Efficient Enumeration of the Optimal Solutions to the Correlation Clustering problem

Nejat Arinik, Vincent Labatut, Rosa Figueiredo

Efficient Enumeration of the Optimal Solutions to the Correlation Clustering problem

Nejat Arınık, Rosa Figueiredo & Vincent Labatut

January 12, 2023

Abstract

According to the *structural balance* theory, a signed graph is considered structurally balanced when it can be partitioned into a number of modules such that positive and negative edges are respectively located inside and between the modules. In practice, real-world networks are rarely structurally balanced, though. In this case, one may want to measure the *magnitude* of their imbalance, and to identify the set of edges causing this imbalance. The correlation clustering (CC) problem precisely consists in looking for the signed graph partition having the least imbalance. Recently, it has been shown that the space of the optimal solutions of the CC problem can be constituted of numerous and diverse optimal solutions. Yet, this space is difficult to explore, as the CC problem is NP-hard, and exact approaches do not scale well even when looking for a *single* optimal solution. To alleviate this issue, in this work we propose an efficient enumeration method allowing to retrieve the complete space of optimal solutions of the CC problem. It combines an exhaustive enumeration strategy with neighborhoods of varying sizes, to achieve computational effectiveness. Results obtained for middle-sized networks confirm the usefulness of our method.

Keywords: Correlation Clustering, Enumeration Strategy, Local Search, Optimal Solution Space, Signed Graph, Structural Balance.

Cite as: N. Arınık, R. Figueiredo & V. Labatut. Efficient Enumeration of the Optimal Solutions to the Correlation Clustering problem. Journal of Global Optimization, 2023 (forthcoming)

1 Introduction

Many real-world social systems involve antagonistic relations, a feature that can be directly modeled through signed networks, which contain both positive and negative edges. Alternatively, in some cases, signed edges can be extracted from originally unsigned relations [29]. The exact semantic of these edges depends on the considered system: friendship/foe [35] and trust/distrust [38] in social networks, inhibition/activation in biological networks [16, 45], agreement/disagreement in political vote networks [5, 7, 19] and so on.

Determining the structural balance of a signed network is a key aspect in the investigation of the structure of polarized relations. According to the *structural balance* theory, a signed graph is considered to be balanced if it can be partitioned into two [12] or more [17] modules (i.e. clusters), such that all positive edges are located inside these modules, whereas negative ones lie between them.

However, it is very rare for a real-world network to be *perfectly* balanced, in which case one wants to assess the *magnitude* and the *causes* of the imbalance. For a given partition, this imbalance is traditionally measured by counting the number of *frustrated edges* [12, 46], i.e. positive edges located in-between modules and negative ones located inside them. Computing the graph imbalance amounts to identifying the partition corresponding to the lowest imbalance measure over the space of all possible partitions. The optimal solution found exhibits the source of the imbalance, i.e., the frustrated edges. This minimization problem is known as the *correlation clustering* (CC) problem, proven to be NP-hard [8].

When solving an instance of the CC problem, the standard approach in the literature is to find a *single* solution and focus the rest of the analysis on it, as if it were the *only* optimal solution [23, 44]. Yet, as shown empirically, it is possible that several, and even many, alternative optimal solutions exist for the considered instance [6, 10, 11]. All these alternate solutions are equally relevant in terms of the objective function of the CC problem. Yet, they can be very different, in terms of how they partition the graph. One can then wonder how many solutions shape the space of optimal solutions, and if many,

how different/diverse they are. From an application point of view, these are important questions to answer in order to identify how many of these solutions are more suitable to the situation at hand.

In order to answer these questions, we recently conducted an empirical study in our previous work [6]. Therein, we first enumerated the optimal solutions of a collection of instances of the CC problem through Integer Linear Programming (ILP) modeling [18]. We applied the best state-of-the-art optimal solution enumeration method, as proposed by Danna *et al.* [15] and incorporated in the commercial solver CPLEX [30]. Then, we studied the obtained spaces of optimal solutions through complex network analysis. In that work, we found out that slightly imbalanced networks tend to have fewer and structurally more similar optimal solutions forming a single solution class, whereas a higher imbalance leads to many diverse solutions possibly forming multiple classes of solutions. Consequently, we concluded that it is of great importance that the decision-maker generates the full set, or as many optimal solutions as possible, for a given instance.

In such an analysis, guaranteeing the completeness of the space of optimal solutions requires employing exact approaches. However, it is well known that, due to their NP-hard nature, generic exact approaches able to solve most clustering problems (including the CC problem) are very costly in terms of execution time and do not scale well, even when looking for a *single* optimal solution [27]. In that respect, the enumeration method by Danna *et al.* is the best state-of-the-art enumeration method able to handle the CC problem so far. However, as it is a generic method designed to handle any combinatorial problem formulated in ILP, it cannot take full advantage of the problem knowledge. Consequently, the maximum number of vertices (i.e. graph order) that we could handle with their method in our previous work was only 36. Moreover, the execution time of the enumeration process for such small graphs was very long (typically more than one day). This highlights the need for an efficient optimal space enumeration method for the CC problem.

In this work, we aim to obtain this efficiency by putting the problem knowledge into the enumeration process. Based on the high similarity observed empirically between optimal solutions belonging to the same class of solutions, it integrates into an exact approach a local search mechanism, which relies on the use of neighborhoods of different sizes. In order to accelerate the proposed enumeration method, we develop pruning strategies based on optimal conditions satisfied by partial solutions. Our main contribution is the combination of local search and pruning strategies, which gives rise to an efficient enumeration method. We present extensive computational experiments established for the proposed method, relying on sparse and dense signed networks.

The rest of the article is organized as follows. In Section 2, we review the literature related to the exact enumeration of all optimal solutions for the CC problem. In Section 3, we give the formal definition of the CC problem. Next, we introduce our enumeration method in Section 4, and detail our complete neighborhood search and pruning strategies in Section 5 and 6, respectively. We put the proposed method into practice on a selection of partially and fully connected signed networks in Section 7 and discuss our results in Section 8. Finally, we review our main findings in Section 9, and identify some perspectives for our work.

2 Related Work

There are several methods proposed in the literature to solve the CC problem exactly [18, 23, 32], however very few were designed to enumerate the whole space of its optimal solutions. In fact, very few methods were proposed to enumerate all the optimal solutions of *any* combinatorial problem (e.g., [4] for maximal covering problem). In this section, we review the existing optimal solution enumeration methods for the CC problem and for related problems.

The literature provides two main approaches to enumerate all optimal solutions of a combinatorial problem: *Branch-and-Bound* and *Fixed-Parameter Enumeration*. In the case of the CC problem, most works focus on the former.

Branch-and-Bound Approach The enumeration of all optimal solutions of the CC problem (like for any combinatorial problem) through a branch-and-bound method can be performed through two algorithmic methods: Integer Linear Programming (ILP) vs. *ad hoc* branch-and-bound programming. Their main difference is that the former can tackle any problem translated in the language of mathematical programming through any industrial optimization solver, whereas in the latter the construction of the branch-and-bound tree relies on problem-specific idiosyncrasies.

Based on how the branch-and-bound tree is used, the ILP approach can be implemented in two different ways. The first one relies on a simple sequential process: a slightly modified version of the original problem is solved as many times as there are solutions in the optimal solution space (like in [4] for the maximal covering problem). At each iteration, already-found optimal solutions are sequentially

added as constraints into the mathematical model, in order to exclude them during future iterations. However, the drawback of this approach is that a branch-and-bound tree needs to be built from scratch for each new optimal solution found. The second type of ILP implementation is called *OneTree*, and was proposed by Danna *et al.* [15]. Instead of the previous simple sequential approach, it relies on a more efficient two-step method that allows reducing the computational effort. As its name suggests, it constructs a single branch-and-bound tree. It first builds and explores the search tree in order to find efficiently the first optimal solution, and then enumerate all the alternative optimal solutions based on the same tree. This method is available in the industrial optimization solver CPLEX [30], and we explain in Section 4.3 how we use it for the CC problem. In our previous work [6], we used this approach for generating the optimal solution space of complete random graphs up to 36 vertices.

An *ad hoc* B&B programming method constructs the branch-and-bound tree differently from what we described before. For a given clustering problem, these methods systematically construct partial solutions by assigning the vertices to one of the existing modules. This produces a search tree, whose branches correspond to assignments of vertices to modules, and whose nodes correspond to partial assignments. Similar to Danna *et al.* [15], Brusco and Steinly [10] propose a two-step method for generating all optimal CC solutions. They first identify the optimal objective function value by finding an optimal solution, then use the optimal value as input when building another branch-and-bound tree from scratch. As shown by Figueiredo & Moura [23], this method does not deal well with finding a single optimal solution for graphs with more than 20 vertices.

Fixed-Parameter Enumeration Approach This is also an *ad hoc* method which requires to transform a part of the considered problem into a new input parameter. This in turn guarantees to bound the overall running time as a function of an input parameter. Assuming that this input parameter is small, the parameterized enumeration is efficient in practice. Damaschke [14] proposes an FPT (Fixed-Parameter Tractable) algorithm to enumerate all optimal solutions in a given graph for the *Cluster Editing* problem, which is equivalent to the CC problem when the input graph is complete and unweighted. Concretely, this FPT algorithm makes the number of frustrated edges in structural balance parameterized to solve the problem. However, as shown in our previous work [6], a drawback of this approach is that the amount of imbalance, the input parameter, can be very large. Furthermore, Damaschke does not provide any computational results in his theoretical work.

The computational results presented in the works mentioned in this section identify the ILP branch-and-bound as the more efficient method for exactly solving the CC problem. This is explained by a tightened initial linear relaxation of the ILP formulation and the quality of the cuts used. In this work, we apply an ILP branch-and-bound method for the complete enumeration of the optimal CC solutions, and propose a compromise strategy between the sequential approach and the *OneTree* method from Danna *et al.* [15].

3 Correlation Clustering Problem

Let us now introduce the notations necessary for defining the correlation clustering problem. Let $G = (V, E)$ be an *undirected graph*, where V and E are the sets of vertices and edges, respectively. We note $n = |V|$ and $m = |E|$ the numbers of vertices (i.e. graph order) and edges, respectively. We assume that the graph contains no loops, i.e. edges connecting a vertex to itself. Moreover, $G[S]$ denotes the subgraph induced by vertex subset $S \subset V$.

Consider a function $s : E \to \{+, -\}$ that assigns a sign to each edge in E. An undirected graph G together with a function s is called a *signed graph*, denoted by $G = (V, E, s)$. An edge (u, v) is called negative if $s(u, v) = -$ and positive if $s(u, v) = +$. We note E^- and E^+ the sets of negative and positive edges in the signed graph, respectively. Let also define the positive graph G^+ of a given signed graph G as the subgraph $(V, E^+, \{+\})$ (i.e. same vertices, but only the positive edges). The entries a_{uv} of the signed adjacency matrix $\mathbf{A}$ associated with a signed graph G are defined in Equation 1.

$$a_{uv} = \begin{cases} 1, & \text{if } (u, v) \in E^+, \\ -1, & \text{if } (u, v) \in E^-, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Let $P = \{M_1, ..., M_\ell\}$ $(1 \leq \ell \leq n)$ be an ℓ-partition of V, i.e. a division of V into ℓ non-overlapping and non-empty subsets M_i $(1 \leq i \leq \ell)$ called *modules*. The partition P is called a *solution* of the CC problem for the given graph G. Given a solution P, an edge (u, v) is called *internal* if it is located inside a module, i.e., u and v belong to the same module. Likewise, an edge (u, v) is called *external* if it is located

between any two modules, i.e., u and v belong to two different modules. Given $\sigma \in \{+,-\}$, we define the set of positive/negative (depending on σ) edges connecting two modules $M_i, M_j \in P$ as $E^\sigma(M_i,M_j) = \{(u,v) \mid (u,v) \in E^\sigma, u \in M_i \text{ and } v \in M_j\}$. We also define $E(M_i,M_j) = E^-(M_i,M_j) \cup E^+(M_i,M_j)$ as the set of all edges connecting these modules. Similarly, we note $\Omega^\sigma(M_i,M_j) = \sum_{(u,v)\in E^\sigma(M_i,M_j)} a_{uv}$ the *signed* number of positive/negative edges connecting these modules: note that this value can be negative if $\sigma = -$. We also define $\Omega(M_i,M_j) = \Omega^-(M_i,M_j) + \Omega^+(M_i,M_j)$ as the signed sum of the edges connecting the same modules. It can also be negative, if there are more negative than positive edges between M_i and M_j.

The *Imbalance* $I(P)$ of a partition P is defined as the total number of frustrated edges, i.e. the numbers of *positive* edges located *between* modules and of *negative* edges located *inside* them:

$$I(P) = \sum_{1\le i<j\le \ell} \Omega^+(M_i,M_j) - \sum_{1\le i\le \ell} \Omega^-(M_i,M_i). \tag{2}$$

The objective of the CC problem is to minimize the number of frustrated edges. This problem can be formally described as follows.

Problem 1 (CC problem)**.** *Given a signed graph $G = (V,E,s)$, the correlation clustering problem consists in finding a partition P of V such that the imbalance $I(P)$ is minimized.*

To the best of our knowledge, this NP-hard minimization problem appears under this name for the first time in Bansal's paper [8], although it is addressed before in the literature (e.g. [20]).

4 Enumeration of Optimal CC Solutions

Any method that enumerates optimal solutions needs to find an initial optimal solution by optimally solving the problem at hand. This is why we first describe an efficient method for finding an optimal solution for the CC problem (Section 4.1), before presenting the methods for identifying an alternative optimal solution (Section 4.2) and enumerating all optimal solutions (Section 4.3).

4.1 Finding an Initial Optimal Solution

As largely illustrated in the literature (e.g. [18], [23]), the CC problem can be modeled by means of an ILP proposed for the *Uncapacitated Graph Clustering* problem [39]. Note that the model can handle weighted graphs, but we use it on unweighted ones in the context of this article.

For each pair of vertices $u,v \in V : u < v$, a binary variable is first defined to describe the relative module membership of pairs of vertices, i.e.,

$$x_{uv} = \begin{cases} 1, & \text{if } u \text{ and } v \text{ are in a same module,} \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Then, the ILP formulation of the CC problem for unweighted signed graphs is written as follows:

$$\text{Min} \quad \sum_{u,v\in V: uv\in E^-} x_{uv} + \sum_{u,v\in V: uv\in E^+} (1 - x_{uv}) \tag{4}$$

$$\text{s.t.} \quad x_{uv} + x_{vr} - x_{ur} \le 1, \quad \forall u < v < r \in V \tag{5}$$

$$x_{uv} - x_{vr} + x_{ur} \le 1, \quad \forall u < v < r \in V \tag{6}$$

$$-x_{uv} + x_{vr} + x_{ur} \le 1, \quad \forall u < v < r \in V \tag{7}$$

$$x_{uv} \in \{0,1\}, \quad \forall u,v \in V. \tag{8}$$

The objective function in (4) looks for a feasible solution minimizing the imbalance defined by Equation (2). A feasible solution, i.e., decision variables corresponding to a partition, must satisfy the triangle inequalities (5), (6) and (7). Finally, (8) describes the domain constraints for the variables in the formulation.

This ILP model is sufficient to find an optimal solution through an optimization solver, but it can be very time-consuming. One way to deal with this issue is to strengthen it through a *cutting plane* approach [40]. We use the 2-partition and the 2-chorded cycle valid inequalities as described by Grötschel and Wakabayashi in [28] (see Appendix A for more details). In the cutting plane approach, we add these two tight valid inequalities (only) during the root relaxation phase as described by Ales *et al.* [2], before proceeding to the construction of the branch-and-bound search tree. As reported in [2]

for a similar clustering problem, we also verified that adopting this cut strategy improves the overall processing time. There are other inequalities in the literature that can further tighten the LP relaxation for the CC problem. These are the 2-*chorded even wheel* [28] inequality and those defined in [42]. Nevertheless, in practice, together the 2-partition and the 2-chorded cycle inequalities describe a tight linear relaxation [28].

Throughout this work, we denote $ILP(G)$ as the formulation defined by Equations (4)–(8) and *B&C(ILP(G))* as the branch-and-cut procedure described above. Also, we denote $ILP{+}(G)$ as the strengthened ILP obtained after executing $B\&C(ILP(G))$, i.e., the formulation obtained by adding to $ILP(G)$ all the cuts generated by $B\&C(ILP(G))$. Finally, let $B\&B(ILP{+}(G))$ be the branch-and-bound procedure based on $ILP{+}(G)$. In the following, for the sake of simplicity, the term *solution* refers to a graph partition, as well as a feasible solution to the formulation $ILP{+}(G)$.

4.2 Finding an Alternative Optimal Solution

An enumeration method needs to constantly find an optimal solution that would be different from those identified before, and stored in a set S. For this purpose, we need to define an extended model of $ILP{+}(G)$, that we denote $ILP{+}jump(G,S)$.

Consider a partition P. Let $\mathbf{X^P} \in \{0,1\}^{n\times n}$ be the representative matrix of P defined as: $x^p_{ij} = 1$ if i and j belong to the same module in P; and $x^p_{ij} = 0$ otherwise. $ILP{+}jump(G,S)$ includes the set of constraints defined in Equation (9) on top of $ILP{+}(G)$:

$$\sum_{u,v\in V:u<v} |x^p_{uv} - x_{uv}| > 0, \forall P \in S. \tag{9}$$

We refer the reader to [24] for the implementation details of these constraints.

We see that this extended formulation allows us to find an alternative optimal solution other than the ones in S. Furthermore, given an optimal solution $P \in S$, to ensure that this alternative solution has the same objective value as $I(P)$, we add the objective function in (4) as a constraint, as shown in Equation (10).

$$\sum_{u,v\in V:uv\in E^-} x_{uv} + \sum_{u,v\in V:uv\in E^+} (1 - x_{uv}) \leq I(P). \tag{10}$$

Equation (9) along with Equation (10) ensures that each feasible solution to $ILP{+}jump(S)$ is an optimal solution to the original problem. Finally, we denote the corresponding branch-and-bound procedure based on formulation $ILP{+}jump(G,S)$ by $B\&B(ILP{+}jump(G,S))$.

4.3 Enumerating All Optimal Solutions

As we have mentioned before, the CC problem can have multiple optimal solutions. In this section, we are interested in methods enumerating all optimal solutions of the CC problem for a given signed graph G. In the following, we first present $OneTreeCC$ [15], which is the best general method to enumerate solutions available in the literature. It is incorporated in CPLEX [30], and we apply it to the formulation $ILP{+}(G)$. Then, we introduce our *ad hoc* method $EnumCC$, in the aim of obtaining a faster method able to handle relatively larger graphs.

We start with $OneTreeCC(G)$, which takes as input a signed graph G. The sketch of this two-step method is shown in Algorithm 1. In the first step (line 2), it finds an initial optimal solution based on $ILP{+}(G)$. We note $T_{B\&B}$ the branch-and-bound search tree constructed during this step. The search tree as well as the nodes considered at this first step are stored for further examination during the second step. Moreover, Danna *et al.* completely turn off dual tightening in the branch-and-bound procedure in order to guarantee exhaustive enumeration. According to the authors, this fact can have a negative impact on performance, in terms of computational time. In the second step (line 3), $OneTreeCC(G)$ enumerates the set S of all alternative optimal solutions by expanding the search tree $T_{B\&B}$ until obtaining the complete set of all optimal solutions.

To compete with $OneTreeCC(G)$, we propose a new method for the CC problem in order to completely enumerate the optimal partitions of a given signed graph. We call it $EnumCC(G, r_{max})$, and it takes as input a signed graph G and a maximum distance parameter r_{max}. As shown in Algorithm 2, it can be viewed as an improved version of the sequential approach proposed by Arthur *et al.* [4] for the Maximal Covering problem (described in Section 2). In the first step (line 2), $EnumCC(G, r_{max})$ obtains an initial optimal solution thanks to $B\&B(ILP{+}(G))$. In the second step, instead of directly "*jumping*" onto undiscovered optimal solutions one by one through $ILP{+}jump(G,S)$ (like in the sequential approach), we slightly change this process. Our method first discovers the recurrent

Algorithm 1: $OneTreeCC(G)$

Result: The set of all optimal solutions S

1 $S = \emptyset$

/* 1st step */

2 Solve $B\&B(ILP{+}(G))$ to obtain an optimal solution P, and let $T_{B\&B}$ be the constructed branch-and-bound search tree

/* 2nd step */

3 Expand fathomed nodes in $T_{B\&B}$ until enumerating the set S of all alternative optimal solutions.

neighborhood $RN_{\leq r_{max}}(P)$ of the current optimal solution P (line 4), with the hope of discovering new optimal solutions. The recurrent neighborhood $RN_{\leq r_{max}}(P)$ of an optimal solution P, is the set of optimal solutions reached directly or indirectly from P, depending on the maximum distance parameter r_{max}. The complete description of the Recurrent Neighborhood Search (RNS) is given in Section 5. Whether a new solution is found or not through $RNS(G, P, r_{max})$, the method then moves on to the next step, consisting in jumping onto a new solution P (line 6). These RNS and jumping phases are repeated, until all optimal solutions are discovered.

Clearly, $EnumCC(G, r_{max})$ can only improve the sequential approach as used in [4], provided that RNS discovers at least one new solution and that its execution time is shorter than that of $B\&B(ILP{+}jump(G, S))$. Furthermore, the choice of r_{max} is sensitive: it does not contribute much to the method when it is small, whereas the method becomes slower than the sequential approach when r_{max} is large.

Algorithm 2: $EnumCC(G, r_{max})$

Result: The set of all optimal solutions S

1 $S = \emptyset$

/* 1st step */

2 Solve $B\&B(ILP{+}(G))$ to obtain an optimal solution P

/* 2nd step */

3 **while** $P \neq$ *null* **do**

/* step 2.1 */

4 Apply $RNS(G, P, r_{max})$ to generate the recurrent neighborhood $RN_{\leq r_{max}}(P)$ of P

5 $S = S \bigcup RN_{\leq r_{max}}(P)$

/* step 2.2 */

6 Jump onto an undiscovered optimal solution P, with $P \notin S$, through $B\&B(ILP{+}jump(G, S))$ // if not possible, then $P = null$

7 **end**

The jumping phase in step 2.2 of $EnumCC(G, r_{max})$ can be time-consuming for two reasons. First, even finding an alternative solution can be costly. Second, the number of jumps, denoted by the notation $n_{jump}(.)$ (here, $n_{jump}(EnumCC(G, r_{max}))$), can be very large, despite the use of the neighborhood $RN_{\leq r_{max}}(P)$. Nonetheless, we expect to save time and compensate the time spent in the jumping phase thanks to $RN_{\leq r_{max}}(P)$.

In the rest of this work, we leave out the common parameter G from the notations of the mentioned methods and ILP models, for the sake of convenience: OneTreeCC(), $EnumCC(r_{max})$ and $ILP{+}jump(S)$.

In the next section, we detail the recurrent neighborhood search RNS used in Algorithm 2.

5 Recurrent Neighborhood Search (RNS)

Recurrent Neighborhood Search (RNS) is the common and undoubtedly the most important part of our method *EnumCC*. As mentioned in the previous section, if we want this method to compete with $OneTreeCC(G)$, *RNS* needs to be efficient and fast.

Before defining the neighborhood of a solution, i.e. of a partition, we need to select a distance function between partitions. In the following, we first present the edit distance (Section 5.1), then

introduce the algorithmic details of the *Complete Neighborhood Search (CoNS)* used in our Recurrent Neighborhood Search (Section 5.2). Finally, we explain the main procedure for *Recurrent Neighborhood Search* (Section 5.3).

Let us first introduce some definitions used in the rest of the article. A partition P with ℓ modules can be associated with one or more *membership vectors*. Such a vector of size n, denoted by π, defines a function over $V \rightarrow \{1, 2, ..., \ell\}$. For a given vertex u, $\pi(u)$ denotes the *label* of the module of P that contains u, while $M_{\pi(u)}$ denotes the module itself. Finally, given a partition P, we define an r-neighborhood structure, denoted by $N_r(P)$, as the family of partitions obtained by moving r vertices into different modules of P. We also define $N_r(\pi)$ as the family of corresponding membership vectors obtained by moving r vertices.

5.1 Edit Distance

In Natural Language Processing, and more generally in Computer Science, the edit distance [31] is defined as the minimum number of edit operations required to transform one string into another (or more precisely, their total cost). In the context of graph partitioning, with fixed vertices and edges, edit operations are used as *neighborhood search operators* in local search heuristics developed for graph problems [1, 9, 37]. Such an operation consists in moving one or more vertices (called *moving vertices*) from their current modules (called *source* modules) to other ones (called *target* modules). From the perspective of the membership vectors corresponding to the concerned partitions, this transforms a *source* vector into a *target* one. The number of moving vertices constitutes the cost of the edit operation. In this work, we are interested in edit operations whose cost is minimal.

Definition 2 (Min-edit operation)**.** *Consider an edit operation transforming a source membership vector into a target membership one. If the set of moving vertices is the minimum set required for this transformation, then it is a* ***min-edit operation****. Otherwise, we call it* ***non-min-edit operation****.*

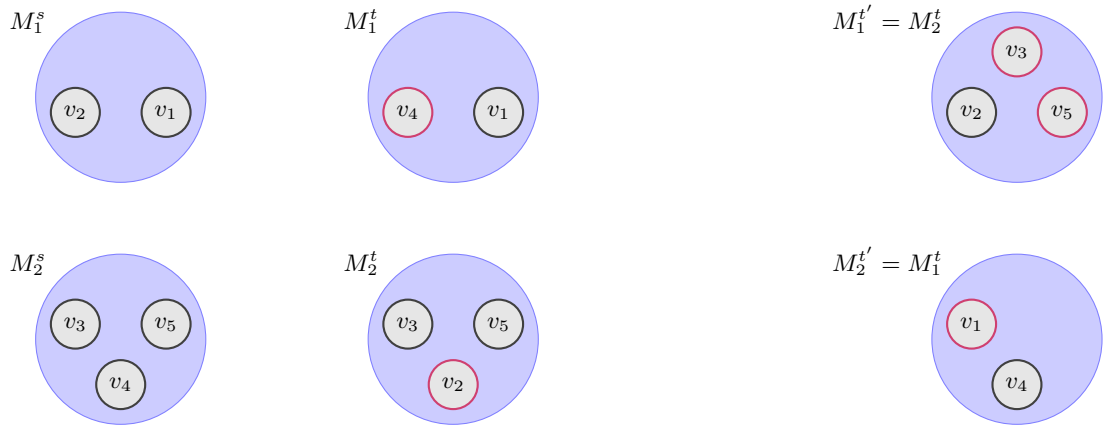
Notice that the cost of an edit operation is not always minimal. This is because two membership vectors, i.e. the module labels of two partitions, can be very different, but essentially suggest very similar module assignments for the vertices. The distinction between *min-edit* and *non-min-edit* operations is illustrated in Figure 1.

Importantly, the *edit distance* between two membership vectors is the cost of the min-edit operation allowing to turn one vector into the other. The calculation of such distance between two membership vectors can be done in polynomial time by solving an assignment problem (see Appendix B for more details).

Let us now introduce our notations related to the edit distance. Let $\boldsymbol{\pi}^s$ and $\boldsymbol{\pi}^t$ represent the source and target membership vectors for an edit operation, respectively. They are associated with the source and target partitions $P^s = \{M_1^s, M_2^s, ..., M_{\ell^s}^s\}$ and $P^t = \{M_1^t, M_2^t, ..., M_{\ell^t}^t\}$, containing ℓ^s and ℓ^t modules, respectively. Since the edit distance is symmetric, without loss of generality, let $\ell^s \leq \ell^t$. Moreover, we note $\pi^s \rightarrow \pi^t$ an edit operation applied onto P^s to obtain P^t, whose set of moving vertices is defined as ${}^{s}\vec{V}^{t} = \{u \mid \pi^s(u) \neq \pi^t(u)\}$. This means that the remaining vertices, called *non-moving* vertices, do not change modules, hence their module labels are the same in π^s and π^t. The cost of this edit operation is simply the number of vertices whose module has changed. We denote this cost with $cost(\pi^s \rightarrow \pi^t)$ and compute it by taking the cardinality of ${}^{s}\vec{V}^{t}$. In the rest of the text, we use r for short whenever we are interested only in the value of $cost(\pi^s \rightarrow \pi^t)$. Also, a *$r$-edit operation* (resp. *min-r-edit operation*) is any edit operation (resp. min-edit operation) whose cost equals r. Consider an edit operation $\pi^s \rightarrow \pi^t$ whose moving vertices are in set ${}^{s}\vec{V}^{t}$. The distinct labels of the source and target modules of a subset $V' \subset V$ of vertices are denoted, respectively, by $\pi^s(V') = \bigcup_{u \in V'} \pi^s(u)$ and $\pi^t(V') = \bigcup_{u \in V'} \pi^t(u)$. Finally, we sometimes need to refer to several modules in an edit operation $\pi^s \rightarrow \pi^t$. For this reason, we also define $M_L^s = \bigcup_{l \in L} M_l^s$ for any $L \subseteq \{1, .., \ell^s\}$ in the source partition P^s and $M_L^t = \bigcup_{l \in L} M_l^t$ for any $L \subseteq \{1, .., \ell^t\}$ in the target partition P^t.

To illustrate the aforementioned concepts, let us consider the example of Figure 1. As shown in Figure 1a, $\pi^s = (1, 1, 2, 2, 2)$, where $M_1^s = \{v_1, v_2\}$ and $M_2^s = \{v_3, v_4, v_5\}$. Let us consider an edit operation $\pi^s \rightarrow \pi^t$ with the moving vertices ${}^{s}\vec{V}^{t} = \{v_2, v_4\}$ (Figure 1b). The edit operation consists in moving v_2 into M_2^s and v_4 into M_1^s, which produces π^t. Hence, we have $M_1^t = \left(M_1^s \setminus \{v_2\}\right) \cup \{v_4\}$ and $M_2^t = \left(M_2^s \setminus \{v_4\}\right) \cup \{v_2\}$. The labels of the source and target distinct modules of ${}^{s}\vec{V}^{t}$ are $\pi^s({}^{s}\vec{V}^{t}) = \{1, 2\}$ and $\pi^t({}^{s}\vec{V}^{t}) = \{1, 2\}$, respectively. Also, we have $M^s_{\pi^s({}^{s}\vec{V}^{t})} = M_1^s \cup M_2^s$ and $M^t_{\pi^t({}^{s}\vec{V}^{t})} = M_1^t \cup M_2^t$ for this example. The sets $M^s_{\pi^s({}^{s}\vec{V}^{t})}$ and $M^t_{\pi^t({}^{s}\vec{V}^{t})}$ could be different, if some of

the moving vertices move into a module M_i^s other than $M^s_{\pi^s(\overrightarrow{{}^sV^t})}$.



(a) Source membership vector $\pi^s = (1, 1, 2, 2, 2)$, where $M_1^s = \{v_1, v_2\}$ and $M_2^s = \{v_3, v_4, v_5\}$.

(b) Target membership vector $\pi^t = (1, 2, 2, 1, 2)$, where $M_1^t = \{v_1, v_4\}$ and $M_2^t = \{v_2, v_3, v_5\}$. The moving vertices are shown with a purple border. The operation $\pi^s \rightarrow \pi^t$ is min-2-edit, where $\overrightarrow{{}^sV^t} = \{v_2, v_4\}$.

(c) Target membership vector $\pi^{t'} = (2, 1, 1, 2, 1)$, where $M_1^{t'} = \{v_2, v_3, v_5\}$ and $M_2^{t'} = \{v_1, v_4\}$. The moving vertices are shown with a purple border. Since $M_1^{t'} = M_2^t$ and $M_2^{t'} = M_1^t$, $\pi^s \rightarrow \pi^{t'}$ is not a min-3-edit operation, where $\overrightarrow{{}^sV^{t'}} = \{v_1, v_3, v_5\}$.

Figure 1: Illustrative example for two edit operations applied to the same source membership vector π^s, whose associated partition is shown in Fig. 1a. The first operation, $\overrightarrow{{}^sV^t}$, is shown in Fig. 1b, and the second, $\overrightarrow{{}^sV^{t'}}$, in Fig. 1.c. The underlying partition in P^t and $P^{t'}$ is the same, but the corresponding membership vectors π^t and $\pi^{t'}$ are different. Consequently, $\overrightarrow{{}^sV^t}$ is a *min-edit* operation, whereas $\overrightarrow{{}^sV^{t'}}$ is not.

5.2 Complete Neighborhood Search (*CoNS*)

Our neighborhood search method $CoNS(G, \pi^s, r)$ takes as input a signed graph G, a membership vector π^s associated with an optimal partition P^s of ℓ modules and an edit distance r. It returns the set $N_r(\pi^s)$ of membership vectors associated with *all* optimal partitions, obtained by applying min-r-edit operations to a given membership vector π^s. To ensure the completeness of set $N_r(\pi^s)$, $CoNS(G, \pi^s, r)$ analyses all combinations of moving vertices and their target modules. We use two pruning strategies in order to eliminate some combinations that do not lead to an optimal partition when an edit operation is applied. Both of them are described in this section, whereas the properties upon which they are based are detailed in Section 6. In simple terms, they are based on the relations between a set of moving vertices and their target modules. In the following, we present $CoNS(G, \pi^s, r)$ as a pipeline procedure consisting of four parts, as shown in Algorithm 3.

In the first part, $CoNS(G, \pi^s, r)$ starts by selecting a candidate set of r moving vertices $\overrightarrow{{}^sV^t}$ among all vertices in V (line 2). The set of all possible r moving vertices is obtained by generating all combinations $\binom{V}{r}$. We know that probably not every $\overrightarrow{{}^sV^t}$ leads to a min-r-edit operation and an optimal partition P^t. To detect such cases, we apply the first pruning procedure (line 3), called *internalPruning* and depicted in Algorithm 4. This procedure checks, without the knowledge of target modules, if $\overrightarrow{{}^sV^t}$ satisfies some connectivity conditions related to the atomicity property (defined in Section 6.2) and the MVMO property up to three moving vertices (defined in Section 6.3). As we will see in Section 6, whenever one of these two conditions is not satisfied, the candidate set $\overrightarrow{{}^sV^t}$ can be pruned.

At this point, the set of moving vertices $\overrightarrow{{}^sV^t}$ is fixed. Next, the algorithm defines step by step the target membership vector π^t by setting the target module of each vertex in $\overrightarrow{{}^sV^t}$. We handle the process of generating the target membership vector π^t in three steps, which constitutes the second, third and fourth parts of Algorithm 3, in order to take advantage of our pruning strategy. The pruning strategy used in these parts is slightly different from that of the first part: it is based on external connections from $\overrightarrow{{}^sV^t}$ to $\overrightarrow{{}^sV^t}$. The concerned procedure is called *externalPruning* and depicted in Algorithm 5. It verifies, based on external relations among $\overrightarrow{{}^sV^t}$, if $\overrightarrow{{}^sV^t}$ satisfies the min-edit (Section 6.1), atomicity

Algorithm 3: $CoNS(G, \pi^s, r)$.

Input : signed graph $G(V, E, s)$, source membership vector π^s with ℓ module labels, edit distance r
Output: $N_r(\pi^s)$

1 $T = \{1, 2, .., \ell\}$ // module labels in π^s
/* 1st part */
2 **for** *each* $^{s\vec{V}^t} \in \binom{V}{r}$ **do**
3 **if** *internalPruning*(G, π^s, $^{s\vec{V}^t}$) **then**
4 go to line 2
5 **end**
/* 2nd part */
6 $T_0 = \pi(^{s\vec{V}^t}) \bigcup \{Unknown\}$ // initial target module labels
7 $T_{rem} = T \setminus T_0$ // remaining module labels
8 **for** *each* $T_0' \in \mathcal{P}(\binom{T_0}{r})$, *s.t.* $\pi^s(u) \notin T_0', \forall u \in {}^{s\vec{V}^t}$ **do** // $\mathcal{P}(.)$: permutations
9 Let π^t be the partition after assigning T_0' to $^{s\vec{V}^t}$
10 **if** *externalPruning*(G, π^s, π^t, $^{s\vec{V}^t}$) **then**
11 go to line 8
12 **end**
/* 3rd part */
13 Let B be the the number of *Unknown* labels in π^t
14 **if** $B \neq 0$ **then**
15 Let $^{s\vec{V}^t}{}_{rem}$ be the moving vertices, whose $\pi^t(^{s\vec{V}^t}{}_{rem}) = Unknown$
16 **for** $b \in \{1, .., B\}$ **do**
17 Let $\mathcal{I}$ be the set of all vectors of size B with elements belonging to $\{1, .., b\}$, s.t. each element for each vector appears once
18 **for** *each* $I \in \mathcal{I}$ **do**
19 Update π^t with the assignment of I to $\pi^t(^{s\vec{V}^t}{}_{rem})$
20 **if** *externalPruning*(G, π^s, π^t, $^{s\vec{V}^t}$) **then**
21 go to line 18
22 **end**
/* 4th part */
23 $T_{rem}^{+} = T_{rem} \cup \{\ell + 1, .., \ell + b\}$ // expand for empty modules
24 **for** *each* $T_r' \in \mathcal{P}(\binom{T_{rem}^{+}}{b})$ **do**
25 Replace the assignment of I to $^{s\vec{V}^t}{}_{rem}$ by T_{rem} in π^t
26 **if** $I(\pi^s) = I(\pi^t)$ *and* $\neg$*externalPruning*(G, π^s, π^t, $^{s\vec{V}^t}$) **then**
27 $N_r(\pi^s) = N_r(\pi^s) \bigcup \pi^t$
28 **end**
29 **end**
30 **end**
31 **end**
32 **else**
33 **if** $I(\pi^s) = I(\pi^t)$ **then**
34 $N_r(\pi^s) = N_r(\pi^s) \bigcup \pi^t$
35 **end**
36 **end**
37 **end**
38 **end**

Algorithm 4: *internalPruning*$(G, \pi^s, {}^{s}\vec{V}^{t})$

Input : signed graph $G(V, E, s)$, source membership vector π^s, moving vertices ${}^{s}\vec{V}^{t}$
Output: Boolean variable
/* 1st part */
1 **if** $\neg$*intAtomic*$(G, \pi^s, {}^{s}\vec{V}^{t})$ **then** // Properties 6 and 7a
2 | **return** true
3 **end**
/* 2nd part */
4 **if** $2 \leq |{}^{s}\vec{V}^{t}| \leq 3$ *and* $\neg$*intMVMO*$(G, \pi^s, {}^{s}\vec{V}^{t})$ **then** // Lemma 11.1 and Lemma 12
5 | **return** true;
6 **end**
7 **return** false;

(Section 6.2) and MVMO (Section 6.3) properties. As discussed in Section 6, whenever one of these three conditions is not satisfied, the possibility of target modules for candidate set ${}^{s}\vec{V}^{t}$ is pruned. These pruning strategies can be applied even when some target modules are not already decided. Therefore, they are invoked, whenever the target modules of ${}^{s}\vec{V}^{t}$ are updated (lines 10, 20 and 26).

In the second part, a moving vertex is allowed to move into a module, whose label is in T_0. The set T_0, defined at line 6, contains the labels of all unique source modules of ${}^{s}\vec{V}^{t}$, as well as a label *Unknown*. We use this label *Unknown* to indicate that the target module of a vertex is not already assigned. We also define T_{rem} as the remaining modules. Notice that, from this point, a target membership vector π^t can contain multiple vertices with target modules labeled as *Unknown*. In the third part, this *Unknown* label allows us to handle the other modules in T_{rem} as target possibilities. At line 8, we consider all possible permutations in $\mathcal{P}(\binom{T_0}{r})$ such that, for each vertex in ${}^{s}\vec{V}^{t}$, the target and source modules are different.

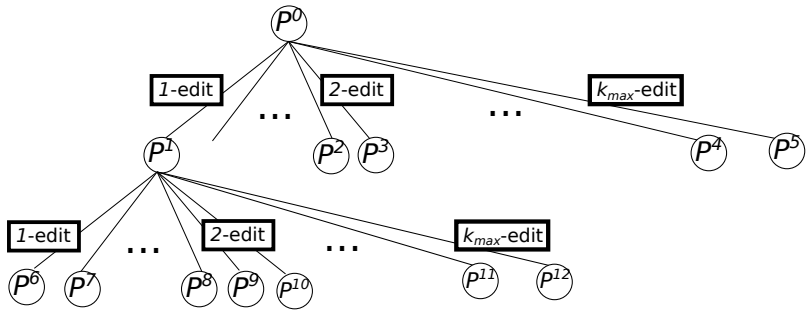
At this point, the *externalPruning* procedure can be applied for vertices whose target modules are already assigned. If the pruning conditions are not satisfied, the algorithm proceeds with the third part. Line 14 checks if no target module in π^t is unknown ($B = 0$). If so, then a new optimal partition has been found, and the algorithm goes on with the final test (line 32). However, if it is not the case, the algorithm enters the fourth part: all the combinations in T_{rem} (more precisely, T_r^+) are considered as a possible assignment of unknown target modules. One can expect this task to be computationally expensive. Thus, instead of directly determining the target modules of each vertex in ${}^{s}\vec{V}^{t}{}_{rem}$, we first generate all the *coupling* scenarios of moving vertices going into the same *Unknown* target module. To handle this, let b be the number of distinct *Unknown* target modules (line 16), for each value in the range of $\{1, .., B\}$. We build all possible *coupling* scenarios $\mathcal{I}$ for the value b (line 17). We couple a pair of remaining moving vertices, if they move into the same *Unknown* target module. For instance, $\{Unknown1, Unknown1, Unknown2\}$ indicates that the last moving vertex moves into a different *Unknown* target module than the first two. Once the target membership vector π^t is enriched with a *coupling* scenario $\mathcal{I}$, the external pruning is repeated in line 20. If the pruning conditions are not satisfied, the algorithm finally determines the target modules of ${}^{s}\vec{V}^{t}{}_{rem}$, by respecting the actual *coupling* scenario $\mathcal{I}$ (line 25). In the end, we add the current $\pi^s \rightarrow \pi^t$ into $N_r(\pi^s)$, if the optimality condition is met (line 26).

Without the problem-specific pruning strategies (internal and the external verification of properties from Section 6.3), Algorithm 3 amounts to a brute force method. Although $CoNS(G, \pi^s, r)$ can be very costly (even for small values of r), the properties described in Section 6 allow us to develop a method whose cost is relatively lower than the brute force method. Indeed, given a membership vector of size n with ℓ module labels, the number of non-repetitive permutations of all the combinations of r moving vertices is $n^r \times r!$ and the number of non-repetitive permutations of all the combinations of target modules is $\ell^r \times r!$. In the end, the brute force method is $\Theta(n^r \times r! \times \ell^r \times r!)$, where we assume that $\ell > r$. Computational results in Section 8.1 show that the pruning strategies reduce the computational cost of this procedure.

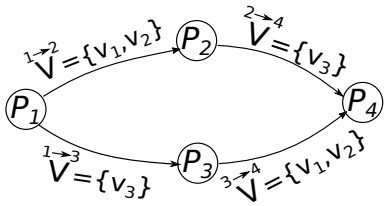
Algorithm 5: *externalPruning*$(G, \pi^s, \pi^t, {}^s\vec{V}^t)$

Input : signed graph $G(V, E, s)$, source membership vector π^s, target membership vector π^t, moving vertices ${}^s\vec{V}^t$
Output: Boolean variable

```
/* 1st part                                                                    */
```
1 **if** $\neg$*minEdit*$(\pi^s, \pi^t, {}^s\vec{V}^t)$ *or* $\neg$*extAtomic*$(G, \pi^s, \pi^t, {}^s\vec{V}^t)$ **then** // Properties 3, 5, 7b and 8
2 | **return** true
3 **end**
```
/* 2nd part                                                                    */
```
4 **if** $2 \leq |{}^s\vec{V}^t|$ *and* $\neg$*extMVMO*$(G, \pi^s, \pi^t, {}^s\vec{V}^t)$ **then** // Lemmas 11.2, 12 and 13
5 | **return** true
6 **end**
7 **return** false;



(a) RNS Tree.



(b) Example of duplication in RNS. Both P^2 and P^3 generate P^4.

Figure 2: Illustrations for the Recurrent Neighborhood Search (RNS). In both figures, circles represent membership vectors associated with optimal partitions.

5.3 Main Procedure

The main procedure for *Recurrent Neighborhood Search (RNS)* consists in applying *CoNS* from Section 5.2 in a recursive manner, in order to build a search tree called the *RNS tree*. This recursive procedure is depicted in Figure 2a. The root node of *RNS tree* corresponds to the initial partition P. All other nodes of the RNS tree constitute the set of optimal solutions reached directly or indirectly from P. A solution associated with a node at some level h of the *RNS tree* is obtained after $h-1$ applications of the *CoNS* procedure. The height of the *RNS* tree is unknown but necessarily finite, as we will see next.

Method $RNS(G, P, r_{max})$, which takes as input a signed graph G, an optimal partition P and a maximum edit distance r_{max}, is detailed in Algorithm 6. It starts from the initial partition P (line 1), and enumerates a set of its neighbor optimal partitions up to edit distance r_{max} (line 4), denoted by $RN_{\leq r_{max}}(P)$. Each partition P^t associated with $\pi^t \in N_r(\pi^s)$ is considered as a potential initial partition for seeking new neighbor partitions (line 8). Despite the pruning techniques which avoid some repetitions, it is possible that P^t has been already discovered and belongs to $RN_{\leq r_{max}}(P)$ (see such an example in Figure 2b). Line 9 checks this case, and discards P^t, if required. This process is repeated in a recursive manner, until there is no new partition obtained.

In the rest of this work, for the sake of convenience, we leave out the common parameter G and P^s (or P) when referring to the mentioned methods: $CoNS(r)$ and $RNS(r_{max})$.

6 Pruning Strategies

In this section, we present the pruning strategies incorporated in the method $CoNS(G, \pi^s, r)$, described in Section 5. But before this, we need to introduce some additional definitions. Consider an edit operation $\pi^s \rightarrow \pi^t$ with moving vertices in set ${}^s\vec{V}^t$, where membership vector π^s (resp. π^t) is associated

Algorithm 6: $RNS(G, P, r_{max})$

Input : signed graph $G(V, E, s)$, partition P, maximum edit distance r_{max}
Output: $RN_{\leq r_{max}}(P)$

$U = \{P\}$ // a set of unprocessed partitions
$RN_{\leq r_{max}}(P) = \emptyset$ // a set of discovered partitions
while $U \neq \emptyset$ **do**
  for *each* $r \in \{1, .., r_{max}\}$ **do**
    Let P^s be an element of U, where π^s is an associated membership vector of P^s
    $RN_{\leq r_{max}}(P) = RN_{\leq r_{max}}(P) \bigcup P^s$
    $N_r(\pi^s) = CoNS(G, \pi^s, r)$ // Complete Neighborhood Search
    for $\pi^t \in N_r(\pi^s)$ **do** // P^t is the associated partition of π^t
      /* memory-based duplication removal */
      if $P^t \notin RN_{\leq r_{max}}(P)$ **then**
        $U = U \bigcup P^t$
      end
    end
  end
end

with source partition P^s (resp. target partition P^t). Let us define $\vec{V}^s_{\pi^s(u)} = {}^s\vec{V}^t \cap M^s_{\pi^s(u)}$ (resp. $\vec{V}^t_{\pi^s(u)} = {}^s\vec{V}^t \cap M^t_{\pi^s(u)}$) as the set of moving vertices being in the source module of vertex u in P^s (resp. in P^t). Also, let $\vec{V}^s_{\pi^t(u)} = {}^s\vec{V}^t \cap M^s_{\pi^t(u)}$ (resp. $\vec{V}^t_{\pi^t(u)} = {}^s\vec{V}^t \cap M^t_{\pi^t(u)}$) be the set of moving vertices being in the target module of u in P^s (resp. in P^t). Finally, we define ${}^s\vec{V}^t_u = \left(\vec{V}^s_{\pi^s(u)} \cup \vec{V}^t_{\pi^s(u)} \cup \vec{V}^t_{\pi^t(u)} \cup \vec{V}^s_{\pi^t(u)}\right) \setminus \{u\}$ as the set of moving vertices connected to vertex u in $\widetilde{G}[{}^s\vec{V}^t]$. To illustrate the aforementioned notations related to ${}^s\vec{V}^t$, we consider again the same example from Figure 1. For $v_1 \in {}^s\vec{V}^t$, we have $\pi^s(v_1) = 1$ and $\pi^t(v_1) = 3$. Then, $\vec{V}^s_{\pi^s(v_1)} = \{v_1, v_2\}$ is the set of vertices in M^s_1 and $\vec{V}^t_{\pi^t(v_1)} = \{v_1\}$ since v_1 is the only moving vertex belonging to M^t_3. Also, $\vec{V}^s_{\pi^t(v_1)} = \vec{V}^s_{\{3\}} = \emptyset$ since there is no moving vertex in M^s_3, and $\vec{V}^t_{\pi^s(v_1)} = \{v_3\}$ since $v_3 \in M^t_1$. Finally, we get ${}^s\vec{V}^t_{v_1} = \{v_2, v_3\}$.

6.1 Non-Minimum Edit Operation Pruning

When generating the set $N_r(\pi^s)$ in RNS, only considering min-edit operations is sufficient. Next, we define a property allowing to detect, in some cases, that an edit operation is not minimal. The idea behind it is to define two sufficient conditions on the set of moving vertices implying that there exist an operation of smaller cost leading to the same partition. In the following, let $\pi^s \rightarrow \pi^t$ be an edit operation with moving vertices in ${}^s\vec{V}^t$.

Property 3 (Non-min-edit operation)**.** *Let $\vec{V_a} \subseteq {}^s\vec{V}^t$ be a subset of ${}^s\vec{V}^t$. Assume that the vertices in $\vec{V_a}$ constitute the majority part of module M^s_a (i.e. $|\vec{V_a}| > |M^s_a \setminus \vec{V_a}|$) and that they are in the same target module M^t_b in P^t (i.e. $|\pi^t(\vec{V_a})| = 1$). Moreover, suppose there exists a subset of moving vertices $\vec{V_b} \subseteq {}^s\vec{V}^t$, such that $\emptyset \subseteq \vec{V_b} \subseteq M^s_b$. Let us define $\overline{V}_a$ (resp. $\overline{V}_b$) as $M^s_a \setminus {}^s\vec{V}^t$ (resp. $M^s_b \setminus {}^s\vec{V}^t$) in P^s. Each condition in the following is a sufficient condition for $\pi^s \rightarrow \pi^t$ to be a non-min-edit operation:*

(a) If $\vec{V_b}$ does not move into M^s_a and $|\vec{V_a}| > |\overline{V}_a| + |\overline{V}_b|$, then $\pi^s \rightarrow \pi^t$ is not a min-edit operation.

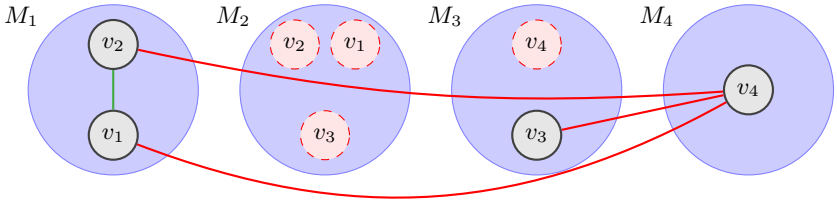
(b) If $\vec{V_b}$ moves into M^s_a and $|\vec{V_a}| + |\vec{V_b}| > |\overline{V}_a| + |\overline{V}_b|$, then $\pi^s \rightarrow \pi^t$ is not a min-edit operation.

Proof. Let us first handle the condition *(3a)*, which is illustrated in Figure 3. When this condition is satisfied, keeping $\vec{V_a}$ in M^s_a and rather moving $\overline{V}_a$ (resp. $\overline{V}_b$) into M^s_b (resp. M^s_a) results in an r'-edit operation with $r' < r$. In the case where condition *(3b)* is satisfied, keeping $\vec{V_b}$ in M^s_b while moving $\overline{V}_a$ and $\overline{V}_b$ results in an r'-edit operation with $r' < r$. Let us take the following example to illustrate Property 3.b. Two modules exchange all of their vertices with each other, such that $\vec{V_a} = M^s_a$,

(a) Source partition P^s.

(b) Target partition P^t.

Figure 3: Illustrative example for Property 3a, where vertices in $\overrightarrow{V_a}$ (resp. $\overrightarrow{V_b}$) move from their source module M_a^s (resp. M_b^s) (Fig. 3a) into target module M_b^s (resp. M_c^s, which is purposely not shown) (Fig. 3b). $\overline{V}_a$ and $\overline{V}_b$ represent the non-moving vertices.



(a) An illustration of an r-edit operation $\pi^s \to \pi^t$ with ${}^{s}\overrightarrow{V}^{t} = \{v_1, v_2, v_3, v_4\}$. The membership vector π^s associated with partition P^s is defined by $\pi^s = (1, 1, 3, 4)$, whereas $\pi^t = (2, 2, 2, 3)$. Positive (resp. negative) edges between moving vertices are drawn in green (resp. red).

(b) The corresponding interaction graph $\widetilde{G}[{}^{s}\overrightarrow{V}^{t}]$ of the r-edit operation illustrated in Fig. 4a

Figure 4: Illustrative example regarding the definition of an interaction subgraph, here $\widetilde{G}[{}^{s}\overrightarrow{V}^{t}]$.

$\overline{V}_a = \emptyset$, $\overrightarrow{V_b} = M_b^s$ and $\overline{V}_b = \emptyset$. In the end, we obtain the same two modules. Since the inequality $|\overrightarrow{V_a}| + |\overrightarrow{V_b}| > |\overline{V}_a| + |\overline{V}_b|$ holds in this particular scenario, this is not a min-edit operation. □

6.2 Decomposable Edit Operation

The next set of properties is related to the decomposability of an r-edit operation $\pi^s \to \pi^t$. We assume that $I(P^s) = I(P^t)$, which is in line with our objective of enumerating all optimal solutions of a given graph G.

Definition 4 (Decomposable edit operation)**.** *We consider an r-edit operation $\pi^s \to \pi^t$ as decomposable if there is an intermediate membership vector $\pi^{t'}$, associated with a partition $P^{t'}$, between π^s and π^t such that:*

1. $I(P^s) = I(P^t) = I(P^{t'})$;
2. ${}^{s}\overrightarrow{V}^{t'} \subset {}^{s}\overrightarrow{V}^{t}$; *and*
3. $\pi^{t'}(u) = \pi^t(u)$ *for each* $u \in {}^{s}\overrightarrow{V}^{t'}$.

Property 5 (Atomic edit operation)**.** *We consider that an r-edit operation $\pi^s \to \pi^t$ is atomic if the condition $I(P^s) < I(P^{t'})$ is satisfied for any r'-edit operation $P^s \to P^{t'}$ with $r' < r$ satisfying (2) and (3) in Definition 4.*

Proof. We assume that there is an r'-edit operation such that $I(P^s) = I(P^{t'})$. Then, we can construct an r''-edit operation $\pi^{t'} \to \pi^t$ satisfying (1) in Definition 4 and $\pi^{t'}(u) = \pi^t(u)$ for each $u \in {}^{s}\overrightarrow{V}^{t} \setminus {}^{s}\overrightarrow{V}^{t'}$. □

Atomicity requires three types of connectivity conditions. We state the first one in the following property.

Property 6 (Edge connectivity)**.** *In an atomic edit operation $\pi^s \to \pi^t$, $G[{}^s\vec{V}^t]$ is a single connected component.*

Proof. Let ${}^s\vec{V}^{t'}$ and ${}^{t'}\vec{V}^t$ be any two proper subsets of ${}^s\vec{V}^t$, such that ${}^s\vec{V}^{t'} \cap {}^{t'}\vec{V}^t = \emptyset$ and $E({}^s\vec{V}^{t'}, {}^{t'}\vec{V}^t) = \emptyset$. Then, we can construct with ${}^s\vec{V}^{t'}$ an r'-edit operation satisfying Definition 4. □

Nevertheless, $G[{}^s\vec{V}^t]$ being connected is not a sufficient condition. The next property states that the signs of edges between moving vertices play a key role with respect to atomicity. In the following, we assume that $G[{}^s\vec{V}^t]$ is connected.

Property 7 (Spurious edge connectivity)**.** *Each condition in the following is a sufficient condition for $\pi^s \to \pi^t$ to be a decomposable edit operation:*

(a) Let $S \subseteq {}^s\vec{V}^t$ be a subset such that $|\pi^s(S)| = 1$ and $E(S, \vec{V}^s_{\pi^s(S)}) \neq \emptyset$. Let $G' = ({}^s\vec{V}^t, E', w)$, where $E' = E \setminus E(S, \vec{V}^s_{\pi^s(S)})$ and $w(e) = 1$ for each $e \in E'$. If $\Omega(S, \vec{V}^s_{\pi^s(S)}) = 0$ and G' is not connected, then $\pi^s \to \pi^t$ is decomposable.

(b) Let $S \subseteq {}^s\vec{V}^t$ be a subset such that $|\pi^t(S)| = 1$ and $E(S, \vec{V}^t_{\pi^t(S)}) \neq \emptyset$. Let $G' = ({}^s\vec{V}^t, E', w)$, where $E' = E \setminus E(S, \vec{V}^t_{\pi^t(S)})$ and $w(e) = 1$ for each $e \in E'$. If $\Omega(S, \vec{V}^t_{\pi^t(S)}) = 0$ and G' is not connected, then $\pi^s \to \pi^t$ is decomposable.

Proof. Straightforwardly, we can imagine S as a contracted vertex v in both cases, such that the vertices in S and the edges between them are replaced by a single vertex v. Since $\Omega(S, \vec{V}^s_{\pi^s(S)}) = 0$ ($\Omega(S, \vec{V}^t_{\pi^t(S)}) = 0$), the edges between v and $\vec{V}^s_{\pi^s(v)}$ (resp. $\vec{V}^t_{\pi^t(v)}$) do not play a role (i.e., as if they did not exist). Therefore, the proof is the same as for Property 6, with subsets S and ${}^s\vec{V}^t \setminus S$. □

The last connectivity condition allows to verify if a moving vertex depends on the simultaneous movement of a subset of other vertices in ${}^s\vec{V}^t$. It can be checked through the concept of *interaction subgraph* $\widetilde{G}[{}^s\vec{V}^t]$. We define the interaction subgraph defined for the edit operation $\pi^s \to \pi^t$ as $\widetilde{G}[{}^s\vec{V}^t] = ({}^s\vec{V}^t, \widetilde{E})$, where $\widetilde{E} = \{(u,v) \mid u, v \in {}^s\vec{V}^t \text{and } (u,v) \in E \text{ and } (\pi^s(u) = \pi^s(v) \vee \pi^t(u) = \pi^s(v) \vee \pi^s(u) = \pi^t(v) \vee \pi^t(u) = \pi^t(v))\}$. Its extraction from its original graph G is illustrated in Figure 4.

Property 8 (Interaction connectivity)**.** *Consider an atomic edit operation $\pi^s \to \pi^t$. The interaction subgraph $\widetilde{G}[{}^s\vec{V}^t]$ is a single connected component.*

Proof. Assume that $\widetilde{G}[{}^s\vec{V}^t]$ is not connected (e.g. Figure 4). This implies that there is a subset ${}^s\vec{V}^{t'} \subseteq {}^s\vec{V}^t$, such that $\widetilde{E}({}^s\vec{V}^{t'}, {}^s\vec{V}^t \setminus {}^s\vec{V}^{t'}) = \emptyset$. Let $\pi^s \to \pi^{t'}$ be an edit operation with moving vertex set ${}^s\vec{V}^{t'}$, which satisfies (2)-(3) in Definition 4. When moving a vertex $u \in {}^s\vec{V}^{t'}$ from $M^s_{\pi^s(u)}$ to $M^s_{\pi^{t'}(u)}$ the contribution of vertex u to the imbalance is

$$\frac{\Omega(\{u\}, \vec{V}^s_{\pi^s(u)}) - \Omega(\{u\}, \vec{V}^s_{\pi^{t'}(u)}) + \Omega(\{u\}, \vec{V}^{t'}_{\pi^{t'}(u)}) - \Omega(\{u\}, \vec{V}^{t'}_{\pi^s(u)})}{2}. \tag{11}$$

From the definition of $\widetilde{E}$ and from $\widetilde{E}({}^s\vec{V}^{t'}, {}^s\vec{V}^t \setminus {}^s\vec{V}^{t'}) = \emptyset$, there is no vertex $v \in {}^s\vec{V}^t \setminus {}^s\vec{V}^{t'}$ which contributes to the contribution of vertex u, as shown in Equation (11). Therefore, Equation (1) in Definition 4 is also satisfied for $\pi^s \to \pi^{t'}$. □

For the sake of simplicity, in Algorithm 4, we define the function *intAtomic*(G, π^s, ${}^s\vec{V}^t$) which is used to indicate whether $\pi^s \to \pi^t$ satisfies Properties 6 and 7a. Notice that we do so when the target modules of ${}^s\vec{V}^t$ are not known. Likewise, in Algorithm 5, we define the function *extAtomic*(G, π^s, π^t, ${}^s\vec{V}^t$), which indicates whether $\pi^s \to \pi^t$ satisfies Properties 3, 5, 7b and 8. Notice that we do so when the target modules of ${}^s\vec{V}^t$ are partially or completely known, i.e., in lines 10, 20 and 26 of Algorithm 3.

6.3 Multiple Vertex Moves between Optima (MVMO) Property

In this section, we introduce our last family of properties, which rely on the assumption that both partitions associated to an edit operation are optimal.

Property 9 (MVMO on weighted signed networks)**.** *Consider an atomic edit operation $\pi^s \rightarrow \pi^t$ with $r > 1$, where P^s and P^t are optimal. The following property holds for each $u \in {}^{s}\vec{V}^{t}$:*

$$\gamma_u^{left} > \gamma_u^{right}, \tag{12}$$

where

$$\gamma_u^{left} = \Omega(\{u\}, \vec{V}^s_{\pi^s(u)}) - \Omega(\{u\}, \vec{V}^s_{\pi^t(u)}), \tag{13}$$

$$\gamma_u^{right} = -\Omega(\{u\}, \vec{V}^t_{\pi^t(u)}) + \Omega(\{u\}, \vec{V}^t_{\pi^s(u)}). \tag{14}$$

$$\tag{15}$$

Proof. Let $\pi^s \rightarrow \pi^t$ be an atomic min-r-edit operation applied on P_s to obtain P_t. First, we recall a simple condition satisfied by any optimal partition for the CC problem [13] when moving a single vertex, i.e. when $r = 1$. Given an optimal partition P_s, the placement of any vertex u in its module $M^s_{\pi^s(u)}$ is also optimal with respect to any other modules. This means that any vertex u maximizes (resp. minimizes) its number of positive (resp. negative) edges in module $M^s_{\pi^s(u)}$ of P_s, so that moving u to any other module does not improve the objective function value. We can write this condition as

$$\Omega(\{u\}, M^s_{\pi^s(u)}) \geq \Omega(\{u\}, M^s_{\pi^t(u)}). \tag{16}$$

Now, when we move more than one vertex in $\pi^s \rightarrow \pi^t$, i.e. if $r > 1$, this condition becomes

$$\Omega(\{u\}, M^s_{\pi^s(u)}) > \Omega(\{u\}, M^s_{\pi^t(u)}), \tag{17}$$

for each $u \in {}^{s}\vec{V}^{t}$, due to the atomicity assumption. Next, in order to take into account the existence of other moving vertices in the source and target modules of vertex u, Equation (17) can be rewritten as $\gamma_u^{left} > {}^{s}\vec{\Delta}^{t}_u$, where ${}^{s}\vec{\Delta}^{t}_u = \Omega(\{u\}, M^s_{\pi^t(u)} \setminus \vec{V}^s_{\pi^t(u)}) - \Omega(\{u\}, M^s_{\pi^s(u)} \setminus \vec{V}^s_{\pi^s(u)})$. Similarly, considering the atomic min-r-edit operation $P^t \rightarrow P^s$, we have also $-{}^{t}\vec{\Delta}^{s}_u > \gamma_u^{right}$, where ${}^{t}\vec{\Delta}^{s}_u = \Omega(\{u\}, M^t_{\pi^s(u)} \setminus \vec{V}^t_{\pi^s(u)}) - \Omega(\{u\}, M^t_{\pi^t(u)} \setminus \vec{V}^t_{\pi^t(u)})$. Since both ${}^{s}\vec{\Delta}^{t}_u$ and ${}^{t}\vec{\Delta}^{s}_u$ are defined on relations of vertex u with non-moving vertices in P^s and P^t, we have ${}^{s}\vec{\Delta}^{t}_u = -{}^{t}\vec{\Delta}^{s}_u$. Finally, by rewriting the previous equations we obtain $\gamma_u^{left} > {}^{s}\vec{\Delta}^{t}_u = -{}^{t}\vec{\Delta}^{s}_u > \gamma_u^{right}$. □

Corollary 10 (MVMO on unweighted signed networks)**.** *The inequality $\gamma_u^{left} - \gamma_u^{right} \geq 2$ holds for each vertex u, on top of Property 9.*

Proof. The proof is straightforward from the proof of Property 9. □

Figure 5 can be used to illustrate Property 9 and Corollary 10. For instance, for vertex v_1, the equation in Property 9 becomes

$$a_{12} - a_{13} > a_{15} - a_{12} - a_{14}. \tag{18}$$

Similarly, for vertex v_3, the equation in Property 9 becomes

$$-a_{34} - a_{35} > a_{13} + a_{23} + a_{34}. \tag{19}$$

6.4 Tractable Cases of the MVMO Property

Notice that the MVMO property requires π^s and π^t to be known, which makes this property not very useful for computational purposes (in Algorithm 3). In this section, we analyze the MVMO property when only partial information of π^t is known (lines 10 and 20 in Algorithm 3).

We start by analysing some special relational patterns for 2-edit and 3-edit operations. Lemma 11 focuses on 2-edit operations. Recall that $\widetilde{E} = \{(u,v) \in E \mid u, v \in {}^{s}\vec{V}^{t} \ \wedge \ (\pi^s(u) = \pi^s(v) \vee \pi^t(u) = \pi^s(v) \vee \pi^s(u) = \pi^t(v) \vee \pi^t(u) = \pi^t(v))\}$.

Lemma 11 (MVMO 2-edit)**.** *Consider an atomic 2-edit operation $\pi^s \rightarrow \pi^t$, where ${}^{s}\vec{V}^{t} = \{u, v\}$. Since it is an atomic edit operation, we have $(u,v) \in \widetilde{E}$. Then*

1. *If $(\pi^s(u) = \pi^s(v)) \vee (\pi^t(u) = \pi^t(v))$, then $a_{uv} > 0$.*

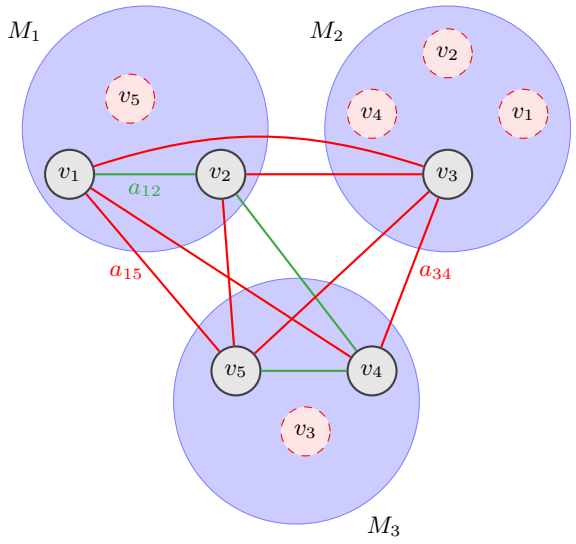


Figure 5: Illustrative example, where five vertices v_1, v_2, v_3, v_4 and v_5 move from their source modules to their target ones. Dashed circles indicate the moving vertices, when they are in their target modules after the edit operation. Note that we do not show non-moving vertices for the sake of simplicity. Positive (resp. negative) edges between moving vertices are drawn in green (resp. red).

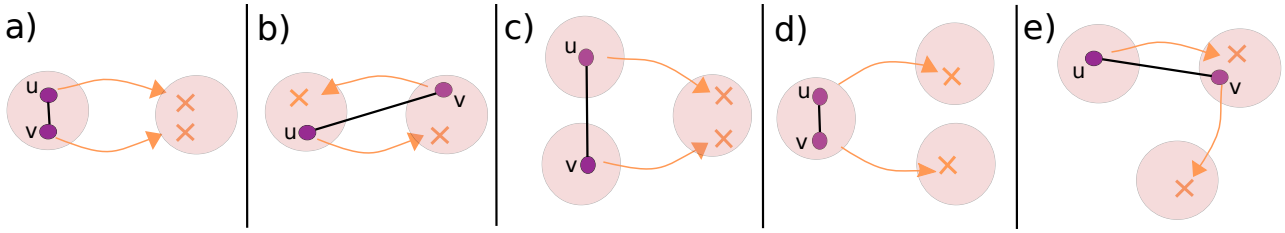


Figure 6: All 2-edit operation scenarios, where $(u, v) \in \widetilde{E}$. Note that only scenarios (6a) and (6b) are atomic 2-edit operations, with $a_{uv} > 0$ and $a_{uv} < 0$, respectively.

2. *If* $(\pi^s(u) = \pi^t(v)) \vee (\pi^t(u) = \pi^s(v))$, *then* $a_{uv} < 0$.

Proof. From Property 6, we have $(u, v) \in E$. In this proof, we use an exhausting strategy: moving vertices u and v, with $(u, v) \in \widetilde{E}$, can be in one of the five scenarios presented in Figure 6. Since Corollary 10 holds for each vertex $u \in {}^{s}\vec{V}^{t}$, only scenarios (6a) and (6b) are atomic. Therefore, $a_{uv} > 0$ for scenario (a) whereas $a_{uv} < 0$ for scenario (b):

a) We have $(\gamma_u^{left} = a_{uv}) > (\gamma_u^{right} = -a_{uv})$ and $(\gamma_v^{left} = a_{uv}) > (\gamma_v^{right} = -a_{uv})$. We see that a_{uv} must be positive. Since Corollary 10 is satisfied with $a_{uv} > 0$, it is atomic.

b) We have $(\gamma_u^{left} = -a_{uv}) > (\gamma_u^{right} = a_{uv})$ and $(\gamma_v^{left} = -a_{uv}) > (\gamma_v^{right} = a_{uv})$. We see that a_{uv} must be negative. Since Corollary 10 is satisfied with $a_{uv} < 0$, it is atomic.

c) We have $(\gamma_u^{left} = 0) > (\gamma_u^{right} = -a_{uv})$ and $(\gamma_v^{left} = 0) > (\gamma_v^{right} = -a_{uv})$. We see that a_{uv} must be positive and this satisfies Property 9. However, Corollary 10 is not satisfied, which makes this edit operation decomposable.

d) We have $(\gamma_u^{left} = a_{uv}) > (\gamma_u^{right} = 0)$ and $(\gamma_v^{left} = a_{uv}) > (\gamma_v^{right} = 0)$. We see that a_{uv} must be positive and this satisfies Property 9. However, Corollary 10 is not satisfied, which makes this edit operation decomposable.

e) We have $(\gamma_u^{left} = -a_{uv}) > (\gamma_u^{right} = 0)$ and $(\gamma_v^{left} = 0) > (\gamma_v^{right} = -a_{uv})$. We see that a_{uv} must be negative and this satisfies Property 9. However, Corollary 10 is not satisfied, which makes this edit operation decomposable.

□

Next, we focus on 3-edit operations. We verify some cases in which Lemma 11 is also valid.

Lemma 12 (MVMO 3-edit)**.** *Consider an atomic r-edit operation $\pi^s \to \pi^t$ with $r = 3$, where ${}^{s}\vec{V}^{t} = \{u, v, z\}$. Since it is an atomic edit operation, we have $(u, v), (u, z), (v, z) \in \widetilde{E}$. Lemma 11 holds true for each pair (u, v) of vertices, with two exceptions:*

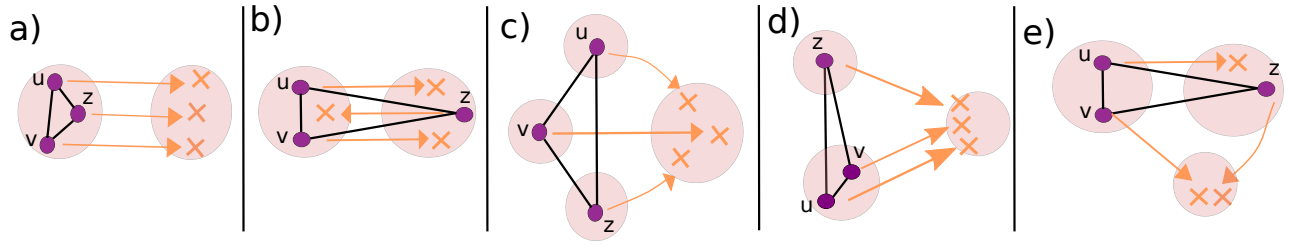


Figure 7: Five of the possible 3-edit operation scenarios. The full list can be found in the Appendix (Figure 12). In this figure, all edit operations are atomic.

1. *all vertices in* $\overset{s\rightarrow t}{V}$ *are in the same source module and are moved into the same target module (Figure 7a),*
2. *only two of* $\overset{s\rightarrow t}{V}$ *are in the same source module and are moved into the source module of the third moving vertex. Reciprocally, the third moving vertex moves into the source module of the others' source module (Figure 7b).*

Proof. Without these two exceptions, $G[\overset{s\rightarrow t}{V}]$ forms a triangle. Nevertheless, in these two exceptions, $G[\overset{s\rightarrow t}{V}]$ can form a path, and, as will see next, this path ensures that moving vertices from the same source module are positively connected. Similar to Lemma 11, the proof is based on all possible 17 scenarios of three moving vertices with $(u,v),(u,z),(v,z) \in \widetilde{E}$. Some of those scenarios are shown in Figure 7, and we provide the full list in the Appendix (Figure 12). The proof is straightforward when one adapts Corollary 10 to those scenarios. We verify below only those in Figure 7. The full list of verifications is found in Appendix C.1.

a) We have $(\gamma_u^{left} = a_{uv} + a_{uz}) > (\gamma_u^{right} = -a_{uv} - a_{uz})$, $(\gamma_v^{left} = a_{uv} + a_{vz}) > (\gamma_v^{right} = -a_{uv} - a_{vz})$ and $(\gamma_z^{left} = a_{uz} + a_{vz}) > (\gamma_z^{right} = -a_{uz} - a_{vz})$. We see that a_{uv}, a_{uz} and a_{vz} cannot be negative. Since Corollary 10 is satisfied with a positive path formed in $G[\overset{s\rightarrow t}{V}]$ for each vertex $u \in \overset{s\rightarrow t}{V}$, it is atomic.

b) We have $(\gamma_u^{left} = a_{uv} - a_{uz}) > (\gamma_u^{right} = -a_{uv} + a_{uz})$, $(\gamma_v^{left} = a_{uv} - a_{vz}) > (\gamma_v^{right} = -a_{uv} + a_{vz})$ and $(\gamma_z^{left} = -a_{uz} - a_{vz}) > (\gamma_z^{right} = a_{uz} + a_{vz})$. We see that a_{uv} (resp. a_{uz} and a_{vz}) cannot be negative (resp. positive). Since Corollary 10 is satisfied with a path formed in $G[\overset{s\rightarrow t}{V}]$, it is atomic.

c) We have $(\gamma_u^{left} = 0) > (\gamma_u^{right} = -a_{uv} - a_{uz})$, $(\gamma_v^{left} = 0) > (\gamma_v^{right} = -a_{uv} - a_{vz})$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = -a_{uz} - a_{vz})$. We see that a_{uv}, a_{uz} and a_{vz} must be positive. Since Corollary 10 is satisfied with a positive triangle formed in $G[\overset{s\rightarrow t}{V}]$, it is atomic.

d) We have $(\gamma_u^{left} = a_{uv}) > (\gamma_u^{right} = -a_{uv} - a_{uz})$, $(\gamma_v^{left} = a_{uv}) > (\gamma_v^{right} = -a_{uv} - a_{vz})$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = -a_{uz} - a_{vz})$. We see that a_{uv}, a_{uz} and a_{vz} must be positive. Since Corollary 10 is satisfied with a positive triangle formed in $G[\overset{s\rightarrow t}{V}]$, it is atomic.

e) We have $(\gamma_u^{left} = a_{uv} - a_{uz}) > (\gamma_u^{right} = 0)$, $(\gamma_v^{left} = a_{uv}) > (\gamma_v^{right} = -a_{vz})$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = a_{uz} - a_{vz})$. We see that a_{uv} and a_{vz} (resp. a_{uz}) must be positive (resp. negative). Since Corollary 10 is satisfied with a triangle formed in $G[\overset{s\rightarrow t}{V}]$, it is atomic.

□

Unlike Lemma 12, we cannot completely generalize Lemma 11 for more than three moving vertices. Nevertheless, there are some circumstances, where Lemma 11 is still valid for a subset of $\overset{s\rightarrow t}{V}$, and this is formalized in Lemma 13.

Lemma 13 (MVMO r-edit)**.** *Consider an atomic r-edit operation $\pi^s \rightarrow \pi^t$ with $r \geq 4$ and a vertex $u \in \overset{s\rightarrow t}{V}$. If $2 \leq |u \cup \overset{s\rightarrow t}{V}_u| \leq 3$, Lemma 11 holds true for each pair (u,v) with $v \in \overset{s\rightarrow t}{V}_u$.*

Proof. The condition of $2 \leq |u \cup \overset{s\rightarrow t}{V}_u| \leq 3$ ensures that subset $u \cup \overset{s\rightarrow t}{V}_u$ represents one of the scenarios in 2- or 3-edit operation. We note that two exceptional scenarios mentioned in Lemma 12 are not of interest here, because they necessarily satisfy the definition of atomic edit operation, hence $|u \cup \overset{s\rightarrow t}{V}_u| > 3$. □

For the sake of simplicity, in Algorithm 4, we define the function *intMVMO*(G, π^s, $\overset{s\rightarrow t}{V}$) which indicates whether $\pi^s \rightarrow \pi^t$ satisfies Lemmas 11.1 and 12, when the target modules of $\overset{s\rightarrow t}{V}$ are not known. Likewise, in Algorithm 5 we define the function *extMVMO*(G, π^s, π^t, $\overset{s\rightarrow t}{V}$), which indicates

whether $\pi^s \rightarrow \pi^t$ satisfies Lemmas 11.2, 12 and 13, when the target modules of ${}^{s}\vec{V}^{t}$ are partially or completely known (10, 20 and 26 of Algorithm 3).

7 Dataset

This section is dedicated to the description of the dataset used in our experiments. As mentioned in the introduction, in this article we focus on *unweighted* graphs as we have done in [6]. Application-wise, unweighted signed networks fit certain modeling situations and methodological choices. Indeed, in some works, binary values better represent the studied relations (e.g. alliance/conflict between countries in international relationships [21]), or the authors prefer to use such values for practical reasons (e.g. limited or unreliable information [22]).

We conduct our experiments on two datasets of random signed networks. All these data as well as the corresponding optimal solutions are publicly available online[1].

Dataset1: We generate these networks through the random signed network generator proposed in our previous work [6], which is publicly available online[2]. For complete unweighted signed networks, this model relies on only three parameters: n (number of vertices), ℓ_0 (initial number of modules) and q_m (proportion of misplaced edges, i.e. edges meant to be frustrated by construction). Moreover, we make the assumption that the proportion of misplaced edges is the same inside and between the modules. When it comes to incomplete unweighted signed networks, we introduce two more parameters, which are the density d of the graph and the proportion q_{neg} of the negative edges. The last parameter q_{neg} is defined as $|E^-|/|E|$ and allows controlling the ratio of positive to negative edges. For complete unweighted signed networks with $d = 1$, we generate 20 replications for parameter values $\ell_0 = 3$, $n \in \{32, 36, 40, 45, 50\}$ and $q_m = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$. In these networks, the value of q_{neg} with the considered parameters is approximately 0.7. For incomplete unweighted signed networks with $d = \{0.25, 0.50\}$, we generate 20 replications for parameter values $\ell_0 = 3$, $n \in \{32, 36, 40\}$, $q_m = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$ and $q_{neg} = \{0.3, 0.5, 0.7\}$. In total, we produce 600 and $1{,}080$ instances for complete and incomplete networks, respectively, which makes a total of $1{,}680$ instances. We use this dataset in Section 8.2.

Dataset2: This dataset is different than *Dataset1* in that the optimal solution for a generated network is known by construction. This allows to consider relatively large graph orders n without being limited by long running time of an exact partitioning method. For given n, d and ℓ_0, we first create a perfectly structurally balanced (i.e., internally positive and externally negative) signed network with a built-in module structure. Clearly, the underlying module structure constitutes the optimal partition. Then, in order to take into account different positive to negative ratio values for internal and external edges we generate several signed networks by perturbing the initial signed network without affecting its underlying optimal partition, thanks to its definition of stability range [41]. We generate signed networks with parameter values $n \in \{30, 40, 50, 60, 70, 90\}$, $d \in \{0.25, 1.00\}$ and $\ell_0 = \{2, 4, 6\}$ through our random signed network generator, which is publicly available online[3]. In total, we produce 214 and 184 instances for complete and incomplete networks, respectively, which makes a total of 398 instances. In each generated network, the associated optimal solution corresponds to the planted partition defined for the corresponding network without perturbation. We use this dataset only for Section 8.1.

Dataset3: In addition to artificial graphs, we also considered 8 sparse real-world networks from two sources: four networks from *Correlated of War (CoW)* [43] and four biological networks retrieved from Samin Aref's Figshare repository[4], see [3] for more details. For each real-world signed network G, we apply a preprocessing step by retrieving the largest connected component defined in G^+, for computational purposes.

8 Results

We now assess the performance of our enumeration method $EnumCC(r_{max})$ when generating the space of optimal solutions to the CC problem. We first investigate the efficiency of the pruning conditions used in Algorithms 4 and 5 based on the MVMO property (Section 8.1). Then, we proceed with the evaluation of $EnumCC(r_{max})$, which includes all the pruning strategies used in the recurrent neighborhood search. We compare our method with OneTreeCC(), the best enumeration method available in the literature (Section 8.2).

[1] DOI: 10.6084/m9.figshare.15043911
[2] https://github.com/CompNet/SignedBenchmark.
[3] https://github.com/CompNet/SignedStabilityBenchmark.
[4] DOI: 10.6084/m9.figshare.5700832.v5

8.1 Evaluation of the MVMO-based Pruning Strategies

The MVMO property has an important role in EnumCC, due to its ability to prune unfeasible edit operations in several parts of $CoNS(r)$. To be able to consider relatively large graph orders n without being limited by the long running time of an exact partitioning method, we evaluate its performance based on *Dataset2*. The complete results[1] as well as our source code[3] are publicly available online.

We apply $CoNS(r)$ *with* and *without* the MVMO property onto all generated networks with $r \in \{3, 4\}$. The version *with* MVMO property refers to the method described in Section 4, whereas the version *without* it is obtained by removing this property from Algorithms 4 and 5. Due to space considerations, Figure 8 illustrates only the results related to 4-edit distance. The results obtained for $d = 0.25$ and $d = 1.00$ are shown in separated subfigures. In each subfigure, the x-axis represents the graph order n, and execution times (in seconds) are shown on the log-scaled y-axis. The solid (resp. dashed) plot lines correspond to $CoNS(r)$ with (resp. without) the MVMO property. Each plot line corresponds to a specific value of ℓ_0 and is represented with a specific color. Each shaded region depicts a range of execution times around the plot line of the same color, based on the corresponding initial signed network and its perturbed versions.

The first thing to notice is how graph density affects the execution times with increasing values of n. Indeed, the computational costs with and without the MVMO property are much larger with $d = 1.00$ than those with $d = 0.25$. Second, substantially increasing n affects the execution times without the MVMO property, whereas including the MVMO property allows to better handle this effect. Indeed, considering $d = 0.25$ and the 4-edit distance, we observe that average execution times without the MVMO property are approximately ℓ_0 times larger than with the MVMO property. Finally, including the MVMO property also allows to better handle increasing values of ℓ_0. Indeed, when $n = 70$ and $d = 0.25$, the difference of average execution times between $\ell_0 = 2$ and $\ell_0 = 6$ without (resp. with) the MVMO property is 3.8s (resp. 0.6s) with the 3-edit distance, and 595s (resp. 63s) with the 4-edit distance.

To conclude this part, the MVMO property makes a substantial improvement on $CoNS(r)$. This improvement is apparent even with small values of n. Note that, even a small improvement ($1s$ or $2s$) can make a clear difference in terms of execution time for a solution space with 100 or more solutions, since $CoNS(r)$ is repeated for each solution. Nevertheless, our results suggest that $CoNS(4)$ should not be used for computational purposes, even with the MVMO property. In the following, we investigate if the improvements brought by the MVMO property and other pruning strategies allow $EnumCC(r_{max})$ to compete with OneTreeCC().

8.2 Evaluation of EnumCC

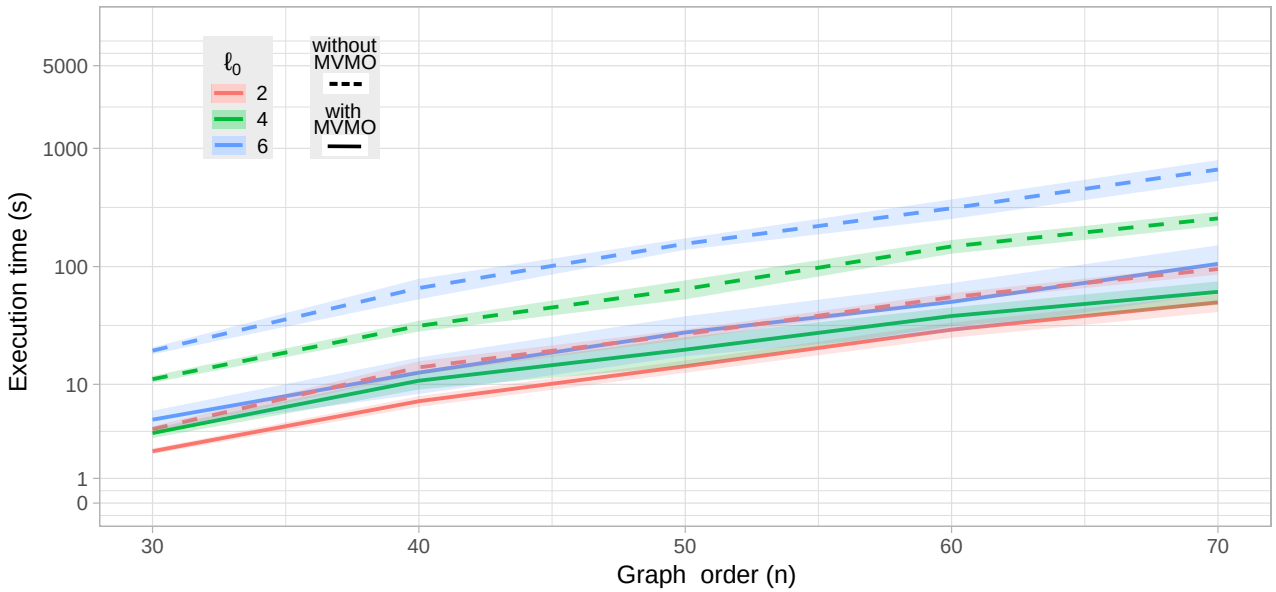
We now compare the results of $EnumCC(r_{max})$ against OneTreeCC() based on the synthetic signed networks from *Dataset1* and the real-world signed networks from *Dataset3*. As shown in Section 8.1, since $CoNS(4)$ takes a substantial time, we rather apply EnumCC(3). We run both methods with a time limit of 12h and a limit of $50,000$ on the number of optimal solutions found. We first evaluate the results from *Dataset1* for several values of density d with a fixed value of n (Section 8.2.1) and several values of n with a fixed value of d (Section 8.2.2), then pass to the evaluation of the results from *Dataset3* (Section 8.2.3).

Regarding the synthetic graphs, we present a selection of the most relevant results in Figures 9a–10c, for $d \in \{0.25, 1.00\}$. The complete results[1] as well as our source code[5,6] are available online, though. We first describe these plots generically here, before interpreting them. In these figures, there are 3 subfigures. Each subfigure corresponds to a specific value of q_m (hence, a specific parameter set), and displays the difference of execution times between EnumCC(3) and OneTreeCC() (i.e., EnumCC(3) minus OneTreeCC()), represented on the log-scaled y-axis of the plots. When such difference takes a negative value, this means that our proposed method runs faster than OneTreeCC(). The set of 20 graphs generated for each parameter set are indexed and shown on the x-axis. Finally, to guide our discussion, we show for each graph the maximal number of solutions found by the method(s) within a time limit and the number of jumps related to EnumCC(3), i.e. $n_{jump}(EnumCC(3))$, where the latter is shown in parentheses.

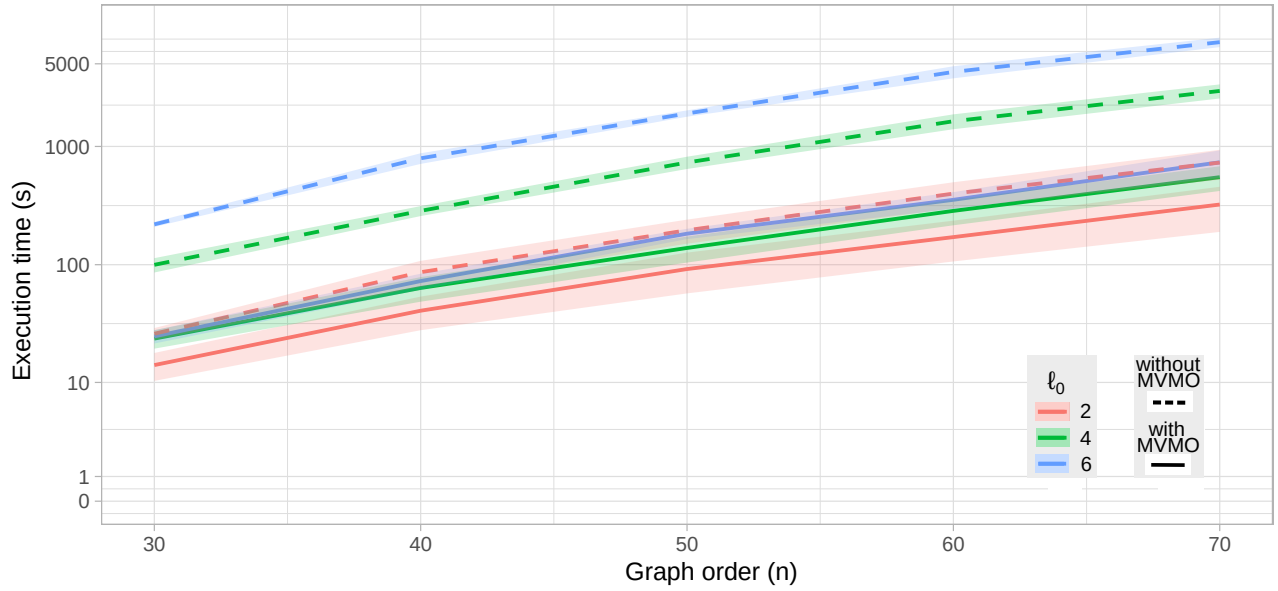
8.2.1 Evaluation of EnumCC by Density

In this section, the results are for $n = 36$ and $d \in \{0.25, 0.5, 1.0\}$. They are representative enough of the results obtained with other values of n. We first consider the $d = 0.25$ results, shown in Figure 9.

[5] https://github.com/CompNet/Sosocc
[6] https://github.com/CompNet/EnumCC



(a) Instances with $d = 0.25$.



(b) Instances with $d = 1.00$.

Figure 8: Benchmark results for *CoNS*($r = 4$) with vs. without MVMO-based pruning for $d = 0.25$ (8a) and $d = 1.00$ (8b). The x-axis represents graph order n, and execution times (in seconds) are shown in the log-scaled y-axis.

The first thing that we notice is how the performance is affected by q_{neg}, independently of q_m. Indeed, we first see for $q_{neg} = 0.3$ that EnumCC(3) runs faster than OneTreeCC() in the overwhelming majority of instances. Then, for $q_{neg} = 0.5$, OneTreeCC() increases the number of instances, where it runs faster, but the dominance of EnumCC(3) is still preserved to a lesser extent. Finally, for $q_{neg} = 0.7$, OneTreeCC() dominates EnumCC(3) on almost all instances.

We observe that increasing q_{neg} essentially results in the emergence of three features, which advantages OneTreeCC() over EnumCC(3). The first one is large values of $n_{jump}(EnumCC(3))$. Since a branch-and-bound tree needs to be built from scratch, each additional jump has an extra cost for EnumCC(3) in terms of execution time. We observe that the values of $n_{jump}(EnumCC(3))$ are relatively larger for mainly $q_{neg} = 0.5$ and $q_m = \{0.4, 0.6\}$. This shows that increasing q_m is likely to increase the dissimilarity between solutions, a fact that we have already pointed out in [6], for complete signed networks.

The second feature is a very large size of the solution space. The results show that OneTreeCC() does a much better job in this case. Indeed, when $q_{neg} = 0.7$, OneTreeCC() can solve instances associated with a very large number of solutions (e.g. $50,000$) within 5 minutes, whereas the same process often takes several hours for EnumCC(3). Nevertheless, such an extreme case happens only with some specific graph topology, as in $q_{neg} = 0.7$. Indeed, in the random network generation, increasing q_{neg} with a low graph density is more likely to produce instances having a large number of vertices with only few positive edges. Then, these vertices are often placed at the periphery of modules in the solutions, i.e. they can easily change their module from one solution to another. OneTreeCC() seems to handle well these vertices in the enumeration process, thanks to the mathematical modeling.

Finally, the third feature, complementary to the previous one, is that instances having vertices with only few positive edges are often easier to solve, so that an initial optimal solution is often found at root relaxation, before passing to branch-and-bound. This usually results in a branch-and-bound tree with fewer branches for enumerating alternative optimal solutions in OneTreeCC(). It seems that this substantially advantages OneTreeCC() over EnumCC(3), when there are many optimal solutions to enumerate.

For space considerations, we do not present the results for $d \in \{0.5, 1.0\}$. Note that for these values, we do not observe any extreme case with a very large number of solutions (as happens with $d = 0.25$ and $q_{neg} = 0.7$). Indeed, the maximum number of solutions for those instances is $2,455$. This is probably because it is not as easy to break the underlying partition structure when graph density is large. This fact seems to advantage EnumCC(3) over OneTreeCC(). Indeed, the dominance of EnumCC(3) persists for $q_{neg} \in \{0.3, 0.5\}$ and even better than those for $d = 0.25$ (see Table 2). This holds for $q_{neg} = 0.7$, too. OneTreeCC() outperforms EnumCC(3) only for $q_m = 0.1$, whereas this was true for all values of q_m for $d = 0.25$. These results confirm our previous observation: OneTreeCC() outperforms EnumCC(3) on instances with specific graph topology, where low density and large proportion of negative edges produce a very large number of solutions. In the other cases, EnumCC(3) performs much better. In the following, we study whether increasing graph order n affects these observations.

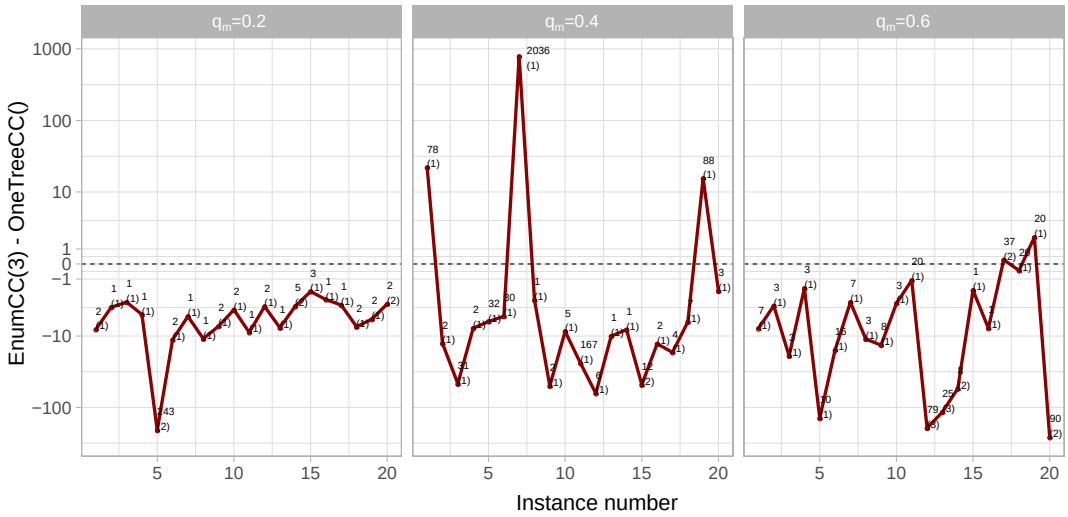
8.2.2 Evaluation of EnumCC by Graph Order

In this section, we analyze the effect of increasing the graph order, for $n \in \{40, 45, 50\}$. To do so, we focus only on instances with $d = 1.0$ in order to reduce the total number of instances, hence the processing time of our analysis. Note that q_{neg} approximately equals 0.7 in these instances. The corresponding results are shown in Figure 10. Overall, we see that the observations we made for $n = 36$ and $d \in \{0.5, 1.0\}$ in Section 8.2.1 are still valid when increasing n: EnumCC(3) performs much better. Furthermore, unlike the results obtained with $d = 0.25$, EnumCC(3) handles instances with a large value of $n_{jump}(EnumCC(3))$ much better (when we exclude some exceptional instances with more than 10 jumps), and runs faster than OneTreeCC() in most of the instances. This point is even more valid when increasing n further.

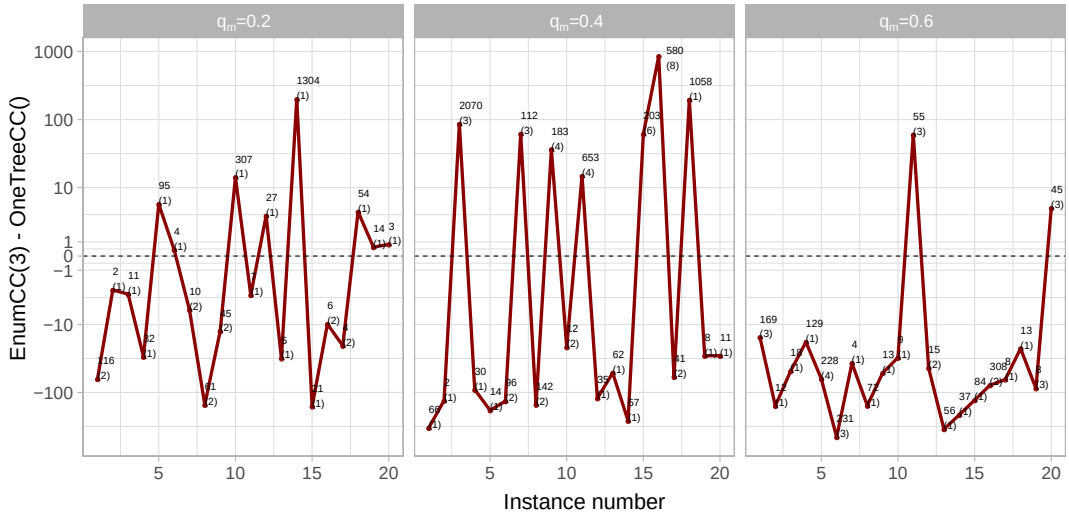
Table 1: Number of optimal solutions found by the considered methods within the time limit of 12h for unsolved instances with $n = 50$.

instance no / methods	$q_m = 0.4$						$q_m = 0.5$							$q_m = 0.6$							
	1	5	6	11	16	19	2	11	12	13	15	18	20	2	6	10	12	13	17	19	20
OneTreeCC()	4	4	1	1	13	2	7	21	3	1	10	2	2	3	1	3	4	5	3	1	3
EnumCC(3)	10	13	6	5	255	16	217	44	8	1	115	46	13	45	10	32	6	60	7	4	65

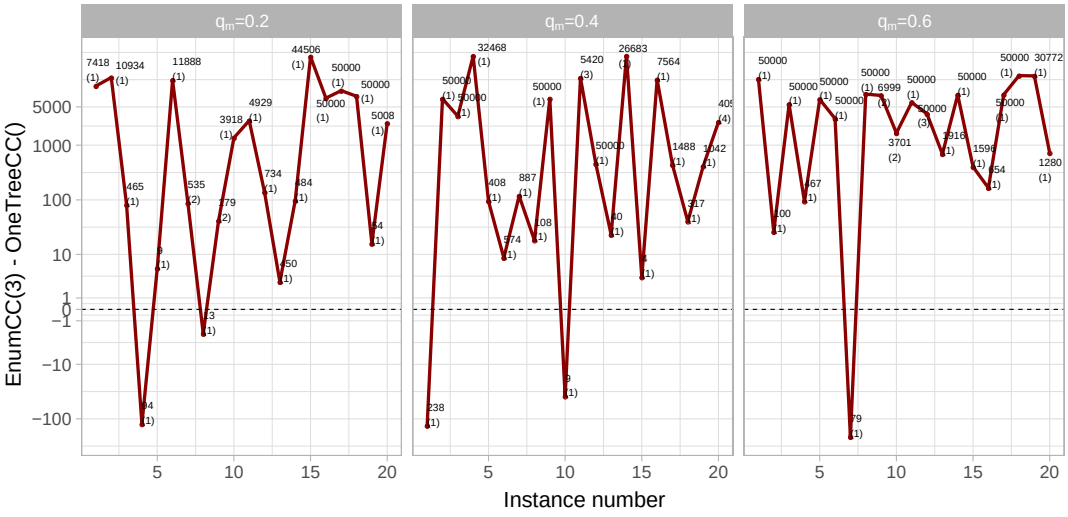
Another interesting point is the hardness of instances, when increasing n. We say that an instance is hard to solve when the resolution process takes too long, which amounts to exceeding a time limit.



(a) Instances with $d = 0.25$ and $q_{neg} = 0.3$.



(b) Instances with $d = 0.25$ and $q_{neg} = 0.5$.



(c) Instances with $d = 0.25$ and $q_{neg} = 0.7$.

Figure 9: Results for $n = 36$, $d = 0.25$ and $q_{neg} \in \{0.3, 0.5, 0.7\}$. The log-scaled y-axis indicates the difference of execution times between EnumCC(3) and OneTreeCC() (i.e., EnumCC(3) minus OneTreeCC()), in seconds. We show for each graph the maximal number of solutions found by EnumCC(3) within a 12h time limit, as well as the corresponding number of jumps, i.e. $n_{jump}(EnumCC(3))$, between parentheses.

It is very important to analyze such cases in terms of the number of optimal solutions found by both methods. Moreover, the existence of such instances also indicates the maximum graph order, for which the enumeration methods can produce full enumeration of all alternate optimal solutions. In our experiments, a total of 21 such instances with $n = 50$ and $q_m > 0.3$ are not solved within the time limit of $12h$ by both methods. These instances can be identified as the ones with $y = 0$ in Figure 10c, and they are summarized in Table 1. Each row corresponds to one of these methods, and each column represents an instance, identified by its id and its associated q_m value. We see from Table 1 that EnumCC(3) handles these time-consuming instances better than OneTreeCC() does. Indeed, among the 21 instances not solved by any method, EnumCC(3) finds more solutions than OneTreeCC() for 20 instances. Moreover, EnumCC(3) could solve 7 instances that OneTreeCC() could not, within the limit of $12h$.

To summarize the evaluation of EnumCC(3) with synthetic signed networks, there is not a single method which always gives the best results. On the one hand, OneTreeCC() handles very well the instances with a very large solution space. This extreme case is associated with a specific graph topology, based on low graph density and large q_{neg}. On the other hand, EnumCC(3) performs better in the rest of the instances, which constitutes the overwhelming majority. This point is illustrated by Table 2, which summarizes our results by showing the proportion of cases where EnumCC(3) runs faster.

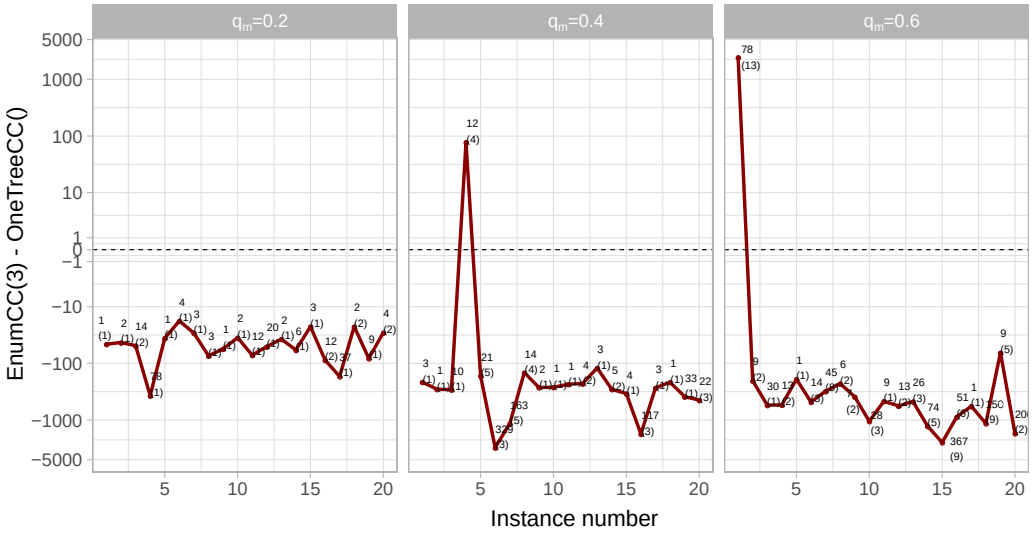
Table 2: Summary regarding the proportion of the cases where EnumCC(3) is faster than OneTreeCC. We obtain these results by aggregating the instances by graph order n (excluding the instances, when the difference of execution times between EnumCC(3) and OneTreeCC() is in the margin of 5 seconds). N/A indicates that there is no available entry due to the exclusion of the instances, when the time difference is in the margin of 5 seconds.

		$q_m = 0.1$	$q_m = 0.2$	$q_m = 0.3$	$q_m = 0.4$	$q_m = 0.5$	$q_m = 0.6$
$d = 0.25$	$q_{neg} = 0.3$	1.00	0.92	1.00	0.87	0.96	0.92
	$q_{neg} = 0.5$	0.00	0.52	0.77	0.61	0.61	0.86
	$q_{neg} = 0.7$	0.00	0.08	0.08	0.05	0.08	0.03
$d = 0.50$	$q_{neg} = 0.3$	N/A	1.00	1.00	1.00	1.00	1.00
	$q_{neg} = 0.5$	N/A	1.00	1.00	0.92	0.84	0.75
	$q_{neg} = 0.7$	0.11	0.80	0.95	0.90	0.89	0.87
$d = 1.00$	$q_{neg} \approx 0.7$	1.00	0.98	0.98	0.94	0.95	0.98

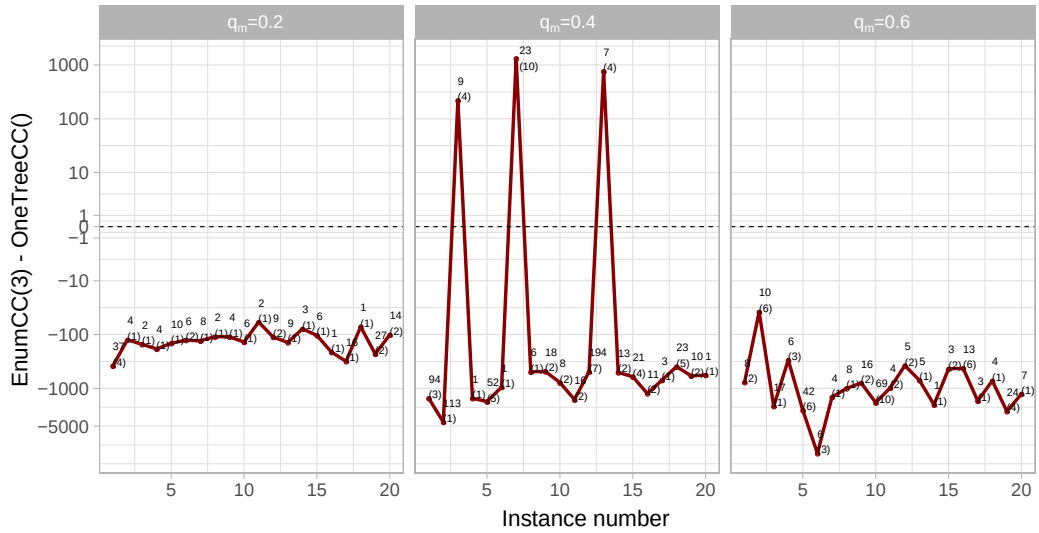
8.2.3 Evaluation Based on Real-World Instances

Finally, we compare the results of EnumCC(3) against OneTreeCC() obtained on the real-world networks of *Dataset3*. We run both methods with a time limit of 12h and a limit of $50,000$ on the number of optimal solutions found. The results are summarized in Table 3. Each column corresponds to a real-world network, and they are sorted by increasing graph order n. We consider the first five networks as medium-sized and the last three as large-sized. The rows of Table 3 are organized in two parts. In the first part, we detail the characteristics of the considered networks, as well as the graph imbalance, for the sake of completeness. The second part corresponds to the evaluation results of OneTreeCC() and EnumCC(3). Therein, we indicate the execution time of the enumeration process, in seconds, and the number of optimal solutions found by these two methods within a time limit of 12h.

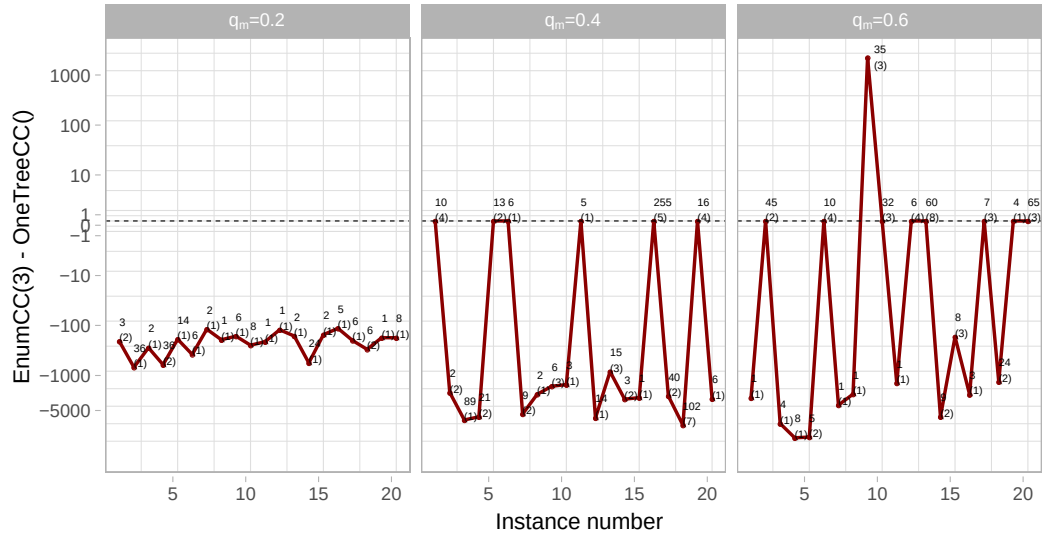
We can summarize these results in three points. First, we notice that although the medium-sized networks are larger than the synthetic graphs from *Dataset2*, in practice both OneTreeCC() and EnumCC(3) can handle real-world medium-sized networks in a very reasonable time. Second, we observe for medium-sized networks that EnumCC(3) is faster than OneTreeCC() for the first one, whereas OneTreeCC() dominates EnumCC(3) for the last four ones. EnumCC(3) takes longer because, in average, it spends 86% of its execution time to prove that there is no alternate optimal solution in the end. Finally, as expected, both methods could not solve all three large-sized networks within the time limit of 12h. The particularity of these networks is that they are very sparse ($d \approx 1\%$) and their respective solution spaces contain at least $50,000$ optimal solutions. For all these networks, EnumCC(3) always finds more optimal solutions than OneTreeCC() does within the time limit, and it quickly explores a set of $50,000$ optimal solutions thanks to its RNS component. Since OneTreeCC() struggles to find more alternate solutions



(a) Instances with $n = 40$ and $d = 1.00$.



(b) Instances with $n = 45$ and $d = 1.00$.



(c) Instances with $n = 50$ and $d = 1.00$.

Figure 10: Results for $n \in \{40, 45, 50\}$ and $d = 1.00$. The log-scaled y-axis indicates the difference of execution times between EnumCC(3) and OneTreeCC() (i.e., EnumCC(3) minus OneTreeCC()), in seconds. We show for each graph the maximal number of solutions found by EnumCC(3) within a time limit, as well as the corresponding number of jumps, i.e. $n_{jump}(EnumCC(3))$, between parentheses.

Table 3: Evaluation results for sparse real-world signed networks from Dataset3. Each column corresponds to a real-world network and they are sorted by graph order n. The rows are organized in two parts. The first six rows detail the characteristics of the considered networks, as well as the graph imbalance. The last four rows indicate the execution time of the enumeration process in seconds and the number of optimal solutions found by OneTreeCC() and EnumCC(3) within a time limit of 12h.

		Medium-sized networks					**Large-sized networks**		
		CoW 51-54	CoW 54-57	CoW 55-58	CoW 61-64	Senate 108th	EGFR	Yeast	Macro-phage
Characteristics	graph order $n = \|V\|$	61	67	72	75	99	313	575	660
	graph size $m = \|E\|$	413	461	508	544	2,413	755	910	1,397
	density d	0.23	0.21	0.20	0.19	0.50	0.015	0.005	0.006
	prop. of neg. edges q_{neg}	0.08	0.09	0.08	0.08	0.41	0.34	0.09	0.33
	# frustrated edges $I(P)$	15	27	29	34	1157	181	35	309
	prop. of frustrated edges $I(P)/\|E\|$	0.04	0.06	0.06	0.06	0.48	0.24	0.04	0.22
Results	# opt. solutions for OneTreeCC()	46	34	41	201	22	112	50,000	251
	# opt. solutions for EnumCC(3)	46	34	41	201	22	50,000	50,000	50,000
	exec. time for OneTreeCC()	87.70s	**14.67s**	**15.62s**	**17.75s**	**858s**	43,200s	24,600s	43,200s
	exec. time for EnumCC(3)	**66.05s**	67.38s	134.87s	755.81s	2,424s	**3,751s**	**1,871s**	**2,280s**

within the time limit in two networks (i.e. EGFR and Macrophage), RNS appears to be useful in these cases.

To conclude this part, all these results also support our conclusion from Section 8.2.2, in that there is not a single method which always gives the best results and both methods are complementary. Moreover, these results allow us to make our conclusions from Section 8.2.2 more precise, in two ways. First, it is more appropriate to use EnumCC(3), rather than OneTreeCC(), to explore a large number of optimal solutions within a time limit, for large signed networks. Nevertheless, it is more suitable to favor OneTreeCC() over EnumCC(3) for sparse signed networks with $65 < n < 100$.

9 Conclusion

For most clustering problems, due to their NP-hard nature, exact approaches do not scale well even when looking for a *single* optimal solution. In this work, we proposed an efficient enumeration method to identify all optimal solutions of the CC problem for a given signed graph. It combines an exhaustive enumeration strategy with neighborhoods of different sizes, designed for our problem. In our experiments based on synthetic and real-world signed networks, we first confirmed the findings of our previous work [6] about the existence of multiple optimal solutions for incomplete signed networks. We showed that such networks can have a very large number of optimal solutions for the CC problem, e.g., $50{,}000$ and more for graphs containing only tens of vertices. Further investigation indicates that this extreme case of a very large number of optimal solutions is associated with two specific graph topologies, corresponding to 1) low graph density and large proportion of negative edges for small- and medium-sized networks (i.e. $n \leq 100$), and 2) very low graph density for large-sized networks (i.e. $n > 100$). Otherwise, multiple solutions can still exist, mostly for the graphs with considerably more imbalance. Finally, we also showed that our method EnumCC(3) performs better than CPLEX's OneTreeCC() in the overwhelming majority of the considered networks. Nevertheless, there is not a single method which always gives the best results. Indeed, OneTreeCC() handles very well sparse medium-sized signed networks in general. We conclude that it is more appropriate to use OneTreeCC() in this case, whereas EnumCC(3) is more suitable for all the remaining cases.

We believe that this work opens new directions for future research. First, the most straightforward perspective is to consider weighted signed graphs. This would require dealing with the generation of the edge weights in our random graph model, and to extend the pruning strategies proposed in this work to weighted signed graphs. Second, as we see in our experiments, the process of complete enumeration can be very costly, particularly because of the process of finding an undiscovered solution. To accelerate this process, it could be very beneficial to use a heuristic. Existing heuristics from the

literature are designed to find a single high-quality solution though, so a better approach would be to redesign them by considering some tabu search-based operations allowing to escape from the discovered optimal solutions. Third, one could study how robust the solution spaces are, by slightly introducing some perturbations into the considered networks. This would also allow identifying critical vertices when the underlying space of optimal solutions is changed, akin to the concept of vitality [33]. This could be done through either repeating the process from scratch for each perturbed signed graph, or based on the definition of stability range [41]. Fourth, exploring the space of optimal solutions for other clustering problems with unsigned networks would be another interesting research line. Indeed, the work of Good *et al.* [26] showed the need of identifying multiple quasi-optimal solutions for the Modularity Maximization problem, which consists in detecting a community structure in an unsigned graph. From this perspective, an efficient enumeration method similar to ours could be implemented to see whether their empirical findings apply to optimal solutions too.

Acknowledgments. This research benefited from the support of Agorantic research federation (FR 3621), as well as the FMJH (Jacques Hadamard Mathematics Foundation) through PGMO (Gaspard Monge Program for Optimisation and operational research), and from the support to this program from EDF, Thales, Orange and Criteo.

A 2-partition and 2-chorded cycle inequalities

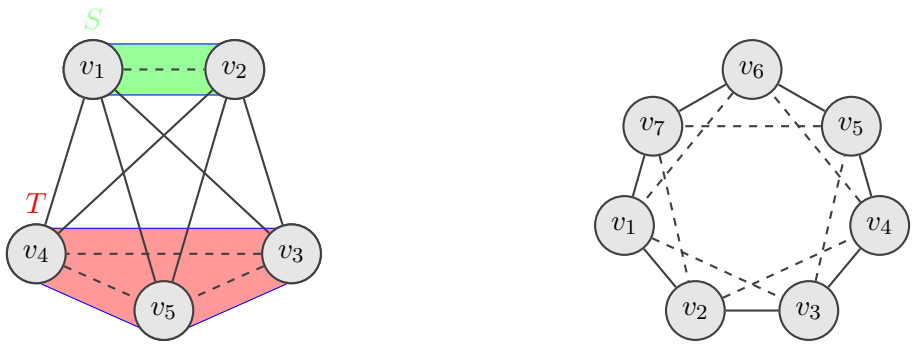
For the sake of completeness, we detail below the 2-partition and 2-chorded cycle inequalities:

- Let $S, T \subseteq V$ be two nonempty disjoint subsets of V. Then, the 2-partition inequality, illustrated in Figure 11a, is defined as

$$\sum_{u \in S} \sum_{v \in T} x_{uv} - \sum_{\substack{(u,v) \in S \\ u \neq v}} x_{uv} - \sum_{\substack{(u,v) \in T \\ u \neq v}} x_{uv} \leq min\{|S|, |T|\}. \tag{20}$$

- Let $C \subseteq E$ be a cycle of length at least 5, and $\overline{C} = \{v_i v_{i+2} | i = 1, .., |C| - 2\} \cup \{v_1 v_{|C|-1}, v_2 v_{|C|}\}$ be a 2-chorded cycle of C. Then, the 2-chorded cycle inequality, illustrated in Figure 11b, is defined as

$$\sum_{(u,v) \in C} x_{uv} - \sum_{(u,v) \in \overline{C}} x_{uv} \leq \lfloor \frac{|C|}{2} \rfloor. \tag{21}$$



(a) 2-partition inequality. (b) 2-chorded cycle inequality.

Figure 11: Illustrations of the 2-partition (left) and 2-chorded cycle (right) inequalities.

B Edit Distance Between Two Membership Vectors

Before calculating the edit distance between two membership vectors, we need to select one of them as the *reference vector*, in order to adapt the module assignments of the other vector based accordingly. Hence, the edit distance is calculated between the reference vector and this newly changed one, that we call *relative vector*.

The task of adapting the relative vector w.r.t the reference one can be expressed as an assignment problem, also known as maximum weighted bipartite matching problem, as already done in the literature [36]. Let π^s and π^t be two membership vectors of length n, associated with the partitions P^s and

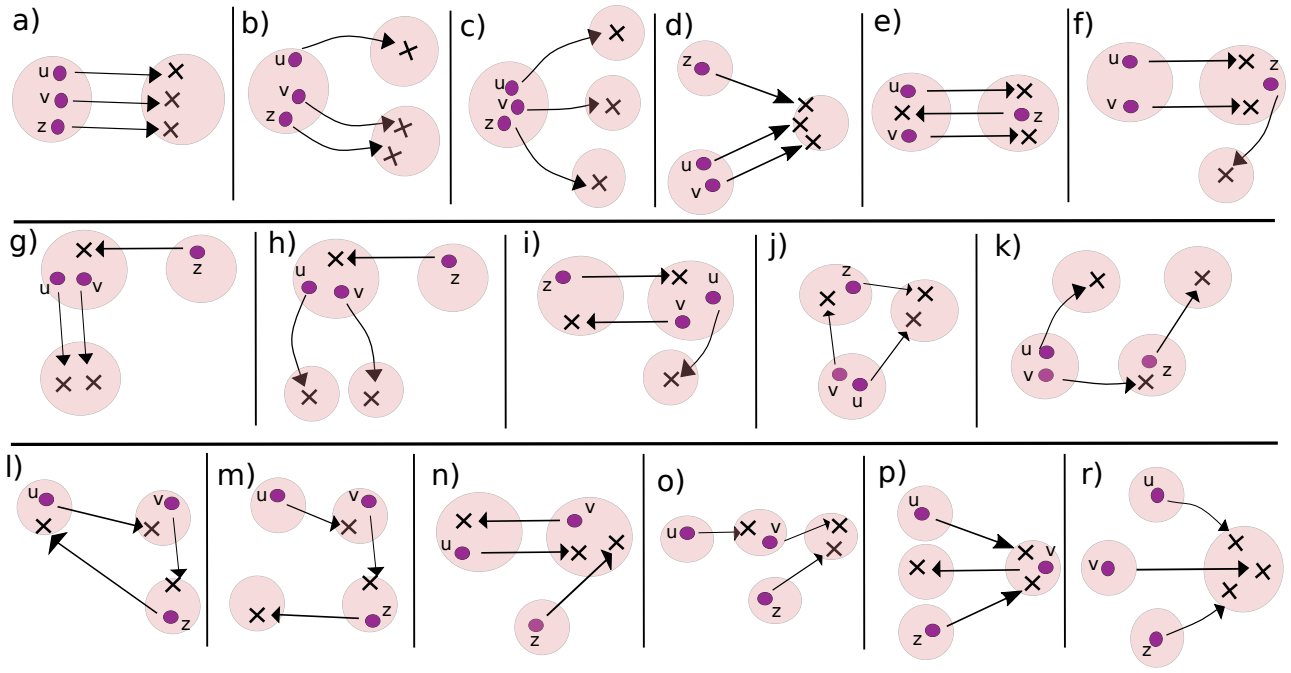


Figure 12: All atomic 3-edit operations.

P^t, which contain ℓ^s and ℓ^t modules, respectively. Also, since the edit distance is symmetric, without loss of generality, let $\ell^s \leq \ell^t$. Moreover, let CM be the $\ell^s \times \ell^t$ confusion matrix of π^s and π^t. The term CM_{ij}, with $1 \leq i \leq \ell^s$ and $1 \leq j \leq \ell^t$, represents the number of vertices in the intersection of modules M_i^s and M_j^t, i.e. $|M_i^s \cap M_j^t|$. Then, we look for a bijection $f : \{1, 2, ..., \ell^s\} \rightarrow \{1, 2, ..., \ell^t\}$ that maximizes the number of vertices common to pairs of modules from both membership vectors, i.e.

$$\max \sum_{i=1}^{\ell^s} CM_{i,f(i)}. \tag{22}$$

Since this problem can be modelled as an assignment or a maximum weighted bipartite matching problem, it can be solved in various ways. One of them is through the well-known Hungarian algorithm, whose complexity is $O(n^3)$ [34]. Nevertheless, the best polynomial time algorithm is currently based on the network simplex method, and it runs in $O(|V||E| + |V|^2 log(|V|))$ time using a Fibonacci heap data structure [25]. One final remark is about the case of $\ell^s < \ell^t$, in which there are $|\ell^t - \ell^s|$ unassigned module labels in π^t. In this case, one can arbitrarily renumber these labels, starting from $\ell^s + 1$.

Finally, the edit distance between two membership vectors is calculated by simply counting the number of cases where the module labels of the vertices in the reference and relative vectors are different.

C Proofs

C.1 All Proofs Related to the MVMO Property for 3-edit Operations on Complete Unweighted Signed Graphs

We complete the proof of Lemma 12 by verifying below the conditions of all atomic 3-edit operations for unweighted complete signed networks (illustrated in Figure 12), where $\overrightarrow{^sV^t} = \{u, v, z\}$ and $(u, v), (u, z), (v, z) \in \widetilde{E}$. Recall that $\widetilde{E} = \{(u, v) \mid (u, v) \in E$ and $u, v \in \overrightarrow{^sV^t}$ and $(\pi^s(u) = \pi^s(v) \vee \pi^t(u) = \pi^s(v) \vee \pi^s(u) = \pi^t(v) \vee \pi^t(u) = \pi^t(v))\}$.

a) We have $(\gamma_u^{left} = a_{uv} + a_{uz}) > (\gamma_u^{right} = -a_{uv} - a_{uz})$, $(\gamma_v^{left} = a_{uv} + a_{vz}) > (\gamma_v^{right} = -a_{uv} - a_{vz})$ and $(\gamma_z^{left} = a_{uz} + a_{vz}) > (\gamma_z^{right} = -a_{uz} - a_{vz})$. We see that a_{uv}, a_{uz} and a_{vz} cannot be negative.

b) We have $(\gamma_u^{left} = a_{uv} + a_{uz}) > (\gamma_u^{right} = 0)$, $(\gamma_v^{left} = a_{uv} + a_{vz}) > (\gamma_v^{right} = -a_{vz})$ and $(\gamma_z^{left} = a_{uz} + a_{vz}) > (\gamma_z^{right} = -a_{vz})$. We see that a_{uv}, a_{uz} and a_{vz} cannot be negative.

c) We have $(\gamma_u^{left} = a_{uv} + a_{uz}) > (\gamma_u^{right} = 0)$, $(\gamma_v^{left} = a_{uv} + a_{vz}) > (\gamma_v^{right} = 0)$ and $(\gamma_z^{left} = a_{uz} + a_{vz}) > (\gamma_z^{right} = 0)$. We see that a_{uv}, a_{uz} and a_{vz} cannot be negative.

d) We have $(\gamma_u^{left} = a_{uv}) > (\gamma_u^{right} = -a_{uv} - a_{uz})$, $(\gamma_v^{left} = a_{uv}) > (\gamma_v^{right} = -a_{uv} - a_{vz})$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = -a_{uz} - a_{vz})$. We see that a_{uv}, a_{uz} and a_{vz} must be positive.

e) We have $(\gamma_u^{left} = a_{uv} - a_{uz}) > (\gamma_u^{right} = -a_{uv} + a_{uz})$, $(\gamma_v^{left} = a_{uv} - a_{vz}) > (\gamma_v^{right} = -a_{uv} + a_{vz})$ and $(\gamma_z^{left} = -a_{uz} - a_{vz}) > (\gamma_z^{right} = a_{uz} + a_{vz})$. We see that a_{uv} (resp. a_{uz} and a_{vz}) cannot be negative (resp. positive).

f) We have $(\gamma_u^{left} = a_{uv} - a_{uz}) > (\gamma_u^{right} = -a_{uv})$, $(\gamma_v^{left} = a_{uv} - a_{vz}) > (\gamma_v^{right} = -a_{uv})$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = a_{uz} + a_{vz})$. We see that a_{uv}, a_{uz} and a_{vz} cannot be negative.

g) We have $(\gamma_u^{left} = a_{uv}) > (\gamma_u^{right} = a_{uz} - a_{uv})$, $(\gamma_v^{left} = a_{uv}) > (\gamma_v^{right} = a_{vz} - a_{uv})$ and $(\gamma_z^{left} = -a_{uz} - a_{vz}) > (\gamma_z^{right} = 0)$. We see that a_{uv}, a_{uz} and a_{vz} cannot be negative.

h) We have $(\gamma_u^{left} = a_{uv}) > (\gamma_u^{right} = -a_{uv})$, $(\gamma_v^{left} = a_{uv}) > (\gamma_v^{right} = -a_{uv})$ and $(\gamma_z^{left} = -a_{uz} - a_{vz}) > (\gamma_z^{right} = 0)$. We see that a_{uv}, a_{uz} and a_{vz} cannot be negative.

i) We have $(\gamma_u^{left} = a_{uv}) > (\gamma_u^{right} = a_{uz})$, $(\gamma_v^{left} = a_{uv} - a_{vz}) > (\gamma_v^{right} = a_{vz})$ and $(\gamma_z^{left} = -a_{uz} - a_{vz}) > (\gamma_z^{right} = +a_{vz})$. We see that a_{uv} (resp. a_{uz} and a_{vz}) cannot be negative (resp. positive).

j) We have $(\gamma_u^{left} = a_{uv}) > (\gamma_u^{right} = -a_{uz})$, $(\gamma_v^{left} = a_{uv}) > (\gamma_v^{right} = -a_{vz})$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = -a_{uz} + a_{vz})$. We see that a_{uv} and a_{vz} (resp. a_{uz}) must be positive (resp. negative).

k) We have $(\gamma_u^{left} = a_{uv}) > (\gamma_u^{right} = 0)$, $(\gamma_v^{left} = a_{uv} - a_{vz}) > (\gamma_v^{right} = 0)$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = a_{vz})$. We see that a_{uv} and a_{vz} (resp. a_{uz}) must be positive (resp. negative).

l) We have $(\gamma_u^{left} = -a_{uv}) > (\gamma_u^{right} = a_{uz})$, $(\gamma_v^{left} = -a_{vz}) > (\gamma_v^{right} = a_{uv})$ and $(\gamma_z^{left} = -a_{uz}) > (\gamma_z^{right} = a_{vz})$. We see that a_{uv} and a_{vz} (resp. a_{uz}) must be positive (resp. negative).

m) We have $(\gamma_u^{left} = -a_{uv}) > (\gamma_u^{right} = 0)$, $(\gamma_v^{left} = -a_{vz}) > (\gamma_v^{right} = a_{uv})$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = a_{vz})$. We see that a_{uv} and a_{vz} (resp. a_{uz}) must be positive (resp. negative).

n) We have $(\gamma_u^{left} = -a_{uv}) > (\gamma_u^{right} = a_{uv} - a_{uz})$, $(\gamma_v^{left} = -a_{uv}) > (\gamma_v^{right} = a_{uv} + a_{vz})$ and $(\gamma_z^{left} = -a_{vz}) > (\gamma_z^{right} = -a_{uz})$. We see that a_{uv} (resp. a_{uz} and a_{vz}) cannot be negative (resp. positive).

o) We have $(\gamma_u^{left} = -a_{uv}) > (\gamma_u^{right} = 0)$, $(\gamma_v^{left} = 0) > (\gamma_v^{right} = a_{uv} - a_{vz})$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = -a_{vz})$. We see that a_{uv} (resp. a_{uz} and a_{vz}) cannot be negative (resp. positive).

p) We have $(\gamma_u^{left} = -a_{uv}) > (\gamma_u^{right} = -a_{uz})$, $(\gamma_v^{left} = 0) > (\gamma_v^{right} = a_{uv} + a_{vz})$ and $(\gamma_z^{left} = -a_{vz}) > (\gamma_z^{right} = -a_{uz})$. We see that a_{uv} (resp. a_{uz} and a_{vz}) cannot be negative (resp. positive).

r) We have $(\gamma_u^{left} = 0) > (\gamma_u^{right} = -a_{uv} - a_{uz})$, $(\gamma_v^{left} = 0) > (\gamma_v^{right} = -a_{uv} - a_{vz})$ and $(\gamma_z^{left} = 0) > (\gamma_z^{right} = -a_{uz} - a_{vz})$. We see that a_{uv}, a_{uz} and a_{vz} must be positive.

References

[1] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. "A survey of very large-scale neighborhood search techniques". In: *Discrete Applied Mathematics* 123.1-3 (2002), pp. 75–102. DOI: 10.1016/s0166-218x(01)00338-9.

[2] Z. Ales, A. Knippel, and A. Pauchet. "Polyhedral Combinatorics of the K-partitioning Problem with Representative Variables". In: *Discrete Applied Mathematics* 211 (2016), pp. 1–14. DOI: 10.1016/j.dam.2016.04.002.

[3] S. Aref and M. C. Wilson. "Balance and frustration in signed networks". In: *Journal of Complex Networks* 7.2 (2018), pp. 163–189. DOI: 10.1093/comnet/cny015.

[4] J. L. Arthur, M. Hachey, Sahr K., M. Huso, and A. R. Kiester. "Finding all optimal solutions to the reserve site selection problem". In: *Environmental and Ecological Statistics* 4.2 (1997), pp. 153–165. DOI: 10.1023/a:1018570311399.

[5] N. Arınık, R. Figueiredo, and V. Labatut. "Multiple partitioning of multiplex signed networks: Application to European parliament votes". In: *Social Networks* 60 (2020), pp. 83 –102. DOI: https://doi.org/10.1016/j.socnet.2019.02.001.

[6] N. Arınık, R. Figueiredo, and V. Labatut. "Multiplicity and Diversity: Analyzing the Optimal Solution Space of the Correlation Clustering Problem on Complete Signed Graphs". In: *Journal of Complex Networks* (2020). DOI: 10.1093/comnet/cnaa025.

[7] N. Arınık, R. Figueiredo, and V. Labatut. "Signed Graph Analysis for the Interpretation of Voting Behavior". In: *International Conference on Knowledge Technologies and Data-driven Business - International Workshop on Social Network Analysis and Digital Humanities*. 2017.

[8] N. Bansal, A. Blum, and S. Chawla. "Correlation Clustering". In: *43rd Annual IEEE Symposium on Foundations of Computer Science*. 2002, pp. 238–247. DOI: 10.1109/SFCS.2002.1181947.

[9] C. Blum and A. Roli. "Metaheuristics in combinatorial optimization". In: *ACM Computing Surveys* 35.3 (2003), pp. 268–308. DOI: 10.1145/937503.937505.

[10] Michael Brusco and Douglas Steinley. "K-balance partitioning: An exact method with applications to generalized structural balance and other psychological contexts". In: *Psychological Methods* 15.2 (2010), pp. 145–157. DOI: 10.1037/a0017738.

[11] J. D. Camm, S. Polasky, A. Solow, and B. Csuti. "A note on optimal algorithms for reserve site selection". In: *Biological Conservation* 78.3 (1996), pp. 353–355. DOI: 10.1016/0006-3207(95)00132-8.

[12] D. Cartwright and F. Harary. "Structural balance: A generalization of Heider's theory". In: *Psychological Review* 63 (1956), pp. 277–293. DOI: 10.1037/h0046049.

[13] Moses Charikar, Neha Gupta, and Roy Schwartz. "Local Guarantees in Graph Cuts and Clustering". In: *Integer Programming and Combinatorial Optimization*. Springer International Publishing, 2017, pp. 136–147. DOI: 10.1007/978-3-319-59250-3_12.

[14] P. Damaschke. "Fixed-Parameter Enumerability of Cluster Editing and Related Problems". In: *Theory of Computing Systems* 46.2 (2010), pp. 261–283. DOI: 10.1007/s00224-008-9130-1.

[15] E. Danna, M. Fenelon, Z. Gu, and R. Wunderling. "Generating Multiple Solutions for Mixed Integer Programming Problems". In: *International Conference on Integer Programming and Combinatorial Optimization*. Springer, 2007, pp. 280–294. DOI: 10.1007/978-3-540-72792-7_22.

[16] B. DasGupta, G. A. Enciso, E. Sontag, and Y. Zhang. "Algorithmic and complexity results for decompositions of biological networks into monotone subsystems". In: *Biosystems* 9.1 (2007), pp. 161–178. DOI: 10.1016/j.biosystems.2006.08.001.

[17] J. A. Davis. "Clustering and structural balance in graphs". In: *Human Relations* 20.2 (1967), pp. 181–187. DOI: 10.1177/001872676702000207.

[18] Erik D. Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. "Correlation clustering in general weighted graphs". In: *Theoretical Computer Science* 361.2-3 (2006), pp. 172–187. DOI: 10.1016/j.tcs.2006.05.008.

[19] P. Doreian, P. Lloyd, and A. Mrvar. "Partitioning large signed two-mode networks: Problems and prospects". In: *Social Networks* 35.2 (2013), pp. 178–203. DOI: 10.1016/j.socnet.2012.01.002.

[20] P. Doreian and A. Mrvar. "A partitioning approach to structural balance". In: *Social Networks* 18.2 (1996), pp. 149–168. DOI: 10.1016/0378-8733(95)00259-6.

[21] P. Doreian and A. Mrvar. "Structural balance and signed international relations". In: *Journal of Social Structure* 16 (2015), p. 1. DOI: 10.21307/joss-2019-012.

[22] J. Esteban, L. Mayoral, and D. Ray. "Ethnicity and Conflict: An Empirical Study". In: *American Economic Review* 102.4 (2012), pp. 1310–1342. DOI: 10.1257/aer.102.4.1310.

[23] R. Figueiredo and G. Moura. "Mixed Integer Programming Formulations for Clustering Problems Related to Structural Balance". In: *Social Networks* 35.4 (2013), pp. 639–651. DOI: 10.1016/j.socnet.2013.09.002.

[24] Matteo Fischetti, Fred Glover, and Andrea Lodi. "The feasibility pump". In: *Mathematical Programming* 104.1 (2005), pp. 91–104. DOI: 10.1007/s10107-004-0570-3.

[25] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615. DOI: 10.1145/28869.28874.

[26] B. H. Good, Y.-A. de Montjoye, and A. Clauset. "Performance of modularity maximization in practical contexts". In: *Physical Review E* 81.4 (2010). DOI: 10.1103/physreve.81.046106.

[27] M. Grötschel and Y. Wakabayashi. "A cutting plane algorithm for a clustering problem". In: *Mathematical Programming* 45.1-3 (1989), pp. 59–96. DOI: 10.1007/bf01589097.

[28] M. Grötschel and Y. Wakabayashi. "Facets of the clique partitioning polytope". In: *Mathematical Programming* 47.1-3 (1990), pp. 367–387. DOI: 10.1007/bf01580870.

[29] F. Gürsoy and B. Badur. "Extracting the signed backbone of intrinsically dense weighted networks". In: *arXiv e-prints*, arXiv:2012.05216 (2020), arXiv:2012.05216. arXiv: 2012.05216.

[30] IBM. *IBM ILOG CPLEX 12.8 User Manual IBM Corporation*. 2018.

[31] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing,Computational Linguistics, and Speech Recognition*. 2nd edition. Prentice-Hall, Inc., 2000.

[32] M. Keuper, J. Lukasik, M. Singh, and J. Yarkony. "A Benders Decomposition Approach to Correlation Clustering". In: *The International Conference for High Performance Computing, Networking, Storage, and Analysis*. 2020.

[33] D. Koschützki, K. A. Lehmann, L. Peeters, S. Richter, D. Tenfelde-Podehl, and O. Zlotowski. "Centrality Indices". In: *Network Analysis: Methodological Foundations*. Vol. 3418. Lecture Notes in Computer Science. 2005, pp. 16–61. DOI: 10.1007/978-3-540-31955-9_3.

[34] H. W. Kuhn. "The Hungarian method for the assignment problem". In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97. DOI: 10.1002/nav.3800020109.

[35] J. Kunegis, A. Lommatzsch, and C. Bauckhage. "The slashdot zoo". In: *Proceedings of the 18th international conference on World wide web*. ACM Press, 2009, pp. 741–750. DOI: 10.1145/1526709.1526809.

[36] Xin Liu, Hui-Min Cheng, and Zhong-Yuan Zhang. "Evaluation of Community Detection Methods". In: *IEEE Transactions on Knowledge and Data Engineering* (2019), pp. 1–1. DOI: 10.1109/tkde.2019.2911943.

[37] F. Ma and J.-K. Hao. "A multiple search operator heuristic for the max-k-cut problem". In: *Annals of Operations Research* 248.1 (2016), pp. 365–403. DOI: 10.1007/s10479-016-2234-0.

[38] P. Massa and P. Avesani. "Controversial users demand local trust metrics: an experimental study on Epinions.com community". In: *Proceedings of the 20th national conference on Artificial intelligence*. Vol. 1. 2005, pp. 121–126.

[39] A. Mehrotra and M. A. Trick. "Cliques and clustering: A combinatorial approach". In: *Operations Research Letters* 22.1 (1998), pp. 1–12. DOI: 10.1016/s0167-6377(98)00006-6.

[40] George Nemhauser and Laurence Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1999.

[41] S. Nowozin and S. Jegelka. "Solution stability in linear programming relaxations". In: *Proceedings of the 26th Annual International Conference on Machine Learning - ICML*. ACM Press, 2009. DOI: 10.1145/1553374.1553473.

[42] M. Oosten, J. H. G. C. Rutten, and F. C. R. Spieksma. "The clique partitioning problem: Facets and patching facets". In: *Networks* 38.4 (2001), pp. 209–226. DOI: 10.1002/net.10004.

[43] J. Pevehouse, T. Nordstrom, and K. Warnke. "The Correlates of War 2 International Governmental Organizations Data Version 2.0". In: *Conflict Management and Peace Science* 21.2 (2004), pp. 101–119. DOI: 10.1080/07388940490463933.

[44] E. Queiroga, A. Subramanian, R. Figueiredo, and Y. Frota. "Integer programming formulations and efficient local search for relaxed correlation clustering". In: *Journal of Global Optimization* (2021). DOI: 10.1007/s10898-020-00989-7.

[45] S. Tan and J. Lü. "An evolutionary game approach for determination of the structural conflicts in signed networks". In: *Scientific Reports* 6.1 (2016), p. 22022. DOI: 10.1038/srep22022.

[46] T. Zaslavsky. "Balanced decompositions of a signed graph". In: *Journal of Combinatorial Theory, Series B* 43.1 (1987), pp. 1–13. DOI: 10.1016/0095-8956(87)90026-8.