Efficient Location-Aware Influence Maximization

Guoliang Li[†] Shuo Chen[†] Jianhua Feng[†] Kian-lee Tan[‡] Wen-Syan Li[*]
[†]Department of Computer Science, Tsinghua University, Beijing, China
[‡]Department of Computer Science, National University of Singapore, Singapore
[*]SAP Lab, Shanghai, China
{liguoliang,fengjh,s-chen13}@tsinghua.edu.cn; tankl@comp.nus.edu.sg; wen-syan.li@sap.com

ABSTRACT

Although influence maximization, which selects a set of users in a social network to maximize the expected number of users influenced by the selected users (called influence spread), has been extensively studied, existing works neglected the fact that the location information can play an important role in influence maximization. Many real-world applications such as location-aware word-of-mouth marketing have location-aware requirement. In this paper we study the location-aware influence maximization problem. One big challenge in location-aware influence maximization is to develop an efficient scheme that offers wide influence spread. To address this challenge, we propose two greedy algorithms with $1 - 1/e$ approximation ratio. To meet the instant-speed requirement, we propose two efficient algorithms with $\epsilon \cdot (1 - 1/e)$ approximation ratio for any $\epsilon \in (0, 1]$. Experimental results on real datasets show our method achieves high performance while keeping large influence spread and significantly outperforms state-of-the-art algorithms.

Categories and Subject Descriptors
H.2 [Database Management]: Database applications;
H.2.8 [Database applications]: Spatial databases and GIS

1. INTRODUCTION

In influence maximization, the goal is to select a set of users in a social network to maximize the expected number of influenced users (called influence spread). The prevalence of social networks, e.g., Twitter and Facebook, has prompted both industrial and academic communities [2,10] to pay close attention to the influence maximization problem. However, existing studies neglected the fact that location information can play an important role in influence maximization. Many real-world applications such as location-aware word-of-mouth marketing have location-aware requirement in influence maximization. For example, a social network system (e.g., Twitter) wants to provide new companies (e.g., restaurants) with marketing services by locating their potential customers in a spatial region (e.g., Snowbird, Utah) to promote their businesses. When a company has a limited budget to target only k such initial users, it becomes critical to be able to select those users (who may not be in the region) who can influence their friends, their friends' friends and so on, who are in the region. Through the word-of-mouth effect (or viral marketing), a large number of users close to the company would know the company. Existing studies show that people are more likely to trust the information obtained from their friends than that from general advertisement channels, e.g., TV and newspaper [17,18].

Given a location-aware social network, where each user has a geographical location, and a query with a geographical region and an integer k, the location-aware influence maximization problem selects k initial users as seeds to maximize the influence spread, i.e., the expected number of users in the query region that are influenced by these selected seeds. There are two main issues in location-aware influence maximization. The first is to obtain the users' locations. Fortunately, there are many location-aware social networks such as Foursquare (foursquare.com) and Jiepang (jiepang.com), which inherently capture the location for each user. In addition, there are many studies on obtaining users' locations from social networks [14–16], which also provide location-aware opportunities in influence maximization. The second is to meet the high-performance requirement because many applications aim to support online queries. In the above example, a large number of companies want to promote their business, and the system should support online location-aware influence maximization queries efficiently. Although there are many influence maximization algorithms [2,10], they cannot meet the high-performance requirement because they have to enumerate large number of users and cannot prune insignificant users that have small influences. As the location-aware influence maximization problem is NP-hard (see Section 2.1), it calls for effective methods to achieve high performance while not sacrificing much influence spread.

To address the challenge, we propose two efficient algorithms with $1 - 1/e$ approximation ratio. The first is an expansion-based method, which first checks the users in the query region and then progressively expands to their friends. It is worth noting that it is rather expensive to compute multiple users' influence because a user may have multiple ways to influence another user and selecting a user will affect other users' influences. To address this issue, we adopt a best-first search framework, which efficiently estimates the upper bounds of users' influences and preferentially accesses the user with large upper bounds so as to prune insignificant users. The second is to use spatial-based indexes to improve

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2588561.

search performance. We divide the whole space into small regions. For each small region, we precompute and maintain the users that have influences to users in the region, with the corresponding influences. Given a query, we assemble those small regions that have intersections with the query region and utilize the precomputed users and their influences to facilitate identifying top-k seeds.

However for large k, these two algorithms are still expensive. To meet the instant-speed requirement for online queries, we propose two efficient algorithms with $\epsilon \cdot (1 - 1/e)$ approximation ratio for any $\epsilon \in (0, 1]$. The first is a bound-based method which utilizes the expansion-based or assembly-based algorithms to estimate the upper bound and lower bound of the top-k seeds' influences. If the lower bound is not smaller than ϵ times the upper bound, the bound-based algorithm can terminate prematurely. The second is a hint-based method that avoids computing too many bounds in the bound-based method. It first precomputes the top-k seeds (called hints) for each small region. Then for each query, it uses these hints to estimate these upper and lower bounds more accurately and efficiently.

We make the following contributions. (1) We formulate the location-aware influence maximization problem. We devise two greedy algorithms with $1-1/e$ approximation ratio. The first is an expansion-based algorithm which estimates the upper bound of users' influences and adopts a best-first method to eliminate the insignificant users. The second is an assembly-based algorithm which assembles the precomputed information on small regions to answer a query. (2) We propose two efficient algorithms with $\epsilon \cdot (1-1/e)$ approximation ratio for any $\epsilon \in (0, 1]$. The first is a bound-based algorithm that uses the estimated upper bounds and lower bounds to select top-k seeds. The second is a hint-based algorithm that utilizes precomputed hints to identify top-k seeds. (3) Experimental results on real datasets show our method achieves high performance while keeping large influence spread and significantly outperforms state-of-the-art algorithms by 2-3 orders of magnitude.

The rest of this paper is structured as follows. We formulate our problem in Section 2. An expansion-based method is presented in Section 3 and an assembly-based method is proposed in Section 4. We develop efficient algorithms with $\epsilon \cdot (1 - 1/e)$ approximation ratio in Section 5. We analyze the complexity and discuss the update issues in Section 6. Experimental results are reported in Section 7. We review related works in Section 8 and conclude in Section 9.

2. PRELIMINARY

2.1 Problem Formulation

We model a location-based social network as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where vertices in $\mathcal{V}$ are users and edges in $\mathcal{E}$ are follower/followee relationships. Each vertex $v \in \mathcal{V}$ has a geographical location (x, y) with longitude x and latitude y. Initially, each vertex is *inactive*. If a vertex u is selected as a seed, u becomes *active* and it will also activate its out-neighbors. If u's out-neighbor v becomes active, v will in turns activate v's out-neighbors. There are many methods to model this process and a widely-adopted method is the independent cascade (IC) model [2,10]. Consider an activated vertex u. For each of u's inactive out-neighbor v, u has an independent probability $\mathcal{P}(u, v)$ to activate vertex v through edge (u, v). The newly activated vertices will attempt to activate their inactive out-neighbours independently. It is worth noting that a vertex has only one chance to activate its out-neighbors and after activating all of its out-neibours, it will stay active and will not activate other vertices again. This process terminates when there is no newly activated vertex.

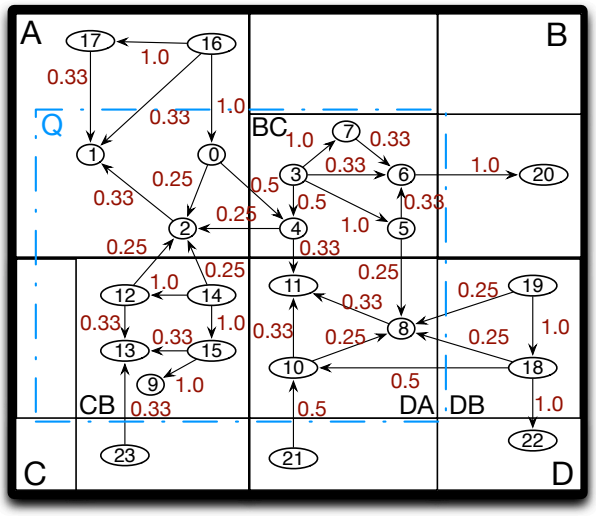


Figure 1: A Running Example.

Formally, given a query $\mathcal{Q} = (\mathcal{R}, k)$ with a geographical region $\mathcal{R}$ and an integer k, let $\mathcal{V}_\mathcal{R}$ denote the set of vertices in $\mathcal{R}$. We want to find a set of k seeds $\mathcal{S}$ from the graph (i.e., a subset of $\mathcal{V}$ with k vertices) to activate the maximum number of vertices in $\mathcal{V}_\mathcal{R}$. The number of activated vertices in $\mathcal{V}_\mathcal{R}$ is called *influence spread*, denoted by $\sigma(\mathcal{S}, \mathcal{V}_\mathcal{R})$. As each vertex has a probability to be activated through multiple vertices, it requires to compute the *expected number* of activated vertices. Next we formulate the location-aware influence maximization problem.

Definition 1. (*Location-Aware Influence Maximization*) *Given a location-aware social network $\mathcal{G}$ and a query $\mathcal{Q} = (\mathcal{R}, k)$, find a k-vertex set $\mathcal{S} \in \mathcal{G}$, such that for any other k-vertex set $\mathcal{K} \in \mathcal{G}$, $\sigma(\mathcal{S}, \mathcal{V}_\mathcal{R}) \geq \sigma(\mathcal{K}, \mathcal{V}_\mathcal{R})$. $\mathcal{S}$ is called a* `seed set` *and each vertex in $\mathcal{S}$ is called a* `seed`.

Example 1. *Figure 1 shows a location-aware social network with 24 vertices. The numbers on each directed edges are probabilities and in this paper we use the weighted cascade model as an example which sets $\mathcal{P}(u, v) = \frac{1}{N_v}$, where N_v is the number of v's in-neighbor. For example, $\mathcal{P}(14, 2) =$ 0.25 since vertex 2 has four in-neighbors, i.e., vertices 0, 4, 12, 14. Given the query $\mathcal{Q}$ with the dotted rectangle as the query region and $k = 5$, the top-k seed set of the query is $\mathcal{S} =$ {14, 3, 16, 10, 8}. Notice that $\mathcal{S}$ contains vertex 16 which is not located in the query region. Thus we cannot simply use vertices located in the query region to identify top-k seeds. In addition, although vertex 8 is influenced by vertex 10, it has additional influence to itself and vertex 11.*

Different from existing influence maximization algorithms [2] which compute top-k vertices to maximize the influence spread on all vertices, we focus on maximizing the influence spread on vertices within a given query region. We aim to support online queries. The location-aware influence maximization problem can be proved to be NP-hard by a reduction from the influence maximization problem [10] and computing the exact location-aware influence spread can be proved to be

#P-hard by a reduction from the influence spread problem [2], by setting $\mathcal{V}_\mathcal{R} = \mathcal{V}$. To meet instant-speed requirement (e.g., within 1 second) for online queries, in this paper we propose efficient algorithms to achieve high performance while not sacrificing much influence spread.

2.2 Tree-based Approximation Model

We extend state-of-the-art tree-based approximation model for influence maximization [2] to approximate our problem. We use this model as an example, and our method can be easily extended to support other approximation models.

Under the IC model, when a vertex u activates its inactive out-neighbors such as v, u also has a probability to activate v's out-neighbors such as w, even though u has no direct edge to w. This prompted us to introduce the concept of *influence* and *influenced-by*. Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with propagation probability $\mathcal{P}(u, v)$ on edge $(u, v) \in \mathcal{E}$. Let $p = \langle u = w_1, w_2, \ldots, w_m = v\rangle$ denote a path from u to v. Using the IC model, the probability that v is *influenced by* u (or u influences v) through this path equals to the product of propagation probabilities on edges along this path, denoted as $\mathcal{P}(p) = \prod_{i=1}^{m-1} \mathcal{P}(w_i, w_{i+1})$. Since each vertex has only one chance to influence its neighbors, the best chance that vertex v is influenced by u is through the path from u to v with the maximum probability. Let $\mathcal{P}(u \rightsquigarrow v)$ denote the influence of u to v, which is the maximum probability that u influences v, i.e.,

$$\mathcal{P}(u \rightsquigarrow v) = \max\{\mathcal{P}(p) | p \textit{ is any path from } u \textit{ to } v\}. \quad (1)$$

It is worth noting that given a set $\mathcal{S}$, the influence of $\mathcal{S}$ to v, denoted as $\mathcal{P}(\mathcal{S}, v)$, *cannot* be simply computed as $\sum_{u\in\mathcal{S}} \mathcal{P}(u \rightsquigarrow v)$, because different vertices in $\mathcal{S}$ may have correlations when they influence v. To address this issue, we use a tree-based model to compute the influence. Formally, given a vertex v, for each $u \in \mathcal{S}$, we find a path with the maximum probability, denoted by $u \rightsquigarrow v$. We construct a tree by combining these paths from each vertex in $\mathcal{S}$ to v, taking v as the root and use the tree to compute $\mathcal{P}(\mathcal{S}, v)$.

Obviously if $v \in \mathcal{S}$, $\mathcal{P}(\mathcal{S}, v) = 1$; otherwise v must be influenced by the children of v. The probability that v is influenced by its child c which is in turns influenced by $\mathcal{S}$ is $\mathcal{P}(\mathcal{S}, c) \cdot \mathcal{P}(c, v)$. We can combine these probabilities for all of v's children to compute the influence of $\mathcal{S}$ on v,

$$\mathcal{P}(\mathcal{S}, v) = 1 - \prod_{c\in \textsc{Child}(v)} 1 - \mathcal{P}(\mathcal{S}, c) \cdot \mathcal{P}(c, v),$$

where $\textsc{Child}(v)$ is the set of children of v in the tree.

In summary, we can compute $\mathcal{P}(\mathcal{S}, v)$ as follows.

$$\mathcal{P}(\mathcal{S}, v) = \begin{cases} 1 & v \in \mathcal{S} \\ 1 - \prod_{c\in \textsc{Child}(v)} 1 - \mathcal{P}(\mathcal{S}, c) \cdot \mathcal{P}(c, v) & v \notin \mathcal{S} \end{cases} \quad (2)$$

Example 2. *Suppose we already select vertices* 4 *and* 12 *as our seeds. We show how to calculate their co-influence to vertex* 1*, i.e.,* $\mathcal{P}(\{12, 4\}, 1)$*. Since the maximum influence paths of vertices* 4 *and* 12 *to vertex* 1 *must go through vertex* 2*, we have* $\mathcal{P}(\{12, 4\}, 1) = 1-\big(1-\mathcal{P}(\{12, 4\}, 2)\cdot\mathcal{P}(2, 1)\big)$*. As* $\mathcal{P}(\{12, 4\}, 2) = 1 - \big(1 - \mathcal{P}(12, 2)\big)\cdot\big(1 - \mathcal{P}(4, 2)\big) = 0.4375$ *and* $\mathcal{P}(2, 1) = 0.33$*,* $\mathcal{P}(\{12, 4\}, 1) = 0.4375 * 0.33 \approx 0.144$*.*

We propose an influence spread function $\hat{\sigma}$ to approximate the influence spread function σ as below.

$$\hat{\sigma}(\mathcal{S}, \mathcal{V}_\mathcal{R}) = \sum_{v\in\mathcal{V}_\mathcal{R}} \mathcal{P}(\mathcal{S}, v). \quad (3)$$

Using the approximate influence spread function in Equation 3, we can extend existing greedy algorithms to support our problem. We first select the vertex s with maximum $\hat{\sigma}(\{s\}, \mathcal{V}_\mathcal{R})$, then select vertex u with the maximum $\hat{\sigma}(\{s, u\}, \mathcal{V}_\mathcal{R})$ based on Equation 3, and terminate after k vertices are selected. Since the function $\mathcal{P}(\mathcal{S}, v)$ is submodular and monotone, this greedy algorithm has $1 - 1/e$ approximation ratio based on the submodular theory [10].

Example 3. *Given the query* $\mathcal{Q}$ *in Figure 1, the algorithm first computes the initial influence of each vertex. For example, the influence of vertex 14 is* $\mathcal{P}(\{14\}, \mathcal{V}_\mathcal{R}) = \mathcal{P}(14 \rightsquigarrow 14) + \mathcal{P}(14 \rightsquigarrow 15) + \mathcal{P}(14 \rightsquigarrow 12) + \mathcal{P}(14 \rightsquigarrow 9) + \mathcal{P}(14 \rightsquigarrow 13) + \mathcal{P}(14 \rightsquigarrow 2) + \mathcal{P}(14 \rightsquigarrow 1) = 1 + 1 + 1 + 1 + 0.333 + 0.25 + 0.25 * 0.33 = 4.667$*. It selects the vertex with the maximum influence as the first seed and here the first seed is vertex 14. Then for each vertex* u*, it computes the influence* $\mathcal{P}(\{14\} \cup \{u\}, \mathcal{V}_\mathcal{R})$ *based on Equation 3. Iteratively it selects the following seeds 3, 16, 10, and 8.*

3. EXPANSION-BASED METHOD

Existing algorithms have two main limitations. First, they treat all vertices equally and consider all vertices to select the seeds. To alleviate this problem, our approach (Section 3.1) first selects a set of vertices (called candidate seeds) which have the potential to be selected as seeds and then utilizes these candidate seeds to identify the real results. Second, to select the next seed u with the maximum influence given the current seed set $\mathcal{S}$ (i.e., $\mathcal{P}(\mathcal{S} \cup \{u\}, \mathcal{V}_\mathcal{R})$), they update the influences of those candidates that are influenced by the selected seeds and enumerate all candidates to select the one with the maximum influence as the next seed. However for many vertices, we do not need to compute $\mathcal{P}(\mathcal{S} \cup \{u\}, \mathcal{V}_\mathcal{R})$ and we want to avoid these unnecessary computations, especially for insignificant vertices. To this end, we propose a best-first based method (Section 3.2), which estimates an upper bound of $\mathcal{P}(\mathcal{S} \cup \{u\}, \mathcal{V}_\mathcal{R})$, accesses the vertices with large upper bounds, and utilizes the bound to eliminate insignificant vertices. To achieve these goals, we devise an expansion-based algorithm (Section 3.3).

3.1 Candidate Seeds Selection

We want to identify a set of vertices (denoted as $\mathcal{C}$) which include all possible seeds. In other words, vertices not in candidate set $\mathcal{C}$ cannot be selected as seeds. Thus we only need to consider the candidate seeds in $\mathcal{C}$ to identify the seed set $\mathcal{S}$. Existing algorithms take $\mathcal{V}$ as the candidate set $\mathcal{C}$ and our goal is to reduce the set as much as possible.

Obviously, vertices in $\mathcal{V}_\mathcal{R}$ will be candidate seeds. In addition, many vertices have influences to (vertices in) $\mathcal{V}_\mathcal{R}$, and some of them have large influences and some have small influences. To differentiate them, we want to eliminate those insignificant vertices with small influences. For example, if vertex u's influence to $v \in \mathcal{V}_\mathcal{R}$ is smaller than a threshold θ, i.e., $\mathcal{P}(u \rightsquigarrow v) < \theta$, u is an insignificant vertex to v. If u is an insignificant vertex to every vertex in $\mathcal{V}_\mathcal{R}$, it is an insignificant vertex to $\mathcal{V}_\mathcal{R}$. And in this case, we will not take it as a candidate seed. (Existing algorithms also use θ to remove insignificant vertices [2].) Next we formally define the candidate seeds.

Definition 2 (Influencer and Influencee). *Given two vertices* u *and* v*,* v *is called an* `influencee` *of* u *if* $\mathcal{P}(u \rightsquigarrow v) \geq \theta$ *and* u *is called an* `influencer` *of* v*.*

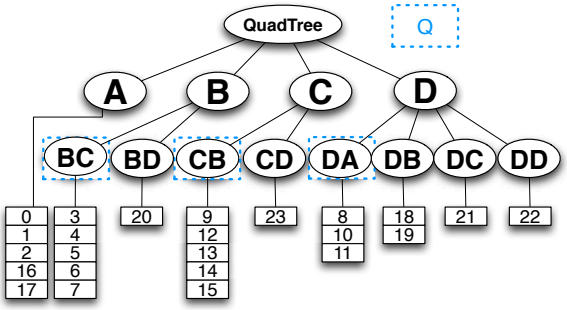


Figure 2: `QuadTree` Index.

DEFINITION 3 (CANDIDATE SEED). *Given a query $\mathcal{Q} = (\mathcal{R}, k)$, u is a candidate seed if it has an influencee in $\mathcal{V}_\mathcal{R}$.*

Let $\mathcal{I}_v^r = \{u|\mathcal{P}(u \rightsquigarrow v) \geq \theta\}$ denote the ***influencer set*** of vertex v and $\mathcal{I}_v^e = \{u|\mathcal{P}(v \rightsquigarrow u) \geq \theta\}$ denote the ***influencee set*** of vertex v. To support online query efficiently, we precompute these lists for each vertex (which is also adopted in our comparison method extended from existing algorithms in Section 2.2). Obviously, $\mathcal{C} = \cup_{v \in \mathcal{V}_\mathcal{R}} \mathcal{I}_v^r$ is a candidate seed set. To efficiently identify candidate seeds, we build a `QuadTree` for vertices in $\mathcal{V}$ based on their locations and utilize the `QuadTree` to compute the set of vertices in the query region, i.e., $\mathcal{V}_\mathcal{R}$. Then for each vertex $v \in \mathcal{V}_\mathcal{R}$, we enumerate v's in-neighbors, e.g., u. If $\mathcal{P}(u \rightsquigarrow v) \geq \theta$, we add u into $\mathcal{C}$ and continue traversing u's in-neighbors. Iteratively we get $\mathcal{C}$. For example, Figure 2 shows the `QuadTree` of vertices in Figure 1. Suppose $\theta = 0.05$. For query $\mathcal{Q}$, vertices in the query region are candidate seeds. Vertices 16, 17, 18, 19, 21, 23 are candidate seeds as they have influencees to the query region. Vertices 20 and 22 are not candidate seeds as they have no influencees to the query region.

3.2 The Best-first Method to Identify a Seed

Existing algorithms require to update the influences of all vertices that have co-influence with selected seeds. Alternatively, we propose a best-first method which estimates upper bounds of influences and utilizes the bounds to select seeds.

In the first iteration to select the first seed, we calculate the influence $\mathcal{P}(\{u\}, \mathcal{V}_\mathcal{R}) = \sum_{v \in \mathcal{V}_\mathcal{R}} \mathcal{P}(u \rightsquigarrow v)$ for each vertex $u \in \mathcal{C}$. The vertex with the maximum influence is the first seed. To facilitate selecting seeds, we maintain a max-heap for each vertex $u \in \mathcal{C}$ with initial influence $\mathcal{P}(\{u\}, \mathcal{V}_\mathcal{R})$. Obviously the top vertex on the max-heap is the first seed, denoted by s. We pop s, add s into $\mathcal{S}$, and adjust the heap.

In the second iteration, we check the top vertex u in the max-heap. If $\mathcal{I}_u^e \cap \mathcal{I}_s^e = \phi$, u and s have no co-influence, and $\mathcal{P}(\{u, s\}, v) - \mathcal{P}(\{s\}, v) = \mathcal{P}(u \rightsquigarrow v)$ for any vertex v; otherwise u and s will co-influence vertices in $\mathcal{I}_u^e \cap \mathcal{I}_s^e$ and we need to compute $\mathcal{P}(\{u, s\}, \mathcal{V}_\mathcal{R})$. As it is expensive to compute $\mathcal{P}(\{u, s\}, \mathcal{V}_\mathcal{R})$, we estimate an upper bound. First, if u and s have independent influences to v,

$$\mathcal{P}(\{u, s\}, v) = \mathcal{P}(u \rightsquigarrow v) + \mathcal{P}(s \rightsquigarrow v) - \mathcal{P}(u \rightsquigarrow v) \cdot \mathcal{P}(s \rightsquigarrow v);$$

otherwise

$$\mathcal{P}(\{u, s\}, v) \leq \mathcal{P}(u \rightsquigarrow v) + \mathcal{P}(s \rightsquigarrow v) - \mathcal{P}(u \rightsquigarrow v) \cdot \mathcal{P}(s \rightsquigarrow v).$$

Second, for any child c of v, $\mathcal{P}(\{s\}, c) \leq 1$. Based on Equation 2, we have

$$\mathcal{P}(\{u, s\}, v) \leq 1 - \prod_{c \in \text{CHILD}(v)} (1 - \mathcal{P}(c, v)).$$

Accordingly, we have an upper bound of $\mathcal{P}(\{u, s\}, v)$:

$$\hat{\mathcal{P}}(\{u, s\}, v) = \min \begin{cases} \mathcal{P}(u \rightsquigarrow v) + \mathcal{P}(s \rightsquigarrow v) - \mathcal{P}(u \rightsquigarrow v)\mathcal{P}(s \rightsquigarrow v). \\ 1 - \prod_{c \in \text{CHILD}(v)} (1 - \mathcal{P}(c, v)) \end{cases}$$

Similarly we can estimate $\mathcal{P}(\{u, s\}, \mathcal{V}_\mathcal{R})$ by

$$\hat{\mathcal{P}}(\{u, s\}, \mathcal{V}_\mathcal{R}) = \sum_{v \in \mathcal{V}_\mathcal{R}} \hat{\mathcal{P}}(\{u, s\}, v). \quad (4)$$

Let $\mathcal{P}(\{u\}|\{s\}, \mathcal{V}_\mathcal{R})$ denote u's incremental influence given a seed set $\{s\}$. We have $\mathcal{P}(\{u\}|\{s\}, \mathcal{V}_\mathcal{R}) = \mathcal{P}(\{u, s\}, \mathcal{V}_\mathcal{R}) - \mathcal{P}(\{s\}, \mathcal{V}_\mathcal{R})$. Obviously we can estimate $\mathcal{P}(\{u\}|\{s\}, \mathcal{V}_\mathcal{R})$ by,

$$\hat{\mathcal{P}}(\{u\}|\{s\}, \mathcal{V}_\mathcal{R}) = \hat{\mathcal{P}}(\{u, s\}, \mathcal{V}_\mathcal{R}) - \mathcal{P}(\{s\}, \mathcal{V}_\mathcal{R}). \quad (5)$$

EXAMPLE 4. *Recall computing $\mathcal{P}(\{12, 4\}, 1) = 0.144$ in Example 2. We show how to estimate its bound $\hat{\mathcal{P}}(\{12, 4\}, 1) = 1 - \big(1 - \mathcal{P}(12 \rightsquigarrow 1)\big) \cdot \big(1 - \mathcal{P}(4 \rightsquigarrow 1)\big) = 1 - (1 - 0.25 * 0.33) \cdot (1 - 0.25 * 0.33) \approx 0.158$. Since $\mathcal{P}(12 \rightsquigarrow 1)$ and $\mathcal{P}(4 \rightsquigarrow 1)$ have correlations on vertex 2, the estimated influence is slightly larger than the real influence. If the tree is large, the estimation-based method is much more efficient than computing the real influence. In our example, the expansion-based method estimates the bounds 20 times and computes influences 5 times.*

Generally, suppose we get a seed set $\mathcal{S}_i = \{s_1, s_2, \cdots, s_i\}$. We estimate vertex u's incremental influence given $\mathcal{S}_i$ by

$$\hat{\mathcal{P}}(\{u\}|\mathcal{S}_i, \mathcal{V}_\mathcal{R}) = \hat{\mathcal{P}}(\mathcal{S}_i \cup \{u\}, \mathcal{V}_\mathcal{R}) - \mathcal{P}(\mathcal{S}_i, \mathcal{V}_\mathcal{R}), \quad (6)$$

where

$$\hat{\mathcal{P}}(\mathcal{S}_i \cup \{u\}, \mathcal{V}_\mathcal{R}) = \sum_{v \in \mathcal{V}_\mathcal{R}} \hat{\mathcal{P}}(\mathcal{S}_i \cup \{u\}, v),$$

$$\hat{\mathcal{P}}(\mathcal{S}_i \cup \{u\}, v) = \min \begin{cases} \mathcal{P}(u \rightsquigarrow v) + \mathcal{P}(\mathcal{S}_i, v) - \mathcal{P}(u \rightsquigarrow v)\mathcal{P}(\mathcal{S}_i, v) \\ 1 - \prod_{c \in \text{CHILD}(v)} (1 - \mathcal{P}(c, v)) \end{cases}$$

Using the bounds, we discuss how to find the seeds. Let $\mathcal{I}_{\mathcal{S}_i}^e$ denote the set of influencees for seeds in $\mathcal{S}_i$, i.e., $\mathcal{I}_{\mathcal{S}_i}^e = \cup_{s_j \in \mathcal{S}_i} \mathcal{I}_{s_j}^e$. For the top vertex u in the heap, if u has no co-influence with $\mathcal{S}_i$, i.e., $\mathcal{I}_u^e \cap \mathcal{I}_{\mathcal{S}_i}^e = \phi$, u is the next seed. We terminate this iteration. If u has co-influence with $\mathcal{S}_i$, (1) if its influence is still the initial influence (called ***outdated***), we estimate bound $\hat{\mathcal{P}}(\{u\}|\mathcal{S}_i, \mathcal{V}_\mathcal{R})$ based on Equation 6, add $\langle u, \hat{\mathcal{P}}(\{u\}|\mathcal{S}_i, \mathcal{V}_\mathcal{R})\rangle$ into max-heap $\mathcal{H}$, and set it ***estimated***; (2) if the influence of u is already ***estimated***, we compute $\mathcal{P}(\{u\}|\mathcal{S}_i, \mathcal{V}_\mathcal{R})$ based on Equation 2, add $\langle u, \mathcal{P}(\{u\}|\mathcal{S}_i, \mathcal{V}_\mathcal{R})\rangle$ into heap $\mathcal{H}$, and set it ***computed***; (3) if the influence of u is ***computed***, u is the next seed. We terminate this iteration. We use a hash map $\mathcal{M}$ to maintain whether a vertex is outdated, estimated or computed. As the top vertex always has the largest (outdated/estimated/computed) influence, we can employ the best-first method to select the next seed.

3.3 Expansion-based Algorithm

We devise an expansion-based algorithm and the pseudo-code is illustrated in Algorithm 1. It first precomputes the influencee sets and influencer sets (line 1). Given a query, it first computes the candidate set $\mathcal{C}$ (line 2) and builds a max-heap for each candidate $u \in \mathcal{C}$ with the corresponding initial influence $\mathcal{P}(\{u\}, \mathcal{V}_\mathcal{R})$ (line 3). It also maintains a set of selected seeds $\mathcal{S}$ (line 4) and the influencee set $\mathcal{I}^e$ of

Algorithm 1: Expansion-based Algorithm

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: A graph; $\mathcal{Q} = (\mathcal{R}, k)$: A query.
Output: $\mathcal{S}$: k-vertex set

```
// Offline - Indexing
```
1 Precompute $\mathcal{I}^r_v$ and $\mathcal{I}^e_v$;
```
// Online - Search
```
2 Compute the candidate set $\mathcal{C}$ for vertices in $\mathcal{V}_\mathcal{R}$;
3 Build max-heap $\mathcal{H}$ for $u \in \mathcal{C}$ with influence $\mathcal{P}(\{u\}, \mathcal{V}_\mathcal{R})$;
4 Initialize a seed set $\mathcal{S} = \emptyset$;
5 Initialize an influencee set $\mathcal{I}^e = \emptyset$;
6 **for** $i \leftarrow 1$ **to** k **do**
7 Initialize a hash map $\mathcal{M} = \emptyset$;
8 **while** $\mathcal{H} \neq \emptyset$ **do**
9 $u = \mathcal{H}.pop()$;
10 **if** $\mathcal{I}^e_u \cap \mathcal{I}^e = \phi$ **then**
11 $\mathcal{S} = \mathcal{S} \cup \{u\}$; $\mathcal{I}^e = \mathcal{I}^e \cup \mathcal{I}^e_u$; break;
12 **else**
13 **if** $u \notin \mathcal{M}$ *(i.e., outdated)* **then**
14 Estimate $\hat{\mathcal{P}}(\{u\}|\mathcal{S}, \mathcal{V}_\mathcal{R})$;
15 Add $\langle u, \hat{\mathcal{P}}(\{u\}|\mathcal{S}, \mathcal{V}_\mathcal{R})\rangle$ into $\mathcal{H}$;
16 Add $\langle u,$ estimated$\rangle$ into $\mathcal{M}$;
17 **if** $u \in \mathcal{M}$ & *estimated* **then**
18 Compute $\mathcal{P}(\{u\}|\mathcal{S}, \mathcal{V}_\mathcal{R})$;
19 Add $\langle u, \mathcal{P}(\{u\}|\mathcal{S}, \mathcal{V}_\mathcal{R})\rangle$ into $\mathcal{H}$;
20 Use $\langle u,$ computed$\rangle$ to update $\mathcal{M}$;
21 **else if** $u \in \mathcal{M}$ & *computed* **then**
22 $\mathcal{S} = \mathcal{S} \cup \{u\}$; $\mathcal{I}^e = \mathcal{I}^e \cup \mathcal{I}^e_u$; break;

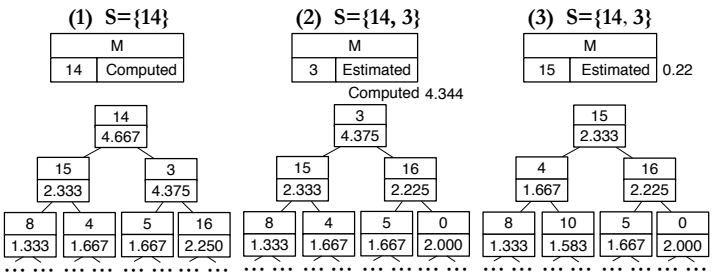


Figure 3: An Example for Expansion-based Method.

seeds in $\mathcal{S}$ (line 5). Then it iteratively selects a seed with the maximum incremental influence from the heap (lines 6-22). For each top vertex u in the max-heap, it pops u from the heap. If u has no co-influence with selected seeds in $\mathcal{S}$, i.e., $\mathcal{I}^e_u \cap \mathcal{I}^e = \phi$, u is a seed. It adds u into $\mathcal{S}$ and updates $\mathcal{I}^e = \mathcal{I}^e \cup \mathcal{I}^e_u$ (line 11). If $u \notin \mathcal{M}$, i.e., its influence is outdated, it estimates its influence based on Equation 7 and adds $\langle u, \hat{\mathcal{P}}(\{u\}|\mathcal{S}, \mathcal{V}_\mathcal{R})\rangle$ into heap $\mathcal{H}$ and $\langle u,$ estimated$\rangle$ into map $\mathcal{M}$ (lines 13-16). If $u \in \mathcal{M}$ and its influence is estimated, it computes u's real influence based on Equation 2, adds $\langle u, \mathcal{P}(\{u\}|\mathcal{S}, \mathcal{V}_\mathcal{R})\rangle$ into heap $\mathcal{H}$, and utilizes $\langle u,$ computed$\rangle$ to update $\mathcal{M}$ (lines 17-20). If $u \in \mathcal{M}$ and its influence is computed, u is a seed. It adds u into $\mathcal{S}$ and updates $\mathcal{I}^e = \mathcal{I}^e \cup \mathcal{I}^e_u$ (line 22).

EXAMPLE 5. *For query $\mathcal{Q}$ in Figure 1, we first identify the candidate seeds and build a max-heap (Figure 3). In the heap, the top four vertices are 14, 3, 15, 16 with initial influences of* 4.667, 4.375, 2.333, *and* 2.25 *respectively. In the first iteration, vertex* 14 *has the largest initial influence* 4.667 *and it is the first seed. Then the next top vertex is vertex* 3. *Since it has co-influence with vertex* 14, *we estimate its incremental influence* $\hat{\mathcal{P}}(\{3\}|\{14\}, \mathcal{V}_\mathcal{R}) = 4.344$ *using Equation 5. Then we add* $\langle 3, 4.344\rangle$ *into the heap. As vertex 3 is still the top vertex, we calculate vertex* 3*'s accurate incremental influence using Equation 3. Since vertex 3's incremental influence is larger than vertex 15's initial influence, vertex 3 is the next seed. Next the top vertex in the heap is vertex 15. We estimate its incremental influence. Since vertex 15 only influences vertices 15, 9, 13,* $\hat{\mathcal{P}}(\{15\}|\{14, 3\}, \mathcal{V}_\mathcal{R}) = \hat{\mathcal{P}}(\{15\}|\{14, 3\}, 15) + \hat{\mathcal{P}}(\{15\}|\{14, 3\}, 9) + \hat{\mathcal{P}}(\{15\}|\{14, 3\}, 13) = 0 + 0 + \big((1 - (1 - 0.33) * (1 - 0.33)) - 0.33\big) = 0.22$. *Then the next top vertex in the heap is 16. After estimating its incremental influence, it will be the next seed. Similarly, we select the following seeds 10 and 8.*

4. ASSEMBLY-BASED METHOD

The expansion-based method has two limitations. First, it requires to calculate the candidates set $\mathcal{C}$ and initial influence $\mathcal{P}(\{u\}, \mathcal{V}_\mathcal{R})$ for every vertex $u \in \mathcal{C}$. If the query region is large, there will be large number of candidate seeds in $\mathcal{C}$. To address these problems, we propose an index-based method to efficiently compute the candidate set and initial influences (Section 4.1). Second, the candidate set $\mathcal{C}$ contains many insignificant candidates which will not be selected as seeds and thus we do not want to compute their influences. In other words, we want to examine superior vertices with large influences earlier to prune inferior vertices with small influences. To achieve this goal, we propose an assembly-based framework (Section 4.2) and devise an efficient assembly-based algorithm (Section 4.3).

4.1 Indexes

Node-Influencer Index $\mathcal{L}$. We maintain a node-to-influencer index $\mathcal{L}$. For each `QuadTree` node $\mathcal{R}_i$, we precompute the influencer set for vertices in node $\mathcal{R}_i$, denoted by $\mathcal{I}^r_{\mathcal{R}_i}$, which is the union of influencer sets of vertices in node $\mathcal{R}_i$, i.e., $\mathcal{I}^r_{\mathcal{R}_i} = \cup_{v \in \mathcal{R}_i} \mathcal{I}^r_v$. In addition, for each vertex $u \in \mathcal{I}^r_{\mathcal{R}_i}$, we also precompute its influence to vertices in $\mathcal{R}_i$, denoted by $\mathcal{P}(\{u\}, \mathcal{V}_{\mathcal{R}_i}) = \sum_{v \in \mathcal{R}_i} \mathcal{P}(u \rightsquigarrow v)$. We use a list $\mathcal{L}_{\mathcal{R}_i}$ to maintain all the vertices in $\mathcal{I}^r_{\mathcal{R}_i}$ with their influences to $\mathcal{R}_i$, sorted by a decreasing order, i.e., $\mathcal{L}_{\mathcal{R}_i} = \{\langle u, \mathcal{P}(u, \mathcal{V}_{\mathcal{R}_i})\rangle | u \in \mathcal{I}^r_{\mathcal{R}_i}\}$. We use a bottom-up method to efficiently compute the lists. First, for each leaf node, we compute $\mathcal{I}^r_{\mathcal{R}_i}$ and $\mathcal{P}(\{u\}, \mathcal{V}_{\mathcal{R}_i})$ by traversing vertices in the leaf node. Then for each non-leaf node $\mathcal{R}_c$, its candidate set is the union of the candidate sets of its children, i.e., $\mathcal{I}^r_{\mathcal{R}_c} = \cup_{\mathcal{R}_i \in \text{CHILD}(\mathcal{R}_c)} \mathcal{I}^r_{\mathcal{R}_i}$ where $\text{CHILD}(\mathcal{R}_c)$ is the child set of $\mathcal{R}_c$. For $u \in \mathcal{R}_c$, its influence to $\mathcal{R}_c$ is $\mathcal{P}(\{u\}, \mathcal{V}_{\mathcal{R}_c}) = \sum_{\mathcal{R}_i \in \text{CHILD}(\mathcal{R}_c)} \mathcal{P}(\{u\}, \mathcal{V}_{\mathcal{R}_i})$.

Influencer-Node Index $\mathcal{F}$. We maintain an influencer-to-node index $\mathcal{F}$. For each vertex u, we keep a list of tree nodes whose influencer set contains u, with the corresponding influences, i.e., $\mathcal{F}_u = \{\langle \mathcal{R}_i, \mathcal{P}(\{u\}, \mathcal{V}_{\mathcal{R}_i})\rangle | u \in \mathcal{I}^r_{\mathcal{R}_i}\}$.

For example, Figure 4 shows the node-influencer index on `QuadTree` nodes. Consider node BC with vertices 3, 4, 5, 6, 7. Its node-influencer list includes $\{\langle 3, 3.833\rangle, \langle 5, 1.333, \rangle, \cdots\}$. That is vertex 3 will influence node BC with influence 3.833. Table 1 illustrates the influencer-node index. The list of vertex 3 is $\{\langle A, 0.125\rangle, \langle BC, 3.833\rangle, \langle DA, 0.417\rangle\}$. That is vertex 3 has influences on nodes A, BC, and DA.

4.2 Assembly-based Framework

Initialization. Given a query $\mathcal{Q}$, we identify `QuadTree` nodes that are fully covered by $\mathcal{R}$, denoted by $\mathcal{R}_\mathcal{Q} = \{\mathcal{R}_1,$

Table 1: Influencer-Node Index.

Vertex	Nodes and Influences
3	$\langle A, 0.125\rangle, \langle BC, 3.833\rangle, \langle DA, 0.417\rangle$
14	$\langle A, 0.333\rangle, \langle CB, 4.333\rangle$
16	$\langle A, 1.583\rangle, \langle BC, 0.500\rangle, \langle DA, 0.167\rangle$
...	...

$\mathcal{R}_2, \cdots, \mathcal{R}_r\}$, and the set of vertices whose corresponding nodes are not fully covered by $\mathcal{R}$, denoted by $\mathcal{V}_o$. For each $\mathcal{R}_i$, we identify the corresponding list $\mathcal{L}_{\mathcal{R}_i}$ from the node-influencer index $\mathcal{L}$. For $\mathcal{V}_o$, we on-the-fly generate such list. We first compute its influencer list $\mathcal{I}_o^r = \cup_{u \in \mathcal{V}_o} \mathcal{I}_u^r$. For each vertex $u \in \mathcal{I}_o^r$, we compute $\mathcal{P}(\{u\}, \mathcal{V}_o)$ based on Equation 3 and generate a sorted list of vertex u based on their influences, denoted by $\mathcal{L}_{\mathcal{V}_o}$. We also generate a map $\mathcal{F}_o = \{\langle u, \mathcal{P}(\{u\}, \mathcal{V}_o)\rangle | u \in \mathcal{I}_o^r\}$.

Using these lists, given a query $\mathcal{Q}$, for each vertex u, we can compute its influence to $\mathcal{R}$, i.e., $\mathcal{P}(\{u\}, \mathcal{V}_\mathcal{R})$ as below,

$$\mathcal{P}(\{u\}, \mathcal{V}_\mathcal{R}) = \mathcal{P}(\{u\}, \mathcal{V}_o) + \sum_{1 \le i \le r} \mathcal{P}(\{u\}, \mathcal{V}_{\mathcal{R}_i}) \qquad (7)$$

Next we devise an assembly-based algorithm to compute k-vertex set $\mathcal{S}$. For simplicity let $\mathcal{R}_0 = \mathcal{V}_o$. Now we have a set of lists, $\mathcal{L}_{\mathcal{R}_0} = \mathcal{L}_{\mathcal{V}_o}, \mathcal{L}_{\mathcal{R}_1}, \mathcal{L}_{\mathcal{R}_2}, \cdots, \mathcal{L}_{\mathcal{R}_r}$. We also have a influencer-node index $\mathcal{F}$ from vertices to nodes $\mathcal{R}_i$ and $\mathcal{V}_o$. We still use the max-heap $\mathcal{H}$ to identify seeds. Different from the expansion-based method, we do not insert all candidate seeds into the max-heap. Instead we add candidate seeds into the max-heap on-demand as follows.

Finding the First Seed. As vertices $\mathcal{L}_{\mathcal{R}_0}, \mathcal{L}_{\mathcal{R}_1}, \cdots, \mathcal{L}_{\mathcal{R}_r}$ are sorted by their influences in a descending order, we check the vertices in order and add the vertices with the largest influences into the max-heap. We obtain a lower bound $\mathcal{B}_\mathcal{H}$ for the next seed's influence using the heap and an upper bound $\mathcal{B}_\mathcal{L}$ for the unvisited vertices' influences using these lists. If $\mathcal{B}_\mathcal{H} \ge \mathcal{B}_\mathcal{L}$, we find the first seed and terminate; otherwise we add more vertices into the heap.

Formally, we first check the first vertex of each list, e.g., $u_0 = \mathcal{L}_{\mathcal{R}_0}[1], u_1 = \mathcal{L}_{\mathcal{R}_1}[1], u_2 = \mathcal{L}_{\mathcal{R}_2}[1], \cdots, u_r = \mathcal{L}_{\mathcal{R}_r}[1]$. Let $\mathcal{B}_\mathcal{L} = \sum_{0 \le i \le r} \mathcal{P}(\{u_i\}, \mathcal{V}_{\mathcal{R}_i})$ where $\mathcal{P}(\{u_i\}, \mathcal{V}_{\mathcal{R}_i})$ is kept in the list $\mathcal{L}_{\mathcal{R}_i}$. Obviously $\mathcal{B}_\mathcal{L}$ is an upper bound of influences of unvisited vertices. (Notice that different from the threshold-based algorithm [7], the influences in the lists will be dynamically changed.) For each vertex u_i, we compute $\mathcal{P}(\{u_i\}, \mathcal{V}_\mathcal{R}) = \sum_{u_i \in \mathcal{R}_j} \mathcal{P}(\{u_i\}, \mathcal{V}_{\mathcal{R}_j})$ where each $\mathcal{P}(\{u_i\}, \mathcal{V}_{\mathcal{R}_j})$ for $0 \le j \le r$ can be easily obtained from the influencer-node index $\mathcal{F}$. We insert these first vertices into the heap $\mathcal{H}$. After inserting all first vertices into the heap, let $\mathcal{B}_\mathcal{H}$ denote the influence of the top vertex in the heap. Obviously if $\mathcal{B}_\mathcal{H} \ge \mathcal{B}_\mathcal{L}$, the top vertex must be the seed and we terminate this iteration; otherwise we check the second vertices of each list, update $\mathcal{B}_\mathcal{L}$ using the sum of influences of the second vertices, and insert all second vertices into the heap.

EXAMPLE 6. *For query $\mathcal{Q}$, we get three fully covered nodes, BC, CB, DA, and the corresponding lists are illustrated in Figure 4. We also get other three vertices 0, 1, 2. We generate a list of these vertices: $\{\langle 16, 1.583\rangle, \langle 0, 1.333\rangle, \langle 2, 1.333\rangle, \langle 1, 1.0\rangle, \cdots\}$ as shown in Figure 4. In the first step, we access vertices $\langle 16, 1.583\rangle$, $\langle 3, 3.883\rangle$, $\langle 14, 4.333\rangle$, $\langle 10, 1.583\rangle$. We get a bound $\mathcal{B}_\mathcal{L} = 11.333$. Using the influencer-node index $\mathcal{F}$, we compute $\langle 16, 2.25\rangle$, $\langle 3, 4.375\rangle$, $\langle 14, 4.666\rangle$, $\langle 10, 1.583\rangle$ and insert them into $\mathcal{H}$. We have $\mathcal{B}_\mathcal{H} = 4.667$. Then we insert the second, third, fourth vertices of each list into the heap. We have $\mathcal{B}_\mathcal{L} = 3.917 < \mathcal{B}_\mathcal{H}$. Thus we get the first seed*

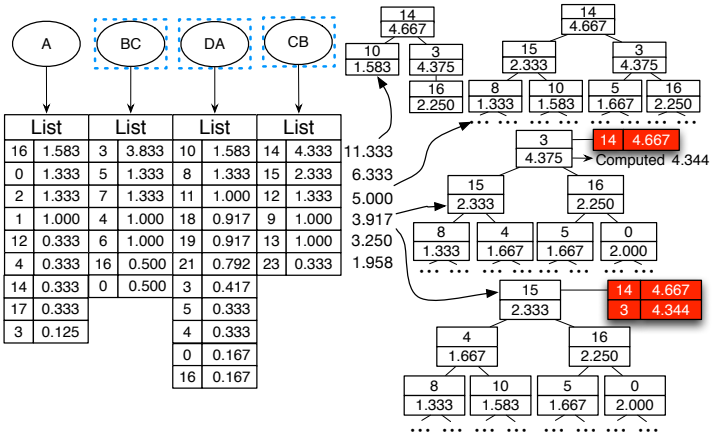


Figure 4: An Example for Assembly-based Method.

14. Here we only access 16 vertices to find the first seed and the expansion-based method visits 22 vertices.

Finding the Next Seed. Selecting the second seed is different from selecting the first seed, because once a seed s is selected, some vertices which have co-influence with the seed will be affected. We need to update their influences. To address this issue, we use the expansion-based method to find the next seed from the heap, e.g., u. We use its influence $\mathcal{P}(\{u\}|\{s\}, \mathcal{V}_\mathcal{R})$ as a lower bound of the next seed, also denoted by $\mathcal{B}_\mathcal{H}$. If $\mathcal{B}_\mathcal{H} \ge \mathcal{B}_\mathcal{L}$, then the top vertex of the heap is the next seed; otherwise, we access the next vertices of each list and insert them into the heap and update $\mathcal{B}_\mathcal{L}$ using the sum of influences of these vertices. Then we use the expansion-based method to find the next seed from the heap and update $\mathcal{B}_\mathcal{H}$ using the top vertex. Iteratively we can find top-k seeds. Unlike the expansion-based method, the max-heap has a much smaller number of candidate seeds.

EXAMPLE 7. *We continue with Example 6. The top vertex in the heap is $\langle 3, 4.344\rangle$. Thus we have $\mathcal{B}_\mathcal{H} = 4.344$. Since the upper bound for unvisited vertices is $\mathcal{B}_\mathcal{L} = 3.917 < \mathcal{B}_\mathcal{H}$, there is no vertex with better incremental influence than vertex 3. Thus the second seed is vertex 3. Notice that here we do not need to visit vertices 6, 13, 19, 23, 21, 17 to find the top-2 seeds. Therefore the assembly-based method can reduce the size of and operations on $\mathcal{H}$, especially when there are large numbers of candidate seeds.*

4.3 Assembly-based Algorithm

In this section, we give the assembly-based algorithm and the pseudo-code is shown in Algorithm 2. In the offline processing, it requires to materialize the node-influencer list $\mathcal{L}_{\mathcal{R}_i}$ for each node $\mathcal{R}_i$ (line 1), influencer-node list $\mathcal{F}_u$ for each vertex u (line 2) and influencee set $\mathcal{I}_u^e$ and influencer set $\mathcal{I}_u^r$ (line 3). In the online search, it first computes the tree nodes covered by $\mathcal{R}$ and loads the node-vertex lists $\mathcal{L}_{\mathcal{R}_1}, \mathcal{L}_{\mathcal{R}_2}, \cdots, \mathcal{L}_{\mathcal{R}_r}$ from the node-vertex index $\mathcal{L}$ (line 4). It calculates $\mathcal{L}_{\mathcal{R}_0}$ for other vertices in $\mathcal{R}$ (line 5). Then it initializes a max-heap $\mathcal{H}$, a lower bound for the heap $\mathcal{B}_\mathcal{H}$ and an upper bound for the lists $\mathcal{B}_\mathcal{L}$ (lines 6-7). Then it pops the top vertices from the lists $\mathcal{L}_{\mathcal{R}_0}, \mathcal{L}_{\mathcal{R}_2}, \cdots, \mathcal{L}_{\mathcal{R}_r}$ and inserts them into max-heap $\mathcal{H}$ (lines 10-11). Next it estimates an upper bound of the lists using the influences of the current vertices, $\mathcal{B}_\mathcal{L}$ (line 12) and uses the expansion-based algorithm to identify the next seed using the heap (line 13). Then it takes the influence of the top vertex u of the heap as a lower bound $\mathcal{B}_\mathcal{H}$ of the next seed (line 14). If $\mathcal{B}_\mathcal{H} \ge \mathcal{B}_\mathcal{L}$, the top vertex u is the next seed, and the algorithm pops it from the heap and adds it into $\mathcal{S}$ (lines 16-18).

Algorithm 2: Assembly-based Algorithm

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: A graph; $\mathcal{Q} = (\mathcal{R}, k)$: A query.
Output: $\mathcal{S}$: k-vertex set
`// Offline - Indexing`
Precompute Node-influencer List $\mathcal{L}_{\mathcal{R}_i}$ for each node $\mathcal{R}_i$;
Precompute Influencer-Node List $\mathcal{F}_u$ for each vertex u;
Precompute $\mathcal{I}_u^r$ and $\mathcal{I}_u^e$ for each vertex u;
`// Online - Search`
Identify $\mathcal{L}_{\mathcal{R}_1}, \mathcal{L}_{\mathcal{R}_2}, \cdots, \mathcal{L}_{\mathcal{R}_r}$ from Node-Vertex Index $\mathcal{L}$;
Compute $\mathcal{L}_{\mathcal{R}_0}$;
Initialize seed set $\mathcal{S} = \emptyset$;
Bound for heap $\mathcal{B}_{\mathcal{H}} = 0$; Bound for lists $\mathcal{B}_{\mathcal{L}} = 0$;
while $\cup_{0 \le i \le r} \mathcal{L}_{\mathcal{R}_i} \neq \emptyset$ **do**
  for $i \in [0, r]$ **do**
    Pop u_i from $\mathcal{L}_{\mathcal{R}_i}$;
    Insert $\left(u_i, \sum_{u_i \in \mathcal{R}_j} \mathcal{P}(\{u_i\}, \mathcal{V}_{\mathcal{R}_j})\right)$ into heap $\mathcal{H}$;
  $\mathcal{B}_{\mathcal{L}} = \sum_{0 \le i \le r} \mathcal{P}(\{u_i\}, \mathcal{V}_{\mathcal{R}_i})$;
  $u = \textsc{Expansion}(\mathcal{G}, \mathcal{Q})$;
  $\mathcal{B}_{\mathcal{H}}$ is the influence of the top element u in $\mathcal{H}$;
  if $\mathcal{B}_{\mathcal{H}} \ge \mathcal{B}_{\mathcal{L}}$ **then**
    $s = \mathcal{H}.pop()$;
    $\mathcal{S} = \mathcal{S} \cup \{s\}$;
    if $|\mathcal{S}| = k$ **then** return ;

5. ALGORITHMS WITH $\epsilon \cdot (1 - 1/e)$ APPROXIMATION RATIO

The expansion- and assembly-based algorithms greedily identify the top-k seeds to achieve $1 - 1/e$ approximation ratio. However for large k, they are still expensive since they have to visit large number of vertices and update their influences. To meet the instant-speed requirement, we propose efficient algorithms with $\epsilon \cdot (1 - 1/e)$ approximation ratio for any $\epsilon \in (0, 1]$. As such, we propose a bound-based algorithm in Section 5.1 and a hint-based algorithm in Section 5.2.

5.1 Bound-based Algorithm

We can utilize the assembly-based algorithm to estimate an upper bound and a lower bound of $\mathcal{P}(\mathcal{S}, \mathcal{V}_{\mathcal{R}})$. If the lower bound is not smaller than ϵ times the upper bound, we can terminate the algorithm and obviously this method can achieve $\epsilon \cdot (1 - 1/e)$ approximation ratio for any $\epsilon \in (0, 1]$. Next we introduce the details.

Estimate Upper Bound of $\mathcal{P}(\mathcal{S}, \mathcal{V}_{\mathcal{R}})$. We first use the assembly-based algorithm to greedily select the seeds. Suppose at the i-th iteration, it gets a seed set $\mathcal{S}_i = \{s_1, s_2, \cdots, s_i\}$ with i seeds. Since s_i is the i-th best seed, the incremental influences of other vertices will not be larger than $\mathcal{P}(s_i | \mathcal{S}_{i-1}, \mathcal{V}_{\mathcal{R}})$ based on the submodularity. We can use the i-th seed to substitute the following $k - i$ seeds and get an upper bound,

$$\mathcal{B}^u = \mathcal{P}(\mathcal{S}_i, \mathcal{V}_{\mathcal{R}}) + (k - i) \cdot \mathcal{P}(s_i | \mathcal{S}_{i-1}, \mathcal{V}_{\mathcal{R}}).$$

However the incremental influences of other vertices may be much smaller than $\mathcal{P}(s_i | \mathcal{S}_{i-1}, \mathcal{V}_{\mathcal{R}})$, and the upper bound would be loose. We want to improve the upper bound by selecting $k - i$ vertices from the max-heap. Let $\mathcal{P}_{k-i}$ denote the $(k-i)$-th largest influence in the max-heap. If the upper bound $\mathcal{B}_{\mathcal{L}}$ of the lists $(\mathcal{L}_{\mathcal{R}_i})$ is larger than $\mathcal{P}_{k-i}$, we add the vertices in the lists into the heap, until $\mathcal{B}_{\mathcal{L}}$ is not larger than the updated $(k - i)$-th largest influence in the max-heap. Here we only use Equation 6 to estimate the upper bound of such vertices and do not compute their real incremental influences. Suppose the largest $k - i$ influences in the heap are $\mathcal{B}_1^u, \mathcal{B}_2^u, \cdots, \mathcal{B}_{k-i}^u$. We get a tighter upper bound,

$$\mathcal{B}^u = \mathcal{P}(\mathcal{S}_i, \mathcal{V}_{\mathcal{R}}) + \sum_{1 \le j \le k-i} \mathcal{B}_j^u. \quad (8)$$

Estimate Lower Bound of $\mathcal{P}(\mathcal{S}, \mathcal{V}_{\mathcal{R}})$. For any vertex u, if we do not consider its influencees that are also influencees of $\mathcal{S}_i$ (in other words we only consider its influencees that are not influenced by $\mathcal{S}_i$), we can get a lower bound

$$\hat{\mathcal{P}}(u, \mathcal{V}_{\mathcal{R}}) = \sum_{w \in \mathcal{V}_{\mathcal{R}} \cap (\mathcal{I}_u^e \setminus \mathcal{I}_{\mathcal{S}_i}^e)} \mathcal{P}(u \rightsquigarrow w), \quad (9)$$

where $\mathcal{I}_{\mathcal{S}_i}^e = \cup_{s \in \mathcal{S}_i} \mathcal{I}_s^e$ is the influencee set of $\mathcal{S}_i$. As the incremental influence of u to w will not be smaller than $\mathcal{P}(u \rightsquigarrow w) - \mathcal{P}(\mathcal{S}_i, w)$, we can get a tighter lower bound,

$$\hat{\mathcal{P}}(u, \mathcal{V}_{\mathcal{R}}) = \max(0, \sum_{w \in \mathcal{V}_{\mathcal{R}} \cap \mathcal{I}_u^e} \mathcal{P}(u \rightsquigarrow w) - \mathcal{P}(\mathcal{S}_i, w)) \quad (10)$$

To select the top-$(k-i)$ vertices with the largest lower bound, we can use a priority queue to keep these vertices. When we add a vertex into the max-heap, we estimate its lower bound. If its lower bound is larger than the $(k - i)$-th largest influence in the priority queue, we use it to replace the one in the priority queue with the smallest value. Then we can use the vertices in the priority queue to estimate the lower bound. For simplicity, suppose $v_1, v_2, \cdots, v_{k-i}$ are the $k - i$ vertices with the largest lower bounds in the priority queue. We compute their incremental lower bounds, denoted by $\mathcal{B}_{v_1}^l, \mathcal{B}_{v_2}^l, \cdots, \mathcal{B}_{v_{k-i}}^l$. (When computing the incremental lower bound of v_j, we temporarily treat $v_1, v_2, \cdots, v_{j-1}$ as seeds.) Using the i seeds and the $k-i$ vertices in the priority queue, we get a lower bound of $\mathcal{P}(\mathcal{S}, \mathcal{V}_{\mathcal{R}})$,

$$\mathcal{B}^l = \mathcal{P}(\mathcal{S}_i, \mathcal{V}_{\mathcal{R}}) + \sum_{1 \le j \le k-i} \mathcal{B}_{v_j}^l. \quad (11)$$

Obviously if $\mathcal{B}^l \ge \epsilon \cdot \mathcal{B}^u$, we can terminate the algorithm, since we can use vertices $v_1, v_2, \cdots, v_{k-i}$ as the $k - i$ seeds, and the influences of these vertices plus the i seeds are not smaller than ϵ times the influence of $\mathcal{S}$. If $\mathcal{B}^l < \epsilon \cdot B^u$, we identify the next seed and update the lower and upper bounds. Since the upper bound $\mathcal{B}^u$ monotonically decreases and the lower bound $\mathcal{B}^l$ monotonically increases, iteratively we can find seeds with $\epsilon \cdot (1 - 1/e)$ approximation ratio.

EXAMPLE 8. *Suppose $k = 5$ and $\epsilon = 80\%$. After we select two seeds $\langle 14, 4.667\rangle, \langle 3, 4.344\rangle$, we use heap $\mathcal{H}$ and lists $\mathcal{L}_{\mathcal{R}_i}$ to find another 3 vertices with largest incremental influences to get 5 vertices so as to compute the upper bound. As influences popped from $\mathcal{H}$ may be outdated, we estimate their upper bound based on Equation 6. The top vertices in the heap are $\langle 16, 1.858\rangle, \langle 4, 1.667\rangle, \langle 5, 1.667\rangle$ and thus the upper bound is $\mathcal{B}^u = 14.203$. The top three vertices in the priority queue are vertices 16, 10 and 0. We compute their incremental lower bounds, $\langle 16, 1.0\rangle$, $\langle 10, 1.0\rangle$, $\langle 0, 0.0\rangle$ using Equation 9. Thus the lower bound is $\mathcal{B}^l = 11.01$ computed by seeds $\langle 14, 4.667\rangle$, $\langle 3, 4.344\rangle$ and three lower bounds. As $14.203 * 80\% > 11.01$, we need to select the third seed, i.e. vertex 16. Then the upper bound is updated to $\mathcal{B}^u = 13.251$ and the new lower bound is $\mathcal{B}^l = 11.813$. As $13.251 * 80\% < 11.813$, the algorithm terminates. Here we only select 3 seeds and use vertices 14, 3, 16, 10, 0 as the top-5 seeds.*

5.2 Hint-based Algorithm

The bound-based method uses the vertices in the priority queue to substitute some real seeds. Since it does not consider the co-influences among these vertices, the influence spread may be much smaller. To address this issue, we propose a hint-based method.

Basic Idea. For each tree node $\mathcal{R}_i$, we can precompute its top-k seeds with the corresponding influence to $\mathcal{R}_i$, i.e., the answer of $\mathcal{Q}_{\mathcal{R}_i} = (\mathcal{R}_i, k)$, denoted by $\mathcal{H}_{\mathcal{R}_i}$. Each vertex in $\mathcal{H}_{\mathcal{R}_i}$ is called a hint for node $\mathcal{R}_i$. Obviously, we have considered the co-influences between the hints in $\mathcal{H}_{\mathcal{R}_i}$ and we can use them to estimate a much tighter lower bound. In addition, for the vertex set $\mathcal{V}_o$ (the vertices in $\mathcal{R}$ whose corresponding node is not fully covered by $\mathcal{R}$), we also compute its top-k seeds, denoted by $\mathcal{H}_{\mathcal{R}_0} = \mathcal{H}_{\mathcal{V}_o}$. Since there are small number of vertices in $\mathcal{V}_o$, we can efficiently compute $\mathcal{H}_{\mathcal{R}_0}$ and also take them as hints. Given a query $\mathcal{Q} = (\mathcal{R}, k)$, we assemble the hints in $\mathcal{H}_{\mathcal{R}_0}, \mathcal{H}_{\mathcal{R}_1}, \cdots, \mathcal{H}_{\mathcal{R}_r}$ to estimate a tighter lower bound of $\mathcal{P}(\mathcal{S}, \mathcal{V}_{\mathcal{R}})$ as follows.

Hint-based Lower Bound. We can prove that for any k distinct vertices $v_1, v_2, \cdots, v_k$ in these hint lists, the sum of their influences in the corresponding lists must be a lower bound. That is $\sum_{1\le i\le k}\sum_{v_i\in\mathcal{H}_{\mathcal{R}_j}} \mathcal{H}_{\mathcal{R}_j}[v_i] \le \mathcal{P}(\mathcal{S}, \mathcal{V}_{\mathcal{R}})$ as stated in Theorem 1, where $\mathcal{H}_{\mathcal{R}_j}[v_i]$ is the incremental influence of v_i in the hint list $\mathcal{H}_{\mathcal{R}_j}$.

THEOREM 1. *Given a query $\mathcal{Q} = (\mathcal{R}, k)$, let $\mathcal{H}_{\mathcal{R}_0}$, $\mathcal{H}_{\mathcal{R}_1}$, $\cdots$, $\mathcal{H}_{\mathcal{R}_r}$ denote the hint lists of $\mathcal{Q}$'s vertex set $\mathcal{V}_o$ and fully covered regions $\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_r$, for any k distinct vertices in these lists, $v_1, v_2, \cdots, v_k$, we have*

$$\sum_{1\le i\le k}\sum_{v_i\in\mathcal{H}_{\mathcal{R}_j}} \mathcal{H}_{\mathcal{R}_j}[v_i] \le \mathcal{P}(\mathcal{S}, \mathcal{V}_{\mathcal{R}}),$$

where $\mathcal{S}$ is the real seed set of $\mathcal{Q}$.

PROOF SKETCH. First, since the query region $\mathcal{R}$ covers regions $\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_r$ and vertices in $\mathcal{V}_o$ and the influences of hints in these lists are only for local region $\mathcal{R}_i$, we have $\sum_{v_i\in\mathcal{H}_{\mathcal{R}_j}} \mathcal{H}_{\mathcal{R}_j}[v_i] \le \mathcal{P}(v_i, \mathcal{V}_{\mathcal{R}})$. Thus vertices in different lists have no influence overlap in any region. Second, if vertices v_i and v_j are in the same list, we have considered their co-influence in the list. Thus the Theorem is correct. □

Based on Theorem 1, we want to select the k distinct vertices with the largest total influences. To this end, we can scan the hints in these lists to find k hints to maximize the total influence. We can also use the threshold-based algorithm [7] to find top k hints. As there are a small number of hints, it is very efficient to find top k hints. Let $\mathcal{H}_{\mathcal{R}}$ denote the set of top-k hints with the largest influences. Using these k hints, we can get a tighter lower bound,

$$\mathcal{B}^l_{\mathcal{H}} = \sum_{v_i\in\mathcal{H}_{\mathcal{R}}}\sum_{v_i\in\mathcal{H}_{\mathcal{R}_j}} \mathcal{H}_{\mathcal{R}_j}[v_i]. \tag{12}$$

Hint-based Method. Given a query $\mathcal{Q}$, we first estimate a lower bound $\mathcal{B}^l_{\mathcal{H}}$ based on Equation 12. Then we use the assembly-based method to find seeds. Suppose we select a seed and estimate the upper bound $\mathcal{B}^u$ based on Equation 8. If $\mathcal{B}^l_{\mathcal{H}} \ge \epsilon\cdot\mathcal{B}^u$, we terminate and use the hints as the seed set $\mathcal{S}$; otherwise if the seed is in the hint set $\mathcal{H}_{\mathcal{R}}$, we keep the hint set and go to the next iteration. If the seed is not in the hint set, we add the selected seed into the hint set $\mathcal{H}_{\mathcal{R}}$, remove the hint with the minimal influence from $\mathcal{H}_{\mathcal{R}}$, and update the influence of the hint set and the lower bound $\mathcal{B}^l_{\mathcal{H}}$. Next we use the assembly-based method to find the next seed. Iteratively, we can find the seed set $\mathcal{S}$ with $\epsilon\cdot(1-1/e)$ approximation ratio. It is worth noting that the hint-based method will retreat to the bound-based method if any initial hint is not a selected seed.

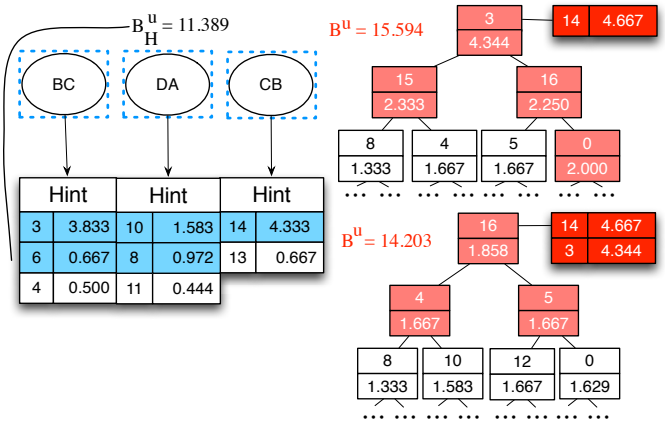


Figure 5: Hint-based Method.

EXAMPLE 9. *Suppose $k = 5$ and we use the fully covered regions to select hints as shown in Figure 5. We choose 5 hints with the largest incremental influences from hints, i.e., 14, 3, 10, 8, 6. We sum up their incremental influences and get a lower bound $\mathcal{B}^l_{\mathcal{H}} = 11.389$. Recall Example 8, the first seed is vertex 14. The upper bound is $\mathcal{B}^u = 15.594$. As $15.594 * 80\% > 11.389$, we select the next seed. As vertex 14 is in the hint set, we do not update lower bound. The second seed is vertex 3 and the upper bound is updated to $\mathcal{B}^u = 14.203$. As $14.203*80\% < 11.389$, the algorithm terminates. We take vertices 14, 3, 10, 8, 6 as top-k seeds. Compared with the bound-based method which computes 3 seeds to do estimation, the hint-based method only identifies 2 seeds.*

6. COMPLEXITY AND UPDATE

6.1 Space Complexity

Influencer/Influencee Index $\mathcal{I}^r_v/\mathcal{I}^e_v$. Suppose the average number of influencers/influencees is $\tau^r_\theta/\tau^e_\theta$. The complexity of the influencer/influencee index is $\mathcal{O}(\tau^r_\theta|\mathcal{V}|)/\mathcal{O}(\tau^e_\theta|\mathcal{V}|)$.
`QuadTree` Index. If each leaf node contains m vertices, the number of leaf nodes in the `QuadTree` is $\frac{|\mathcal{V}|}{m}$ and the total number of nodes (denoted by $\mathtt{N_{QT}}$) is $\mathtt{N_{QT}} = 2\frac{|\mathcal{V}|}{m}$. The complexity of the `QuadTree` is $\mathcal{O}(\mathtt{N_{QT}} + |\mathcal{V}|) = \mathcal{O}(|\mathcal{V}|)$.
Influencer-Node Index $\mathcal{F}$. As each vertex appears in $\min(\tau^e_\theta, \mathtt{N_{QT}})$ `QuadTree` nodes in average, the space complexity of the influencer-node index is $\mathcal{O}(\min(\tau^e_\theta, \mathtt{N_{QT}})|\mathcal{V}|)$.
Node-Influencer Index $\mathcal{L}$. The influencer-node index is the inverted index of the node-influencer index and its complexity is also $\mathcal{O}(\min(\tau^e_\theta, \mathtt{N_{QT}})|\mathcal{V}|)$.
Hint Index $\mathcal{H}$. Each `QuadTree` node contains at most k hints, and the space complexity is $\mathcal{O}(k\mathtt{N_{QT}})$.

6.2 Time Complexity

The expansion-based method first generates the candidates. The time complexity to retrieve the vertices in the query region is $\mathcal{O}(\mathtt{N_{QT}} + |\mathcal{V}_{\mathcal{R}}|)$. For each $v \in \mathcal{V}_{\mathcal{R}}$, it needs to compute its influencer set $\mathcal{I}^r_v$ and takes $\cup_{v\in\mathcal{V}_{\mathcal{R}}}\mathcal{I}^r_v$ as the candidate set. For each $u \in \cup_{v\in\mathcal{V}_{\mathcal{R}}}\mathcal{I}^r_v$, it computes the influence $\mathcal{P}(\{u\}, \mathcal{V}_{\mathcal{R}})$ and the cost is $|\mathcal{I}^e_u|$. Thus the time complexity to generate the candidates is $\mathcal{O}(|\mathcal{V}_{\mathcal{R}}|\tau^r_\theta\tau^e_\theta)$, where $\tau^r_\theta/\tau^e_\theta$ is

the average number of influencers/influencees. Next it uses the heap to select k seeds. The heap construction complexity is $\mathcal{O}(|\mathcal{C}|)$. Each candidate is estimated and computed at most k times. Each selected seed affects $\min(|\mathcal{C}|, \tau_\theta^r \tau_\theta^e)$ vertices that are required to compute the incremental influence. The estimation-based method estimates the incremental influence of vertex v based on $\mathcal{I}_v^e$ and the complexity is $\mathcal{O}(\tau_\theta^e)$. To actually compute the incremental influence, it uses the Dijkstra algorithm and the complexity is $\mathcal{O}(\tau_\theta^e \log \tau_\theta^e + E) = \mathcal{O}(\tau_\theta^e \log \tau_\theta^e + \tau_\theta^e \tau_o)$, where E is the number of edges accessed in the Dijkstra algorithm, τ_o is the average out-degree of vertices, and E can be estimated by $\tau_\theta^e \tau_o$. The heap adjustment complexity is $\mathcal{O}(\log |\mathcal{C}|)$. Thus the time complexity of the expansion-based method is $\mathcal{O}\big(|\mathcal{V}_\mathcal{R}|\tau_\theta^e \tau_\theta^r + |\mathcal{C}| + k \min(|\mathcal{C}|, \tau_\theta^e \tau_\theta^r)(\tau_\theta^e \log \tau_\theta^e + \tau_\theta^e \tau_o + \log |\mathcal{C}|\big)$.

The assembly-based method avoids finding the candidates and the number of vertices in the heap will be much smaller. The time complexity is $\mathcal{O}(|\mathcal{C}_a| + k \min(|\mathcal{C}_a|, \tau_\theta^e \tau_\theta^r)(\tau_\theta^e \log \tau_\theta^e + \tau_\theta^e \tau_o + \log |\mathcal{C}_a|))$, where $\mathcal{C}_a$ is the set of vertices added into the heap. The bound- and hint-based methods further reduce the number of vertices added into the heap. The time complexities are $\mathcal{O}(|\mathcal{C}_b| + k \min(|\mathcal{C}_b|, \tau_\theta^e \tau_\theta^r)\ (\tau_\theta^e \log \tau_\theta^e + \tau_\theta^e \tau_o + \log |\mathcal{C}_b|))$ and $\mathcal{O}(|\mathcal{C}_h| + k \min(|\mathcal{C}_h|, \tau_\theta^e \tau_\theta^r)(\tau_\theta^e \log \tau_\theta^e + \tau_\theta^e \tau_o + \log |\mathcal{C}_h|))$ respectively, where $\mathcal{C}_b$ and $\mathcal{C}_h$ are respectively the sets of vertices added into heaps of bound- and hint-based methods.

6.3 Updates

First we discuss how to support location updates. Suppose the location of vertex v is updated to `QuadTree` node n_2 from node n_1. If $n_1 = n_2$, we do not update our index; otherwise for the `QuadTree`, we move the vertex from n_1 to n_2 with $\mathcal{O}(1)$ complexity. For the node-influencer index, we move vertex v from the node-influencer list of node n_1 to that of node n_2. As we use a hash map to maintain the list, the time complexity is $\mathcal{O}(1)$. For the influencer-node list of v, we replace n_1 with n_2 and the time complexity is also $\mathcal{O}(1)$. For the hint index, for each hint h in node n_1, as v is moved away from node n_1, we update h's influence by subtracting $\mathcal{P}(h, v)$ using $\mathcal{I}_h^e$ and the time complexity is $\mathcal{O}(1)$. As there are at most k hints, the total complexity is $\mathcal{O}(k)$. For each hint in node n_2, its influence will not decrease. We can still use the influence as a lower bound and do not update hints in n_2. It is worth noting that the motivation of the hint-based method is to provide a promising candidate set and using a lower bound will not affect the result of our algorithm.

Then we discuss how to support graph updates.
(1) Add an edge (u, v). For each $u' \in \mathcal{I}_u^r$, we run the Dijkstra algorithm to update $\mathcal{I}_{u'}^e$ and the time complexity is $\mathcal{O}(\tau_\theta^r(\tau_\theta^e \log \tau_\theta^e + \tau_\theta^e \tau_o))$. For each $v' \in \mathcal{I}_{u'}^e$, if u' is not in $\mathcal{I}_{v'}^r$, we add u' in to $\mathcal{I}_{v'}^r$ with time complexity $\mathcal{O}(1)$. We also update the influencer-node index. Considering a `QuadTree` node $\mathcal{R}_i$, we update u''s influence to $\mathcal{R}_i$ as follows. For each $v' \in \mathcal{I}_{u'}^e \cap \mathcal{R}_i$, we update $\mathcal{P}(u', \mathcal{R}_i)$ using the new influence $\mathcal{P}(u', v')$ and the time complexity is $\mathcal{O}(\tau_\theta^e \tau_\theta^r)$. Similarly, we update the node-influencer list with the same complexity. We do not update the hint index as the influences of hints will not decrease and we still use them as lower bounds. The time complexity of adding an edge is $\mathcal{O}(\tau_\theta^r(\tau_\theta^e \log \tau_\theta^e + \tau_\theta^e \tau_o))$.
(2) Delete an edge (u, v). It is the same as adding an edge except for each hint h, the influence of h maybe decrease. If h is in $\mathcal{I}_v^r$, we update its influence by subtracting $\mathcal{P}(h, v')$ for $v' \in \mathcal{I}_v^e$ and the complexity is $\mathcal{O}(k\tau_\theta^e)$.
(3) Add/Delete an isolated vertex(with degree of 0). We insert/delete it into/from the `QuadTree` with complexity $\mathcal{O}(\log |\mathcal{V}|)$.

7. EXPERIMENTAL STUDY

Datasets. We used four real datasets `Gowalla`, `Twitter`, `Foursquare`, and `Weibo`. `Gowalla` was downloaded from an open dataset website *snap.stanford.edu/data/loc-gowalla.html*. `Twitter`, `Foursquare` and `Weibo` were respectively crawled from *twitter.com*, *foursquare.com*, and *weibo.com*. The user location was the place the user most frequently checked in. The four datasets are directed graphs and the details are shown in Table 2, where AvgD denotes the average degree, and MaxID/MaxOD denotes the maximum in-/out-degree.

We utilized two widely-used models to set the edge proabilites [2]. For the weighted cascade model, we set the probability of (u, v) as $\frac{1}{N_v}$, where N_v is number of v's in-neighbors. For the trivalency model, we randomly and uniformly set a probability in $\{0.1, 0.01, 0.001\}$, respectively corresponding to high, medium and low probabilities.

Table 2: Datasets.

Datasets	#Vertices	#Edges	AvgD	MaxID	MaxOD
`Gowalla`	197K	1.9M	9.67	739	735
`Twitter`	554K	4.29M	7.75	1,143	639
`Foursquare`	4.9M	53.7M	11.6	4702	727
`Weibo`	1.02M	166.7M	166.2	1000	4979

Queries. We randomly generated three types of queries with different region sizes. (1) Small region queries: the query region contained about 10,000 vertices. (2) Medium region queries: the query region contained about 100,000 vertices. (3) Large region queries: the query region contained about 1 million vertices. There were 1000 queries in each type and we report the average performance.

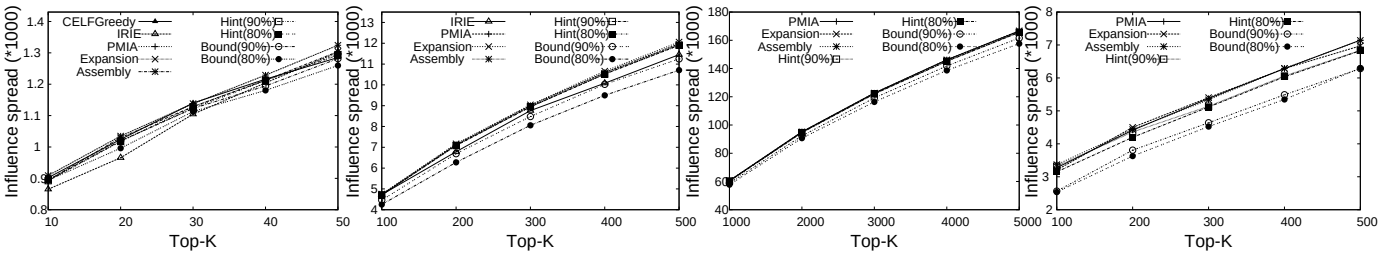
Algorithms. We compared with the state-of-the-art algorithms `PMIA` [2] and `IRIE` [9]. We obtained the source codes from the authors and extended them to support our problem as discussed in Section 2.2. We also implemented our algorithms, `Expansion`, `Assembly`, `Bound`, and `Hint`. All the algorithms were implemented using C++.

Index Sizes and Time. Due to space constrains, we only report the index sizes and time on the `Foursquare` dataset. All algorithms utilized the `QuadTree` (QT), where leaf nodes contained at most 500 vertices and the height was 7. `PMIA` and our algorithms used influencer sets $\mathcal{I}_v^r$ and influencee sets $\mathcal{I}_v^e$. `Assembly`, `Bound` and `Hint` also used the node-influencer index $\mathcal{L}$ and influencer-node index $\mathcal{F}$. `Hint` utilized an additional hint index $\mathcal{H}$. Besides these indexes, the memory usage also included the dataset and runtime usage (RT). We report the preprocessing time on the `Foursquare` dataset with $\theta = 0.05$ and $\alpha = 0.3$ using the weighted cascade model, where α was a factor in `IRIE` to tune the influence spread and `IRIE` achieved the best influence spread at $\alpha = 0.3$. `IRIE`, `PMIA`, `Expansion`, `IRIE`, `Assembly`/`Bound` and `Hint` respectively took 5.3, 118, 124, 800, and 2000 seconds. The details are shown in Table 3.

Table 3: Memory&Time on `Foursquare` (GB,k=5000).

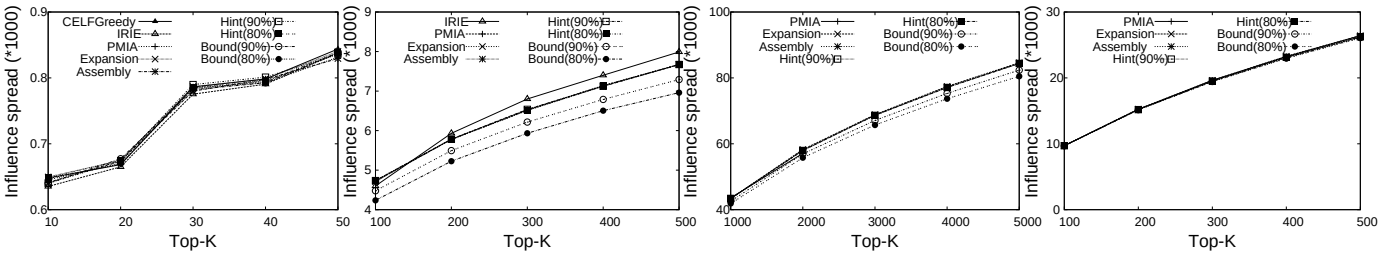
Methods	Data	QT	$\mathcal{I}_v^r/\mathcal{I}_v^e$	$\mathcal{L}\&\mathcal{F}$	$\mathcal{H}$	RT	Total	Time(s)
`IRIE`	1.4	0.13	–	–	–	0.2	1.7	5.3
`PMIA`	1.4	0.13	1.4	–	–	1.2	4.1	118
`Expansion`	1.4	0.13	1.4	–	–	1.2	4.1	124
`Assembly`	1.4	0.13	1.4	2.4	–	1.2	6.5	800
`Bound`	1.4	0.13	1.4	2.4	–	1.2	6.5	800
`Hint`	1.4	0.13	1.4	2.4	0.2	1.2	6.7	2000

Experiment settings. All the experiments were conducted on a computer with two Intel(R) Xeon(R) E5630 3.0GHZ processors and 48GB RAM, running Ubuntu 10.04.



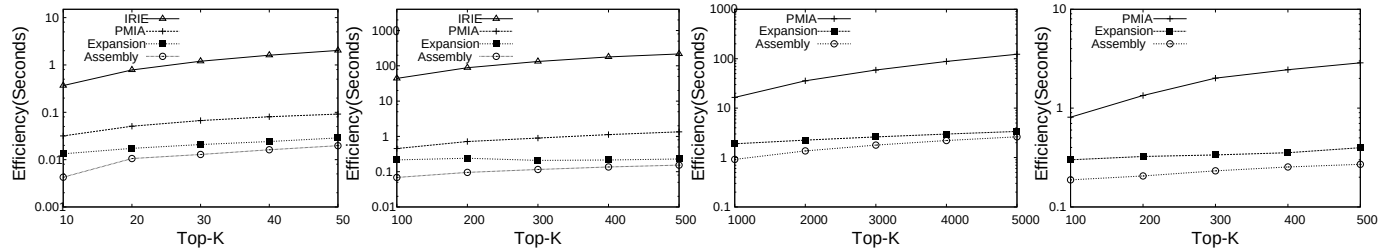
(a) `Gowalla` ($|\mathcal{V}_\mathcal{R}| \approx 10K$) (b) `Twitter` ($|\mathcal{V}_\mathcal{R}| \approx 100K$) (c) `Foursquare` ($|\mathcal{V}_\mathcal{R}| \approx 1M$) (d) `Weibo` ($|\mathcal{V}_\mathcal{R}| \approx 100K$)

Figure 6: Influence Spread on Weighted Cascade Model ($(a)(b)(c)$: $\theta = 0.05$, (d) : $\theta = 0.001$).



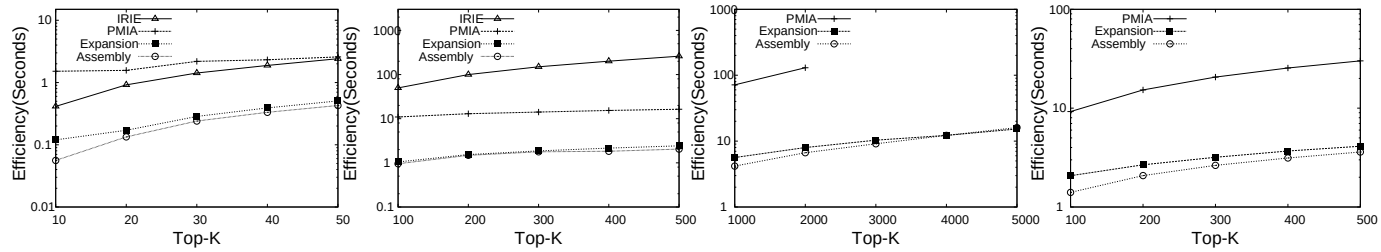
(a) `Gowalla` ($|\mathcal{V}_\mathcal{R}| \approx 10K$) (b) `Twitter` ($|\mathcal{V}_\mathcal{R}| \approx 100K$) (c) `Foursquare` ($|\mathcal{V}_\mathcal{R}| \approx 1M$) (d) `Weibo` ($|\mathcal{V}_\mathcal{R}| \approx 100K$)

Figure 7: Influence Spread on Trivalency Model ($(a)(b)(c)$: $\theta = 0.001$, (d) : $\theta = 0.01$).



(a) `Gowalla` ($|\mathcal{V}_\mathcal{R}| \approx 10K$) (b) `Twitter` ($|\mathcal{V}_\mathcal{R}| \approx 100K$) (c) `Foursquare` ($|\mathcal{V}_\mathcal{R}| \approx 1M$) (d) `Weibo` ($|\mathcal{V}_\mathcal{R}| \approx 100k$)

Figure 8: Efficiency: IRIE vs PMIA vs Expansion vs Assembly on Weighted Cascade($(a)(b)(c)$:θ=0.05,(d):θ=0.001).



(a) `Gowalla` ($|\mathcal{V}_\mathcal{R}| \approx 10K$) (b) `Twitter` ($|\mathcal{V}_\mathcal{R}| \approx 100K$) (c) `Foursquare` ($|\mathcal{V}_\mathcal{R}| \approx 1M$) (d) `Weibo` ($|\mathcal{V}_\mathcal{R}| \approx 100K$)

Figure 9: Efficiency: IRIE vs PMIA vs Expansion vs Assembly on Trivalency ($(a)(b)(c)$:θ=0.001,(d):θ=0.01).

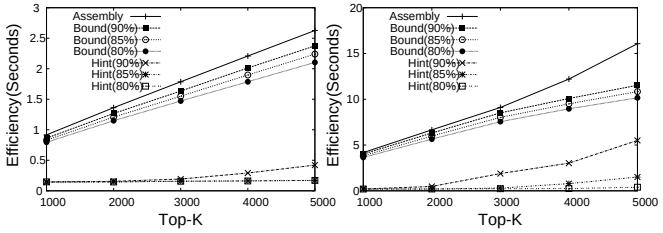
7.1 Influence Spread

We compared the influence spread of different methods. Figures 6 and 7 show the results on the weighted cascade model and trivalency model respectively, where 80% refers to $\epsilon = 80\%$. For `IRIE`, we tuned its parameter α and showed the best result at $\alpha = 0.3$. For the other algorithms, we set $\theta = 0.05$. We also compared with `CELFGreedy`, which was the original greedy algorithm with the cost-effective outbreak detection optimization [13].[1] We can see that `CELFGreedy`, `IRIE`, `PMIA`, `Expansion`, and `Assembly` achieved nearly the same influence spread on the two models, since they employed the greedy algorithm and had $1 - 1/e$ approximation ratio. As `CELFGreedy` was rather slow, we only evaluated it on the `Gowalla` dataset. Although `IRIE` achieved similar influence spread, it was rather hard to tune the parameter α to achieve high influence spread. `Hint` achieved nearly the same influence spread as the other algorithms because it can accurately estimate the lower bounds and considered the co-influences among hints. For example, `Hint` nearly achieved more than 95% approximation. `Bound` achieved smaller influence spreads as its selected vertices for substituting the seeds may have low influence spreads (as it did not consider the co-influences among these vertices).

[1] We used R = 20000 to obtain accurate estimation.

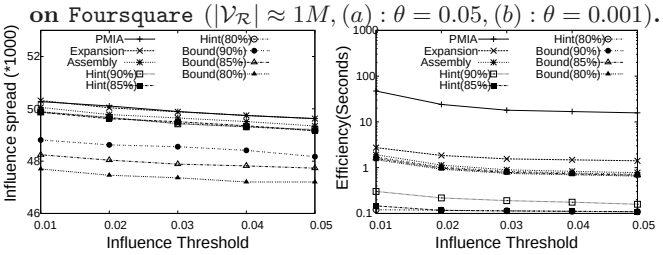
7.2 Efficiency

We evaluated the efficiency of different algorithms. We first compared `PMIA`, `IRIE`, `Expansion`, and `Assembly` with $1 - 1/e$ approximation ratio. We still used the three types of queries. Figures 8(a), (b)/(d), and (c) show the results for queries with small query regions, medium regions, and large regions respectively on the weighted cascade model and similarly Figure 9 shows the results on the trivalency model. We have five observations. First, `IRIE` had the worst performance. The main reason is that to select the next seed in each iteration, `IRIE` was more expensive than `PMIA` and our method, because `IRIE` had to update the incremental influences for all vertices (as it used linear equations to update influences) while `PMIA` and our method only updated vertices



(a) Weighted Cascade (b) Trivalency

Figure 10: Efficiency: Assembly vs Bound vs Hints on `Foursquare` ($|\mathcal{V}_\mathcal{R}| \approx 1M, (a) : \theta = 0.05, (b) : \theta = 0.001$).



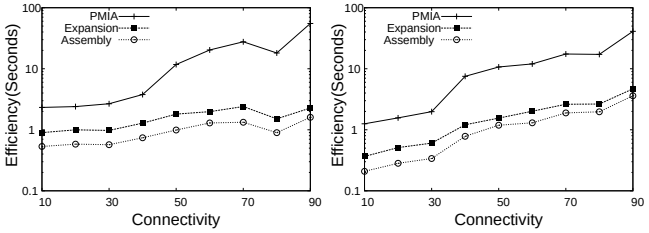
(a) Spread – `Foursquare` (b) Efficiency – `Foursquare`

Figure 11: Varying θ ($k = 1000, |\mathcal{V}_\mathcal{R}| = 1M$).

that were actually affected by the selected seeds. When k was large, `IRIE` was much worse than other methods. Although `IRIE` achieved better efficiency than `PMIA` in influence maximization, it achieved this by avoiding constructing local influence structures (i.e., shortest-path trees). To support online queries efficiently, these structures can be indexed in an offline phase, which was not included in the online query time. Since `IRIE` took much time on the `Foursquare` and `Weibo` datasets (more than 1000 seconds), we did not show the results in the figures. Second, `Assembly` outperformed `Expansion` which in turns was better than `PMIA`, because `PMIA` required to update influences for many vertices in each iteration, `Expansion` used the estimated upper bounds to prune large numbers of insignificant vertices, and `Assembly` assembled the precomputed results on small regions to reduce the heap size and facilitate computing the influence of each vertex. Third, as k increased, the elapsed time of these algorithms also increased. `PMIA` and `IRIE` increased linearly as they required to update influences for large numbers of vertices in each iteration. `Expansion` and `Assembly` increased sublinearly because they used the bounds to prune many insignificant vertices. Fourth, for large k, `Expansion` and `Assembly` significantly outperformed `PMIA` and `IRIE`, even in two orders of magnitude, because for large k, `PMIA` and `IRIE` required to compute incremental influences for large numbers of vertices while our method significantly pruned many insignificant vertices. For example, on the `Foursquare` dataset, `PMIA`, `Expansion`, `Assembly` respectively took about 150, 4, and 2 seconds. Fifth, different probability models had no much difference on the efficiency.

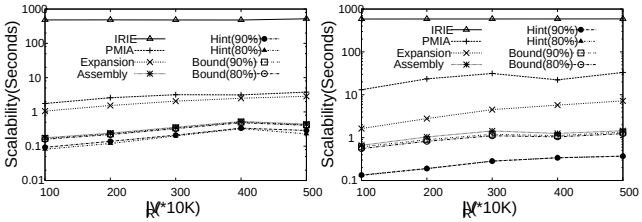
Then, we compared our algorithms `Bound` and `Hint` with $\epsilon \cdot (1 - 1/e)$ approximation ratio to `Assembly` with $1 - 1/e$ approximation ratio. Figure 10 shows the results on the `Foursquare` dataset. We have three observations. First, `Bound` and `Hint` outperformed `Assembly` significantly since they can terminate prematurely with a given approximation threshold ϵ. Second, `Hint` outperformed `Bound`, especially for large k, since `Hint` can better estimate the lower bounds using the precomputed hints. For $k = 5000$, `Bound` took 2.5 seconds while `Hint` only took 0.4 seconds. Third, the smaller ϵ, the better performance of `Bound` and `Hint`, because for a smaller ϵ, `Bound` and `Hint` can terminate earlier.

Comparison of Our Algorithms: For applications caring about efficiency, we suggest to use the `Hint` based algorithm.



(a) Weighted Cascade (θ=0.001) (b) Trivalency ($\theta = 0.01$)

Figure 12: Varying Vertex Connectivity in $\mathcal{V}_\mathcal{R}$ on `Weibo` ($k = 500$, $|\mathcal{V}_\mathcal{R}| \approx 100K$).



(a) Weighted Cascade ($\theta = 0.05$) (b) Trivalency ($\theta = 0.001$)

Figure 13: Scalability on `Foursquare` ($k = 50$).

Table 4: Average Update Cost (Milliseconds).

Datasets	Location	Graph	
		Weighted Cascade	Trivalency
`Gowalla`	$8*10^{-4}$	0.023 ($\theta = 0.05$)	0.01 ($\theta = 0.01$)
`Twitter`	$9*10^{-4}$	0.028 ($\theta = 0.05$)	0.014 ($\theta = 0.01$)
`Foursquare`	$10*10^{-4}$	0.024 ($\theta = 0.05$)	0.012 ($\theta = 0.01$)
`Weibo`	$9*10^{-4}$	0.017 ($\theta = 0.001$)	0.49 ($\theta = 0.01$)

For applications caring about approximation ratio, we recommend the `Assembly` based algorithm.

7.3 Varing θ

We evaluated our algorithm by varying θ from 0.01 to 0.05. Figure 11 shows the results for both influence spared and the elapsed time. We see that with the increase of θ, the influence spread decreased because the influence of a vertex will decrease, and the elapsed time also decreased since the influence paths of a vertex were shorter for larger θ. The determination of θ has been discussed in [2] which depends on the average degree and structure of the graph and in the paper we do not discuss the details.

7.4 Varying Connectivity

We evaluated different methods by varying the vertex connectivity (i.e., the minimum number of vertices whose removal disconnects the graph) in the query region. Figure 12 shows the results. We can see that with the increase of the connectivity, the performance became worse as more vertices were involved to compute the incremental influences. Our method still achieved high performance as we pruned large number of insignificant vertices by estimating the incremental influences. For the weighted cascade model, with the increase of the connectivity, the performance of our method slightly degrade, because although the connectivity increased, the edge probabilities decreased and vertices required to update influences will not significantly increased.

7.5 Updates

We evaluated the efficiency on updates. We randomly updated 100K locations and 100K edges (80K insertions and 20K deletions). We evaluated the average time of updating locations and updating graph edges respectively. Table 4 shows the results. We can see that the average time of updating locations was only about 1 microsecond. The average time of updating graph edges was about 0.02 millisecond for

the weighted cascade model. For the trivalency model, the update time on the Weibo dataset was larger than that on other datatets because the Weibo dataset was much denser. These experimental results are also consistent with our update complexity analysis and our method can support location and graph structure updates efficiently.

7.6 Scalability

We evaluated the scalability by varying the numbers of vertices in the query region on the Foursquare datasets using the two models. Figure 13 shows the results. We can see that our method scaled very well and still outperformed IRIE and PMIA by 2-3 orders of magnitude, because even if the graphs were rather large our method did not need to update the influences for many vertices and still pruned large number of insignificant vertices.

8. RELATED WORKS

Influence Maximization in Social Networks. The influence maximization problem was proposed in [6,19]. The two proposed methods are probabilistic and had no bounded influence spread guarantee. Kempe et. al. [10] proposed two discrete influence spread models, Independent Cascade (IC) model and Linear Thresholds model. They proved the influence maximization problem using the two models can be solved by a greedy algorithm with $1-\frac{1}{e}$ approximation ratio.

Since the influence maximization problem is NP-hard, there are many studies on improving the performance. Kimura et. al. [12] used shortest paths to estimate the IC model. Leskovec et. al. [13] developed a "lazy-forward" algorithm which was much better than the simple greedy algorithms. Chen et. al. [2] proposed the PMIA algorithm to solve the influence spread maximization problem using the IC model. The main idea is to estimate the global influence on vertex v by its local maximum influence in-arborescence (PMIIA), which is a tree structure representing the union of maximum influence paths from other vertices to v. The similar idea has been applied to support the linear threshold model [4]. Chen et. al. [3] proposed degree-discount heuristics for a special case of the IC model where all propagation probabilities between vertices are the same. Chen et. al. [5] utilized the community structure to aggregate the features of vertices to reduce the number of vertices they need to check. Kim et. al. [11] proposed independent path algorithm for the IC model, which can be parallelized by OpenMP metaprogramming expressions. Jung et. al. [9] proposed linear equations to approximate the real influence.

Different from existing studies, we study the location-aware influence maximization problem and focus on answering online quires in real-time. We consider how to efficiently calculate the incremental influence of a vertex being selected as a seed. Notice that the influence maximization problem differs from traditional ranking problem in that the influence overlap between top-k seeds should be taken into consideration. Thus the PageRank algorithm [1] cannot be applied directly to the influence maximization.

Learn Influence Spread. There are some studies on learning the influence spread function. Zhang et. al. [20] focused on evaluating the influence between users by considering both their social relationships and geological locations. Different from our problem, they focused on finding top influential events for users while we emphasized on selecting top-k seeds in a spatial region. Goyal et. al. [8] tried to learn a user's influence based on historical data which used propagation trace logs to estimate the influence spread.

9. CONCLUSION

We have studied the location-aware influence maximization problem. We proposed two greedy algorithms with $1-1/e$ approximation ratio. The expansion-based algorithm estimated the upper bound of users' influences and used a best-first method to eliminate the insignificant users. The assembly-based algorithm assembled the precomputed information on small regions to answer a query. We proposed two algorithms with $\epsilon \cdot (1-1/e)$ approximation ratio for any $\epsilon \in (0,1]$. The bound-based algorithm utilized the estimated upper/lower bounds to select top-k seeds. The hint-based algorithm used precomputed hints to identify top-k seeds. Experimental results showed our algorithms achieve high performance while keeping large influence spread and significantly outperform state-of-the-art algorithms.

Acknowledgement. This work was partly supported by NSF of China (61272090 and 61373024), 973 Program of China (2011CB302206), Beijing Higher Education Young Elite Teacher Project (YETP0105), Tsinghua-Tencent Joint Laboratory, "NExT Research Center" funded by MDA, Singapore (WBS:R-252-300-001-490), and FDCT/106/2012/A3.

10. REFERENCES

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.
[2] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *KDD*, pages 1029–1038, 2010.
[3] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *KDD*, pages 199–208, 2009.
[4] W. Chen, Y. Yuan, and L. Zhang. Scalable influence maximization in social networks under the linear threshold model. In *ICDM*, pages 88–97, 2010.
[5] Y.-C. Chen, W.-C. Peng, and S.-Y. Lee. Efficient algorithms for influence maximization in social networks. *Knowl. Inf. Syst.*, 33(3):577–601, 2012.
[6] P. Domingos and M. Richardson. Mining the network value of customers. In *KDD*, pages 57–66, 2001.
[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
[8] A. Goyal, F. Bonchi, and L. V. S. Lakshmanan. A data-based approach to social influence maximization. *PVLDB*, 5(1):73–84, 2011.
[9] K. Jung, W. Heo, and W. Chen. Irie: Scalable and robust influence maximization in social networks. In *ICDM*, pages 918–923, 2012.
[10] D. Kempe, J. M. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.
[11] J. Kim, S.-K. Kim, and H. Yu. Scalable and parallelizable processing of influence maximization for large-scale social networks? In *ICDE*, pages 266–277, 2013.
[12] M. Kimura and K. Saito. Tractable models for information diffusion in social networks. In *PKDD*, pages 259–271, 2006.
[13] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. M. VanBriesen, and N. S. Glance. Cost-effective outbreak detection in networks. In *KDD*, pages 420–429, 2007.
[14] G. Li, J. Hu, K. lee Tan, and J. Feng. Effective location identification from microblogs. In *ICDE*, 2014.
[15] G. Li, Y. Wang, T. Wang, and J. Feng. Location-aware publish/subscribe. In *KDD*, pages 802–810, 2013.
[16] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang. Towards social user profiling: unified and discriminative influence model for inferring home locations. In *KDD*, pages 1023–1031, 2012.
[17] I. R. Misner and V. Devine. The world's best known marketing secret: Building your business with word-of-mouth marketing. In *Bard Press; 2nd Edition edition*, 1999.
[18] J. Nail, C. Charron, and S. Baxter. The consumer advertising backlash. In *Forrester Research:*, 2004.
[19] M. Richardson and P. Domingos. Mining knowledge-sharing sites for viral marketing. In *KDD*, pages 61–70, 2002.
[20] C. Zhang, L. Shou, K. Chen, G. Chen, and Y. Bei. Evaluating geo-social influence in location-based social networks. In *CIKM*, pages 1442–1451, 2012.