Computational Study of a Branching Algorithm for the Maximum k-Cut Problem

Computational study of a branching algorithm for the maximum k-cut problem

Vilmar Jefté Rodrigues de Sousa* Miguel F. Anjos†
Sébastien Le Digabel‡

June 11, 2021

Abstract. This work considers the graph partitioning problem known as maximum k-cut. It focuses on investigating features of a branch-and-bound method to obtain global solutions. An exhaustive experimental study is carried out for the two main components of a branch-and-bound algorithm: Computing bounds and branching strategies. In particular, we propose the use of a variable neighborhood search metaheuristic to compute good feasible solutions, the k-chotomic strategy to split the problem, and a branching rule based on edge weights to select variables. Moreover, we analyze a linear relaxation strengthened by semidefinite-based constraints, a cutting plane algorithm, and node selection strategies. Computational results show that the resulting method outperforms the state-of-the-art approach and discovers the solution of several instances, especially for problems with $k \geq 5$.
Keywords. Maximum k-cut, branch-and-cut, graph partitioning, semidefinite programming, eigenvalue constraint.
AMS subject classifications. 90C57, 90C27

1 Introduction

The maximum k-cut problem (max-k-cut) is a graph partitioning problem in an undirected graph $G = (V, E)$ with weights on the edges. The goal is to maximize the sum of the weights that are cut, i.e., those on edges with endpoints (vertices) in different partitions. Additionally, the vertex set V can be partitioned into at most k subsets. Max-k-cut is a challenging combinatorial problem that is known to be $\mathcal{NP}$-hard [41] , and it has attracted much scientific attention from the discrete community, e.g., [5, 14, 22, 30, 40, 42].

We perform a computational study of some of the main components of a branch-and-bound method; our goal is to solve max-k-cut problem to optimality. For the upper bounds, we investigate the performance of five relaxations and the use of a cutting plane algorithm (CPA). For the lower bounds, we compare four methods and propose two new heuristics to find good feasible solutions.

*vilmar-jefte.rodrigues-de-sousa@polymtl.ca GERAD and Département de Mathématiques et Génie Industriel, Polytechnique Montréal
†www.miguelanjos.com GERAD and School of Mathematics, University of Edinburgh
‡www.gerad.ca/Sebastien.Le.Digabel GERAD and Département de Mathématiques et Génie Industriel, Polytechnique Montréal

We also propose several methods for splitting, branching, and branch selection. To the best of our knowledge, no research has investigated all these components of branch-and-bound for max-k-cut.

This work is organized as follows. Section 2 reviews techniques for obtaining strong relaxations and solving the problem. Section 3 presents a computational study of a large branch-and-bound framework. Section 4 provides a comparison with the state-of-the-art approach. Finally, Section 5 discusses the results and provides concluding remarks.

2 Literature review

This section presents popular recent methods for max-k-cut. It recalls some of the most important formulations and then presents solution approaches designed to find global solutions.

2.1 Formulations

This section presents three integer formulations of max-k-cut. The first considers only the edges of the graph as variables, the second takes into account the edges and the vertices, and the third includes only the vertices. We point out that in all the formulations, a variable x_{ij} or X_{ij} is equal to one if the vertices i and j are in the same partition.

2.1.1 Edge-only formulation

In the 0–1 edge formulation [8] the integer variable x_{ij} for each edge $\{i,j\} \in E$ is defined as

$$x_{ij} = \begin{cases} 0 & \text{if edge } \{i,j\} \text{ is cut,} \\ 1 & \text{otherwise.} \end{cases}$$

Hence, from an edge perspective, max-k-cut is formulated as

$$\max_{x} \sum_{\{i,j\}\in E} w_{ij}(1 - x_{ij}) \tag{1}$$

$$x_{ih} + x_{hj} - x_{ij} \leq 1 \qquad \forall \{\{i,h\},\{h,j\},\{i,j\}\} \subseteq E \tag{2}$$

$$\sum_{i,j\in Q, i<j | \{i,j\}\in E} x_{ij} \geq 1 \qquad \forall Q \subseteq V \text{ with } |Q| = k+1 \tag{3}$$

$$x_{ij} \in \{0,1\} \qquad \forall \{i,j\} \in E \tag{4}$$

where Constraints (2) and (3) are called *triangle* and *clique* inequalities, respectively. The triangle Inequalities (2) correspond to the logical conditions that if edges $\{i,h\}$ and $\{h,j\}$ are not cut, then edge $\{i,j\}$ is not cut. Constraints (3) ensure that at least one edge in a clique with $k+1$ vertices cannot be cut. For the edge-only formulation, the graph must be at least chordal; if it is not, dummy edges (of weight zero) must be added (see [48]). Note however that for sparse nonchordal graphs, a chordal extension [24] will generally add many dummy edges.

2.1.2 Vertex-and-edge formulation

In [8], the authors present a vertex-and-edge formulation with $|V|k + |E|$ variables and constraints. For each $v \in V$, $\{i, j\} \in E$, and $p \in \{1, 2, \ldots, k\}$, the binary variables are

$$x_{ij} = \begin{cases} 0 & \text{if edge } \{i,j\} \text{ is cut,} \\ 1 & \text{otherwise} \end{cases} \quad \text{and } y_{vp} = \begin{cases} 1 & \text{if vertex } v \text{ is in partition } p, \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, the vertex-and-edge integer formulation of max-k-cut is:

$$\max_{x,y} \sum_{\{i,j\} \in E} w_{ij}(1 - x_{ij}) \tag{5}$$

$$\sum_{p=1}^{k} y_{vp} = 1 \qquad \forall\, v \in V \tag{6}$$

$$x_{ij} \geq y_{ip} + y_{jp} - 1 \qquad \forall\, (\{i,j\} \in E, p = 1, 2, \ldots, k) \tag{7}$$

$$x_{ij} \leq y_{ip} - y_{jp} + 1 \qquad \forall\, (\{i,j\} \in E, p = 1, 2, \ldots, k) \tag{8}$$

$$x_{ij} \leq -y_{ip} + y_{jp} + 1 \qquad \forall\, (\{i,j\} \in E, p = 1, 2, \ldots, k) \tag{9}$$

$$x_{ij} \in \{0, 1\} \qquad \forall\, \{i,j\} \in E \tag{10}$$

$$y_{vp} \in \{0, 1\} \qquad \forall\, (v \in V, p \in \{1, 2, \ldots, k\}) \tag{11}$$

Constraint (6) ensures that every vertex is in exactly one partition, and Constraints (7) to (9) ensure that edges are cut when their vertices are in different partitions. Constraint (7) can be removed if the respective edge weight is negative. Constraints (8) and (9) may be removed when all the edge weights of an instance are non-negative.

2.1.3 Vertex-only formulation

The third formulation is based on the vertices of the graph. In [17], the authors observe that defining the variables $x_i \in \{1, 2, \ldots, k\}$ for each vertex $i \in V$ does not generate useful integer formulations. Instead, they define x to be one of the k vectors $a_1, a_2, \ldots, a_k$ defined as $a_i = b_i - c$ for $1 \leq i \leq k$ where $b_1, b_2, \ldots, b_k$ are the vertices of an equilateral $(k-1)$-simplex in $\mathbb{R}^{k-1}$, and $c = (b_1 + b_2 + \ldots + b_k)/k$ is its centroid. Assume that this simplex is scaled so that the two-norm of a is one. Then the inner product

$$a_i^\top a_j = \frac{-1}{k-1} \qquad \forall\, i, j \in V, i \neq j.$$

This is a property of the vectors a_i. The value $\frac{-1}{k-1}$ provides the best possible separation [17, Lemma 2]. Therefore, for the variables $x_i \in \{a_1, a_2, \ldots, a_k\}$:

$$x_i^\top x_j = \begin{cases} \frac{-1}{k-1} & \text{if } x_i \neq x_j \text{ (i.e., vertices } i \text{ and } j \text{ are in different partitions),} \\ 1 & \text{if } x_i = x_j. \end{cases}$$

This leads to the following quadratic formulation of max-k-cut [17]:

$$\max_{x} \frac{(k-1)}{k} \sum_{\{i,j\} \in E} w_{ij}(1 - x_i^\top x_j) \tag{12}$$

$$x_i \in \{a_1, a_2, \ldots, a_k\} \qquad \forall\, i \in V. \tag{13}$$

where the objective function adds up precisely the weights corresponding to the edges with endpoints in different partitions.

2.2 Branch-and-bound components

This section reviews the main components of the branch-and-bound algorithm which is widely applied to obtain global solutions of max-k-cut and other combinatorial problems [38]. Fundamentally, the branch-and-bound method is a tree search strategy. The algorithm starts from a node (the *root node*) providing an upper bound solution of a max-k-cut. At each iteration i, a node not yet explored, is selected (*node selection*) and the algorithm checks if this node can be fathomed (pruned), i.e., if it is integer feasible, or infeasible, or its upper bound is not better than the best feasible solution so far (also called the incumbent solution). If the node cannot be fathomed, the algorithm will generate child nodes (*branching*) with a more restricted relaxation of the problem. In this way, the upper and lower bounds are iteratively improved until optimality can be proved.

2.2.1 Upper bound

We now recall some linear programming (LP) and semidefinite programming (SDP) relaxations as well as the most important valid inequalities for max-k-cut.

LP relaxation: An advantage of applying an LP relaxation in branch-and-bound is the possibility to use the standard technique of reduced-cost fixing [9], and the simplex method can readily exploit a starting basis [34]. Therefore, solvers (e.g., `mosek` [4]) can use the solution of previously selected nodes in the tree to reduce the computational time. The edge-only LP relaxation is obtained from Formulation (1)-(4) by relaxing the binary Constraints (4) to:

$$0 \leq x_{ij} \leq 1 \qquad \forall \{i,j\} \in E \tag{14}$$

A disadvantage of this relaxation is the potentially large number of inequalities. While the triangle Inequalities (2) will be no more than $3\binom{|V|}{3}$ (in the case of a complete graph), the clique Inequalities (3) can be up to $\binom{|V|}{k+1} = O(|V|^{k+1})$, and thus even for sparse graphs, the number of clique inequalities may be impractical.

The vertex-and-edge LP relaxation is obtained from Formulation (5)-(11) by replacing Constraints (10) and (11) by the following continuous constraints:

$$0 \leq x_{ij} \leq 1 \qquad \forall \{i,j\} \in E \tag{15}$$

$$0 \leq y_{vp} \leq 1 \qquad \forall (v \in V, p \in \{1,2,\ldots,k\}) \tag{16}$$

This relaxation is weak and suffers from symmetry [14]. It can be improved by the so-called representative formulations [2] where a representative variable is added to break some symmetry. However, this extended formulation can add several variables and constraints to the model.

SDP relaxation: An SDP relaxation is obtained from the quadratic vertex-only Formulation (12)–(13) by replacing Constraint (13) with

$$x_i \in B_{k-1} \quad \forall\, i \in V \tag{17}$$

where B_n is the unit sphere in n dimensions. To eliminate solutions in the relaxation where $x_i^\top x_j = -1$, we add the constraints $x_i^\top x_j \geq \frac{-1}{k-1}$ for all $i, j \in V$. Then, replacing the inner product $x_i^\top x_j$ by X_{ij} for all $i, j \in V$ in the positive semidefinite matrix ($X \succeq 0$) gives the following SDP Relaxation [17]:

$$\max_X \frac{(k-1)}{k} \sum_{i,j \in V, i<j} w_{ij}(1 - X_{ij}) \tag{18}$$

$$X_{ii} = 1 \qquad \forall\, i \in V \tag{19}$$

$$X_{ij} \geq \frac{-1}{k-1} \qquad \forall\, i, j \in V, i < j \tag{20}$$

$$X \succeq 0 \tag{21}$$

In [25], the author pointed out that adding all Constraints (20) can increase the computational time for solving the SDP formulation of max-k-cut. Therefore, it is more efficient to start with only Constraints (19) and to separate $X_{ij} \geq \frac{-1}{k-1}$ successively in a CPA [25]. Many other approaches have been proposed, e.g., [10, 13].

An advantage of the SDP relaxation is that it provides strong bounds, especially because some hypermetric inequalities are implicit [12]. However, it has an expensive processing time: See the recent analysis of [3, 45].

Cutting planes: In [7], the authors show that the following classes of inequalities (known as combinatorial inequalities) define facets for max-k-cut when $k \geq 3$:

- The *general clique* inequalities have the form:

$$\sum_{i,j \in Q} x_{ij} \geq z \qquad \forall\, Q \subseteq V, |Q| = p \tag{22}$$

  where $z = \frac{1}{2}t(t-1)(k-q) + \frac{1}{2}t(t+1)q$ and $p = tk + q$ for positive integers t, q.

- The *wheel* inequalities have the form:

$$\sum_{v_i \in C} x_{v_i u} - \sum_{(v_i v_j) \in O} x_{v_i v_j} \leq \left\lfloor \frac{|C|}{2} \right\rfloor \qquad \forall\, C \subseteq V, |C| \geq 3 \tag{23}$$

  where the hub vertex $u \in V$ such that $u \notin C$ links all the vertices in the odd cycle; and O is the set of edges in the cycle (the solid edges in Figure 1). It is proved in [7] that *wheel* inequalities are facet defining for max-k-cut if $|C|$ is odd and $k \geq 4$.

- The *bicycle wheel* inequalities have the form

$$\sum_{j \in \{1,2\}} \sum_{v_i \in C} x_{v_i u_j} - \sum_{(v_i v_j) \in O} x_{v_i v_j} - x_{u_1 u_2} \leq 2 \left\lfloor \frac{|C|}{2} \right\rfloor \qquad \forall\, C \subseteq V, |C| \geq 3 \tag{24}$$

  where $u_1 \in V$ and $u_2 \in V$ are vertices in the hub; and O is the set of edges in the cycle. In contrast to the wheel inequalities, these ones have two vertices in the hub. It is proved in [7] that, for max-k-cut, *bicycle wheel* inequalities are facet defining if $|C|$ is odd.

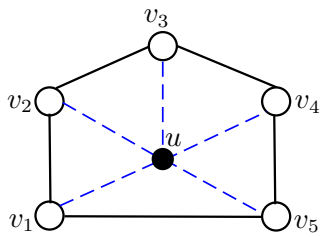

Figure 1: Example wheel constraint for $|C| = 5$.

Combinatorial Inequalities (22)-(24) can reinforce the three formulations presented above. In [47], the authors compare triangle Inequalities (2) and clique Inequalities (3) in the SDP relaxation. Their experiments show that triangle inequalities perform better. In [44], the authors show that triangle, wheel, and bicycle wheel are the most important classes of inequalities.

In [45], the authors proposed an SDP-based inequality to reinforce both edge-only or vertex-and-edge relaxations of max-k-cut. The following constraint is based on the linear semi-infinite programming approach for generic SDPs [46]:

$$\sum_{i,j \in V, i<j} \mu_i \mu_j x_{ij} \geq \frac{1}{k} \sum_{i,j \in V, i<j} \mu_i \mu_j - \frac{k-1}{2k} \sum_{i \in V} \mu_i \mu_i \qquad \forall\, \mu \in \mathbb{R}^n \tag{25}$$

where the Euclidean norm of μ is typically one.

Constraint (25) has an infinite number of rows, so a separation routine based on eigenvalues is also introduced in [45]. The computational results show that SDP-based inequalities are helpful when $k \geq 7$.

In [8], the authors investigate facet-defining hypermetric inequalities for max-k-cut. This class of constraints generalizes some of the previous inequalities, such as the triangle Inequalities (2) and clique Inequalities (3). For brevity and because the complexity of the separation in a CPA is unknown, we omit the details.

2.2.2 Lower bound

In the branch-and-bound algorithm, the incumbent solution is used to prune subproblems. If the method stops before optimality, this solution will be returned. Heuristics and metaheuristics are usually used to find initial feasible solutions and to improve the existing incumbent. For the max-cut problem ($k = 2$), many heuristics have been proposed (e.g., [29, 50]). For max-k-cut, there are few options.

In the 0.878-approximation algorithm for the max-cut problem [22], the idea is to solve an SDP formulation and then select a random hyperplane that passes through the origin. The vertices $v_i \in V$ are partitioned according to on which side of the hyperplane they fall. Extensions of the 0.878-approximation to the max-k-cut problem have been proposed [17, 28, 39].

In [20], the authors propose an iterative clustering heuristic (ICH) that finds feasible solutions based on the SDP primal solution. Tests show that ICH provides better feasible solutions than those obtained by [22] (for $k = 2$) and by [17] (for $k \geq 3$). In summary, ICH sums the value of the relaxed solution of the edges between three vertices; then, if the sum is greater than a constant term, the three vertices are assigned to the same partition.

In [51], the authors propose a multistart algorithm called the dynamic convexized (DC) method. In DC a local search algorithm is applied in a random initial solution using a dynamically updated auxiliary function.

The multiple operator heuristic (MOH) for max-k-cut [33] is an iterative method that starts from a random solution and applies five operators organized into three search phases. The first phase, descent-based improvement, finds a local optimum using two intensification-oriented operators. In the second phase, the solution is diversified via a tabu search method [21] with two operators. In the third phase, a random perturbation is applied to the incumbent solution. The tests show that MOH provides better bounds in less time than DC in 90% of the tested instances.

The variable neighborhood search (VNS) [37] metaheuristic is an iterative method that seeks an improved solution in a distant neighborhood of the initial random feasible solution. In addition, a local search is applied to find a new local optimum. The greedy randomized adaptive search procedure (GRASP) [15] is iterative and has three stages: Construction (using a random greedy algorithm), local search, and solution analysis. VNS and GRASP have been successfully applied to the max-cut problem [16].

2.2.3 Branching rules

Many researchers have investigated variable selection strategies in the branch-and-bound framework, e.g., [1, 32]. A strategy that has attracted considerable interest is *strong branching* [19, 31]. This method tests all the fractional candidates to find the one that gives the best improvement. Another popular approach is *pseudo-cost branching* [6], which uses information on previous changes to choose the next variable. *Reliability branching* [1] is a generalization of strong and pseudo-cost branching; the results of [1] show that this rule is better than its predecessors.

In [3], the authors investigate four branching rules for max-k-cut. In the first rule, the variable selected corresponds to the edge that is closest to 0 or 1. The more elaborate second rule selects the edge of vertices i' and j' such that

$$i' \in \operatorname*{argmin}_{i \in V} \sum_{r \neq i, r=1}^{n} (0.5 - |\hat{x}_{ir} - 0.5|)^2, \tag{26}$$

$$j' \in \operatorname*{argmin}_{j \in V, j \neq i'} \sum_{r \neq j, r=1}^{n} (0.5 - |\hat{x}_{jr} - 0.5|)^2 \tag{27}$$

where $\hat{x}$ is the optimal (upper bound) solution of the current node. In the third rule, the variable selected corresponds to the edige that is closest to 0.5. The fourth rule is similar to the second except that for vertex j', argmin is replaced by argmax. The results show that the second rule performs best and the first rule gives the worst results.

2.2.4 Node selection

Node selection is the classical task of deciding how to select the next node of the branch-and-bound tree to explore. Usually the strategy depends on the number of explored nodes in the tree or the memory capacity of the computer. Best-first, depth-first, and breadth-first are among the most commonly used strategies; for other variants see [32, 38].

2.3 Related methods approaches

In [35, 36], the author considered the realignment of a football league, applying a branch-and-cut algorithm based on the LP formulation of the k-way equipartition problem ($k \in \{4, 8\}$ and $|V| = 32$). In [35], for graphs with 100 to 500 vertices, the problem is solved with a gap of less than 2.5% for $k = 4$.

A branch-and-bound framework based on the edge formulation of max-k-cut is studied in [48]. The authors show that in a chordal graph, the LP and SDP formulations can be defined with $|E|$ variables rather than $\frac{n(n-1)}{2}$. In a few seconds, they solve to optimality some sparse graphs with $n = 200$ and $k \in \{3, 4\}$.

The branch-and-cut algorithm based on the vertex-and-edge LP formulation of max-k-cut is applied in [14, 27]. In both cases, the LP relaxation is tightened by general clique and triangle inequalities. In [14], the authors apply branch-and-bound to a two-level graph partitioning problem in mobile wireless communications; they solve problems with $n = 100$ and $k \in \{2, 3, 4\}$ for sparse graphs. In [27], the authors study the connected max-k-cut problem for applications in gas and power networks. They apply a modified version of the combinatorial Constraints (2), (3), (22)–(24) to reinforce their LP relaxation of max-k-cut.

In [20], the authors design a branch-and-cut algorithm based on the SDP formulation (called SBC) of the minimum k-partition problem. SBC includes triangle and clique inequalities and uses a dynamic version of the bundle method [26] to solve the SDP relaxation. SBC is able to compute optimal solutions for dense graphs with 60 vertices and for sparse graphs with 100 vertices.

The state-of-the-art BundleBC algorithm [3] is an improvement of SBC that extends the ideas of the `Biq Mac` solver [42] to max-k-cut. In [3], the separation of clique inequalities and rules for choosing the variables are investigated. The results show that the use of clique inequalities reduces the number of subproblems analyzed in the branch-and-bound tree and that BundleBC is faster than SBC.

3 Computational study of a branch-and-bound framework

This section presents a computational study of the main aspects of the bounding procedure, it investigates strategies for selecting variables, and techniques for exploring the branch-and-bound tree.

3.1 Outline

The following is a brief summary of some components of branch-and-bound that are studied in detail in the next sections.

1. *Upper bound.* The upper bound (z_{ub}) is obtained by a solution of the max-k-cut relaxation. It is important to apply a method that gives a good trade-off between the execution time and the quality of the bounds. Section 3.4 investigates the SDP and LP relaxations. The inclusion of a CPA in the branch-and-bound tree is evaluated in Section 3.5.

2. *Feasible solutions as lower bounds.* Feasible solutions (z_{lb}) that can be found quickly and have a value close to optimality can speed up the branch-and-bound method by pruning branches. Section 3.6 analyzes the following four heuristic methods:

- Iterative clustering heuristic (ICH) [20],
- Multiple operator heuristic (MOH) [33],
- Variable neighborhood search (VNS) [23, 37], and
- Greedy randomized adaptive search procedure (GRASP) [15].

3. *Splitting problem.* An important factor that impacts the effectiveness of branch-and-bound is the strategy used to partition the feasible region [18]. For max-k-cut, the branching can be applied to a vertex or an edge of the graph. Hence, we study the following two types of splitting in Section 3.7:
   - *Dichotomic*, where the branching variables correspond to an edge $\{i, j\} \in E$ of the graph and each node is split into two child nodes: In the first, the edge $\{i, j\}$ is *cut*, and in the second, it is *not-cut*, i.e., vertices i and j must be in the same partition,
   - *Polychotomic* or k-chotomic: A vertex $i \in V$ is assigned to a different partition in each child node. Max-k-cut problems can have at most k partitions, so each node can generate k children.
4. *Branching rule.* Commonly a variable with a fractional value is selected for branching. Section 3.8 investigates five techniques (rules) for selecting this variable.
5. *Node selection.* At each branch-and-bound iteration, a node is selected from a set of *active* (unexplored) nodes. This procedure is also known as the exploration tree strategy. We study three selection strategies in Section 3.9.

3.2 Computational settings

We now describe the computational environment, the comparison procedure, and the instances.

Computational environment The tests are performed on a Linux PC with two Intel® Xeon® 3.07 GHz processors. The `mosek` 8.1 solver [4] is used for both the SDP and LP relaxations.

3.2.1 Comparison of results

We analyze the results of the branch-and-bound algorithm with a modified version of *performance profiles* [11] that show the proportion of problems that are solved to optimality (y-axis) within a given time (x-axis). Hence, these profiles show the temporal progression of the methods. Moreover, we use a *performance table* to measure the performance of the methods for generating feasible solutions.

3.2.2 Set of instances

We use some of the most difficult instances from the literature, especially from the BundleBC [3] and `Biq Mac` [49] libraries. We also generated random instances using the `rudy` graph generator [43]. The instances are divided into:

Set A: This set has two classes with a total of 65 instances. The first class consists of 40 of the most difficult instances from [3]. Some of these instances are complete graphs, and others come from applications in physics to determine the energy-minimum states of so-called Potts glasses. The second class was generated by `rudy`; the instances have 20 to 50 vertices ($|V|$) and the edges are chosen randomly in such a way that the graphs have edge densities of 25% and 50%.

Set B: This set has some of the largest instances from [49], namely:

- bqp: ten weighted graphs with dimension 100, density 0.1, and edge weights in $\{-100, 0, 100\}$;
- g05: ten unweighted graphs with edge probability 0.5 and dimension 100;
- pm1d: ten weighted graphs with edge weights in $\{-1, 0, 1\}$, density 0.99, and dimension 100;
- ising2: nine one-dimensional Ising chains with dimension $|V| \in \{200, 250, 300\}$;
- be: ten Billionnet and Elloumi instances with density $d = 0.8$, edge weights in $\{-50, 50\}$, and dimension 200.

3.3 Standard parameters of proposed branch-and-bound framework

In the following sections, each component of the branch-and-bound algorithm is studied separately, with the other procedures fixed to their standard values. For the bound procedures, the standard approaches are: The *Edge-EIG* formulation (see Section 3.4), the VNS metaheuristic to find feasible solutions, and the branch-and-cut that applies a CPA at every node (`BC-all`). The edge-weight rule (R4 in Section 3.8) and depth-first search (DFS) are the standard methods for branching and selection, respectively, and k-chotomic is the standard strategy for splitting a node.

3.4 Computing upper bounds

This section presents the computational results of the branch-and-bound algorithm for edge-only, vertex-and-edge, and vertex-only relaxations.

For the LP relaxations, we investigate the performance of the SDP-based constraints since it has been shown in [45] that these constraints are strong but expensive. In this section we consider five methods:

- *SDP*: Applies the SDP relaxation reinforced with the combinatorial Inequalities (2), (3), (22)–(24);
- *Edge*: Uses the edge-only relaxation of Section 2.2.1, reinforced with the combinatorial inequalities;
- *Edge-EIG*: *Edge* with the addition of the SDP-based Inequalities (25) proposed in [44];
- *NoEd*: Uses the vertex-and-edge relaxation of Section 2.2.1 reinforced with the combinatorial inequalities;
- *NoEd-EIG*: *NoEd* with the addition of the SDP-based Inequalities (25).

Figure 2 shows the results for Set A and $k \in \{3, 5, 7, 10\}$. SDP is the most efficient method when $k = 3$, and *Edge-EIG* is the best when $k > 3$. The strength of SDP for small values of k

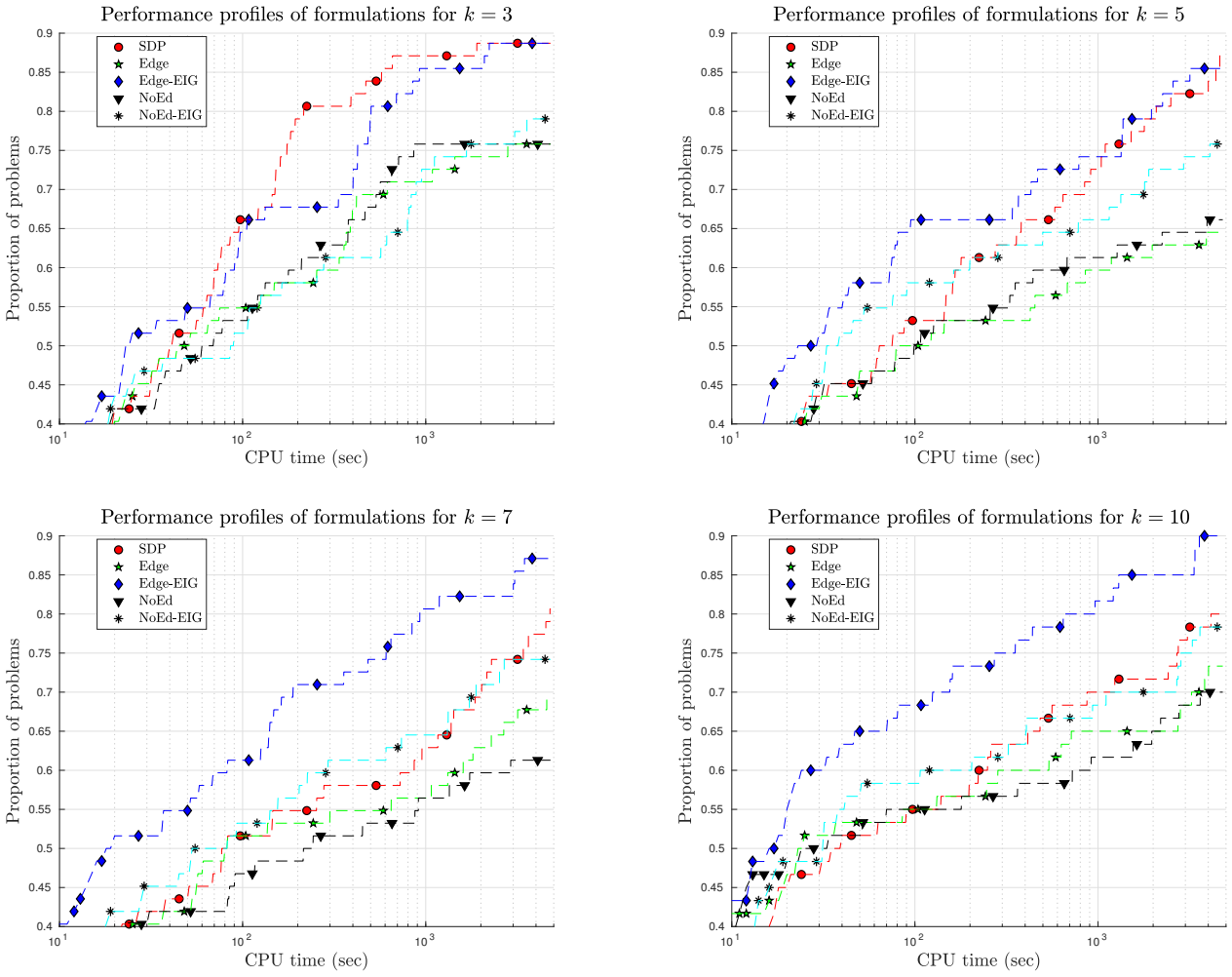

Figure 2: Performance profiles of relaxations for the instances of Set A and $k \in \{3, 5, 7, 10\}$.

can be explained by the fact that only a few bound Constraints (20) needed to be separated until feasibility. For $k \in \{7, 10\}$, *Edge-EIG* solves more than 70% of the instances within 200s while the other methods take at least $1,000$s to achieve this. Moreover, the results indicate that the SDP-based Inequalities (25) perform well in LP formulations: *Edge-EIG* and *NoEd-EIG* obtained better solutions than *Edge* and *NoEd*, respectively. The performance of *Edge* and *NoEd* is similar, but *Edge-EIG* is much better than *NoEd-EIG*.

3.5 Cutting plane algorithm

CPA is an iterative method that solves a relaxation and then finds and adds to the relaxation some of the violated inequalities (or cutting planes). We apply the CPA proposed in [45] that implements the early-termination technique in an interior point method. In [45], at each iteration of the CPA, at most $2|V|$ of the most violated constraints are added and the resulting new problem is solved from scratch.

In [45], a CPA stops after it cannot find violated constraints. We incorporate two additional

stopping criteria: *iteration time* (it_time) and *iteration improvement* (it_impr).

The time to solve a subproblem is limited to 10s (it_time = 10s); when this limit is reached the last bound obtained is returned. The largest CPA improvements occur in the early iterations [45], so we stop the algorithm when the improvement is less than 10^{-4}. Let z_i be the value of the relaxation solution at the current CPA iteration i. The improvement, it_impr, is calculated as

$$\text{it\_impr} = \frac{z_{i-1} - z_i}{z_i}. \tag{28}$$

Since a CPA is stopped prematurely, the upper bounds obtained are not as strong as those observed in [45], but the branch-and-bound can explore more nodes.

We analyze the following strategies:

- `cut-BB`: Applies a CPA only at the root node (the cut-and-branch method).
- `BC-i` ($i \in \{2, 7, 21\}$): Executes a CPA at the root node and when the level (l) of a node in the tree is a multiple of i. The level of a node is its distance from the root node of the search tree. The `BC-2` strategy applies CPA in a level and skip the next. The choice of level $i \in \{7, 21\}$ was empirical to test more distant executions of CPA.
- `BC-all`: branch-and-cut method that executes a CPA at every node.

Figure 3 shows the results for the five methods of this section for Set A and $k \in \{3, 5, 7, 10\}$. We see that `BC-all` is the most efficient method, especially for $k \in \{3, 5, 7\}$, followed by `BC-2`. For instance, for $k \in \{3, 5\}$, `BC-all` and `BC-2` solve 60% of the instances within 200s while `BC-21` and `cut-BB` take at least $1,000$s to achieve this.

3.6 Computing lower bounds

This section presents the implementation details of the four heuristics discussed in Section 2.2.2: ICH, MOH, VNS, and GRASP. To obtain a good trade-off between quality and CPU time, some parameters are different from those in the literature.

In a feasible solution of max-k-cut, each vertex $v \in V$ is assigned to exactly one partition. Two solutions a and b are neighbors if the distance between them is one ($d(a, b) = 1$), i.e., if and only if one vertex of a is assigned to a different partition than the assignment in b.

A local search that uses simple transfer will move, at each iteration, to a neighbor with a better solution. If no neighbor has a better solution, then the current solution is a local maximum. In double transfer, the best neighbors with a distance of 2 are considered. Below we describe the parameters and implementation of each heuristic.

1. For ICH, that exploit a fractional primal solution, the constant term tol that decides if three nodes should be in the same partition is set to $tol = 1.7$ (as proposed in [20]), and the time of each iteration of the ICH is limited to 1s.

2. For MOH, in the improvement phase (second stage), only a simple transfer (called O_3 in [33]) is considered since operations of high transfers are quite expensive. In addition, the tabu size λ, that is the number of stored previous solutions to avoid repetition, is set to $\lambda = 10$. The perturbation strength γ that measures the perturbation allowed in the third phase of the method is set to $\gamma = 50\%$.

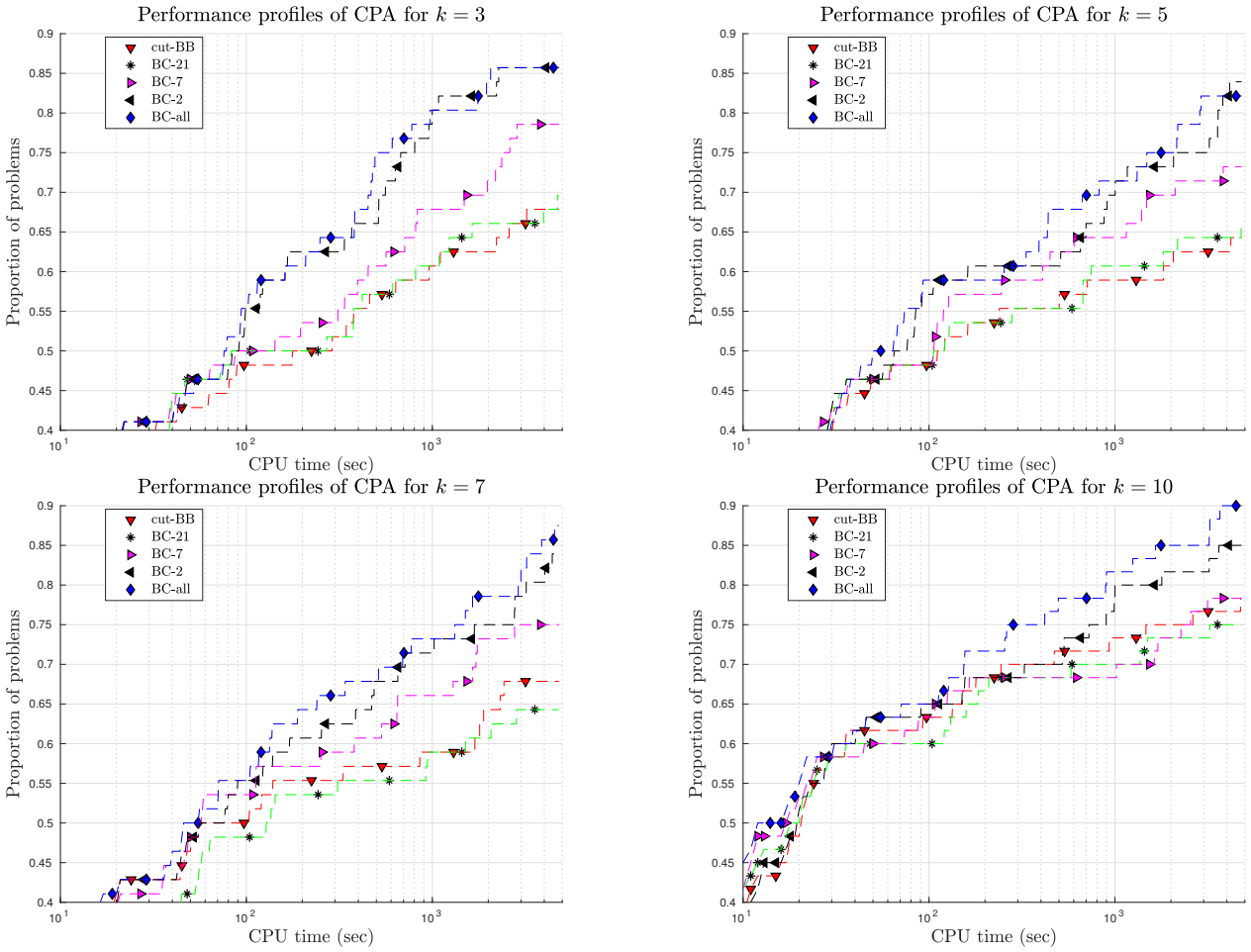

Figure 3: Performance profiles of CPA within branch-and-bound for the instances of Set A and $k \in \{3, 5, 7, 10\}$.

3. VNS explores a distant neighborhood from a local maximum a^*. We consider a solution b to be a neighbor solution of a^* if their distance $d(a^*, b)$ is at most 12 so it can explore more solutions in the local search phase.

4. GRASP applies a local search that uses single- and double-transfer moves to find the best neighbor solutions of a random feasible solution obtained from the construction phase.

To reduce the total running time, the heuristics are stopped after 2s, and they are not executed at every node of the tree. They are executed at the root node and at every 15 iterations of the branch-and-bound algorithm, i.e. at 15 sub-problems processed; this choice worked well in our experiments.

Table 1 shows the results of the four heuristics for $k \in \{3, 5, 7, 10\}$. The first columns show the name and number of partitions k of each set of instances. The remaining columns give the average gap and the time of execution in seconds for each method.

The *gap* is calculated from the best known feasible solution (Blb_i) of Instance i. Therefore, the

percentage gap of Instance i of Method m is

$$gap(\%)_{i,m} = 100 \times \frac{Blb_i - lb_{i,m}}{Blb_i} \tag{29}$$

where $lb_{i,m}$ is the average value of ten executions of Method m on Instance i. The $gap(\%)$ presented in Table 1 is the mean value over all instances in Sets A and B. The best results are highlighted. We see that on average the gap of all four methods is below 2.3% and that the VNS metaheuristic gives, for almost all the tests, the best bound with a competitive CPU time. VNS has an average gap of 0.5% for the tested instances.

Table 1: Results for ICH, MOH, VNS, and GRASP heuristics: The best gap found is highlighted.

Instance		ICH		MOH		VNS		GRASP	
Set	k	$gap(\%)$	time (s)	$gap(\%)$	time (s)	$gap(\%)$	time (s)	$gap(\%)$	time (s)
A	3	0.67	8.1	**0.24**	2.0	0.33	1.2	0.54	1.7
B	3	1.27	25.1	1.08	2.0	**0.80**	2.1	1.36	2.0
A	5	0.96	13.2	0.53	1.9	**0.21**	1.3	0.77	1.7
B	5	0.99	31.8	1.32	2.0	**0.89**	2.1	1.64	2.0
A	7	0.55	7.4	0.85	1.9	**0.33**	1.4	0.91	1.7
B	7	1.04	30.5	1.77	2.0	**0.83**	2.1	1.80	2.0
A	10	0.59	7.30	0.94	1.9	**0.22**	1.3	1.30	1.6
B	10	1.12	30.5	1.84	2.0	**0.42**	2.1	2.30	2.0

3.7 Splitting strategies

The branch-and-bound algorithm converges if the solution space of each subproblem is contained in the solution space of the original problem and if the original problem has finite solutions. Typically, the subproblems generated at each branch (or node) are disjoint, thereby avoiding the occurrence of the same solution in different subspaces.

Dichotomic split. For max-k-cut, all the existing exact methods apply the split to an edge $\{i, j\}$, so they impose $x_{ij} = \lfloor \hat{x}_{ij} \rfloor = 0$ in one subproblem and $x_{ij} = \lceil \hat{x}_{ij} \rceil = 1$ in the other. Dichotomic branching can easily be implemented. However, the dichotomic strategy can generate subproblems that can violate valid inequalities not presented in the relaxed solution. Therefore, the branch-and-bound can waste memory and time by storing and evaluating a node that is not feasible.

k-chotomic split. We propose a vertex branching similar to that applied in max-cut [42] that will always generate valid subproblems. In this strategy, for each subproblem i, the selected vertex $v \in V$ is assigned to a different partition $P_i \in \{P_1, P_2, \ldots, P_k\}$. Thus, a node of the tree is split into k child nodes. This polychotomic branching is called a k-chotomic split. One drawback is that the number of unexplored nodes in the tree can grow quickly, and this can cause memory issues. However, a combination of this strategy with depth-first search (see Section 3.9) can mitigate this drawback.

Fixing extra variables. For both strategies, in the LP formulation, the reduced cost of the relaxed solution is used to fix variables. Moreover, for the dichotomic split, when a variable is selected it is possible to fix more variables by analyzing the triangle and clique inequalities. For the former, if an edge x_{ih} has already been fixed (in a related node), and the chosen variable at the current iteration is x_{hj}, then x_{ij} is fixed if:

$$x_{ih} = 1 \quad \& \quad x_{hj} = 1 \quad \Rightarrow \quad x_{ij} = 1, \tag{30}$$
$$x_{ih} = 1 \quad \& \quad x_{hj} = 0 \quad \Rightarrow \quad x_{ij} = 0, \tag{31}$$
$$x_{ih} = 0 \quad \& \quad x_{hj} = 1 \quad \Rightarrow \quad x_{ij} = 0. \tag{32}$$

To avoid solutions that may violate inequalities not present in the relaxed formulation, it is necessary to certify that all the clique inequalities are also satisfied at each subproblem. However, as observed in [44], it is computationally expensive to enumerate all the cliques even for small instances.

For the k-chotomic strategy, any vertex can initially be assigned to the first partition (P_1) because all the vertices must be in a partition and all the partitions are similar. We select the vertex $v \in V$ with the largest sum of incident edges weight in the input graph ($v \in \operatorname{argmax}_{v \in V} \sum_{j=1}^{n} w_{vj}$).

Figure 4 shows the results of the dichotomic and k-chotomic splits for Set A and $k \in \{3, 5, 7, 10\}$. For $k = 3$, the results show that the k-chotomic split is more efficient: It can solve 85% of the instances in $1,000$s, whereas the dichotomic split can solve less than 75% in the same time. For $k \in \{5, 7\}$, the dichotomic split has slightly better performance for instances that can be solved in under 300s, but for the most expensive instances the k-chotomic split can solve 5% more instances. For $k = 10$, the two methods have similar performance.

3.8 Branching rules

The variable and node selections are both critical decisions that impact performance. Branching on a variable that does not give any significant improvement in any of the child nodes can lead to an extremely large and expensive search tree.

This section investigates the most successful existing rules, e.g., the reliability branching of [1] and the best rules of [3]. Additionally, we propose an edge-weight rule and investigate the following rules:

- R1. We branch on the most decided variables, i.e. variables that are the closest to 0 or 1 in the relaxed solution.
- R2. This is the third rule of [3]: It chooses the variable that is the least decided, the one the is the closest to 0.5.
- R3. This is the second rule of [3] presented in Section 2.2.3.
- R4. This strategy branches on the variable with the largest weight. For k-chotomic split, it selects the variable of the corresponding vertex with the largest sum of incident edges weight.

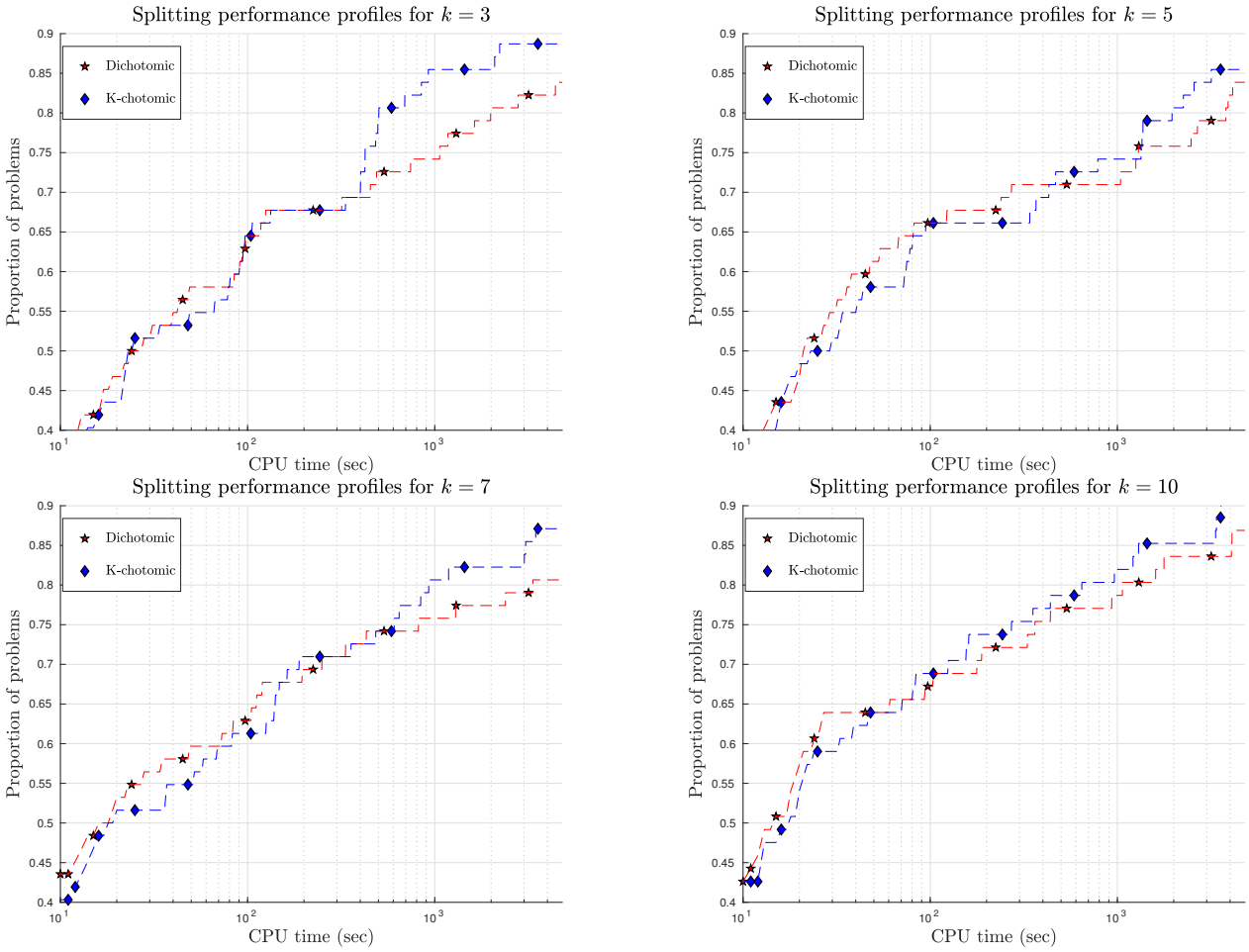

Figure 4: Performance profiles of dichotomic and k-chotomic strategies for the instances of Set A and $k \in \{3, 5, 7, 10\}$.

- R5. This is an adaptation of the *reliability branching* of [1]. It uses the pseudo-cost score of a variable $\hat{x}_{ij}$. The pseudo-cost is estimated by summing all the improvements of previous branches in which the variable was selected, and if the number of times that $\hat{x}_{ij}$ was selected is less than a reliability constant $(\eta = 4)$, the rule uses the strong branching strategy that tests the improvement of a variable by simulating a branching.

For k-chotomic split and for rules R1, R2, R3 or R5, we impose that one of the vertices i or j of the selected variable $\hat{x}_{ij}$ must already be assigned to a partition in a related previous node of the branch-and-bound search tree.

Figure 5 shows the results for the five rules for Set A and $k \in \{3, 5, 7, 10\}$. For $k = 3$, R4 has the best performance: It solves 87% of the instances while the others can solve at most 83% in 1 hour. For $k \in \{5, 7, 10\}$, R2 and R4 have the best results, especially for the most difficult instances.

3.9 Node selection

Node selection is the task of deciding how to select the next node of the tree to explore. We consider three strategies:

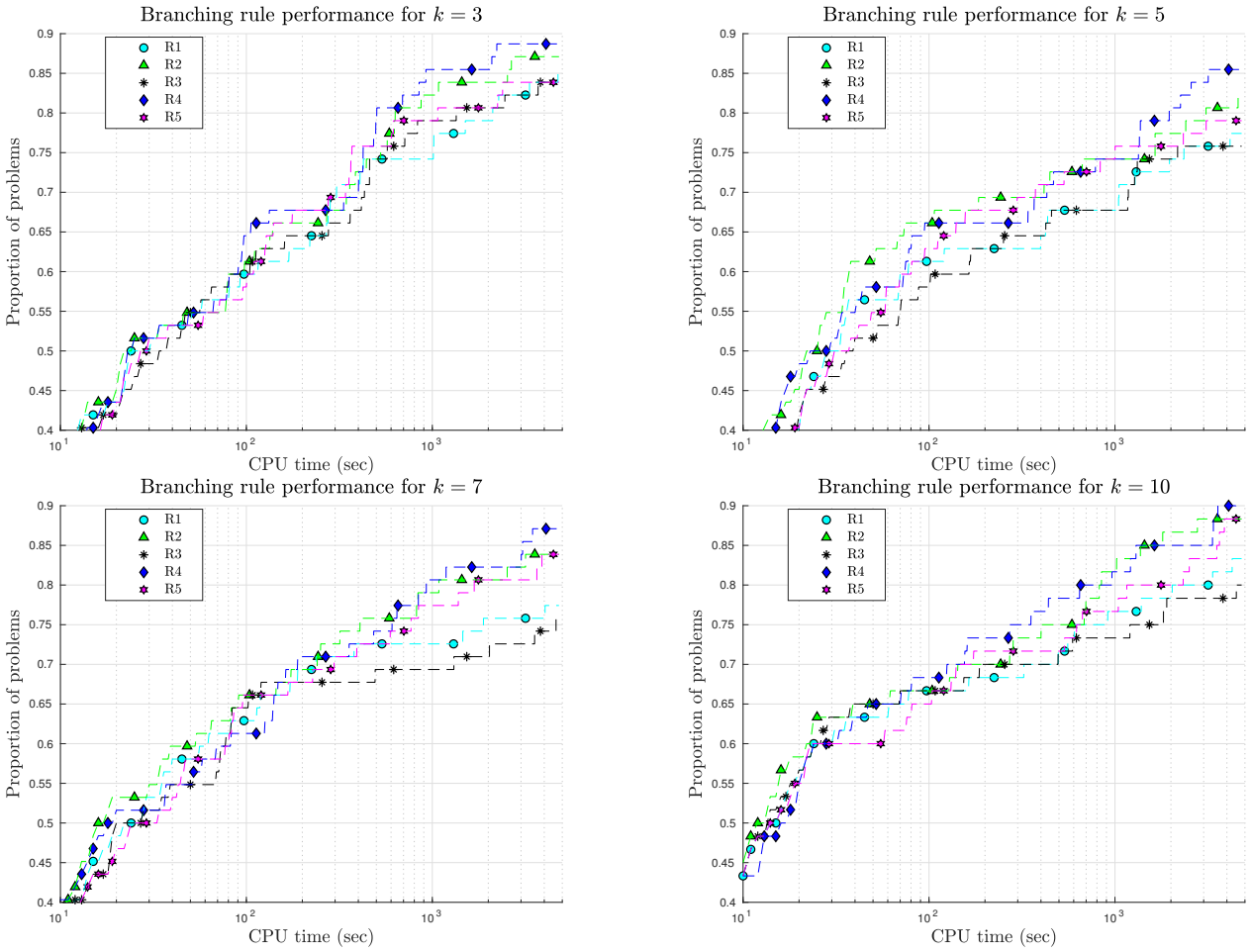

Figure 5: Performance profiles of five branching rules for the instances of Set A and $k \in \{3, 5, 7, 10\}$.

- *Best-first search (BeFS):* BeFS selects the best active node, i.e., the node with the largest upper bound. Usually, BeFS explores fewer nodes than the other strategies but it maintains a larger tree in terms of memory.

- *Breadth-first search (BrFS):* BrFS is implemented with a first-in-first-out data structure. BrFS performs well on unbalanced search trees and finds solutions that are close to the root node. However, as for BeFS, the memory requirements can be high, and the performance of this approach depends on the heuristics used to find good feasible solutions.

- *Depth-first search (DFS):* DFS strategy uses a last-in-first-out data structure. The method goes deep into the search tree and starts backtracking only when a node is pruned. It can easily be implemented, the memory requirements are low, and it is likely to finds feasible solutions faster than the other strategies. However, it is sensitive to the branching rule for the first nodes in the tree.

Figure 6 shows the results for the three strategies for Set A and $k \in \{3, 5, 7, 10\}$. We see that DFS has slightly better performance, especially for $k = 7$.

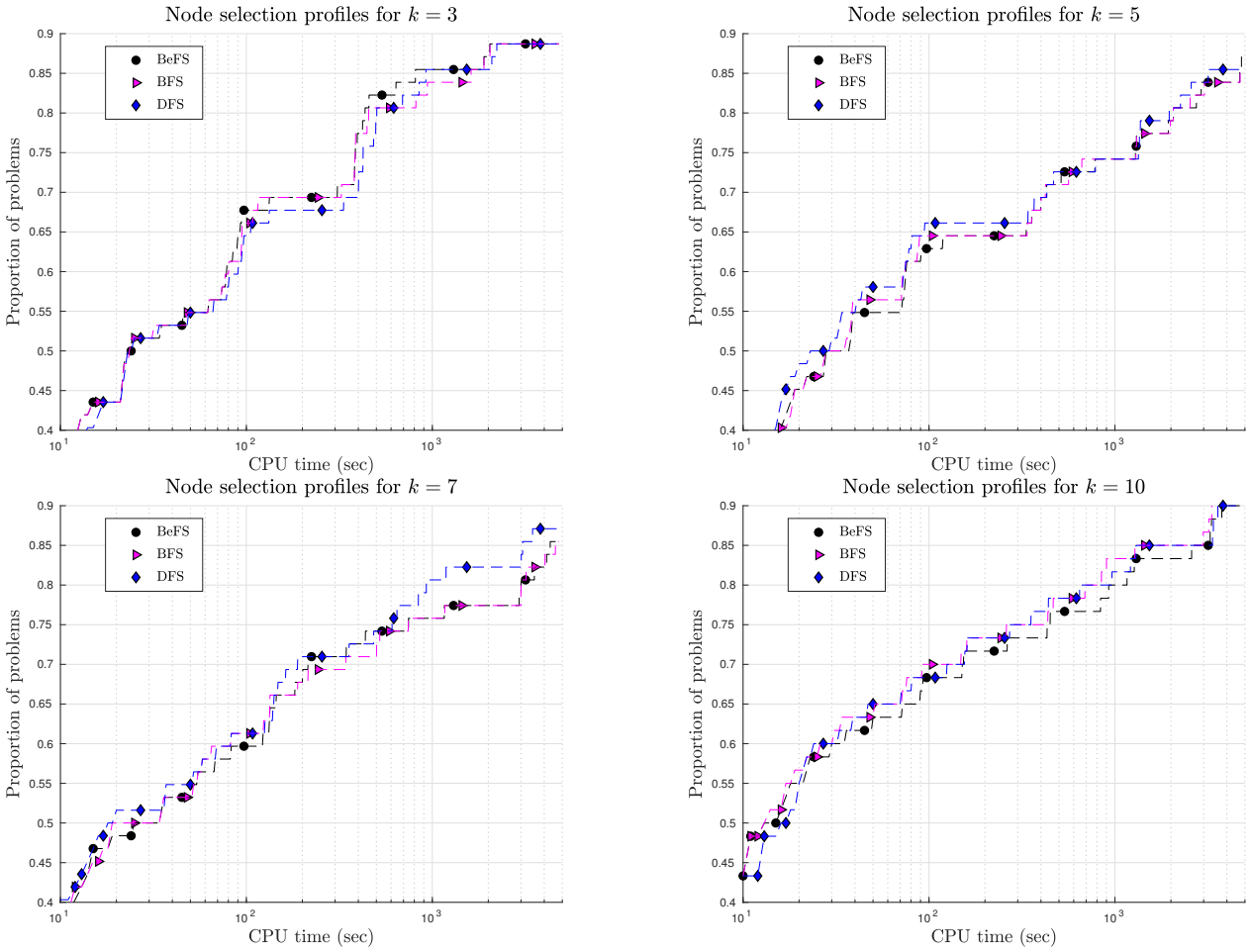

Figure 6: Performance profiles of BeFS, BrFS, and DFS for the instances of Set A and $k \in \{3, 5, 7, 10\}$.

4 Comparison with BundleBC

We now carry out branch-and-bound for the *Edge-EIG* and *SDP* formulations and compare their performance with the BundleBC solver. We use the following settings:

- *Type of branch-and-bound*: `BC-all` is used to find the optimal solutions;
- *feasible solution*: The VNS metaheuristic is applied;
- *branching rule*: Rule R4 is used;
- *node selection*: DFS is used;
- *splitting problem*: The k-chotomic strategy is used.

BundleBC uses a cutting plane on the triangle and clique inequalities. The methods all terminate after 1.5h, they use only one CPU thread, and the tests are executed on the same machine. Figure 7 shows the performance profiles of *SDP*, *Edge-EIG*, and BundleBC for $k \in \{3, 5, 7, 10\}$. We see that BundleBC is the most efficient method for $k = 3$. However, for $k \geq 5$ *Edge-EIG*, which is the LP edge-only formulation reinforced with combinatorial and SDP-based inequalities, obtains the best results, especially for $k \geq 7$. For example, for $k = 7$ *Edge-EIG* can solve 60% of the problems in less than 100s while *SDP* and BundleBC take almost $1,000$s to achieve this.

Table 2 shows the optimal solution value, the number of subproblems (# nodes) of the branch-and-bound, and the running times for *Edge-EIG* and *SDP* for the instances that BundleBC could not solve to optimality within the time limit. The results show that *Edge-EIG* is faster than *SDP* for the instances that BundleBC could not solve, and for 30% of the instances *Edge-EIG* can solve the problem to optimality in the root node.

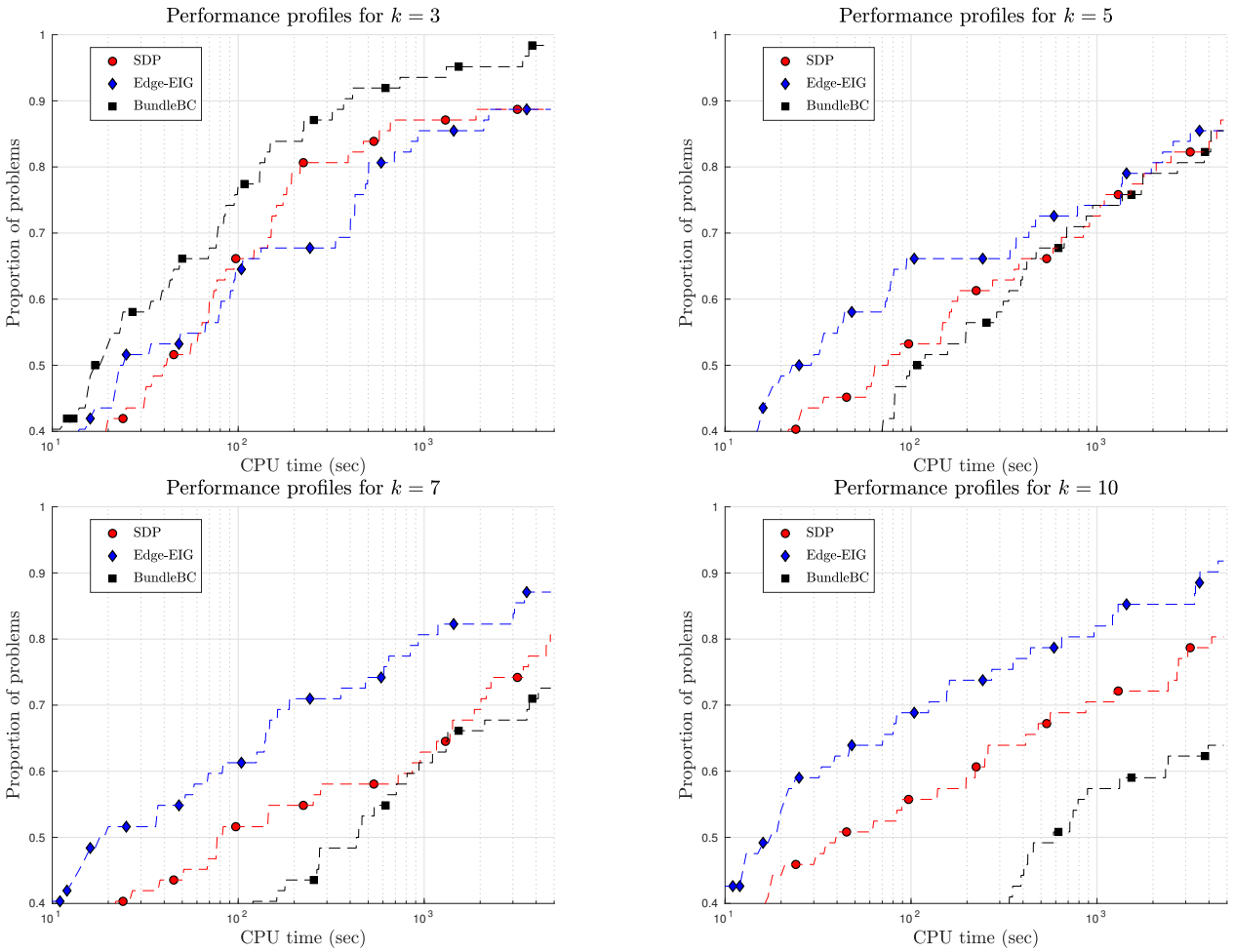

Figure 7: Performance profiles of *SDP*, *Edge-EIG*, and BundleBC for the instances of set A and $k \in \{3, 5, 7, 10\}$.

5 Discussion

We have investigated some of the most important features of the branch-and-bound method for max-k-cut. For the upper bound, we explored the SDP and LP relaxation and the inclusion of the CPA in the branch-and-bound tree. For the lower bound, we explored four heuristics: ICH, MOH, VNS, and GRASP. For the split strategy, we considered the dichotomic split (two subproblems), and the k-chotomic split (k subproblems). We studied five rules for choosing the next variable to branch on and three strategies for node selection: BeFS, DFS, and BrFS.

Our results are as follows:

Table 2: Solutions found for instances from [3]. Instances that could not be solved to optimality within the time limit are indicated by "−".

Instance			*SDP*		*Edge-EIG*	
Name	k	Optimal value	#nodes	time (s)	#nodes	time (s)
data_2g_10_1001	5	7,011,340	44	844.3	0	22.3
data_2g_10_1001	7	7,011,340	–	–	760	162.3
data_2g_8_37	7	4,785,210	663	4,769.0	920	147.7
data_2g_8_37	10	4,785,210	–	–	380	70.1
data_2g_8_648	7	4,733,950	2	50.3	0	12.0
data_2g_8_648	10	4,733,950	2	62.9	2	17.9
data_2g_9_819	10	5,416,010	17	259.9	0	21.6
data_2g_9_9211	7	5,878,090	–	–	6,280	929.0
data_2g_9_9211	10	5,878,090	–	–	24,380	3,576.0
data_clique_60	10	35,640	–	–	0	19.3
data_clique_70	10	56,595	–	–	212	124.1
data_random_30_k=2	10	2,135	74	197.6	1,726	160.4
data_random_30_k=3	10	2,151	2	17.6	209	32.7
data_random_40_k=2	10	3,821	79	560.2	5,202	963.6

1. *Formulations.* *Edge-EIG* and *SDP* are the most efficient methods. While semidefinite-based methods (*SDP* and BundleBC) have the best results for $k = 3$, *Edge-EIG* performs better for $k \geq 5$.

2. *branch-and-bound with CPA.* The strategy that includes a CPA at all the nodes of the tree is the most efficient.

3. *Feasible solutions.* VNS has the best performance.

4. *Splitting.* The k-chotomic split performs better, especially for the most difficult instances.

5. *Branching rules.* The rule that selects variables with the largest weights (R4) gives the best results.

6. *Node selection.* DFS has slightly better performance for the most difficult instances.

7. *Comparison with BundleBC.* BundleBC is the most efficient method for small values of k, and *Edge-EIG* performs best when $k \geq 5$.

8. *Sparsity.* While we did not carry out a study of the impact of sparsity of the instances, we observed that our method and the BundleBC were able to solve only instances corresponding to sparse graphs (edge density below 10%), and that none of the methods was able to solve the instances in SET B to optimality.

In future research, it would be interesting to design learning methods to select the "right" combination of branch-and-bound components for any instance and to guide the selection of violated inequalities in a CPA, especially for the SDP-based constraints. Second, it is likely that sparsity plays a role in the performance of the different formulations on specific classes of instances. However, understanding the importance of sparsity for SDP-based approaches requires studying how to exploit chordal extensions for the various classes of tested instances.

References

[1] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42 – 54, 2005.

[2] Z. Ales and A. Knippel. An extended edge-representative formulation for the k-partitioning problem. *Electronic Notes in Discrete Mathematics*, 52(Supplement C):333–342, 2016.

[3] M. F. Anjos, B. Ghaddar, L. Hupp, F. Liers, and A. Wiegele. Solving k-way graph partitioning problems to optimality: The impact of semidefinite relaxations and the bundle method. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization*, pages 355–386. Springer Berlin Heidelberg, 2013.

[4] Mosek ApS. mosek. http://www.mosek.com, 2019.

[5] F. Barahona, M. Grötschel, M. Jünger, and G. Reinelt. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research*, 36(3):493–513, 1988.

[6] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971.

[7] S. Chopra and M. R. Rao. The partition problem. *Mathematical Programming*, 59(1):87–115, 1993.

[8] S. Chopra and M.R. Rao. Facets of the k-partition polytope. *Discrete Applied Mathematics*, 61(1):27–48, 1995.

[9] H. Crowder, E. L. Johnson, and M. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31(5):803–834, 1983.

[10] E. de Klerk, D. V. Pasechnik, and J. P. Warners. On approximate graph colouring and max-k-cut algorithms based on the θ-function. *Journal of Combinatorial Optimization*, 8(3):267–294, 2004.

[11] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[12] A. Eisenblätter. The semidefinite relaxation of the k-partition polytope is strong. In *Integer Programming and Combinatorial Optimization*, volume 2337 of *Lecture Notes in Computer Science*, pages 273–290. Springer Berlin Heidelberg, 2002.

[13] J. Fairbrother and A. N. Letchford. Projection results for the k-partition problem. *Discrete Optimization*, 26:97–111, 2017.

[14] J. Fairbrother, A. N. Letchford, and K. Briggs. A two-level graph partitioning problem arising in mobile wireless communications. *Computational Optimization and Applications*, 69(3):653–676, 2018.

[15] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.

[16] P. Festa, P. M. Pardalos, M. G. C. Resende, and C. C. Ribeiro. Randomized heuristics for the max-cut problem. *Optimization Methods and Software*, 17(6):1033–1058, 2002.

[17] A. Frieze and M. Jerrum. Improved approximation algorithms for max k-cut and max bisection. *Algorithmica*, 18(1):67–81, 1997.

[18] G. Gamrath. Improving strong branching by propagation. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 347–354. Springer Berlin Heidelberg, 2013.

[19] G. Gamrath. Improving strong branching by domain propagation. *EURO Journal on Computational Optimization*, 2(3):99–122, 2014.

[20] B. Ghaddar, M. F. Anjos, and F. Liers. A branch-and-cut algorithm based on semidefinite programming for the minimum k-partition problem. *Annals of Operations Research*, 188(1):155–174, 2011.

[21] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[22] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[23] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449–467, 2001.

[24] P. Heggernes. Minimal triangulations of graphs: A survey. *Discrete Mathematics*, 306(3):297–317, 2006.

[25] C. Helmberg. *Semidefinite Programming for Combinatorial Optimization*. Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 2000.

[26] C. Helmberg and F. Rendl. A spectral bundle method for semidefinite programming. *SIAM Journal on Optimization*, 10(3):673–696, 2000.

[27] C. Hojny, I Joormann, H Lüthen, and M. Schmidt. Mixed-integer programming techniques for the connected max-k-cut problem. *Mathematical Programming Computation*, 2020. DOI 10.1007/s12532-020-00186-3.

[28] D. Karger, R. Motwani, and M. Sudan. Approximate graph coloring by semidefinite programming. *Journal of the ACM*, 45(2):246–265, 1998.

[29] Y.-H. Kim, Y. Yoon, and Z. W. Geem. A comparison study of harmony search and genetic algorithm for the max-cut problem. *Swarm and Evolutionary Computation*, 44:130–135, 2019.

[30] N. Krislock, J. Malick, and F. Roupin. Improved semidefinite bounding procedure for solving max-cut problems to optimality. *Mathematical Programming*, 143(1):61–86, 2012.

[31] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.

[32] A. Lodi and G. Zarpellon. On learning and branching: A survey. *TOP*, 25(2):207–236, 2017.

[33] F. Ma and J.-K. Hao. A multiple search operator heuristic for the max-k-cut problem. *Annals of Operations Research*, 248(1):365–403, 2017.

[34] I. Maros and G. Mitra. Strategies for creating advanced bases for large-scale linear programming problems. *INFORMS Journal on Computing*, 10(2):248–260, 1998.

[35] J. E. Mitchell. Branch-and-cut for the k-way equipartition problem, 2001. http://www.optimization-online.org/DB_HTML/2001/02/288.html.

[36] J. E. Mitchell. Realignment in the national football league: Did they do it right? *Naval Research Logistics*, 50(7):683–701, 2003.

[37] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11):1097–1100, 1997.

[38] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016.

[39] A. Newman. Complex semidefinite programming and max-k-cut, 2018. http://arxiv.org/abs/1812.10770.

[40] C. Niu, Y. Li, R. Q. Hu, and F. Ye. Femtocell-enhanced multi-target spectrum allocation strategy in LTE-A HetNets. *IET Communications*, 11(6):887–896, 2017.

[41] C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991.

[42] F. Rendl, G. Rinaldi, and A. Wiegele. Solving max-cut to optimality by intersecting semidefinite and polyhedral relaxations. *Mathematical Programming*, 121(2):307–335, 2010.

[43] G. Rinaldi. rudy, a graph generator. https://www-user.tu-chemnitz.de/~helmberg/sdp_software.html, 2018.

[44] V. J. Rodrigues de Sousa, M. F. Anjos, and S. Le Digabel. Computational study of valid inequalities for the maximum k-cut problem. *Annals of Operations Research*, 265(1):5–27, 2018.

[45] V. J. Rodrigues de Sousa, M. F. Anjos, and S. Le Digabel. Improving the linear relaxation of maximum k-cut with semidefinite-based constraints. *EURO Journal on Computational Optimization*, 7(2):123–151, 2019.

[46] H. D. Sherali and B. M. P. Fraticelli. Enhancing RLT relaxations via a new class of semidefinite cuts. *Journal of Global Optimization*, 22(1–4):233–261, 2002.

[47] E. R. van Dam and R. Sotirov. Semidefinite programming and eigenvalue bounds for the graph partition problem. *Mathematical Programming*, 151(2):379–404, 2015.

[48] G. Wang and H. Hijazi. Exploiting sparsity for the min k-partition problem. *Mathematical Programming Computation*, 12:109–130, 2020.

[49] A. Wiegele. Biq mac library - binary quadratic and max cut library. http://biqmac.uni-klu.ac.at/biqmaclib.html, 2019.

[50] Q. Wu, Y. Wang, and Z. Lü. A tabu search based hybrid evolutionary algorithm for the max-cut problem. *Applied Soft Computing*, 34:827–837, 2015.

[51] W. Zhu, G. Lin, and M. M. Ali. Max-k-cut by the discrete dynamic convexized method. *INFORMS Journal on Computing*, 25(1):27–40, 2013.