Retrieving Regions of Interest for User Exploration

Xin Cao[1] Gao Cong[1] Christian S. Jensen[2] Man Lung Yiu[3]

[1]School of Computer Engineering, Nanyang Technological University, Singapore
xcao1@e.ntu.edu.sg, gaocong@ntu.edu.sg

[2]Department of Computer Science, Aalborg University, Denmark
csj@cs.aau.dk

[3]Department of Computing, Hong Kong Polytechnic University, Hong Kong
csmlyiu@comp.polyu.edu.hk

ABSTRACT

We consider an application scenario where points of interest (PoIs) each have a web presence and where a web user wants to identify a region that contains relevant PoIs that are relevant to a set of keywords, e.g., in preparation for deciding where to go to conveniently explore the PoIs. Motivated by this, we propose the *length-constrained maximum-sum region* (LCMSR) query that returns a spatial-network region that is located within a general region of interest, that does not exceed a given size constraint, and that best matches query keywords. Such a query maximizes the total weight of the PoIs in it w.r.t. the query keywords. We show that it is NP-hard to answer this query. We develop an approximation algorithm with a $(5 + \epsilon)$ approximation ratio utilizing a technique that scales node weights into integers. We also propose a more efficient heuristic algorithm and a greedy algorithm. Empirical studies on real data offer detailed insight into the accuracy of the proposed algorithms and show that the proposed algorithms are capable of computing results efficiently and effectively.

1. INTRODUCTION

The web is undergoing a rapid transformation from being predominantly desktop-based to being used predominantly from mobile devices such as smartphones and tablets. In addition, increasing volumes of geo-textual objects are becoming available on the web that represent Points of Interest (PoIs). Specifically, a *geo-textual object* contains the geo-location of its PoI and describes the PoI by means of text and possibly other elements such as user ratings. Geo-textual objects are available from numerous sources, including business directories such as Google Places for Business and Yahoo! Local, as well as rating and review services such as TripAdvisor. We illustrate geo-textual objects from Google Places for Business in Figure 1.

A range of proposals have been published that aim to return relevant geo-textual objects in response to a user's topical interests and geographical preference. Such proposals may be distinguished according to several dimensions. First, proposals differ w.r.t. *result granularity*: some return individual objects [2, 5, 6, 10, 15, 20],

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 9
Copyright 2014 VLDB Endowment 2150-8097/14/05.

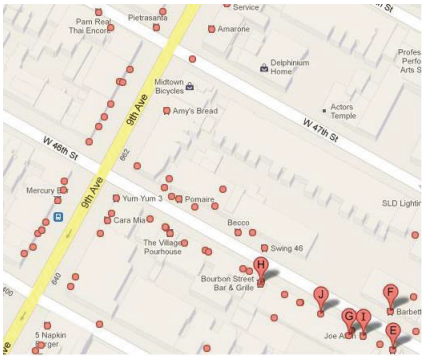


Figure 1: Example of a Region of Restaurants

while others return sets of objects [3, 4, 11, 12, 16]. Second, objects differ w.r.t. the assumed *underlying space*: some consider two-dimensional Euclidean space [2–6, 10–12, 16, 20], while others consider a spatial network [15]. Third, it is often attractive for similar PoIs to *co-locate*, which attracts more customers [7]. For example, cities often have regions with high concentrations of bars, restaurants, or different kinds of shops. Proposals differ in how they account for such *co-location* relationships: some do not take into account the co-location of a PoI with other PoIs when rating the PoI, while others do [2, 4, 11, 16]. Fourth, proposal target different *use cases*: some implicitly assume that the user needs to find the single nearest and best matching PoI [2, 3, 5, 6, 10, 12, 15, 20], while others assume that the user wishes to browse, or explore, several relevant PoIs before selecting one [4, 11, 16].

Along these dimensions, the majority of proposals adopt the single-object result granularity, model the underlying space as Euclidean, disregard co-location, and target non-browsing behavior. In contrast, the proposed length-constrained maximum-sum region query (LCMSR) assumes a set-of-objects result granularity, uses an underlying spatial network, addresses co-location, and targets browsing behavior. In doing so, it aims to be practical in real applications that enable users to explore regions of interests.

Previous proposals by Choi et al. [4], Tao et al. [16], and Liu et al. [11] address the *maximum range sum* problem that accounts for co-location and can be used to support user exploration. These proposals retrieve regions that are either a rectangle with fixed length and width [4, 11, 16] or a circle with fixed radius [4]. The retrieved regions must maximize the total weight of the objects contained in them. To improve on these proposals, we first consider a spatial network setting rather than an underlying Euclidean space, and the regions we retrieve are always connected by road segments. Second, as we believe that the rectangle width and length and the radius parameters are, at best, difficult to set, we eliminate such

shape-related parameters and permits regions of arbitrary shape. The shapes of the retrieved regions depend on the road network topology and thus vary across a road network. To illustrate the issue, consider the following scenario.

Example 1: Suppose that a user wishes to find a region in Manhattan to explore in order to find a restaurant for dinner. Figure 1 shows a map of part of Manhattan, where the balloons and dots represent restaurants. As these are located along the streets, the shape of a region with many restaurants is not easily described by a rectangle or circle of predefined shape and size, as is required by previous work [4, 11]. It is better to define a region as an arbitrarily shaped subgraph of the spatial network. □

In detail, we aim to support users who wish to *explore a region containing multiple PoIs*, located in a spatial network $\mathcal{G}$, each of which is relevant to given query keywords. An LCMSR query Q takes as arguments (i) a set of query keywords $Q.\psi$, e.g., "restaurant" or "coffee," (ii) a length constraint $Q.\Delta$ that indicates how large a region the user is willing to explore, e.g., is willing to walk before selecting an appropriate restaurant, and (iii) a general region of interest $Q.\Lambda$, e.g., Manhattan. The query returns a *region* in the general region of interest $Q.\Lambda$ that does not exceed length constraint $Q.\Delta$ and that contains objects relevant to $Q.\psi$ with the largest total weight. Our proposal is open to different definitions of an object's weight: popularity as measured by numbers of check-ins, user ratings, degree of relevance to the query keywords, etc.

Computing this query calls for the consideration of very large numbers of combinations of objects. This, in combination with arbitrarily shaped result regions, renders the problem of computing LCMSR queries NP-hard.

We first devise a technique that scales node weights into integers using a scaling parameter α. We denote the network with integer node weights by $\mathcal{G}_S$. If we can find a region with the largest scaled weight from $\mathcal{G}_S$, it can be guaranteed that its weight is at least $(1 - \alpha)$ times that of the optimal region. However, finding an optimal region on a graph with integer node weights is still NP-hard.

Based on the scaling technique, we design an approximation algorithm (called APP) with a $(5 + \epsilon)$ approximation ratio, i.e., the region found has at least $1/(5 + \epsilon)$ times the weight of the optimal region. This algorithm first finds a candidate tree from $\mathcal{G}_S$ that has length not exceeding $3Q.\Delta$ (to be explained in Section 4.2.1) and has scaled weight larger than $1/(1 + \beta)$ ($\beta > 0$) times the optimal scaled weight. Then the feasible region with the largest scaled weight is computed from the tree to approximate the optimal region. Finding the feasible region with the largest scaled weight from the candidate tree is also NP-hard, and we devise a pseudo-polynomial time algorithm, utilizing dynamic programming for this.

Extended from this dynamic programming approach, we devise an algorithm that heuristically finds the feasible region with the largest scaled weight from $\mathcal{G}_S$. We represent a region as a five-tuple storing the total length, the original weight, the scaled weight, and the set of all nodes and edges of the region. Each node stores an array of currently enumerated region tuples it belongs to, from which we generate new tuples. The algorithm visits nodes in breadth-first order and processes each edge only once. We call this algorithm TGEN (tuple generation). While it is computationally expensive to enumerate all feasible regions, TGEN's enumeration is bounded and can be done in polynomial time. This algorithm achieves better accuracy and efficiency compared to APP.

Finally, we present a greedy algorithm, Greedy, that works by greedily expanding a candidate region and taking into account both the node weight and the road segment length. This algorithm is efficient, but has low accuracy. We also extend the proposed algorithms, i.e, APP, TGEN, and Greedy, to find the top-k best regions for an LCMSR query.

In summary, our contribution is threefold. First, we define the LCMSR query and show that answering it is NP-hard. Second, we develop an algorithm with a $(5 + \epsilon)$ approximation bound. We also develop a heuristic algorithm inspired by this algorithm and develop a greedy algorithm for answering LCMSR queries. Third, we study the properties of the proposals empirically using two large road network graphs. The results demonstrate that APP is capable of excellent efficiency and accuracy, while TGEN can achieve even better efficiency and accuracy. Greedy performs much worse than other methods in terms of accuracy, but is the fastest.

The rest of the paper is organized as follows. Section 2 formalizes the LCMSR query and establishes its computational complexity. Section 3 presents the indexing used by the algorithms. Section 4 presents the node weight scaling technique and describes the algorithm APP. Section 5 presents the algorithm TGEN. Section 6 covers the algorithm Greedy and how to process top-k LCMSR queries. We report on empirical studies in Section 7. Finally, we cover related work in Section 8 and offer conclusions in Section 9.

2. PROBLEM STATEMENT

Definition 1: Road Network Graph. A road network graph $\mathcal{G} = (V, E, \tau, \lambda)$ consists of a set of nodes V, a set of undirected edges $E \subseteq V \times V$, a distance function $\tau : E \to \mathcal{R}$, and a spatial mapping $\lambda : V \to \mathcal{R}^2$.

Each node $v \in V$ represents a road junction, a dead-end, or is the location of a geo-textual object, in which case it has associated with a text description, i.e., a set of keywords denoted by $v.\psi$. Each edge $e_{ij} = (v_i, v_j) \in E$ represents a road segment connecting nodes v_i and v_j, and its road segment length is denoted by $\tau(v_i, v_j)$. Next, we define a region as follows.

Definition 2: Region. A region R in a road network graph $\mathcal{G}$ is a connected subgraph of $\mathcal{G}$. We denote the vertices and edges in R by $R.\mathcal{V}$ and $R.\mathcal{E}$, respectively.

We need to score a region with respect to an LCMSR query. To do so, we define the score of a region as the sum of the scores of the vertices in the region that represent geo-textual objects. The next step is to score individual geo-textual objects with respect to the query. An LCMSR query contains a set of keywords. For an object o, we can score the object by its text relevance to the query keywords using an information retrieval model (to be introduced in Section 3). Alternatively, the scoring can also take into account both the relevance to the query keywords and the object's other attributes, such as its rating or popularity, by the following strategy: its score will be the object's rating or popularity if it matches the query keywords and zero otherwise. Thus the score of a region represents the popularity of a relevant region. Our proposals are equally applicable to the different scoring methods.

We are now ready to define the LCMSR query. Intuitively, the query computes a region whose score w.r.t. the given query is the largest, with the constraints that it must locate in a region of interest and must not exceed a length constraint.

Definition 3: Length-Constrained Maximum-Sum Region Query (LCMSR). Given a graph $\mathcal{G}$, an LCMSR query Q takes the arguments $\langle \psi, \Delta, \Lambda \rangle$, where ψ is a set of query keywords, Δ is a length constraint, and Λ specifies a rectangular region of interest. The query returns the region R_{opt} in G such that

$$R_{opt} = \arg\max_R \text{Score}(R, Q)$$
$$\text{subject to} \quad \sum_{(v_i,v_j) \in R.\mathcal{E}, v_i, v_j \in Q.\Lambda} \tau(v_i, v_j) \leq Q.\Delta,$$

where $\text{Score}(R, Q)$ computes the score of region R w.r.t. $Q.\psi$.

Given a query Q, a region R that satisfies the query length constraint given by $Q.\Delta$ is called a *feasible region*, and a feasible region with the largest total score is called an *optimal region*. In the rare case that there is more than one optimal region, we return the one with shortest length, with ties broken randomly.

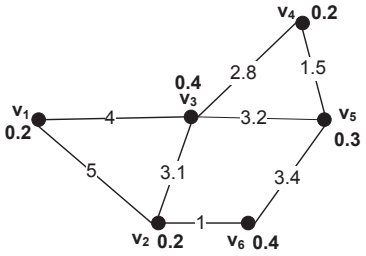


Figure 2: Example of $\mathcal{G}$

Figure 2 shows an example graph $\mathcal{G}$. We assume that the weight (score) of each object w.r.t. the query is already computed. If $Q.\Delta$ is set to 6, the optimal region R has total weight 1.1 and length 5.9, i.e., $R.\mathcal{V} = \{v_2, v_4, v_5, v_6\}$ and $R.\mathcal{E} = \{(v_2, v_6), (v_6, v_5), (v_5, v_4)\}$.

Next, we prove the hardness of the problem.

Theorem 1: Computing the LCMSR query is NP-hard.

PROOF. The computation of the LCMSR query can be proven to be NP-hard by a reduction to it from the well-known k minimum spanning tree (k-MST) problem [8]. Given an instance of the k-MST problem on a graph $G' = (V, E)$, where each edge has a non-negative length, the objective is to find a tree of minimum length that spans k nodes. The decision version of this problem is to decide if given Δ there is a tree with length smaller than Δ spanning k nodes. Now we construct a graph $G = G'$, and each node in G has unit weight 1. It is obvious that this straightforward mapping is polynomial. Given a query $Q = \langle \psi, \Delta, \Lambda \rangle$ where Λ is the whole space, G has a subtree with weight k if and only if G' has a k-node subtree with length smaller than Δ. Hence, we reduce an instance of k-MST to an instance of the problem of answering LCMSR query where each node has unit weight. □

It might be tempting to try to find optimal regions using the following idea: we cluster the objects in a pre-processing step. At query time, we then return the cluster that is most relevant to the query. However, this method has several drawbacks, which renders it unsuitable for LCMSR querying. First, observe that the clustering is query independent and that the objects in each cluster are close in terms of location and text similarity, assuming that we use a spatial-textual similarity measure for the clustering. However, a user is interested in a region with objects that have text descriptions that are relevant to given query keywords, rather than one with the objects being similar to each other. Figure 3 exemplifies this drawback. A clustering algorithm may partition the example graph in Figure 2 into three clusters, as shown with the ellipses. Given a query with keywords $\{t_1, t_2\}$, the best region R has $R.\mathcal{V} = \{v_2, v_3\}$. However, this region is split into two clusters in the clustering-based method. Second, the number of generated clusters and their sizes are fixed. However, it appears infeasible to determine the number and sizes of clusters before querying occurs. Moreover, predefined clusters may not satisfy the length constraint specified in a query.

3. INDEXING STRUCTURE

In this work, we use the text relevance of an object to a given query as the object's weight, as it is used in existing studies of spatial keyword querying [1]. The text relevance of a region R to

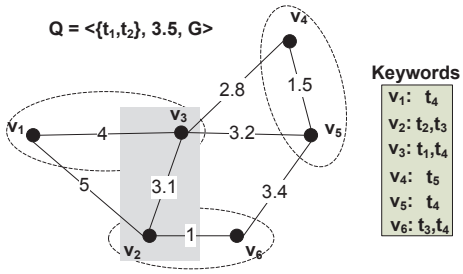


Figure 3: Drawback of Clustering-Based Methods

a query Q is computed using an information retrieval model. We use the vector space model [21] (other models can also be used, e.g., the language model [13]). Specifically, given an object o and a query Q, function $\sigma(o.\psi, Q.\psi)$ computes the similarity between $o.\psi$ and $Q.\psi$ using the vector space model.

$$\sigma(o.\psi, Q.\psi) = \frac{\sum_{t \in Q.\psi \bigcap o.\psi} w_{Q.\psi,t} w_{o.\psi,t}}{W_{Q.\psi} W_{o.\psi}}, \text{ where}$$

$$w_{Q.\psi,t} = \ln(1 + \frac{|D|}{f_t}),\ w_{o.\psi,t} = 1 + \ln(tf_{t,o.\psi}), \quad (1)$$

$$W_{Q.\psi} = \sqrt{\sum_t w^2_{Q.\psi,t}},\ W_{o.\psi} = \sqrt{\sum_t w^2_{o.\psi,t}}$$

Here, f_t is the number of objects whose text description contains the term t, and $tf_{t,o.\psi}$ is the frequency of term t in $o.\psi$; $w_{o.\psi,t}$ captures TF and $w_{Q.\psi,t}$ captures IDF. Then we compute the text relevance between a region R and a query Q as the sum of scores of objects in R: $\text{Score}(R, Q) = \sum_{o \in R.\mathcal{V}} \sigma(o.\psi, Q.\psi)$. With a bit abuse of notation, we denote $\sigma(o.\psi, Q.\psi)$ by σ_o for simplicity.

We use a grid index to organize the geo-textual objects. We partition the entire space according to a uniform grid, and each object is stored in the grid cell that its point location belongs to.

In each grid cell, we maintain an inverted list with the keywords of the objects stored in this cell. Each inverted list has: a) A vocabulary of all distinct words appearing in the description of an object; and b) A postings list for each word t that is a sequence of pairs $(o, wt_o(t))$, where o is the identifier of an object, and $wt_o(t)$ is the term weight of t in the description of object o.

Since we utilize the vector space model to compute the text similarity score, on each node o, for each word t in its description $o.\psi$, we compute the term weight of t with regard to o as: $wt_o(t) = \sigma(o.\psi, t) = \frac{w_{t,t} w_{o.\psi,t}}{w_{t,t} W_{o.\psi}} = \frac{w_{o.\psi,t}}{W_{o.\psi}}$. Hence, given a query Q, utilizing the inverted lists, we first read the postings lists corresponding to the keywords in $Q.\psi$ and then compute the text similarity score $\sigma(o.\psi, Q.\psi)$ between each object o containing some query keywords and the query Q as follows:

$$\sigma(o.\psi, Q.\psi) = \frac{\sum_{t \in Q.\psi \bigcap o.\psi} w_{Q.\psi,t} w_{o.\psi,t}}{W_{Q.\psi} W_{o.\psi}} = \frac{1}{W_{Q.\psi}} \sum_{t \in Q.\psi \bigcap o.\psi} \frac{w_{Q.\psi,t} w_{o.\psi,t}}{W_{o.\psi}} = \frac{1}{W_{Q.\psi}} \sum_{t \in Q.\psi \bigcap o.\psi} w_{Q.\psi,t} wt_o(t), \quad (2)$$

where $w_{Q.\psi,t}$ and $W_{Q.\psi}$ are computed as defined in Equation 1.

The inverted lists may not fit in memory, and we use a disk-based B$^+$-tree to index them for each grid cell. The spatial grid index is maintained in memory since it consumes little memory. We note that other spatial-keyword indices [5] can also be used.

4. APPROXIMATION ALGORITHM APP

Due to the hardness of answering LCMSR queries, it is hard to design algorithms that can find the optimal region efficiently.

We devise a method that scales node weights (scores) into integers in Section 4.1. This enables us to design an algorithm APP with guaranteed approximation bounds as presented in Section 4.2.

4.1 Scaling of Node Weights

Specifically, we define a scaling factor $\theta = \alpha\sigma_{max}/|V_Q|$, where α is the scaling parameter, V_Q denotes the set of nodes within the query region $Q.\Lambda$, and σ_{max} is the maximum weight score among all nodes in V_Q. Next, for each location v, its weight score σ_v is scaled to $\hat{\sigma}_v = \lfloor \sigma_v/\theta \rfloor$. We call the graph with scaled node weights the **scaled graph** and denote it by $\mathcal{G}_S$.

Example 2: Take the graph in Figure 2 as an example. We set α to 0.15, and let $Q.\Lambda$ cover the whole graph. Therefore, $\theta = \alpha\sigma_{max}/|V_Q| = 0.15 * 0.4/6 = 0.01$, which means that the weight of each node is scaled to 100 times its original value. □

We use the following definition to represent a region in the scaled graph $\mathcal{G}_S$.

Definition 4: Region Tuple. Each region R is stored as a 5-tuple $T = (l, s, \hat{s}, \mathcal{V}, \mathcal{E})$, where l is the total length of all road segments in R, s is the original weight, $\hat{s}$ is its scaled weight, $\mathcal{V}$ is a set storing all the nodes in R, and $\mathcal{E}$ is the set of all edges in R.

Example 3: Take the graph in Figure 2 as an example, and suppose that we scale the weight 100 times as we do in Example 2. Given a region R with $R.\mathcal{V} = \{v_2, v_4, v_5, v_6\}$ and $R.\mathcal{E} = \{(v_2, v_6), (v_6, v_5), (v_5, v_4)\}$, its region tuple is $T = (5.9, 1.1, 110, R.\mathcal{V}, R.\mathcal{E})$. □

Next, we prove that given an LCMSR query, if the region with the largest scaled weight can be found, its total weight is at least $(1-\alpha)$ times that of the optimal region.

Given a query Q, we denote the region with the largest scaled weight in $\mathcal{G}_S$ by R^S_{opt}, and the optimal region in the original graph $\mathcal{G}$ by R_{opt}. We have the following theorem:

Theorem 2: $R^S_{opt}.s \geq (1-\alpha)R_{opt}.s$.

PROOF. From $\hat{\sigma}_v = \lfloor \sigma_v/\theta \rfloor$, we know that $\sigma_v - \theta < \theta\hat{\sigma}_v \leq \sigma_v$. Therefore, $R^S_{opt}.s = \sum_{v \in R^S_{opt}.\mathcal{V}} \sigma_v \geq \sum_{v \in R^S_{opt}.\mathcal{V}} \theta\hat{\sigma}_v$. Since R^S_{opt} is the best region in the scale graph, we know that $R^S_{opt}.\hat{s} = \sum_{v \in R^S_{opt}.\mathcal{V}} \hat{\sigma}_v \geq R_{opt}.\hat{s}$, and thus,

$$R^S_{opt}.s \geq \theta R_{opt}.\hat{s} = \sum_{v' \in R_{opt}.\mathcal{V}} \theta\hat{\sigma}_{v'} \geq \sum_{v' \in R_{opt}.\mathcal{V}} (\sigma_{v'} - \theta)$$
$$\geq \sum_{v' \in R_{opt}.\mathcal{V}} \sigma_{v'} - |V_Q|\theta = \sum_{v' \in R_{opt}.\mathcal{V}} \sigma_{v'} - \alpha\sigma_{max}$$

Because $\sum_{v' \in R_{opt}.\mathcal{V}} \sigma_{v'} \geq \sigma_{max}$, we can conclude that:

$$R^S_{opt}.s \geq (1-\alpha) \sum_{v' \in R_{opt}.\mathcal{V}} \sigma_{v'} = (1-\alpha)R_{opt}.s.$$

We complete the proof. □

However, even on the scaled graph $\mathcal{G}_S$, finding the region with the largest scaled weight remains NP-hard. The proof of Theorem 1 can still be used to prove this. Therefore, it is also challenging to find a region that approximates the optimal region from $\mathcal{G}_S$.

4.2 Algorithm Details of APP

We proceed to describe the approximation algorithm based on the node weight scaling technique introduced in Section 4.1. This algorithm runs polynomially with $1/\alpha$, $Q.\Delta$, and the number of nodes and edges in $Q.\Lambda$, and has an approximation ratio of $(5+\epsilon)$.

4.2.1 Algorithm Overview

Theorem 2 shows that the feasible region with the largest scaled weight can achieve an approximation ratio of $(1-\alpha)$, where α can be an arbitrarily small value. However, finding the feasible region with the largest scaled weight is NP-hard. This inspires us to find a feasible region to approximate the region with the largest scaled weight, and then based on Theorem 2 we can further approximate the original optimal weight.

Lemma 1: If we can find a feasible region R such that $R.\hat{s} > 1/(1+\beta)R^S_{opt}.\hat{s}$, where β is an arbitrarily small value, an LCMSR query can be approximately answered with an ratio of $(1+\epsilon)$.

PROOF. From Theorem 2 we know $\theta\hat{\sigma}_v \leq \sigma_v$, and thus $R.s = \sum_{v \in R.\mathcal{V}} \sigma_v \geq \sum_{v \in R.\mathcal{V}} \theta\hat{\sigma}_v = \theta R.\hat{s}$. Because $R.\hat{s} > 1/(1+\beta)R^S_{opt}.\hat{s}$, we have $R.s \geq \theta R.\hat{s} \geq 1/(1+\beta)\theta R^S_{opt}.\hat{s}$. Theorem 2 shows that $\theta R^S_{opt}.\hat{s} \geq (1-\alpha)R_{opt}.s$, and we obtain $R.s > (1-\alpha)/(1+\beta)R_{opt}.s$. Because both α and β can be arbitrarily small, $(1-\alpha)/(1+\beta)$ is equivalent to $(1+\epsilon)$, and the proof is completed. □

Before describing how to find such a region satisfying the requirement in Lemma 1, we first introduce the node-weighted k minimum spanning tree problem (denoted by k-MST, extended from the k minimum spanning tree problem [14]), which is defined over a graph with integer node weights. Given a node weight constraint X, the problem aims to find the tree with the smallest length such that the nodes it spans have total weight at least X. The k-MST problem is NP-hard, but there exists efficient approximation algorithms with performance guarantees for the problem. Note that the k-MST problem is different from the problem of answering the LCMSR query, where the constraint is on the total edge length and the aim is to maximize the total node weight.

We aim to utilize algorithms for k-MST to find a feasible region R as described in Lemma 1. Given a weight constraint X, suppose that we have a k-MST solver for answering the k-MST problem exactly, we are able to find the region with the smallest length with weight at least X. We first introduce the following lemma before we describe how the solver can be utilized to find a region with scaled weight larger than $1/(1+\beta)R^S_{opt}.\hat{s}$.

Lemma 2: Given a weight constraint X, if the exact k-MST solver returns a tree T_C with length no larger than $Q.\Delta$, then $X \leq T_C.\hat{s} \leq R^S_{opt}.\hat{s}$. If $T_C.l > Q.\Delta$, we have $T_C.\hat{s} \geq X > R^S_{opt}.\hat{s}$.

PROOF. $X \leq T_C.\hat{s}$ is always true, because according to the definition of k-MST, T_C is the tree with the smallest length among all trees with node weight at least X.

The first statement is obvious. $R^S_{opt}.\hat{s}$ is the largest scaled weight given length constraint $Q.\Delta$, and thus $T_C.\hat{s}$ cannot exceed $R^S_{opt}.\hat{s}$.

The second statement can be proved by contradiction. If $X \leq R^S_{opt}.\hat{s}$, because T_C has the smallest length among all trees with weight at least X, we know $T_C.l \leq R^S_{opt}.l$. Since R^S_{opt} is a feasible region, $T_C.l \leq R^S_{opt}.l \leq Q.\Delta$, thus leading to a contradiction. □

According to Lemma 2, in order to find a feasible region R satisfying $R.\hat{s} \geq 1/(1+\beta)R^S_{opt}.\hat{s}$, or equivalently, $R^S_{opt}.\hat{s} \leq (1+\beta)R.\hat{s}$, we only need to find a feasible region R, such that under the weight constraint $(1+\beta)R.\hat{s}$, the tree returned by the solver has length larger than $Q.\Delta$. Here the challenge is that we do not know the scaled weight of R. Our idea is to estimate the lower and upper bounds for the weight of R, and then use binary search to find such a region by repeatedly invoking the k-MST solver.

Let X denote the weight constraint under which the k-MST solver returns the region R satisfying Lemma 1. Figure 4 illustrates the

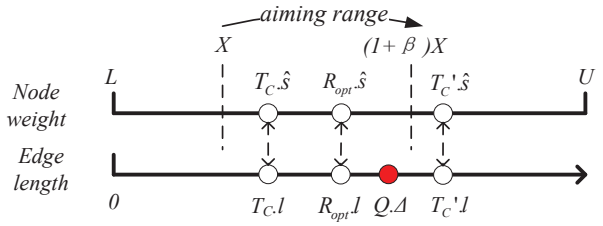


Figure 4: Binary Search Sketch

objective of the binary search. We aim to find a weight X, such that the tree T_C returned by the k-MST solver under weight constraint X has length no larger than $Q.\Delta$ and the tree T'_C returned under constraint $(1+\beta)X$ has length larger than $Q.\Delta$.

The process can be described as follows. Based on the lower and upper bounds (to be presented in Lemma 5) of the scaled weight of R, we initially set X to be the middle of the two bounds. We invoke the k-MST solver with X as the argument. If the tree returned has length larger than $Q.\Delta$, we need to decrease the value of X and recursively invoke the k-MST solver. When the tree returned has length no larger than $Q.\Delta$, we know the tree is a feasible region. Then we invoke the k-MST solver again with weight $(1+\beta)X$ as the argument. If the tree T'_C returned has length smaller than $Q.\Delta$, we know $(1+\beta)X \leq R^S_{opt}.\hat{s}$ (according to Lemma 2), and thus we need to increase the value of X. The binary search stops until X reach a value such that the tree T_C returned under constraint X has length not exceeding $Q.\Delta$ and the tree T'_C returned under constraint $(1+\beta)X$ has length larger than $Q.\Delta$. It can be proved that the tree T_C is the feasible region we aim to find. The following example illustrates the binary search procedure.

Example 4: We denote the lower bound by L and the upper bound by U, and we set $Q.\Delta$ to 100 and β to 0.5. The binary search steps are shown in Table 1.

Steps	L	U	X	$T_C.l$	1.5X	$T'_C.l$
0	10	1000	*	*	*	*
1	10	1000	505	150	*	*
2	10	505	258	77	387	92
3	258	505	381	90	572	158

Table 1: Example of the Binary Search Procedure

We first initialize L and U. In step 1, by invoking the k-MST solver, we obtain a tree with length larger than $Q.\Delta$. Thus, we set the upper bound to X in the next step, and we get a tree satisfying the length constraint. However, after invoking the k-MST solver again with weight $(1+\beta)X$ as the argument, the tree obtained still has length smaller than $Q.\Delta$. We set the lower bound to X in step 3, and we now we obtain a tree satisfying Lemma 1. □

Lemma 3: Such a tree T_C has an approximation ratio of $(1+\epsilon)$.

PROOF. According to Lemma 2, when the binary search terminates, we know that $X \leq R^S_{opt}.\hat{s} < (1+\beta)X$. In addition, since $T_C.\hat{s} \geq X$, we know $T_C.\hat{s} > 1/(1+\beta)R^S_{opt}.\hat{s}$. According to Lemma 1, $T_C.s > 1/(1+\epsilon)R_{opt}.s$. □

Unfortunately, the k-MST problem (also the node-weighted version) is NP-hard [14], and there exists no efficient exact algorithm for it. We adopt the 3-approximation algorithm (denoted by kMST) proposed by Garg [8], a state-of-art algorithm, for solving the node-weighted k-MST problem. Garg's algorithm approximately solves the k-MST problem by applying the GW-algorithm [9], which is a general approximation technique for constrained forest problems. kMST is able to find a subtree with length at most 3 times that of the optimal tree satisfying the node weight constraint. Since we use the 3-approximation algorithm kMST instead of an exact algorithm, we need to modify the binary search process a bit. We have the following lemma.

Lemma 4: Given a weight constraint X, if kMST returns a tree T_C with length no larger than $3Q.\Delta$, and kMST returns a tree T'_C with length larger than $3Q.\Delta$ under constraint $(1+\beta)X$, we have $T_C.\hat{s} > 1/(1+\beta)R^S_{opt}.\hat{s}$.

PROOF. It is obvious that $X \leq T_C.\hat{s}$ due to the definition of k-MST. We denote the tree with the smallest length under the constraint $(1+\beta)X$ by T^O. Because T'_C is returned by a 3-approximation algorithm of k-MST, we know $3Q.\Delta < T'_C.l \leq 3T^O.l$, and thus $T^O.l > Q.\Delta$. According to Lemma 2, we can derive $T^O.\hat{s} \geq (1+\beta)X > R^S_{opt}.\hat{s}$. Therefore, we have $T_C.\hat{s} > 1/(1+\beta)R^S_{opt}.\hat{s}$. □

According to Lemma 4, during the binary search, we now compare the returned tree with $3Q.\Delta$ rather than $Q.\Delta$.

We show the basic steps of the approximation algorithm, called APP, in Figure 5 and Algorithm 1. Given a query Q, we first scale the node weights, and we obtain $\mathcal{G}_S$. Then we invoke the binary search method (to be detailed in Section 4.2.2) to find the candidate tree T_C satisfying Lemma 4. Finally, we find the feasible region with the largest scaled weight from T_C to approximate the optimal result in $findOptTree()$ (to be detailed in Section 4.2.3). This algorithm has an approximation ratio of $(5+\epsilon)$, as will be shown in Section 4.2.4.

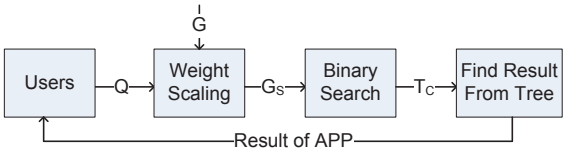


Figure 5: Basic steps of APP

Algorithm 1: APP(Q, $\mathcal{G}$)

```
1 G_S ← the scaled graph w.r.t. Q;
2 T_C ← binarySearch(Q, G_S);
3 if T_C.l < Q.Δ then return T_C;
4 else return findOptTree(T_C);
```

4.2.2 Function binarySearch()

First, we obtain the upper and lower bounds for the optimal scaled weight of a feasible region (i.e., $R^S_{opt}.\hat{s}$) based on the following lemma, which are used as the binary search range.

Lemma 5: Given an LCMSR query Q, the lower bound of the optimal scaled weight of a feasible region is $\lfloor |V_Q|/\alpha \rfloor$, and the upper bound is $|V_Q|\lfloor |V_Q|/\alpha \rfloor$.

PROOF. The maximum scaled weight of a node in $Q.\Lambda$, i.e., $\hat{\sigma}_{max} = \lfloor |V_Q|/\alpha \rfloor$. It must be smaller than the best scaled weight of a region and is used as the lower bound. The region to be found must be formed by a subset of nodes in V_Q, and thus we can derive an upper bound as $|V_Q|\lfloor |V_Q|/\alpha \rfloor$. □

The binary search process is presented in binarySearch(). We first establish lower and upper bounds for the optimal scaled weight of a region according to Lemma 5 (line 1). Next, if the length of T_C exceeds $3Q.\Delta$, we set the upper bound as X (line 5). Otherwise, we check if T'_C has length exceeding $3Q.\Delta$. If so we stop the search and otherwise we set the lower bound L as X (lines 7–9).

4.2.3 Function findOptTree()

The last step of APP (line 3) is to find the feasible region with the largest scaled weight from T_C. However, this is still not an easy problem.

Function binarySearch(Q, $\mathcal{G}_S$)

```
1  L ← ⌊|V_Q|/α⌋; U ← |V_Q|⌊|V_Q|/α⌋;
2  while true do
3      X ← (L + U)/2;
4      T_C ← kMST(X) ;   // call Garg's algorithm
5      if T_C.l > 3Q.Δ then U ← X;
6      else
7          T'_C ← kMST((1 + β)X);
8          if T'_C.l > 3Q.Δ then break;
9          else L ← X;
10 return T_C;
```

Theorem 3: Finding the region with the largest scaled weight whose length does not exceed $Q.\Delta$ from T_C is NP-hard.

PROOF. This problem can be reduced from the well-known knapsack problem. In the knapsack problem, there are m objects and a weight limit Δ, and each object o_i has a value c_i and weight w_i. The aim is to find a set of objects such that the total weight does not exceed Δ and the total value is as large as possible. We construct a starlike tree in which each node is connected only with the center node. Each object o_i in the knapsack problem corresponds to a non-center node v_i in T_C, its value c_i corresponds to the weight of the node v_i, and its weight w_i corresponds to the length of the edge between the node v_i and the center node. The weight limit Δ in the knapsack problem corresponds to the query length constraint $Q.\Delta$. It is obvious that this straightforward mapping is polynomial, thus completing the proof. □

Two observations inspire us to design a pseudo-polynomial time algorithm based on dynamic programming for this problem. First, the weights of nodes are scaled into integers. Second, since we are finding the region with the largest scaled weight from a tree, we can fix a node in T_C as the root, and each node in T_C has a level according to the number of edges in the path from it to the root. Then each region R can be viewed as rooted at the node $v \in R.\mathcal{V}$ that has the highest level in T_C. Based on the two observations, we have the following lemma.

Lemma 6: Assume that region R rooted at v has the smallest length among all regions with the scaled weight $R.\hat{s}$ rooted at v. Given any node $v_i \in R.\mathcal{V}$, the sub-region rs of R rooted at v_i must also have the smallest length among all regions with scaled weight $rs.\hat{s}$ rooted at v_i.

PROOF. If there is another region rs' rooted at v_i with scaled weight $rs.\hat{s}$, we can replace rs with rs' in R to obtain a region R' rooted at v that has smaller length than does R. However, this contradicts that R has the smallest length among all regions rooted at v with weight $R.\hat{s}$. □

Lemma 6 lays the basis of the dynamic programming approach, and it implies that on each node v, we only need to keep the tuple with the smallest length for each scaled weight value, and thus the number of regions rooted at v can be bounded. Specifically, we use the following structure to organize the region tuples on each node.

Definition 5: Region Tuple Array. Each node v in T_C maintains an array of region tuples rooted at v, denoted by $v.tp$, and $v.tp[S]$ stores the tuple representing the feasible region with the smallest length among all regions rooted at v having scaled weight S.

Using this structure, we can compute the node region tuple arrays in a bottom-up manner. We treat the nodes with only one neighbor as the leaf nodes, and the tree is constructed consequently. First, each leaf node v_l initializes its tuple array using the region formed by itself, i.e., $v_l.tp[\hat{\sigma}_{v_l}] = (0, \sigma_{v_l}, \hat{\sigma}_{v_l}, \{v_l\}, \emptyset)$. Then, the tuple arrays of leaf nodes are used to compute the tuple arrays of nodes in the higher level. This step is stopped until we reach the root node of T_C, and we can select the feasible region with the largest scaled weight from tuple arrays of all nodes in $T_C.\mathcal{V}$. Next, we proceed to describe how to compute the tuple array of a parent node when the tuple array of each of its child node has been computed.

If the parent node v_p has only one child node v_c, it is quite simple. We only need to combine each region tuple in $v_c.tp$ with v_p to obtain new regions, and they are stored in $v_p.tp$. Specifically, for each tuple T in $v_c.tp$, we have $v_p.tp[T.\hat{s} + \hat{\sigma}_{v_p}] = (T.l + \tau(v_p, v_c), T.s + \sigma_{v_p}, T.\hat{s} + \hat{\sigma}_{v_p}, T.\mathcal{V} \cup \{v_p\}, T.\mathcal{E} \cup \{(v_p, v_c)\})$.

However, it is more complicated when parent node v_p has multiple child nodes. The idea is to process the child node one by one. Assume that v_p has m child nodes, each of which is represented by v_{c_k} where $1 \le k \le m$. When computing $v_p.tp[S]$, we denote the region obtained after processing the first k child nodes of v_p by $T[v_p, S, k]$. When the first $(k-1)$ child nodes have been processed and the k^{th} child node is to be processed, the length of $T[v_p, S, k]$ can be computed by the following lemma.

Lemma 7: Denoting $T[v_p, S - \hat{s}, k-1]$ by t_1 and $v_{c_k}.tp[\hat{s}]$ by t_2,

$$T[v_p, S, k] = \min(T[v_p, S, k-1], \arg \min_{1 \le \hat{s} < S} (t_1.l + t_2.l + \tau(v_p, v_{c_k}), t_1.s + t_2.s, S, t_1.\mathcal{V} \cup t_2.\mathcal{V}, t_1.\mathcal{E} \cup t_2.\mathcal{E} \cup \{(v_p, v_{c_k})\})).$$

PROOF. After the first $(k-1)$ child nodes of v_p are processed, for each value S, $T[v_p, S, k-1]$ stores the region with the smallest length among all regions rooted at v_p constituted by nodes rooted at the $(k-1)$ nodes and v_p. In order to compute the region $T[v_p, S, k]$, there are only two possibilities: either it contains no nodes rooted at v_{c_k}, or it contains some nodes rooted at v_{c_k}. In the first case, $T[v_p, S, k] = T[v_p, S, k-1]$. In the second case, we need to consider all possible combinations that can generate a region with scaled weight S. Finally, we select the feasible region with the smallest length and store it in $T[v_p, S, k]$. □

After all m child nodes of v_p are processed, for each scaled weight value S we set $v_p.tp[S]$ as $T[v_p, S, m]$. After the region tuple arrays of all nodes are computed, we find the region with the largest scaled score from these arrays and return it as the result.

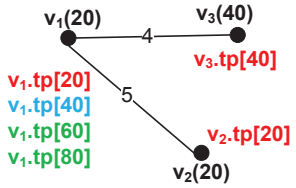


Figure 6: Example of Updating Tuple Arrays

Example 5: Assume that the tree T_C returned is as shown in Figure 6. Nodes v_2 and v_3 are treated as leaf nodes and v_1 is the root. First, we have $v_1.tp[20]$, $v_2.tp[20]$ and $v_3.tp[40]$. When computing $v_1.tp$, we first consider node v_2, and we obtain $v_1.tp[40] = (5, 0.4, 40, \{v_1, v_2\}, \{(v_1, v_2)\})$. Next, v_3 is considered, and $v_1.tp$ has two tuples now. Both are combined with $v_3.tp[40]$, and $v_1.tp[60]$ and $v_1.tp[80]$ are generated. □

The function is detailed in Function findOptTree(). Initially, we create a region tuple for each node v in tree T_C using the node itself, and the tuple is used to initialize $v.tp[\hat{\sigma}_v]$. For nodes that have only one neighbor in T_C, we insert them into $nodeQ$, and these nodes are treated as the leaf nodes in T_C (lines 3–5). While there is only one node in $nodeQ$, it means that the root node is reached, and thus the algorithm can be terminated. Otherwise, we

select a node v from $nodeQ$ and use its tuple array to compute the tuple array of its parent node (lines 7–12). Each node v in $nodeQ$ has only one neighbor v_n which is treated as the parent node of v. We then use $v.tp$ and $v_n.tp$ to generate new tuples and to update $v_n.tp$ according to Lemma 7 (line 9). We also update $bestR$ using the newly generated tuple with the largest scaled score (line 10). After v is processed, we remove it from T_C, and if v_n now has only one neighbor, we insert it into $nodeQ$ (lines 11–12). Finally, we return $bestR$, the region with the largest scaled weight in T_C, as the result to approximately answer the given query.

Function findOptTree(T_C)

```
1  initialize a queue nodeQ;
2  bestR ← null;
3  foreach node v in T_C do
4  |  v.tp[σ̂_v] ← (0, σ_v, σ̂_v, {v}, ∅);
5  |  if v has only one neighbor then nodeQ.enqueue(v);
6  while nodeQ contains more than one node do
7  |  v ← nodeQ.dequeue();
8  |  v_n ← the neighbor node of v in T_C;
9  |  generate new tuples using v.tp and v_n.tp to update v_n.tp;
10 |  update bestR using the new region with the largest weight;
11 |  remove v from T_C;
12 |  if v_n has only one neighbor then nodeQ.enqueue(v_n);
13 return bestR;
```

4.2.4 Approximation Ratio and Complexity

Approximation ratio. In order to establish the approximation ratio of Algorithm APP, we first introduce the following lemma.

Lemma 8: A tree Tr with total weight W and total length L can be split into three edge-disjoint subtrees, and there always exists a subtree with length no more than $1/3L$ and weight at least $1/5W$.

PROOF. It is obvious that no matter how Tr is split, there must exist one subtree with length no more than $1/3L$. The idea is to show that we can split Tr into three subtrees and each has weight larger than $1/5W$. We construct those subtrees to complete the proof. If there is a node whose weight exceeds $1/5W$, then it is the subtree we want. Otherwise, no node in the subtree has weight more than $1/5W$. Suppose a subtree t_1 has weight smaller than $1/5W$. We can move some nodes to it from subtree t_2 connected to t_1, until t_1 has weight between $1/5W$ and $2/5W$. Thus, the total weight of t_2 and t_3 exceeds $3/5W$. If either one has weight smaller than $1/5W$, the other one must have weight larger than $2/5W$, and we can move some nodes from the one with larger weight to the one with smaller weight to make sure that all subtrees have weight larger than $1/5W$. □

Theorem 4: APP is a $(5+\epsilon)$-approximation algorithm.

PROOF. When the binary search terminates, we have $R^S_{opt}.\hat{s} < (1+\beta)T_C.\hat{s}$ according to Lemma 4 . Lemma 8 tells that T_C has a subtree satisfying the length constraint $Q.\Delta$ with scaled node weight larger than $1/5T_C.\hat{s}$. The region returned by APP is the best tree from T_C (denoted by R^S_{app}), and thus $R^S_{app}.\hat{s} \geq 1/5T_C.\hat{s} > 1/(5+5\beta)R^S_{opt}.\hat{s}$.

Because $R^S_{app}.s \geq \theta R^S_{app}.\hat{s}$, we can obtain $R^S_{app}.s > 1/(5+5\beta)\theta R^S_{opt}.\hat{s}$. Recall the proof of Theorem 2, it is true that $\theta R^S_{opt}.\hat{s} = \sum_{v \in R^S_{opt}} \theta\hat{\sigma}_v \geq (1-\alpha)R_{opt}.s$.

Hence, we can conclude that $R^S_{app}.s > (1-\alpha)/(5+5\beta)R_{opt}.s$. Both α and β can be arbitrarily small, and thus APP is a $(5+\epsilon)$-approximation algorithm. □

Complexity. We first analyze the complexity of function findOptTree(). The nodes of a region returned is a subset of $T_C.\mathcal{V}$. In addition, since the optimal region must be a tree, there are at most $\lfloor Q.\Delta/d_{min} \rfloor$ edges contained in a region (d_{min} is the minimum edge length in $Q.\Lambda$). Hence, the maximum number of nodes in a region can be estimated by $\min(|T_C.\mathcal{V}|, \lfloor Q.\Delta/d_{min} \rfloor + 1)$, and we denote this value as N_{max}. We know that the largest score of a node in $Q.\Lambda$ is σ_{max}. Therefore, the maximum number of region tuples in a node's tuple array is bounded by $N_{max}\lfloor \sigma_{max}/\theta \rfloor = N_{max}\lfloor |V_Q|/\alpha \rfloor$, and we denote this value by T_{max}.

In the function each edge in T_C is only processed once. When an edge (v_i, v_j) is processed, tuples in $v_i.tp$ and tuples in $v_j.tp$ are combined to generate new tuples. Hence, the worst complexity of this step is $O(|T_C.\mathcal{E}|T^2_{max})$. The binary search takes at most $O(\log_{1+\beta}|V_Q|)$ iterations to find the subtree T_C. kMST runs polynomially with $|V_Q|$ and $|E_Q|$ [8]. Therefore, the algorithm runs polynomially with $|V_Q|$, $|E_Q|$, $Q.\Delta$, and $1/\alpha$.

5. TUPLE GENERATION ALGORITHM

In Section 4.2.3, we devise a dynamic programming method to find the feasible region with the largest scaled weight from a tree, in which we compute the region tuple arrays in a bottom-up manner. Unfortunately, we cannot apply this method in $\mathcal{G}_S$ because of the cycles in the graph. However, we can extend this method to design an algorithm that heuristically finds a region to approximate the feasible region R^S_{opt} that has the largest scaled weight from graph $\mathcal{G}_S$. We denote this algorithm by TGEN (tuple generation).

Recall that in Function findOptTree(), each edge in the candidate tree is only processed once, and each node stores an array of regions rooted at it. In order to find R^S_{opt}, we can do as follows. We also process each edge only once to generate new regions, and during the enumeration each node stores all enumerated feasible regions it belongs to in a list. When processing an edge (v_i, v_j), each enumerated region containing v_i and each enumerated region containing v_j are combined with this edge to form a new region. If it satisfies the length constraint, we store it in the region list of each node it contains.

In this method, after m edges are processed, each feasible region R formed by these edges is stored on each node contained in R. When we process a new edge e, since all the enumerated feasible regions from the region lists of the two nodes of e are combined, all possible feasible regions formed by the first $m+1$ edges are generated and stored. Therefore, after all edges are processed, any feasible region cannot be missed, and the correctness of the algorithm is assured.

The problem of this exact approach is that too many region tuples need to be enumerated and stored. Given a node, the number of regions it belongs to is exponential with the maximum number of nodes in a feasible region (bounded by $Q.\Delta$). Hence the approach is computational prohibitive.

Recall that in findOptTree(), we only keep one tuple that has the smallest length for a scaled weight rooted at each node. This inspires us to adopt the node weight scaling technique and the concepts of region tuples and tuple arrays to design TGEN. The idea is that, given a query Q, we scale the node weights as introduced in Section 4.1, and the regions are stored as tuples as defined in Definition 4. We use the structure in Definition 6, which is similar to that defined in Definition 5, to store the tuples on each node.

Definition 6: Explored Region Tuple Array. Each node v in $Q.\Lambda$ maintains an array of regions containing v, denoted by $v.tp$. $v.tp[S]$ stores the region tuple with the smallest length among all enumerated region tuples containing v having scaled weight S.

Notice that the explored region tuple array is different from the region tuple array in Definition 5 where a node only stores the regions rooted at it. The analysis established on the maximum number of regions a node needs to store in Section 4.2.4 is still applicable here, i.e., the number of tuples stored on a node can be bounded by T_{max} in TGEN.

Using the tuple arrays as defined in Definition 6, we describe the process of TGEN as follows. Given a query Q, we select any unprocessed node in $Q.\Lambda$, denoted by v_0. We then traverse the remaining nodes in $Q.\Lambda$ in a breadth-first order starting from v_0 to generate new regions. When we reach a node v_i, we process each of its incident edges that has not yet been processed, and this ensures that each edge is processed only once. If all the nodes reachable from v_0 are processed and some nodes are yet to be processed, we again randomly select one of them and repeat the above steps. Finally we return the best region from the explored tuple arrays.

When processing an edge (v_i, v_j), each region from $v_i.tp$ and each region from $v_j.tp$ are combined by this edge to generate a new region T. If T is feasible, we update the tuple array of each node v in $T.\mathcal{V}$: If $v.tp[T.\hat{s}].l > T.l$, we update $v.tp[T.\hat{s}]$ as T. Because we discard some regions in this step, it is possible that the optimal region is missed. Since now we are finding the region from a graph instead of a tree, it is possible that some regions containing cycles are generated, which are not necessary. We have the following lemma to avoid generating such unnecessary regions.

Lemma 9: When processing an edge (v_i, v_j), if a tuple T_i^m from $v_i.tp$ and a tuple T_j^n from $v_j.tp$ share common nodes, they do not need to be combined by this edge.

PROOF. In this case, the tuple combining T_i^m and T_j^n by (v_i, v_j) must contain a cycle. Any region contains a cycle cannot be the result, since there must exist another region with the same set of nodes has smaller length. Thus, it can be discarded safely. □

We can process the edges in other orders (e.g., the edges can be processed in ascending order of their lengths). However, after testing several orders, we observe that the accuracy only varies slightly while the order we adopt yields better efficiency. This is because: a) we do not need to spend extra time on ordering edges; and b) after a node is processed, i.e., all its incident edges are processed, we can discard its node tuple array. In later steps, if it is contained in a newly generated tuple, it is not necessary to update its tuple array. As shown in the experimental study, TGEN can achieve better accuracy than does APP.

The method is presented in Algorithm 2We first obtain an unprocessed node and insert it into $nodeQ$ (line 3–4). Lines 7-14 describes how an edge is processed as discussed above. We use a new generated tuple T to update the tuple array of each node in $T.\mathcal{V}$ (lines 12–14). Finally, we return $bestR$ as the result.

Complexity. Each edge in $Q.\Lambda$ is only processed once. When an edge (v_i, v_j) is processed, TGEN combines all the tuples from $v_i.tp$ and $v_j.tp$. Hence, TGEN has the complexity $O(|E_Q|T_{max}^2)$.

Algorithm 2: TGEN(Q, $\mathcal{G}_S$)

```
1  Initialize a queue nodesQ; bestR ← null;
2  while there exist unprocessed nodes do
3      v0 ← an unprocessed node;
4      nodesQ.enqueue(v0);
5      while nodesQ is not empty do
6          vi ← nodesQ.dequeue();
7          for each unvisited edge (vi, vj) do
8              if τ(vi, vj) > Q.Δ then continue;
9              nodesQ.enqueue(vj);
10             generate new tuples using vi.tp and vj.tp;
11             update bestR using the new region tuples;
12             foreach new tuple T do
13                 foreach unprocessed node v in T.V do
14                     if T.l < v.tp[T.ŝ].l then v.tp[T.ŝ] ← T;
15 return bestR;
```

6. GREEDY METHOD AND EXTENSIONS

6.1 Greedy Method

We propose a approach that expands the region greedily. We use R_C to represent the currently explored region. We initialize R_C using the node with the largest weight in $Q.\Lambda$. Then in each following step we greedily select a node connected to R_C to expand R_C. Since we are trying to maximize the total weight of a region satisfying the query length constraint, we can have two methods to select the next node greedily: 1) we select a node with the largest weight; or 2) we select a node such that the edge connecting it with the explored region has the smallest length. This step is repeated until no more nodes can be added to the region (i.e., the total length of the region exceeds $Q.\Delta$ if including more nodes). We denote this algorithm as Greedy.

However, both methods have obvious drawbacks. The first method only considers the weight of a region. It performs poorly when the edge connecting the selecting node and the region has a large length value. The second method takes into account only the road segment length, and thus its performance is poor if the selected node is not relevant to the query.

To avoid the drawbacks of the two aforementioned methods, we devise a combined model, in which both the weight and the road segment length are taken into account. A parameter μ is used to balance the importance of them when greedily selecting a node to extend the explored region R_C.

Specifically, given a query Q and an currently explored region R_C, for each node v_i connected to R_C via v_j (i.e., $v_j \in R_C.\mathcal{V}$ and $v_i \notin R_C.\mathcal{V}$), we compute the ranking score for v_i as follows:

$$\rho(v_i) = \mu(1 - \tau(v_i, v_j)/\tau_{max}) + (1 - \mu)\sigma_{v_j}/\sigma_{max},$$

where τ_{max} denotes the maximum length in the area of $Q.\Lambda$. After we compute the scores for all such nodes, we select the one with the largest score and include it in R_C.

In each step, the greedy algorithm only checks the nodes connecting to the already explored region R_C. There are at most N_{max} steps and in each step we select the node with the largest score from at most $|V_Q|$ nodes, and thus the complexity is O($N_{max} \log |V_Q|$).

6.2 Top-k LCMSR Query

We extend the LCMSR query to the *top-k length-constrained maximum-sum region* query. Instead of finding the best region defined in LCMSR, the top-k LCMSR query returns the top-k regions such that they have the highest ranking scores and satisfy the given query length constraint within the query region.

Due to the space limitation, we only briefly discuss how to modify APP, TGEN, and Greedy to answer the top-k LCMSR query. In APP, after the candidate tree T_C is returned, we utilize Function findOptTree() to compute the tuple arrays for all nodes in T_C, and we find the best k regions from these arrays. In TGEN, after all edges in $Q.\Lambda$ are processed, we scan the tuple arrays of all nodes to find top-k regions whose scaled weights are the largest. In Greedy, after the first region is detected greedily, we select a node with the largest weight that is not contained in the first region to find the second one. This procedure is repeated until k best regions are found greedily.

7. EXPERIMENTAL STUDY

7.1 Experimental Settings

Algorithms. We study the performance of the following proposed algorithms: the APP algorithm in Section 4, the TGEN algorithm in Section 5, and the greedy method Greedy in Section 6.1.

Data and queries. We use two datasets in our experimental study. The first dataset is a real-life dataset that comprises of the road network of the New York City downloaded from a public website[1], and the geo-textual objects crawled using Google Place API[2]. The road network contains 264,346 nodes and 733,846 arcs. We crawl 0.5 million objects using Google Place API and we map each object to its nearest node on the road network (note that our algorithm can handle objects that lie on an edge a road network). Each crawled object has a name and a type such as "food" and "restaurant," and they are used as the textual description of the objects. The dataset contains 55,230 distinct keywords.

The second dataset is generated from two real-world datasets. We use a larger road network in the area of the northwest part of USA. It contains 1,207,945 nodes and 2,840,208 arcs. We randomly generated a set of objects of the same size of the vertices in the road network following the network distribution. The keywords were collected from Flickr using its public API .We collected more than one million photos with tags taken by 30,664 unique users in the region of USA. Each photo is associated with a set of user-annotated tags. The set of tags of a photo is used as the textual description of an object, and the object is then mapped to its nearest vertex. We remove tags used by only 1 user, which are likely to be noisy, and the dataset has 107,956 keywords.

The locations have a pair of latitude and longitude. In order to compute the Euclidean distance between locations, we convert the data to the UTM (Universal Transverse Mercator coordinate system) format, using World Geodetic System 84 specification.

We generated 5 query sets with different numbers of keywords 1, 2, 3, 4, and 5 for the New York dataset (denoted by NY) and the northwest USA dataset (denoted by USANW), respectively. Note that most of search engine queries with local intent contain only 1-2 keywords [1]; and it is reported in an analysis on a large Map query log [17], nearly all queries contain fewer than 5 words. Each set comprises 50 queries. When generating a query, we first randomly select a query area with the largest $Q.\Lambda$ following the network distribution (100 km^2 for NY and 150 km^2 for USANW). Then, within this area, we randomly select the terms that appear in this area according to their frequency.

All algorithms were implemented in C++ on Windows 7, and run on an Intel(R) Xeon(R) CPU X5650 @2.66GHz with 8GB RAM.

7.2 Experimental Results on NY

7.2.1 *Varying Parameters Used in Algorithms*

APP uses the scaling parameter α to scale node weights into integers, and uses β to do the binary search, and they together decide the approximation ratio of APP. TGEN also uses α to do scaling. Greedy uses μ to balance the importance of node weights and edge lengths when greedily select a node to expand the currently explored region. We study their effect on both runtime and accuracy in the three algorithms. Based on this parameter tuning process, we select the value for each parameter that can achieve good accuracy efficiently. We set the number of query keywords to 3, $Q.\Delta$ to 10 km, and $Q.\Lambda$ to 100 km^2 by default.

[1] http://www.dis.uniroma1.it/~challenge9/download.shtml

[2] https://developers.google.com/places/

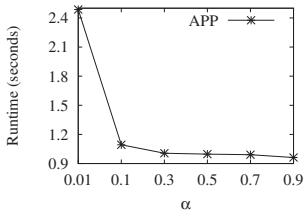


Figure 7: Runtime (NY)

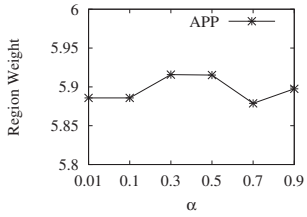


Figure 8: Result Quality (NY)

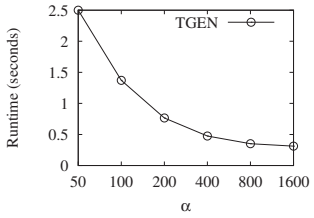


Figure 9: Runtime (NY)

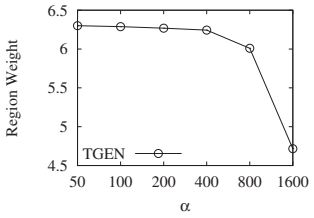


Figure 10: Result Quality (NY)

Varying the scaling parameter α. The scaling parameter α is used in both APP and TGEN. We set β to 0.1 in this set of experiments, and we vary the value of α from 0.01 to 0.9. Figures 7 and 8 show the runtime and accuracy of APP, respectively. It can be observed that the runtime decreases as α becomes larger. This is because that the maximum number of tuples kept on a node is bounded by $T_{max} = N_{max}\lfloor|V_Q|/\alpha\rfloor$. More tuples collide at the same scaled weight as α increases, and thus more enumerations are pruned. The accuracy does not vary much (region weights are all between 5.85 and 5.95). We set α to 0.5 by default for APP in subsequent experiments.

TGEN also utilizes α to do the weight scaling. If we set α to a value smaller than 1, TGEN runs extremely slowly and consumes huge amounts of memory. This is because that the number of tuples on a node can be estimated by $T_{max} = N_{max}\lfloor|V_Q|/\alpha\rfloor$. If there are 10 thousand nodes in $Q.\Lambda$, hundreds of millions of tuples need to be maintained. In contrast, in APP there are at most hundreds of nodes in the candidate tree. In TGEN, we set α to a larger value to decrease the complexity and the space cost. We test the values from 50 to 1600, and Figures 9 and 10 show the results. As expected, as α increases, both the runtime and accuracy decrease. We set α to 400 for TGEN in subsequent experiments by default because using this value, we can achieve both good accuracy and efficiency. We also test α using values 50 and 400 in APP. The average region weight drops to 5.47 and the runtime is 0.786 seconds when α = 50, and the average region weight drops to 5.35 and the runtime is 0.762 seconds when α = 400. We can see that the runtime of APP is not improved much by using larger α, and APP is still slower than TGEN and has worse accuracy.

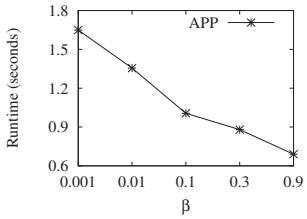


Figure 11: Runtime (NY)

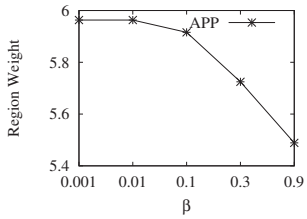


Figure 12: Result Quality(NY)

Varying the binary search parameter β. The value of β affects the range of the binary search and also the accuracy in APP. We vary β from 0.001 to 0.9. Figures 11 and 12 show the runtime and accuracy, respectively. Both the runtime and accuracy drop as β becomes larger. The reason is that, with a larger β, the binary search

can finish earlier because there are more candidate trees. Recall that the approximation ratio of APP is $(1-\alpha)/(5+5\beta)$, and this explains why the accuracy decreases. We notice that APP achieves the same accuracy when setting β to 0.01 and 0.001, which implies that the accuracy cannot be further improved by reducing β. We set β to 0.1 in subsequent experiments for APP.

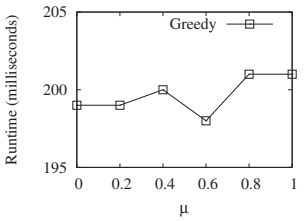


Figure 13: Runtime (NY)

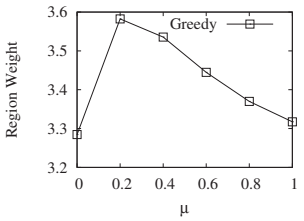
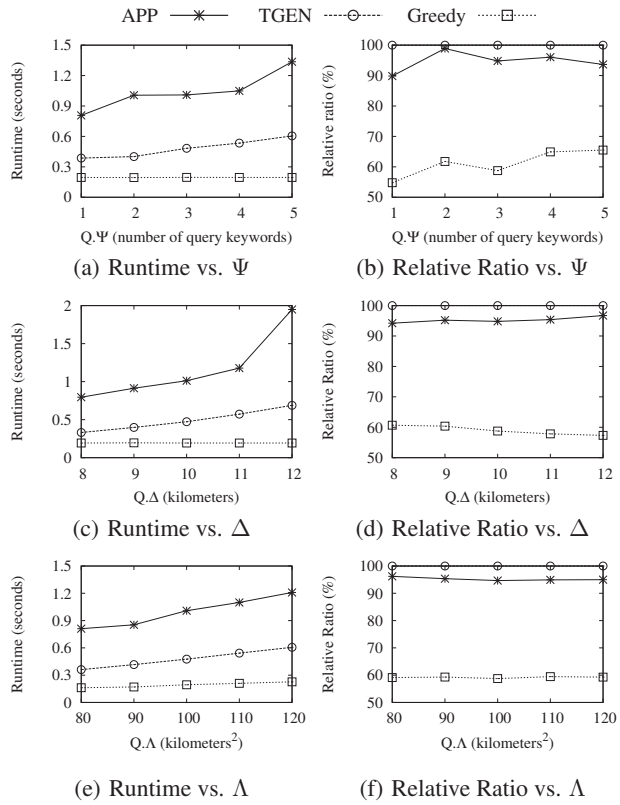


Figure 14: Result Quality(NY)

Varying the parameter μ. The parameter μ balances the importance of node weights and edge lengths when selecting a node to expand the current explored region. It is shown that better accuracy can be achieved by taking into account both node weights and edge lengths than only considering one of them in Greedy. We set μ to 0.2 by default in subsequent experiments.

7.2.2 *Varying Query Arguments*



(a) Runtime vs. Ψ (b) Relative Ratio vs. Ψ

(c) Runtime vs. Δ (d) Relative Ratio vs. Δ

(e) Runtime vs. Λ (f) Relative Ratio vs. Λ

Figure 15: Vary Query Arguments (NY)

An LCMSR query has three arguments, i.e., the query keywords $Q.\psi$, the length constraint $Q.\Delta$, and the region of interest $Q.\Lambda$. We vary the three arguments and study the performance of the three proposed algorithms. We set the number of query keywords to 3, $Q.\Delta$ to 10 km, and $Q.\Lambda$ to 100 km^2 by default.

Due to the absence of efficient exact methods, when evaluating the accuracy we compute the ratio of an algorithm over TGEN (which always has the best accuracy) under the same setting as follows: For each query, we compute the ratio of the weight returned by this algorithm to the weight returned by TGEN, and the average ratio over all queries is finally reported as the measure.

Varying the number of query keywords. Figure 15(a) shows the runtime of the three algorithms when we vary the number of query keywords. It can be observed that Greedy runs very fast due to its simplicity, and its runtime increases slightly as the number of query keywords increases. TGEN runs faster than does APP consistently. All the algorithms run slowly as queries contain more keywords. The reason behind is that, when the number of query keywords increases, more locations become relevant to the query. All these relevant locations have scaled weights, and thus more regions need to be considered during the algorithm execution.

Figure 15(b) shows the accuracy of the algorithms. The ratio of APP is over 90% ratio on all query sets. Greedy has much worse accuracy compared with APP and TGEN, and its accuracy increases a bit as queries contain more keywords. This might be because more nodes used to expand the explored region are relevant to the query.

Varying the query length constraint. Figure 15(c) shows the runtime of the three algorithms when we vary $Q.\Delta$ from 8 km to 12 km. Greedy still runs the fastest. Its runtime increases slightly as $Q.\Delta$ increases, because the region is larger and more nodes need to be checked to expand the explored region. APP runs slower than does TGEN. The runtime of both APP and TGEN increases as $Q.\Delta$ becomes larger. The reason is that under the constraint of a larger $Q.\Delta$, the maximum number of nodes in a region N_{max} becomes larger, and more tuples are stored on each node.

Figure 15(d) shows the accuracy of the three algorithms. APP always has a ratio over 90%. Greedy has the worst ratio, and its accuracy decreases a bit as $Q.\Delta$ becomes larger.

Varying the size of query region. Figure 15(e) shows the runtime of the three algorithms when we vary $Q.\Lambda$ from 50 km^2 to 150 km^2. Since a larger query region contains more nodes and edges in the road network, and all algorithms runs slower as the size of $Q.\Lambda$ increases. The runtime of all algorithms increases almost linearly with the size of $Q.\Lambda$. Greedy runs the fastest, and TGEN is faster than APP over all query sets.

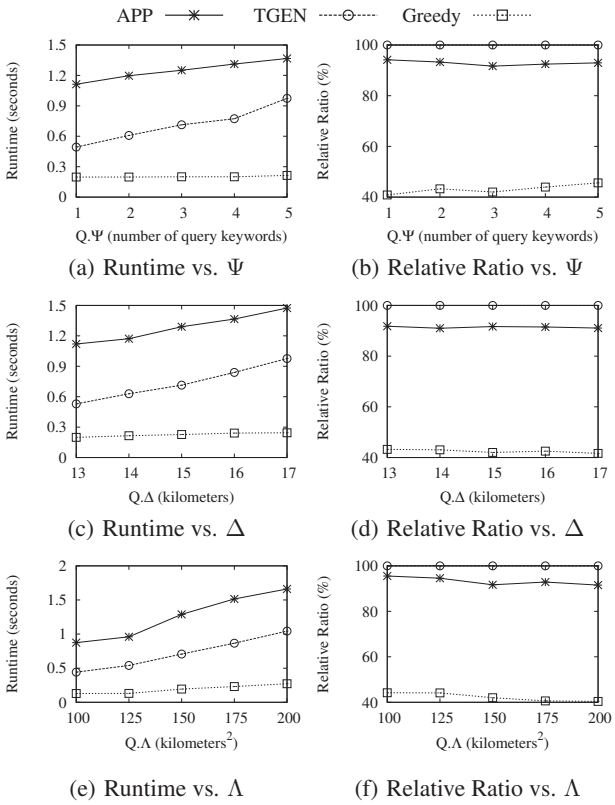
The accuracy of the three algorithms is shown in Figure 15(f). It can be observed that APP still always achieves a ratio of above 90%, and Greedy performs the worst in terms of accuracy.

In conclusion, APP is capable of both excellent accuracy and efficiency. However, TGEN can achieve even better accuracy and efficiency. Greedy performs much worse than other methods in terms of accuracy although it is the fastest.

7.3 Experimental Results on USANW

We also vary α, β, and μ to tune the best parameter values for each algorithm, due to the space limitation the results are not shown, and we finally set α to 0.1 and β to 0.1 for APP, α to 300 for TGEN, and μ to 0.4 for Greedy. We set the number of query keywords to 3, $Q.\Delta$ to 15 km, and $Q.\Lambda$ to 150 km^2 by default.

Figures 16(a) and 16(b) show the runtime and accuracy when we vary the number of query keywords from 1 to 5, Figures 16(c) and 16(d) show the runtime and accuracy when we vary $Q.\Delta$ from 13 km to 17 km, and Figures 16(e) and 16(f) show the runtime and accuracy when we vary $Q.\Lambda$ from 100 km^2 to 200 km^2, respectively. We can observe similar results as we do on NY. The runtime of all algorithms increases as the number of query keywords, $Q.\Delta$, and $Q.\Lambda$ become larger. TGEN always has the best accuracy, APP can always achieve a ratio above 90%, and Greedy only achieves a ratio about 40%. TGEN has the best performance in terms of both accuracy and efficiency.



(a) Runtime vs. Ψ (b) Relative Ratio vs. Ψ

(c) Runtime vs. Δ (d) Relative Ratio vs. Δ

(e) Runtime vs. Λ (f) Relative Ratio vs. Λ

Figure 16: Vary Query Arguments (USANW)

7.4 Real Examples

Figures 17, 18, and 19 show the regions returned by TGEN, APP, and Greedy, when given the same arguments:Bronx, New York, the keywords "cafe" and "restaurant," and a length constraint of 8 km. Greedy returns a region containing only 7 objects with weight 3.6. The region returned by APP contains 11 objects with weight 4.8, while the region returned by TGEN contains 15 objects with weight 5.9. It can also be observed that the objects in the returned regions locate along the roads, and that the returned regions have irregular shapes. This is especially noticeable in Figure 17.

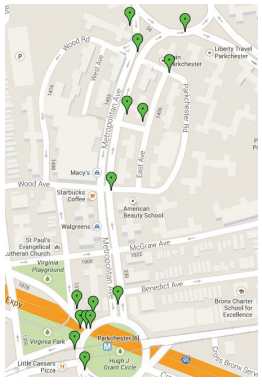


Figure 17: Region (TGEN)

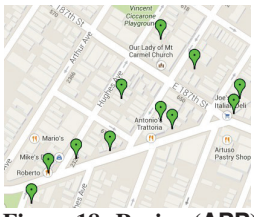


Figure 18: Region (APP)

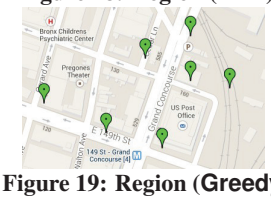
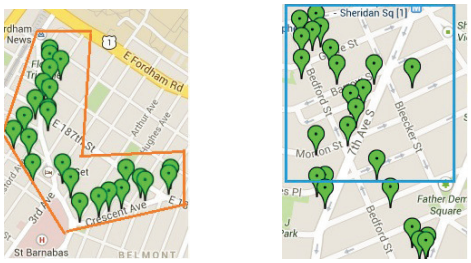


Figure 19: Region (Greedy)

7.5 Comparison with Existing Studies

These experiments compare the quality of regions returned by the LCMSR query and regions returned using width-and height-fixed rectangles by the maximum range sum (MaxRS) query ([4, 11]). Note that the size constraints in LCMSR and MaxRS are different and the two return different types of regions. Thus, it is difficult to set comparable size constraints to compare returned regions. We did a survey with 30 students, and they consistently find that the length constraint of LCMSR is much more user-friendly to specify than the width and height of MaxRS. To compare the two types of queries, we adopt the following procedure: We set both the width and the height of the query rectangle of MaxRS to 500 meters, and we find the rectangular region with the largest total weight for each query. Then we compute the minimum total length of the road segments connecting all relevant objects in this region, and we use this value as the length constraint in the LCMSR query (we use the TGEN algorithm).

For each of 20 queries, we ask 5 annotators to manually compare the quality of two regions returned by LCMSR and MaxRS. Given regions R_1 and R_2 for the same set of query keywords, if no less than 3 users judge that R_1 is better than R_2, we believe that R_1 has better quality. According to the annotation results, the regions returned by LCMSR has better quality on 90% of all testing queries. This is because that real-world regions tend to have arbitrary shapes. Moreover, the objects in the regions returned by MaxRS may not be connected while those returned by LCMSR are always connected by road segments. An example is shown in Figure 20. LCMSR successfully finds an "L"-shaped region, while MaxRS only finds a rectangle covering part of the real region.



(a) LCMSR (b) MaxRS

Figure 20: Example of two types of regions

The regions returned by MaxRS are better for two queries because the real regions happen to be rectangular and TGEN only returns approximate results and thus finds fewer relevant objects.

7.6 Performance of Answering Top-k LCMSR

Figures 21 and 22 show the runtime of the three algorithms for processing the top-k LCMSR query, respectively. On NY, we set the number of query keywords to 3, $Q.\Delta$ to 10 km, and $Q.\Lambda$ to 100 km^2. On USANW, we set the number of query keywords to 3, $Q.\Delta$ to 15 km, and $Q.\Lambda$ to 150 km^2. It can be observed that all algorithms runs a bit slower when increasing the value of k increases. Greedy always runs the fastest, and TGEN consistently runs faster than does APP.

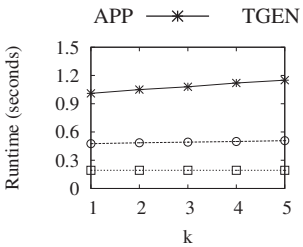


Figure 21: NY

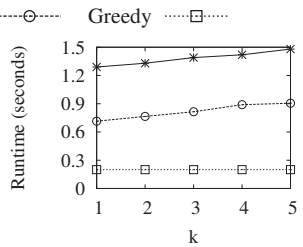


Figure 22: USANW

8. RELATED WORK

Querying Geo-textual Objects. As presented in the introduction, most studies on querying geo-textual objects focus on computing results with a single-object granularity in Euclidean space (e.g., [2, 5, 6]) or a spatial network [15]. They retrieve lists of single objects, each of which is close to the query and relevant to the query keywords. In contrast, the LCMSR query retrieves a region containing a set of objects, targeting users who wish to physically browse, or explore, PoIs.

Several proposals [3, 12, 18, 19] feature a set-of-objects result granularity. In one line of study [18, 19], the query takes a set of keywords and aims to find a set of geo-textual objects such that the union of their text descriptions cover all query keywords and such that the diameter of the objects is minimized. In another line of work [3, 12], the query takes a set of keywords and a query location. The idea is that several objects that satisfy a query together (i.e., cover all the query keywords) may be more convenient for the user than a single, far-away object that satisfies the query by itself. Unlike the LCMSR query, these proposals return a set of objects that together satisfy the query when the objects are taken as one, and they must all be visited. The result cardinality never exceeds the number of query keywords. In contrast, the LCMSR query retrieves many relevant objects enclosed by a region satisfying the size constraint. For example, given query keywords "shoes, jeans," the existing proposals return a set of at most two objects that together cover "shoes" and "jeans," while the LCMSR query returns a compact region with many objects that each is relevant to the query. Next, the existing proposals do not support user browsing behavior because only one group of objects is returned. Third, they do not assume a spatial network and do not take into account co-location, as does our work. Fourth, they use Boolean keyword matching, while we compute the text relevance between the objects and the query.

One proposal [2] accounts for co-location. It delivers single-object granularity results, does not assume a spatial network, and does not support user browsing behavior. In contrast, our proposal exploits the co-location phenomenon. The LCMSR query is often able to find confined regions with a high concentration of PoIs relevant to a user's need, and it accounts for co-location by returning such regions.

Region Search. Liu et al. [11] study the problem of finding subject-oriented top-k hot regions in spatial databases. Choi et al. [4] and Tao et al. [16] propose to solve the maximizing range sum problem in spatial databases. These proposals define regions as width-and height-fixed rectangles or radius-fixed circles, which we find to be less appropriate than connected subgraphs (as exemplified in Figure 3). In practice, regions are often arbitrarily shaped, and using particular size-fixed shapes may only find parts of confined regions with a high concentration of relevant PoIs.

9. CONCLUSION AND FUTURE WORK

We define the length-constrained maximum-sum region query, which takes a set of query topics, a region of interest, and a region size threshold as parameters. Given a set of geo-textual objects mapped to a road network graph, the query retrieves the region, formalized as a connected subgraph with the relevant objects, such that it has the largest weight and is located inside the region of interest and it does not exceed the size threshold. The weight of a result region is defined as the sum of the score, e.g., relevance, of each of its objects w.r.t. the query keywords. The query aims to support users who wish to conveniently explore multiple objects that are relevant to the query, e.g., Italian restaurants.

The problem of answering this query is NP-hard. We devise a node weights scaling technique, and based on this we design an approximation algorithm with performance bound. Extended from this algorithm, we design an algorithm that heuristically finds the region with the largest scaled weight. We also design a greedy approach that takes into account both node weight and edge length when expanding the region. Results of empirical studies show that APP and TGEN are capable of answering queries efficiently and effectively, and TGEN performs better than does APP in terms of both accuracy and efficiency.

In future, we plan to develop some preprocessing techniques to accelerate the proposed algorithms. It is also of interests to consider the number of query keywords covered by a region when retrieving the relevant hot regions.

Acknowledgments X. Cao and G. Cong were supported in part by a grant awarded by a Singapore MOE AcRF Tier 2 Grant (ARC30/12) and a Singapore MOE AcRF Tier 1 Grant. C. S. Jensen was supported by the Geocrowd Initial Training Network, funded by the European Commission as an FP7 Peoples Marie Curie Action under grant agreement number 264994. M. L. Yiu was supported by ICRG grant G-YN38 from the Hong Kong Polytechnic University.

10. REFERENCES

[1] X. Cao, L. Chen, G. Cong, C. S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M. L. Yiu. Spatial keyword querying. In *ER*, pages 16–29, 2012.
[2] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.
[3] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384, 2011.
[4] D.-W. Choi, C.-W. Chung, and Y. Tao. A scalable algorithm for maximizing range sum in spatial databases. *PVLDB*, 5(11):1088–1099, 2012.
[5] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.
[6] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.
[7] S. N. Durlauf and L. E. Blume. *The New Palgrave Dictionary of Economics*. Palgrave Macmillan, 2nd edition, 2008.
[8] N. Garg. A 3-approximation for the minimum tree spanning k vertices. In *FOCS*, pages 302–309, 1996.
[9] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, 1995.
[10] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, page 16, 2007.
[11] J. Liu, G. Yu, and H. Sun. Subject-oriented top-k hot region queries in spatial dataset. In *CIKM*, pages 2409–2412, 2011.
[12] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu. Collective spatial keyword queries: a distance owner-driven approach. In *SIGMOD Conference*, pages 689–700, 2013.
[13] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *SIGIR*, pages 275–281, 1998.
[14] R. Ravi, R. Sundaram, M. V. Marathe, D. J. Rosenkrantz, and S. S. Ravi. Spanning trees short or small. In *SODA*, pages 546–555, 1994.
[15] J. B. Rocha-Junior and K. Nørvåg. Top-k spatial keyword queries on road networks. In *EDBT*, pages 168–179, 2012.
[16] Y. Tao, X. Hu, D. Choi, and C. Chung. Approximate MaxRS in spatial databases. *PVLDB*, 6(13):1546–1557, 2013.
[17] X. Xiao, Q. Luo, Z. Li, X. Xie, and W.-Y. Ma. A large-scale study on map search logs. *TWEB*, 4(3):1–33, 2010.
[18] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.
[19] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, pages 521–532, 2010.
[20] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, pages 155–162, 2005.
[21] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006.