Bump hunting in the dark: Local discrepancy maximization on graphs

Aristides Gionis, Michael Mathioudakis, Antti Ukkonen Helsinki Institute for Information Technology HIIT Aalto University, Finland firstname.lastname@aalto.fi

Abstract—We study the problem of discrepancy maximization on graphs: given a set of nodes Q of an underlying graph G, we aim to identify a connected subgraph of G that contains many more nodes from Q than other nodes. This variant of the discrepancy-maximization problem extends the well-known notion of "bump hunting" in the Euclidean space.

We consider the problem under two access models. In the *unrestricted-access model*, the whole graph G is given as input, while in the *local-access model* we can only retrieve the neighbors of a given node in G using a possibly slow and costly interface.

We prove that the basic problem of discrepancy maximization on graphs is NP-hard, and empirically evaluate the performance of four heuristics for solving it. For the local-access model we consider three different algorithms that aim to recover a part of G large enough to contain an optimal solution, while using only a small number of calls to the neighbor-function interface. We perform a thorough experimental evaluation in order to understand the trade offs between the proposed methods and their dependencies on characteristics of the input graph.

I. INTRODUCTION

Bump hunting is a common approach to extracting insights from data [8]. It consists in looking for regions of a dataset where a property of interest occurs frequently. In this paper, we apply that approach on graphs. A subset of the nodes exhibit a property of interest and are provided as input. We refer to them as *query nodes*. The goal is to find a connected subgraph (the "bump") where query nodes appear more often compared to non-query nodes. We find such a subgraph by maximizing the *linear discrepancy* [6], i.e., the (possibly weighted) difference between the number of query and non-query nodes in the subgraph. Existing literature has addressed linear discrepancy maximization in the Euclidean space, as well as its extensions to non-linear discrepancy functions [1], [2], [6].

Moreover, we consider the problem under a *local-access* model. Specifically, we assume that only query nodes are provided as input, while all other nodes and edges can be discovered only via calls to a costly get-neighbors function from a previously discovered node. For example, accessing the social graph of an online social network (e.g., Twitter) is based on such a function.

We illustrate our setting in more detail with Figure 1. On the left, the figure illustrates the graph, with eight query nodes shown in orange. These query nodes are given as input and are thus considered discovered. Nodes shown in transparent blue are not given as input; they can be discovered via calls to



Figure 1. *Left*: The Karate club graph with 8 query nodes (orange color). *Right*: The subtree formed by including one extra node (purple color) and indicated by thick edges is the maximum discrepancy subgraph that we want to locate given the query nodes.

the get-neighbors function from a previously discovered neighbor. Additionally, none of the edges are discovered at this point. On the right, the figure illustrates the same graph after the execution of our algorithms, and highlights some concepts of our setting: (i) discovered nodes are shown in non-transparent color (orange, purple or blue); (ii) similarly, discovered edges are shown as non-transparent (normal or thick); (iii) the maximum-discrepancy connected component is the subgraph formed by the thick edges; it contains a subset of query nodes $(\{4, 5, 7, 8, 11\})$ and one discovered non-query node $(\{1\})$ shown in purple; (iv) the rest of the query nodes $(\{16, 25, 31\})$ and other discovered nodes (all nodes shown in blue) are not part of the output subgraph-indeed, including any of these three query nodes in the connected subgraph would result in a lower discrepancy, as too many connecting non-query nodes would be needed to maintain connectivity; (v) finally note that discovering the maximum-discrepancy subgraph does not require discovering the whole graph.

To motivate the local-access model, we highlight two application scenarios below.

Scenario 1: Twitter social network. When one submits a text query to Twitter's search engine, Twitter returns a list of messages that match the query, along with the author of each message. For example, when one submits "Ukraine," Twitter returns all recent messages that contain that keyword, together with the users who posted them. We wish to perform the following task: among all Twitter users who posted a relevant



Figure 2. The author-similarity graph contains one node for each author and an edge between two authors if their similarity exceeds a user-defined threshold. Some of the authors (nodes) are part of a query result set. Our goal is to find a connected subgraph that contains many more nodes from the result set than other nodes.

message, find a subset of them that form a local cluster on Twitter's social network. Our goal is to discover a community of users who talk about that topic. Our input consists only of those users who have recently published a relevant message. Note that we do not have the entire social graph; we can only retrieve the social connections of a user by submitting a query to Twitter's API.

Scenario 2: Non-materialized similarity graph (Figure 2). Consider an online library system that allows its users to perform text search on top of a bibliographic database. Upon receiving a query, the system returns a list of authors that match the query. For example, when one submits the keyword "discrepancy," the system returns a list of authors who have published about the topic. We wish to perform the following task: among all authors in this particular result set, identify a subset of similar authors.¹ In this scenario, the underlying graph models the similarity between authors. Nodes represent authors, and edges indicate pairs of authors whose similarity exceeds a user-defined threshold. As the similarity threshold is user-defined, the graph cannot be pre-computed. Moreover, an all-pairs similarity join to materialize the graph at query time is impractical, or even infeasible, to perform. Given an author a, however, it is relatively easy to obtain the set of all other authors whose similarity to a exceeds the similarity threshold. This corresponds to calling the get-neighbors function. We can thus solve the problem without materializing the entire similarity graph. П

To summarize the local-access model, a single call to the get-neighbors function retrieves all neighbors of a given node. The function only returns the neighbor nodes, not any edges that possibly exist between the neighbors; to obtain those, further function calls are needed. Our objective is to devise algorithms that, given as input a set of query nodes, find the maximum-discrepancy connected subgraph using as few calls to the get-neighbors function as possible.

In addition, we also consider the discrepancy-maximization problem under an *unrestricted-access model*. In that model, the algorithm can access nodes and edges of the graph in an arbitrary manner. To the best of our knowledge, the discrepancy-maximization problem has not been considered in graphs even under the unrestricted-access model. We also provide algorithms that *aim to find the maximum-discrepancy connected subgraph in the unrestricted-access model*.

The solution for the local-access model that we propose is based on a two-phase approach. In the first phase, we make calls to the get-neighbors function to retrieve a subgraph. We show that, in many cases, the optimal subgraph can be found without retrieving the entire graph. In the second phase, we can use any algorithm for the unrestricted-access model on the retrieved subgraph.

The rest of the paper is structured as follows. In Section II, we revisit previous work on bump hunting and discrepancy maximization and we position our work relative to it. In Section III, we provide a formal problem definition. Sections IV and V detail our technical contributions:

- We prove that the problem of linear-discrepancy maximization on graphs is **NP**-hard in the general case (Section IV-A). To the best of our knowledge, it is the first time this problem is considered in the literature.
- We prove that the problem has a polynomial-time solution when the graph is a tree, and exploit that observation to define fast heuristic algorithms to produce solutions for the general case (Section IV-A).
- We explain how to tailor the aforementioned algorithms to the local-access model (Section IV-B).
- We compare the performance of the discussed algorithms on real and synthetic data (Section V).

We conclude in section VI with a summary of our findings, and discuss possible directions of future work.

II. RELATED WORK

Bump-hunting. Bump hunting as a general problem has been studied under various names in the literature. Kulldorff [11] phrases the problem as that of finding a single region in a Euclidean space where occurrences of some observations are significantly more frequent than explained by a baseline process. A lot of research has focused on efficient methods for two-dimensional (spatial) input. Dobkin et al. [6], Neill and Moore [12], [13], as well as Agarwal et al. [1], [2] consider fast algorithms for a two-dimensional grid. For high-dimensional data the problem was considered by Friedman and Fisher [8], and later by Neill et al. [14]. Moreover, Wang et al. [19] consider a generalization of scan statistics to graphs, looking for subgraphs with statistically large number of edges. This line of work is also connected to data-mining literature about subgroup discovery [20], contrast set mining [5], and emerging pattern mining [7]. For a review of bump-hunting research we refer the reader to the survey of Novak et al. [15].

Finding interesting subgraphs. The discrepancy maximization problem for graphs is related to works that aim to find interesting subgraphs for a given set of "seed nodes." Along those lines, Andersen and Lang [4] study algorithms that, given a seed set, find a subgraph of small conductance, while

¹Similarity between authors can be defined in a variety of ways, for example as set-of-documents similarity between the papers produced by two authors.

Sozio and Gionis [17] aim to find a subgraph with large minimum degree. Center-piece subgraphs [18] consist of nodes that connect a given (small) set of query nodes well. Somewhat related is work by Akoglu et al. [3] that proposes to find a good connection subgraph for a large set of input points. Perhaps the work conceptually closest to ours is that by Seufert et al. [16], that aims to find cardinality-constrained trees in nodeweighted graphs, with non-negative weights, that maximize the sum of node weights. The problem they consider differs from ours in that the cardinality (size) of the result subgraph is not specified as input in our problem, and that we consider a different weighting scheme for nodes (common positive value for query nodes and common negative value for non-query nodes, rather than individual positive-value weights).

III. SETTING & PROBLEM DEFINITION

The discrepancy function. We consider a graph G = (V, E) with n nodes V and m edges E. Additionally, a set of query nodes $Q \subseteq V$ is provided as input. Regarding terminology, for the purposes of presentation, we will be using the term *component* to refer to any *connected subgraph* of G.²

Let $C = (V_C, E_C)$ be any component (i.e., connected subgraph) of G. Let Q_C be the set of query nodes Q contained in the component C, that is, $Q_C = Q \cap V_C$, and define p_C to be the number of those query nodes, that is, $p_C = |Q_C|$. Similarly define $n_C = |V_C \setminus Q|$ the set of non-query nodes contained in C.

Definition 1: Given a graph G = (V, E) and a component C of G, and given some parameters $\alpha > 0$ and $\beta < 0$, we define the *linear discrepancy* g(C) of C as

$$g(C) = \alpha p_C + \beta n_C$$

Without loss of generality, for the rest of the paper, we fix the value of β to $\beta = -1$ and let the value of $\alpha > 0$ vary. In other words, we define linear discrepancy as

$$g(C) = \alpha p_C - n_C$$

Note that the only requirement we set for a component C is to be connected. Also note that the discrepancy function g(C)takes into account information only about the nodes in C (the number of nodes in C that are query nodes vs. the number of those that are not) and no information regarding the edges in C. Thus, the discrepancy function g(C) is independent of the edge structure of C, except the fact that C is connected. Note also that between components with the same fraction (density) of query nodes, the linear discrepancy will favor the largest component.

In what follows, a component C of the graph G is defined by its set of nodes V_C . In a similar manner, we will be using set notation to denote node-based operations on components. Specifically, for two components C_1 and C_2 , the expression $C_1 \oplus C_2$, where \oplus denotes any set operation \cup, \cap, \setminus , etc., has the following meaning

$$C_1 \oplus C_2 = G(V_C, E_C),$$

such that $V_C = V_{C_1} \oplus V_{C_2}$ and $E_C = \{(u, v) \mid u, v \in V_C\}$. In other words, $C_1 \oplus C_2$ is the *subgraph induced* from the node set $V_{C_1} \oplus V_{C_2}$. Note that, according to our definitions, the subgraph $C_1 \oplus C_2$ is not necessarily a component as it may not be connected.

Having defined the discrepancy function, we are now ready to state the generic problem that we consider in this paper.

Problem I (MAXDISCREPANCY): Given a graph G = (V, E) and a set of query nodes $Q \subseteq V$, find a connected component C of G that maximizes the discrepancy q(C).

We consider solving the MAXDISCREPANCY problem in two different settings, the *local-access model* and the *unrestricted-access model*. An access model here refers to how we are accessing information about the graph G.

Local-access model. In the local-access model we assume that initially only information about the query nodes Q is available. Information about the rest of graph is revealed through calls to a *node-neighbor* function N. In particular, we assume that the graph G is stored in a database, which provides an implementation to a function $N: V \to 2^V$ that takes as input a node $u \in V$, and returns as output the set of all neighbors of u in G, i.e., $N(u) = \{v \in V \mid (u, v) \in E\}$.

In the local-access model we assume that a set of connected components of G is known at any time. Initially, this set of components consists of the query nodes Q as singleton components. At any time instance, we can select one node ufrom the "boundary" of a connected component and issue the query N(u). A node u is considered to be in the boundary of its connected component, if the query N(u) has not being issued before for that node. Once the query N(u) is issued, the neighborhood of u is discovered, and the node u does not belong to the boundary of its component any more. Some of the nodes returned by N(u) may be in the boundary of a connected component, in which case it means that we have discovered new edges and expanded our knowledge of the graph structure. In particular, if a query N(u) returns a node v that belongs in another connected component, we can merge the connected components of u and v.

The cost of an algorithm that operates in the local-access model is the number of times that the algorithm issues a query to function N. For components that have been discovered, we assume that we can apply any process of polynomial-time complexity, and this complexity does not account in cost model of the local-access algorithm. In practice of course, we may want to restrict the complexity of the computation that we can perform for discovered components, for instance, in linear, $n \log n$, or at most quadratic.

Unrestricted-access model. The unrestricted-access model is the standard computational model, in which the graph G is given as input, and the cost model accounts for all operations. Note that the model allows that only a part of the whole underlying graph is known. However, computation is performed only on the known part of the graph, there is no exploration phase to discover new parts of the graph.

IV. Algorithms

In this section, we first establish the complexity of MAX-DISCREPANCY and then present our algorithms. We start our

²Note that our usage of the term 'component' deviates from its usual definition as *maximal* connected subgraph.



Figure 3. Illustration of the transformation used in the proof of Proposition 1

discussion with the unrestricted-access model, since it is the more standard and familiar setting.

A. Unrestricted-access model

Problem complexity. It can be shown that the MAX-DISCREPANCY problem in the unrestricted-access model is **NP**-hard.

Proposition 1: The MAXDISCREPANCY problem is **NP**-hard in the unrestricted-access model.

Proof: We provide a transformation of the SETCOVER problem to the MAXDISCREPANCY problem. Recall that an instance of the SETCOVER problem is specified by a ground set $U = \{u_1, \ldots, u_n\}$ of n elements, a collection $\mathcal{C} = \{S_1, \ldots, S_m\}$ of m subsets of U, and an integer k, and the decision question is whether there are at most k sets in \mathcal{C} whose union contains all the elements in the ground set U.

Given an instance of SETCOVER, we create an instance of the MAXDISCREPANCY problem, as follows. We create a graph G with n + m + 1 nodes, in particular, we create one node for each element u_i , one node for each set S_j , and one additional node u_{n+1} . We then create an edge (u_i, S_j) if and only if $u_i \in S_j$, and m additional edges (S_j, u_{n+1}) , for all $j = 1, \ldots, m$. The set of queries nodes in G is defined to be $Q = \{u_1, \ldots, u_n, u_{n+1}\}$. The construction is illustrated in Figure 3.

We then set $\alpha = k$ and we ask whether there is a component C of G that has discrepancy $g(C) \ge nk$. We can show that the answer to the latter question is affirmative if and only if the given SETCOVER instance has a solution.

To verify our claim observe the following facts.

- 1. Any connected subgraph C of G with at least two nodes needs to contain at least one non-query node.
- 2. Any connected component C of G with $g(C) \ge nk$ needs to contain all n + 1 query nodes. Indeed, the maximum discrepancy of any connected component C' of G with nor less query nodes will be $g(C') \le nk - 1 < nk$ (the '-1' follows from the fact that C' should contain at least one non-query node).
- 3. Any connected component C of G with $g(C) \ge nk$ needs to contain at most k non-query nodes. Indeed, the discrepancy of any component C that contains all n + 1query nodes and ℓ non-query nodes is $g(C) = (n+1)k - \ell$. Requiring $(n+1)k - \ell \ge nk$ gives $\ell \le k$.

From the above three observations it follows that any connected component C of G that has discrepancy $g(C) \ge nk$ should contain all query nodes and at most k non-query nodes. It is easy to see that such a component C corresponds to a solution to the SETCOVER problem.

Conversely, it is easy to see that a solution to the SETCOVER problem corresponds to a connected component C of G with discrepancy g(C) = nk.

Connection to Steiner trees. Even though we obtained the hardness proof via a transformation from the SETCOVER problem, it turns out that MAXDISCREPANCY problem is also related to the *prize-collecting Steiner-tree* problem (PCST). This is an interesting connection, because it can guide the algorithmic design for the MAXDISCREPANCY problem.

The PCST problem, in the general case, is defined as follows. We are given a graph G = (V, E, d), where $d : E \to \mathbb{R}$ is a distance function on the edges of G. We are also given a set of terminal nodes $S \subseteq V$ and a weight function $w : S \to \mathbb{R}$ that assigns positive weights on the terminals. The goal is to find a *Steiner tree* T in G, so as to minimize the objective

$$D(T) + \sum_{u \in S \setminus T} w(u), \tag{1}$$

where D(T) is the sum of distances of all the edges in the tree T. The term "prize collecting" conveys the intuition that the weights on the nodes of the graph represent prizes to be collected and the goal is to find a tree that minimizes the tree cost and the total value of prizes not collected.

It is not difficult to see that the MAXDISCREPANCY problem is a special instance of the PCST problem: Let $C = (V_C, E_C)$ be a component in the MAXDISCREPANCY problem, given an input graph G and query nodes Q. The discrepancy on C is

$$\begin{array}{lll} g(C) &=& \alpha |Q \cap V_C| - |V_C \setminus Q| \\ &=& (\alpha + 1)|Q \cap V_C| - (|Q \cap V_C| + |V_C \setminus Q|) \\ &=& (\alpha + 1)|Q \cap V_C| - |V_C|. \end{array}$$

Maximizing the latter expression is equivalent to minimizing

$$(\alpha + 1)|Q| - 1 - g(C) = (\alpha + 1)|Q \setminus V_C| + |V_C| - 1;$$

since the term $(\alpha + 1)|Q| - 1$ is a constant.

The term $(\alpha + 1)|Q \setminus V_C|$ can be interpreted as the total weight of query nodes not covered by C, assuming that each query node has weight $(\alpha + 1)$, while the term $|V_C| - 1$ can be interpreted as the sum of edges of any tree spanning C, assuming that the all edges have distance 1.

Thus, the component C that maximizes discrepancy in G with query nodes Q, is the optimal tree in a PCST instance where the terminal nodes are the query nodes, all terminal nodes have weight $(\alpha + 1)$, and all edges have distance 1.

The PCST problem is also **NP**-hard, however, it can be approximated within a constant-factor. In particular, Goemans and Williamson have designed a $(2 - \frac{1}{n-1})$ -approximation algorithm where n is the number of nodes in the input graph [9]. The algorithm of Goemans and Williamson relies on the primal-dual schema.

Although an optimal solution for the MAXDISCREPANCY problem corresponds to an optimal solution for the PCST problem, since our mapping involves subtracting the objective functions from a constant, it follows that the approximation guarantee for the PCST problem does not carry over to MAXDISCREPANCY. Nevertheless, the primal-dual algorithm of Goemans and Williamson is an intuitive algorithm for MAXDISCREPANCY, too, and we employ it as a heuristic for this problem. Additionally, we can show that in the special that the graph G is a tree, the MAXDISCREPANCY problem can be solved optimally in linear time. This is discussed next.

Optimal algorithm for trees. When the graph G is a tree, we can solve the MAXDISCREPANCY problem optimally, in linear time $\mathcal{O}(|G|)$, using dynamic programming. The algorithm, named TreeOptimal, is shown as Algorithm 1. Note that any connected component of a tree is also a tree. TreeOptimal exploits the following optimal substructure of the problem: let r_G be a node of G, arbitrarily selected as root, and T_1, \ldots, T_h be the sub-trees below r_G , each rooted at a different node r_1, \ldots, r_h . For any tree T with root r, let opt(T) be the discrepancy of an optimal solution of MAXDISCREPANCY on T, and let con(r, T) be the maximum discrepancy of any component of T that contains the root r. Then, for graph G and its root r_G we have

$$con(r,T) = g(r_G) + \sum_{con(r_i,T_i)>0} con(r_i,T_i),$$
(2)

and

$$\operatorname{opt}(T) = \max\left\{\operatorname{con}(r, T), \max_{i=1,\dots,h} \{\operatorname{opt}(T_i)\}\right\}, \quad (3)$$

where $g(r_G) = g(1,0) = \alpha$ if $r_G \in Q$ and $g(r_G) = g(0,1) = -1$ otherwise. Equation (2) expresses the fact that among all the connected components of G that include its root r_G , the component that includes all the sub-trees of r_G that have positive discrepancy $con(r_i, T_i)$, is the one that maximizes the discrepancy. Equation (3) expresses the fact that the optimal solution of MAXDISCREPANCY either includes the root r_G or is entirely included in one of the sub-trees T_1, \ldots, T_h of root r_G . TreeOptimal returns both values con(r, T) and opt(T) when applied on a tree T, and the optimal discrepancy for G is opt(G), and the values are computed recursively, as specified in Algorithm 1.

Heuristics for the general graph case. Next we discuss the heuristics for the general case, when the graph G is not necessarily a tree. As mentioned above, in this general case, the MAXDISCREPANCY problem in the unrestrictedaccess model is NP-hard. All heuristics aim at first finding a subtree of the input graph, and then applying the TreeOptimal algorithm described before. We study the following heuristics: (*i*) BFS-trees from each query node, (*ii*) minimum weight spanning tree where edge weights are assigned according at random, (*iii*) minimum weight spanning tree where edge weights are assigned according to a simple heuristic based on their endpoints, and (*iv*) the Primal-Dual algorithm for PCST. They are described in detail below.

Algorithm 1 TreeOptimal

Breadth-first search trees (BFS-Tree): A very simple way to obtain trees for a given graph and a set of query nodes Q is to perform breadth-first search (BFS) from every node $u \in Q$. The BFS-Tree heuristic follows exactly this strategy. It computes all BFS trees, one for each query node, it computes the maximum discrepancy solution for each tree, using the TreeOptimal algorithm, and it returns the best solution.

Random spanning tree (Random-ST): Instead of computing BFS from every query node, we can work with a *random* tree that spans the query nodes. We sample such a random tree, by assigning a random weight (uniformly from [0, 1]) to every edge, and computing the minimum weight spanning tree. The Random-ST heuristic works by computing a number of such random spanning trees, computing the maximum discrepancy solution for each tree, using the TreeOptimal algorithm, and returning the best solution found.

Smart spanning tree (Smart-ST): The previous two heuristics run TreeOptimal possibly hundreds of times. A more efficient method is to first find a good tree, and run the TreeOptimal algorithm once on this tree. Intuitively a tree is good if the connectivity between the query nodes is maintained well. That is, if the distance between two query nodes is low in the graph, their distance in the tree should be low as well. A simple heuristic to achieve this is to systematically assign weights to the edges so that the minimum spanning tree avoids edges that are not adjacent to at least one query node. More formally, we assign every edge (u, v) the weight

$$w(u, v) = 2 - I\{u \in Q\} - I\{v \in Q\},$$
(4)

where $I\{\cdot\}$ is the indicator function. The Smart-ST heuristic works by first assigning the edge weights according to Equation 4, finding the minimum weight spanning tree, and finally computing the optimal solution from the tree using TreeOptimal.

Prize-collecting Steiner-tree heuristic (PCST-Tree): As discussed above, the MAXDISCREPANCY problem can be viewed as the prize-collecting Steiner-tree problem (PCST). We convert an instance of MAXDISCREPANCY to a PCST instance by

letting w(u, v) = 1 for every edge (u, v), and setting the cost of a query node node (w(u)) in Equation 1) to $\alpha + 1$, and the cost of every other node to 0. An optimal Steiner tree for this PCST instance will also have maximum discrepancy as measured by the function g.

The PCST-Tree heuristic first does the above conversion, then uses the Goemans-Williamson approximation algorithm for PCST [9] to compute a forest of disjoint trees. Then, for every tree in the resulting forest, the heuristic runs TreeOptimal, and it returns the best solution that it finds.

Note that the factor-2 approximation guarantee for PCST does not translate into a constant factor approximation for MAXDISCREPANCY. However, since there is a direct correspondence between the solutions of the two problems, we opt to use this algorithm as a reasonable heuristic.

B. Local-access model

Having discussed the complexity of the problem and presented heuristic algorithms to solve it under the unrestrictedaccess model, we now turn our focus to the *local-access model*. Under the local-access model, our input consists only of the query nodes Q, while the the rest of graph G = (V, E) is accessible through a node-neighbor function N. The function N takes as argument one node and returns the list of its neighbors in the graph G. Unlike in the unrestricted-access model, we can now access the edges of G only through the node-neighbor function.

To solve the problem under the local-access model, a bruteforce approach would be to invoke the function N repeatedly, until we retrieve the entire graph G, i.e., first invoke function N to retrieve the neighbors of nodes in Q, then invoke N to retrieve the neighbors of neighbors, and so on, until we retrieve the entire G; and then apply on G the algorithms from the unrestricted-access model.

In many settings however, as we discussed in our introduction, invoking the function N can be slow and costly. Moreover, having access to the entire graph is not necessary as long as we have access to *a subgraph that contains the optimal solution*. Ideally, we should be able to solve MAX-DISCREPANCY even over an *infinite* graph G, as long as the set of query nodes Q is finite. We are thus interested in limiting the number of invocations of function N, retrieving only a small part of graph G that contains the optimal solution, and solving MAXDISCREPANCY on that, by employing one of the algorithms from the unrestricted-access model.

Specifically, to solve MAXDISCREPANCY, we first invoke function N a number of times to retrieve a subgraph G_x of G, and then, as a second step, we apply one of the aforementioned heuristics for the unrestricted-access model. We refer to the first step of our approach as the "expansion" step, since it builds G_x by expanding the neighborhood of nodes Q through invocations of the function N. Obviously, for the algorithm that implements the expansion step it is desirable that it returns a subgraph G_x that contains the optimal solution, and that it invokes the function N only a small number of times.

We now discuss three algorithms that implement the expansion step: FullExpansion, ObliviousExpansion, and Adaptive-Expansion. All three algorithms build the graph G_x iteratively: at each iteration, they invoke the function N on some or all nodes of G_x on which N was not invoked before.

Full expansion. Our first expansion strategy, named Full-Expansion and shown in Algorithm 2, is a conservative strategy that is guaranteed to return a subgraph G_x of G that contains the optimal solution. It constructs one or more components, the sum of the diameters of which is $\mathcal{O}(|Q|)$.³ The algorithm builds the subgraph G_x iteratively; it starts by retrieving the neighbors of nodes Q, then the neighbors of neighbors and so on, until the expansion has gone far enough from all query nodes to be certain the optimal solution is contained within one of the connected components of the expanded graph.

In more detail, among all nodes it has retrieved after each iteration, FullExpansion distinguishes one subset of nodes as the *Frontier* nodes, i.e., the nodes that should be expanded in the next iteration. If c is one of the retrieved nodes that has not been expanded yet at the end of one iteration, and Q_c are the query nodes reachable from c in G_x , then c becomes a Frontier node if the following condition holds:

$$\min_{q \in Q_c} \left\{ d(c,q) \right\} \le |Q_c| \cdot (\alpha+1), \tag{5}$$

where d(c,q) refers to the number of hops between c and q in the graph G_x . The algorithm terminates when the set of Frontier nodes is empty, i.e., when the condition (5) does not hold for any node that has been retrieved (i.e., a node of G_x) but has not been expanded yet.

According to Lemma 2, which we will formulate and prove below, termination according to condition (5) is sufficient to guarantee returning a graph G_x that contains an optimal solution. The proof for Lemma 2 uses the following auxiliary result.

Lemma 1: Let OPT be a solution to MAXDISCREPANCY, and denote by p_{OPT} the number of query nodes in OPT, that is, $p_{\text{OPT}} = |\text{OPT} \cap Q|$. Then,

$$|\text{OPT}| \le (\alpha + 1) \cdot p_{\text{OPT}} - \alpha.$$

Proof: We have that $|OPT| - p_{OPT}|$ is the number of non-query nodes in OPT. Moreover, it is easy to see that the discrepancy of the optimal solution OPT has to be larger or equal to the discrepancy of a component that consists only of one query node. Therefore, by substituting these into the linear discrepancy function, we get

$$\alpha \cdot p_{\text{OPT}} - (|\text{OPT}| - p_{\text{OPT}}) \ge \alpha \cdot 1 - 0,$$

from which we obtain $|OPT| \le (\alpha + 1) \cdot p_{OPT} - \alpha$.

Lemma 2: Let G_x be the graph returned by FullExpansion. Then, one of the connected components of G_x contains the optimal solution to MAXDISCREPANCY as its subgraph.

Proof: For the sake of contradiction, let us assume that FullExpansion returns a graph G_x that consists of disjoint connected components C_1, \ldots, C_k , for some $k \ge 1$, none

 $^{^{3}}$ The diameter of a connected component is the maximum distance between any two of its nodes.

Algorithm 2 FullExpansion

of which fully contains an optimal solution OPT; that is, OPT $\not\subseteq C_i, i = 1, ..., k$.

We know, however, that at least one of the components of G_x overlaps with the optimal solution, since an optimal solution has to contain at least one query node, and all query nodes are contained in G_x . That is, with $\hat{p}_i = |Q \cap C_i \cap \text{OPT}|$, there should exist an $i \in \{1, 2, ..., k\}$ such that $\hat{p}_i > 0$.

We will reach a contradiction with Lemma 1 by showing

$$|OPT| \ge \sum_{C_i: \hat{p}_i > 0} |C_i \cap OPT| > (\alpha + 1) \cdot p_{OPT} - \alpha.$$
(6)

The first inequality of (6) follows immediately from basic set properties. To show the second inequality, we first show that

$$|C_i \cap OPT| > d(c,q) > \hat{p}_i(\alpha+1), \tag{7}$$

where $\hat{p}_i > 0$ and d(c,q) is the distance between any $q \in Q \cap C_i \cap OPT$ and any node $c \in C_i \cap OPT$ that remains unexpanded after the termination of Algorithm 2 (i.e., $c \in C_i \cap OPT$ is a node that was added to C_i but did not satisfy condition (5) to be expanded). The first inequality of (7) follows from the fact that both c and q are nodes of $C_i \cap OPT$; therefore, if the distance between them is d(c,q), then the node-size of the subgraph $C_i \cap OPT$ that contains them has to be larger than d(c,q). The second inequality of (7) follows from the stopping condition of FullExpansion: it terminates when there is no node in G_x that satisfies condition (5). Therefore, for any $q' \in Q \cap C_i$ and any node c' that belongs to C_i but was not expanded by FullExpansion, we have

$$d(c',q') > |Q \cap C_i| \cdot (\alpha+1) \ge |Q \cap C_i \cap OPT| \cdot (\alpha+1)$$

and consequently, since $\hat{p}_i = |Q \cap C_i \cap OPT|$,

$$d(c',q') > \hat{p}_i \cdot (\alpha+1). \tag{8}$$

Therefore, since $q \in Q \cap C_i \cap OPT \subseteq Q \cap C_i$ and $c \in C_i \cap OPT \subseteq C_i$, inequality (8) holds for q and c as well:

$$d(c,q) > \hat{p}_i \cdot (\alpha+1). \tag{9}$$

Having proved (7), we take the sum over all C_i 's with $\hat{p}_i > 0$, to get

$$\sum_{C_i:\hat{p}_i>0} |C_i \cap \operatorname{OPT}| > \sum_{C_i:\hat{p}_i>0} \hat{p}_i(\alpha+1) = (\alpha+1) \sum_{C_i:\hat{p}_i>0} \hat{p}_i$$



Figure 4. The example shows two problem instances with $\alpha = 1$. In case (a), FullExpansion and ObliviousExpansion behave identically, as they retrieve the same set of nodes before they terminate. In case (b), FullExpansion expands more nodes than ObliviousExpansion and uncovers a solution of higher discrepancy.

Algorithm 3 ObliviousExpansion
Input: Query nodes Q
$G_{_X} := (Q, \emptyset)$
for $i = 1$ to $(1 + \alpha)$ do
Expand all unexpanded nodes in G_x
Add their neighbors and edges to G_{X}
return G _x

$$= (\alpha + 1) \cdot p_{\text{OPT}} > (\alpha + 1) \cdot p_{\text{OPT}} - \alpha.$$

We have now proved the second inequality of (6), thus completing the proof of the lemma.

While FullExpansion guarantees to return a graph G_x that contains the optimal solution, we found that in practice it is extremely inefficient. This is because the sum of the diameters of all the connected components of graph G_x can grow up to $\mathcal{O}(|Q|)$, which means that, even for moderately dense graphs and query sets Q of moderate size, FullExpansion eventually retrieves the entire graph G.

To alleviate this problem, we propose two improved expansion strategies, ObliviousExpansion and AdaptiveExpansion.

Oblivious expansion. This expansion strategy simply performs $(1+\alpha)$ expansion iterations from query nodes Q before it terminates. ObliviousExpansion is outlined as Algorithm 3. To compare its behavior with FullExpansion, let us consider two different cases for graph G, as depicted in Figure 4.

In Figure 4(a), the graph G is a linear graph, where nodes Q fall far from each other (the distance between two consecutive query nodes is larger than $2(1+\alpha)$). In that case, FullExpansion and ObliviousExpansion behave identically: they expand by $(1+\alpha)$ from each query node and stop, with a retrieved graph G_x that consists of |Q| distinct connected components, one for each query node.

In Figure 4(b), on the other hand, the graph G is a linear graph again, however query nodes $C_2 \subseteq Q$ are clustered in tightly in two areas of the graph. They are separated by l non-

Algorithm 4 AdaptiveExpansion

Input: Query nodes Q $G_x \leftarrow (Q, \emptyset)$ Expanded $\leftarrow \emptyset$ Frontier $\leftarrow Q$ while Frontier $\neq \emptyset$ and # components of $G_x > 1$ do NewFrontier $\leftarrow \emptyset$ for connected component C of G_x do f = random node from Frontier(C)for n in N(f) do Add edge (f, n) to G_X If $n \notin Expanded$ Then add n to NewFrontier Expanded \leftarrow Expanded \cup Frontier Frontier \leftarrow NewFrontier if Time to update solution quality estimate then Calculate $g_{\scriptscriptstyle\rm LB}$ and $g_{\scriptscriptstyle\rm UB}$ if $g_{\text{LB}}/g_{\text{UB}} \ge 1$ then return G_x return G_x

query nodes, where

$$2(1+\alpha) < l = |C_1| \cdot \alpha + |C_2| \cdot \alpha.$$

In that case, ObliviousExpansion will only expand $(1+\alpha)$ nodes from each query node (thick-ringed nodes in Figure 4(b)), while FullExpansion will expand far enough to retrieve a connected component that includes all query nodes (gray-filled nodes in Figure 4(b)) and has higher discrepancy than C_1 or C_2 alone, as it is easy to check.

Adaptive expansion, shown as Algorithm 4, takes a different approach than the previous two expansion algorithms. The main differences are the following:

- in each iteration, AdaptiveExpansion randomly selects a small number $\mathcal{O}(|Q|)$ of not-yet-expanded nodes to expand; and
- the termination condition of AdaptiveExpansion depends on a heuristic estimate of its approximation ratio (i.e., how close is the current optimal solution on G_x with respect to the optimal solution on G).

Unlike the previous two expansion algorithms that might invoke the node-neighbor function N on all not-yet-expanded nodes at each iteration, AdaptiveExpansion is more frugal in invoking the function N. The rationale for this approach is that for densely connected graphs, as real networks usually are, a small number of edges is enough to preserve the connectivity of a connected graph. In such settings, therefore, it is possible for AdaptiveExpansion to uncover the nodes of a large and densely connected component of G that contains many of the query nodes Q. The advantage of this approach is that Algorithm 4 can quickly uncover a graph G_x that provides a solution that is close to optimal.

At the core of the AdaptiveExpansion algorithm is a stopping condition that allows it to avoid unnecessary expansions. To decide whether expansion should be terminated, Adaptive-Expansion periodically computes an upper bound $g_{\rm UB}$ to the



O unexpanded node on the periphery of component

Figure 5. To calculate $g_{\rm LB}$ and $g_{\rm UB}$, AdaptiveExpansion builds BFS trees for each connected component of G_X . The maximum discrepancy on any of those trees is used as $g_{\rm LB}$. To calculate $g_{\rm UB}$, it sums the positive discrepancies of BFS trees that include at least one node on the periphery of the connected components.

optimal discrepancy g(OPT), as well as a lower bound $g_{\rm LB}$ of the discrepancy of the optimal MAXDISCREPANCY solution on G_x . Computing these estimates can be computationally involved, therefore the algorithm does not update them after every expansion, but at predefined intervals.

Specifically, at the end of such an interval, Adaptive-Expansion selects randomly k not-yet-expanded nodes r_1, \ldots, r_k from each component of G_x , for some small k specified as a parameter of the algorithm.⁴ Subsequently, it computes k BFS trees with each r_i as its root, and computes the discrepancy of these trees using the TreeOptimal algorithm. The lower bound $g_{\rm LB}$ is the maximum discrepancy found in these BFS trees.

Regarding $g_{\rm UB}$, it is computed as follows: for the unexpanded nodes r_i and the corresponding BFS trees that provided the best discrepancy for each component of G_x , AdaptiveExpansion maintains the discrepancy d_r of the best solutions that include r. For the components that have $d_r > 0$, the algorithm computes the sum $s = \sum d_r$ and estimates $g_{\rm UB}$ as $g_{\rm UB} = \max\{g_{\rm LB}, s\}$.

The rationale is that, as nodes r have not been expanded yet, it is possible they are connected on G. If some of them are connected, then it is possible to have a solution of discrepancy $s = \sum d_r; d_r > 0$, as described above. The approach is illustrated in figure 5.

The algorithm terminates when $g_{\rm \tiny UB} \leq g_{\rm \tiny LB}$.

V. EXPERIMENTS

In this section, we present the results from an experimental evaluation on both synthetic and real-world graphs. The purpose of our experiments is to study and compare the performance of the different expansion strategies, as well as the algorithms that solve the MAXDISCREPANCY problem under the unrestricted-access model. The code and data used for our experiments are publicly available.⁵

⁴In all our experiments, we used k = 5, as we observed well behaving estimates for that value of k.

⁵http://research.ics.aalto.fi/dmg/software.shtml

Table I. DATASET STATISTICS (NUMBERS ARE ROUNDED).

Dataset	V	E
Geo	$1 \cdot 10^{6}$	$4 \cdot 10^6$
BA	$1 \cdot 10^{6}$	$10 \cdot 10^6$
Grid	$4 \cdot 10^{6}$	$8 \cdot 10^6$
Livejournal	$4.3 \cdot 10^{6}$	$69 \cdot 10^{6}$
Patents	$2 \cdot 10^6$	$16.5 \cdot 10^{6}$
Pokec	$1.4 \cdot 10^{6}$	$30.6 \cdot 10^{6}$

A. Datasets

We use three synthetic graphs (Grid, Geo, and BA) and three real-world graphs (Livejournal, Pokec, Patents). All graphs used in the experiments are undirected and their sizes are reported in Table I.

Grid is a simple $2M \times 2M$ grid, in which most nodes (all other than the ones on the periphery of the grid) have degree equal to four (4). Geo is a geographical near-neighbor network: It is generated by selecting 1M random points in the unit square in \mathbb{R}^2 , and then connecting as neighbors all pairs of points whose distance is at most 0.0016 from each other, yielding an average degree of ≈ 8 . BA is a random graph generated by the Barabási-Albert model, with parameters n = 1M, and m = 10.

Livejournal, Pokec, and Patents are all real-world graphs obtained from the Stanford Large Network Dataset Collection.⁶ Livejournal and Pokec are extracted from the corresponding online social networks, while Patents is a citation network.

B. Evaluation methodology

We now describe our evaluation framework. One experiment in our evaluation framework is defined by (1) a graph G, given as input to the problem, (2) a set of query nodes Q, given as input to the problem, (3) an expansion algorithm, to invoke API function N and expand Q to G_x , and (4) a MAXDISCREPANCY algorithm, to solve the problem on G_x in the unrestricted-access model.

Specifically, the graph is always one of the datasets described in Section V-A. The expansion algorithm is either ObliviousExpansion or AdaptiveExpansion, both described in Section IV-B. Results from FullExpansion are not reported here, as it proved impractical for larger datasets. The algorithm to solve MAXDISCREPANCY is one of BFS-Tree, Random-ST, PCST-Tree, and Smart-ST, described in Section IV-A. And lastly, query nodes Q are selected randomly, with the process described next.

Query nodes Q are generated as follows. As a first step, we select one node c from graph G, uniformly at random. As a second step, we select a sphere $S(c, \rho)$ of predetermined radius ρ , with c as a center. As a third step, from sphere $S(c, \rho)$ we select a set of query nodes Q_s of predetermined size s. Selection is done uniformly at random. Finally, we select a predetermined number of z random query nodes from outside all spheres. To generate Q, we set varying values to:

• the number k of spheres $S(c, \rho)$,

- the radius ρ of spheres,
- the number of query nodes $s = |Q_s|$ in each sphere $S(c, \rho)$,
- the number of query nodes z outside all spheres.

Note that, while generating Q, we make sure that the randomly selected sphere $S(c, \rho)$ is large enough to accommodate s query nodes; if this is not the case, then we repeat the random selection until we obtain a sphere with more than s nodes.

We create experiments with all possible combinations of graphs, expansion algorithms, and MAXDISCREPANCY algorithms, and for each combination we create 20 different instances, each with a different random set of query nodes. For each experiment, we measure the following quantities: (1) number of API calls to expand G into G_x , (2) size of G_x as number of edges, (3) discrepancy of solution, (4) accuracy of solution, (5) running time of MAXDISCREPANCY algorithm.

The number of API calls, as well as the size of G_x are used to compare expansion algorithms: the first measure is of obvious interest under the local-access model, while the second one influences the running time of MAXDISCREPANCY algorithms. The rest of the measures are used to compare the performance of MAXDISCREPANCY algorithms. Discrepancy and running time measure the quality of the solution and the efficiency of algorithms. Accuracy is defined as the Jaccard coefficient between query nodes in the returned solution, and the best matching sphere $S(c, \rho)$ in the planted query nodes Q. All quantities are measured as averages over all experiment

instances with the same parameters.

We also note that for all the experiments reported, the value of parameter α of the discrepancy function g is set to $\alpha = 1$. As per Section III, α can be set to any positive value, and thus account for different weighting between query and nonquery nodes. For our experiments, in absence of a particular weighting preference, and due to space constraints, we present results only for $\alpha = 1$.

Implementation: All algorithms are implemented in Python 2.7 and each individual experiment was run on a dedicated Intel Xeon 2.83 GHz processor, on a 32 GB machine. Each graph G is stored in a separate MongoDB collection.⁷ Each document in the collection stores the adjacency list of one node in the form

with node_id indexed as a key of the collection. One invocation of the API function N then, corresponds to the selection of one document with a specified node_id and the projection of the associated adjacency list [neighbor_id,...].

To make the experiments run in a reasonable amount of time, we gave the MAXDISCREPANCY algorithms 5 minutes to terminate their run in a single experiment. If they failed to produce a solution in 5 minutes, the process was killed and the evaluation moved on to the next experiment.

C. Results: expansion algorithms

To compare ObliviousExpansion and AdaptiveExpansion, we ran a large number of experiments with different parameters

⁶http://snap.stanford.edu/

⁷http://www.mongodb.org

Table II. EXPANSION TABLE (AVERAGES OF 20 RUNS)

			Oblivi	ousExpansion	Adapti	veExpansion
dataset	s	$_{k}$	cost	size	cost	size
Grid	20	2	302	888	2783	7950
Grid	60	1	261	784	534	1604
Geo	20	2	452	2578	4833	30883
Geo	60	1	418	2452	578	3991
BA	20	2	3943	243227	114	6032
BA	60	1	4477	271870	135	7407
Patents	20	2	605	3076	13436	25544
Patents	60	1	620	3126	5907	13009
Pokec	20	2	3884	217592	161	7249
Pokec	60	1	4343	240544	116	5146
Livejournal	20	2	3703	348933	234	13540
Livejournal	60	1	4667	394023	129	7087

to generate Q, and in interest of presentation, here we report what we consider to be representative results.

Table II shows the cost (number of API calls) as well as the size (number of edges) of the retrieved graph G_x . Our main observation from this is that for Grid, Geo, and Patents, ObliviousExpansion results in fewer API calls than AdaptiveExpansion, while for BA, Pokec, and Live-journal the situation is reversed. This agrees with the intuition discussed in section IV-B that, for densely connected graphs, AdaptiveExpansion should be able to uncover the nodes of a large and densely connected component of G that contains many of the query nodes Q. Indeed, graphs BA, Pokec, and Livejournal are more densely connected than Grid, Geo, and Patents, and it appears that AdaptiveExpansion is able to terminate significantly earlier than the $(\alpha + 1)$ expansion iterations of ObliviousExpansion.

Notice that the number of edges in G_x is proportional to the number of API calls, as expected. The number of edges is of interest as it affects directly the running time of MAX-DISCREPANCY algorithms, as shown in Figure 6. Figure 6 contains one point for each experiment we ran, with different MAXDISCREPANCY algorithms indicated with different color.

Figure 7 shows a comparison of the expansion algorithms in terms of how they affect the MAXDISCREPANCY algorithms. Every point in the figure corresponds to the same input (graph and set of query nodes), while the x and y axes show the accuracy obtained when the expansion is done using AdaptiveExpansion and ObliviousExpansion, respectively. If the expansion algorithms had no effect, all points would fall on the diagonal. However, we observe that in particular with Random-ST using ObliviousExpansion leads to substantially worse accuracy than when using AdaptiveExpansion. For BFS-Tree and Smart-ST the effect is not as strong, with ObliviousExpansion leading to slightly better performance (points are more likely to reside above than below the diagonal). We observe similar behavior for the other networks.

D. Results: discrepancy maximization

Continuing our discussion on Figure 6, we observe that Random-ST, BFS-Tree and Smart-ST scale to up to two orders of magnitude larger inputs than PCST-Tree. This behavior is well-aligned with the theoretical complexity of the algorithms.

Running time (in sec)



Figure 6. Running times of the different algorithms as a function of expansion size (number of edges). We can see that in comparison to PCST-Tree Smart-ST scales to inputs that are up to two orders of magnitude larger.

Indeed, the running time of BFS-Tree is $\mathcal{O}(|Q||E|)$, the running time of Random-ST is $\mathcal{O}(I|E|\log|E|)$, where I is the number of random trees sampled, and the running time of Smart-ST is $\mathcal{O}(|E|\log|E|)$, that is, they all scale very well. On the other hand, the best implementation for PCST-Tree is $\mathcal{O}(|V|^2 \log |V|)$ [10], while our non-optimized implementation has complexity $\mathcal{O}(|V||E|)$. Thus, theory and practice suggest that, from the perspective of efficiency, PCST-Tree is the least attractive algorithm.

To compare the MAXDISCREPANCY algorithms in terms of the quality of results, we measure and report the accuracy and discrepancy of the returned solutions. The results are shown in Tables III and IV.

Tables III shows the accuracy of the algorithms for different graphs, query sets, and the two expansion algorithms. Next to each reported value, we cite in parenthesis the number of times the algorithm failed to finish in 5 minutes.

Our main observation is that there are no major differences across the different algorithms in terms of the accuracy of the solution found. The only exception to that rule appears to be the case of ObliviousExpansion on the graphs of Pokec and Livejournal, where BFS-Tree outperforms the others. However, observe that if the solution must be computed very fast, Smart-ST can be a feasible choice, as it always finished within the 5 minute time limit.

Furthermore, we observe that for the synthetic networks Grid and Geo the expansion algorithm used (Oblivious-Expansion and AdaptiveExpansion) does not affect the accuracy of the solutions we obtain. (For BA, most experiments exceeded the imposed time limit and therefore we do not compare accuracy in its case). However, the measurements in Table III show that ObliviousExpansion leads to solutions of higher accuracy on real graphs. We believe this is again explained by the larger expansions that are produced by

Table III.	ACCURACY,	AVERAGES	of 20	RUNS
------------	-----------	----------	-------	------

				ObliviousE	xpansion		AdaptiveExpansion			
dataset	s	$_{k}$	BFS-Tree	Random-ST	PCST-Tree	Smart-ST	BFS-Tree	Random-ST	PCST-Tree	Smart-ST
Grid	20	2	0.88 (0)	0.81 (0)	0.93 (0)	0.93 (0)	0.88 (0)	0.85 (0)	0.93 (0)	0.93 (0)
Grid	60	1	1.00 (0)	0.94 (0)	1.00 (0)	1.00 (0)	0.99 (0)	0.98 (0)	1.00 (0)	1.00 (0)
Geo	20	2	1.00 (0)	0.95 (0)	1.00 (0)	1.00 (0)	1.00 (0)	0.98 (0)	1.00 (0)	1.00 (0)
Geo	60	1	1.00 (0)	0.96 (0)	1.00 (0)	1.00 (0)	0.99 (0)	0.98 (0)	0.99 (0)	0.99 (0)
BA	20	2	0.47 (12)	0.18 (12)	NaN (20)	0.46 (0)	0.46 (0)	0.44 (0)	0.46 (0)	0.45 (0)
BA	60	1	NaN (20)	NaN (20)	NaN (20)	0.77 (0)	0.76 (0)	0.76 (0)	0.77 (3)	0.76 (0)
Patents	20	2	0.92 (0)	0.86 (0)	0.91 (0)	0.90 (0)	0.72 (0)	0.74 (0)	0.77 (3)	0.74 (0)
Patents	60	1	0.89 (0)	0.76 (0)	0.89 (0)	0.89 (0)	0.74 (0)	0.73 (0)	0.74 (0)	0.74 (0)
Pokec	20	2	0.53 (2)	0.13 (3)	NaN (20)	0.46 (0)	0.43 (0)	0.41 (0)	0.42 (2)	0.40 (0)
Pokec	60	1	0.74 (6)	0.09 (6)	NaN (20)	0.61 (0)	0.48 (0)	0.46 (0)	0.45 (1)	0.45 (0)
Livejournal	20	2	0.62 (5)	0.19 (5)	NaN (20)	0.54 (0)	0.56 (0)	0.53 (0)	0.58 (5)	0.56 (0)
Livejournal	60	1	0.88 (12)	0.26 (9)	NaN (20)	0.68 (0)	0.65 (0)	0.62 (0)	0.62 (1)	0.62 (0)

Table IV.DISCREPANCY, AVERAGES OF 20 RUNS

				ObliviousE	xpansion		AdaptiveExpansion			
dataset	s	$_{k}$	BFS-Tree	Random-ST	PCST-Tree	Smart-ST	BFS-Tree	Random-ST	PCST-Tree	Smart-ST
Grid	20	2	14.5 (0)	11.8 (0)	16.8 (0)	16.7 (0)	14.8 (0)	13.8 (0)	16.4 (0)	16.3 (0)
Grid	60	1	41.0 (0)	36.9 (0)	41.0 (0)	41.0 (0)	40.5 (0)	38.9 (0)	40.9 (0)	40.9 (0)
Geo	20	2	19.9 (0)	18.4 (0)	20.0 (0)	20.0 (0)	19.9 (0)	19.2 (0)	20.0 (0)	20.0 (0)
Geo	60	1	22.0 (0)	20.6 (0)	22.0 (0)	22.0 (0)	21.8 (0)	21.6 (0)	21.8 (0)	21.8 (0)
BA	20	2	15.0 (12)	2.8 (12)	NaN (20)	15.2 (0)	15.6 (0)	14.4 (0)	14.4 (0)	15.0 (0)
BA	60	1	NaN (20)	NaN (20)	NaN (20)	36.1 (0)	37.4 (0)	35.3 (0)	35.9 (3)	35.5 (0)
Patents	20	2	17.4 (0)	15.8 (0)	17.7 (0)	17.6 (0)	14.9 (0)	13.8 (0)	15.8 (3)	14.8 (0)
Patents	60	1	40.0 (0)	31.1 (0)	40.8 (0)	40.6 (0)	33.0 (0)	32.2 (0)	33.2 (0)	33.3 (0)
Pokec	20	2	11.6 (2)	2.6 (3)	NaN (20)	11.8 (0)	8.6 (0)	8.0 (0)	8.2 (2)	8.0 (0)
Pokec	60	1	36.6 (6)	4.7 (6)	NaN (20)	28.6 (0)	20.9 (0)	17.4 (0)	18.3 (1)	18.5 (0)
Livejournal	20	2	14.3 (5)	3.5 (5)	NaN (20)	13.8 (0)	11.8 (0)	9.8 (0)	10.8 (5)	10.2 (0)
Livejournal	60	1	45.6 (12)	12.0 (9)	NaN (20)	31.1 (0)	29.8 (0)	25.6 (0)	26.8 (1)	27.4 (0)

ObliviousExpansion for denser graphs.

Finally, Table IV reports the discrepancy of returned solutions. These measurements paint a picture similar to that of Table III: ObliviousExpansion can lead to better performance at the cost of more API calls and for large, dense graphs (BA, Pokec, Livejournal) PCST-Tree fails to produce results within the set time limit. Additionally, we observe that Random-ST is consistently outperformed by the other algorithms, and the difference in performance is most pronounced in the case of real-world networks (Patents, Pokec, Livejournal) and ObliviousExpansion.

E. Discussion on state-of-the-art methods

To the best of our knowledge, this is the first work to study the discrepancy-maximization problem on graphs, under the local-access model, so there is lack of a natural competitor to compare the performance of our expansion algorithms.

With respect to solving the MAXDISCREPANCY problem in the unrestricted-access model, the most similar approaches are the discovery of center-piece subgraphs [18], the "cocktailparty" approach [17], and the DOT2DOT family of algorithms [3]. However, all of those algorithms are distinct enough so that direct comparison is problematic. Firstly, they all optimize functions that are very different than the discrepancy. Secondly, they all return solutions that are required to contain all query nodes, while our problem formulation allows solutions with subsets of the query nodes.

We also note that once a subgraph has been discovered in the expansion phase, any of the above-mentioned algorithm can be applied on the resulting subgraph, and in this sense these methods can be considered complementary to our approach. The caveat here, however, is that the expansion algorithms have been designed having in mind that in the second phase we aim to maximize the discrepancy function.

VI. CONCLUSION

We introduce the problem of discrepancy maximization in graphs, which we formulate as a generalization of discrepancy maximization in Euclidean spaces, a family of problems often referred to as "scan statistics." We are particularly interested in settings where only a set of initial "query nodes" is available, while the rest of the graph is hidden and it needs to be discovered via an expansion phase. This setting, which we call local-access model, is motivated by real-world application scenarios, where accessing the graph is expensive or the graph is not materialized. The challenge in the local-access model is to decide when to stop the expensive expansion phase, while ensuring that the discovered subgraph contains a maximumdiscrepancy solution. Conceptually, the model allows to work with graphs that are potentially infinite.

We then study how to find a maximum-discrepancy solution, once a graph has been discovered and it can be stored in the main memory. We refer to this setting as unrestricted-access model. The problem is **NP**-hard in the general case, but we show that if the graph is a tree the problem can be solved optimally in linear time, via dynamic programming. Based on this observation, we propose four different algorithms for the general case of the discrepancy-maximization problem, three of which scale extremely well as they are almost linear.



Figure 7. Accuracy and discrepancy of the algorithms when using either ObliviousExpansion or AdaptiveExpansion on the same set of query nodes in BA.

Our empirical evaluation shows that the best choice for the expansion strategy depends on the structure of the graph. For sparse graphs an oblivious strategy works best, while for dense graphs, an adaptive strategy is preferable. Our results also indicate that the four algorithms we considered for the unrestricted-access model yield comparable performance. In this respect, the BFS-Tree algorithm is a reasonable choice, as it is both efficient and the quality of its solutions compares favorably to other heuristics.

Our work opens many opportunities for future research. As an immediate next step, one could study alternative expansion strategies based on different rules for selecting which nodes to expand next, and different stopping criteria. Additionally, node information (attributes, text, tags, etc.) could be used to refine the expansion strategy. Another very interesting direction is to consider other families of discrepancy functions, e.g., when the discrepancy function depends also on the edges of the component, and not only on its nodes.

REFERENCES

- D. Agarwal, A. McGregor, J. M. Phillips, S. Venkatasubramanian, and Z. Zhu. Spatial scan statistics: approximations and performance study. In *KDD*, 2006.
- [2] D. Agarwal, J. M. Phillips, and S. Venkatasubramanian. The hunting of the bump: on maximizing statistical discrepancy. In SODA, 2006.
- [3] L. Akoglu, D. H. Chau, C. Faloutsos, N. Tatti, H. Tong, and J. Vreeken. Mining connection pathways for marked nodes in large graphs. In SDM, 2013.
- [4] R. Andersen and K. J. Lang. Communities from seed sets. In WWW, 2006.
- [5] S. D. Bay and M. J. Pazzani. Detecting group differences: Mining contrast sets. *Data Min. Knowl. Discov.*, 5(3):213–246, 2001.
- [6] D. P. Dobkin, D. Gunopulos, and W. Maass. Computing the maximum bichromatic discrepancy, with applications to computer graphics and machine learning. *Journal of Computer and System Sciences*, 52(3):453–470, 1996.
- [7] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *KDD*, 1999.
- [8] J. H. Friedman and N. I. Fisher. Bump hunting in high-dimensional data. Statistics and Computing, 9(2):123–143, 1999.
- [9] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal of Computing*, 24(2):296–317, 1995.
- [10] D. S. Johnson, M. Minkoff, and S. Phillips. The prize collecting Steiner tree problem: Theory and practice. In SODA, 2000.
- [11] M. Kulldorff. A spatial scan statistic. Communications in Statistics-Theory and methods, 26(6):1481–1496, 1997.
- [12] D. B. Neill and A. W. Moore. A fast multi-resolution method for detection of significant spatial disease clusters. In *NIPS*, 2003.
- [13] D. B. Neill and A. W. Moore. Rapid detection of significant spatial clusters. In KDD, 2004.
- [14] D. B. Neill, A. W. Moore, F. Pereira, and T. M. Mitchell. Detecting significant multidimensional spatial clusters. In *NIPS*, 2004.
- [15] P. K. Novak, N. Lavrač, and G. I. Webb. Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining. *The Journal of Machine Learning Research*, 10:377– 403, 2009.
- [16] S. Seufert, S. Bedathur, J. Mestre, and G. Weikum. Bonsai: Growing interesting small trees. 2013 IEEE 13th International Conference on Data Mining, 0:1013–1018, 2010.
- [17] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, 2010.
- [18] H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In KDD, 2006.
- [19] B. Wang, J. M. Phillips, R. Schreiber, D. M. Wilkinson, N. Mishra, and R. Tarjan. Spatial scan statistics for graph clustering. In *SDM*, pages 727–738. SIAM, 2008.
- [20] S. Wrobel. An algorithm for multi-relational discovery of subgroups. In *PKDD*, 1997.