Islands and Bridges: Making Sense of Marked Nodes in Large Graphs*

Leman Akoglu* **Jilles Vreeken†**
Hanghang Tong‡ **Duen Horng Chau***
Christos Faloutsos*

August 2012
CMU-CS-12-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

* CMU, SCS {lakoglu,dchau,christos}@cs.cmu.edu
†University of Antwerp. jilles.vreeken@ua.ac.be
‡IBM T. J. Watson. htong@us.ibm.com

* Research was sponsored by the National Science Foundation under Grant No. IIS1017415 and the Army Research Laboratory under Cooperative Agreement No. W911NF-09-2-0053. It is continuing through participation in the Anomaly Detection at Multiple Scales (ADAMS) program sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under Agreements No. W911NF-11-C-0200 and W911NF-11-C-0088. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory, of the National Science Foundation, of the U.S. Government, or any other funding parties. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. This work is also partially supported by an IBM Faculty Award. Jilles Vreeken is supported by a Post-Doctoral Fellowship of the Research Foundation – Flanders (FWO).

Keywords: link mining, connection subgraphs, sensemaking

Abstract

Suppose we are given a large graph in which, by some external process, a handful of nodes are marked. What can we say about these marked nodes? Are they all close-by in the graph, or are they segregated into multiple groups? How can we automatically determine how many, if any, groups they form as well as find simple paths that connect the nodes in each group? We formalize the problem in terms of the Minimum Description Length principle: a set of paths is simple when we need few bits to describe each path from one node to another. For example, we want to avoid high-degree nodes, unless we need to visit many of its spokes. As such, the best partitioning requires the least number of bits to describe the paths that visit all marked nodes. We show that our formulation for finding simple paths between groups of nodes has connections to well-known other problems in graph theory, and is NP-hard. We propose fast effective solutions, and introduce DOT2DOT, an efficient algorithm for partitioning marked nodes as well as finding simple paths between nodes within parts. Experimentation shows DOT2DOT correctly groups nodes for which good connection paths can be constructed, while separating distant nodes.

1 Introduction

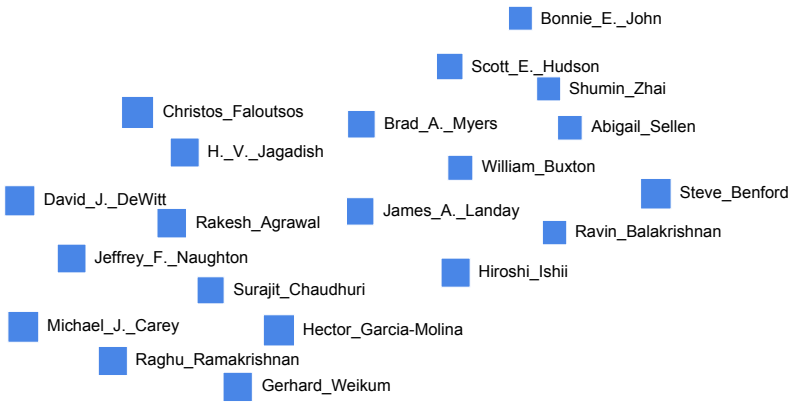
Suppose we are given a large graph $G = (V, E)$ in which, by some external process, a handful of nodes $M \subseteq V$ are marked. How can we explain the relations among these marked nodes? Do they 'talk to' each other and thus are 'close-by' or do they 'dislike' or 'not-know' each other and thus are 'far apart' in the graph? How many groups are they segregated into, if at all? How are they connected and who are good connectors? More formally, the key question we address in this paper is: How can we use the network structure of G to explain the marked nodes M, by partitioning them such that for each part we have simple paths connecting the grouped nodes, while nodes in different parts are not easily reachable?

For example, consider Figure 1. In (a), a list of 20 authors from DBLP are marked. In this plot, there is no information (other than author names) that explains any correlation among the authors. In (b), the marked nodes are projected to and highlighted in the co-authorship graph. In contrast, here it is hard to observe any patterns as there is information overload. We show our result in (c), which explains the marked nodes with two well-separated groups (islands) as well as revealing simple connections and connectors (bridges) among all the marked nodes.

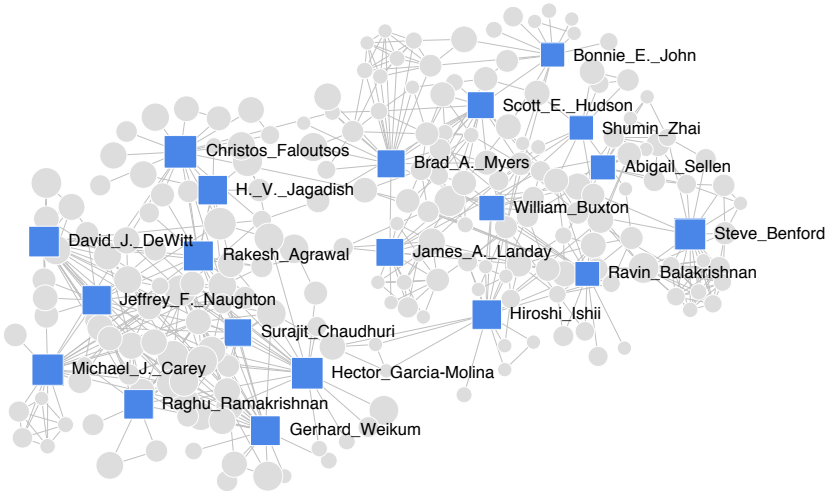
To the best of our knowledge, the problem that we introduce above is a new problem that has not been studied before. In this paper, we formally define and formulate the problem using information and graph theoretic principles and propose effective solutions. While the problem is interesting in its own right, it also has several motivating applications. We list some examples in the following.

- Given a gene interaction network, an experiment may reveal for particular conditions a number of genes to be up (or down) regulated [19], how can we partition them with respect to their closeness in the graph? This would give us a good summary of groups of close-by correlated genes as well as possible pathways the genes may be involved in.
- Given k nodes marked by an anomaly detection algorithm, how can we explain the anomalies? Instead of simply listing them, we could group them and reveal their associations within groups.
- Given an event affecting a set of nodes in a graph (e.g. people affected by a certain disease, people buying a particular product), how can we group the nodes such that the network structure can be associated with the spread of the event within groups but not quite across groups?
- Given a social graph like Facebook and user attributes of interest (e.g. white girls and black boys in the same school), how can we explain their relations using the graph structure? The partitions, if any, and connection paths among the chosen students may be of interest to segregation studies [26] in social sciences among others.
- Given the Web graph and a set of top ranked pages (nodes) returned by some keyword search, how can we group these nodes matching the keywords into groups and reveal connection paths among them, rather than simply listing them?

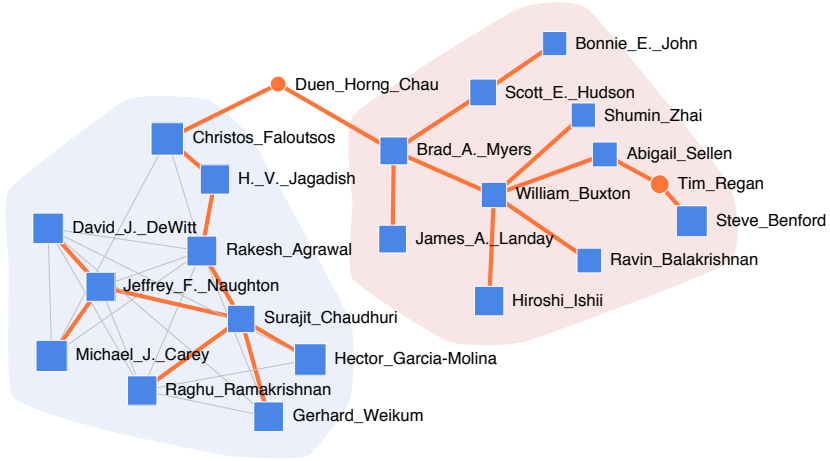
Intuitively, we want to partition the given set of marked nodes such that the nodes within a part are 'close-by' while nodes across parts are far apart. In addition, for the 'close-by' nodes in each part, we want to find a 'succinct' subgraph connecting them. Moreover, we rather not visit nodes



(a) What to say about this "list" of authors?



(b) Any patterns? "Too many" connections.



(c) The "right" connections → Better sensemaking

Figure 1: 20 chosen authors from DBLP. Edges denote co-authorship relations.

of very high degree, such as hubs in social networks, because those nodes connect to virtually everything in the graph, and they do not provide much information by association. Therefore, we think of such high degree nodes as separators, and try to avoid including them in our 'succinct' subgraphs.

We formalize this problem in terms of the Minimum Description Length principle [23]: a collection of paths is simple when we need few bits to direct the user from one node to the other. Hence we typically do not want to visit nodes of high degree, as it is more expensive to identify which edge to follow leading from it. Similarly, we require more bits if we have to visit many unmarked nodes in order to arrive to the next marked node. As such, the best 'explanation', is the one for which we need the least number of bits to identify all marked nodes.

We show this problem is NP-hard, and has connections to well-known other problems in graph theory. We discuss a number of fast methods for finding a partitioning and its respective simple paths, and introduce DOT2DOT, an efficient algorithm for explaining marked nodes in large graphs. Experimentation shows that DOT2DOT correctly groups nodes for which simple paths can be constructed, while separating distant nodes.

As we mentioned before, explaining marked nodes in a graph is a novel problem setting that has not been explored before. There, however, exists relations to earlier proposals. Graph clustering [10, 11, 16] is related in that we aim at grouping nodes that are nearby—yet our goal is not to cluster the full graph, but only the marked nodes, making use of the graph structure. Other related methods include connection subgraphs [9], center-piece subgraphs [28], local graph clustering [3], seed set expansion [22] etc., however, these proposals each targets a different problem and does not provide sensemaking via partitioning. §5 includes more detailed discussion on related work.

Our major contributions are: we formally define the problem of 'describing marked nodes in graphs' and formulate it in terms of information theoretic principles (§2), propose fast methods for finding the partitions as well as high quality connection paths within parts, that gives the best 'description' (§3), and experimentally evaluate our methods on real and synthetic data (§4).

2 Formulating Paths: Theory

In this section, we give the formal definition of the problem we consider. Later, we formulate the problem and show its hardness.

2.1 Problem Definition

Given a graph $G = (V, E)$ and a set of marked nodes $M \subseteq V$, we consider the following two related problems.

- **Problem 1. Optimal partitioning** Find a 'coherent' partitioning P of M. Find the optimal number of partitions $|P|$ automatically.
- **Problem 2. Optimal connection subgraphs** Find the 'minimum cost' set of subgraphs that connect the nodes in each part $p_i \in P$ efficiently.

The two problems we consider above involve three sub-problems:

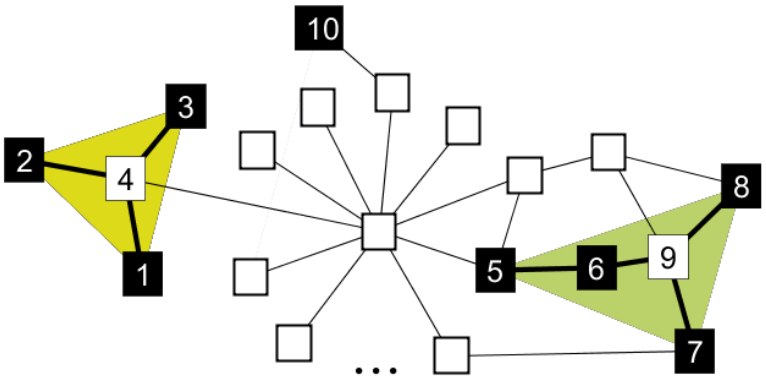


Figure 2: Toy graph with 8 marked (black) nodes. Our DOT2DOT algorithm automatically 'describes' them in 3 groups, discovering 'missing connectors' (nodes '4' and '9').

- **(sub-problem 1)** how to define the 'coherence' of a set of nodes,
- **(sub-problem 2)** how to define the 'cost' for a connection subgraph, and
- **(sub-problem 3)** how to find the connection subgraph(s) quickly in large graphs.

As an example, let us consider Figure 2. In it, we depict a simple graph in which 8 nodes have been marked. It is clear to see that given the graph structure, the marked nodes naturally form three groups: $p_1 = \{1, 2, 3\}$, $p_2 = \{10\}$, and $p_3 = \{5, 6, 7, 8\}$, which are all well separated by the big star-node in the middle. While only nodes 5 and 6 have a direct connection, for each part there exists a connection subgraph, here highlighted with bold edges, such that we can construct a simple dot-to-dot 'road map' of which branches to follow in order to visit all marked nodes in a part—without having to visit too many unmarked connector nodes, like node 4 for p_1, and node 9 for p_3.

Since we are interested in summarizing, that is, in describing the given marked nodes as succinctly as possible, we borrow ideas from Information Theory and employ the Minimum Description Length (MDL) principle (see Section **??** for background on MDL).

2.2 Formulating Tours: Theory and Objective

The key idea behind our method builds on an encoding scheme, involving one sender and one receiver. We assume both the sender and the receiver already know graph $G = (V, E)$ and only the sender knows the set of marked nodes M. The goal of the sender, then, is to come up with an encoding scheme to *transmit to the receiver the information of which nodes are marked, using as few bits as possible*.

One straightforward way to encode the set of marked nodes is to encode their node-id's separately, using $\log |V|$ bits each. On the other hand, it might be more efficient to exploit the neighborhood information for 'close-by' nodes. In the simplest case, for example, if nodes u and v are both marked and they are direct neighbors, i.e. $e(u, v) \in E$, then the sender can encode u's id and then use only $\log d(u)$ bits to encode v, where we assume a canonical order on nodes (say by increasing node-id) and $d(u)$ denotes the degree of u in G. As such, the sender follows a path from

one marked node to the other to encode 'close-by' nodes. Depending on the graph structure, from time to time it might be more efficient to restart encoding from a new node by directly providing its id, in case the cost of the path to that node exceeds $\log |V|$ bits. In fact, that is exactly what determines the partitioning P of the nodes.

Simply put, one can imagine the way of encoding discussed above as hopping from node to node for encoding close-by nodes and from time to time flying to a completely new node for encoding farther nodes until all marked nodes are encoded. This resembles a *tour* (union of paths) on the graph that travels from a marked node to another which succinctly describes the marked set (hence the name DOT2DOT).

In effect, we are after the shortest description for a group of marked nodes $M \subseteq V$ in a graph $G = (V, E)$. More generally, the idea is that per part p_i of P, we find the easiest/simplest subgraph T in G that spans *at least* all marked nodes in p_i. Simplicity of T is determined by the number of nodes we visit in this tour, how many unmarked nodes we visit, and in particular how easily per visited node we can identify which edge we have to follow next; nodes with (very) high degree hence make the path more complex, or, less likely. Also notice that the simplest subgraph would in fact be a ***tree*** since it would require less bits to refer to a node we have already visited in our encoding.

In this section, we describe the cost function for a given partitioning P and the given connection trees for each part p_i. We use this cost function as our objective function that we aim to minimize for model selection.

More formally, we first transmit the number of partitions $|P|$, for we need at most $\log |V|$ bits as there will be at most $|V|$ marked nodes in total, which in the worst case are all put in separate parts.

$$L(|P|) = \log |V| \tag{1}$$

Then, per part $p_i \in P$, we have a tree T spanning at least the marked nodes of p_i. To identify the root node of T in G we have to spend $\log |V|$ bits.

Then, recursively per node $t \in T$, we transmit how many branches t has, denoted by $|t|$. As t corresponds to a node v_t in G, and $d(v_t)$ gives us the out-degree of v_t, we can transmit $|t| \leq d(v_t)$ in $\log d(v_t)$ bits. On the other hand, since a simple tree would presumably have small branch-out factor, we choose to encode $|t|$ using universal integer encoding [**?**]. This encoding specifies that in order to encode a non-zero positive integer n we require $L_{\mathbb{N}}(n) = \log^{\star} n + \log(c)$ bits with $c = \sum 2^{-\log n} \approx 2.865064$, and $\log^{\star}(n) = \log n + \log\log n + \cdots$ sums over all positive terms. So, as $|t|$ can be zero (for leaves), we transmit its value in $L_{\mathbb{N}}(|t| + 1)$ bits.

Next, we identify which out-edges of v_t have to be visited. This we can encode most succinctly by assuming a canonical order of all possible subsets of selected edges of that size, and transmitting the index of the actual subset. This takes $\log \binom{d(v_t)}{|t|}$ bits.

Leaf-nodes are easily identified as their number of branches is given as 0. If such is the case, we traverse back up the tree until we find a node with an unvisited branch. Once all branches have been visited, we stop transmitting the structure of the tree.

Now that we know which nodes $p_i \subset V$ are in our tour, we need to know which of these are marked. Let $|T|$ denote the number of nodes in T, and $||T||$ the number of marked nodes in T. As the recipient now knows the tree, and hence $|T|$, and $||T|| \leq |T|$ we need $\log |T|$ bits to

transmit $||T||$. Next, we need to identify which $||T||$ nodes of T are marked, which we again do by a binomial: $\log \binom{|T|}{||T||}$.

As such, for one part $p_i \in P$, we have

$$L(p_i) = \log |V| + L(t) + \log |T| + \log \binom{|T|}{||T||} \tag{2}$$

in which per node in the tree of p_i

$$L(t) = L_{\mathbb{N}}(|t| + 1) + \log \binom{d(v_t)}{|t|} + \sum_{j}^{|t|} L(b(t, j))$$

where $b(t, j)$ identifies the node t' in T we reach from node t by descending branch j (notice that by its recursive nature, $L(t)$ encodes the branching cost for all tree-nodes).

Putting this together, we get

$$L(P, M \mid G) = L(|P|) + \sum_{i} L(p_i) \quad . \tag{3}$$

Note that our initial assumption that both the sender and the receiver know G does not affect model selection: as G is constant for all possible sets of marked nodes on G, explicitly transmitting G would only add a constant cost for all possible models under consideration, and hence not influence measuring the quality of a model.

Given the above formalization, we now only have to find the optimal partitioning P of M, and the optimal tours per part $p_i \in P$ such that $L(P, M \mid G)$ is minimized.

2.3 NP-hardness

The total encoding cost $L(P, M \mid G)$ can score a given set of solutions and point to the best among them, however it does not provide direct means to find the optimal solution.

In this section, we show that this problem is NP-hard, with a reduction to the well-known Steiner tree problem: given an undirected unweighted graph $G = (V, E)$ and a subset of nodes $X \subseteq V$, the objective is to find the minimum cost tree that spans all the nodes in X. The cost of a tree is defined as the total number of nodes, i.e. total number of edges, in the tree. Note that the tree may include nodes in X, as well as other nodes which are typically referred to as Steiner nodes. Steiner trees are well-known in graph theory, and find application in multicast models for finding good server nodes in computer networks for avoiding network congestion and reducing latency in multi-user streaming data settings.

Theorem 1 *Minimizing $L(P, M \mid G)$ is NP-hard.*

Proof 1 *Let $G = (V, E)$ be a graph and let $M \subseteq V$ be the marked nodes. Let $k = |V|$ the number of nodes. We will construct a graph $G' = (V', E')$ such that solving the Steiner tree problem for G will reduce to finding a partition with optimal cost in G'.*

In order to do that, let d be the maximal degree of a node in G. We are now ready to define the graph G'. The nodes V' consists of the nodes V and for each edge $e \in E$ we will add s auxiliary nodes. We will define s later. In addition, we will add a large amount, say r, of isolated nodes, we will specify r later. For each edge $e(v, w) \in E$ we will add a path of $s + 2$ nodes connecting v and w with s nodes that were created specifically for this edge. We will refer to these paths as auxiliary paths.

Note that auxiliary nodes always have a degree of 2 and they will never be leaves of trees corresponding to the optimal partition. That is, either the whole auxiliary path is in the partition or none of the nodes in that path is included.

The cost of having a non-root auxiliary node in a partition is $c = L_{\mathbb{N}}2 + \log \binom{2}{1}$. If auxiliary node is a root, then the cost is $c_r = L_{\mathbb{N}}3 + \log \binom{2}{2}$.

The encoding of one part is equal to

$$\log |V'| + ncs + corr + \log |T| + \log \binom{|T|}{\|T\|} + \sum_{t \in T, v_t \in V} L_{\mathbb{N}}|t| + 1 + \log \binom{d(v_t)}{|t|},$$

where n is the number of auxiliary paths in T. The correction term $corr$ takes into account the possibility that auxiliary node may be a root term. If this is the case, then $corr = c_r - c$, otherwise $corr = 0$. We can bound the last 4 terms by

$$b = \log k + k \log k + k(L_{\mathbb{N}}d + 1 + d \log d).$$

We can now bound the cost by

$$\log |V'| + ncs + c_r - c + b.$$

Let us define $s = (c_r + c + b)/c$. It is easy to see that the smaller the n, the better is the code for the tree.

Our last step is to make sure that we will have only part in our partition. In order to do that we need to make sure that $|V'|$ is so large that is not beneficial to have more than 1 parts. In order to do this, we set $r = 2^{kcs+c_r+c+b}$ which will guarantee that we will have only one partition.

The optimal partition in G' will now have only one part and will have minimal number of auxiliary paths. Since each path corresponds to an edge in G, we can transform this to a Steiner tree in G with minimal number of edges. This proves the theorem.

3 Finding Good Paths: Methods

Since minimizing $L(P, M \mid G)$ is NP-hard, we are interested in fast approximations. To find fast solutions, we will exploit heuristics for the directed Steiner (d-Steiner) tree problem [5]: given a directed weighted graph $G = (V, E)$ and a subset of nodes $X \subseteq V$, the objective is to find the minimum cost *arborescence* that spans all the nodes in X, in other words, a rooted minimum cost tree that has a directed path from the root to every node in X. The cost of a tree is defined as the sum of the costs of the edges in the tree.

The d-Steiner problem is a well-studied combinatorial optimization problem, for which algorithms have been proposed that provide approximation guarantees [5, 13, 20, 32]. However, while providing fairly good bounds, these algorithms all require great amounts of computing power, making them intractable for application on large graphs. In this section we therefore propose fast heuristics for large graphs.

Before we proceed with proposed solutions, we first define how we transform the input graph G to a directed weighted graph G', which we use in obtaining a solution.

Definition 1 *Given an undirected unweighted graph $G = (V, E)$ and a set of marked nodes $M \subseteq V$, the transformed graph $G' = (V, E')$ is a directed weighted graph in which each edge $e(u, v) \in E$ is replaced by two directed edges $e'(u, v) \in E'$ and $e'(v, u) \in E'$, for which the weights $w'(u, v) = \log d(u)$ and $w'(v, u) = \log d(v)$.*

3.1 Proposed Algorithms

Given the transformed directed weighted graph G', we want to find *the set of trees with minimum total cost* on the marked nodes. Without loss of generality we assume that the marked nodes will be the leaf nodes in the resulting trees (if a marked node is a non-leaf node, we can always add its copy and connect it to its copy with a zero-cost edge), therefore from hereafter we refer to the marked nodes as the *terminals*.

3.1.1 Finding Bounded-length Paths

Most of our proposed methods use short paths between the terminals as a starting point. Therefore, we first present an algorithm, called `FindBoundedPaths`, to find *multiple* short paths of up to a certain length (hence, indirectly, up to a certain cost), and in order of increasing length between the terminal nodes. The threshold on the length of the paths is not a parameter; as it takes $\log |V|$ bits to start a new partition, we set it to $\log |V|$.

`FindBoundedPaths` employs a BFS-like expansion starting from each terminal until the threshold path length is reached. The paths from the terminal to the nodes encountered over this expansion as well as their total lengths are stored. A major advantage of our transformed graph formulation for the BFS-like expansion is that the cost of all out-edges for a node v are the same and equal to $\log d(v)$. As a result we only need to keep the length per node, rather than per edge, in our BFS-list.

The pseudo-code is given in Procedure 1. First, $lengths$ is a vector of size $|V|$ that holds $\log d(v)$ for each node $v \in V$ (line 1). Second, $paths$ is a dynamic list that stores all current list of paths during the BFS expansion. Third, $pathcosts$ stores their respective lengths. Last, SP is a structure list that stores the paths from a terminal to other nodes encountered in the BFS expansion (2). The paths are indexed by their end nodes and stored in increasing order of their lengths.

In each iteration, we expand the node(s) with the minimum length (7). For each such node v (8), we expand to its neighbors N_v (10). We first store the path information to these neighbors (12), and then add the new paths from the root to these neighbors to the $paths$ list (13), and their respective lengths to the $pathcosts$ list (14) for further expansion. Note that a node can appear

multiple times in our BFS lists since each expansion is associated with only a single unique path (that is, the path from the root in the BFS tree), and a node may belong to multiple paths. We continue iterating, i.e. expanding nodes with minimum length (16) until the threshold length is reached (6).

At the end of `FindBoundedPaths`, we have all the paths from every terminal to all the nodes in G' that are within a length $\log|V|$ path, ordered in increasing length. Notice that the expansion can be performed completely in parallel for each terminal for speed.

Procedure 1: `FindBoundedPaths` (G, T)

Input: A graph $G = (V, E)$, terminals $T \subseteq V$
Output: multiple (asc. length) short paths SP from terminals

1 $lengths \leftarrow \log(degree(V))$
2 $paths \leftarrow \emptyset$, $pathcosts \leftarrow \emptyset$, $SP \leftarrow \emptyset$, curlen $\leftarrow 0$
3 **foreach** $t \in T$ **do**
4 $paths.add(\{t\})$, $pathcosts.add(lengths(t))$
5 curlen $\leftarrow$ curlen + $\min(pathcosts)$
6 **while** curlen $< \log(|V|)$ **do**
7 $pathcosts \leftarrow pathcosts - \min(costs)$
8 **foreach** v *s.t.* $pathcosts(v) = 0$ **do**
9 $path \leftarrow paths(v)$
10 **foreach** $n \in N_v$ **do**
11 **if** $n \notin path$ **then**
12 SP.add(from: t, to: n, len: curlen, $path$)
13 $paths$.add($\{path, n\}$)
14 $pathcosts$.add($lengths(n)$)
15 $paths$.remove(v), $pathcosts$.remove(v)
16 curlen $\leftarrow$ curlen $+ \min(pathcosts)$
17 **return** SP

Next, we present our fast approximate solutions for finding a low-cost set of trees on the terminals.

3.1.2 DOT2DOT-ConnectedComponents

A basic heuristic to partition the terminals is to simply consider the connected components they induce on the graph. That is, the terminals that are directly connected are put in the same and otherwise separate parts. By definition, all the edges are reciprocated in the transformed graph, and therefore the subgraph induced on each part including two or more nodes is not a directed acyclic tree (it may as well have cycles of length greater than 2). To find the minimum cost rooted directed tree(s) (a.k.a. arborescence), we use the Chu-Liu algorithm [8].

3.1.3 DOT2DOT-MinArborescence

Our second method (Algorithm 1) uses the *transitive closure* graph of the terminals to find a minimum arborescence. The transitive closure graph $G_t = (T, E_t)$ consists of the terminals and directed edges $e(t_i, t_j) \in E_t$ between terminals having weight equal to the shortest path length $w(t_i \rightsquigarrow t_j)$ from t_i to t_j, $1 \leq i, j \leq |T|$. If the shortest paths between all pairs of terminals are of length less than $\log |V|$, then G_t is simply a directed clique graph. As we find up to length $\log |V|$ paths from every terminal in `FindBoundedPaths`, a terminal does not have an edge in G_t to those terminals that are more than this threshold apart from it.

Having constructed the transitive closure graph (2), we add a so-called universal node u with directed edges $e(u, t_i)$ to every terminal t_i with weight $\log |V|$ (3). We find the minimum weight arborescence A in this new graph (4). Since the universal node u does not have any incoming edges, it constitutes the root of A. In fact, the number of out-edges of u in A gives us the number of parts $|P|$, and its sub-trees constitute the partitioning P (5). Next, we replace the edges in each of u's sub-trees with their corresponding paths in the transformed graph G' (6). The expanded sub-trees may contain both marked and unmarked nodes. Also notice that the expanded sub-trees might no longer be trees but contain cycles. Therefore, we rerun the arborescence algorithm on the expanded sub-trees and remove any unmarked leaf nodes in the resulting arborescences, which yields the final forest of Steiner trees. The min-arborescence algorithm takes $O(|V|^2)$ for dense graphs. As we run it on the closure of marked nodes only, we get $O(|M|^2)$.

Algorithm 1: DOT2DOT-MINARBORESCENCE

Input: A graph $G = (V, E)$, terminals $T \subseteq V$
Output: Partitions P: a Steiner tree on each $p \in P$

1 $SP \leftarrow$ `FindBoundedPaths` (G, T)
2 Construct distance-graph (transitive closure) $G_t = (V_t, E_t, d_t)$, where $V_t = T$, and for every $(v_i, v_j) \in E_t$, $d_t(v_i, v_j) = \min -len(SP(v_i, v_j))$.
3 Add universal node u which connects to $\forall v_i \in V_t$ with $d_t(u, v_i) = \log(|V|)$.
4 Find minimum arborescence A of G_t.
5 Delete universal node u and its out-edges from A.
6 **return** `ExpandPartition` (A, SP, T)

Procedure 2: `ExpandPartition` (R, SP, T)

Input: tree(s) R, short paths SP, terminals T
Output: Partitions P: a Steiner tree on each $p \in P$

1 Construct subgraph(s) R' by replacing each edge in R by its corresponding shortest path.
2 Find minimum arborescence A of R'.
3 Construct Steiner tree(s) S from A by deleting edges in A, if necessary, s.t. all leaves in S are terminal (marked) nodes.
4 **return** connected components of S as P.

3.1.4 DOT2DOT-1-Level Tree

Our third method (Algorithm 2) builds a (set of) level-1 tree(s) from the transitive closure graph on the terminals. Simply put, we try each terminal as the root (3) and connect it to the other terminals with shortest paths on the transformed graph G' (4-5). If the selected root does not have shorter than length $\log|V|$ paths to all the terminals, a (set of) level-1 tree(s) is built on the remaining terminals (7-8). We return the least-cost (w.r.t. Eq. 3) tree(s) as the forest of Steiner trees (10-12). Note that this heuristic algorithm provides a $|M|$-approximation to the optimal solution [5].

Algorithm 2: DOT2DOT-1-LEVELTREE

Input: A graph $G = (V, E)$, terminals $T \subseteq V$
Output: Partitions P: a Steiner tree on each $p \in P$

1 $SP \leftarrow$ `FindBoundedPaths` (G, T)
2 mincost $\leftarrow \infty$
3 **foreach** $t \in T$ **do**
4 $N_t \leftarrow \{v \in T | SP(t, v)$ exists$\}$
5 Construct 1-level tree $L_1 = (V_p, E_p, d_p)$, where $V_p = t \cup N_t$, and for every $(t, v \in N_t) \in E_p$, $d_p(t, v) = \min(SP(t, v))$.
6 $P \leftarrow$ `ExpandPartition` (L_1, SP, T)
7 **if** $|V_p| < |T|$ **then**
8 $P \leftarrow P \cup$ DOT2DOT-1-LEVELTREE $(G, T \backslash V_p, SP)$
9 **else**
10 cost $\leftarrow$ cost of partition P by Eq. 3
11 **if** cost $<$ mincost **then**
12 mincost $\leftarrow$ cost, $minP \leftarrow P$
13 **return** $minP$

3.1.5 DOT2DOT-k-Level Tree

Our final method generalizes the 1-level tree heuristic to k-level trees. The goal is to start with a (set of) 1-level tree(s) and successively refine each for lower cost. The main idea is the following: for a given tree with root r, find one or more intermediate nodes $v \in V$, such that the total cost from r to each v plus the costs of sub-trees rooted at v's (each with a mutual set of terminals as leaves) reduces the initial cost.

More specifically, we construct a k-level tree by a union of sub-trees, each consisting of the root r, exactly one intermediate node v, and all the descendants of this intermediate node. We refer to such sub-trees of level k as *partial k-tree*s, and we find them in a greedy manner described as follows. First, we find a *partial k-tree* L_k^p (if any) that reduces the cost for a subset of terminals that it spans. Next, we remove the terminals spanned by L_k^p from consideration and iterate this process until all terminals are spanned. Algorithm 3 formally describes the k-level tree heuristic.

To complete the k-level heuristic, we need to describe its subroutine `Partial`k`Tree` which, given a root r and a set of terminals S, finds a low-cost partial k-level tree rooted at r and spanning (a subset of) the terminals. The `Partial`k`Tree` heuristic, as given in Procedure 3, is recursive: in order to find a partial k-level tree, we need to first find certain partial $(k$-$1)$-level trees that span all the given terminals (8-12). The base case is reached for level $k = 2$, which works as the following. For each candidate v for the intermediate node, we sort the terminals S according to the potential savings of inserting node v between the root r and each terminal in S (3). The potential saving for a terminal t_i stands for the difference between the shortest path lengths $w(r \rightsquigarrow t_i)$ and $w(v \rightsquigarrow t_i)$. These savings can take positive and negative values and are sorted in decreasing order. Finally, we include consecutive terminals from this sorted list while their inclusion decreases cost of the *partial* 2*-tree* (4-5).

Algorithm 3: DOT2DOT-k-LEVELTREE

Input: A graph $G = (V, E)$, terminals $T \subseteq V$
Output: Partitions P: a Steiner tree on each $p \in P$

1 $SP \leftarrow$ `FindBoundedPaths` (G, T)
2 Construct candidate-graph $G_c = (V_c, E_c)$: union of all top-3 shortest paths between all pairs of terminals.
3 mincost $\leftarrow \infty$
4 **foreach** $t \in T$ **do**
5 $L_k \leftarrow \emptyset$
6 $\mathcal{L}_1 \leftarrow$ min-cost 1-level tree(s), one rooted at t
7 **foreach** $L_1 \in \mathcal{L}_1$ **do**
8 $S \leftarrow leaves(L_1),\ r \leftarrow root(L_1)$
9 **if** $|S| < 2$ **then** continue
10 **while** $S \neq \emptyset$ **do**
11 $L_k^p \leftarrow$ `Partial`k`Tree`(G_c, SP, r, S, k)
12 $L_k \leftarrow L_k \cup L_k^p$
13 $S \leftarrow S \backslash leaves(L_k^p)$
14 $P \leftarrow$ `ExpandPartition` (L_k, SP, T)
15 cost $\leftarrow$ cost of partition P by Eq. 3
16 **if** cost $<$ mincost **then**
17 mincost $\leftarrow$ cost, $minP \leftarrow P$
18 **return** $minP$

3.2 Discussion

Before we conclude this section, we discuss some aspects of the k-level tree heuristic.

Firstly, note that in order to find good intermediate nodes v, Procedure `Partial`k`Tree` considers all the nodes in the given input graph (1). This raises two issues: 1) we would need the

Procedure 3: `Partial`k`Tree` (G_c, SP, r, S, k)

Input: A graph $G_c = (V_c, E_c)$, short paths SP, root r, a set S of nodes, level k
Output: partial k-level tree L_k^p with intermediate node $v \in \{V, \emptyset\}$ and $leaves(L_k^p) \subseteq S$

foreach $v \in V_c$ **do**
  if $k = 2$ **then**
    Sort $S = \{N_1, \ldots, N_{|S|}\}$ such that
    $SP(r, N_i) - SP(v, N_i) \geq SP(r, N_{i+1}) - SP(v, N_{i+1})$
    Find $j \in \{1, \ldots, |S|\}$ that minimizes $cost(v) \leftarrow SP(r, v) + \sum_{i=1}^{j} SP(v, N_i)$
    $S(v) \leftarrow \{N_1, \ldots, N_j\}$
  else
    $S' \leftarrow S$, $L_k^p \leftarrow (r, v)$, $cost(L_k^p) \leftarrow \infty$
    while $S' \neq \emptyset$ **do**
      $L_{k-1}^p \leftarrow$ `Partial`k`Tree`$(G_c, SP, v, S', k\text{-}1)$
      if $cost(L_k^p) \leq cost(L_{k-1}^p)$ **then** exit while
      $L_k^p \leftarrow L_k^p \cup L_{k-1}^p$
      $S' \leftarrow S' \backslash leaves(L_{k-1}^p)$
    $cost(v) \leftarrow cost(L_k^p)$, $S(v) \leftarrow leaves(L_k^p)$
Find v having minimum $cost(v)$
return partial k-level tree L_k^p with intermediate node v rooted at r and leaves $S(v)$

shortest paths from the root r to every v as well as from v to its descendants, and 2) the iteration would become computationally infeasible even for moderate size graphs. As a result, we provide `Partial`k`Tree` with the so-called *candidate graph* $G_c = (V_c, E_c)$. The candidate graph consists of the union of the nodes and edges in all top-t shortest paths between every pair of terminals. Since G_c is much smaller than G' and as we have the shortest paths to the nodes in it computed, both issues are addressed. In the experiments we use $t = 3$, which can be adjusted depending on available computational resources. Considering a larger candidate graph may help finding lower cost trees while a smaller candidate graph provides lower running time as well as better visualization.

Secondly, notice that the cost of a *partial 2-tree* is computed by the sum of the shortest paths from root r to intermediate node v, and from v to a set of terminals (4). Using `FindBoundedPaths` recall that we find short paths from *marked* nodes to the nodes within $\log |V|$ distance. Since v might be an unmarked node, we might need the shortest path length from an *unmarked* node to a marked one. We find the shortest path from unmarked nodes in the candidate graph to marked nodes using the following Lemma.

Lemma 1 *The shortest path from i to j and the shortest path from j to i for all $i, j \in V'$ in the transformed graph G' contain exactly the same nodes with edges in reverse directions. Due to symmetry, the total weights of the reciprocal shortest paths obey the following equation.*

$$w(i \rightsquigarrow j) = w(j \rightsquigarrow i) - \log d(j) + \log d(i)$$

Proof 2 *Let* $\{i \to v_1 \to v_2 \to \dots v_l \to j\}$ *denote the shortest path from* i *to* j*. Notice that any path from* i *to* j *contains an out-edge of* i *to one of its neighbors, with weight* $\log d(i)$*. Thus,* $w(i, v_1) = \log d(i)$*. Let* $R = \sum_{k=1}^{l-1} w(v_k, v_{k+1}) + w(v_l, j)$*, such that* $\log d(i) + R$ *minimizes total length of the path. Since all out-edges of a node* $v \in V'$ *are equal, we can also write* $w(v_1, i) + \sum_{k=l}^{2} w(v_k, v_{k-1}) = R$*. So the reverse path from* j *to* i *has length* $\log d(j) + R$*. As* R *gives the shortest path length from* i *to* j*, it also gives the shortest from* j *to* i*. Hence the lemma holds.*

As a result, given that we know the shortest paths from terminals to other nodes $v \in V'$ within $\log |V|$, we can compute the shortest path from any such v to any terminal in constant time using Lemma 1. On the other hand, notice that for $k \geq 3$ in `Partial`k`Tree`, a descendant of an unmarked intermediate node might also be unmarked, in which case we would need the shortest paths between two unmarked nodes and run `FindBoundedPaths` starting from every new set of intermediate nodes. In our experiments, for speed we restrict ourselves to $k = 2$.

4 Empirical Study

In this section, we evaluate our proposed method; first we give intuitive results on synthetic examples, second we quantitatively compare the performance of the four proposed heuristic methods DOT2DOT-* (COMPONENTS, MINARBORESCENCE, 1-, and 2-LEVELTREE), and finally we provide case studies to show qualitative performance on real-world graphs.

4.1 Synthetic examples

We start by testing our method through three examples on a synthetic 100×100 grid graph, as well as one example on the well-known Zackary's karate graph [31].

First, as shown in Figure 3(a), we select 8 marked (blue square) nodes relatively close to each other on the grid graph, and run all four of our approximation methods. We highlight (by orange bold edges) the minimum description cost tree found (orange nodes: connectors)—notice that it provides succinct connections among the marked nodes. Next we place 4 of the marked nodes farther apart in the grid graph, in which case our method successfully partitions the marked nodes and provides 2 connection trees for each as shown in Figure 3(b).

Second, we place the marked nodes intermittently on the grid as shown in Figure 3(c). Intuitively, human would connect these dots to form a rectangle—and so does our method. Figure 3(d) shows a set of marked nodes forming an almost full rectangle on the grid, except one node in the middle left unmarked. Notice that our method successfully recovers this 'left-behind' node as a significant connector.

Finally, we mark 7 nodes on the Karate graph as depicted in Figure 3(e). Our method partitions the nodes into 3 parts that are well separated through high degree hub nodes.

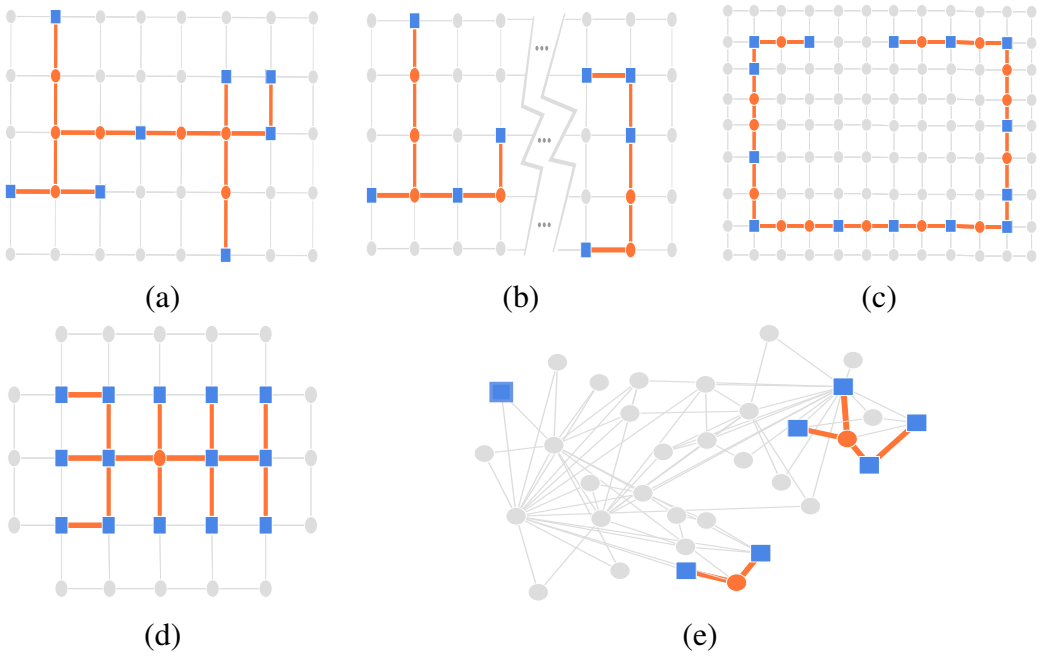


Figure 3: Synthetic examples: (a) 8 marked (square) nodes placed close to each other on a grid, (b) same 8 nodes in (a) spaced out on the grid, forming 2 parts, (c) connecting the dots, (d) recovering missing connector, (e) 7 marked nodes on Karate graph, forming 3 parts.

4.2 Comparing the heuristics

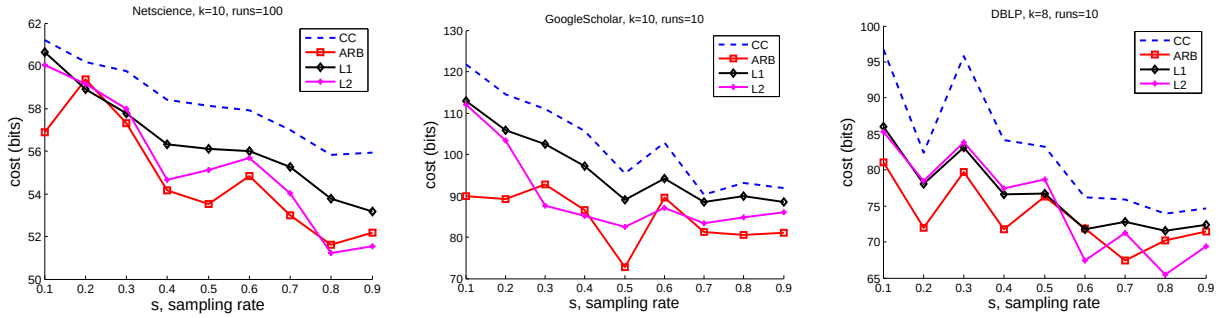
In this section, we aim to understand the average performance of the four approximation methods proposed in Section 3. To do so, we run simulations on three real graphs and compare their average description cost. In each simulated run we test the heuristics on a different set of marked nodes. The set of marked nodes are selected via random walk sampling, which we describe next. The dataset information is given in Table 1.

Name	$\lvert V\rvert$	$\lvert E\rvert$	**Description**
Netscience	379	914	Author collaborations
GScholar	83K	148K	Academic article citations
DBLP	329K	1094K	Author collaborations

Table 1: Dataset summary used for DOT2DOT.

To select a set of k marked nodes, we follow a random walk sampling scheme. First, we fix a sampling rate s. Then, we choose and mark a node at random in the given graph. Next we randomly visit $k' < k$ of its neighbors, and mark each neighbor with probability s. We continue this process until we have k marked nodes in total.

In Figure 4, we show average description cost (in bits) versus sampling rate s=$\{0.1, \ldots, 0.9\}$



(a) *Netscience* (b) *GScholar* (c) *DBLP*

Figure 4: Comparison of our heuristics: total cost (bits) versus various sampling rates s to choose the marked nodes.

of our simulations (k' is set to 3, k'=1, 2 gives similar results). We notice comparable performance results on all three graphs; the simplest heuristic COMPONENTS provides the costliest, while MINARBORESCENCE gives the most succinct description among the four methods on average. In addition, 2-LEVELTREE provides competitive results to MINARBORESCENCE, and outperforms 1-LEVELTREE as would be expected.

In addition, notice the downward trend of the cost for increasing sampling rate. This is expected, as for higher s the marked nodes are chosen among closer nodes. This also explains the relatively larger gap in performance between 2-LEVELTREE and MINARBORESCENCE for small s, as for farther apart nodes a higher than 2 level tree might be required.

Finally, we give the running time of our methods in Figure 5. We notice that due to the iterative nature of looking for good intermediate nodes, k-LEVELTREE heuristics (L1 and L2 respectively) take longer than the others. At the same time, L2 completes in about 50 seconds on our largest graph *DBLP* and can be further sped up by providing it with a smaller candidate graph. Since FINDBOUNDEDPATHS takes the most considerable time in our framework, we propose to run all heuristics and report the best result with the minimum cost.

4.3 Case studies on real graphs

Our method can provide useful insight about the connections and connectors among a group of nodes in a given graph. As such, we develop DOT2DOT as an interactive tool that aids in visualization and sensemaking. In this section, we provide qualitative analysis on two large real-world graphs.

4.3.1 Authors in DBLP

We first employ DOT2DOT among authors from various fields in computer science in the *DBLP* dataset. In particular, we select 2 major conferences in certain fields and mark the top 10 authors from each, who have the most number of papers appearing at that particular conference.

In Figure 6(a) we show the connection tree our method finds among respective authors from VLDB and CHI which are in the fields of databases and human computer interaction, respectively.

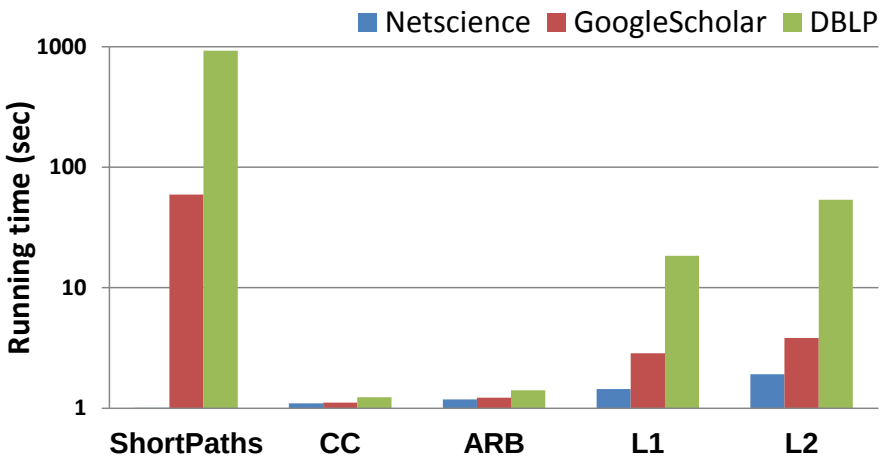


Figure 5: Run time of proposed methods on three real graphs.

Notice that the authors from these two fields are considerably well separated in the tree. We also observe a connector node among the communities: Duen Horng Chau, a PhD student at Carnegie Mellon (also a co-author) whose research focuses on bridging data mining, human computer interaction, and visualization.

We obtain similar results for the authors from RECOMB (computational biology) and KDD (data mining and machine learning) in Figure 6(b). Notice the simple connections among the authors within the same field and otherwise well separated communities. The connector is David Heckerman, the director of the eScience team at Microsoft Research whose work focuses on learning from, and analysis of biological and medical data.

In the next example from *DBLP* we look at authors from NIPS (machine learning) and PODS (database systems), 5 authors from each. DOT2DOT connects the authors from each field through a few connectors as shown in Figure 6(c). Notice there are two parts in our partitioning; which suggests that authors from these two communities are sufficiently apart in the graph.

4.3.2 Articles in GoogleScholar

We next apply DOT2DOT to summarize academic articles on certain topics from *GScholar*. By their titles, we mark nodes in the citation graph containing specified keywords.

In Figure 7(a) we show 8 marked articles containing both 'large graphs' and 'visual' keywords in their title. Our visualization highlights the resulting partitioning on the candidate graph we generate by the union of top-3 shortest paths among the marked nodes. We find that DOT2DOT partitions the marked nodes into 4 parts, 3 of which are singletons. The connection tree for the largest part provides a concise summary for the 5 nodes in this part, with only two connectors. Also notice that the singleton nodes are at least three hops apart from this tree.

In the second example, we mark a random set of the articles with the keywords 'association rule' together with 'visual' and/or 'text' in their title. Here, we expect the articles to naturally split into two groups, one group on visualizing association rules and the other on association rule

mining in text data. Our results, as shown in Figure 7(b), agree with our intuition: visualization articles are linked up through a meaningful connector and 'text' articles form separate multiple parts. It is interesting to observe that they can not be glued together by a single connection tree as in the former scenario.

5 Related Work

One closely related line of work is the connection subgraphs mining problem, introduced by Faloutsos et al. [9]. As they formulated, a connection subgraph is a relatively small, connected subgraph that well captures the relationship (set of interesting paths) between two nodes in a given graph. Later, Ramakrishnan et al. [21] and Sevon et al. [24] extended [9] for querying labeled graphs in which nodes and edges belong to certain types or relations. On similar lines, Jin et al. [14] and Shahaf et al. [25] propose techniques to discover a coherent chain of evidence trails that connect two given concepts in text documents. All these methods mentioned so far, however, are appropriate for only two query nodes; thus, they do not generalize for more than two input nodes.

The Center-Piece Subgraph Problem introduced by Tong et al. [28, 29] finds the most 'centered' node that has strong connections to most or all of the query nodes, and keeps adding important paths from the center-piece node to active nodes until an upper bound on the number of nodes in the output graph is reached. Koren et al. [17] introduce the Cycle-Free Effective Conductance measure for graph proximity and propose methods to extract the subgraph with at most a certain size which retains most of the proximity between query nodes. Proximity and connection subgraphs are also exploited for graph visualization and summarization [6, 7, 15]. For a survey on connection subgraphs, see [1].

Another set of related work focuses on expanding communities around a given set of seed nodes [2, 3, 22, 27]. These methods find a subgraph containing the seed nodes and differ in the specific graph measure like its modularity, conductance, or clustering coefficient that they choose to optimize.

We find it worth emphasizing that graph clustering [10, 11, 16], although related, target a different problem. While their goal is to cluster the *whole* graph, we try to partition and connect a small *subset* of nodes, making use of the graph structure.

Other related methods that operate on subset of input nodes follow. Bhalotia et al. [4] develop methods to find effective connection trees among keywords for browsing and querying. While conceptually similar, our problem definition and formulation have important differences, such as incorporating encoding length. Leskovec et al. [18] use features extracted from query connection subgraphs together with machine learning techniques to predict quality of query results. The connection subgraph is composed of the induced subgraph returned by the query where the its disconnected components are simply joined by shortest paths. Guan et al. [12] study a new measure called structural correlation to assess how strongly a set of query nodes (e.g. nodes affected by an event) are correlated via the graph structure. Their method provides a correlation score but not a connection subgraph. Tong et al. [30] finds the top-k nodes with the highest 'gateway-ness' score with respect to a given source and target node set, such that they collectively lie on most of the

short paths from source nodes to target nodes. Their goal, rather than finding strong connections between query nodes, is finding a group of nodes that can disconnect the source and target nodes to the largest extent in case they are removed from the graph.

Our work differs from all the existing work in two major aspects; (1) we do not assume any specific connection structure among the seed/query nodes (e.g., from-to type connection in [9], the star-shape/centerpiece in [28], a community surrounding seed nodes in [17], etc.); and (2) we employ ideas from information theory to find the optimal partitioning and connections in a principled way, without any user-defined parameters.

6 Concluding Remarks

We propose a novel problem—how to 'describe' a set of marked nodes in a graph. Intuitively, the goal is to find groups of 'close-by' nodes, together with 'simple' connections among them. To the best of our knowledge, this problem has not been studied in the literature; we are the first to give formal problem definitions and initial solutions. Our contributions are summarized as follows.

- *Problem formulation:* We formalize the problem of 'describing a set of chosen nodes in a graph' as an encoding problem. Our formulation exploits the Minimum Description Length principle: the best description is the one for which we need the least number of bits to encode the marked nodes. As such, our method does not require any user-specified parameters such as the number of groups.
- *Fast algorithms:* We show that our problem formulation has connections to the directed Steiner tree problem [5], and that finding the minimum cost (in bits) solution is NP-hard. We propose fast solutions for large graphs.
- *Experiments on real graphs:* Experiments on real academic collaboration and citation as well as synthetic graphs demonstrate the effectiveness of DOT2DOT in discovering good connectors and connections that agree with human intuition.

We note that our problem framework is general and can find applications in numerous settings such as gene pathway discovery, anomaly detection, visualization and sensemaking. As this is a new problem setting, this paper is only a first step—we will focus on alternate solutions for mining good descriptions, as well as generalize our methods to heterogeneous and time-evolving graphs.

References

[1] Nouf M. Kh. Alsudairy, Vijay V. Raghavan, Alaaeldin M. Hafez, and Hassan I Mathkour. Connection subgraphs: A survey. *J. Appl. Sci.*, 11(17):3221–3232, 2011.

[2] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, 2006.

[3] Reid Andersen and Kevin J. Lang. Communities from seed sets. In *WWW*, pages 223–232, 2006.

[4] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[5] Moses Charikar, Chandra Chekuri, To-Yat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. Approximation algorithms for directed steiner problems. In *SODA*, pages 192–200, 1998.

[6] Duen Horng Chau, Christos Faloutsos, Hanghang Tong, Jason I. Hong, Brian Gallagher, and Tina Eliassi-Rad. Graphite: A visual query system for large graphs. In *ICDM Workshops*, pages 963–966, 2008.

[7] Duen Horng Chau, Aniket Kittur, Jason I. Hong, and Christos Faloutsos. Apolo: interactive large graph sensemaking by combining machine learning and visualization. In *KDD*, pages 739–742, 2011.

[8] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[9] Christos Faloutsos, Kevin S. McCurley, and Andrew Tomkins. Fast discovery of connection subgraphs. In *KDD*, 2004.

[10] Gary William Flake, Steve Lawrence, and C. Lee Giles. Efficient identification of web communities. In *KDD*, 2000.

[11] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *PNAS*, 99(12), 2002.

[12] Ziyu Guan, Jian Wu, Qing Zhang, Ambuj Singh, and Xifeng Yan. Assessing and ranking structural correlations in graphs. In *SIGMOD*, pages 937–948, 2011.

[13] Christopher S. Helvig, Gabriel Robins, and Alexander Zelikovsky. An improved approximation scheme for the group steiner problem. *Networks*, 37(1):8–20, 2001.

[14] Wei Jin, Rohini K. Srihari, and Xin Wu. Mining concept associations for knowledge discovery through concept chain queries. In *PAKDD*, pages 555–562, 2007.

[15] José Fernando Rodrigues Jr., Hanghang Tong, Agma J. M. Traina, Christos Faloutsos, and Jure Leskovec. Gmine: A system for scalable, interactive graph visualization and mining. In *VLDB*, pages 1195–1198, 2006.

[16] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proc. of Supercomputing*, pages 1–13, 1998.

[17] Yehuda Koren, Stephen C. North, and Chris Volinsky. Measuring and extracting proximity in networks. In *KDD*, 2006.

[18] Jure Leskovec, Susan Dumais, and Eric Horvitz. Web projections: learning from contextual subgraphs of the web. In *WWW*, pages 471–480, 2007.

[19] Anthony Liekens, Jeroen De Knijf, Walter Daelemans, Bart Goethals, Peter De Rijk, and Jurgen Del Favero. BioGraph: unsupervised biomedical knowledge discovery via automated hypothesis generation. *Genome Biology*, 12(6), 2011.

[20] Vardges Melkonian. New primal-dual algorithms for steiner tree problems. *Computers and Operations Research*, 34:2147–2167, 2007.

[21] Cartic Ramakrishnan, William H. Milnor, Matthew Perry, and Amit P. Sheth. Discovering informative connection subgraphs in multi-relational graphs. *SIGKDD Explor.*, 7(2):56–63, 2005.

[22] Jason Riedy, David A. Bader, Karl Jiang, Pushkar Pande, , and Richa Sharma. Detecting communities from given seeds in social networks. *Technical Report GT-CSE-11-01*, 2011.

[23] J. Rissanen. Modeling by shortest data description. 14(1):465–471, 1978.

[24] Petteri Sevon and Lauri Eronen. Subgraph queries by context-free grammars. *J. Integrative Bioinformatics*, 5(2), 2008.

[25] Dafna Shahaf and Carlos Guestrin. Connecting the dots between news articles. In *KDD*, pages 623–632, 2010.

[26] W Shrum, N. H. Cheek, and S. M. Hunter. Friendship in school: Gender and racial homophily. *Sociology of Education*, 61:227–239, 1988.

[27] Daniel A. Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. *CoRR*, abs/0809.3232, 2008.

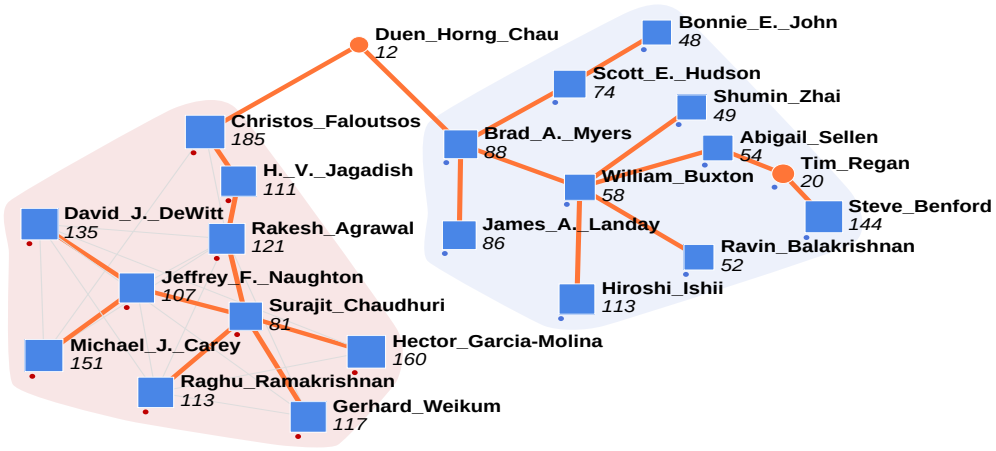
[28] Hanghang Tong and Christos Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, pages 404–413, 2006.

[29] Hanghang Tong, Christos Faloutsos, and Yehuda Koren. Fast direction-aware proximity for graph mining. In *KDD*, pages 747–756, 2007.

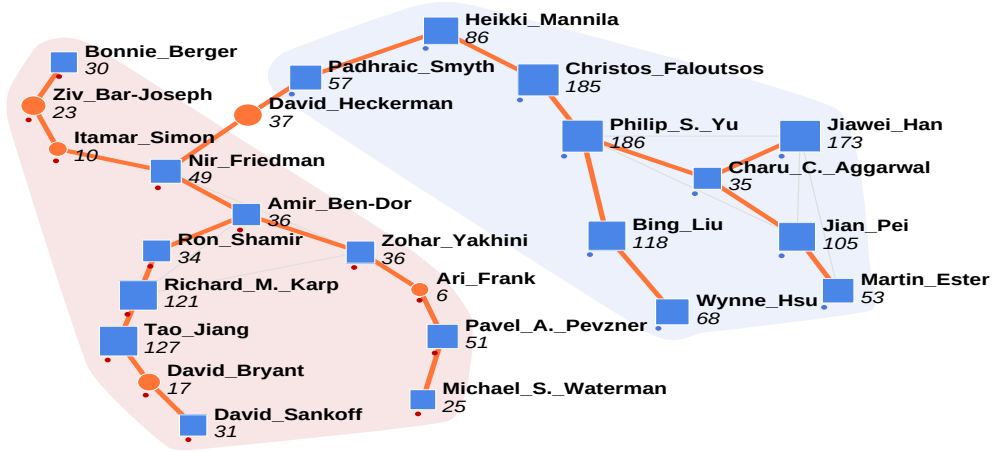
[30] Hanghang Tong, Spiros Papadimitriou, Christos Faloutsos, Philip S. Yu, and Tina Eliassi-Rad. Basset: Scalable gateway finder in large graphs. In *PAKDD*, pages 449–463, 2010.

[31] W.W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33:452–473, 1977.

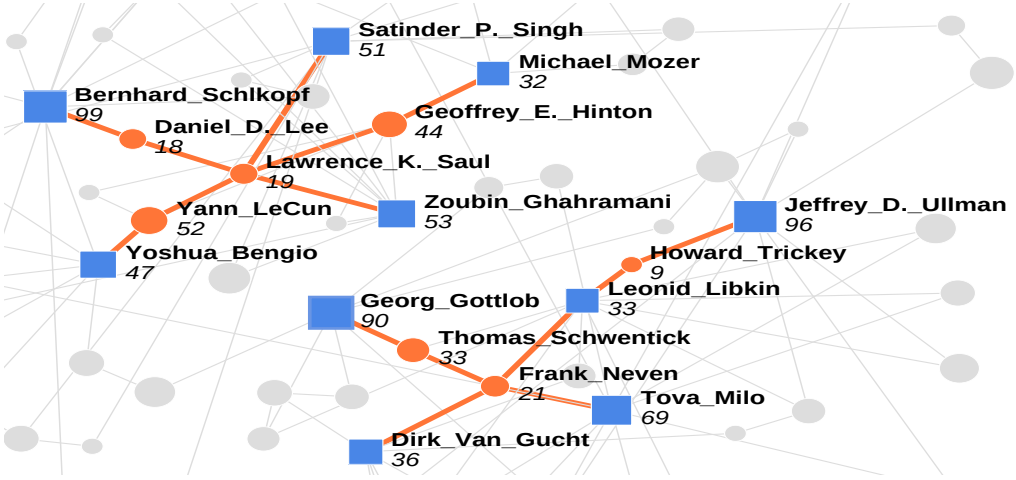
[32] Alexander Zelikovsky. A series of approximation algorithms for the acyclic directed steiner tree problem. *Algorithmica*, 18(1):99–110, 1997.



(a) *DBLP*: VLDB vs. CHI

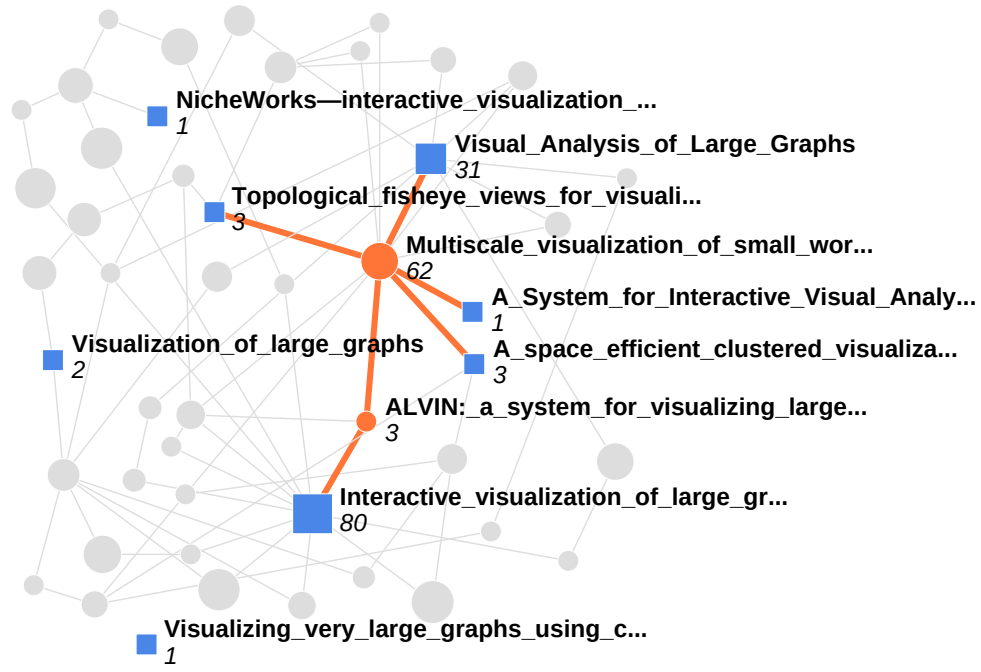


(b) *DBLP*: RECOMB vs. KDD

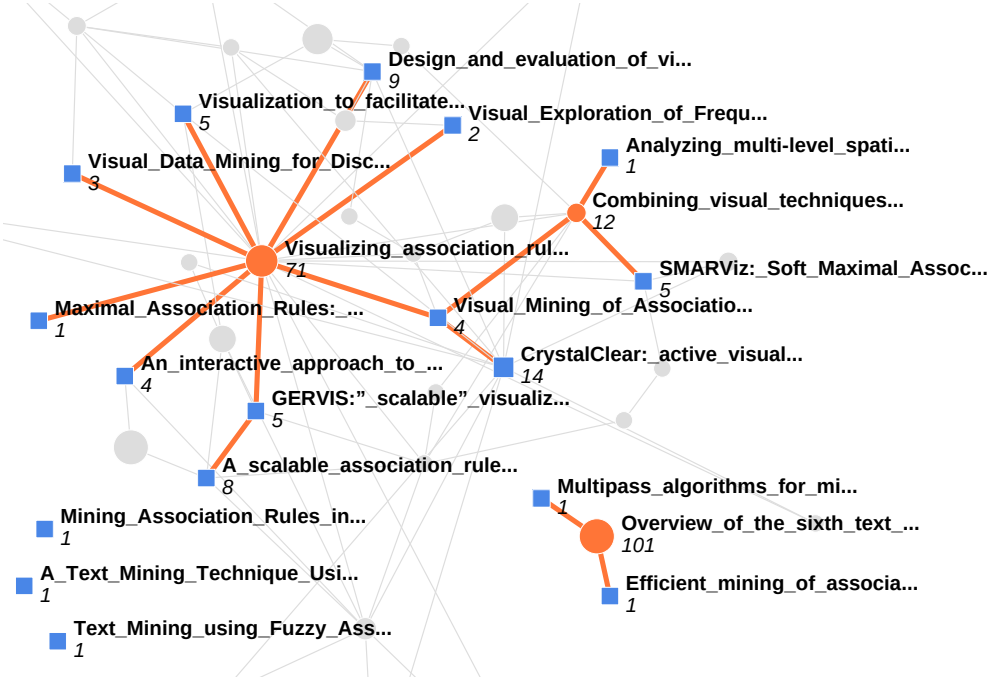


(c) *DBLP*: NIPS vs. PODS

Figure 6: Connection trees among authors from *DBLP* with most number of papers at specified conferences. (a) connector Duen Horng Chau: bridging data mining and human-computer interaction, (b) connector David Heckerman: mining biomedical data, (c) two trees sufficiently apart.



(a) *GScholar*: 'large graphs', 'visual'



(b) *GScholar*: 'association rule', 'visual', 'text'

Figure 7: Connection trees among articles from *GScholar* with the specified keywords in their title. (a) one tree summarizing 5 marked nodes while singletons reside farther apart, (b) one tree summarizing 'visualization' articles while 'text' articles are scattered.