TAGs: Scalable Threshold-Based Algorithms for Proximity Computation in Graphs

A. Lyritsis
Department of Informatics
Aristotle University
54124 Thessaloniki, Greece
lyritsis@csd.auth.gr

A. N. Papadopoulos
Department of Informatics
Aristotle University
54124 Thessaloniki, Greece
papadopo@csd.auth.gr

Y. Manolopoulos
Department of Informatics
Aristotle University
54124 Thessaloniki, Greece
manolopo@csd.auth.gr

ABSTRACT

A fundamental and very useful operation in graphs is the computation of the proximity between nodes, i.e., the degree of dissimilarity (or similarity) between two nodes v and u. This is an important tool both in graph databases and graph mining applications, because it provides the base to support more complex tasks such as graph partitioning, clustering, classification, to name a few. All methods proposed in the literature assume that proximity is computed on a single graph by using a single distance measure. In addition, most of them focus on the proximity between node pairs. In this work, we present for the first time, scalable algorithms that: (i) they support proximity computation in multiple graph instances, (ii) they enable the utilization of several distance measures, (iii) they support proximity queries around a source node without limiting to node pairs and (iv) they support extensions for metric-based and skyline query processing. The main result of our work is the design of Threshold Algorithms for Graphs (denoted as TAGs), which are studied and evaluated experimentally by using real-life as well as synthetic graphs, based on both the $G(n, p)$ Erdõs-Rényi model and power law degree distributions.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems; H.2.8 [**Database Management**]: Database Applications—*Data Mining*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph Algorithms*

General Terms

Algorithms

Keywords

graph databases, proximity query processing

1. INTRODUCTION

Graph mining is an important research direction [1, 3, 6] that recently has attracted significant interest. The main reason for this trend, is that real-life networks have grown in size and therefore scalable mining tasks are required for network analysis purposes. Node proximity [5] is a fundamental operation that is required by complex mining tasks, such as partitioning, clustering and outlier detection. In its simplest form, node proximity is requested for a particular pair of entities (nodes) s and t, which generally do not have a direct connection between them. The way node proximity is computed depends on application requirements and also on the data representation and data semantics.

As an example, consider a co-authorship network, where nodes represent authors and a connection between two authors declares that they have a paper in common. In this application, one may be interested in the similarity between authors s and t, who may not have a common paper. The larger the proximity between s and t in the co-authorship network, the larger the similarity of their research interests. As a second example, assume an information retrieval application where nodes represent named entities (e.g., place names) extracted from a text collection, and two entities are directly connected if they appear in the same document, in the same section or in the same paragraph (according to application needs). The network-based proximity between two entities s and t quantifies the semantic similarity between these entities.

Different techniques have been proposed to determine proximity in networks [5, 21]. However, previously proposed methods are characterized by three important limitations. First, proximity computation is supported for a specific vertex pair s, t. In many applications, like graph-based information retrieval for example [25], there is a need to support additional operations that are considered important: (i) given a query entity s determine all entities with distance at most r from s, and (ii) given a query entity s determine the k most proximal entities to s. A second limitation of proposed methods is that they are based on a single distance measure to compute proximity (e.g., random walk with restart, shortest path, maximum flow). Finally, a third shortcoming is that only a single graph has been used to determine proximity. However, entities may be associated in many different ways [26] resulting in many graph instances. In such a case, multiple graphs are required for proximity computation, and therefore distance measures must be combined in a convenient way to obtain a semantically meaningful result. For example, researchers may form a graph G with paper

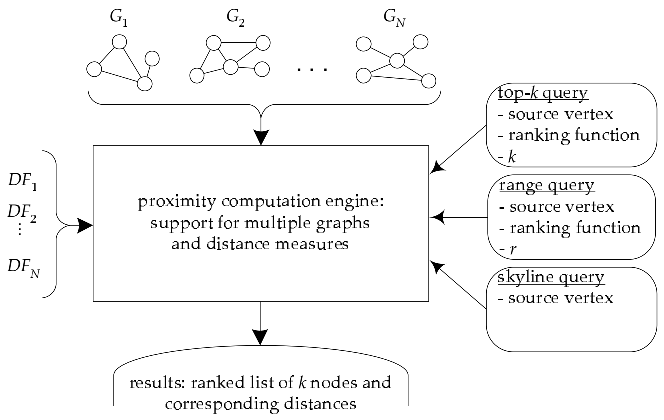


Figure 1: Query processing with multiple graphs and multiple distance measures.

co-authorship information and also may formulate another graph H capturing their cooperation in project proposals. Both graphs should be utilized to compute the similarity between two researchers regarding their research interests in general.

Figure 1 depicts a bird's eye view of a system offering query processing services in multiple graphs with multiple distance measures. The illustrated query types are just a few examples of selection-based queries that may be supported, since more complex queries may be defined on the input graphs (e.g., joins). Among these query types we focus on top-k queries, whereas skyline queries are briefly discussed in Section 5.

Figure 2 depicts two graphs G and H. For G we assume that the distance between two entities is given by the *shortest path* distance, whereas for H proximity is determined according to the *maximum flow* that can be pushed between the nodes. We are interested in determining the $k = 3$ nodes that are closer to a source node s, by considering both graphs and both distance measures. To provide a global distance measure we must combine the two measures in a meaningful way. Note that using shortest paths the smaller the distance the larger the proximity between the nodes. On the other hand, the smaller the maximum flow the smaller the proximity. In fact, we are searching for nodes such that their shortest path distance to s is minimized whereas the corresponding maximum flow is maximized. One way to provide the answer set is to use a ranking function that combines the two measures. In this example, we choose to use the function $F(s,t) = sp(s,t,G) \cdot (n_h - mf(s,t,H))$, where $sp(G,s,t)$ denotes the shortest path distance between s and t in graph G, $mf(s,t,H)$ denotes the maximum flow between s and t in H and n_h is the number of vertices of H. Note that we subtract the maximum flow value from n_h, in order to convert it to a *distance measure* since the maximum flow is by definition a *similarity measure*. It is evident, that the derived ranking function assumes values in the interval $[0, (n-1)^2]$. Based on Figure 2, if v_1 is selected as the source node, the score of each node v_1, ..., v_8 is 0, 6, 6, 10, 6, 12, 12, 18 respectively. Therefore, the $k = 3$ most proximal nodes to v_1 according to the ranking function $F()$ are v_2, v_3 and v_5 with score 6.

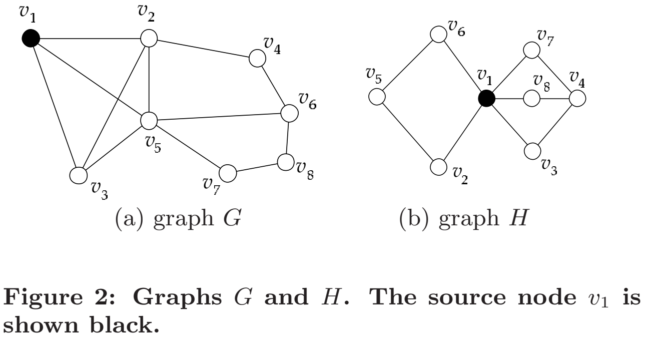


Figure 2: Graphs G and H. The source node v_1 is shown black.

The focus of this paper is to study efficient techniques for fast proximity computation, when multiple, and generally different, proximity measures should be combined in a synergetic manner towards processing queries in multiple graphs. To the best of our knowledge, this is the first work investigating the concept of proximity computation in multiple graph instances. In summary, our contributions have as follows:

- We focus on proximity queries around a source node, whereas the majority of previous work in the area has focused on proximity computation between node pairs.
- Proximity computation is based on multiple graph instances that may be available. Each of these graphs captures different relationships among the entities of interest and therefore all of them must be used to produce a concrete answer.
- Since each graph instance may require different proximity measures, we study the use of two popular measures that have been used in the literature as standalone measures for proximity computation, namely shortest path and maximum flow. We show that the synergy of these measures provides a meaningful way to calculate proximity. However, any other measure may be used as long as some fundamental properties hold, such as monotonicity.
- The first approach is based on the fact that no preprocessing is allowed to the graphs, and may be used in mining highly dynamic networks. If preprocessing is allowed, we provide faster algorithms that take advantage of the preprocessing performed. To avoid excessive storage consumption, we allow only linear additional space (with respect to the number of nodes) to store precomputed results.
- Performance evaluation results are offered which are based on real-life as well as synthetic networks. The results show the efficiency and scalability of the proposed algorithms.

The rest of the article is organized as follows. Section 2 presents research contributions related to our work and also briefly summarizes fundamental concepts. The proposed methodology is analyzed in Section 3, where four threshold-based algorithms ate studied in detail. Performance evaluation results based on real-life as well as synthetic networks are offered in Section 4. A discussion of additional important concepts and extensions is offered in Section 5. More specifically, metric-based and skyline-based proximity queries are discussed briefly. Finally, Section 6 concludes the work and briefly presents future research in the area.

2. BACKGROUND

The role of this section is twofold: (i) to briefly discuss related research in the area and (ii) to present some fundamental concepts regarding the algorithms studied.

2.1 Related Research

The problem of proximity computation in networks has attracted significant interest due to its importance in more complex data mining tasks. An interesting proposal for proximity computation is offered in [20]. However, this method cannot be used to answer neighborhood queries since it only supports pair-wise proximity computation. The nice property of this algorithm is that it does not require the investigation of the whole graph. However, in order to answer neighborhood queries around a source vertex s, all node proximities from s must be computed, leading to increased computational cost.

Random walk with restart (RWR) has been used in [27] to determine proximity. In fact, this method supports neighborhood formation [25] around a source vertex since it computes proximities in one-to-all fashion. To determine the proximity between s and t, a random walk is performed, and the proximity between the nodes is determined by using the steady-state probability. This method can be easily incorporated in our proposal as a potential distance measure. A limitation of RWR is that since the triangle inequality does not hold, we cannot use metric-based access methods to organize the nodes.

The main limitation with all proposed approaches is that a single graph is assumed and only one distance measure is applied to determine proximity. To support multiple graphs and multiple distance measures we need efficient tools to combine the graphs and provide efficient means of graph search to get the final answer. Our work is inspired by Fagin's significant work [8] for threshold-based top-k computation in information retrieval. According to Fagin's idea, if multiple streams of object attributes are available sorted in non-increasing order of significance, a global ranking may be achieved by providing a monotone scoring function. In our case, each node v may be represented as a tuple where the i-th attribute corresponds to the distance between v and the source s in the i-th graph. This observation allows the use of threshold algorithms in our case.

Although many distance measures could have been used, we focus on shortest paths and maximum flows, because of their simplicity, their extensive use in diverse applications and the nice properties that arise from their combination. As we are going to demonstrate in the sequel, all maximum flow values between nodes can be recorded in a tree-like structure, the Gomory-Hu tree [15], which can be constructed by executing $n-1$ maximum flow operations. This structure allows for efficient sorted access around a source node s, and saves computational cost since maxflow operations are expensive. The Gomory-Hu tree has been used also in [12] for community discovery in networks and in [19] for partitioning web graphs. In general, it may be used when efficient maxflow computations are required.

Another work that motivated us to conduct this research is [26], where the authors study graph clustering by taking into consideration multiple graph instances. In this paper, we describe threshold-based algorithms to allow for proximity computation using multiple graphs and (where applicable) multiple distance measures.

2.2 Fundamental Concepts

For the rest of the paper we assume that each network is modeled by an undirected graph G, where V_G is the set of vertices (nodes) and E_G the set of edges (direct connections). Moreover, without loss of generality we assume that between two vertices v_i and v_j there could be at most one connection, represented by the edge $e_{i,j}$, which is possibly weighted. Extensions for multigraphs are straight-forward. The support of directed graphs is also possible, provided that the absence of the symmetry property in node proximity does not pose any problems; otherwise, adaptations are required. Table 1 shows the most frequently used symbols.

Symbol	Interpretation
G	an undirected (possibly weighted) graph
V_G, E_G	nodes and edges of G
n_g,m_g	number of nodes and edges of G
v_i	the i-th node of a graph
$e_{i,j}$	edge joining nodes v_i and v_j
k	number of requested proximal nodes
$w(v_i, v_j)$	weight of the edge between v_i and v_j
$sp(v_i, v_j, G)$	shortest distance between v_i, v_j in G
$mf(v_i, v_j, G)$	maxflow between v_i, v_j in G
$T(G)$	the Gomory-Hu tree of graph G

Table 1: Frequently used symbols.

Without loss of generality, we assume that the relationships of our entities (represented as graph vertices) are captured by two graphs G and H (generalizations for any number of graphs are straight-forward) where multiple vertex proximity measures are available. In addition, we assume that both graphs contain the same set of nodes. In a different case, several approaches may be followed. For example, if there is a node v contained in G but not in H then its distance from the source node may be set to infinite to exclude it from the result set. It is evident that the source node s must be present in both graphs.

We focus on two very popular distance measures that have been used in the literature for proximity computation, namely the *shortest path* (applied on G) and the *maximum flow* (applied on H). The use of these two measures is quite intuitive. In an undirected and unweighted graph, the shortest path distance between nodes s and t captures how far or near t lies with respect to the source node s. In fact, the shortest path distance equals the minimum number of hops required to reach t starting at s. The smallest the shortest path the larger the similarity between s and t. According to Menger's theorem [23], the maximum flow between s and t equals the number of *edge disjoint* paths connecting s and t. This measure captures the number of different ways one can use to reach t from s (and vice versa). The more paths available the stronger the connection between s and t. The problem tackled in this work has as follows:

PROBLEM DEFINITION
Given two graphs G and H, where proximity in G is measured by using the shortest path and proximity in H is given by the maximum flow, determine an efficient method to compute the k vertices that are closer to a source vertex s, by taking into account both measures.

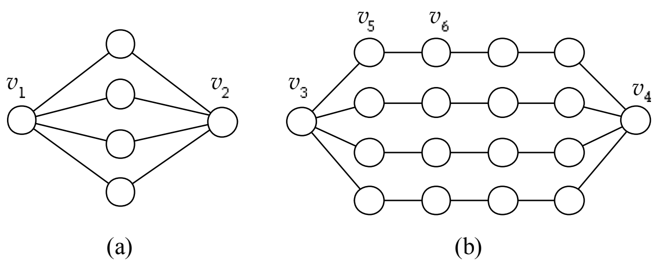


Figure 3: Vertices with different proximities.

A justification regarding the usefulness of a synergetic combination of shortest path and maximum flow is illustrated in Figure 3. Recall that the best proximity between two nodes is achieved when the shortest path distance is as minimum as possible and the maximum flow similarity is as maximum as possible. There are four possible cases: (i) small shortest path distance and large max flow similarity (e.g., nodes v_1, v_2), (ii) small shortest path distance and small max flow similarity (e.g., nodes v_5, v_6), (iii) large shortest path distance and large max flow similarity (e.g., nodes v_3, v_4) and (iv) large shortest path distance and small max flow similarity (e.g., nodes v_5, v_4). For example, the proximity between v_1 and v_2 should be larger than the proximity between v_3 and v_4, because although the maximum flow between the two pairs is 4, the shortest path distance between v_1 and v_2 is smaller than the one between v_3 and v_4. For the rest of the work we assume that the score of a node v is computed by taking the product $score(v) = sp(s,v,G) \cdot (n_h - mf(s,v,H))$, where $sp(s,v,G)$ is the shortest path distance between s and v in graph G, and $mf(s,v,H)$ is the maxflow between s and v in graph H. Although the two measures if they are used alone are not adequate [10] their synergy is very important. The product enforces that both factors should be minimized to get the best result [28]. Alternative ranking functions as well as different approaches are discussed in Section 5.

A simple technique that can be used to solve the problem is to compute the score for each node and then select the k nodes with the best scores. Although this algorithm is simple to implement, it suffers from performance degradation, especially in large graphs. For this reason, we exclude this naive solution from the subsequent study. In the sequel, we describe in detail the TAG (Threshold Algorithms in Graphs) family of algorithms, which use thresholding with the necessary early-termination conditions, to avoid unnecessary computational costs. The algorithms presented in the sequel, differ in the following aspects: (i) the degree of preprocessing of the input graphs and (ii) the degree of intermediate result materialization that may be used.

3. THRESHOLD ALGORITHMS

In this section, we develop our proposal for proximity computation using threshold-based algorithms. The fundamental differences among the techniques presented lie in the preprocessing requirements and the materialization of intermediate results.

3.1 TAG-I

In this section, we present the basic version of the threshold-based algorithm which is applied when no time-consuming preprocessing is allowed to the input graph[1]. Such a case is frequent in modern applications dealing with fresh data (e.g., time-evolving data), where processing must be applied without any prior knowledge about the input.

We define two operations on graphs G and H: (i) a *graph expansion* operation involves the computation of the next most proximal node of the source node s of graph G with respect to the shortest path distance, (ii) a *maximum flow* operation involves the computation of the maximum flow between two nodes s and t of graph H. These operations are related to the *sorted access* and *random access* respectively proposed in [8]. An expansion is executed on G by requesting the next proximal node u with respect to the source s. Then, a lower bound is computed for the best possible score that u may have. If this bound is larger than the k-th score determined so far, then the algorithm terminates. Otherwise, a maxflow operation is executed on H for nodes s and u and if the determined score is better than the current k-th best score, a substitution is performed.

The outline of TAG-I algorithm is shown in Figure 4. The main loop is executed until either all nodes have been examined (line 4) or the termination condition is met (line 10). The nodes of G are organized in a minheap data structure allowing the progressive scanning of proximal nodes with respect to the source s. The set A_k contains pairs of the form $< u, score(u) >$, where u is a node and $score(u)$ the current score of u. Therefore, if the best possible score of a node is larger than the k-th score in A_k, the algorithm may terminate because it is impossible to discover a node with a better score. The key issue in this threshold is that the maximum

Algorithm TAG-I (s, k, G, H)
Input. G, H: input graphs, s: source, k: num of results
Output. A_k: k vertices with the best scores

```
1:  init priority queue $minheap$; init $A_k \leftarrow \emptyset$
2:  $done \leftarrow$ false;
3:  while (not $done$)
4:      if ($minheap$.isEmpty()) then
5:          $done \leftarrow$ true; exit loop;
6:      $entry \leftarrow minheap$.removeTop();
7:      $u \leftarrow entry$.vertex; $sp(u) = entry$.distance;
8:      if ($|A_k| = k$) then
9:          $bestScore \leftarrow sp(u) \cdot (n_h$ - degree$(s,H))$;
10:         if ($bestScore >$ largest score in $A_k$) then
11:             $done \leftarrow$ true; exit loop;
12:         else
13:             remove the pair with the worst score from $A_k$;
14:     if ($u$ != $s$) then
15:         $score(u) \leftarrow sp(u) \cdot (n_h - maxflow(s,v,H))$;
16:         insert the pair $u$, $score(u)$ into $A_k$;
17:     for (each neighbor $v$ of node $u$) do
18:         if ($v$ has not been deheaped from $minheap$) then
19:             $alt \leftarrow sp(u)$ + weight($u$,$v$);
20:             if ($alt < sp(v)$) then
21:                 $sp(v) \leftarrow alt$;
22:                 update $minheap$;
23: return $A_k$;
```

Figure 4: Outline of TAG-I algorithm.

[1]In many cases, preprocessing requiring linear time and linear additional space is acceptable.

flow between two nodes s and t is bounded by the minimum degree among the two nodes. This is easily obtained taking into account that the maximum value of a mincut in a graph is bounded by the minimum degree of the graph [30].

LEMMA 3.1. *Let s_k be the k-th best score determined so far, u the node at the top of the minheap and $bs(u)$ the best possible score that node u may achieve. If $bs(u) > s_k$ then the algorithm may terminate since it is impossible to improve further the result set A_k.*

PROOF. By the definition of the expansion operation, it is guaranteed that nodes in G are discovered in non-decreasing distance from the source node s. This is because the main loop of TAG-I is similar to Dijkstra's algorithm. Therefore, every time we get the next node from minheap, we are sure that its shortest path distance from s will be larger or equal to that of the previously discovered node (see [7] for a thorough study of this issue). According to the way the best score of u is computed, it holds that $bs_u = sp(u) \cdot (n - mindegree(s))$. The only factor that depends on u is the shortest path distance $sp(u)$. The second factor $(n - mindegree(s))$ depends only on the number of nodes and the degree of the source node. Therefore, if we discover a node u such that the value $bs(u)$ is larger than the k-th score contained in A_k, we are sure that u as well as all nodes discovered after u cannot contribute to the final result. The opposite would suggest that if node v is discovered after u then $bs(v) < bs(u)$, meaning that $sp(v) < sp(u)$ which is a contradiction. □

To illustrate the way TAG-I works an example is given based on the graphs shown in Figure 2. We run a top-3 proximity query starting at node v_1. The first deheaped node is v_1 and therefore its neighbors v_2, v_3 and v_5 are checked and the appropriate decrease-key operations are performed. The next deheaped node is v_2. It holds that $sp(v_2) = 1$ and since $|A_k| < 3$ a maxflow operation is performed. The score of v_2 is $score(v_2) = sp(v_2) \cdot (n - mf(v_1, v_2)) = 1 \cdot 6 = 6$. In the same way, the scores of v_3 and v_5 are also set to 6. The next deheaped node is v_4. Since $|A_k| = 3$ we check the best possible score of v_4, which is $bs(v_4) = 2 \cdot (8 - 5) = 6$. This means that a maxflow computation will be performed for v_4 as well as for v_6 and v_7. However, when v_8 is deheaped, its best score is set to $bs(v_8) = 3 \cdot 3 = 9$. Since $bs(v_8) > 6$ this means that the termination condition is satisfied and the algorithm terminates since no further improvement on A_k may be achieved. The final answer comprises the pairs $< v_2, 6 >$, $< v_3, 6 >$ and $< v_5, 6 >$.

LEMMA 3.2. *Let $\alpha(s)$ denote the number of nodes of G reached through expansion operations starting at source node s (evidently $\alpha(s) \geqslant k$). Then, the worst case complexity of* TAG-I *algorithm is:*

$$O\left(m_g \log n_g + \alpha(s)(\log n_g + m_h n_h \log(n_h^2/m_h))\right)$$

PROOF. As in the case of Dijkstra's algorithm, at most m_g decrease-key operations are required, due to the loop of Figure 4. By assuming an ordinary binary heap, this cost is $O(m_g \cdot \log n_g)$. In addition, the complexity of a maxflow computation by using the push-relabel algorithm of Goldberg and Tarjan [13] which uses dynamic trees, is $O(n_h \cdot m_h \log(n_h^2/m_h))$. A maxflow operation is required for every deheaped node, until the termination condition is met. Since the number of deheaped nodes is $\alpha(s)$ the result follows. □

The previous lemma suggests that the most important factor affecting the performance of TAG-I algorithm is the number of maxflow computations involved, which depends on the value of $\alpha(s)$. In addition, every maxflow computation on graph H requires $O(n_h \cdot m_h \log(n_h^2/m_h))$ cost which is significant. In the sequel, we investigate techniques that lead to more efficient processing, by introducing some form of preprocessing on the input graphs.

3.2 TAG-II

Recall that algorithm TAG-I does not use any kind of preprocessing and therefore, it can be applied immediately to the input graphs. However, in many cases we are allowed to apply preprocessing towards speeding up subsequent proximity computations. Based on this fact, we perform preprocessing to the input graph H where maxflow computations are performed. Our goal is to provide a boost on the performance of maxflow computation by using only linear additional space, offering at least linear time on the application of each maxflow computation. The resulting algorithm is termed TAG-II, and it is based on the concept of *flow-equivalent* trees.

Evidently, a straight-forward way to speed-up maxflow operations, which are invoked in line 15 of Figure 4, is to precompute the maxflow values for each pair of nodes and store them in a matrix. Although this offers $O(1)$ time for maxflow computations, it requires $O(n_h^2)$ space, which is not acceptable for large graphs. Note that this matrix is dense (all cells are full) and therefore, techniques used for sparse matrices cannot be applied. According to [15], for any graph H one can build a *flow-equivalent* tree $T(H)$, such that the maxflow between two nodes in H equals their maxflow in $T(H)$. Such a tree is termed Gomory-Hu tree or mincut tree. Essentially, the existence of a flow-equivalent tree suggests that for a graph H with n_h nodes there are at most $n_h - 1$ different maxflow values between node pairs. This is a direct consequence of the fact that the Gomory-Hu tree contains exactly $n_h - 1$ edges, since it is connected (assuming that the input graph is also connected).

Using the Gomory-Hu for maxflow computations is quite easy. The maxflow between two nodes s and t equals the minimum weight contained in the unique path connecting s and t. More formally, if $p(s,t)$ denotes the path joining s and t, e is an edge and $w(e)$ the edge weight, then:

$$mf(s,t) = \min\{w(e), e \in p(s,t)\}$$

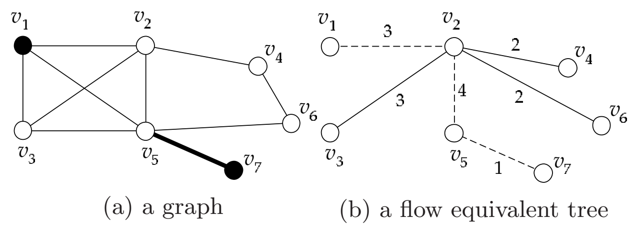
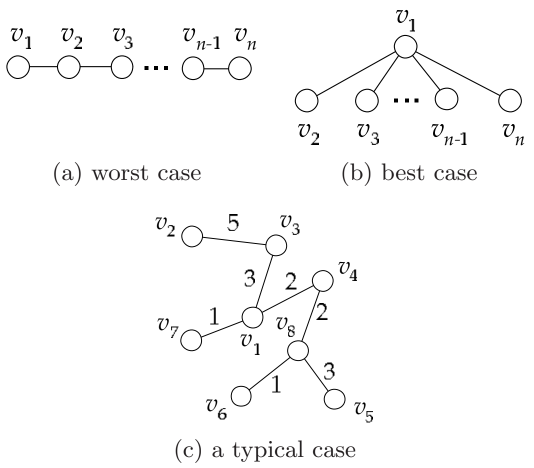


Figure 5: Gomory-Hu tree example.



v_1 v_2 v_3 $v_{n\text{-}1}$ v_n
(a) worst case
v_1 v_2 v_3 $v_{n\text{-}1}$ v_n
(b) best case
v_2 5 v_3 3 2 v_4 v_7 1 v_1 v_8 2 3 1 v_6 v_5
(c) a typical case

Figure 6: Worst, best and typical tree formation.

Figure 5 depicts an example of a graph and its associated flow-equivalent Gomory-Hu tree. According to the properties of Gomory-Hu trees, the maxflow between nodes v_1 and v_7 is 1, since it is the minimum edge weight in the path from v_1 to v_7 (shown with dashed lines). This may be easily validated by inspecting the graph in Figure 5(a) and verifying that indeed the maxflow is 1, since if the bold line is deleted then v_1 and v_7 become disconnected.

Since there is a unique path joining any pair of nodes s, t in a Gomory-Hu tree, maxflow computations may be supported in linear (worst case) time with respect to the number of nodes if no specialized data structures are being used and no preprocessing has been applied to the Gomory-Hu tree. In such a case, the performance of the maxflow computation is directly dependent on the structure of the tree. The worst case for the maxflow computation (which is $O(n)$) happens when the tree is degenerated to a single path as shown in Figure 6(a). On the other hand, the best tree formation that guarantees a constant time maxflow computation is the star formulation, shown in Figure 6(b). In this case, at most two tree edges need to be checked. In a typical case however, the shape of the tree is arbitrary, as shown in Figure 6(c).

If more sophisticated techniques based on the concept of *nearest common ancestor* [18] are used, then the complexity may drop to $O(1)$ for finding the maxflow value between any pair of nodes [4]. This very efficient method requires only linear preprocessing to the input tree and it is very simple to implement. More specifically, the problem of determining the smallest value along any path in the tree is reduced to the problem of determining the nearest common ancestor of the two nodes. Therefore, with an additional linear preprocessing applied to the Gomory-Hu tree the maxflow value between two nodes may be computed in constant time, which is excellent compared to the $O(n_h \cdot m_h \log(n_h^2/m_h))$ time required by the push-relabel algorithm.

We note that for a graph with n vertices and m edges the Gomory-Hu tree can be built in $O(n \cdot F(n,m))$ time [15], where $F(n,m)$ is the cost of executing a maxflow computation on a graph with n vertices and m edges. Therefore, if the push-relabel algorithm is being used, the tree construction complexity becomes $O(n^2 \cdot m \log(n^2/m)))$. As it has been shown in the literature [14] Gusfield's algorithm [16] is easier to implement than the original proposal by Gomory and Hu [15], although the two algorithms have the same complexity. Based on the previous discussion, and substituting the maxflow complexity with $O(1)$, the following result follows easily:

LEMMA 3.3. *If the Gomory-Hu tree $T(H)$ is used on graph H and the nearest common ancestor method is used to compute the maxflow between pairs of nodes, then the running time of algorithm* TAG-II *is given by:*

$$O(m_g \cdot \log n_g + \alpha(s) \log n_g)$$

3.3 TAG-III

Algorithms TAG-I and TAG-II perform sorted accesses on graph G by using network expansion operations and random accesses on graph H by computing the maxflow between nodes. Algorithm TAG-III, studied in the sequel, operates in the opposite way and it utilizes the Gomory-Hu tree of graph H for expansion operations (sorted accesses), whereas shortest path computations are performed on G upon request (random accesses).

TAG-III utilizes a heap data structure to prioritize nodes in $T(H)$. Prioritization is based on the edge weights associated with each node (emanating edges). In addition, nearest common ancestors and DFS are used as tools for efficient determination of the next node that must be checked. When the next candidate node is determined, the termination condition is checked first, and if necessary a shortest path computation is performed on G.

According to a well-known result regarding shortest paths, the worst case complexity of the one-to-all shortest path problem is the same to that of the one-to-one version [7]. This observation suggests that it is not efficient to execute from scratch a node-to-node shortest path operation for node pairs. Instead, a one-to-all shortest path operation is executed on G and the result is maintained, enabling the efficient computation of subsequent operations in constant time. This approach, however, has the additional constraint that the shortest path distances from s to all nodes in G need to be materialized for the duration of query execution. Although this is not a problem for small graphs, for large graphs there may be an issue, taking into account that many queries with potentially different source nodes may be running concurrently. An alternative is to use preprocessing on G in order to efficiently support shortest path queries. However, usually these techniques require superlinear additional space and significant preprocessing time. Since we are interested in solutions with linear additional space we do not elaborate more in these alternatives.

LEMMA 3.4. *If the Gomory-Hu tree is used for sorted accesses, and $\beta(s)$ is the number of nodes scanned through expansions in the tree, then the running time of* TAG-III *is given by:*

$$O\left(m_g \log n_g + n_g \log n_g + \beta(s) \log n_h\right)$$

PROOF. The result follows by considering that expansions in $T(H)$ are performed by means of a heap, and that a one-to-all shortest path is executed on G. $\square$

3.4 TAG-IV

The main property of the previously studied algorithms is that sorted accesses are performed to one of the two graphs, whereas random accesses are performed on H for maxflow computation (for TAG-I, TAG-II algorithms) and on G for shortest path computation (for TAG-III algorithm). A natural extension is to consider a more general scheme, where a synchronized traversal is used to control sorted and random accesses. Algorithm TAG-IV, described in the sequel, does exactly this.

TAG-IV works as the threshold algorithm (TA) proposed in [8]. More specifically, a buffer of k slots is maintained hosting the k best nodes determined so far. In each sorted access, two expansion operations are performed, one in G and one in $T(H)$. If different nodes are retrieved, then two random accesses are performed to fill-in the missing values, one in G to compute the shortest path distance and one in $T(H)$ to compute the maxflow distance. If the score of the newly discovered nodes is better than the k-th best score found in the buffer, then a substitution is performed, otherwise the buffer remains as it is. Then, a threshold th is computed by computing the scoring function for the values seen in the last sorted access. If all scores in the buffer are better than the current threshold, then the algorithm terminates since it is not possible to improve the results further. Otherwise, another sorted access is performed, a new value for the threshold is determined and the algorithm continuous as previously. The following lemma follows from the previous discussion, where $\gamma(s)$ is the number of sorted accesses performed by the algorithm.

LEMMA 3.5. *If $\gamma(s)$ is the number of sorted accesses performed, then the running time of TAG-IV is given by:*

$$O\left(m_g \log n_g + n_g \log n_g + \gamma(s) \log n_h\right)$$

TAG-IV is the most general algorithm since in addition to the flexibility regarding the synchronization of sorted and random accesses, it can be adapted to operate as any of the previously studied algorithms. The pseudocode of TAG-IV is given in Figure 7.

```
Algorithm TAG-IV ($s$, $k$, $G$, $H$)
Input. $G$, $H$: input graphs, $s$: source, $k$: num of results
Output. $A_k$: $k$ vertices with the best scores

1:  execute a one-to-all shortest path on $G$ starting at $s$;
2:  init priority queues; init $A_k \leftarrow \emptyset$
3:  $done \leftarrow$ false;
4:  while (not $done$)
5:      $v$ = get the next best node of $G$; /* sorted access */
6:      $u$ = get the next best node of $T(H)$; /* sorted access */
7:      compute $sp(u,s)$ and $mf(v,s)$; /* random accesses */
8:      compute $score(v)$ and $score(u)$;
9:      if ($score(v) <$ largest score in $A_k$) then update $A_k$;
10:     if ($score(u) <$ largest score in $A_k$) then update $A_k$;
11:     $threshold \leftarrow sp(v,s) \cdot mf(u,s)$;
12:     if (all values in $A_k < threshold$) then $done \leftarrow$ true;
13: return $A_k$;
```

Figure 7: Outline of TAG-IV algorithm.

4. PERFORMANCE EVALUATION

All algorithms are implemented in C++ and all experiments have been performed on an Intel Core Duo @ 2.2 GHz, with 2GB RAM running Windows Vista. The performance evaluation study is based on real-life graph data sets, as well as on synthetically generated ones. To simplify the experimentation process, in each experiment the same graph is being used as G and H (i.e., H is a copy of G). The synthetic graphs are basically used to have control upon the basic parameters of the graph, such as order, size and degree sequence, towards investigating performance by varying these parameters. Two types of synthetic graphs have been used:

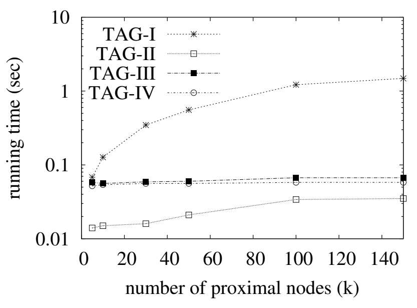
- Random graph (RA), which is generated based on the Erdős-Rényi model by fixing the number of vertices and then by deciding the existence of each edge with probability p.
- Power-law graph (PL), where node degrees follow a power law distribution [9]. More specifically, the probability that a node has degree deg is proportional to $deg^{-\epsilon}$, where ϵ is the power-law exponent, which is usually set between 1.5 and 3. We have used the GenGraph tool to generate power-law graphs, which is based on the work reported in [29].

We have used several real-life graphs from different application domains. The data sets are summarized in Table 2 and are briefly described as follows:

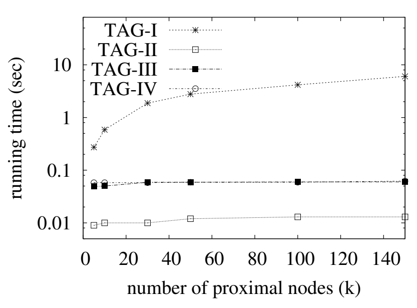
- The Gene Expression (GE) graph represents coexpression of human genes. Each vertex corresponds to a gene and an edge between two genes denotes a high coexpression level. In [31] the coexpression level between two genes v_i and v_j, denoted by $r_{i,j}$, is measured by using the minimum of the absolute values of leave-one-out Pearson correlation coefficient. Then, statistics are used to determine the p-value of a coexpression. An edge is formed between the genes if the p-value is less than a specified threshold (usually 0.01).
- The San Francisco (SF) graph represents the road network of San Francisco. Vertices correspond to road intersections and edges correspond to connections between intersections. Multiple edges between vertices have been removed.
- The Web Links (WL) graph, used in [2], containing web links in the domain *nd.edu* (University of Notre Dame). For the purposes of our work, the graph has been converted to an undirected one, by ignoring link directionality and loops that may exist.
- The Reuters News (RN) graph is based on all stories released during 66 consecutive days by the news agency Reuters concerning the attack of September 11. The vertices of RN are terms and there is an edge between two terms u and v iff they both appear in the same sentence. We have used the largest connected component of the network containing 13,308 nodes and 148,045 edges.
- The Enron Emails (EE) graph contains information regarding email messages. There is an edge between nodes v and u if there was at least a message exchange

Graph	#nodes	#edges	min degree	max degree	avg degree	availability
Reuters News (RN)	13,308	148,035	1	2265	22.24	http://vlado.fmf.uni-lj.si/pub/networks/data/
Microarray (MA)	8791	314,816	1	409	71.62	thanks to Xifeng Yan [31]
San Francisco (SF)	174,956	221,802	1	7	2.54	http://www.rtreeportal.org
Web Links (WL)	325,729	1,497,135	1	10,721	9.19	http://vlado.fmf.uni-lj.si/pub/networks/data/
Enron Emails (EE)	36,692	367,662	1	1385	11.57	http://snap.stanford.edu/data/index.html
Co-Authors (CA)	18,772	396,160	1	504	22.53	http://snap.stanford.edu/data/index.html

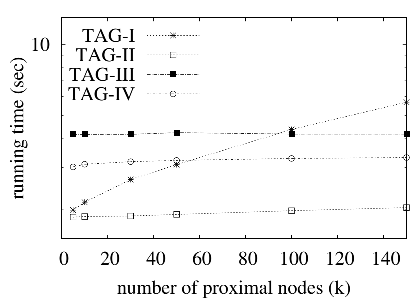
Table 2: Real-life graphs from different domains used in performance evaluation.



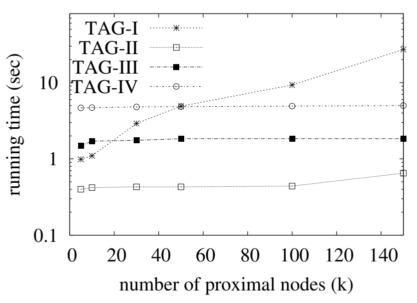
(a) Reuters News



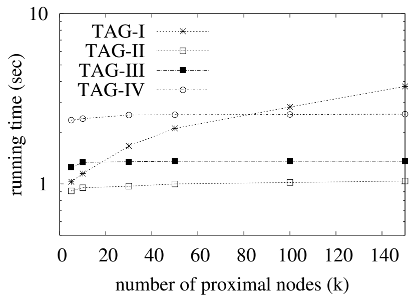
(b) Gene Expression



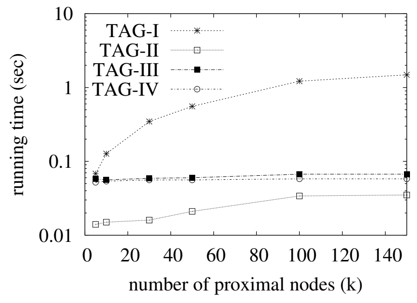
(c) San Francisco



(d) Web Links

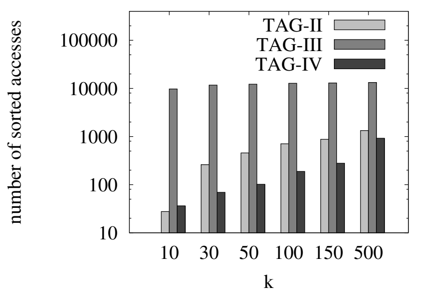


(e) Enron Emails

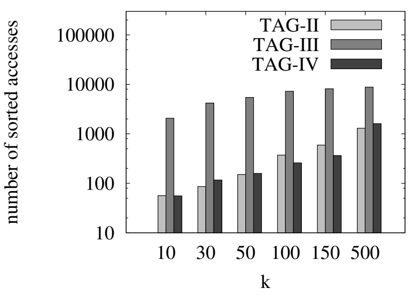


(f) Co-Authors

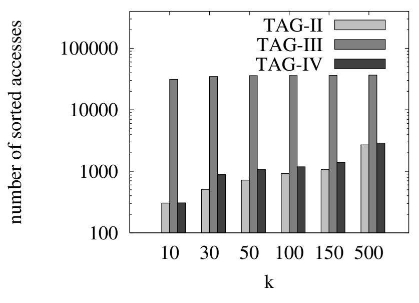
Figure 8: Running time (sec.) vs k for real-life graphs.



(a) Reuters News



(b) Gene Expression



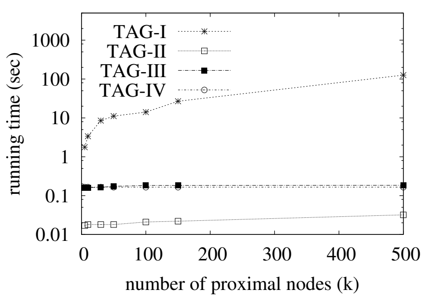
(c) Enron Emails

Figure 9: Number of sorted accesses vs k for real-life graphs.

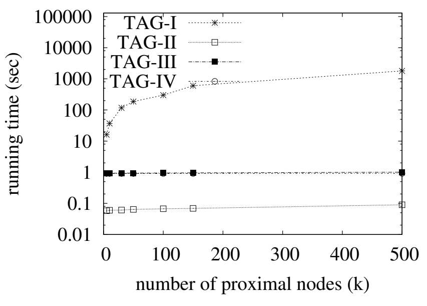
between them. This data set has been also used in [22] to explore graph properties over time.

- The Co-Authors (CA) graph has been extracted from arXiv and represents scientific collaborations between authors regarding papers submitted to Astro Physics area. There is an edge between authors u and v iff they both appear as co-authors in at least one paper. This data set has been also used in [22].

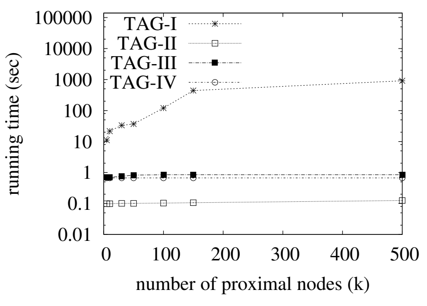
In all experiments, the proximity measure between nodes is expressed by the product of the shortest path and the maxflow distance. Extensions for other proximity measures are discussed briefly in the next Section 5. In the first series of experiments, we compare the performance of the algorithms by executing proximity queries on the real-life graphs, modifying the number of proximal nodes k. The results are illustrated in Figure 8, which shows the average running



(a) 20K nodes, 1.2M edges (p=0.01)

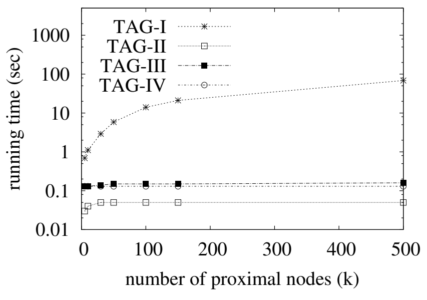


(b) 50K nodes, 6.3M edges (p=0.01)

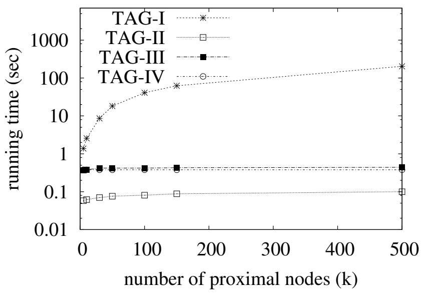


(c) 100K nodes, 2.5M edges (p=0.001)

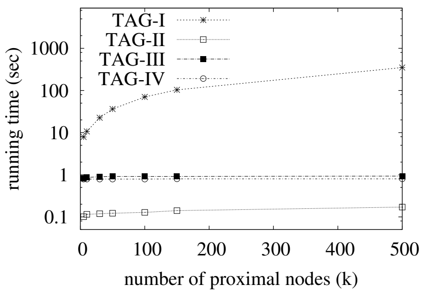
Figure 10: Running time (sec.) vs k for Erdõs-Rényi random graphs.



(a) 20K nodes, 1M edges, ϵ=2.3



(b) 50K nodes, 2.5M edges, ϵ=2.3



(c) 100K nodes, 5M edges, ϵ=2.3

Figure 11: Running time (sec.) vs k for power-law graphs.

time of 200 queries. For each run, a different source node is selected randomly. As expected, TAG-I is inferior to the other approaches, since it does not use any kind of preprocessing on the input graphs. More specifically, TAG-II shows the best overall performance regarding running time, being more than an order of magnitude more efficient than the baseline algorithm. Moreover, TAG-II is consistently more efficient than TAG-III and TAG-IV. The main reason for this behavior is that maxflow computations are performed very efficiently by using the Gomory-Hu tree, whereas sorted accesses performed for shortest path computations are fast. On the other hand, TAG-III performs sorted accesses on the Gomory-Hu tree, and executes a one-to-all shortest path computation which is responsible for the major part of the running time. Regarding the relative performance of TAG-III and TAG-IV, in some cases the two algorithms have similar running times (e.g., Figure 8(b)), in some cases TAG-III outperforms TAG-IV (e.g., Figure 8(d) and (e)) and finally in some other cases TAG-IV is faster than TAG-III (e.g., Figure 8(a, (c) and (f))). Another observation is that all algorithms, except TAG-I, show excellent scalability with respect to the number of the requested proximal nodes.

Regarding the previous results, Figure 9 shows the number of sorted accesses vs the parameter k for three real-life graphs, namely Reuters News, Gene Expression and Enron Emails. The number of sorted accesses performed is an important cost factor, since it gives an indication regarding the performance of the algorithms. Algorithm TAG-I is not shown because the number of sorted accesses is identical to that of TAG-II. As observed, TAG-III is outperformed by both TAG-II and TAG-IV, since it requires a significant number of sorted accesses before the thresholding criterion is satisfied. The explanation for this behavior is that sorted accesses are performed on the Gomory-Hu tree. Since for n nodes there are only n-1 different maxflow values between node pairs, meaning that maxflow values increase slowly, which delays the satisfaction of the threshold. On the other hand, TAG-II and TAG-IV manage to keep the number of sorted accesses low. In some cases TAG-IV outperforms TAG-II whereas in other cases TAG-II is better (see Figure 9).

In the sequel, we study the performance of the algorithms for the synthetic networks. Figure 10 shows the running time vs k for Erdõs-Rényi random graphs with various orders and sizes. Figure 10(a) contains the results for 20,000 nodes with p=0.01 and around 1M edges. In Figure 10(b) we give the results for 50,000 nodes with $p = 0.01$ containing approximately 6.5M edges, whereas Figure 10(c) shows the results for 100,000 nodes with $p = 0.001$ containing approximately 2.7M edges. Moreover, Figure 11 shows the results for graphs with a power-law degree distribution. Again, the generated graphs contain 20K, 50K and 100K nodes respectively whereas the number of edges are 1M, 2.5M and 5M. The power-law exponent was set to 2.3 in all cases. We observe the same performance in both Figures 10 and 11. More specifically, TAG-II remains the most efficient algorithm, whereas TAG-III and TAG-IV show a similar performance (with a small lead for TAG-IV). Again we observe the excellent scalability of the algorithms. In fact TAG-II, TAG-III and TAG-IV remain almost unaffected by increasing k up to 500. In all cases tested, TAG-II shows the best performance

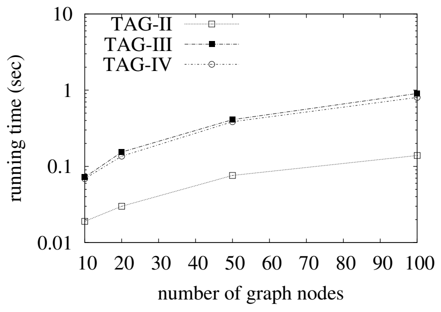


Figure 12: Running time (sec) vs number of nodes.

in terms of running time.

The last experiment shows the running time for answering proximity queries with $k = 100$, by increasing the number of graph nodes in the case of power-law degree distributions. The corresponding results are given in Figure 12, where the number of nodes varies from 10K to 100K, keeping the power-law exponent fixed at 2.3. As previously, the figure shows the average running time of 200 proximity queries. Notice that TAG-I is not shown, because of the significant difference in comparison to the other algorithms. TAG-II shows the best scalability in running time by increasing the number of nodes, being several times faster than TAG-III and TAG-IV.

In conclusion, TAG-II shows the best overall performance, both regarding the running time and the number of probes. The materialization performed by TAG-III and TAG-IV does no pay off, since it is expected that the network expansion will terminate without the requirement to check all nodes. On the other hand, algorithms TAG-III and TAG-IV require a one-to-all shortest path computation which is more expensive than the consecutive network expansions used by TAG-II. Note that these observations hold for the case where shortest paths and maximum flows are used as the available distance measures. More investigation is needed in order to test the performance of the methods when other distance measures are applied (e.g., random walks) and also when more than two graphs are given in the input.

	TAG-I	TAG-II	TAG-III	TAG-IV
sorted access sp	yes	yes	no	yes
sorted access mf	no	no	yes	yes
random access sp	no	no	yes	yes
random access mf	yes	yes	no	yes
graph preprocessing	no	yes	yes	yes
materialization	no	no	yes	yes
adaptivity	no	no	no	yes

Table 3: Summary of algorithm characteristics.

Among the studied algorithms, TAG-IV is the most general one, since it can operate as any of the other algorithms and moreover, may work as an adaptive technique, by changing its strategy according to the problem parameters. On the other hand, the rest of the algorithms do not show any flexibility since their query processing strategy is fixed: they perform sorted access on one graph and probe the other. TAG-IV is the only algorithm that may accept optimizations regarding the scheduling of probes, since it combines sorted and random accesses on the input graphs. This means that according to the processing cost of the distance measures and statistics collected at runtime, a different synchronization between sorted and random accesses may be proven more efficient. Table 3 shows the basic characteristics of the studied algorithms.

5. EXTENSIONS

In this section, we investigate briefly two query processing alternatives that are easily supported by the TAG family of algorithms. More specifically, we center our attention to (a) *metric-based* query processing, where the scoring function used in proximity computation respects the metric properties and (b) *skyline-based* query processing where there is no need for a scoring function.

5.1 Metric-based proximity computation

An important property of the proposed framework is that it provides flexibility regarding the scoring function that may be used to combine the different distance measures. The previous discussion was based on the fact that the scoring function is the product of the two distances, providing AND semantics. It is not hard to prove that generally the product of distances does not respect the metric properties, and more specifically, triangular inequality. However, since triangular inequality is the basic search mechanism in metric access methods, in many cases it is highly appreciated. To support this property, alternative distance functions are required.

When multiple distance measures are available, they may be combined by using a linear combination (i.e., weighted sum). In our case, the shortest path and maximum flow measures may be combined as follows:

$$F(s,t) = w_1 \cdot sp(s,t,G) + w_2 \cdot (n - mf(s,t,H)) \qquad (1)$$

where w_1 and w_2 are weights, that usually $0 \leqslant w_1, w_2 \leqslant 1$, and $w_1+w_2=1$. As the following lemma shows, this distance function respects all metric space properties.

LEMMA 5.1. *The distance function $F(s,t)$ as defined in Equation 1 respects the metric space properties (positivity, symmetry and triangular inequality).*

PROOF. It is not hard to prove that if two or more distance measures respect the metric space properties then so does any linear combination of them. Based on this result, and taking into consideration that the distance measure $sp(s,t,G)$ respects the metric space properties in undirected graphs, all we have to prove is that the measure $n - mf(s,t,H)$ satisfies positivity, symmetry and triangular inequality. Let G be an undirected graph and $T(G)$ a Gomory-Hu tree of G. Since G is flow-equivalent to $T(G)$, the maxflow in G may be computed by using the maxflow in $T(G)$. Thus, in the sequel we focus on $T(G)$. Let v, u be two nodes of $T(G)$. It is evident that if $v \neq u$ then the maxflow value is non-zero, thus positivity is satisfied. Moreover, since we deal with an undirected graph, the maxflow from v to u equals the maxflow from u to v, thus symmetry is also satisfied. It suffices to prove that triangular inequality also holds, i.e., $n - mf(v,u,G) \leqslant n - mf(v,x,G) + n - mf(x,u,G)$, $\forall v,u,x$. Let $p(v,u)$ denote the unique path joining v and u in $T(G)$. By the definition of $T(G)$, the maxflow between v and u is given by the edge with the minimum weight contained in $p(v,u)$. Therefore, since maxflow values are subtracted from n, the maxflow distance between v

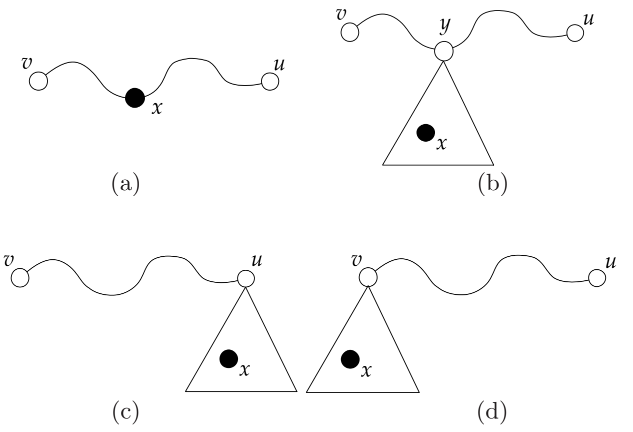


Figure 13: The four different cases for placing node x relative to nodes v and u in a Gomory-Hu tree.

and u is given by the edge with the maximum weight contained in $p(v, u)$. For convenience we set $MF(v, u, G) = n - mf(v, u, G)$. We will prove that for any node triplet v, u and x such that $v \neq u$, $x \neq v$ and $x \neq u$ it holds that:

$$MF(v, u) \leqslant MF(v, x) + MF(x, u) \tag{2}$$

Figure 13 depicts all possible cases regarding the location of node x in relation to v and u. We examine each case separately, showing that in any case triangular inequality does hold. Let w_{max} denote the maximum weight along the path $p(v, u)$.

Case 1. In this case, which is depicted in Figure 13(a), node x lies on the unique path $p(v, u)$ joining v and u. Without loss of generality, assume that the maximum weight appears in the path $p(v, x)$. Therefore, the inequality $MF(v, u, G) \leqslant MF(v, x, G) + MF(x, u, G)$ is satisfied since by substitution we get $w_{max} \leqslant w_{max} + C$, where C is the maximum weight along the path $p(x, u)$, which is always true since all weights are positive numbers.

Case 2. This case, shown in Figure 13(b), assumes that node x lies in a tree branch emanating from one of the nodes of $p(v, u)$ (node y in our case). Without loss of generality, we assume that the maximum weight between v and u appears in the path $p(v, y)$. Thus, the inequality $MF(v, u, G) \leqslant MF(v, x, G) + MF(x, u, G)$ becomes $w_{max} \leqslant MF(v, x, G) + MF(x, u, G)$. However, the value of $MF(v, x, G)$ is at least w_{max}, since the subpath $p(x, y)$ may contain an edge with a weight larger than w_{max}, and this means that again the inequality is satisfied.

Case 3. In this case, shown in Figure 13(c), node x lies in a subtree emanating from node u. Since the path $p(x, v)$ contains w_{max} it is evident that $MF(v, u, G) \leqslant MF(v, x, G) + MF(x, u, G)$, since $MF(v, u, G) = w_{max}$ and $MF(v, x, G) \geqslant w_{max}$.

Case 4. It is symmetric to the previous case, and thus, we do not elaborate.

Therefore, in any case triangular inequality is satisfied, which means that the maxflow distance respects all metric space properties. Therefore, metric access methods may be used to organize the nodes, which is important considering the size of graphs required by modern applications. It is noted that there is no restriction on the selection of the scoring function, except from the requirement that the function must be monotone, in order for the threshold algorithms to work correctly. □

5.2 Skyline-based proximity computation

In several cases, the selection of an appropriate scoring function is difficult. In such a case, an alternative proximity computation method may be applied which does not require the use of a scoring function and it is based on the concept of *Pareto dominance*. The main idea is to return to the user the nodes, called *skyline nodes*, that are not dominated by other nodes. If s is the source node, then each node v is represented as a record $r(v) =< v_1, v_2 >$, with two attributes v_1 and v_2 corresponding to the shortest path and the maxflow distance respectively. A node v dominates node u ($v \prec u$), iff v is at least as good as u in every dimension, and it is strictly better that u in at least one of them, formally $v \prec u \iff \forall i, v_i \leqslant u_i \wedge \exists j : v_j < u_j$. Therefore, the skyline nodes corresponding to the source node s is given by $SKY(s) = \{v : \nexists u, u \prec v\}$.

According to the previous discussion it is evident that our techniques can handle proximity computation even if there is no scoring function provided. This feature is extremely useful in cases where the selection of a scoring function is difficult. The set of skyline nodes is easily provided by the proposed algorithms by performing minor modifications to the threshold-based algorithms. An important issue that must be considered in this case is to prevent result overflow, since the number of different proximity measures or input graphs is directly related to the number of skyline nodes.

6. CONCLUSIONS

Proximity computation in graphs is an important problem in graph mining. In this work, we introduce threshold-based algorithms to compute the most proximal nodes around a source node s by using multiple graph instances and multiple distance measures. More specifically, we have focused on the synergy between shortest path and maximum flow to derive intuitive proximity measures. Four algorithms have been studied. The first one, TAG-I enables proximity computation without the need for preprocessing, which is an important property for time-evolving data. Algorithm TAG-II may be applied if preprocessing is allowed to the graph where maxflow operations are applied and provides significant performance boost. TAG-III changes the way sorted accesses are performed by executing expansions on the Gomory-Hu tree of H and random accesses on G. Finally, TAG-IV performs a synchronized traversal by executing sorted accesses on G and $T(H)$ and filling missing values by executing shortest path computations on G and maxflow computations on $T(H)$. In general, TAG-II is the most efficient variation regarding running time, followed by TAG-IV in most cases. However, TAG-IV requires significantly less probes, and therefore is very attractive for middleware-based proximity computation. There are several directions for future work:

- An interesting extension is to use randomized algorithms for the computation of shortest path and maxflow distances, to speed-up processing. Randomization may be applied either on the strategy of performing sorted and random accesses or on the computation of maxflow. This could be a nice alternative when no preprocessing is allowed to the input graphs, meaning that TAG-I is the only alternative. The loss in accuracy should be bounded in order to have probabilistic guarantees (i.e., the result must be valid with high probability) regarding effectiveness.

- A second direction is the investigation of the I/O and communication cost incurred in distributed processing of graph proximity queries. Reducing the I/O cost requires efficient and effective graph partitioning schemes to break the graph in disk pages. Reducing the communication cost requires careful scheduling of sorted and random accesses when graphs are distributed in multiple servers.

- Finally, a challenging problem is the adaptation of the algorithms for dynamic graphs, where insertions and deletions of nodes and edges are possible. In such a case, all algorithms, except TAG-I, may face problems because they require preprocessing. The challenge is to adapt the algorithms in the dynamic case without sacrificing result accuracy.

Acknowledgements

The authors are grateful to Xifeng Yan for providing the gene expression network.

7. REFERENCES

[1] C. C. Aggarwal, H. Wang. "*Managing and Mining Graph Data*", Springer, 2010.

[2] R. Albert, H. Jeong, A.-L. Barabasi. "Diameter of the World Wide Web ", *Nature*, 401, pp.130-131, 1999.

[3] D. Chakrabarti, "*Tools for Large Graph Mining*", PhD thesis, CMU-CALD-05-107, Carnegie Mellon University, 2005.

[4] B. Chazelle, "Computing on a Free Tree via Complexity-Preserving Mappings", *Algorithmica*, 2, pp.337-361, 1987

[5] P. Yu. Chebotarev, E. V. Shamis. "On Proximity Measures for Graph Vertices", *Automation and Remote Control*, 59(10), pp.1443-1459, 1998.

[6] D. J. Cook, L.B. Holder (eds). "*Mining Graph Data*", Wiley, 2007.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. "*Introduction to Algorithms*", Second Edition, MIT Press and McGraw-Hill, 2001.

[8] R. Fagin, "Optimal Aggregation Algorithms for Middleware", *Proceedings of ACM PODS Conference*, 2001.

[9] M. Faloutsos, P. Faloutsos, C. Faloutsos: "On Power-Law Relationships of the Internet Topology", *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999.

[10] C. Faloutsos, K. S. McCurley, A. Tomkins. "Fast Discovery of Connection Subgraphs", *Proceedings of the ACM SIGKDD Conference*, 2004.

[11] G. W. Flake, S. Lawrence, C.L. Giles. "Efficient Identification of Web Communities", *Proceedings of ACM SIGKDD Conference*, 2000.

[12] G. W. Flake, R. E. Tarjan, K. Tsioutsiouliklis. "Graph Clustering and Minimum Cut Trees", *Internet Mathematics*, 1(4), 2003.

[13] A. V. Goldberg, R. E. Tarjan. "A New Approach to the Maximum Flow Problem". *ACM Symposium on Theory of Computing*, 1986.

[14] A. V. Goldberg, K. Tsioutsiouliklis. "Cut-tree Algorithms: an experimental study", *Journal of Algorithms*, 38(1), pp.51-83, 2002.

[15] R. E. Gomory, T. C. Hu. "Multi-terminal Network Flows", *Journal of the Society for Industrial and Applied Mathematics*, 9(4), pp.551-570, 1961.

[16] D. Gusfield. "Very Simple Methods for All Pairs Network Flow Analysis", *SIAM Journal of Computing*, 19(1), pp.143-155, 1990.

[17] J. Hao, J.B. Orlin. "A Faster Algorithm for Finding the Minimum Cut in a Graph", *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, 1992.

[18] D. Harel, R. E. Tarjan: "Fast Algorithms for Finding Nearest Common Ancestors", *SIAM Journal of Computing*, 13, pp.338-355, 1984.

[19] H. Ino, M. Kudo, A. Nakamura. "Partitioning of Web Graphs by Community Topology", *Proceedings of the 14th International Conference on World Wide Web (WWW)*, 2005.

[20] Y. Koren, S. C. North, C. Volinsky. "Measuring and Extracting Proximity in Networks", *Proceedings of ACM SIGKDD Conference*, 2006.

[21] E. A. Leicht, P. Holme, M. E. J. Newman. "Vertex Similarity in Networks", *Physics Review E*, 73, 2006.

[22] J. Leskovec, J. Kleinberg, C. Faloutsos. "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations", *Proceedings of ACM SIGKDD Conference*, 2005.

[23] K. Menger. "Zur Allgemeinen Kurventheorie", *Fund. Math.*, 10, 1927.

[24] M. Stoer, F. Wagner. "A Simple Min-Cut Algorithm". *Journal of the ACM*, 44(4), pp.585-591, 1997.

[25] J. Sun, H. Qu, D. Chakrabarti, C. Faloutsos. "Neighborhood Formation and Anomaly Detection in Bipartite Graphs", *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM)*, 2005.

[26] W. Tang, Z. Lu, I. S. Dhillon. "Clustering with Multiple Graphs", *Proceedings of the 9th International Conference on Data Mining (ICDM)*, 2009.

[27] H. Tong, C. Faloutsos, J.-Y. Pan. "Fast Random Walk with Restart and Its Applications", *Proceedings of the 6th International Conference on Data Mining (ICDM)*, 2006.

[28] H. Tong, C. Faloutsos. "Center-Piece Subgraphs: Problem Definition and Fast Solutions", *Proceedings of the ACM SIGKDD Conference*, 2006.

[29] F. Vigen, M. Latapy. "Efficient and simple generation of random simple connected graphs with prescribed degree sequence". *Proceedings of the 11th International Conference on Computing and Combinatorics*, 2005.

[30] H. Whitney. "Congruent Graphs and the Connectivity of Graphs", *Amer. J. Math*, 54, pp.150-168, 1932.

[31] X. Yan, M.R. Mehan, Y. Huang, M.S. Waterman, P.S. Yu, X.J. Zhou. A Graph-Based Approach to Systematically Reconstruct Human Transcriptional Regulatory Modules. *Bioinformatics*, 23(13), pp.i577-i586, 2007.