal complete* subgraph.

Clearly, a clique in G corresponds to a valid inequality with right-hand side 1. In fact, $G(V, E)$ contains two types of cliques, each yielding a family of inequalities that are facet-defining for P_I. To formally define the two types of clique inequalities, let

- for each $c \in V : Q(c) = \{d \in V : |c \cap d| \geq 2\}$,
- for each $c \in V : \text{co}Q(c) = \{d \in V : |c \cap d| = 1\}$, and
- for each $c, d \in V$ with $|c \cap d| = 0 : Q(c, d) = \{c\} \cup \{Q(d) \cap \text{co}Q(c)\}$.

Thus, $Q(c)$ is the set of triples sharing at least two indices with triple c, while $\text{co}Q(c)$ is the set of triples that has exactly one index in common with triple c. Finally, when given two disjoint triples c and d, $Q(c, d)$ is the set of triples that has two indices in common with d, and one with c, together with triple c. Notice that $Q(c, d)$ has exactly four elements. As usual, we write $x(A)$ for $\sum_{q\in A} x_q$.

Definition 1 For each $c \in V$, the facet-defining inequality $x(Q(c)) \leq 1$ is called a *clique inequality of type I.*

Once we organize the variables x_{ijk} in a three-dimensional array (a cube), a clique inequality of type I can be seen as the sum of those x-variables that lie on the three "axes" through a particular cell (see Fig. 1 for a geometric illustration).

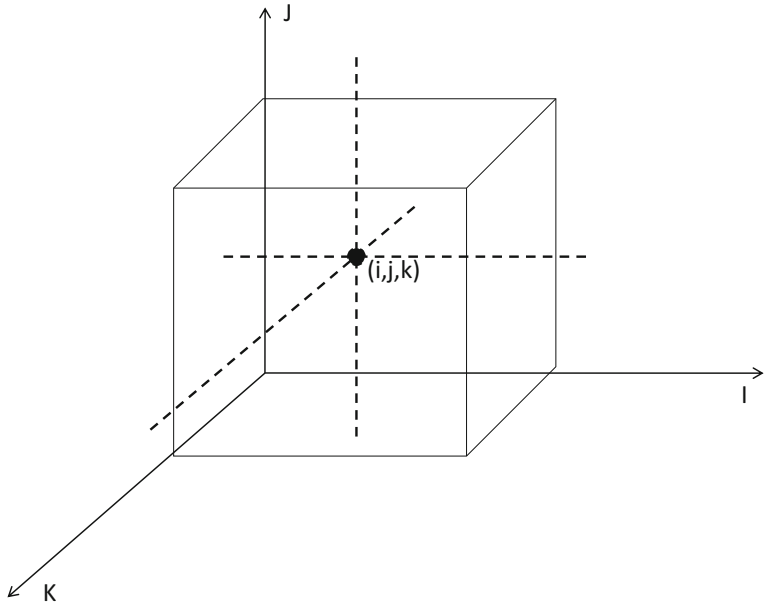

Fig. 1 Geometric illustration of a clique inequality of type I; the three dotted axes correspond to the variables in this inequality

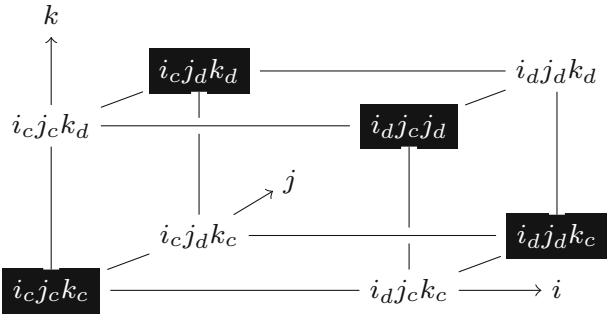

Fig. 2 Geometric illustration of a clique inequality of type II; the four *highlighted* cells correspond to the four variables in this inequality

Definition 2 For each $c, d \in V$ with $|c \cap d| = 0$, the facet-defining inequality $x(Q(c, d)) \leq 1$ is called a *clique inequality of type II*.

An illustration of a clique inequality of type II is given in Fig. 2.

The separation of inequalities induced by cliques of type I and II was first treated in [5] through algorithms of $O(n^4)$ time complexity. Improved $O(n^3)$ algorithms (i.e., of complexity linear in the number of variables) appear in Balas and Qi [4], in which they are characterized as "best-possible". We illustrate them also here as Algorithms 1 and 2.

To discuss how improved separation algorithms can be obtained by using a compact input, let us recall from linear programming that the number of non-zero variables in a vertex of the LP-relaxation cannot exceed the number of constraints.

Remark 1 A solution corresponding to a vertex of P^n has at most $3n$ non-zero variables.

Given a vector $x \in P^n$, let $supp(x) = \{c \in V : x_c > 0\}$ and $T = |supp(x)|$. To ease the presentation, let us assume that $supp(x)$ contains the triples indexing the non-zero entries of x and the corresponding (positive) fraction per triple.

In the next sections we discuss some preliminaries and then obtain algorithms of $O(T)$ time, i.e., of $O(n)$ time when x is vertex of P^n by Remark 1. This is significantly smaller than the $O(n^3)$ time complexity of Algorithms 1 and 2; notice that $O(T)$ reaches $O(n^3)$ only if x is 'fully dense'. Our algorithms achieve such a speed-up by pre-calculating certain sums of x's entries and, as expected, by utilising the fact that the input vector contains only non-zero values. Thus their main difference from Algorithms 1 and 2 is their effective use of sparsity.

Algorithm 1 Balas & Qi Separation Algorithm for cliques of type I

{Input: the vector x and an integer $v \geq 4$}
for all $c \in V$ **do**
  $d_c := 0$;
end for
for all $c \in V$ **do**
  if $x_c \geq \frac{1}{v-n}$ **then**
    for all $t \in Q(c)$ **do**
      $d_t := d_t + x_c$;
      if $d_t > 1$ **then**
        return $x(Q(t)) \leq 1$ as violated;
      end if
    end for
  end if
end for
for all $c \in V$ **do**
  if $d_c > \frac{v-3}{3}$ **then**
    compute $x(Q(c))$;
    if $x(Q(c)) > 1$ **then**
      return $x(Q(c)) \leq 1$ as violated;
    end if
  end if
end for

Algorithm 2 Balas & Qi Separation Algorithm for cliques of type II

{Input: the vector x}
for all $c \in V : \frac{1}{4} < x_c < 1$ **do**
  for all $t \in V : |t \cap c| = 1$ **do**
    if $x_t > \frac{1-x_c}{3}$ **then**
      for all $d \in V : |d \cap c| = 0$ and $|d \cap t| = 2$ **do**
        compute $x(Q(c, d))$;
        if $x(Q(c, d)) > 1$ **then**
          return $x(Q(c, d)) \leq 1$ as violated;
        end if
      end for
    end if
  end for
end for

4 Notation and preliminaries

Before describing the separation algorithms exploiting sparsity for the 3AP, we give some intuition on their structure. An inequality in each of the classes examined here corresponds to a set of 'core' elements $C \subset I \cup J \cup K$, in the sense that given C and $supp(x)$ we can explicitly compute the left-hand side of the inequality. For example, each clique inequality of type I corresponds to a C having one element from each of I, J and K. Similarly, for a clique inequality of type II, the corresponding C contains two elements from each of I, J and K. Therefore, a natural idea when searching for a violated inequality from a particular class is to identify the set C the inequality corresponds to. A naive algorithm would check all possible C's; clearly, this can be prohibitively expensive as the cardinality of C and n increases.

The algorithms presented here find the C of a violated inequality by exploiting the properties of a point in P^n. To show these properties, we introduce some notation. Define, for $x \in P^n$:

- for each $j \in J, k \in K$: $\text{SUM}I(j,k) = \sum_{i \in I} x_{ijk}$,
- for each $i \in I, k \in K$: $\text{SUM}J(i,k) = \sum_{j \in J} x_{ijk}$,
- for each $i \in I, j \in J$: $\text{SUM}K(i,j) = \sum_{k \in K} x_{ijk}$.

Informally, each of these quantities corresponds to an axis in the geometric description given in Fig. 1.

Lemma 1 *Let $\epsilon > 0$. For all $i \in I$, the number of pairs (i, j)* (respectively number of pairs (i,k)), *$j \in J$ ($k \in K$), such that $\text{SUM}K(i,j) > \epsilon$* (respectively $\text{SUM}J(i,k) > \epsilon$) *is at most $\frac{1}{\epsilon}n$.*

Proof If not, at least $\frac{1}{\epsilon}n$ pairs $(i, j) \in I \times J$ have $\text{SUM}K(i,j) > \epsilon$. This implies we have

$$\sum_i \sum_j \text{SUM}K(i,j) > \frac{1}{\epsilon} \cdot \epsilon n = n = \sum_i \sum_j \sum_k x_{ijk} = \sum_i \sum_j \text{SUM}K(i,j)$$

which is a contradiction. □

Further, we define

- for each $i \in I$: $J(i) = \{j \in J : \text{SUM}K(i,j) > \frac{1}{3}\}$.

Notice that $|J(i)| \leq 2$ for each $i \in I$, due to (2) and Lemma 1.

We can compute all aforementioned quantities by scanning $supp(x)$ only once. For example, to compute $\text{SUM}I(j,k)$ for a specific j, k, it suffices to add the values of x_v's for each triple $v \in V$ having $|v \cap \{j,k\}| = 2$ encountered while scanning $supp(x)$. Algorithm 3 describes all steps in detail.

Algorithm 3 Calculating SUM$I(j,k)$ and $J(i)$

```
{Input: supp(x)}
for all c ∈ supp(x) with c = (i_c, j_c, k_c) do
  if variable SUMK(i_c, j_c) does not exist then
    create variable SUMK(i_c, j_c);
    SUMK(i_c, j_c) := 0;
  end if
  SUMK(i_c, j_c) := SUMK(i_c, j_c) + x_c;
  if SUMK(i_c, j_c) > 1/3 then
    if J(i_c) does not exist then
      create variable J(i_c);
    end if
    J(i_c) := J(i_c) ∪ {j_c};
  end if
end for
```

Lemma 2 *For an arbitrary $x \in P^n$, all positive SUM$I(j,k)$, SUM$J(i,k)$ and SUM$K(i,j)$ values* (for $i \in I, j \in J, k \in K$) *as well as the sets $J(i)$ for each $i \in I$ can be calculated in $O(T)$ steps.*

Proof Algorithm 3 shows how to compute SUM$I(j,k)$, for each j,k, as well as $J(i)$ for each i in $O(T)$ steps. A straightforward extension of Algorithm 3 implies the result. □

For each $i \in I$ (resp $j \in J, k \in K$), let $val(i)$ (resp. $val(j)$, $val(k)$) be the value of the third largest variable indexed by a triple containing i (resp. j,k). Then we define for any $x \in P^n$:

- for each $i \in I$: $A(i) = \{(i,j,k) \in V : x_{ijk} \geq val(i), j \in J, k \in K\}$,
- for each $j \in J$: $B(j) = \{(i,j,k) \in V : x_{ijk} \geq val(j), i \in I, k \in K\}$,
- for each $k \in K$: $C(k) = \{(i,j,k) \in V : x_{ijk} \geq val(k), i \in I, j \in J\}$.

It follows that the sets $A(i)$ (resp. $B(j)$, $C(k)$) contain the largest three such variables, i.e. the three largest variables indexed by a triple containing i (resp. j,k). In addition, we define

- for each $i \in I$: $A^{>\frac{1}{4}}(i) = \{(i,j,k) \in V : x_{ijk} > \frac{1}{4}, j \in J, k \in K\}$,
- for each $j \in J$: $B^{>\frac{1}{4}}(j) = \{(i,j,k) \in V : x_{ijk} > \frac{1}{4}, i \in I, k \in K\}$,
- for each $k \in K$: $C^{>\frac{1}{4}}(k) = \{(i,j,k) \in V : x_{ijk} > \frac{1}{4}, i \in I, j \in J\}$.

Observe further that all elements from a specific $A^{>\frac{1}{4}}(i)$ (resp. $B^{>\frac{1}{4}}(j)$, $C^{>\frac{1}{4}}(k)$) occur in a single equality among equalities (2) (resp. (3), (4)). Since the right-hand side of any equality is 1, and each element in any such set has value strictly larger than $\frac{1}{4}$, it follows that $|A^{>\frac{1}{4}}(i)| \leq 3$, $|B^{>\frac{1}{4}}(j)| \leq 3$, and $|C^{>\frac{1}{4}}(k)| \leq 3$, for $i \in I, j \in J, k \in K$. In fact, we can state the following.

Lemma 3 *For every fixed $\epsilon > 0$ and $i \in I$, the number of triples (i,j,k), with $j \in J$ and $k \in K$, such that $x_{ijk} > \epsilon$ is at most $\frac{1}{\epsilon} - 1$.*

Again, scanning $supp(x)$ only once allows us to compute the sets defined above. That is, whenever we encounter a triple $v \in V$ in $supp(x)$ with $|v \cap i_1| = 1$, then if $x_v > \frac{1}{4}$

we add v to $A^{>\frac{1}{4}}(i_1)$ and if x_v is larger than x_u with $u \in A(i_1)$ we replace u with v in $A(i_1)$. Algorithm 4 contains all related steps.

Algorithm 4 Calculating $A(i)$ and $A^{>\frac{1}{4}}(i)$

```
{Input: the list supp(x)}
for all c ∈ supp(x) with c = (i_c, j_c, k_c) do
  if x_c > 1/4 then
    if set A^{>1/4}(i_c) does not exist then
      create set A^{>1/4}(i_c);
      A^{>1/4}(i_c) := ∅;
    end if
    A^{>1/4}(i_c) := A^{>1/4}(i_c) ∪ {c};
  end if
  if set A(i_c) does not exist then
    create set A(i_c);
    A(i_c) := ∅;
  end if
  if |A(i_c)| < 3 then
    A(i_c) := A(i_c) ∪ {c}
  else
    if x_c > x_d where x_d = min_{t∈A(i_c)} x(t) then
      A(i_c) := A(i_c)\{d} ∪ {c}
    end if
  end if
end for
```

Lemma 4 *For an arbitrary $x \in P^n$, the sets $A(i)$ and $A^{>\frac{1}{4}}(i)$ for $i \in I$, can be calculated in $O(T)$ steps.*

Proof Algorithm 4 shows how to compute $A(i)$ and $A^{>\frac{1}{4}}(i)$, for each $i \in I$ in $O(T)$ steps. A straightforward extension of Algorithm 4 implies the result. □

5 Improved separation algorithms

This section describes $O(T)$ separation algorithms for clique inequalities of types I and II.

Regarding clique inequalities of type I, recall that, with $c = (i_c, j_c, k_c)$, $Q(c) = \{(i_c, j_c, k_c)\} \cup \{(i_c, j_c, k), k \in K\backslash\{k_c\}\} \cup \{(i_c, j, k_c), j \in J\backslash\{j_c\}\} \cup \{(i, j_c, k_c), i \in I\backslash\{i_c\}\}$ (see Fig. 1). Thus:

$$x(Q(c)) = \mathrm{SUM}K(i_c, j_c) + \mathrm{SUM}J(i_c, k_c) + \mathrm{SUM}I(j_c, k_c) - 2 \cdot x_{i_c j_c k_c} \leq 1. \quad (6)$$

In the rest of the section we assume that a violated inequality corresponds to a triple (i, j, k) satisfying $\mathrm{SUM}K(i, j) \geq \mathrm{SUM}J(i, k) \geq \mathrm{SUM}I(i, k)$. Notice that this assumption is without loss of generality since we can interchange the roles of i, j, k. Algorithms similar to this case can be constructed for the other cases by interchanging

the role of i, j, k and pre-computing the sets similar to $J(i)$ via a slight modification of Algorithm 3.

Let us first give an informal description of our separation algorithm for this class. As mentioned in Sect. 4, the set C in this case contains one element from each of the sets I, J, K, thus our separation algorithm checks whether such a triple (i, j, k) corresponds to a violated inequality. The separation algorithm, Algorithm 5, contains two 'for' loops. The first loop examines all triples (i_c, j_c, k_c) in the support, and uses, in the subsequent if-statement, the values of i_c and k_c. (Algorithm 5 presents only the case where these two elements are in I, K since the other two cases are implemented similarly). The second 'for' loop checks the inequality explicitly for each possibility of the third element.

Lemma 5 *Given its input, Algorithm 5 determines in $O(T)$ steps whether an arbitrary $x \in P^n$ violates a clique inequality of type I.*

Proof Let us first show that a clique inequality of type I is violated only if there is (i_c, j_c, k_c) such that

$$\text{SUM}K(i_c, j_c) > \frac{1}{3} \tag{7}$$

and

$$\text{SUM}J(i_c, k_c) > 0. \tag{8}$$

Notice that (7) follows directly from (6) and our assumption that $\text{SUM}K(i_c, j_c) \geq \text{SUM}J(i_c, k_c) \geq \text{SUM}I(j_c, k_c)$. Concerning (8), observe that $\text{SUM}J(i_c, k_c) = 0$ yields $\text{SUM}I(j_c, k_c) = 0$, while the remaining terms of (6) cannot sum to a total of more than 1 and hence (6) cannot be violated.

Next, observe that each $(i_c, k_c) \in I \times K$ satisfying (8) is contained in some triple in $supp(x)$. Algorithm 5 does consider each triple in $supp(x)$. Once i_c, k_c are fixed, then by definition all j_c for which (7) is satisfied are stored in the, pre-calculated by Algorithm 3, $J(i_c)$. Algorithm 5 proceeds by checking the inequality for each (i_c, j, k_c) with $j \in J(i_c)$, given i_c, k_c. Hence Algorithm 5 is correct.

Regarding complexity, the first 'for' loop runs $O(T)$ times. The second 'for' loop runs for $O(1)$ times as the cardinality of $J(i)$ is at most 2. Therefore, the overall complexity of Algorithm 5 is $O(T)$. □

Theorem 1 *For any $x \in P^n$, clique inequalities of type I can be separated in $O(T)$ steps.*

Proof Lemma's 2 and 5 imply that applying first Algorithm 3 and applying Algorithm 5, determines whether there is a violated clique inequality of type I. Total complexity follows easily from the complexity of these algorithms. □

Algorithm 5 A separation algorithm for clique inequalities of type I

{Input: $supp(x)$; SUM$I(j,k)$, SUM$J(i,k)$, SUM$K(i,j)$, and $J(i)$ for all i, j, k}
for all $c \in supp(x)$ with $c = (i_c, j_c, k_c)$ **do**
  for all $j \in J(i_c)$ **do**
    if SUM$J(i_c, k_c)$ + SUM$I(j, k_c)$ + SUM$K(i_c, j) - 2 \cdot x_{i_c,j,k_c} > 1$ **then**
      return $x(Q(c')) \leq 1$ as violated, $c' = (i_c, j, k_c)$;
    end if
  end for
end for

Let us now focus on clique inequalities of type II. For $c = (i_c, j_c, k_c)$ and $h = (i_h, j_h, k_h)$, $Q(h) \cap \text{co}Q(c) = \{(i_c, j_h, k_h), (i_h, j_c, k_h), (i_h, j_h, k_c)\}$. Hence $Q(c,h) = \{c\} \cup (Q(h) \cap \text{co}Q(c))$ is a clique with exactly four nodes, any pair of which share exactly one index; notice also that any node in $Q(c,h)$ can play the role of c. Last, observe that any three nodes in $Q(c,h)$ are a subset of the node set of a clique of type I; for example, $\{(i_c, j_h, k_h), (i_h, j_c, k_h), (i_h, j_h, k_c)\} \subseteq Q(h)$, $\{(i_c, j_c, k_c), (i_c, j_h, k_h), (i_h, j_c, k_h)\} \subseteq Q((i_c, j_c, k_h))$, and so on.

Assuming that no clique inequality of type I is violated, i.e., Algorithm 5 has returned no violated inequality, we provide an $O(T)$ separation algorithm for clique inequalities of type II.

The main idea of our algorithm is again to construct the set C associated with a violated inequality, which in this case contains two elements from each of I, J, K. Our separation algorithm for this class is called Algorithm 6 and has three 'for' loops. In the first 'for' loop, three of the six elements are added to the set C associated with the inequality to be checked. The second and third loop use the precomputed sets mentioned in Sect. 4 to add the remaining three elements to C. Then, the inequality is explicitly checked for the constructed C.

Lemma 6 *Given its input, Algorithm 6 determines in $O(T)$ steps whether an arbitrary $x \in P^n$ violates a clique inequality of type II.* □

Proof Let us first examine the correctness of the algorithm, by considering some $x \in P^n$ such that

$$x(Q(c,d)) = x_{i_c j_d k_d} + x_{i_d j_c k_d} + x_{i_d j_d k_c} + x_{i_c j_c k_c} > 1. \tag{9}$$

Recall that (9) is symmetric in the sense that any three variables in it appear together in a clique inequality of type I and any two variables in it have exactly one index in common. Therefore, let us assume without loss of generality that

$$x_{i_c j_c k_c} \geq \max\{x_{i_c j_d k_d}, x_{i_d j_c k_d}, x_{i_d j_d k_c}\}. \tag{10}$$

Then, the triple indexing the variable with the smallest value in (9) can be (i_c, j_d, k_d) or (i_d, j_c, k_d) or (i_d, j_d, k_c); it suffices to prove the correctness of the algorithm for the case where $x_{i_c j_d k_d}$ is the smallest of the four variables in (9).

Due to the assumption that no clique inequality of type I is violated, it follows that all four variables from (9) must be positive, and hence appear in $supp(x)$.

Therefore, $(i_c, j_d, k_d) \in supp(x)$. Algorithm 6 proceeds by considering each possibility of (i_c, j_d, k_d) in $supp(x)$. Further, observe that at least one of the variables of (9) must have a value greater than $\frac{1}{4}$. By (10) we have $x_{i_c j_c k_c} > \frac{1}{4}$, for some $(i_c, j_c, k_c) \in supp(x)$. Note that, for a fixed i_c, such an (i_c, j_c, k_c) should be in $A^{>\frac{1}{4}}(i_c)$. Moreover, for any fixed $c = (i_c, j_c, k_c)$ such that $x_c > \frac{1}{4}$, there are at most 4 possible triples h that do not contain i_c, and $x_h > \frac{1-x_c}{3}$. By definition they all should be in $B(j_c)$ and $C(k_c)$. Consequently, Algorithm 6 is correct.

Concerning the complexity of Algorithm 6, note that there are four loops, namely one 'outer', one 'inner' and two 'innest' (with a slight language abuse). The 'outer' loop is performed $O(T)$ times. Since the cardinality of $A^{>\frac{1}{4}}(i)$, $B^{>\frac{1}{4}}(j)$, $C^{>\frac{1}{4}}(k)$ is at most 3, the inner loop is performed at most 3 times. Therefore 'inner' loop runs for $O(T)$ times. For each $c \in supp(x)$ such that $x_c > \frac{1}{4}$, the two 'innest' loops are performed a constant number of times as the cardinality of $A(i_c)$,$B(j_c)$,$C(k_c)$ is constant. Therefore, the two 'innest' loops are run $O(T)$ times. In total, the complexity of Algorithm 6 is $O(T)$. □

Algorithm 6 Separation algorithm for clique inequalities of type II

{Input: the list $supp(x)$, sets $A(i)$, $B(j)$, $C(k)$ for $i \in I$, $j \in J$, and $k \in K$}
for all $c' \in supp(x)$, let $c' = (i_c, j_d, k_d)$ **do**
  for all $(i_c, j_c, k_c) \in A^{>\frac{1}{4}}(i_c)$ such that $j_c \neq j_d$,$k_c \neq k_d$ **do**
    for all $h \in B(j_c) \cup C(k_d)$ such that $h = (i_d, j_c, k_d)$ and $x_h > \frac{1-x_c}{3}$ **do**
      if $x(Q(c,d)) > 1$ **then**
        return $x(Q(c,d)) \leq 1$ as violated;
      end if
    end for
    for all $h \in B(j_d) \cup C(k_c)$ such that $h = (i_d, j_d, k_c)$ and $x_h > \frac{1-x_c}{3}$ **do**
      if $x(Q(c,d)) > 1$ **then**
        return $x(Q(c,d)) \leq 1$ as violated;
      end if
    end for
  end for
end for

Theorem 2 *For any $x \in P^n$, clique inequalities of type II can be separated in $O(T)$ steps.*

Proof Lemma's 4 and 6 imply that first applying Algorithm 4, and then applying Algorithm 6 determines whether there is a violated inequality of type II. The complexity follows easily. □

6 The k-index assignment polytope

The ideas presented in the previous section are also applicable to the axial assignment problem comprising more than 3 sets, see Appa et al. [2]. The k-index axial assignment problem can be defined using k disjoint n-sets $M_1, \ldots, M_k$ and a cost-function w :

$M_1 \times \cdots \times M_k \longrightarrow \mathbb{R}$. Let $M \equiv M_1 \times M_2 \times \cdots \times M_k$ be the cartesian product of the sets M_i. The problem is to find a minimum-cost collection of n disjoint k-tuples such that every element of each set (i.e., every $m(i) \in M_i, i \in \{1, \ldots, k\}$) appears in exactly one k-tuple.

First let us introduce some notation which is useful to explain the formulation and algorithms.

$$\min \quad \sum_{m \in M} w_m \cdot x_m \tag{11}$$

$$\text{s.t.} \sum_{m:m(j)=i} x_m = 1, \quad \forall j = 1, \ldots, k, i \in M_j, \tag{12}$$

$$x_m \in \{0, 1\}^{n^k}, \quad \forall m \in M. \tag{13}$$

Let $A^{(k,n)}$ denote the (0, 1) matrix of the constraints (12). The matrix $A^{(k,n)}$ has n^k columns and $k \cdot n$ rows, while each constraint includes n^{k-1} variables. The associated (axial assignment) polytope is $P_I^{(k,n)} = conv\{x \in \{0, 1\}^{n^k} : A^{(k,n)}x = e\}$, while its LP-relaxation is $P^{(k,n)} = \{x \in R^{n^k} : A^{(k,n)}x = e, x \geq 0\}$. As in Remark 1, any solution of $P^{(k,n)}$ has at most $k \cdot n$ non-zero variables.

The *column intersection* graph of $A^{(k,n)}$, namely $G(V^k, E^k)$, has a node for each column of $A^{(k,n)}$ and an edge for every pair of columns that have a $+1$ entry in the same row. It is easy to see that the node set V^k is identical to M and that if $c, d \in V^k$ then $(c, d) \in E^k$ if and only if $|c \cap d| \geq 1$ (i.e. two nodes are connected if the corresponding tuples have at least one index in common).

Let us now generalize the concepts of Sect. 5 by defining

- for each $c \in V^k$: $Q^k(c) = \{d \in V^k : |c \cap d| = k - 1\}$,
- for each $c \in V^k$: $\text{co}Q^k(c) = \{d \in V^k : |c \cap d| = 1\}$, and
- for each $c, d \in V^k$ with $|c \cap d| = 0$: $Q^k(c, d) = \{c\} \cup \{Q^k(d) \cap \text{co}Q^k(c)\}$.

We refer to the inequalities $x(Q^k(c)) \leq 1$ as *generalized* inequalities of type 1, and to the inequalities $x(Q^k(c, d)) \leq 1$ as *generalized* inequalities of type 2. We point out here that these particular inequalities, while valid, are not facet-defining for $k \geq 4$. However, separating these inequalities efficiently is still relevant, since they can either be added directly to the linear program, or strengthened by lifting some coefficients to obtain facets (see Magos and Mourtos [19]).

By generalizing the ideas and algorithms of the previous sections, we obtain the following results, whose proofs can be found in Dokka [13].

Theorem 3 *The generalized inequalities of type I can be separated in $O(k^2T)$ steps.*

Theorem 4 *The generalized inequalities of type II can be separated in $O(k^3T)$ steps.*

7 Computational experiments

In this section we report on the computational performance of the separation algorithms for clique inequalities of types I and II. We compare the running times of our algorithms with the performance of the traditional separation algorithms described in Balas and Qi [4]. All algorithms have been coded in C++ using Visual Studio C++ 2005 and

ILOG concert technology. All the experiments have been conducted on a Dell Latitude E6400 PC with Intel core 2 Duo processor with 2.8 GHz and 1.59 GB RAM under Windows XP. CPLEX 12.4 has been used for solving the LP-relaxation.

7.1 Instances

For our experiments we consider seven classes of instances used in the literature.

The first class includes the instances proposed by Balas and Saltzman [6]. The integer cost coefficients $c_{i,j,k}$ are generated uniformly in the interval [0, 100]. These instances are denoted as UNIFORM instances.

The instances of the second class are generated using again the method in [6], the only difference being that the cost coefficients are generated uniformly in the interval [0, 9999]. We denote these instances as UNIFORM10K.

The third class of instances consists of the 18 instances from Crama and Spieksma [11]. These include are 9 instances size 33 and 9 instances of size 66. In these instances, the cost function is decomposable and the details on the generation procedure can be found in [11].

The fourth and fifth classes of instances are taken from Höfler and Fügenschuh [15] and denoted as QUAD and CLUSTER respectively. The QUAD instances are randomly generated, with their cost coefficients having a value $10{,}000 \cdot z^2$ where z is uniformly distributed in the interval [0,1]. The cost coefficients of the CLUSTER instances are chosen out of three clusters [0, 49], [450, 499], [950, 999], where each coefficient lies in a specific cluster with probability equal to $1/3$ and is then chosen uniformly from the range of that cluster.

The sixth class of instances, denoted as BRW, is generated using the method described in Burkard et al. [7]. Each cost coefficient is decomposable, assuming the form $c_{i,j,k} = a_i \cdot b_j \cdot c_k$, where each of a_i, b_j, c_k is uniformly distributed in the interval [1, 10].

The last class of instances, denoted as GP, is generated using the algorithm proposed in Grundel and Pardalos [14] with random costs coefficients selected uniformly from the interval [1, 300]. A detailed explanation on the generation of the cost coefficients can be found in [14].

The largest instances used in the literature are of size 26. To better understand the improvement achieved by our fast separation algorithms, we create larger instances for the each of the classes described above. That is, for each among the first six classes, we generate 5 instances for each of the sizes 25, 54, 66, 80, 100, 120. For GP instances we create 5 instances for each of the sizes 25, 54, 66, 80 and 100. Table 1 gives a summary of the problem classes used in our experiments. All instances can be found at http://www.lancaster.ac.uk/staff/dokka/download.htm.

7.2 Implementation details

There are different ways of implementing a cutting plane algorithm. For instance, one can add a single violated inequality or all violated ones in each iteration. Although having experimented with both options, we only report the results of the implementation

Table 1 Problem classes

Instance class	Parameters	Method
UNIFORM	[0, 99]	[6]
UNIFORM10K	[0, 9999]	[6]
QUAD	$\lfloor 10,000 \cdot z^2 \rfloor$, $z \in [0, 1[$	[15]
CLUSTER	[0, 49], [450, 499], [950, 999]	[15]
GP	[0, 300]	[14]
BRW	$[1, 10] \times [1, 10] \times [1, 10]$	[7]
FY	[0, 100]	[11]

Algorithm 7 Algorithm Outline

```
0. Let lp = LP relaxation of (2)–(5)
1. Solve lp and find sup(x*)
2. Input sup(x*) to separation algorithm for clique inequalities of type I
if violated clique inequalities of type I have been found then
  update lp by adding all violated clique inequalities of type I; goto step 1
else
  Input sup(x*) to the separation algorithm for clique inequalities of type II
  if violated clique inequalities of type II have been found then
    update lp by adding all violated clique inequalities of type II; goto step 1
  end if
  STOP: if no violated clique inequalities of type I or type II
end if
```

where all violated inequalities are added per iteration. This is because, when adding a single inequality, the inequality found by our separation algorithms may differ from the violated inequality found by the traditional algorithms, thus yielding a different number of iterations and hence a less transparent comparison.

Further, we opt for the following strategy: first, we separate (and add to the LP-relaxation) clique inequalities of type I; next, we separate clique inequalities of type II only if no violated type I inequalities are detected (see Algorithm 7). Of course, this procedure favors the detection of violated type I inequalities.

Clearly, when implementing Algorithm 7 (see line 1), we need to find the support, i.e., we need to detect which entries are nonzero. To do that we use a tolerance level of 1.0e-05. We ran the experiments with other tolerance levels, and found no significant differences.

7.3 Results and discussion

The outcomes of our experiments are described in Table 2. We denote the cutting plane scheme that uses our separation algorithms by *DMS*, and the scheme using the traditional algorithms by *BQ*. The first and second columns specify the *class* and the *size* of the instances. The third and fourth columns are times taken by DMS and BQ, respectively. The time reported for each size and class is the average time over the 5 (4 for each size of UNIFORM class) instances of that size (and class). The last column

Table 2 Comparison of average computation times

Type	Size	CPU(s)		CPU(s)		
		DMS	BQ	DMS	BQ	*Cuts*
		Avg	Avg	Stdev	Stdev	(#)
UNIFORM	25	0.006	0.021	0.008	0.008	5.25
	54	0.037	0.35	0.023	0.052	8.5
	66	0.054	0.764	0.019	0.054	7
	80	0.152	2.195	0.023	0.232	8.67
	100	0.262	4.008	0.060	0.872	10.4
	120	0.408	7.799	0.126	0.804	10.6
QUAD	25	0.004	0.019	0.007	0.018	6.4
	54	0.032	0.331	0.009	0.017	7
	66	0.065	0.785	0.021	0.031	7.75
	80	0.124	1.603	0.052	0.434	14.2
	100	0.307	4.74	0.110	0.780	12.8
	120	0.382	7.836	0.085	0.608	13.4
BRW	25	0.004	0.022	0.013	0.008	7.6
	54	0.043	0.371	0.017	0.028	16.6
	66	0.083	0.814	0.041	0.054	17.2
	80	0.166	1.729	0.047	0.124	19.4
	100	0.265	4.161	0.135	1.454	17.75
	120	0.487	9.379	0.118	3.059	21.2
UNIFORM10K	25	0.003	0.018	0.011	0.007	6.6
	54	0.027	0.307	0.015	0.017	4.6
	66	0.051	0.693	0.022	0.027	5.8
	80	0.089	1.407	0.026	0.433	4.6
	100	0.159	3.368	0.041	0.480	5.6
	120	0.293	7.03	0.077	0.696	7
CLUSTER	25	0.004	0.024	0.007	0.018	7.4
	54	0.037	0.313	0.013	0.039	9.2
	66	0.074	0.788	0.024	0.046	10.2
	80	0.148	1.673	0.019	0.152	15.6
	100	0.313	4.323	0.077	0.448	13
	120	0.446	8.505	0.104	0.655	13.8
GP	25	0.003	0.004	0.007	0.007	6.4
	50	0.014	0.038	0.009	0.023	7.5
	66	0.035	0.121	0.015	0.046	9.4
	80	0.067	0.251	0.028	0.146	13
	100	0.1326	0.6488	0.007	0.052	16.2
FY	33	0.006	0.019	0.007	0.008	0.89
	66	0.031	0.224	0.009	0.015	2.45

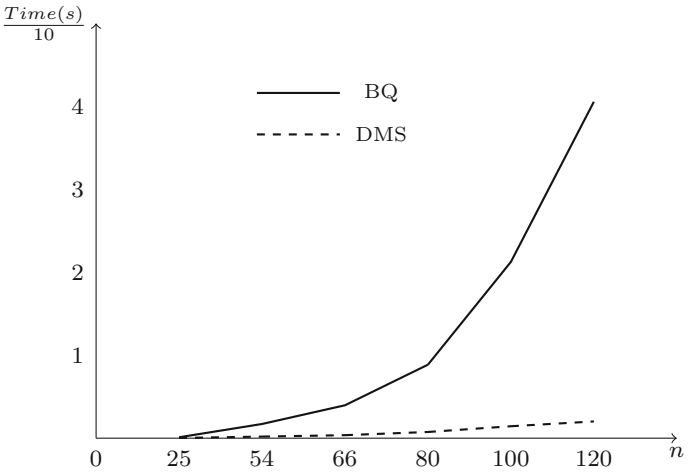

Fig. 3 DMS vs BQ with increasing n

gives the average number of cuts added for DMS. Clearly, the LP-bound found by the two separation algorithms is identical; however, the total number of cuts added by both separation algorithms can be different. The reason for this is that, after the first iteration, CPLEX may find different LP solutions (albeit with the same LP value, of course) which is caused by the fact that even if the set of violated inequalities is the same for both separation algorithms, the order in which these inequalities are added to the formulation can be different, and this may influence the LP solution found by the CPLEX. Indeed, when we sorted the inequalities found, and added them in the same order to the formulation, the total number of cuts found by the two separation algorithms is identical. Sorting the inequalities, however, slightly increases the running time, and we chose not to do this. Moreover, we found that the total number of cuts for both implementations differs only marginally; so, we chose to report the average number of cuts for DMS.

Further, since our main focus is on running times of the separation algorithms, we do not report the increase of the LP value (recall that we solve the 3-index assignment as a minimization problem).

Surprisingly, no violated clique inequalities of type II are found in almost all instances. Only in 3 instances, namely 2 from the BRW class and 1 from there UNIFORM class, there have been violated clique inequalities of type II. For this reason, we only report the average number of cuts added. Although this behavior appears uncommon at first sight, similar behavior has been observed for other problems, e.g., for the node packing problem regarding clique and odd-hole inequalities [20]. This may be caused (partly) by the set-up of the separation routine, i.e., there would violated clique of type II if separated before cliques of type I. However, recall that Algorithm 6 relies on the fact no clique inequality of type I is violated in order to exploit the property that any three variables in a clique inequality of type II appear in a clique inequality of type I (and thus improve significantly its running time compared to the traditional algorithm).

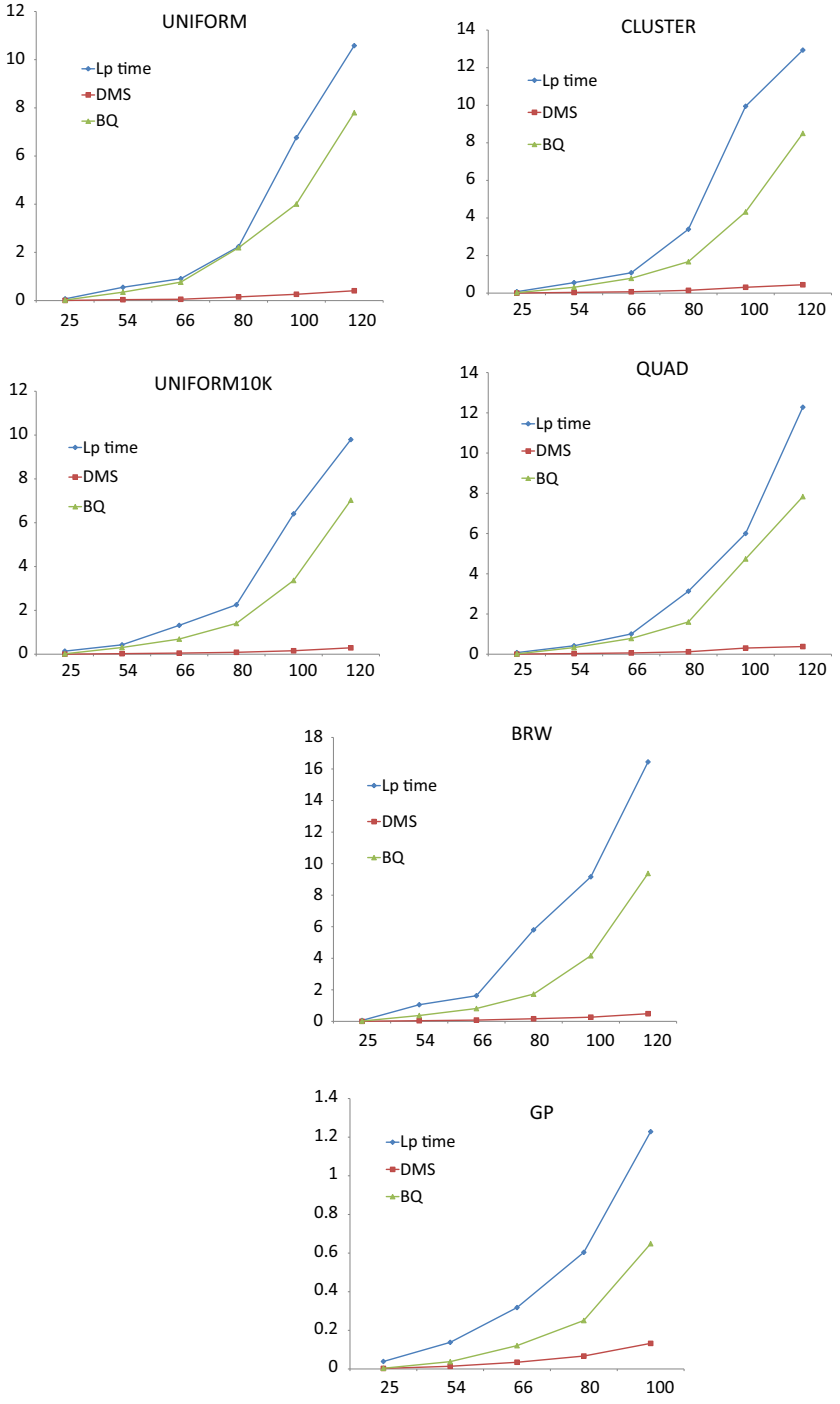

Fig. 4 Comparison of lp times with separation times

The results of Table 2, show that DMS outperforms BQ in all instances, the improvement being more impressive as the size of the instance grows. Also, compared to BQ the variance in running time is also low for DMS. Thus, also in practice, exploiting sparsity yields separation algorithms that are much faster, often by an order of magnitude: on average, DMS is more than 15 times faster than BQ. This behavior is better illustrated in Fig. 3 (whose y-axis reports the average over *all* classes per size). This shows that T indeed grows very slowly compared to the n^3 and in general scales-up pretty well.

One might be interested in comparing the separation time with the time needed to re-optimize the LP after cut addition. This is done in Fig. 4, where 'lp time' is the time spent only on re-solving the LP (without the time for solving it initially). It can be seen that the relative amount of time spent in separation is much smaller if DMS is used instead of BQ. Let us also note that the 'lp time' may in general vary significantly depending upon the instance.

8 Conclusion

In this paper we have revisited the idea of exploiting sparsity in separation algorithms. We have also suggested to express the time complexity of such algorithms in terms of sparsity (i.e., the number of non-zero entries in the input). By allowing the input vector of a separation algorithm to be described by its support, we have obtained more efficient separation algorithms for the clique inequalities of the 3-index assignment problem.

Indeed, the improvement achieved for the 3-index axial assignment problem is significant if the vectors to be separated are vertices of the LP-relaxation because the corresponding formulation contains more variables than constraints. Therefore, analogous improvements could be plausible for other problems whose formulations share this property. However, notice that for problems where column generation is used to solve a linear programming formulation, this idea seems not applicable, since, in such a setting, variables are generated instead of violated inequalities. Thus, formulations with more variables than constraints that are not being solved by a column generation approach are susceptible to the design of efficient separation algorithms that exploit sparsity.

Acknowledgements We are thankful to Armin Fügenschuh for providing the routines to generate instances used in [15], and to Yves Crama for stimulating discussions on this subject.

References

1. Alvarez-Valdes, R., Parreo, F., Tamarit, J.: A branch-and-cut algorithm for the pallet loading problem. Comput. Oper. Res. **32**, 3007–3029 (2005)
2. Appa, G., Magos, D., Mourtos, I.: On multi-index assignment polytopes. Linear Algebra Appl. **416**, 224–241 (2006)
3. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: The Traveling Salesman Problem: A Computational Study. Princeton University Press, Princeton (2006). (ISBN 978-0-691-12993-8)
4. Balas, E., Qi, L.: Linear-time separation algorithms for the three-index assignment polytope. Discrete Appl. Math. **43**, 1–12 (1993)

5. Balas, E., Saltzman, M.: Facets of the three-index assignment polytope. Discrete Appl. Math. **23**, 201–229 (1989)
6. Balas, E., Saltzman, M.: An algorithm for the three index assignment problem. Oper. Res. **39**, 150–161 (1991)
7. Burkard, R., Rudolf, R., Woeginger, G.: Three-dimensional axial assignment problems with decomposable cost coefficients. Discrete Appl. Math. **65**, 123–139 (1996)
8. Caprara, A., Salazar, J.: Separating lifted odd-hole inequalities to solve the index selection problem. Discrete Appl. Math. **92**, 111–134 (1999)
9. Cheng, E., Cunningham, W.: Separation problems for the stable set polytope. In: Proceedings of The 4th Integer Programming and Combinatorial Optimization Conference Proceedings (1995)
10. Cheng, E., Cunningham, W.: Wheel inequalities for stable set polytopes. Math. Program. **77**, 389–421 (1997)
11. Crama, Y., Spieksma, F.: Approximation algorithms for three-dimensional assignment problems with triangle inequalities. Eur. J. Oper. Res. **60**, 273–279 (1992)
12. Van den Akker, J., van Hoesel, C., Savelsbergh, M.: A polyhedral approach to single machine scheduling problems. Math. Program. **85**, 541–572 (1999)
13. Dokka, T.: Algorithms for multi-index assignment problems. PhD thesis, KU Leuven (2013)
14. Grundel, D., Pardalos, P.: Test problem generator for the multidimensional assignment problem. Comput. Optim. Appl. **30**(2), 133–146 (2005)
15. Höfler, B., Fügenschuh, A.: Parametrized grasp heuristics for three-index assignment. EvoCOP Lect. Notes Comput. Sci. **3906**, 61–72 (2006)
16. Kaparis, K., Letchford, A.: Separation algorithms for 0–1 knapsack polytopes. Math. Program. **124**, 69–91 (2010)
17. Kececioglu, J., Lenhof, Hans-Peter, Mehlhorn, K., Mutzel, P., Reinert, K., Vingron, M.: A polyhedral approach to sequence alignment problems. Discrete Appl. Math. **104**(1–3), 143–186 (2000)
18. Landete, M., Escudero, L., Marìn, A.: A branch-and-cut algorithm for the winner determination problem. Decis. Support Syst. **46**, 649–659 (2009)
19. Magos, D., Mourtos, I.: Clique facets of the axial and planar assignment polytopes. Discrete Optim. **6**, 394–413 (2009)
20. Nemhauser, G.L., Sigismondi, G.: A strong cutting plane/branch-and-bound algorithm for node packing. J. Oper. Res. Soc. **5**, 443–457 (1992)
21. Qi, L., Sun, D.: Polyhedral methods for solving three index assignment problems. In: Pardalos, P.M., Pitsoulis, L. S., (eds.) Nonlinear Assignment Problems: Algorithms and Applications, pp. 91–107. Springer, US (2000)
22. Rebennack, S., Oswald, M., Oliver Theis, D., Seitz, H., Reinelt, G., Pardalos, P.: A branch and cut solver for the maximum stable set problem. J. Comb. Optim. **21**(4), 434–457 (2011)
23. Schrijver, A.: Combinatorial Optimization, Polyhedra and Efficiency. Springer, Berlin (2003)
24. Spieksma, F.: Multi-index assignment problems: complexity, approximation, applications. In: Pardalos, P.M., Pitsoulis, L. (eds.) Nonlinear Assignment Problems: Algorithms and Applications, pp. 1–12. Kluwer Academic Publisher, Nowell (2000)