http://dx.doi.org/10.1287/ics.2015.0009

14th INFORMS Computing Society Conference
Richmond, Virginia, January 11–13, 2015
pp. 119–133

Row-Partition Branching for Set Partitioning Problems

Namsuk Cho, Jeff Linderoth
Department of Industrial and Systems Engineering, University of Wisconsin-Madison, Madison, WI 53706, ncho6@wisc.edu
Department of Industrial and Systems Engineering, University of Wisconsin-Madison, Madison, WI 53706, linderoth@wisc.edu

Abstract We introduce the *Row-Partition Branching Method* for set partitioning and set packing problems. Branching is accomplished by selecting a subset of the rows of the problem that must be covered and partitioning the variables according to the number of rows of the subset that each variable covers. The method can be viewed as a generalization of the well-known Ryan-Foster branching scheme. We discuss implementation issues associated with our method and show through computational experiments that the method is competitive with other branching methods for set partitioning and packing problems.

Keywords Set Partitioning Problem, Branching Methods.

1. Introduction

In this paper, we introduce a novel branching method for solving integer programming formulations of the set partitioning problem. The set partitioning problem (SPP) may be stated mathematically as

$$\min_x \{c^\top x \mid Ax = \mathbf{1}, x \in \{0,1\}^n\}, \tag{SPP}$$

where the matrix $A \in \{0,1\}^{m\times n}$ and $\mathbf{1}$ is an m-dimensional vector of ones. We denote the row index set by $I = \{1, 2, \ldots, m\}$ and the column index set by $J = \{1, 2, \ldots, n\}$. The set partitioning problem is used to model many important applications, including vehicle routing [4], crew scheduling for airlines [23], and many others. The reader may see the bibliography of [3] for a list of many other successful practical applications of the SPP.

Given its practical relevance, it is no surprise that significant research has been done on solution approaches for (SPP). Exact solution approaches include those based on integer programming [23, 16, 6] and constraint programming [24, 27]. Heuristic methods for solving the problem may use mathematical programming as an integral component, [2, 20, 7], or they may be based on meta-heuristics [19, 8].

Our work will have applicability for solution methods that are based on enumeration. We investigate a new branching method that exploits the structure of the constraints in (SPP). Each row in (SPP) is of the form

$$\sum_{j\in J_i} x_{ij} = 1, \tag{1}$$

where for each row $i \in I$, $J_i = \{j \in J \mid a_{ij} = 1\}$ is the set of columns that intersects that row. The constraints ensure that the ground set of rows is partitioned—exactly one column in J must intersect with each row of I. The constraints (1) are often called Specially Ordered

Sets of Type 1 (SOS-1), implying that in the set of variables J_i at most one of the variables is allowed to be positive. We will also refer to the constraints (1) as Generalized Upper Bound (GUB) constraints.

In a linear programming (LP)-based branch and bound approach to solving (SPP), if the LP relaxation to (SPP) is fractional, the problem must be recursively divided into smaller subproblems by branching. It is part of the computational integer programming folklore that traditional variable branching—where a single fractional variable x_j is selected, and two child subproblems enforcing either $x_j = 0$ or $x_j = 1$ are created—is ineffective for solving (SPP). The fundamental problem is that the search tree can become "imbalanced". Specifically, branching on a single variable does a very poor job of balancing the number of feasible solutions allocated to each side of the branching dichotomy. The $x_j = 0$ branch still contains a very large percentage of feasible solutions, and the $x_j = 1$ branch will have a very small percentage of the feasible solutions.

To overcome this imbalance, a traditional method of branching in integer programs that have GUB constraints (1) is to choose a subset $Q \subset J_i$ of the variables appearing in row $i \in I$ and to enforce that either

$$\sum_{j \in Q} x_j = 1 \qquad \text{or} \qquad \sum_{j \in J_i \setminus Q} x_j = 1.$$

Note that since all feasible solutions satisfy (1), this branching dichotomy is equivalent to

$$\sum_{j \in J_i \setminus Q} x_j = 0 \qquad \text{or} \qquad \sum_{j \in Q} x_j = 0, \tag{2}$$

and (2) has significant computational advantage, since it may be implemented by fixing bounds $x_j = 0 \; \forall j \in J_i \setminus Q$ on the first branch and $x_j = 0 \; \forall j \in Q$ on the second branch. Computational experiments demonstrating the superiority of GUB branching over traditional variable branching on some instances were given by [10, 17, 12].

Another branching methodology for SPP was proposed by Ryan and Foster [25]. This method is based on the logic that in any feasible solution to (SPP), each pair of rows $(p, q) \in I \times I$ is either covered by the same column or by two different columns. The branching rule is especially relevant in column-generation-based approaches to solving (SPP), since the branching restrictions may be enforced without modifying the structure of the pricing subproblem [5]. There have been some computational studies that demonstrate that the rule may also be useful for (SPP) instances not solved by column generation [26, 13, 1].

In this paper, we propose a generalization of the Ryan-Foster branching method that we call the *row-partition branching method*. We will discuss implementation issues associated with the method and perform an empirical comparison of the proposed method with other branching methods designed for (SPP). The paper is organized into five sections. In Section 2 we describe the proposed branching method, while Section 3 deals with implementation issues associated with the method. We report on a series of computational experiments in Section 4, and we offer some conclusion about the new method in Section 5.

2. Row-Partition Branching

In Row-Partition Branching, a subset of the rows is selected, and variables are partitioned according to the number of rows that each variable covers. In this section, we explain the method in more detail, including how to implement the branching constraints by only fixing variable bounds to zero. We also establish the correctness and completeness of the method by appealing to the Ryan-Foster branching method.

2.1. Description

For any subset of rows $S \subseteq I$ with $|S| = s$, each of the rows in S is covered in any feasible solution to (SPP). The rows in S may all be covered by exactly one column, or it may take exactly two columns to cover the rows, or in general it may take up to s different columns to cover the rows of S. This defines for us a possible branching partition. The number of child nodes in this branching dichotomy obviously depends on the number of ways $s \in \mathbb{Z}_+$ can be expressed as the sum of positive integers. This value is known as the *partition function* and is typically denoted by $p(s)$. For notational purposes, we assume that we may express the integer partition of $s \in \mathbb{Z}_+$ using values $\beta_{ij} \in \mathbb{Z}_+$, where

$$s = \sum_{j=1}^{n_i} \beta_{ij} \quad \forall i = 1, 2, \ldots, p(s),$$

and numbers β_{ij} may be repeated if the same value occurs multiple times in the element of the partition. For example, $p(5) = 7$, and the seven ways in which 5 may be expressed as the sum of positive integers is the following:

$$\begin{aligned}
&(i=1) \quad 5 = 5\\
&(i=2) \quad 5 = 4+1\\
&(i=3) \quad 5 = 3+2\\
&(i=4) \quad 5 = 3+1+1\\
&(i=5) \quad 5 = 2+2+1\\
&(i=6) \quad 5 = 2+1+1+1\\
&(i=7) \quad 5 = 1+1+1+1+1.
\end{aligned}$$

We make the additional notation that for each $i = 1, 2, \ldots, p(s)$, the value

$$\kappa_{ik} = |\{j : \beta_{ij} = k\}|$$

is the number of times the value $k \in \mathbb{Z}_+$ appears in the ith element of the partition of s. In addition, for any subset of rows $S \subset I$, we define the sets of variable indices

$$C_k(S) \stackrel{\text{def}}{=} \left\{ j \in J \;\middle|\; \sum_{i \in S} a_{ij} = k \right\}$$

as the variables that cover exactly k elements of S. With this notation, we may express the row-partition branching dichotomy as

$$\bigvee_{i=1}^{p(s)} \left(\bigwedge_{k:\kappa_{ik} \geq 1} \sum_{j \in C_k(S)} x_j = \kappa_{ik} \right), \tag{3}$$

which merely expresses the logic that the rows of S must be covered by columns that intersect S according to an element of the integer partition of s.

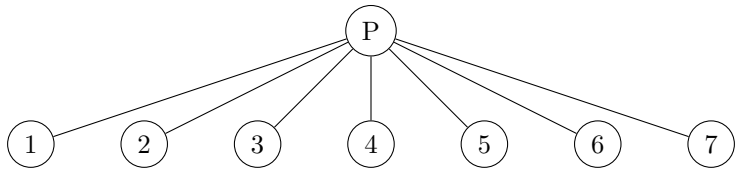
This is admittedly cumbersome notation, but the idea is very simple. As an example, suppose that we choose a subset of 5 rows, so $|S| = s = 5$. In this case, $p(5) = 7$ and there seven ways in which the five rows could possibly be covered. A feasible solution could contain

(1) One column in $C_5(S)$, or
(2) One column in $C_4(S)$ and one column in $C_1(S)$, or
(3) One column in $C_3(S)$ and one column in $C_2(S)$, or
(4) One column in $C_3(S)$ and two columns in $C_1(S)$, or
(5) Two columns in $C_2(S)$ and one columns in $C_1(S)$, or

(6) One columns in $C_2(S)$ and three columns in $C_1(S)$, or
(7) Five columns in $C_1(S)$.

These are all the possibilities for covering S, so we can create 7 child nodes from the current branching tree node as in Figure 1. An algebraic description of the constraints

FIGURE 1. Branching Nodes for Row-Partition Branching with $s = 5$



enforcing the branching logic for each of the child nodes shown in Figure 1 is

$$\begin{aligned}
&\textbf{Node 1:} && \sum_{j\in C_5(S)} x_j = 1;\\
&\textbf{Node 2:} && \sum_{j\in C_4(S)} x_j = 1, \quad \sum_{j\in C_1(S)} x_j = 1;\\
&\textbf{Node 3:} && \sum_{j\in C_3(S)} x_j = 1, \quad \sum_{j\in C_2(S)} x_j = 1;\\
&\textbf{Node 4:} && \sum_{j\in C_3(S)} x_j = 1, \quad \sum_{j\in C_1(S)} x_j = 2;\\
&\textbf{Node 5:} && \sum_{j\in C_2(S)} x_j = 2, \quad \sum_{j\in C_1(S)} x_j = 1;\\
&\textbf{Node 6:} && \sum_{j\in C_2(S)} x_j = 1, \quad \sum_{j\in C_1(S)} x_j = 3;\\
&\textbf{Node 7:} && \sum_{j\in C_1(S)} x_j = 5.
\end{aligned}$$

We give another example for the case that $S = \{r, p, q\}$. The three rows may be partitioned in $p(3) = 3$ ways, resulting in a 3-way branching and demonstrate on the matrix

$$A = \begin{array}{c} \\ row_r \\ row_p \\ row_q \end{array} \begin{array}{c} \begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array}. \tag{4}$$

The three rows may be partitioned in $p(3) = 3$ ways, resulting in a 3-way branching. If there exists a column $j \in J$ that intersects all three rows $\{r, p, k\}$, then fixing such a variable to 1 will cover all 3 rows. In (4), this is done by fixing x_1 to 1. Second, we can cover S with 2 variables by choosing one variable from the set that intersects S exactly twice and another variable that intersects S exactly once. In our example, this implies the branching constraints $(x_2 + x_3 = 1)$ and $(x_4 + x_5 + x_6 = 1)$. The final way to cover the rows in S is by using variables that intersect S exactly once. In our example, it would be implemented by the constraints $x_4 + x_5 + x_6 = 3$.

For the case that $|S| = s = 2$, the row-partition branching method is exactly the branching rule of Ryan and Foster [25]. The partition function $p(2) = 2$, and the elements of the

partition are $(2 = 2, 2 = 1 + 1)$, which gives rise to the branching rule suggested by Ryan and Foster. Specifically, one picks a pair of rows $S = (p, q) \in I \times I$ and enforces that

$$\sum_{j \in C_2(\{p,q\})} x_j = 1 \quad \text{or} \quad \sum_{j \in C_1(\{p,q\})} x_j = 2, \tag{5}$$

ensuring that the rows in S must be covered by exactly 1 column, or they are covered by exactly 2 columns.

2.2. Fixing Variable Bounds

As described, the row-partition branching enforces constraints on the cardinality of sets of variables that intersect the given branching set S a specified number of times. The straightforward manner to implement this branching methodology is by adding constraints to the child subproblems.

While there is some recent work in integer programming on choosing good branching constraints [22, 9], there are significant computational benefits to using branching rules that only change bounds on variables. If only bounds are changed, the size of the basis does not grow larger and existing matrix factors may be used to hot-start child nodes. Thus, we prefer to add only bounds when defining child node subproblems in our method.

It is possible to enforce the constraints in the row-partition branching by only fixing variable bounds. This requires, however, a recursive subdivision of some child nodes that essentially enumerates the specific rows that will be covered by elements of the partition.

The method is best described with a example for $S = \{r, p, q\}$, so that $|S| = s = 3$. In this case the branching scheme there are $p(3) = 3$ branches, since in every feasible solution either

(1) $\sum_{j \in C_3(S)} x_j = 1$, or
(2) $\sum_{j \in C_2(S)} x_j = 1$ and $\sum_{j \in C_1(S)} x_j = 1$, or
(3) $\sum_{j \in C_1(S)} x_j = 3$.

The first branch can be enforced by setting $x_j = 0 \; \forall j \in C_1(s) \cup C_2(s)$, and the third branch can be implemented by setting $x_j = 0 \; \forall j \in C_2(S) \cup C_3(s)$.

Only the second branch cannot immediately be implemented by fixing variables. It remains to enumerate the conditions on *which* of the 3 rows is covered by a column in $C_1(S)$. Specifically, for the set $S = \{p, q, r\}$, we can enumerate the three cases for the row that will be covered by the column in $C_1(S)$. We can then implement the condition $\sum_{j \in C_2(S)} x_j = 1$ and $\sum_{j \in C_1(S)} x_j = 1$ with the following three child nodes:

(2a) (Row p): $x_j = 0 \; \forall j \in C_3(S), \forall j \in C_2(\{p, q\}), \forall j \in C_2(\{p, r\}), \forall j \in C_1(\{q, r\})$
(2b) (Row q): $x_j = 0 \; \forall j \in C_3(S), \forall j \in C_2(\{p, q\}), \forall j \in C_2(\{q, r\}), \forall j \in C_1(\{p, r\})$
(2c) (Row r): $x_j = 0 \; \forall j \in C_3(S), \forall j \in C_2(\{p, r\}), \forall j \in C_2(\{q, r\}), \forall j \in C_1(\{p, q\})$

For node (2a), we have designated row p as the the row covered by column that does not additionally intersect rows q or r. With this additional condition, we can fix $x_j = 0$ if column j intersects all of rows rows p, q and r, or it if it intersects p and q or p and r. Additionally, since the other two rows (q and r) in S must then be covered by exactly one column at node (2a), we can fix variables x_j to zero where $j \in C_1(\{q, r\})$. The logic for nodes (2b) and (2c) are similar. Thus to implement our row-subset branching for $|S| = 3$, wherein we only fix variables to zero, we require 5 child nodes. (1), (2a), (2b), (2c), and (3).

It can be shown that by a similar recursive procedure, one can implement row-partition branching by *only* fixing variables to zero at each of the child nodes. However, due to the enumeration that must be done, the number of child nodes starts to grow rapidly. For example, to implement 4-row partition branching by fixing only variables, one requires 15 child nodes. Our preliminary computational results for 4-partition branching were not encouraging, so the remainder of our computational results will be on evaluating the row-partition branching method for $s \leq 3$.

2.3. Correctness of Row-Partition Branching

The correctness and completeness of the row-partition branching method follows from a similar analysis of the Ryan-Foster method, which as previously explained, is merely the row-partition branching method for the case $s = 2$.

Definition 1. A set of rows $S \subseteq I$ is called *compatible* with a fractional solution $\hat{x}$ if the solution $\hat{x}$ is not in the feasible region of any child node created by the row-partition branching rule (3).

As pointed out in [5], if there is a basic fractional solution $\hat{x}$ to the linear programming relaxation of (SPP), then there must be at least two rows $p, q \in I$ such that

$$0 \leq \sum_{j \in J: a_{pj}=1, a_{qj}=1} \hat{x}_j < 1.$$

Thus, if $\hat{x}$ is a fractional basic solution, then there must exist two columns $j' \in J$, $j'' \in J$, with $0 < \hat{x}_{j'}, \hat{x}_{j''} < 1$ and two rows $p \in I, q \in I$ that form the "forbidden" [15] submatrix:

$$\begin{array}{c} \\ p \\ q \end{array} \begin{array}{c} \begin{array}{cc} j' & j'' \end{array} \\ \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \end{array}.$$

This implies that there always exists a compatible branching set of size $s = 2$ for a fractional basic solution $\hat{x}$. For larger values of $|S| = s$, we cannot always ensure that the current fractional solution $\hat{x}$ will be changed on each branch, but our selection mechanism for choosing the subset S on which to perform row-partition branching will always consider using sets of size $s = 2$, so we can conclude that the branching method is complete and correct.

3. Implementation

In our implementation, we must select either a pair of rows $S = \{p, q\}$ or a triple of rows $S = \{p, q, r\}$ in order to perform row-partition branching. Given a fractional solution $\hat{x}$, there may be many compatible pairs or triples of rows. In this section, we discuss issues related to implementing the method and making a good selection for a row-subset without enumerating all possible pairs or triples.

3.1. Pseudocosts

The ranking mechanisms in our method rely heavily on pseudocosts. Recall that the (down) pseudocost ψ_j^- of a binary variable j is an estimate of the rate of change of the optimal objective value of the LP relaxation if the variable is fixed to take value 0. Likewise, the (up) pseudocost ψ_j^+ of a binary variable x_j is an estimate of the rate of change of the optimal objective value of the LP relaxation if the variable is fixed to take value 1. Our implementation uses the callable library of the commercial optimization package CPLEX, which has a function `CPXgetcallbackpseudocosts()` which will return the values $\psi_j^-, \psi_j^+ \; \forall j \in J$. We define

$$\mathcal{F}(\hat{x}) \stackrel{\text{def}}{=} \{j \in J : 0 < \hat{x}_j < 1\}$$

as the set of fractional variables in a given solution $\hat{x}$ and the set $\mathcal{F}_i(\hat{x}) \stackrel{\text{def}}{=} \mathcal{F}(\hat{x}) \cap J_i$ as the set of fractional variables of $\hat{x}$ that appear in row $i \in I$.

3.2. Ranking Rows

The first step in our implementation method is to rank all of the rows of the based on how "important" this row is with respect to the fractional solution $\hat{x}$. This is an important step

that we use in order to generate a good branching set without completely enumerating all pairs or triples of rows. Rows are ranked (from highest to lowest) according to the score

$$\Psi_i(\hat{x}) = \sum_{j \in \mathcal{F}_i} \psi_j^- \hat{x}_j. \tag{6}$$

The value $\Psi_i(\hat{x})$ is an estimate of how much the objective value of the LP relaxation would change if each fractional variable from row $i \in I$ was (individually) set to zero. We justify the ranking mechanism (6) by noting that in row-partition branching, each variable $x_j, j \in J_i$ will be set to zero on *one* of the branches.

3.3. Finding Good Branching Row Sets

Even if the elements of a pair or triple of rows each have a high score according to (6), together they may not lead to a good branching set for row-partition branching. In fact, they may not even be compatible according to Definition 1. We therefore must consider how the rows "fit-together" with respect to the current fractional solution $\hat{x}$. As previously mentioned, we would like to avoid enumerating of all pairs or triples. Our heuristic for finding a good branching row set is to find a good pair of rows and to then see if the pair can be augmented with a third row that seems likely to significantly improve the strength of our branching decision.

The first step of our method is to create a collection of at most δ_1 "good" pairs of compatible rows. Row pairs are considered for compatibility in the ordered generated by their row-specific rank from (6). We call this list of initial compatible candidates pairs $\mathcal{P}$. For each pair $(p,q) \in \mathcal{P}$, we loop over the rows $q \in I \setminus \{p,q\}$ and determine if the set $S = \{p,q,r\}$ is compatible and if the resulting 3-row partition branching is likely to be a "strong" branch.

For our method, we need a mechanism to compare the relative strength of row partition branches coming from different sets S. To compute the score for a set of rows S, we estimate the change for each child node using pseudocosts. Since we only implement row-partition branching for $|S| = s \in \{2,3\}$, it is simple to explicitly write down the formulas for the estimated increase in LP relaxation objective value for each child.

If $|S| = s = 2$, the estimated LP increase on each child node is

$$\begin{aligned} \text{Node 1:} \quad & D_1(\hat{x}, S) = \sum_{j \in C_1(S)} \psi_j^- \hat{x}_j \\ \text{Node 2:} \quad & D_2(\hat{x}, S) = \sum_{j \in C_2(S)} \psi_j^- \hat{x}_j. \end{aligned}$$

These two values are combined into a single score by taking a weighted sum of the scores of the individual child nodes, with some extra emphasis given for the sets that ensure that the *minimum* improvement is as large as possible, as is suggested to be important in [21].

$$\Upsilon_2(S) = D_1(S) + D_2(S) + \lambda \min\{D_1(S), D_2(S)\}. \tag{7}$$

If $S = \{p,q,r\}$ has $s = 3$ elements, the calculation of a score for ranking different possible 3-sets is a bit more involved, but is based on the variable zero-fixing explained in Section 2.2. If we let F_n be the set of variables fixed to zero on node $n \in \{1, 2a, 2b, 2c, 3\}$, we have

$$\begin{aligned} F_1(S) &= C_1(S) \cup C_2(S) \\ F_{2a}(S) &= C_3(S) \cup C_2(\{p,q\}) \cup C_2(\{p,r\}) \cup C_1(\{q,r\}) \\ F_{2b}(S) &= C_3(S) \cup C_2(\{p,q\}) \cup C_2(\{q,r\}) \cup C_1(\{p,r\}) \\ F_{2c}(S) &= C_3(S) \cup C_2(\{p,r\}) \cup C_2(\{q,r\}) \cup C_1(\{p,q\}) \\ F_3(S) &= C_2(S) \cup C_3(S), \end{aligned}$$

and we can define the estimated LP change on each node n as

$$E_n(S) = \sum_{j \in F_n(S)} \hat{x}_j \psi_j^- .$$

Again, these five values are combined into a single score by adding the individual contributions and giving extra weight to the smallest value.

$$\Upsilon_3(S) = \sum_{n \in N_3} E_n(S) + \lambda \min_{n \in N_3} E_n(S). \tag{8}$$

In both Equations 7 and 8, we use $\lambda = |S|$.

If we do "pure" Ryan-Foster (or $s = 2$) branching, we choose to branch on the set

$$S_2^* \in \arg\max_{S \in \mathcal{P}} \Upsilon_2(S). \tag{9}$$

If we do "pure" 3-row partition branching, we choose to branch on the set

$$S_3^* \in \underset{(p,q,r) \,:\, (p,q) \in \mathcal{P},\, q \in I \text{ such that } \{(p,q,r)\} \text{ is compatible}}{\arg\max} \Upsilon_3(\{p,q,r\}). \tag{10}$$

If we are attempting to do 3-row partition branching, and there are no compatible triples of rows, we branch on the pair given by (9).

As a final branching method, we consider a "dynamic" method that may choose to do 3-row partition branching only if the resulting child nodes are estimated to be "strong" compared to the best two-row set. In this method, we choose the set specified by S_3^* if instead of S_2^* if

$$\Upsilon_3(S_3^*) > \frac{5}{2} \Upsilon_2(S_2^*). \tag{11}$$

4. Computational Results

In this section, we empirically demonstrate the performance of Row-Partition branching. We also test the performance of other branching methods designed to exploit the structure of GUB constraints. We will describe our implementation of these branching methods, explain the test set of SPP instances used, and describe the results.

4.1. Branching variations

The SPP-specific branching methods we compare to the row-partition method are GUB branching and multi-node branching. We briefly describe each of these methods, including necessary algorithmic choices for implementation. It was not the goal of this line of research to do an exhaustive investigation of branching methods, but rather we attempted to implement "reasonable" selection mechanisms for each comparative branching methodology.

4.1.1. GUB Branching As explained in the introduction, GUB branching is one of the most traditional branching methods designed to exploit the structure of the GUB constraints appearing in (SPP). If row $i \in I$ is selected for branching, a subset $Q \subset J_i$ of the variables is selected and child nodes are created to enforce

$$\sum_{j \in J_i \setminus Q} x_j = 0 \qquad \text{or} \qquad \sum_{j \in Q} x_j = 0.$$

In our implementation, given a fractional solution $\hat{x}$, we perform GUB branching on the row $i^* \in I$ of the SPP instance that has the highest score (6):

$$i^* \in \arg\max_{i \in I} \Psi_i(\hat{x}). \tag{12}$$

The set Q is selected by choosing a "branch point" within the set J_i that attempts to put $1/2$ of the current fractional solution $\hat{x}$ on variables in Q and $1/2$ in $J_i \setminus Q$. This can be implemented by choosing a "branch point"

$$B = \sum_{j \in J_i} \mathrm{ord}(j)\, \hat{x}_j,$$

where $\mathrm{ord}(j)$ is the order of the element j in the set J_i. With this definition, the set Q is simply

$$Q = \{j \in J_i \mid j < B\}.$$

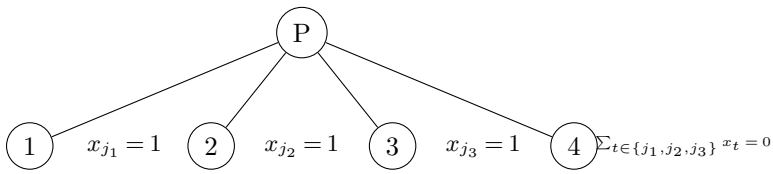
4.2. Multi-Node Branching

In multi-node branching, each fractional variable appearing in the row is individually set to 1 on a child node, with one child node handling the case in which all of these variables simultaneously take the value 0:

$$\left(\bigvee_{j \in \mathcal{F}_i(\hat{x})} (x_j = 1) \right) \vee \left(\sum_{k \in \mathcal{F}_i(\hat{x})} x_k = 0 \right).$$

This branching rule is part of the computational integer programming folklore. It has been used in [18] to create strong valid inequalities for MIP instances that contains GUB constraints. An example multinode branching dichotomy is given in Figure 2, where $\mathcal{F}_i(\hat{x}) = \{j_1, j_2, j_3\}$.

FIGURE 2. Example of (MN) branching



In our implementation, to determine the row $i \in I$ on which to perform the multinode branching, we compute the average estimated change in the LP relaxation value and select the row $i^* \in I$ with the largest average estimated change:

$$i^* \in \arg\max_{i \in I} \frac{\sum_{j \in \mathcal{F}_i(\hat{x})} (1 - \hat{x}_j)\psi_j^+ \; + \; \sum_{j \in \mathcal{F}_i(\hat{x})} \hat{x}_j \psi_j^-}{|\mathcal{F}_i(\hat{x})| + 1}. \tag{13}$$

In (13), for computing the estimated change to LP relaxation value, the "down" pseudocosts are only counted for the "all-zero" branch.

4.3. Test set

Assessing the performance of branching methods is not simple. There are many factors that affect the performance of each branching method other than the branching decision. For example, performance is dramatically affected by the existence of a high-quality feasible solution to be used in pruning. Also, advanced integer programming features found in modern codes, such as presolve ans cutting planes can interact with the instance in a manner that is difficult to anticipate. And recently authors have demonstrated that there is large inherent variability in the search procedure itself [14]. Put together, these factors contribute to a very large variance in the results, which may make it difficult to draw a meaningful conclusions about the relative performance of methods. For this reasons, it is necessary to conduct experiments with a large number of instances.

The first family of instances in our test set are publicly available (SPP) instances from MIPLIB and COR@L. For each of these "base" instances, we create 30 *scrambled* instances by randomly changing the order of the rows and columns. Every scrambled instance has the same objective value as the original instance but has a different coefficient matrix. The second family of instances in our test set are constructed randomly. Each element a_{ij} is an (independent) Bernoulli random variable with parameter μ. In the instances, we induce some correlation between the magnitude of the objective function coefficient for variable j and the number of ones in its associated column of A. These *correlated-coefficient*(CC) instances mimic characteristics of those seen in practice. For example, in crew-scheduling the rows represent flight legs in a schedule, and the cost of a crew pairing should be correlated to the number of flight legs in the pairing. The random instances have sizes ranging from $100 \leq m \leq 1000$ and $1100 \leq n \leq 14000$ variables. The instances are categorized as "sparse" if $\mu < 0.02$, and "dense" otherwise. We also create instances that have *set packing* constraints of the form

$$\sum_{j \in J_i} x_j \leq 1. \tag{14}$$

These constraints can easily be transformed into GUB constraints (1) by the addition of a slack (binary) variable. The instances subsequently categorized as "PAC" instances may have some fraction of their constraints as (14). The instances labeled as PAR are pure set partitioning.

4.4. Experimental Settings

We implemented the branching methods using the branching callback methods available in CPLEX 12.5. We attempted to use nearly all CPLEX default settings, with two major exceptions. First, we disable the presolve aggregation option in CPLEX, since this may modify the structure of the instance so it is no longer a pure SPP instance, which makes performing our branching methods via callback difficult. Second, since branching methods are primarily designed to improve the lower bound of the search, we felt that we may remove a source of variation in performance if we instructed CPLEX to perform a "pure" best-bound search. Thus, for our experiments, we used all CPLEX default parameters except the following:

```
CPX_PARAM_AGGIND = 0
CPX_PARAM_MIPSEARCH = CPX_MIPSEARCH_TRADITIONAL
CPX_PARAM_NODESEL = 1 (Best Bound)
CPX_PARAM_BBINTERVAL = 1 (Always Choose)
CPX_PARAM_BTTOL = 0.0 (Most backtrack)
```

Our computational experiments were run on a cluster of machines that are scheduled via the HTCondor scheduling mechanism. The hardware on the machines running the experiments *was not identical*, so traditional measures of performance (such as CPU time) are not directly relevant. Instead, we use as a primary measure of effectiveness the number of nodes required to solve the problem to optimality.

4.5. Experiments and Results

For the different classes of instances, we compare the following six different branching methods:

(1) CPLEX: CPLEX default branching
(2) GUB: Our GUB branching implementation described in Section 4.1.1
(3) MN: Multinode-branching described in Section 4.2
(4) 2R: Two row (Ryan-Foster) branching with the branching set selected by (7)
(5) 3R: Three row partition-branching with the branching set selected by (8)

(6) DM: The "dynamic method" that attempts to determine if the (compatible) 3R branching is more effective than 2R branching using (11).

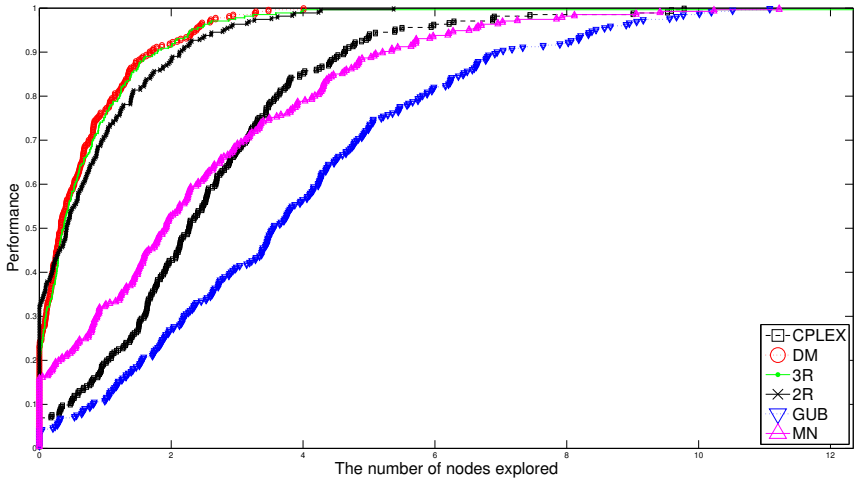
To limit the search for effective row pairs or row triples, we use the parameter $\delta_1 = 100$. Thus, when considering $2R$, $3R$, or DM branching, the initial candidate set of row pairs has cardinality $|\mathcal{P}| \leq 100$.

We summarize the outcome of the computational results by breaking the instances into different categories. For the randomly constructed instances, the categories are based on the size of the instance and whether or not the instance contained some percentage of set packing constraints or if it was a pure set partitioning instance. For these instances, the (arithmetic) average number of nodes explored by each method, the number of times the method reached the CPU limit of 3 hours, and the number of times each method resulted in the smallest search tree is reported for each category of instances in Table 1. The computational results show that in terms of nodes explored, the 2-row-partition branching method, the 3-row

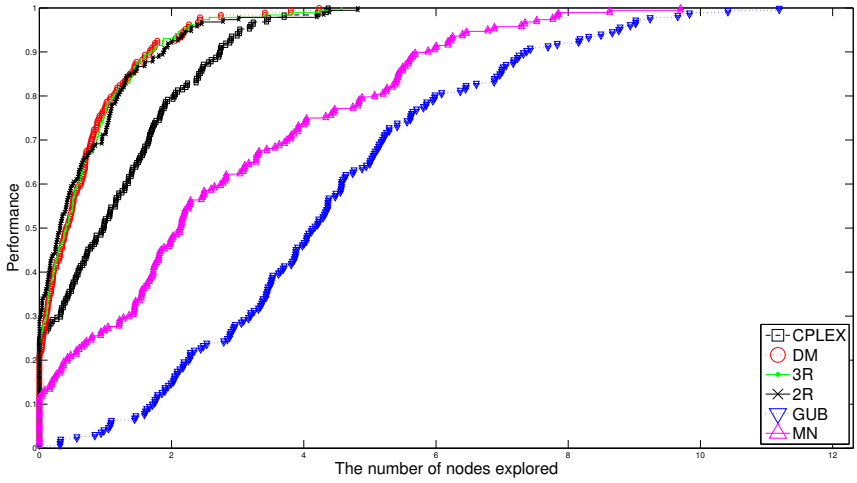
Table 1. Results Summary of CC instances

	avg node explored	# time to win	# reach time limit
PAR, sparse, small size of m (m=100, 200), $n \in [1100, 5900]$, 85 instances			
CPLEX	3.07E+05	4	0
DM	84776	25	1
3-Row	84563	24	0
2-Row	86530	26	1
GUB	1.48E+06	0	14
MN	3.94E+05	7	10
PAR, sparse, medium size of m (m=300, 400), $n \in [1800, 10800]$, 68 instances			
CPLEX	9.97E+05	3	10
DM	94758	17	0
3-Row	95760	14	2
2-Row	99484	34	0
GUB	1.10E+06	1	13
MN	4.60E+05	1	27
PAR, sparse, large size of m (m=500, 600, 700, 800, 1000),$n \in [2000, 14000]$, 54 instances			
CPLEX	7.20E+05	1	8
DM	77716	15	2
3-Row	77564	10	2
2-Row	86218	23	2
GUB	4.75E+05	4	11
MN	2.75E+05	1	13
PAR, dense, small size of m (m=100, 200), $n \in [1200, 10100]$, 55 instances			
CPLEX	60724	8	0
DM	59364	9	0
3-Row	63437	5	0
2-Row	74144	6	1
GUB	4.47E+05	0	2
MN	72299	30	0
PAR, dense, large size of m (m=300,400,500,700), $n \in [3300, 8800]$, 33 instances			
CPLEX	31159	6	4
DM	29566	8	2
3-Row	31896	6	4
2-Row	30154	6	3
GUB	3.62E+05	5	5
MN	1.62E+05	4	5
PAC, sparse (m=100,200,300), $n \in [2000, 6700]$, 141 instances			
CPLEX	1.27E+05	31	0
DM	58772	32	6
3-Row	61332	29	6
2-Row	44514	48	7
GUB	1.70E+06	0	20
MN	4.18E+05	9	17
PAC, dense (m=100,200,500), $n \in [1300, 9100]$, 63 instances			
CPLEX	22120	22	0
DM	20345	11	0
3-Row	21836	14	3
2-Row	30504	9	3
GUB	2.66E+05	1	9
MN	1.32E+05	14	8

FIGURE 3. Performance comparison—number of nodes explored—CC instance



(a) PAR instance



(b) PAC instance

partition branching method, and the dynamic method are all of comparable quality, and all appear to perform better than all other methods. The results over instances solved by each method are summarized with performance profiles [11] in Figure 3.

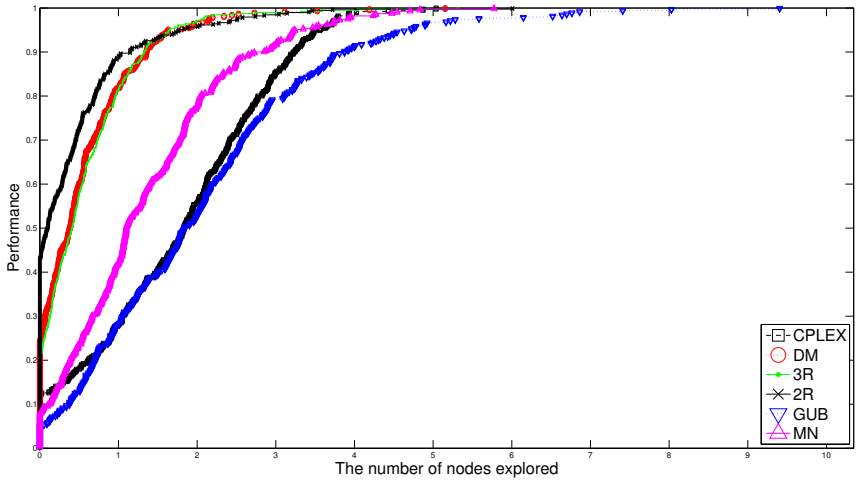
Table 2 summarizes the computational results of the different branching methods on the publicly-available instances used in our tests. In the table, both the arithmetic average of the number of nodes and the standard deviation of the number of nodes is reported from the 30 different scrambled versions of the same instance. The computational results again show that the row-partition-based methods perform well, but the improvement over the default branching mechanism of CPLEX is less pronounced. The performance profile given in Figure 4 shows that the Ryan-Foster branching method is most effective on these instances.

In a final experiment, we compared CPLEX default with 2R branching and the dynamic method DM on large, randomly-generated instances that were very difficult to solve. The results of this experiment are reported in Table 3. The instances are named `Par-m-n-nz`, where m is the number of rows, n is the number of columns, and nz is the number of nonzero elements in the A matrix. For these larger instances, it seems that doing branching based on a 3-row partition can lead to significant computational improvement.

TABLE 2. Results Summary of Benchmark instance

instances	CPLEX		DM		3R		2R		GUB		MN	
	Node	Std	Node	Std	Node	Std	Node	Std	Node	Std	Node	Std
eil33.2(30 instances)	10667	1824	3190	656	3206	817	3277	642	12060	2545	5168	823
eilA76(30 instances)	137.4	61	63	32	64.2	31	67.1	39.3	114.2	81	76.7	40.4
eilB76(30 instances)	436.5	208	268.1	162	272	145	199.7	68	497.1	382	290.3	174
eilC76(30 instances)	85.8	61	29.1	20	28.2	16	23.7	10.4	45.8	28	40.6	14.3
eilD76(30 instances)	128.2	42	36	10.2	36.1	9.9	29	9.3	53.1	17.6	81.7	20.9
eilB101(30 instances)	13382	6814	3033	967	3127	1270	2648	931	4059	2262	4304	1812
air04(30 instances)	279.1	171	941.1	845	878	869	1201	1505	12539	17925	3012	3180
air05(30 instances)	1243	488	1470	546	1626	567	1216	512	4791	2045	3210	1579
nw04(30 instances)	116.7	138	65.1	71	63.5	71.7	68.8	67.9	302	325	169	143
bills.SF(30 instances)	164.3	72.5	152.1	96	147.8	55	129.7	84	151.2	91.7	150.4	90.1
eilC76.2(30 instances)	23414	5357	3295	1100	3816	1721	3343	1031	21925	13921	12045	3635
eilD76.2(30 instances)	1.3e5	29154	30357	9140	29820	8673	20073	5259	1.8e5	64290	27813	5218
eilB101.2(30 instances)	20189	9821	3004	1784	2679	1318	2918	1679	31836	98965	7046	3879

FIGURE 4. Performance comparison - number of nodes explored - benchmark instance



benchmark instance

TABLE 3. Results for very large instances

instances	CPLEX Node explored	DM Node explored	2R Node explored	time limit (hrs)
Par-400-6800-104337	≥1.77E+07	1.34E+006	*8.77E+005	20
Par-400-7400-114035	1.52E+07	*2.53E+006	≥ 3.04E+006	20
Par-400-8200-127005	4.81E+06	1.47E+006	*9.36E+005	20
Par-400-9800-152875	2.12E+07	*1.97E+006	3.33E+006	20
Par-500-7300-138394	≥2.54E+06	6.29E+005	*4.48E+005	20
Par-500-9700-187030	≥2.62E+06	*1.24E+006	≥ 2.67E+006	20
Par-500-9900-191019	≥4.97E+06	*1.38E+006	1.56E+006	20
Par-800-5800-123219	≥ 1.04E+007	*6.80E+005	1.38E+006	20
Par-800-7800-228002	≥ 8.49E+006	*1.06E+006	≥ 1.42E+006	20
Par-900-9700-275822	2.83E+006	*3.70E+004	2.83E+005	20
Par-900-9900-362228	≥ 3.61E+006	*1.53E+006	≥ 1.42E+006	20
Par-1000-9000-485532	≥ 7.79E+006	*4.11E+005	≥ 2.46E+006	20
Par-1000-12000-667103	≥ 3.82E+006	*7.27E+005	1.85E+006	20
Par-1000-14000-657909	9.49E+005	*2.06E+005	5.39E+005	20
Par-800-7600-177515	≥1.57E+06	*2.64E+006	≥ 2.22E+006	30

5. Conclusion

In this study, we described a novel branching strategy based on partitioning a subset of the rows for set partitioning and set packing problems. The method is a natural generalization of

the Ryan-Foster method. We described an implementation that attempts to choose a "good" subset of rows on which to base the branching, and demonstrated through computational experiments that the method may hold some promise as a methodology for branching for these classes of problems. We plan to continue to work on testing additional methods for selecting the branching set, and we hope to identify classes of instances for which row-partition branching is effective for row-subset of size $s > 3$.

References

[1] J. Appleget and R. K. Wood. Explicit-constraint branching for solving mixed-integer programs. In M. Laguna and J. Velarde, editors, *Computing Tools for Modeling, Optimization and Simulation*, volume 12 of *Operations Research/Computer Science Interfaces Series*, pages 245–261. Springer US, 2000.

[2] A. Atamtürk, G. L. Nemhauser, and M. W. P. Savelsbergh. A combined Lagrangian, linear programming, and implication heuristic for large-scale set partitioning problems. *Journal of Heuristics*, 1:247–259, 1995.

[3] E. Balas and M. Padberg. Set partitioning: A survey. *SIAM Review*, 18:710–760, 1976.

[4] M. Balinski and R. Quandt. On an integer program for a delivery problem. *Operations Research*, 12:300–304, 1964.

[5] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch and price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998.

[6] M.A. Boschetti, A. Mingozzi, and S. Ricciardelli. A dual ascent procedure for the set partitioning problem. *Discrete Optimization*, 5:735–747, 2008.

[7] D. Bredström, K. Jörnsten, M. Rönnqvist, and M. Bouchard. Searching for optimal integer solutions to set partitioning problems using column generation. *International Transactions in Operational Research*, 21:117–197, 2014.

[8] P. C. Chu and J. E. Beasley. Constraint handling in genetic algorithms: The set partitioning problem. *Journal of Heuristics*, 4:323–357, 1998.

[9] G. Cornuéjols, L. Liberti, and G. Nannicini. Improved strategies for branching on general disjunctions. *Mathematical Programming*, 130:225–247, 2011.

[10] H. Crowder, E. L. Johnson, and M. W. Padberg. Solving large scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[11] E. Dolan and J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

[12] L. Escudero. S3 sets. An entenion of the Beale-Tomlin special ordered sets. *Mathematical Programming*, 42:113–123, 1988.

[13] M. Esö. *Parallel Branch and Cut for Set Partitioning*. PhD thesis, Department of Operations Research and Industrial Engineering, Cornell University, 1999.

[14] M. Fischetti and M. Monaci. Exploiting erraticism in search. *Operations Research*, 62:114–122, 2014.

[15] A. Hoffman, A. Kolen, and M. Sakarovitch. Totally balanced and greedy matrices. *SIAM Journal on Algebraic and Discrete Methods*, 6:721–730, 1985.

[16] K. Hoffman and M. Padberg. Solving airline crew-scheduling problems by branch-and-cut. *Management Science*, 39:667–682, 1993.

[17] W. Hummeltenberg. Implementations of special ordered sets in MP software. *European Journal of Operations Research*, 17:1–15, 1984.

[18] M. Kılınç, J. Linderoth, J. Luedtke, and A. Miller. Strong branching inequalities for convex mixed integer nonlinear programs. *Computational Optimization and Applications*, 59:639-665, 2014.

[19] D. Levine. *A Parallel Genetic Algorithm for the Set Partitioning Problem*. PhD thesis, Illinois Institute of Technology, Chicago, IL, 1995.

[20] J. T. Linderoth, E. K. Lee, and M. W. P. Savelsbergh. A parallel, linear programming based heuristic for large scale set partitioning problems. *INFORMS Journal on Computing*, 13:191–209, 2001.

[21] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies in mixed integer programming. *INFORMS Journal on Computing*, 11:173–187, 1999.

[22] A. Mahajan and T. K. Ralphs. Experiments with branching using general disjunctions. In *Proceedings of the Eleventh INFORMS Computing Society Meeting*, pages 101–118, 2009.

[23] R. E. Marsten and F. Shepardson. Exact solution of crew problems using the set partitioning mode: Recent successful applications. *Networks*, 11:165–177, 1981.

[24] T. Müller. Solving set partitioning problems with constraint programming. In *Proceedings of the Sixth International Conference on Practical Applications of Prolog and the Fourth International Conference on the Practical Application of Constraint Technology – PAPPACT98*, pages 313–332, London, U.K., 1998. The Practical Application Company Ltd.

[25] D. Ryan and B. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport*, pages 269–280. North Holland, Amsterdam, 1981.

[26] M. B. Ryoo. A constraint branch-and-bound method for set partitioning problems. Master's thesis, Naval Postgraduate School, Monterey, CA, 1990.

[27] R. Saldanha and E. Morgado. Solving set partitioning problems with global constraint propagation. In F. M. Pires and S. Abreu, editors, *Progress in Artificial Intelligence*, volume 2902 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2003.