Noname manuscript No.
(will be inserted by the editor)

Mining Patterns in Graphs with Multiple Weights

Giulia Preti · Matteo Lissandrini · Davide Mottin · Yannis Velegrakis

Received: date / Accepted: date

Abstract Graph pattern mining aims at identifying structures that appear frequently in large graphs, under the assumption that frequency signifies importance. In real life, there are many graphs with weights on nodes and/or edges. For these graphs, it is fair that the importance (score) of a pattern is determined not only by the number of its appearances, but also by the weights on the nodes/edges of those appearances. Scoring functions based on the weights do not generally satisfy the apriori property, which guarantees that the number of appearances of a pattern cannot be larger than the frequency of any of its sub-patterns, and hence allow faster pruning. Therefore, existing approaches employ other, less efficient, pruning strategies. The problem becomes even more challenging in the case of multiple weighting functions that assign different weights to the same nodes/edges. In this work we propose a new family of scoring functions that respects the apriori property, and thus can rely on effective pruning strategies. We provide efficient and effective techniques for mining patterns in multi-weight graphs, and we devise both an exact and an approximate solution. In addition, we propose a distributed version of our approach, which distributes the appearances of the patterns to examine among multiple workers. Extensive experiments on both real and synthetic datasets prove that the presence of edge weights and the choice of scoring function affect the patterns mined, and the quality of the

Giulia Preti
University of Trento, Trento, Italy
E-mail: gp@disi.unitn.eu

Matteo Lissandrini
Aalborg University, Denmark
E-mail: matteo@cs.aau.dk

Davide Mottin
Aarhus University, Denmark
E-mail: davide@cs.au.dk

Yannis Velegrakis
University of Trento, Trento, Italy
E-mail: velgias@disi.unitn.eu

results returned to the user. Moreover, we show that, even when the performance of the exact algorithm degrades because of an increasing number of weighting functions, the approximate algorithm performs well and with fairly good quality. Finally, the distributed algorithm proves to be the best choice for mining large and rich input graphs.[1]

Keywords Multi-weighted Graphs · Graph Mining · Weighted Pattern Mining · Personalized Patterns

1 Introduction

Pattern mining in large graphs has attracted considerable attention, since it finds applications in many real world scenarios like fraud detection [35], biological structures identification [18], anticipation of user intention [37], graph similarity search [22], traffic control [23], and query optimization [49]. It has been studied for graph collections [48], for attributed [40], probabilistic [28], or even generic large graphs [12]. The goal is to identify patterns that occur frequently, given that frequency indicates importance. An interesting property regarding frequency is that a pattern cannot be more frequent that any of its sub-patterns, known as the *apriori* property. This property enables efficient implementations [50], as it ensures that the frequency of a pattern decreases monotonically as the pattern grows in size, thus allowing the mining process to start from small patterns and extend to larger ones only when the frequency of the pattern is above a certain frequency threshold.

In graph databases the frequency of a pattern has been effectively computed as the number of distinct graphs containing an appearance of the pattern. However, the same implementation cannot be used in single large graphs, as each pattern would have frequency either equal to 0 or to 1. Furthermore, if we simply define the frequency as the number of distinct occurrences of the pattern, the apriori property does no hold anymore [45]. In fact, this implementation counts every overlap that may occur among the instances of the same pattern, hence assigning larger frequencies to larger patterns, causing an unwanted (and unjustified) skew in the value of importance for some patterns. For this reason, alternative metrics have been considered in the literature [45,13,8], with the more prevalent one being the MNI support, as it enjoys high effectiveness [12].

Many real world scenarios are naturally modeled through weighted graphs, and in these cases, the importance of a pattern should be determined not only by the frequency, but also by the weights of its appearances. Examples include the discovery of metabolic pathways in genomic networks [25], where weights indicate strength between genomes [10], the identification of topics of interest in large knowledge graphs [33], where weights quantify the degree a piece of data is qualified as an answer to a user [46], or the detection of common problematic cases in computer networks, where weights indicate congestion [7]. Unfortunately, weighted patterns do not possess the apriori property, since the weights of the extra edges/nodes of a larger pattern may offset its lower frequency. As a consequence, some works that considered

[1] The current paper is an extended versions of a recent EDBT'18 article [38]

weighted graphs for pattern mining proposed solutions that are less efficient than those based on the apriori property [50].

Moreover, a requirement in modern applications is to offer personalized products and services rather than generic preferences [39]. Such generic preferences suit the user on average but fail to deliver the right answer for each specific user. That is, we should offer to each user information that is tailored to their specific interests or preferences. The same argument holds when mining for graph patterns. That is, there might be a discrepancy between patterns that are frequent in general, and patterns that are relevant to a single user. For instance, social network systems record user interactions [24] and activities [6] and build graphs by modeling the relationships among users and web content to find frequent patterns of interactions [34]. Yet, some patterns of interactions may be more important than others to an advertiser depending on the product or the specific business model. Since the interaction graph is queried by multiple advertisers, each one with their own targets, multiple weights are required to distinguish the diversity of preferences. Other examples include on-line retailers like Amazon, which build large graphs on product co-purchases and then exploit the discovered patterns to recommend future offers to their customers [41]. Frequency, number of items, recency of the purchase, as well as the company's business intentions affect the importance of some co-purchases with respect to others [39]. The same argument is true for many other use-cases.

Example Consider a heterogeneous citation network that includes authors, papers, venues, and terms (keywords). In his graph, edges connect papers with their authors, the works they cite, the venue where they were presented, and with the keywords appearing in the title and in the keyword list.

Frequent pattern mining in such networks finds patterns that mainly contain top venues and terms related to research fields with high engagement. In fact, these labels appear very often in the graph, and thus are characterized by a larger support.

On the other hand, we can weight the nodes and edges of the network according to the user preferences, which can be inferred from the papers she published, her coauthors, or the keywords she used and liked. Since the graph can be accessed by multiple users, each one with their own preferences, each single edge is associated with multiple weights, one for each user. Mining relevant patterns in such multi-weighted network allows us to guide each user in the exploration of the literature related to their own research interests. For instance, a researcher working in the field of machine translation is unlikely to be interested in patterns describing the largely studied area of data mining.

The examples above highlight the need for a solution that on the one-hand is able to mine patterns based on their weights instead of limiting to their frequency, and on the other hand, accounts for the individual preferences expressed as a multi-weighted graph, as opposed to "one size fits all" solutions where only a single set of weights is considered. We note that the straightforward approach to multi-weighted pattern mining that runs the mining algorithm on each weighted graph separately, is clearly impractical due to the graph size and the large number of users to handle.

In this paper, we propose a novel approach to mine patterns in weighted graphs that goes beyond frequencies, yet has performance not significantly different from the pattern mining in unweighted graphs. We achieve this by defining a novel family of scoring functions that are based on the MNI score [8], a metric that is widely used in graph mining due to its characteristic of respecting the apriori property, while being efficient to compute. The solution we have devised is modeled as a constraint satisfaction problem (CSP), as proposed also for unweighted pattern mining [12], and implements a strategy called pattern growth approach [48]. Furthermore, we extend the idea above for the case of graphs with multiple weights on their edges/nodes. To avoid running the algorithm one time for each different weighting function, we compute all the scores of each pattern at the moment we are visiting it, and keep the patterns that return a high score with respect to at least one weighting function.

In particular, we make the following contributions:

1. We extend the task of pattern mining in weighted large graphs for a novel family of scoring functions based on the MNI support [8] (Section 2).
2. We introduce and formally define the problem of pattern mining in multi-weight graphs with different weighting functions.
3. We devise two efficient and effective techniques for solving the pattern mining problem on multi-weighted graphs (Section 3). The first finds an exact solution, called RESUM, that is less time and space consuming than the (naive) brute force approach. It avoids redundant revisits of the graph, by aggregating and performing once multiple computations on the same parts of the graph, and storing the relevant patterns in a compact way. The second is a conservative approximate solution, called RESUM *approximate*, that reduces the number of weighting functions to consider, by aggregating those having a high probability to generate similar results (Section 4) into a single representative function. In addition, we show that this method introduces only few false positives, while running considerably faster than the exact approach.
4. We propose a distributed version of RESUM able to scale to large graphs. This algorithm runs on top of the distributed graph processing system Arabesque [43].
5. We study four different scoring functions (all based on the MNI support) for devising the score of a pattern in an efficient way (Section 6).
6. We evaluate our approaches with an extensive set of experiments on both real and synthetic graphs and discuss our findings, in particular we showcase advantages and limitations of the exact solution when compared to the approximate solution, as well as to the distributed version of the algorithms (Section 8).

2 Problem Definition

We assume the existence of a countable set of labels Σ that includes the special symbol $\perp$, and a set $\mathscr{I}_0^1 = [0,1] \cup \{\perp\}$ of weights. The symbol $\perp$ in Σ and $\mathscr{I}_0^1$ is used to denote *no label* and *no weight*, respectively. A weighted graph is a structure that consists of a set of nodes, a set of edges between them, an assignment of labels to all the nodes and edges, and an assignment of weights to all the edges.

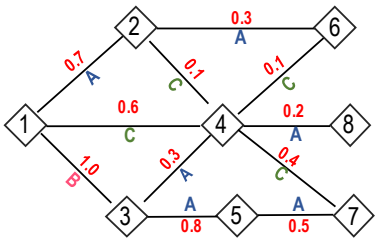

Fig. 1 Example of a edge-labeled, weighted graph.

Definition 1 A **weighted labeled graph**, or simply a **graph**, is a tuple $\langle V, E, \ell, \omega \rangle$ where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, $\ell : E \cup V \rightarrow \Sigma$ is a labeling function, and $\omega : E \rightarrow \mathscr{I}_0^1$ is a weighting function. The symbol $\mathscr{G}$ is used to denote the set of all the possible graphs.

Definition 1 assumes that the weights are in the range of [0, 1]. Some applications may allow the weights of any positive or negative value, but such values can be scaled down to $[0, 1]$. Others may assume categorical values. Even in this case, the values can be mapped to numbers depending on their semantics. For example, in the case in which the values express preference, the value *extremely needed* can be replaced with 1, *moderately needed* with 0.5, and *not wanted* with 0. For the edges that have no weight, the 0 value can be assumed.

Note that we assume weights on edges only, mainly for presentation purposes. Weights on nodes can also be considered with no need for any major modification. A graph $S{:}\langle V_S, E_S, \ell, \omega \rangle$ is said to be a **subgraph** of another graph $G{:}\langle V_G, E_G, \ell, \omega \rangle$, denoted as $S \sqsubseteq G$, if $V_S \subseteq V_G$ and $E_S \subseteq E_G$. Note that the two graphs have the same labeling and weighting function.

To express the fact that two graphs have the same topological structure, we use the notion of *isomorphism*, which is a bijective mapping between the nodes of the two graphs such that the edges between the nodes, alongside their labels, are preserved through the mapping.

Definition 2 A graph $G{:}\langle V, E, \ell, \omega \rangle$ is **isomorphic** to a graph $G'{:}\langle V', E', \ell', \omega' \rangle$, denoted as $G \simeq G'$, if there exists a bijective function $\phi{:}V \rightarrow V'$ such that: $\forall \langle u, v \rangle \in E$: $\langle \phi(u), \phi(v) \rangle \in E'$ and $\ell(\langle u, v \rangle) = \ell'(\langle \phi(u), \phi(v) \rangle)$.

A graph G may have multiple isomorphic graphs. To collectively represent those graphs, we introduce the concept of *pattern*. Intuitively, a pattern is a graph with no weights, serving as a representative of a set of isomorphic graphs and describing their common structure.

Definition 3 A **pattern** is a graph $\langle V, E, \ell, \omega \rangle$, such that $\forall e \in E : \omega(e) = \perp$. The symbol $\mathscr{P}$ denotes the set of all possible patterns. Given a graph G, and a pattern P, the *support set* of the pattern P is the set $S_G(P) = \{g | g \sqsubseteq G \wedge g \simeq P \wedge P \in \mathscr{P}\}$. Each element in $S_G(P)$ is referred to as an *appearance* (also a matching or an embedding) of P in G.

By definition, the support set of P is the set of all the subgraphs of G that are isomorphic to P. By abuse of notation we write $P \sqsubseteq G$ and call the pattern P a subgraph of G if its support set is non-empty. Then, we denote by $\phi_g^P : \mathscr{G} \mapsto \mathscr{P}$ the bijection that maps an isomorphic subgraph g of G to the pattern P.

Given a **scoring function** f: $\mathscr{P} \times \mathscr{G} \to \mathbb{R}$, we will refer to the value $f(P,G)$ as the **score** of P in G. Graph pattern mining is the task that aims at identifying those patterns that have score higher than a threshold τ, or the k patterns with the highest score [12]. A natural scoring function is the one that returns the cardinality of $S_G(P)$, i.e., the number of appearances of the pattern P in the graph G, and the patterns identified by this function are called *frequent patterns*. Nevertheless, it has been shown that this simple function violates the a-priory property, due to the presence of overlapping isomorphisms in G [8]. As an example, the frequency of the pattern $P_1 : [v_1] - B - [v_2] - A - [v_3]$ in the graph in Figure 1 is 3, while the frequency of its sub-pattern $P_2 : [v_1] - B - [v_2]$ is 1. For this reason a number of works have investigated alternative scoring functions [17,45,27,13]. Among them, the MNI support is highly effective and efficient to compute [8].

Definition 4 Given a graph $G : \langle V, E, \ell, \omega \rangle$, the **MNI support** of a pattern P:$\langle V_P, E_P, \ell_P, \omega_P \rangle$ in G is the number $MNI(P,G) = \min_{v' \in V_P} |\mathscr{N}(G,v')|$ where $\mathscr{N}(G,v') = \{v \mid v \in V \wedge \exists g \in S_G(P)$ such that $\phi_g^P(v) = v'\}$.

Intuitively, the set $\mathscr{N}(G,v')$ contains all the nodes of G that are mapped to the pattern node v' by some isomorphism ϕ_g^P from g to P. Then, the MNI support is the minimum cardinality of this set across all the nodes of the pattern P. We can define similar sets also for the pattern edges, i.e., for each $e' \in E_P$, the set $\mathscr{E}(G,e') = \{e | e \in E \wedge \exists g \in S_G(P)$ such that $\phi_g^P(e) = e'\}$ contains all the edges of G that are mapped to the pattern edge e' by some isomorphism $\phi_g^P(e)$. Consider, for instance, the graph G in Figure 1 and the pattern $P : [v_1] - B - [v_2] - A - [v_3]$. Since $\mathscr{N}(G,v_1) = \{1,3\}$, $\mathscr{N}(G,v_2) = \{1,3\}$, and $\mathscr{N}(G,v_3) = \{2,4,5\}$, the MNI support of P is 2. On the other hand, the number of appearances of P in G is 3: $S_G(P) = \{[3] - B - [1] - A - [2]$, $[1] - B - [3] - A - [4]$, $[1] - B - [3] - A - [5]\}$.

In the presence of weights on edges, the score of a pattern cannot be based only on the frequency, but should strike a balance between frequency and weights, allowing also the weights to play a role in assessing the relevance of the pattern. Thus, there is a need for a different scoring function that looks beyond the structure of the subgraphs, by considering the importance of their edges as well. In this case, we talk about *weighted frequent patterns*, or *relevant patterns*. This alternative scoring function, however, has to be carefully selected to satisfy the apriori property [50].

Furthermore, if there are multiple weighting functions, i.e., several functions that assign weights on the graph edges/nodes, then the pattern mining task must be carried out for each individual function. An example of graph with multiple weights on the edges is illustrated in Figure 2. This may happen, for instance, in the case where each set of weight is assigned to a distinct user with a specific set of preferences. Such situation leads to the following specification of the mining task.

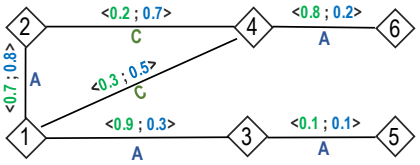

Fig. 2 Graph with two weights $< \omega_1, \omega_2 >$ on each edge.

Pattern Mining in Multi-Weighted Graphs. Given a threshold τ, a scoring function f and a graph $G : \langle V, E, \ell, W \rangle$, where W is a finite set of weighting functions, we must discover, $\forall \omega_i \in W$, the set of patterns $R_i = \{P | G' = \langle V, E, \ell, \omega_i \rangle \wedge f(P, G') \geq \tau\}$.

3 Score-Based Pattern Mining

Our solution to the pattern mining on weighted graphs problem consists of two steps. The first step is the identification of the frequent patterns and their embeddings that satisfy the constraints on the weights imposed by the scoring function used. In the second step, a score is computed for each pattern (and for each weighting function in the case of multi-weighted graphs) in terms of the appearances in its support set that were selected in the first step.

3.1 Assessing the relevance of a pattern

A scoring function can be based on different factors, some of which may be desirable for one applications, while others for another. Thus, there are no scoring function that is consistently better than others, and for this reason we do not advocate for a single one-size-fits-all scoring function. Instead, we propose a framework that can accommodate a wide range of different functions.

Assuming that larger weights indicate higher importance, a desirable scoring function should assigns a large score to patterns that frequent and also have large weights. In particular, we require the scoring function to satisfy the following properties: **(i)** the larger the edge weights in the appearances of a pattern Pin a graph G, the larger the score $f(P,G)$ is; **(ii)** the higher the number of appearances of the pattern P with positive edge weights, the larger the score $f(P,G)$ should again be; and **(iii)** the scoring function f is *MNI-compatible*, i.e., $f(P,G) \geq \tau \implies MNI(P,G) \geq \tau$.

Property **(i)** states that among two patterns with the same number of appearances, the pattern whose appearances have largest edge weights receives the largest score. Property **(ii)** guarantees that when all the appearances have the same edge weights, the pattern with more appearances obtains the largest score. We note that these two properties are a natural consequence of our assumption on the importance of the weights. Last but not least, Property **(iii)** allows efficient implementations of the relevant pattern mining algorithm, as it ensures we can use the same pruning strategies adopted in the MNI-based pattern mining algorithms. Note that, according to **(iii)**, for a pattern to be relevant, it is not enough to be frequent, i.e., $MNI(P,G) \geq \tau$ does not

Algorithm 1 RELEVANTPATTERNMINING

Input: Graph $G : \langle V, E, \ell, \omega \rangle$, score threshold τ
Output: Set of relevant patterns R

```
1: R ← RelevantEdges(G)
2: fE ← FrequentEdges(G)
3: while fE ≠ ∅ do
4:     e ← fE.pop
5:     R ← R ∪ patternExtension(G, e, τ, fE ∪ {e})
6: return R

7: function patternExtension(G, g, τ, fE)
8:     Cand ← ∅; 𝔖 ← ∅
9:     for each e ∈ fE do
10:        Cand ← Cand ∪ {g ⋄ e}
11:    for each c ∈ Cand do
12:        (score, sup) ← examinePattern(G, c)
13:        if sup ≥ τ then
14:            𝔖 ← 𝔖 ∪ patternExtension(G, c, τ, fE)
15:        if score ≥ τ then
16:            𝔖 ← 𝔖 ∪ {c}
17:    return 𝔖
```

guarantee that $f(P,G) \geq \tau$. In Section 6, we introduce a set of scoring functions that satisfy the aforementioned properties.

3.2 Mining weighted graphs

Finding the frequent patterns on weighted graphs requires the computation of the frequency and the score of each pattern. To this end, we propose RESUM, an efficient and effective general algorithm for *any* MNI-compatible score that exploits the pruning power of the anti-monotonicity property of the MNI support.

We model the frequent subgraph mining as a *constraint satisfaction problem* (CSP) [11]. An instance of the CSP problem is a tuple (X,D,C) where X is a set of variables, D is a set of domains corresponding to the variables in X, and C is a set of constraints between the variables in X. A solution for an instance of the CSP is an assignment from the candidates in D to the variables in X that satisfies all the constraints in C. The matching problem for a pattern $P \sqsubseteq G$ is then translated into $CSP(P) = (X_P, D_P, C_P)$, so that any solution for $CSP(P)$ corresponds to a subgraph g isomorphic to P.

Specifically, each node $v \in V_P$ is mapped to a variable $x_v \in X_P$, each domain $D_v \in D_P$ is a subset of V containing all the graph nodes isomorphic to v, and C includes consistency constraints that enforce a topology isomorphic to that of P [30]. Then, for each candidate node $n \in D_v$ we search for a valid assignment that maps n to v. If no assignment is found, n is removed from the domain D_v and the topology constraints are checked again until no invalid candidate is found in the other domains. At the end of the process, the number of elements in the smallest domain, i.e., $\text{argmin}_{D_v \in D_P} |D_v|$, corresponds to the MNI support of P, as defined in Definition 4. Therefore, given a score threshold τ, P is frequent if each variable in X_P has at least τ distinct valid

Algorithm 2 EXAMINEPATTERN

Input: Graph $G{:}\langle V,E,\ell,\omega\rangle$, pattern P, score threshold τ
Output: Score and MNI support of P

1: **for each** $v \in V_P$ **do**
2: $sup_v \leftarrow \emptyset$
3: $D_v \leftarrow \{v' \in V | \ell(v') = \ell(v)\}$
4: $\mathscr{A} \leftarrow$ automorphisms of P
5: STRUCTURALCONSISTENCY($\{D_v | v \in V_P\}, P$)
6: **for each** $v \in V_P$ **do**
7: **if** $\exists w = \mathscr{A}(v)\, s.t.\, D_w$ already computed **then**
8: $D_v \leftarrow D_w$
9: **continue**
10: STRUCTURALCONSISTENCY($\{D_v | v \in V_P\}, P$)
11: **if** $\exists D_u\, s.t. |D_u| < \tau$ **then return** $(-1,-1)$
12: **for each** $n \in D_v$ **do**
13: *search* for g s.t. $g \simeq P \wedge n \in V_g \wedge n \mapsto v$
14: **if** $g \neq Nil$ **then**
15: Valid $\leftarrow$ ISVALID(g,ω)
16: **for each** $n' \in V_g$, $v' \in V_P$ s.t. $n' \mapsto v'$ **do**
17: *mark* n' in $D_{v'}$
18: **if** Valid **then**
19: $sup_{v'} \leftarrow sup_{v'} \cup \{n'\}$
20: **else**
21: remove n from D_v
22: $score \leftarrow$ RELEVANCESCORE($\{sup_v | v \in V_P\}$)
23: $mni \leftarrow min_{v \in V_P} |D_v|$
24: **return** $(score, mni)$

assignments. This means that if the size of some domain D_u is lower than τ, P cannot be frequent. Notice that in general not all the matching subgraphs of a pattern satisfy the constraints on the weights forced by the scoring function used, and thus we must additionally check each of them to determine if it contributes to the score of the pattern. The aggregated score is then computed considering only the matches not discarded.

Algorithm 1 outlines the RESUM framework. First, the relevant and the frequent edges are found (Lines 1-2). Then, each subgraph is recursively extended following the pattern-growth approach introduced by gSpan [48] (Line 5), until no other extension is possible. Each extension is a candidate relevant pattern, whose MNI support is computed alongside its *score* by the EXAMINEPATTERN procedure (Algorithm 2). This procedure first initializes the candidate domain D_v of each pattern node $v \in V_P$ with all the nodes in G with the same label as v (Lines 1-3), and the support set sup_v of each node $v \in V_P$ with the empty set. Then, the algorithm computes the automorphisms of the pattern (Line 4). Automorphisms are isomorphisms of a graph to itself and can be used to compute the valid assignments more efficiently (Lines 7-8), since each assignment valid for a pattern node v is valid for each automorphic node w too. Finally the algorithm iterates over each candidate node $n \in D_v$ to determine if it belongs to some subgraph g isomorphic to P (Lines 12-13). As soon as such subgraph is found, all the domains are updated (Lines 16-17) and the subgraph is checked for validity (Line 15). In particular, the ISVALID procedure compares the edge weights

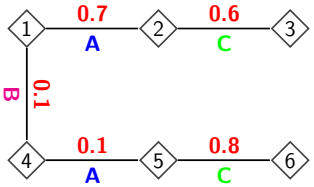

Fig. 3 A weighted labeled graph containing the patterns $P_1 : [v_1] - A - [v_2]$ and $P_2 : [v_1] - A - [v_2] - C - [v_3]$.

in g against the constraints specified by the scoring function f, and if g satisfies the condition, the nodes of the subgraph are stored in the corresponding support sets (Line 19). These nodes will contribute to the relevance *score* of P.

On the other hand, if n does not participate in any isomorphism, it is removed from D_v. As a consequence, in the subsequent iteration, structural constraints like the minimum degree of a node mapped to a $v \in V_P$ are enforced, to remove candidates that can no longer participate to any isomorphism of P (Line 10). The algorithm terminates either when all the pattern nodes have been examined, or when the size of some domain becomes lower than τ, as in this case P can be neither relevant nor frequent (Line 11). In the first case, instead, the MNI support and the relevance *score* of P are calculated and returned. We refer to Section 6 for a discussion about suitable scoring functions that can be implemented in Procedure IsValid.

Finally in Lines 13-17 of Algorithm 1, all the frequent patterns are further extended, while all the relevant patterns are included in the final set of relevant patterns R.

Note that, unlike the MNI support, the scoring function f is not necessarily anti-monotonic. As an example consider the graph G in Figure 3 and a function $f : \mathscr{P} \times \mathscr{G} \to \mathbb{R}$ that counts the number of appearances of P with a large average edge weight. Using the relevance threshold 0.4, the score of the pattern $P_1 : [v_1] - A - [v_2]$ is 1 because only the appearance $[1] - A - [2]$ has average edge weight above 0.4 (0.7). On the other hand, the score of its extension $P_2 : [v_1] - A - [v_2] - C - [v_3]$ is 2 because both the appearances $[1] - A - [2] - C - [3]$ and $[4] - A - [5] - C - [6]$ have a large average edge weight (0.65 and 0.45 respectively). Since the score of a larger pattern can be greater than the score of a smaller one, the function f does not satisfy the apriori property.

As a consequence, we must expand also the patterns with score below τ in order to find all the relevant patterns in the graph. However, Property **iii** guarantees that the score of a pattern is upper bounded by its MNI support, and therefore we can safely call Procedure PatternExtension only for the frequent patterns (Lines 13-14).

Complexity. Even though the computation of the automorphisms ($\mathscr{O}(|V_P|^{|V_P|})$) and the pruning strategy improve the expected performance of the algorithm, in the worst case it takes $C = \mathscr{O}(2^{|V|^2} \cdot |V|^{|V_P|})$ time, which is exponential in the number of nodes and the size of the patterns. In particular, $\mathscr{O}(2^{|V|^2})$ is the time required to compute all the patterns in G, and $\mathscr{O}(|V|^{|V_P|})$ is that needed to find all the isomorphisms of a pattern P.

Algorithm 3 EXAMINESUBGRAPHMULTI

Input: Graph $G{:}\langle V,E,\ell,W\rangle$, pattern P, score threshold τ
Output: Scores and MNI support of P

1: **for each** $v \in V_P$ **do**
2: $D_v \leftarrow \{v' \in V | \ell(v') = \ell(v)\}$
3: **for each** $i \in 1,\ldots,|W|$ **do**
4: $SUP_v[i] \leftarrow \emptyset$
5: $\mathscr{A} \leftarrow$ automorphisms of P
6: STRUCTURALCONSISTENCY($\{D_v | v \in V_P\}, P$)
7: **for each** $v \in V_P$ **do**
8: **if** $\exists w = \mathscr{A}(v)\, s.t.\, D_w$ already computed **then**
9: $D_v \leftarrow D_w$
10: **continue**
11: STRUCTURALCONSISTENCY($\{D_v | v \in V_P\}, P$)
12: **if** $\exists D_u\, s.t.\, |D_u| < \tau$ **then return** $(\{-1,\ldots,-1\}, -1)$
13: **for each** $n \in D_v$ **do**
14: *search* for g s.t. $g \simeq P \wedge n \in V_g \wedge n \mapsto v$
15: **if** $g \neq Nil$ **then**
16: $VAL \leftarrow$ ISVALID(g,W)
17: **for each** $n' \in V_g$, $v' \in V_P$ s.t. $n' \mapsto v'$ **do**
18: *mark* n' in $D_{v'}$
19: **for each** $i \in 1,\ldots,|W|$ **do**
20: **if** $VAL[i]$ **then**
21: $SUP_{v'}[i] \leftarrow SUP_{v'}[i] \cup \{n'\}$
22: **else**
23: remove n from D_v
24: $\mathscr{S} \leftarrow$ RELEVANCESCORES$(\{SUP_v | v \in V_P\})$
25: $mni \leftarrow min_{v \in V_P} |D_v|$
26: **return** $(\mathscr{S}, mni)$

3.3 Mining in multi-weighted graphs

In the case of multiple edge weights assigned by m weighting functions $W = \{\omega_1, \ldots, \omega_m\}$, the naïve approach for solving the Pattern Mining in Multi-Weighted Graphs problem runs Algorithm 1 $|W|$ times, once for each function. This approach becomes impractical for large m, as the process of mining the patterns is computationally intense. In fact, this process would take $\mathscr{O}(C^m)$ to terminate.

The naïve approach recomputes the same patterns multiple times, incurring in a significant time overhead that can be avoided by running the algorithm only once and keeping track of the relevant patterns for each weighting function. This strategy replaces Line 12 in Algorithm 1 with Algorithm 3, which searches for the isomorphisms of the pattern P, while checking their validity with respect to each $\omega_i \in W$, at the same time. Similarly to the single weight case, we initialize each candidate domain and all the support sets for each weighting function (Lines 1-4). When an isomorphic subgraph is found, procedure ISVALID checks *in parallel* each set of edge weights against the constraints set by the scoring function and stores the results in the auxiliary array VAL. If the weights assigned by ω_i satisfy the constraints, the nodes of the subgraph are stored in the corresponding sets $SUP_v[i]$ (Line 21).

Finally, all the scores of the candidate pattern c are evaluated in Line 16 of Algorithm 1, and c is added to the final set R only if at least one of its scores is larger

than τ. As a further optimization, instead of storing in memory the sets of relevant patterns for each function ω_i, we maintain a binary vector of size m for each relevant pattern P, where position i is set to 1 if P is relevant for ω_i.

Complexity. The automorphisms of each pattern P are computed once ($\mathcal{O}(|V|^{|V_P|})$), while the m scores are computed incrementally as new subgraphs isomorphic to P are found ($\mathcal{O}(|V|^{|V_P|} \cdot |V_P| \cdot m)$). Even though the search of isomorphisms stops as soon as all the m scores exceed the relevance threshold, in the worst case we must find all of them. The complexity is therefore $\mathcal{O}(2^{|V|^2} \cdot (|V|^{|V_P|} + m \cdot |V_P| \cdot |V|^{|V_P|})$, which can be approximated to $\mathcal{O}(2^{|V|^2} \cdot m \cdot |V_P| \cdot |V|^{|V_P|})$. If we can assume that the size of the pattern $|V_P|$ is negligible, the complexity becomes $\mathcal{O}(2^{|V|^2} \cdot m \cdot |V|^{|V_P|})$.

4 Approximate Algorithm

The exact algorithm introduced in Section 3 incurs a significant memory overhead when the number of weighting functions is in the order of thousands, which, for example, is the case for recommender systems for big retailers (e.g., Amazon). For such applications, we devise a more conservative approximate solution, called RESUM *approximate*, that significantly reduces the memory consumption by taking advantage of the similarities between the weighting functions $\omega_1, \ldots, \omega_m \in W$.

The RESUM *approximate* algorithm first generates $k \ll m$ representative functions ω_j^*, by clustering and aggregating the original functions ω_i. Then, it runs Algorithm 3 to compute k sets of relevant patterns $R_1^*, \ldots, R_k^*$, which are used to build m approximate sets of relevant patterns $A_1, \ldots, A_m$, returned in place of the exact sets $R_1, \ldots, R_m$. Clearly, the quality of the approximate result depends on the way the representative functions are generated. With our implementation, we aim at returning a set A_j for each ω_i that resembles the exact set R_i as much as possible.

4.1 Generation of the representative functions

The generation of the representative functions is shown in Algorithm 4 and consists of three steps. First, each weighting function $\omega_i \in W$ is transformed into a feature vector (Line 1). Secondly, the weighting functions are clustered into k groups of similar functions (Line 2). Thirdly, the set of k representative functions $W^* = \{\omega_1^*, \ldots, \omega_k^*\}$ is returned (Lines 3-4).

Creation of the feature vectors

In the first step, we construct a feature vector r_i for each ω_i, which is used in the second step to determine the similarities between the functions. Since our final goal is to assign a set of patterns A_i to each ω_i that is as close as possible to the exact set R_i, a straightforward choice is to use the edge weights as features. We call this approach *full-vector strategy*. According to this strategy, Procedure 1 decides an ordering of the graph edges and creates m vectors $\mathbf{r}_1, \ldots, \mathbf{r}_m$ of size $|E|$, where $\mathbf{r}_i[x]$ is the weight assigned by ω_i to the edge in the x^{th} position.

Although similar edge weights lead, with high probability, to similar sets of relevant patterns, the effectiveness and the efficacy of the full-vector strategy decrease as

Algorithm 4 GENERATEREPRESENTATIVEFUNCTIONS

Input: Graph $G : \langle V, E, \ell, W \rangle$, number of buckets b, number of clusters k
Output: Set of representative functions W^*
1: $\mathscr{F} \leftarrow$ CREATEFEATUREVECTORS(E, W, b)
2: $\mathscr{C} \leftarrow$ COMPUTECLUSTERING$(\mathscr{F}, k)$
3: $W^* \leftarrow$ GENERATEMAXWEIGHTVECTORS$(\mathscr{C}, W)$
4: **return** W^*

Algorithm 5 CREATEBUCKETFEATUREVECTORS

1: **function** CREATEBUCKETFEATUREVECTORS(E, W, b)
2: **for each** $l \in \Sigma_E$ **do**
3: $BucketList_l \leftarrow$ COMPUTEBUCKETLIMITS(E_l, W, b)
4: **for each** $\omega_i \in W$ **do**
5: $\mathbf{r}_i^l \leftarrow$ FILLBUCKETS$(E_l, \omega_i, BucketList_l)$
6: **for each** $\omega_i \in W$ **do**
7: $\mathbf{r}_i \leftarrow$ CONCATE$(\{\mathbf{r}_i^l | l \in \Sigma_E\})$
8: **return** $\{\mathbf{r}_1, \ldots, \mathbf{r}_{|W|}\}$

the size of the graph increases. In fact, the high dimensionality of the vectors complicates the detection of functions with similar properties, as a consequence of the curse of dimensionality phenomenon [42].

Thus, we propose also a more efficient approach called *bucket-based strategy*, which overcomes the problem of high dimensionality by considering the edge labels in place of the graph edges, as features to build the vectors. The underlying idea is that, in real scenarios, a preference for an edge is highly correlated with the preference for the label of that edge. This strategy is implemented in Procedure CREATEBUCKETFEATUREVECTORS (Algorithm 5), which takes the set of weighting functions W and the number of buckets b, and generates a set of feature vectors $\mathbf{r}_1, \ldots, \mathbf{r}_m$ each of size $|\Sigma_E| \cdot b$, where Σ_E indicates the set of distinct edge labels. In particular, each vector $\mathbf{r}_i$ is the concatenation of $|\Sigma_E|$ summaries of the edge-weights of ω_i, one for each edge label, and b is the resolution of each summary.

The summary for a label l is obtained by splitting the range of admissible weights $[0,1]$ into b of sub-ranges (buckets) (Line 3), e.g., $[0, x_1)$, $[x_1, x_2)$, and $[x_2, 1.0]$ for $b = 3$. Then Procedure FILLBUCKETS (Line 5) counts, for each sub-range, how many times the function ω_i assigns a weight within that sub-range to an edge with label l. Note that, in the degenerate case of $b = 1$, the vector $\mathbf{r}_i$ simply keeps, for each label, the number of edges with that label and whose weight is greater than 0.

The *bucketization* of a label l is performed by Procedure COMPUTEBUCKETLIMITS (Line 3) following the equi-depth paradigm [15], which assigns the input values to buckets, while trying to balance the number of elements in each bucket. Thus, we consider all the weights assigned by all the weighting functions to edges with label l, and split the range $[0,1]$ into b depth-balanced intervals.

For example, given $b = 2$, the label ordering $A \,|\, C$, and the two weighting functions ω_1 and ω_2 in Figure 1, we obtain the vectors $\mathbf{r}_1 = [1,3,2,0]$ and $\mathbf{r}_2 = [3,1,0,2]$. As such, the buckets of A are the ranges of values $[0, 0.3]$ and $[0.3, 1]$, and those of C the ranges $[0, 0.5]$ and $[0.5, 1]$.

Note that the bucket-based strategy allows us to decide the size of the feature vectors apriori and tune the parameter b to improve the accuracy of the clustering.

Identification of similar functions

Procedure COMPUTECLUSTERING (Algorithm 4, Line 2) implements the Lloyd's clustering algorithm [29], which identifies groups of similar ω_i by comparing the feature vectors $\mathbf{r}_1, \ldots, \mathbf{r}_m \in \mathscr{F}$ using the cosine similarity.

The algorithm can be initialized either providing k random seeds among all the vectors in $\mathscr{F}$, or by selecting the k most diverse feature vectors. Note that finding the most diverse vectors may increase the running time of the algorithm, but this strategy allows the discovery of better separated clusters. Moreover, the algorithm can either be executed until convergence or can be run in iterative steps. In the first case it finds k clusters, while in the second case it runs multiple times with k ranging from 2 to some maximum value k_{max}, and then returns the clustering with largest silhouette coefficient.

Generation of the representative functions

Given the set of clusters $\mathscr{C}$, Procedure GENERATEMAXWEIGHTVECTORS (Algorithm 4, Line 3) generates a representative function ω_j^* for each cluster C_j. Different choices of ω_j^* can lead to different sets of patterns R_j^*, which can contain patterns not relevant for some $\omega_i \in C_j$, as well as missing out patterns relevant for some other $\omega_l \in C_j$. However, as stated in the following theorem, we resort to take the *maximum* among the weights to prevent missing any relevant pattern:

Theorem 1 *Given a cluster C_i, and a MNI-compatible scoring function f, a* complete *set of relevant patterns for C_i can be mined using the representative function ω_i^* defined as $\forall e \in E$, $\omega_i^*(e) = max_{\omega_j \in C_i} \omega_j(e)$.*

Proof By definition, only the subgraphs that satisfy the constraints on the weights through the scoring function f can contribute to the score of a pattern. Moreover, the larger the weights of a subgraph, the higher the chances that such subgraph fulfill those constraints. Since the function ω_i^* assigns to each edge $e \in E$ the largest weight among those of the weighting functions in the cluster C_i, i.e., $\forall \omega_j \in C_i, \omega_i^*(e) \geq \omega_j(e)$, the chances that a matching subgraph contributes to the score of a pattern is higher for ω_i^* than for any $\omega_j \in C_i$. It follows that $\forall \omega_j \in C_i\, f(P, \omega_i^*) \geq f(P, \omega_j)$, so if a pattern is relevant for some $\omega_j \in C_i$, it is also relevant for ω_i^*. Thus, the set of mined patterns is complete.

Given the sets of relevant patterns $R_1^*, \ldots, R_k^*$ discovered by Algorithm 1 using the representative functions $\omega_1^*, \ldots, \omega_k^*$, we create a pattern set A_i for each function ω_i using the patterns in the set R_j^* for $j \leq k$. $\omega_i \in C_j$, i.e., each function ω_i receives the set of relevant patterns of the cluster to which it belongs.

Complexity. The generation of the k representative functions for the m weighting functions requires the creation of the feature vectors ($\mathscr{O}(m \cdot |E|)$), the identification of similar functions ($\mathscr{O}(I \cdot k \cdot m \cdot b \cdot |\Sigma_E|)$, where I is the number of iteration of k-means, and $b \cdot |\Sigma_E|$ is the size of the feature vectors), and the computation of the maximal weights for each edge and for each cluster of functions ($\mathscr{O}(m \cdot |E|)$). Then, the k sets of relevant patterns are found running the algorithm described in Section 3

($\mathcal{O}(2^{|V|^2} \cdot k \cdot |V|^{|V_P|})$). Since k, I and b are negligible, and $|\Sigma_E| = |E|$ in the worst case, the complexity of ReSuM *approximate* reduces to $\mathcal{O}(m \cdot |E| + 2^{|V|^2} \cdot k \cdot |V|^{|V_P|})$.

4.2 Quality of ReSuM *approximate*

The ReSuM *approximate* algorithm reduces the problem of pattern mining in graphs with m weights on each edge to finding k sets of relevant patterns R_j^*, with $k \ll m$. The quality Q of the solution can be measured in different ways, according to the requirements of the user or the application. The most common quality measure used in the literature is the accuracy, which is defined in terms of precision and recall. In our case, since Theorem 1 ensures a total recall, we consider the average precision of the sets A_i with respect to the exact sets R_i:

$$Q = \frac{1}{m} \sum_{i=1}^{m} |R_i \cap A_i| / |R_i| \tag{1}$$

The quality Q can be measured also in terms of the average distance between the patterns in the sets R_i and those in the sets A_i. As shown in Section 8, the distance between two patterns can be calculated using the normalized Levenshtein distance, and the distance between two pattern sets as the average normalized Levenshtein distance among the pairs of closest patterns in the two sets. According to this measure, A_i is a good solution for ω_i if the patterns in A_i have structure and labels similar to the patterns in R_i.

5 Distributed Algorithm

To overcome the challenges of dealing with very large graphs, distributed graph processing systems have been introduced [43,31]. Those systems scale by distributing the computation among multiple machines communicating with each other. Moreover, they are usually designed such that all the details related to the distribution, the message-passing, and the synchronization, are hidden behind simple API that allow non-expert users to implement efficient and scalable algorithms [43,31].

In the following, we show to apply our weighted pattern mining framework in the distributed settings by designing a distributed version of ReSuM. We chose to implement our algorithms on top of Arabesque [43], a framework for distributed graph mining that differs from other existing platforms (e.g., Pregel [31]) in the programming paradigm adopted. In fact, Arabesque follows the Bulk Synchronous Parallel model [44], but centers the computation around the task of searching for embeddings. That is, every machine is delegated to retrieve the appearances of the patterns in the graph. This programming model is specifically designed for the implementation of graph pattern mining algorithms, instead of generic vertex-centric computations (like those supported by Pregel [31]).

5.1 Distributed Relevant Pattern Mining

Since distributed pattern mining algorithms take into account only the frequency of a pattern, to implement relevant pattern mining in this distributed setting, two important extensions are required: an appropriate data-structure for the storage of the embeddings that can keep track of their weights (especially for the case of multiple weighting functions), and the implementation of aggregation functions that for the computation of the MNI-compatible scoring functions. In particular, for the aggregation functions, in the case of multiple weights, it is important to aggregate the support sets of each weighting function.

The computation proceeds via a sequence of supersteps in the Bulk Synchronous Parallel model, where a master coordinates and collects the results from a cluster of workers. Given an initial set of embeddings in the graph, the task of the workers is to identify all the possible expansions of each of them, i.e., embeddings with an additional edge, which will be used to compute the frequency of the corresponding patterns. In the first step of the computation, the initial set contains only a special *undefined* embedding, whose set of expansions is the edge set of the graph. This set is collected by the master as input for the next step of computation. In each of the following supersteps, the master broadcasts the set of embeddings received in the previous superstep, while the workers expand those corresponding to frequent patterns and give back the new expanded embeddings to the master. The computation halts when the new set of embeddings is empty.

Upon receiving the embeddings, the workers use Round Robin on large blocks of embeddings to partition them. A different subset of embeddings is thus assigned to each worker to be filtered and processed. Since the number of embeddings in a graph increases exponentially with the graph and the pattern size, the workers use a special data structure called Overapproximating Directed Acyclic Graph (ODAG) to store them in a compact way. ODAGs trade space for time by over-approximating the set of embeddings they want to store, hence entailing additional work to extract only the actual embeddings from them and avoid the generation of spurious patterns.

Once restored the valid embeddings I, the workers run the procedures shown in Algorithm 6. When processing an embedding e, the worker must first determine if it corresponds to a frequent pattern, since embeddings of infrequent patterns will not be expanded. The frequency values are computed via a MapReduce job (Line 8) where the mappers send the ODAGs of the same pattern to the reducer responsible for that pattern, and the reducers aggregate the domains and the support sets contained in the ODAGs received. The aggregation of the domains (support sets) consists in computing the union of the domains D_v (support sets sup_v) of each vertex v of the pattern P. As described in Section 3, the domains are used to compute the MNI support of P_e, while the support sets to compute its relevance *score*. In the initialization of the support sets of P_e, the mapper runs procedure IsValid to check whether the weights of e satisfy the constraints specified by the scoring function or not. If they pass the validity test, the nodes of e are stored in the support sets; otherwise the sets are left empty.

At the end of the aggregation, if all the domains have size greater than τ (Line 16), the pattern is frequent and thus its embeddings are further processed (Line 2). Sim-

Algorithm 6 DISTRIBUTEDRELEVANTPATTERNMINING

Input: Set of initial embeddings I, score threshold τ
Output: Set of expanded embeddings F
Output: Set of relevant patterns R

```
1: for each e ∈ I do
2:     if AGGREGATIONFILTER(e) then
3:         Cand ← EMBEDDINGEXPANSION(e)
4:         for each e' ∈ Cand do
5:             if ISCANONICAL(e') then
6:                 PROCESS(e')
7:                 F ← F ∪ {e'}
8: PATTERNAGGREGATION(F)

9: function AGGREGATIONFILTER(e)
10:    D_{v_1}, ..., D_{v_n} ← GETDOMAINS(e)
11:    sup_{v_1}, ..., sup_{v_n} ← GETSUPPORTSETS(e)
12:    mni ← min_{v_i} |D_{v_i}|
13:    score ← RELEVANCESCORE({sup_v | v ∈ V_{P_e}})
14:    if score ≥ τ then
15:        R ← R ∪ {P_e}
16:    return mni ≥ τ

17: function PROCESS(e)
18:    MAP(P_e, {D_{v_i}}_{v_i ∈ V_{P_e}}, {sup_{v_i}}_{v_i ∈ V_{P_e}})
19:    REDUCE(P, {D^1_{v_i}, ..., D^m_{v_i}}_{v_i ∈ V_P}, {sup^1_{v_i}, ..., sup^m_{v_i}}_{v_i ∈ V_P})
```

ilarly, if all the size of all the support sets exceeds τ, the pattern is inserted in the relevant pattern set R that will be output to the underlying distributed file system (Line 14). We recall that the MNI support *mni* is the minimum among the sizes of the domains (Line 12), while the evaluation of the *score* depends on the scoring function chosen (Line 13). To speed up the computation, the reducers actually stop merging the values in the domains/ support sets that have already exceeded the threshold, hence terminating the ODAG aggregation when all the domains/ support sets contain enough values.

All the embeddings retained are expanded by Procedure EMBEDDINGEXPANSION(Line 3), which adds one additional edge in all the possible positions. Since the workers have access to a local copy of the graph, they do not need to communicate and exchange information in this phase. Nonetheless, the same embedding can be generated by multiple workers as a result of processing the same set of edges in different orders. To avoid duplicate embeddings, one of the ordering is elected as canonical (Line 5), so that all the others can be safely pruned. All the extensions of canonical embeddings are stored in a set F that will be sent to the master at the end of the superstep (Line 7).

To reduce the amount of messages that will be sent through the network, the workers perform a MapReduce job locally (Line 6) to aggregate the extensions related to the same pattern, hence building an ODAG for that pattern. In the Map phase, the domains D_v and the support sets sup_v of the pattern P_e of e are initialized with the ids of the nodes of e (Line 18). In the reduce phase, the domains and the support sets related to the same pattern P are aggregated in the same way as when running Proce-

dure PATTERNAGGREGATION. To identify the canonical pattern P_e of e, the workers use a technique called two-level pattern aggregation. In the first level, they create a so-called *quick pattern* by scanning all the edges of the embedding and extracting the corresponding labels. In the second level, they compute the canonical pattern of each quick pattern and reorder the list of domains and support sets of the quick pattern according to the canonical vertex ordering. When computing the canonical patterns, the workers also search for automorphisms that will be exploited to insert all the elements in a domain D_v to the domains of the symmetric counterparts of v. Note that the two-level pattern aggregation technique reduces the complexity of Procedure PROCESS, as the graph isomorphism tests required to aggregate the embeddings are performed for the smaller number of quick patterns rather than the larger number of embeddings.

Time complexity. At the beginning of each superstep, each worker must first extract the valid embeddings from the ODAGs, then generate the canonical extensions of the embeddings associated to frequent patterns, and finally produce the ODAGs for the next step. The cost of the first operation is upper bounded by the number of paths contained in the ODAGs, i.e., $\mathcal{O}(|V|^{|V_P|})$. The cost of the second operation is the sum of the cost of computing the frequency of the pattern associated to each ODAG ($\mathcal{O}(|V_P| \cdot |V|^2) = \mathcal{O}(|V|^2)$) and that of creating the canonical extensions of each embedding retained ($\mathcal{O}((|V_P|+1)^{(|V_P|+1)} \cdot |V|^{(|V_P|+1)})$). The last operation consists in performing the two-level pattern aggregation to generate a single ODAG per canonical pattern ($\mathcal{O}((|V_P|+1) \cdot |V|^{(|V_P|+1)})$ to generate the quick patterns, and $\mathcal{O}(|V|^{(|V_P|+1)})$ to generate the canonical patterns). The total time required to perform each superstep is therefore $\mathcal{O}(|V_P|^{(|V_P|+1)} \cdot |V|^{(|V_P|+1)})$.

Space complexity. Each worker has access to a copy of the input graph. In addition, at the beginning of each superstep, it receives every ODAG produced in the previous step, and thus must keep in memory $|V_P|$ vectors of integers. The maximum number of integers to store for all the ODAGs is $|V_P| \cdot |V|^2$.

Machine-to-machine communications. At the beginning of each superstep, the master sends all the ODAGs of the previous step to all the workers. In the worst case, every embedding is associated to a different pattern, and therefore the number of these messages is upper bounded by $num\_workers \cdot |V|^{|V_P|}$. At the end of each superstep, a map-reduce job is executed to aggregate the ODAGs associated to the same canonical pattern, that is $\mathcal{O}(|V|^{(|V_P|+1)})$, as all the ODAGs of the same patterns must be sent to the reducer responsible for that pattern and then all the aggregated ODAGs are sent to the master. The total communication cost is $\mathcal{O}(|V|^{(|V_P|+1)})$.

6 Pattern Evaluation

A number of scoring function satisfying Property **i**, Property **ii**, and Property **iii** can be proposed and implemented in Procedure ISVALID and RELEVANCESCORE in Algorithm 2 and 3. Nevertheless, to demonstrate the flexibility of our framework, we propose here four different scoring functions that can be used to assess the relevance of a pattern in a weighted graph. They are called *ALL*, *ANY*, *SUM* and *AVG*. We chose these functions because of their intuitive semantics and their suitability for various

scenarios that may pose different requirements or provide a different interpretation of the edge weights. Moreover, as they are defined by the MNI support of the pattern over a specific restriction of its support set, they are *MNI-compatible* by definition, and thus they preserve the apriori property.

The *ALL*, *ANY*, *SUM* and *AVG scores* differ in the choice of which subgraphs they include in the support sets of the patterns P and in how they aggregate the edge weights of such subgraphs. In particular, *ALL*, *ANY*, and *SUM* rely on an additional system-dependent parameter, called relevance threshold α, that is used to select the subgraphs that contribute to the *score*, while *AVG* is parameter-free.

In the following we provide a formal definition of the four scoring functions.

ALL The *ALL score* considers only the subgraphs whose edge weights are larger than the threshold α as valid appearances of a pattern P. Specifically, the *ALL score* of P is its MNI support computed over the restricted set of appearances $S'_G(P) = \{g \,|\, g = \langle V_g, E_g, \ell, \omega\rangle \wedge g \in S_G(P) \wedge \forall e \in E_g,\, \omega(e) > \alpha\}$, that is, $f_{ALL}(P,G) = min_{v_P \in V_P} \left|\mathscr{N}(G, v_P) \restriction_{S'_G(P)}\right|$, where $\mathscr{N}(G, v_P) \restriction_{S'_G(P)} = \{v | v \in V \wedge \exists g \in S'_G(P)\,.\, \phi_g^P(v) = v_P\}$ is the restriction of $\mathscr{N}(G, v_P)$ to the subset $S'_G(P) \subseteq S_G(P)$.

In graphs like protein-to-protein interaction networks, this *score* retrieves patterns characterized by an overall confidence greater than a certain value.

ANY The *ANY score* takes into account only the appearances of a pattern having at least one edge with weight above the threshold α. Hence, the *ANY score* of P is the MNI support of P over the set of appearances $S'_G(P) = \{g \,|\, g = \langle V_g, E_g, \ell, \omega\rangle \wedge g \in S_G(P) \wedge \exists e \in E_g\,.\, \omega(e) > \alpha\}$, i.e., $f_{ANY}(P,G) = min_{v_P \in V_P} \left|\mathscr{N}(G, v_P) \restriction_{S'_G(P)}\right|$.

This *score* is suitable especially for the cases in which only partial weights are available (e.g., product reviews for some product), to find patterns that are overall interesting (e.g., the entire transaction comprising the product), as well as super-patterns around relevant core structures.

By definition, the *ANY score* of P is always equal or larger than its *ALL score*, as any appearance of P considered by f_{ALL} is considered also by f_{ANY}, while in general, the opposite is not true. For example, given the graph in Figure 2 and the relevance threshold $\alpha = 0.4$, the subgraph g : [1]–A–[2]–C–[4] does not contribute to the *ALL score* of P : [v_1]–A–[v_2]–C–[v_3], but contributes to its *ANY score*.

SUM For the *SUM score* of P, a subgraph g contributes if the sum of its weights is larger than the threshold α. The restricted support set obtained in this way is $S'_G(P) = \{g \,|\, g = \langle V_g, E_g, \ell, \omega\rangle \wedge g \in S_G(P) \wedge \sum_{e \in E_g} \omega(e) > \alpha\}$. The MNI support over this set is the *SUM score* of P: $f_{SUM}(P,G) = min_{v_P \in V_P} \left|\mathscr{N}(G, v_P) \restriction_{S'_G(P)}\right|$.

This *score* accounts for the overall pattern weight in scenarios like money transactions, where it is beneficial to sum each single contribution in order to judge the complete value of a structure.

Note that if an appearance of P has some weight greater than α, then the sum of all its weights is at least α, and therefore $f_{SUM}(P,G) \geq f_{ANY}(P,G)$. For example, all the appearances considered by *ANY* in computing the score of P : [v_1]–A–[v_2]–A–[v_3] for α=0.4 in Figure 2 are considered also by *SUM*, whereas the subgraph g : [3]–A–[4]–A–[8] contributes to the *SUM score* only.

AVG In contrast to the previous scoring functions, the *AVG score* is not defined in terms of the minimum cardinality among some node sets of the pattern, but in terms of the relative weights of its appearances. In general, the score of a pattern P can be a function of the sum of the weights of the subgraphs in its support set, and this is called the *weighted support* (*WSUP*) of P. In particular, WIGM [50] proposes a measure called *normalized weighted support* (*NWSUP*), which is the weighted support of P divided by its size $|E_P|$, i.e., $NWSUP(G,P) = WSUP(G,P)/|E_P|$. Nevertheless, this scoring function is not MNI-compatible. In order to guarantee the apriori property and be consistent with the other MNI-compatible scoring functions, we compute $WSUP(G,P)$ by first retaining, for each edge set $\mathscr{E}(\mathscr{G},e_P)$ with $e_P \in E_P$, the set $\mathscr{E}(\mathscr{G},e_P)\restriction_\mu$ of μ edges with largest weight, and then summing up all those weights, i.e., $WSUP(G,P) = \sum_{e_P \in E_P} \sum_{e \in \mathscr{E}(\mathscr{G},e_P)\restriction_\mu} \omega(e)$. Setting μ to be the MNI support of P we guarantee that the *AVG score* is bounded by the MNI support, as stated in the following theorem:

Theorem 2 *Given a graph $G{:}\langle V,E,\ell,\omega\rangle$, a pattern P, and an edge $e \in E$, it holds that $f_{AVG}(P \diamond e, G) \leq MNI(P,G)$, where $P \diamond e$ is an extension of P with $E_{P\diamond e} = E_P \cup \{e\}$.*

Proof Since the MNI support has the apriori property [8], it holds that $MNI(P \diamond e,G) \leq MNI(P,G)$. By definition, the pattern $P \diamond e$ has the maximum normalized weight $f^*_{AVG}(P \diamond e, G)$ when all the edges in $\mathscr{E}(\mathscr{G},e)\restriction_\mu$ have weight 1, and hence each subgraph contributes with a total weight of $(|E_P|+1)$. In this case, $f^*_{AVG}(P \diamond e, G) = MNI(P \diamond e,G) \cdot (|E_P|+1)/(|E_P|+1)$, and thus $f_{AVG}(P \diamond e, G) \leq f^*_{AVG}(P \diamond e, G) = MNI(P \diamond e,G) \leq MNI(P,G)$. □

According to this theorem, although *AVG* does not have the apriori property, the *AVG score* of a pattern is at least bounded by the frequency of its sub-patterns, making it MNI-compatible and allowing early pruning during the pattern search. In fact, if the MNI support of P is lower than τ, then all its super-patterns can be discarded. On the other hand, $f_{AVG}(P \diamond e, G)$ can be higher than $f_{AVG}(P,G)$ even though the frequency of $P \diamond e$ is lower, because the weights of the edges in $\mathscr{E}(\mathscr{G},e)\restriction_\mu$ can be so large that they compensate for the lower frequency. For example, the *AVG score* of P : [v_1]–C–[v_2] in the graph G in Figure 2 is 0.6, because $MNI(P,G) = 1$ and $\mathscr{E}(\mathscr{G},C)\restriction_1 = \{(1,4)\}$. Instead, the *AVG score* of P : [v_1]–C–[v_2]–B–[v_3]–A–[v_4] is 0.8, because $\mathscr{E}(\mathscr{G},C)\restriction_1 = \{(1,4)\}$, $\mathscr{E}(\mathscr{G},B)\restriction_1 = \{(1,3)\}$, and $\mathscr{E}(\mathscr{G},A)\restriction_1 = \{(3,5)\}$.

6.1 Implementation

To implement *ALL*, *ANY*, and *SUM* in our framework, function IsValid checks every match g of P in its support set, by comparing its edge weights against the relevance threshold α, according to the corresponding definition of $S'_G(P)$. Then, Procedure RelevanceScore computes the MNI support over the support set $S'_G(P)$. On the other hand, for the *AVG score*, Procedure IsValid returns always *True*, while Procedure RelevanceScore calculates the normalized sum of the top-k edge weights of every pattern edge, where $k = min_{v \in V_g}|D_v|$.

7 Related Work

We survey the main solutions for pattern mining in *graph databases*, *single graphs*, and *probabilistic graphs*. While previous work has tackled the problem of pattern mining in weighted graphs to a certain extent, no solution has been proposed for pattern mining in multi-weighted graphs.

Graph databases. Graph databases are collections of graphs such as chemical compounds, transactions, and workflows. Two main approaches have been proposed for pattern mining in *unweighted* collections of graphs: apriori-based methods, and pattern-growth methods. The *apriori-based* approaches generate frequent structures incrementally, by merging smaller frequent patterns [26]. Pattern-growth methods, on the other hand, generate one structure at a time, expanding each pattern in a depth-first fashion [48, 19].

Regarding *weighted* graphs, a few pattern-growth methods have been recently introduced [21] to embody weights into the support measure. Additionally, WFSM-MR [4] further extends such approaches in a distributed manner on top of the MapReduce framework.

Nevertheless, frequent pattern mining in graph databases employs a support measure, i.e., the number of graphs containing a specific pattern, that cannot be used to mine patterns in large graphs, as each pattern would have a support equal to 1 or 0.

Single Large Graphs. Pattern mining in large graphs requires the support measure to be adjusted to account for edges shared by multiple subgraphs [8]. To this end, alternative support measures satisfying the apriori property have been proposed, alongside efficient algorithms using such measures. SUBDUE [17] is the first pattern mining algorithm in single graphs and adopts an approximate greedy strategy based on the Minimum Description Length (MDL). Other support measures include the maximum number of edge-disjoint matchings [45], the Maximum Independent Set (MIS) support [27], and the Harmful Overlap (HO) [13] support. Nonetheless, the latter two measures require **NP**-complete problems to be solved, rendering them unsuitable in many practical scenarios. In contrast, the Minimum Image-based (MNI) support can be computed efficiently [13]. This measure is used by GraMi [12] and its parallel extension ScaleMine[1], which optimize the computation of the frequent patterns via a constraint satisfaction problem approach. Yet, as opposed to the problem we tackle in this work, GraMi and all the support-based approaches disregard weights on the edges of the graph and do not generalize to the case of multi-weights.

The first work on *weighted* large graphs is WTMaxMiner [14]. However, WTMaxMiner restricts the problem to mining *path* patterns, which can be efficiently discovered as opposed to subgraphs. To the best of our knowledge, WIGM [50] is the only work that deals with weighted pattern mining in large graphs, defining the importance of a pattern as the average weight over its appearances. Although weighted patterns do not naturally possess the apriori property, WIGM adopts a weaker pruning strategy based on the so-called *1-extension property*. Differently from WIGM, our solution, RESUM is scalable and efficient since it uses measures (a.k.a. scoring functions) that satisfy the apriori property and are based on the MNI support. Additionally, RESUM is a more general framework that supports multi-weighted graphs,

				degree	*label frequency*		
dataset	$\|V\|$	$\|E\|$	$\|\Sigma\|$	min/avg/max	min/med/avg/max	τ	α
FREEBASE-T	7.2k	10k	40	1/2.8/504	3/70/251.3/2.8k	90	.05
FREEBASE-C	16.7k	26k	77	1/3.2/1082	1/66/348.5/4.8k	155	.05
AMAZON	163k	296k	4 1710	1/3.6/1072	2k/12k/30k/113k 1/1/95/142k	130	.0001
CITESEER	2.1k	3.6k	21	1/3.5/99	15/55/174.7/988	95	.05
FREEBASE-O	1.9M	2.4M	19294	1/2.4/46k	1/1/103/237k	6000	.05
SHOP-S	11k	12k	80 24	1/3/35	1/60/161/3k 1/100/467/2.8k	76	.05
SHOP-M	163k	296k	81 24	1/3/129	3/606/1.6k/30k 6/1k/4.6k/28k	759	.05
SHOP-L	1.1M	1.2M	81 24	1/3/583	5/5.9k/16k/305k 60/10k/46k/280k	7580	.05
SHOP-XL	11M	13M	81 24	1/3/3868	115/60k/160k/3M 600/100k/467k/2.8M	76124	.05

Table 1 Real (top) and synthetic (bottom) datasets with default τ, α parameters.

as well as a broad family of scoring functions, showcasing the WIGM support measure as one example (see Section 6).

Uncertain graphs. Uncertain graphs include existence probabilities for edges or nodes of the graph. To some extent, uncertain graphs can be seen as a special case of weighted graphs in which probabilities arises, for instance, from random walk approaches, and represent the likelihood that an edge exists between two nodes. Few works have been proposed to mine frequent patterns in uncertain graphs [52,20,36,9, 47,28]. As opposed to weighted graphs, support measures for uncertain graphs must consider the uncertainty in the edges and compute the support as an expected value. Moreover, the time complexity of mining in such graphs is exponential in the worst case, since any edge can either exists or not, and hence all the possible combinations must be considered.

8 Experiments

We first compare the scalability of our exact algorithm with the performance of our approximate algorithm. The results demonstrate that RESUM *approximate* allows faster response time, yet retaining good accuracy in terms of the patterns returned. We then study the behavior of RESUM *distributed* under different settings to identify in which cases we can benefit more from the distribution, as well as understanding when the overhead of a distributed system may lead to performances worse than those of single-machine algorithms [32].

Datasets. We run experiments on both real and synthetic datasets of different sizes, and in particular, we used five real networks and four randomly generated graphs. All the datasets are listed in Table 1 together with their characteristics, i.e., the number of vertices $|V|$, edges $|E|$, and labels $|\Sigma|$; the minimum, average, and maximum node degree; and the minimum, median, average, and maximum edge label frequency.

For the AMAZON and the synthetic datasets we report statistics for both edge (top) and node labels (bottom). We also report the default frequency (τ) and relevance (α) values used in the experiments (unless otherwise stated). Experiments on the quality

of RESUM and RESUM *approximate*, and on their scalability, were conducted on the first four real datasets; the scalability of RESUM and RESUM *distributed* was tested on the last two real datasets and the synthetic datasets.

• FREEBASE-T and FREEBASE-C are directed subgraphs extracted from the knowledge graph FreeBase [2], which is a database collecting structured information about real-world entities like people, places and things for various topics. We obtained the two samples by restricting the graph to the topic *travel* and *computer* respectively, and then we kept only the largest weakly connected component in the restriction.

• AMAZON[3] [16] is a directed graph representing items, purchases, and user ratings. We considered the subgraph of electronic products, in which every node represents a product, a category, or a brand, and a link represents items bought together, bought in subsequent transactions, or viewed on the website one after the other. Weights represent individual user review scores (from 1 to 5), and we considered only users with more than 100 reviews. Given the sparsity of the weights, we used Personalized PageRank to spread the user preferences to products other than those they rated, as it is a standard technique for recommendations [2]. In this way we obtained weights not only for the items reviewed, but also for the most related items. Each edge weight is actually computed as the average between the PageRank value of its endpoint nodes.

• CITESEER [12], is a graph representing Computer Science publications and citations between them. The labels on the edges indicate the area in which the two papers were published (e.g., a database conference).

• FREEBASE-O is a undirected, node-labeled subgraph extracted from FreeBase and restricted to the topics *organizations*, *business*, *finance*, and *government*. The node labels refer to the types of nodes, obtained by following "*instance of*" edges.

• SHOP-S, SHOP-M, SHOP-L, and SHOP-XL are synthetic graphs generated using the *gMark* framework [5], which creates labeled graphs with a user-defined schema that specifies constraints on the number of nodes and labels, the proportions of nodes and edges per label, and the degree distribution. We used the *shop.xml* schema provided by the framework, which encodes an online shop network consisting of sellers, users and products, according to the specifications in the WatDiv default schema [3]. This schema contains 24 node labels, 82 edge labels, default probabilities for each label, and specifies a different degree distribution (uniform, Gaussian, or Zipfian) for each combination of node and edge labels allowed.

Experimental setup. RESUM is implemented in Java 1.8 on top of the constraint satisfaction problem presented in GRAMI [12] whose code was kindly provided by the authors[4]. The code of our implementation and all the datasets we used are publicly available[5]. We also compare with a frequent pattern mining approach (FREQ) based on GRAMI, which is also implemented in Java 1.8. All the single-machine experiments were run on a 24 Cores (2.40GHz) Intel Xeon $E5-2440$ with 188Gb RAM with Linux 3.13.

[2] developers.google.com/Freebase/data

[3] jmcauley.ucsd.edu/data/amazon/

[4] github.com/ehab-abdelhamid/GraMi

[5] https://github.com/lady-bluecopper/ReSuM

RESUM *distributed* is implemented in Java 1.8 on top of the Arabesque framework, which is in turn built as a layer on top of Apache Spark [51] (v. 2.0.0). All the multi-machine experiments were run on a cluster of 7 machines: the master is a 8 Cores machine with 80Gb RAM and Linux 14.04, while the workers are 16 Cores machines with 30Gb RAM and Linux 14.04.

Generating the weights. Since we had real weights only for the AMAZON graph, to test the scalability of our method with a larger number of weighting functions, for the other datasets we created synthetic weights based on the results of a user study we conducted on the Crowdflower[6] platform. We extracted a sample from the FreeBase knowledge base, restricting the domain of the edge labels to five topics (Music, Books, Celebrities, Movies, and Sport). Then we asked the users to rate each graph edge (i.e., fact) according to their preferences, using a relevance value between 1 and 5. Once collected the relevance values from 123 users, we modeled the distribution of the edge weights with respect to the number of facts. We found that the edge weights, after normalization, are distributed as a Gaussian with mean 0.452 and variance 0.02. In addition, we noted that, on average, a user rated above 0 between 10% and 20% of the labels, and thus we concluded that real graph weights are usually quite sparse. Therefore, we uniformly subset edge labels according to our findings and generated weights normally distributed in $[0,1]$.

Furthermore, in order to evaluate the performance of RESUM and RESUM *approximate* with different weight distributions, we generated sets of synthetic edge weights, varying a *focus* parameter representing the ratio of weighted edges for each edge label. The edge weights were sampled from a normal distribution $\mathcal{N}(\mu,\sigma^2)$ and a $Beta(\alpha,\beta)$ distribution, hence allowing us to prove the effectiveness of our algorithms under normally distributed weights and exponentially distributed weights. We set $\mu = 0.5$ and $\sigma = 0.25$ for the normal distribution and $\alpha = 0.7, \beta = 5$ and $\beta = 0.7, \alpha = 5$, for the *Beta* distribution. The two choices of the parameters for the *Beta* distribution represent two extreme of an exponential behavior: the former concentrates the probability mass on low weights, the latter on large weights. The *focus* parameter takes values in the range $\{0.5, 0.8\}$ for the normal distribution and in the range $\{0.25, 0.5, 0.75, 1\}$ for the *Beta* distribution.

8.1 Frequent vs Weighted Pattern Mining

We compared the patterns returned by a frequent pattern mining algorithm (FREQ) and our algorithm RESUM to validate our claim that frequent pattern mining returns a large number of low-weight patterns, which, instead, are correctly discarded in relevant pattern mining. Unless otherwise stated, we report the average of 10 different randomly sampled weighting functions. In particular, these weights were sampled from a normal distribution using *focus* 0.5, as previously described.

Figure 4 reports the average number of patterns found using different scoring functions on the four datasets, with default parameters, as shown in Table 1. We observe that FREQ returns patterns, at least half of which are irrelevant with respect

[6] www.crowdflower.com

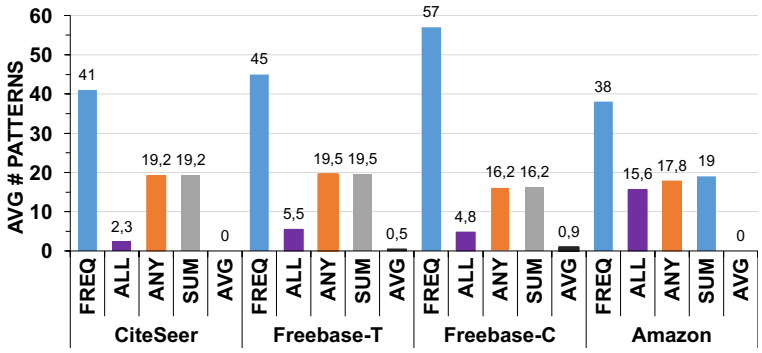

Fig. 4 Average number of patterns found in each dataset, using different *scores* and default parameters.

	FREEBASE-C				FREEBASE-T			
top-*k*	*ALL*	*ANY*	*SUM*	*AVG*	*ALL*	*ANY*	*SUM*	*AVG*
1	0.6	0.6	0.6	0.86	0.5	0.5	0.5	1
3	0.43	0.43	0.43	1	0.45	0.33	0.33	1
10	0.44	0.49	0.49	1	0.8	0.66	0.66	1

Table 2 Quality of FREQ vs RESUM on the top-*k* patterns.

to any of the four scoring functions. As expected, in all the datasets, *ANY* and *SUM* return more patterns than *ALL* and *AVG*, due to the less restrictive conditions on the weights. On the other hand, *AVG* returns a low number of patterns, mainly because more than 50% of the edges have low or zero weight. Therefore, *AVG* is particularly suited in graphs where weights are uniformly distributed in the entire graph, e.g., biological or chemical datasets.

We now discuss quality (Table 2), number of patterns, and running time of RESUM compared to FREQ, when varying relevance (α) and frequency (τ) threshold (Figure 5 and 6). Here we report results for two datasets (FREEBASE-C and FREEBASE-T), however we observe similar results also on the other datasets. In particular, as an example, within the top-5 *frequent* patterns in the AMAZON graph, we found that the most frequently bought products are Sony appliances, but some *relevant* patterns actually involve Nikon products. This result shows that Sony products are popular but not interesting for all the users.

Quality of FREQ vs RESUM. Table 2 shows the quality of the patterns discovered by FREQ, measured on the k most frequent patterns. We selected 10 random weighting functions and mined the relevant patterns for each of them. The quality of FREQ is measured as the average Jaccard similarity between the top-k frequent patterns and the top-k relevant patterns. As expected, frequency is a bad predictor of relevance, since most of the relevant patterns are not in top-k frequent patterns. Notably, for *AVG* the quality is higher mostly due to the small or null number of patterns returned, as reported in Figure 4.

Relevance threshold (α). Recall that the relevance threshold α is a system-dependent parameter set only for *ALL*, *ANY*, and *SUM*. It can be easily tuned on demand and strongly affects the number of patterns (Figure 5 (a) and Figure 5 (b)), because

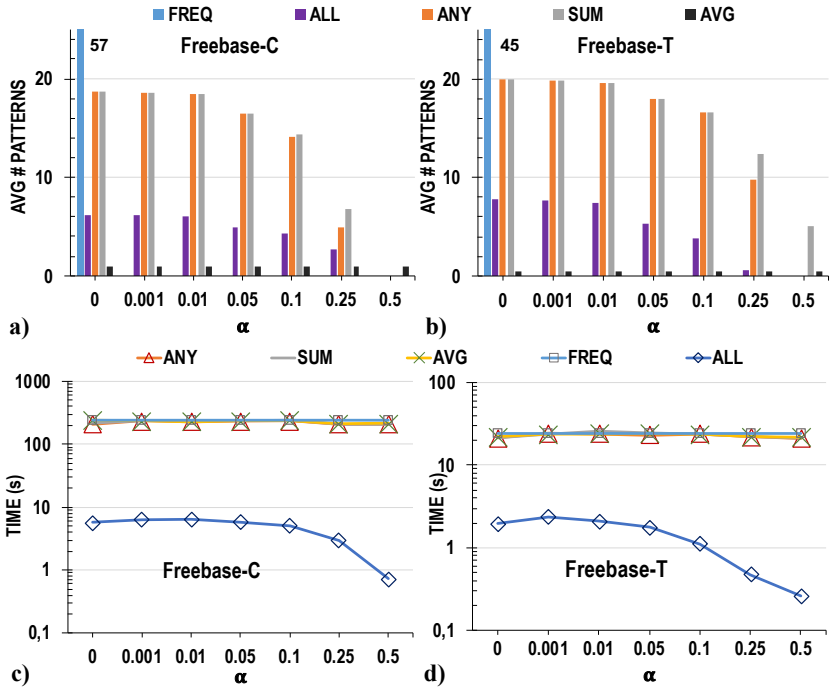

Fig. 5 Varying α: average number of patterns (top) and running time (bottom) in FREEBASE-C (a,c) and FREEBASE-T (b,d).

the larger the value of α, the smaller is the number of appearances that are considered valid, and thus the smaller is the total number of relevant patterns mined. We observe that with α>0 the number of relevant patterns is less than half of the number of the frequent ones. This behavior reflects the characteristics of the weights in the datasets, as half of the edges have zero weight. Moreover, for FREEBASE-T, *SUM*, being the most lenient scoring function, returns patterns even in the restrictive cases when $\alpha >$ 0.5 (Figure 5 (b)). Finally, since *AVG* does not depend on α, it always returns the same patterns.

Figure 5 (c) and Figure 5 (d) show that the threshold α affects the running time of RESUM mostly when *ALL* is used, as this function can prune the irrelevant patterns earlier in the process. In fact, an occurrence of a pattern is discarded and not included in the support set of any extension of the pattern, as soon as one edge weight is found to be below α. On the other hand, for all the other scoring functions, the extension of an invalid occurrence of a pattern can be valid for some super-pattern, and therefore cannot be discarded until all its edge weights have been examined. As a consequence, the running time of the algorithm is almost unaffected by α.

Frequency threshold (τ). Figure 6 reports the behavior of RESUM and FREQ when varying the frequency threshold τ. We performed preliminary tests to decide a reasonable range of values $[\tau_{min}, \tau_{max}]$ for each dataset. In particular, the τ_{min} corresponds to the smallest value that allowed FREQ to terminate the computation within 48 hours, and τ_{max} is the maximum value returning a non-empty set of frequent patterns. The

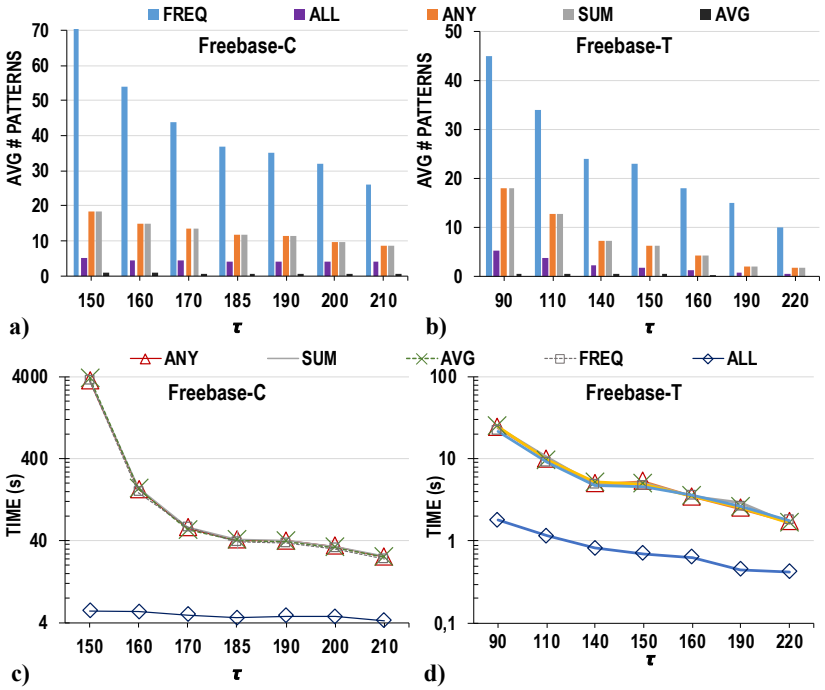

Fig. 6 Varying τ: average number of patterns (top) and running time (bottom) in FREEBASE-C (a,c) and FREEBASE-T (b,d).

choice of different ranges for each dataset is consistent with previous researches [12] and reflects the observation that pattern frequency is dataset-dependent, while relevance is user-dependent.

As we can see in Figure 6 (a) and Figure 6 (b), the number of frequent patterns decreases almost linearly with τ, and consequently the number of relevant patterns decreases as well. Regarding the performance, as opposed to the relevance threshold, the frequency threshold always alters the computation time, since higher values lead to an early pruning of many patterns, and thus the algorithm terminates earlier. Moreover, Figure 6 (c) and Figure 6 (d) show that when τ takes low values (i.e. between 150 and 180), RESUM runs up to two orders of magnitude faster in both the datasets. Finally, as previously noted, *ALL* performs significantly better than the other scoring functions.

8.2 Multiple Weighting Functions

We tested the scalability of RESUM in the case of multiple weighting functions, varying their number between 50 and 50.000 in the real graphs, and between 1 and 1000 in the synthetic graphs. Similarly, we also measured time and quality of RESUM *approximate*. Nevertheless, in the following we do not further discuss and report the number of patterns retrieved for each weighting function and each scoring function, since these results are consistent with what reported in the single edge weight case.

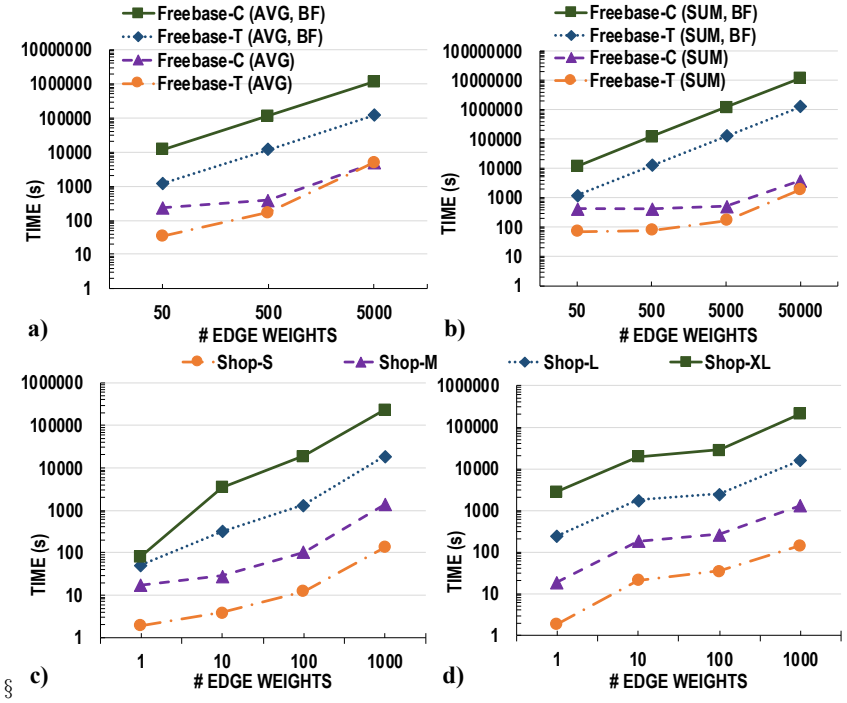

Fig. 7 Scalability of ReSuM: running time in Freebase-C and Freebase-T compared with the brute-force approach (BF), varying number of edge weights, using *AVG* (a) and *SUM* (b); and running time in Shop-S, Shop-M, Shop-L, and Shop-XL, varying number of edge weights, using *AVG* (c) and *SUM* (d).

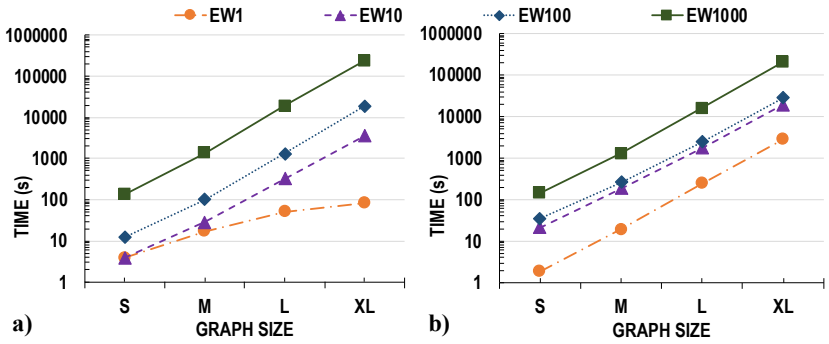

Fig. 8 Scalability of ReSuM: running time using *AVG* (a) and *SUM* (b) with a single edge weight (EW1) and multiple edge weights (EW10, EW100, EW1000), varying the size of the graph.

Scalability: real graphs. Figure 7 shows the impact of the number of weighting functions on the running time. We report the performance obtained with weights sampled from a normal distribution with *focus* 0.5. Figure 7 (a,b) presents the comparison between ReSuM and the brute-force (BF) approach, which computes the patterns for each weighting function separately. While BF scales linearly with the number

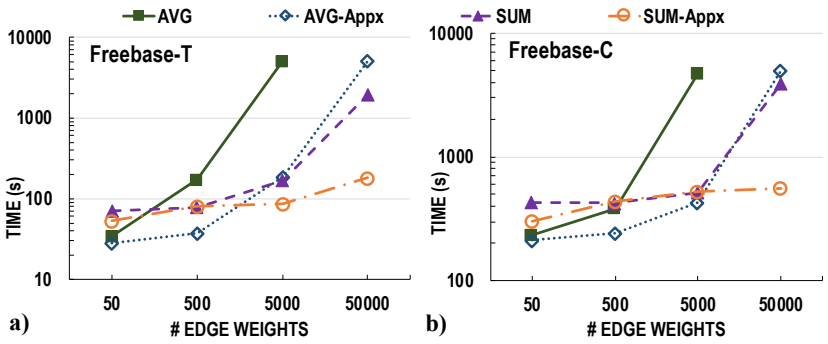

Fig. 9 Scalability of RESUM and RESUM *approximate*: running time in FREEBASE-T (a) and FREEBASE-C (b) using *AVG* and *SUM*, varying number of edge weights.

of weighting functions, the running time of RESUM is nearly constant with 5000 functions, and slowly increases as the number of edge weights approaches 50000. As a pitfall, the memory requirement grows linearly with the number of weights for both algorithms. Note that RESUM keeps all the edge weights and the scores in main memory to speed up the score computation and the pattern evaluation respectively, and thus the number of weighting functions it can handle heavily depends on the available memory. On our machine, we were able to process up to 5000 functions when using the *AVG* score (Figure 7 (a)), while we were able to scale larger than 5000 when using the *SUM* score (Figure 7 (b)).

Scalability: synthetic graphs. The synthetic graphs were generated using the same degree distribution, and assigning the node and edge labels proportionally to their size. As a consequence, they display relatively similar characteristics and can thus be effectively used to test the scalability of our approaches in terms of the input size only. Figure 7(c,d) shows the performances of RESUM in both the single edge weight and the multi-weighted edge setting, when using the *AVG* (c) and the *SUM* (d) scoring functions. The weights were generated using a Beta distribution with parameters $\alpha = 0.7$, $\beta = 5$, and *focus* 0.75. Figure 8 (a,b) shows that adding one order of magnitude to the size of the graph causes a performance degradation by up to one order of magnitude for all the edge weight settings (EW), and this one order of magnitude difference is maintained when increasing the number of weights per edge (Figure 7(c,d)). On the other hand, an increase in the number of weights do not lead to an equally steep increase in the running time.

Finally, we note that the performance with *AVG* is comparable to that of *SUM*, even though *AVG* requires the algorithm to find all the embeddings of the pattern, while *SUM* terminates the algorithm as soon as enough embeddings are found.

In Figure 9 (a, b) instead, we compare RESUM and RESUM *approximate*. For these set of experiments, we generated the representative functions by first clustering the weighting functions using the bucket-based strategy. The clustering phase is performed as a preprocessing and not reported, since it is agnostic to the choice of the various thresholds and depends solely on the clustering algorithm (e.g., k-means, hierarchical, or spectral). In particular, we tried numbers of buckets b of different orders of magnitude and proportional to the frequency of the edge labels in the graph.

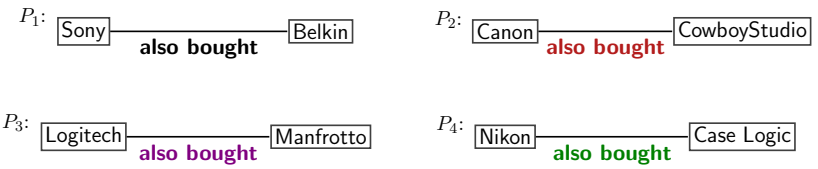

Fig. 10 Case study: the most frequent pattern (above left), the most relevant pattern (above right), and two top-5 patterns (below left and right) found in AMAZON.

Then, we run k-means using different k to study the impact of the number of clusters on the quality and the running time of RESUM *approximate*. Finally, we set the default value of b of each dataset to the number of buckets that allowed the algorithm to use at least one order of magnitude less memory than those consumed using the full-vector strategy, i.e., 12 buckets for FREEBASE-T, 16 for FREEBASE-C, and 10 for CITESEER.

We observe that RESUM becomes impractical as the number of weighting functions increases. As a matter of fact, when *AVG* is used, RESUM exhausts the available memory, hence returning no patterns. This behavior reflects the characteristics of *AVG*, which requires the algorithm to exhaustively search for all the occurrences of a pattern before computing its score. In contrast, RESUM *approximate* terminates the computation. On the other hand, when *SUM* is used, RESUM is able to return the relevant patterns; however, RESUM *approximate* outperforms the exact algorithm again, taking nearly constant time to terminate. In conclusion, in all the cases of large numbers of weighting functions, RESUM *approximate* performs better than RESUM by at least one order of magnitude.

Effectiveness of RESUM: Case study. Figure 10 reports the most frequent pattern (P_1), the most relevant pattern for two randomly selected users u_1 and u_2 (P_2), and a pattern in the top-5 relevant patterns for u_1 and u_2 (P_3 and P_4 respectively), found in the real network Amazon using the *ALL score* and the default settings in Table 1.

Pattern P_1 shows that users who bought Sony products, frequently bought also Belkin products in other transactions. This result makes sense, considering that Sony sells electronics and Belkin accessories for computers, smartphones, and cameras. On the other hand, pattern P_2 confirms our claim that frequent patterns are not necessarily the most interesting patterns for every user, as it contains other less popular node labels, which indicate a more professional user. The high relevance score of P_2 follows from the high ratings that users u_1 and u_2 gave to their Canon and Cowboy Studio purchases. Note that Cowboy Studio is a US retailer selling camera accessories such as tripods, lens, batteries, and flashes, and thus, the appearance of these two node labels in the same pattern is realistic. Thanks to pattern P_2 we know that user u_1 prefers the more professional Cowboy Studio accessories, and thus, if she buys a new camera, we can recommend her a portrait umbrella rather than a cover from Belkin.

Finally, patterns P_3 and P_4 prove that different users like different products, and in particular, u_1 expressed her preference for Logitech and Manfrotto, while u_2 liked Case Logic accessories for her Nikon purchases. As a consequence, we attest that our algorithms can effectively help the design of personalized recommending systems.

	average pattern edit distance			
$\|W\|$	*ALL*	*ANY*	*SUM*	*AVG*
50	0.195	0.069	0.069	0.627
500	0.192	0.062	0.062	0.618
5000	0.203	0.053	0.053	0.609
50000	0.204	0.052	0.051	–

	average pattern edit distance			
clustering	*ALL*	*ANY*	*SUM*	*AVG*
	FREEBASE-T			
A-POST	0.2	0.07	0.07	0.7
BUCK	0.2	0.06	0.06	0.62
clustering	FREEBASE-C			
A-POST	0.28	0.07	0.07	0.45
BUCK	0.27	0.07	0.07	0.39

Table 3 Quality of RESUM *approximate* varying the number of edge weights $|W|$ in FREEBASE-T (left), and using BUCK and A-POST clustering in FREEBASE-T and FREEBASE-C (right).

Quality of RESUM *approximate*. As mentioned in Section 4, we measure the quality of RESUM *approximate* in terms of the average distance between the patterns it returns (sets A_i) and those returned by RESUM (sets R_i). We define the distance between two patterns as the minimum number of edges that should be added or removed from the first to transform it into the second. Thus, the average distance between the two sets of patterns $\{A_1,\ldots,A_m\}$ and $\{R_1,\ldots,R_m\}$ measures the average number of operations required to transform a pattern in A_i to a pattern in R_i. We recall that our method is complete, and therefore no relevant pattern is missing. However, RESUM *approximate* may return spurious patterns, which are patterns not relevant for any function in the cluster. Computing the distance between the two pattern sets allows us to understand how much a spurious pattern, on average, differs from the patterns that are actually relevant for some weighting function in the cluster. Table 3 (left) reports the distances obtained using the four scoring functions in FREEBASE-T. Here, *ANY* and *SUM* exhibit the best quality; *ALL* performs reasonably good, despite being more restrictive and therefore more sensitive to the approximation based on the maximum edge weights. On the other hand, when *AVG* is used, the quality of the answer is quite poor. Nevertheless, this behavior is due to the extremely low number of patterns this scoring function considers interesting, which skews the computation of the pattern set distance. Note that, we do no report any value for the case of 50000 weighting functions with *AVG*, since the algorithm exhausted all the available memory and did not terminate. We conclude that, the additional patterns returned by RESUM *approximate* are indeed closely related to the relevant patterns of each individual weighting function.

Finally, we tested the capability of our bucket-based clustering (BUCK in short) to correctly identify groups of similar weighting functions. To this end, we compared the quality of the results mined using BUCK in the creation of the feature vectors of the weighting functions, with the quality measured using a *ground-truth* clustering (A-POST in short). The A-POST clustering was created using the sets of relevant patterns $R_1,\ldots,R_m$ as feature vectors of $\omega_1,\ldots,\omega_m$, and then running a k-medoid algorithm. We regard it as a ground-truth clustering, because it is obtained knowing what makes two weighting functions really similar, i.e. their relevant patterns, and maximizing the intra-cluster similarity. Table 3 (right) reports the comparison between A-POST and BUCK on FREEBASE-C and FREEBASE-T. We recall that lower values mean higher quality, as they indicate distances. We can see that we experience a quality comparable with that obtained using A-POST, and thus we can conclude that our clustering technique is indeed effective.

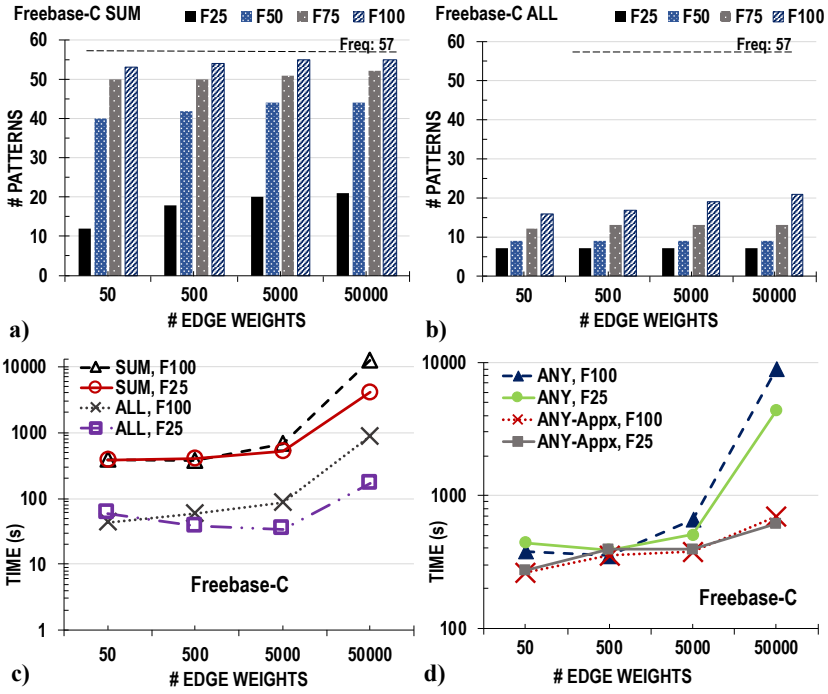

Fig. 11 Varying *focus* in FREEBASE-C: number of patterns using *SUM* (a) and *ALL* (b) with *focus* between 0.25 (F25) and 1 (F100); and running time of RESUM and RESUM *approximate* with $Beta(0.7,5)$ weights with *focus* 0.25 (F25) and 1 (F100), using *SUM*, *ALL* (c), and *ANY* (d).

Impact of the Weights. For the experiments presented above, we weighted the Amazon graph using real weights, and the FREEBASE-T, FREEBASE-C, and CITESEER graphs with synthetic weights generated according to the results of our user study. The common feature of these two kinds of weights is that they are highly sparse. It is worth studying whether weights following other distributions or that are denser, affect the performance of our algorithms. To this end, we performed an additional set of experiments using weighting functions generated following a $Beta(5,0.7)$, a $Beta(0.7,5)$ and a normal distribution with different densities (*focus*), as described at the beginning of Section 8.

One would expect that, with higher densities, the cost of the computation would be higher too. Although these expectations are reasonable, in the following we show that the behavior of RESUM and RESUM *approximate* is consistent with what observed in the case of sparse weights. Figure 11 (a) and Figure 11 (b) report the average number of patterns found using *SUM* and *ALL*, with weights generated using a $Beta(0.7,5)$ distribution with *focus* varying between 0.25 and 1 (i.e., all edges have weight > 0). Comparing these results with those in Figure 4 when *SUM* is used, we can see that the number of relevant patterns is largely affected by the presence of more (or all) edges with non-null weight, meaning that the patterns mined are actually many more. On the other hand, when *ALL* is used, RESUM still finds a larger number of relevant patterns, but the increment is not as large as in the *SUM* case.

average precision

	ALL				*SUM*				*ALL*		*SUM*	
	0.25	0.5	0.75	1	0.25	0.5	0.75	1	0.5	0.8	0.5	0.8
$\|W\|$	$Beta(0.7,5)$								$\mathcal{N}(0.5,0.25)$			
50	0.22	0.20	0.21	0.26	0.17	0.53	0.74	0.91	0.15	0.18	0.36	0.54
500	0.21	0.21	0.24	0.27	0.18	0.53	0.74	0.91	0.18	0.20	0.44	0.57
5000	0.21	0.22	0.24	0.28	0.19	0.54	0.75	0.91	0.22	0.20	0.49	0.59
50000	0.21	0.22	0.25	0.28	0.19	0.54	0.75	0.91	0.22	0.21	0.51	0.59
$\|W\|$	$Beta(5,0.7)$											
50	0.19	0.23	0.42	1	0.34	0.73	0.94	1				
500	0.21	0.24	0.42	1	0.36	0.73	0.94	1				
5000	0.21	0.25	0.43	1	0.36	0.74	0.95	1				
50000	0.21	0.25	0.44	0.99	0.36	0.74	0.95	1				

Table 4 Quality of RESUM *approximate* with *ALL* and *SUM* on FREEBASE-C, with $Beta(\alpha,\beta)$ and normal $\mathcal{N}(\mu,\sigma^2)$ weights generated using *focus* values in $\{0.25,0.5,0.75,1\}$ and $\{0.5,0.8\}$ respectively.

Regarding the running time, Figure 11 (c) and Figure 11 (d) show that the two algorithms behave accordingly to what already seen in the previous experiments, meaning that the fact there more patterns are mined do not downgrade the performance heavily.

Finally, Table 4 displays the quality of RESUM in terms of average precision, as defined in Equation 1. As we can see, our approximate algorithm achieves similar quality values no matter which weight distribution is chosen. In addition, the denser the weights in the graph, the higher is the average precision of the pattern sets mined. Intuitively, this is due to the fact knowing a larger number of positive weights allows the clustering algorithm to better detect which weighting functions are similar.

8.3 Distributed vs Centralized Pattern Mining

We first investigate in which cases the distributed algorithm (noted as Dist) offers an advantage over the centralized one. When running both algorithms on the CITESEER graph, Figure 12 (a,c) shows that the distributed version is always one order of magnitude slower. It is important to note that CITESEER is a small and relatively dense graph with few labels. This small number of labels translates into a small number of candidate relevant patterns with large sets of matching embeddings. In this particular type of graphs, the centralized algorithm can exploit the early termination condition (Algorithm 2) and effectively exploit the CSP problem formulation, hence stopping the materialization of new embeddings for a pattern as soon as the embeddings generated are sufficient to verify that the pattern has a high relevance score. On the other hand, the distributed algorithm has each worker looking for embeddings separately. Since the embeddings are merged only at the end of the computation step, the algorithm cannot exploit the early termination condition, hence computing far more embeddings than necessary.

RESUM *distributed*, instead, provides a clear advantage when we move to the larger and richer FREEBASE-O graph. This graph has a higher number of labels, which, paired with the larger size of the graph, allows for workers to share effectively the workload and provide the answer in more than one tenth of the time required by

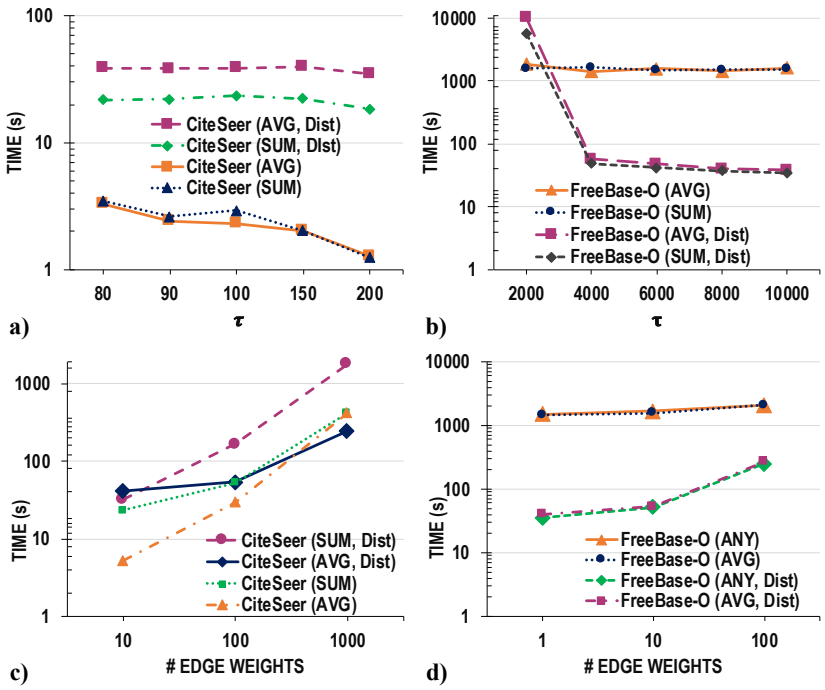

Fig. 12 RESUM vs RESUM *distributed*: running time in CITESEER (a) and FREEBASE-O (b) varying τ, using *SUM* and *AVG*; and running time in CITESEER (c) and FREEBASE-O (d) varying number of edge weights, using *SUM* and *AVG*.

the centralized version (Figure 12 (b)). The striking difference between the performance of the two algorithms in the two datasets, suggests that the distributed version has to be preferred for larger and richer graphs, when many different patterns can be retrieved. The same behavior is observed when changing the number of users (Figure 12 (d)), proving that our strategy for the multi-weight pattern mining is still effective in the distributed environment.

Varying number of workers. In order to better understand which dimensions affect the most the performance of the distributed algorithm, we compared its running times over both the CITESEER and the FREEBASE-O graph, when varying the number of workers[7] Figure 13 (a) shows that, with one or two workers, the RESUM *distributed* is sensibly slower on the much smaller citation network than on the larger knowledge graph. Note that for FREEBASE-O the algorithm returned around 30 relevant patterns, while for CiteSeer it retrieved 66 patterns. Only when using 6 machines we were able to run faster on the smaller graph.

In addition, Figure 13 (b) shows that the effects of the distribution remain the same when varying the number of weighting functions we consider. This substantiates our previous finding, namely, when the graph contains few very frequent patterns the distribution strategy provided by Arabesque is not optimal as it is dominated

[7]For this experiment we kept a single edge-weighting function, and parameter $\alpha = 0.05$, with $\tau = 6000$ for FREEBASE-O, and $\tau = 100$ for CITESEER.

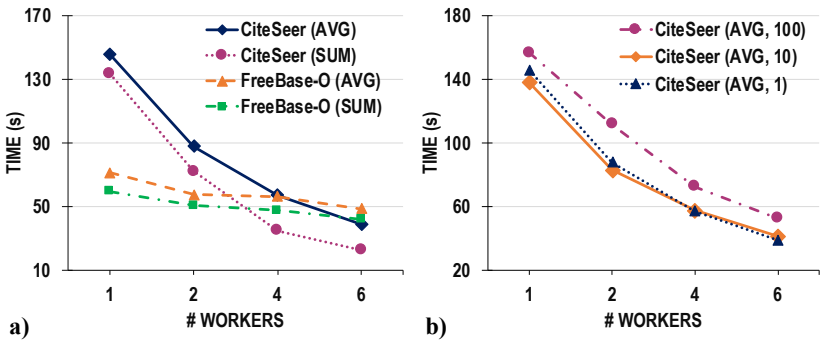

Fig. 13 Varying number of workers: running time of RESUM *distributed* in FREEBASE using *SUM* and *AVG* with single edge weights (a) and using *AVG* varying number of edge weights.

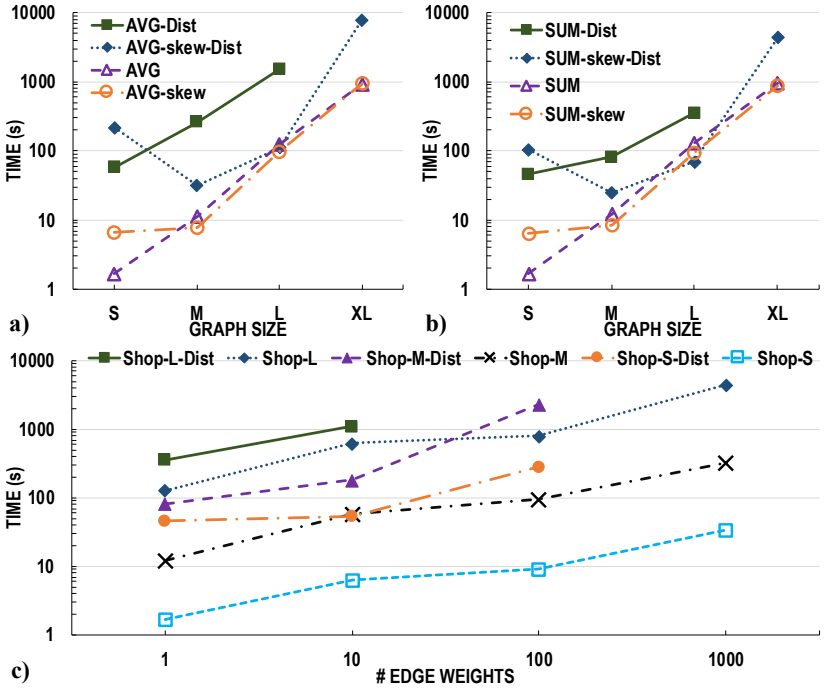

Fig. 14 Comparison between RESUM and RESUM *distributed*: running time varying the size of the graph, using *AVG* (a) and *SUM* (b), with the edge weights generated by a Beta distribution $(0.7, 5)$ and a skewed distribution; and running time in SHOP-S, SHOP-M, SHOP-L, and SHOP-XL, varying the number of edge weights, using *SUM* (c).

by the time required to compute an unnecessary large amounts of embeddings. As a consequence, an embedding-based distribution cannot be generally recommended for relevant pattern mining, although it provides higher load balance and less worker communication than the most popular alternative distributed graph processing systems [43].

Scalability We additionally compared RESUM and RESUM *distributed* on the four synthetic graphs using default relevance threshold and frequency threshold 90, 900, 9000, 90000, respectively. At these frequencies, the graphs contain roughly the same number of frequent patterns (47, 43, 44, and 45 respectively), hence allowing us to analyze how the increasing number of embeddings per pattern affects the performances of the two algorithms. For these experiments, we also sampled the edge weights from two different distributions: a Beta distribution with parameters $(0.7, 5)$ and *focus* 0.75, and a skewed distribution that simulates a user interested in a category of products available in the online shop, and thus assigns a large positive weights to the corresponding edges and zero to the others.

In Figure 14(a, b) we can see that RESUM *distributed* works better when the weights are sparse, with *AVG* achieving performances comparable to *SUM*, similarly to what observed for RESUM. In particular, the algorithm succeeded in extracting the relevant patterns from the largest graphs with skewed weights, but was able to finish the computation up to the graph of size L when the weights were drawn from the Beta distribution. In the case of sparse weights, the number of embeddings satisfying the condition posed by the scoring function is lower, and thus the workers must process and send a lower number of embeddings through the network. The communication between the machines is thus faster. In contrast, the running time of RESUM is not significantly affected by the weight distribution, as it does not generate and keep in memory all the embeddings in the graph, as opposed to the distributed framework. In addition, we note that when the graph is small, the overhead of setting up the distributed environment outweighs the cost of mining the relevant patterns, and thus RESUM *distributed* takes more time than RESUM to complete the computation. On the other hand, when the size of graph is large, so is the number of embeddings in the graph, and thus the algorithm can suffer from delays in machine communication and increasing cost of embedding generation.

Finally, Figure 14 (c) reports the running times of RESUM and RESUM *distributed* in the four synthetic graphs, varying the number of weights per edge. The performance is consistent with the single-weight setting, thus demonstrating the superiority of the centralized algorithm and the complexity of developing practical and scalable distributed solutions to graph mining problems.

9 Conclusions

In this paper we consider the problem of mining relevant patterns in weighted graphs. As opposed to the previous graph pattern mining approaches, which are solely based on the frequency of the patterns, our solution assesses the importance of a pattern also in terms of the weights on the edges of its appearances. To solve this problem efficiently we propose a novel family of scoring functions, called MNI-compatible, which allow effective retrieval of relevant patterns. We study four different scoring functions from this family. Those functions balance between frequency and weights, while retaining the advantages offered by the apriori property, which is a powerful mean to an effective and early pruning of the search space. As a natural extension, we considered the complementary problem of mining patterns in graphs with multi-

ple weights associated to the edges. We devised exact and approximate solutions and proved the effectiveness and efficiency of the algorithms on real datasets. Finally, we compared the performance of the centralized and the distributed version of our approach using graphs with different sizes and characteristics. We describe cases where the distribution may be beneficial, and also show cases where a centralized algorithm is still to be preferred, proving that distributing graph pattern mining algorithms efficiently and effectively is not a straightforward task. As a future work, we plan to study the theoretical bounds on the clustering quality, and automatic approaches for parameter selection.

References

1. E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 716–727, 2016.
2. C. C. Aggarwal. *Recommender Systems*. Springer, 2016.
3. G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of rdf data management systems. In *International Semantic Web Conference*, pages 197–212. Springer, 2014.
4. N. Babu and A. John. A distributed approach to weighted frequent subgraph mining. In *International Conference on Emerging Technological Trends*, pages 1–7, 2016.
5. G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2017.
6. D. Bandari, S. Xiang, and J. Leskovec. Categorizing user sessions at pinterest. *arXiv preprint arXiv:1703.09662*, 2017.
7. P. Bogdanov, M. Mongiovì, and A. K. Singh. Mining heavy subgraphs in time-evolving networks. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 81–90. IEEE, 2011.
8. B. Bringmann and S. Nijssen. What is frequent in a single graph? In *PAKDD*, pages 858–863, 2008.
9. Y. Chen, X. Zhao, X. Lin, and Y. Wang. Towards frequent subgraph mining on single large uncertain graphs. In *2015 IEEE International Conference on Data Mining*, pages 41–50, 2015.
10. J. C. Costello, M. M. Dalkilic, S. M. Beason, J. R. Gehlhausen, R. Patwardhan, S. Middha, B. D. Eads, and J. R. Andrews. Gene networks in drosophila melanogaster: integrating experimental data to predict gene function. *Genome biology*, 10(9):R97, 2009.
11. L. De Raedt and A. Zimmermann. Constraint-based pattern set mining. In *SDM*, pages 237–248, 2007.
12. M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7):517–528, 2014.
13. M. Fiedler and C. Borgelt. Subgraph support in a single large graph. In *ICDM Workshops*, pages 399–404, 2007.
14. R. Geng, X. Dong, P. Zhang, and W. Xu. Wtmaxminer: Efficient mining of maximal frequent patterns based on weighted directed graph traversals. In *CCIS*, pages 1081–1086, 2008.
15. M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, volume 30, pages 58–66, 2001.
16. R. He and J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*, pages 507–517, 2016.
17. L. B. Holder, D. J. Cook, S. Djoko, et al. Substucture discovery in the subdue system. In *KDD Workshop*, pages 169–180, 1994.
18. J. Huan, D. Bandyopadhyay, W. Wang, J. Snoeyink, J. Prins, and A. Tropsha. Comparing graph representations of protein structure for mining family-specific residue-based packing motifs. *J. Comput Biol.*, 12(6):657–671, 2005.
19. J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *SIGKDD*, pages 581–586, 2004.
20. S. Jamil, A. Khan, Z. Halim, and A. R. Baig. Weighted muse for frequent sub-graph pattern finding in uncertain dblp data. In *2011 International Conference on Internet Technology and Applications*, pages 1–6, 2011.

21. C. Jiang, F. Coenen, and M. Zito. Frequent sub-graph mining on edge weighted graphs. In *DAWAK*, pages 77–88. 2010.
22. H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, pages 566–575, 2007.
23. W. Jiang, J. Vaidya, Z. Balaporia, C. Clifton, and B. Banich. Knowledge discovery from transportation network data. In *ICDE*, pages 1061–1072, 2005.
24. X. Jin, C. Wang, J. Luo, X. Yu, and J. Han. Likeminer: a system for mining the power of 'like'in social media networks. In *KDD*, pages 753–756, 2011.
25. M. Kanehisa and S. Goto. Kegg: kyoto encyclopedia of genes and genomes. *Nucleic acids research*, 28(1):27–30, 2000.
26. M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.
27. M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *DMKD*, 11(3):243–271, 2005.
28. J. Li, Z. Zou, and H. Gao. Mining frequent subgraphs over uncertain graph databases under probabilistic semantics. *VLDBJ*, 21(6):753–777, 2012.
29. S. P. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theory*, 28(2):129–137, 1982.
30. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
31. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.
32. F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *HotOS*, volume 15, pages 14–14. Citeseer, 2015.
33. D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: a new way of searching. *VLDB J.*, pages 1–25, 2016.
34. M. E. Newman. Analysis of weighted networks. *Physical review E*, 70(5), 2004.
35. C. C. Noble and D. J. Cook. Graph-based anomaly detection. In *SIGKDD*, pages 631–636, 2003.
36. O. Papapetrou, E. Ioannou, and D. Skoutas. Efficient discovery of frequent subgraph patterns in uncertain graph databases. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 355–366, 2011.
37. J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. Mining access patterns efficiently from web logs. In *PAKDD*, pages 396–407, 2000.
38. G. Preti, M. Lissandrini, D. Mottin, and Y. Velegrakis. Beyond frequencies: Graph pattern mining in multi-weighted graphs. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT*, 2018.
39. M. J. Shaw, C. Subramaniam, G. W. Tan, and M. E. Welge. Knowledge management and data mining for marketing. *Decision support systems*, 31:127–137, 2001.
40. A. Silva, W. Meira Jr, and M. J. Zaki. Mining attribute-structure correlated patterns in large attributed graphs. *PVLDB*, 5(5):466–477, 2012.
41. Q. Song, Y. Wu, and X. L. Dong. Mining Summaries for Knowledge Graph Search. In *ICDM*, pages 1215–1220, 2016.
42. M. Steinbach, L. Ertöz, and V. Kumar. The challenges of clustering high dimensional data. In *New Directions in Statistical Physics*, pages 273–309. 2004.
43. C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.
44. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.
45. N. Vanetik, S. E. Shimony, and E. Gudes. Support measures for graph data. *Data Min. Knowl. Discov.*, 13(2):243–260, 2006.
46. H. Wang and C. C. Aggarwal. A survey of algorithms for keyword search on graph data. In *Managing and Mining Graph Data*, pages 249–273. 2010.
47. D. Wu, J. Ren, and L. Sheng. Uncertain maximal frequent subgraph mining algorithm based on adjacency matrix and weight. *International Journal of Machine Learning and Cybernetics*, pages 1–11, 2017.
48. X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.
49. X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.
50. J. Yang, W. Su, S. Li, and M. M. Dalkilic. Wigm: Discovery of subgraph patterns in a large weighted graph. In *SDM*, pages 1083–1094, 2012.

51. M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.
52. Z. Zou, J. Li, H. Gao, and S. Zhang. Mining frequent subgraph patterns from uncertain graph data. *IEEE Transactions on Knowledge and Data Engineering*, 22(9):1203–1218, 2010.