Fast Query Decomposition for Batch Shortest Path Processing in Road Networks

Lei Li, Mengxuan Zhang, Wen Hua, Xiaofang Zhou
School of Information Technology and Electrical Engineering, The University of Queensland, Australia
{l.li3, mengxuan.zhang, w.hua}@uq.edu.au, zxf@itee.uq.edu.au

***Abstract*—Shortest path query is a fundamental operation in various location-based services (LBS) and most of them process queries on the server-side. As the business expands, scalability becomes a severe issue. Instead of simply deploying more servers to cope with the quickly increasing query number, batch shortest path algorithms have been proposed recently to answer a set of queries together using shareable computation. Besides, they can also work in a highly dynamic environment as no index is needed. However, the existing batch algorithms either assume the batch queries are finely decomposed or just process them without differentiation, resulting in poor query efficiency. In this paper, we aim to improve the performance of batch shortest path algorithms by revisiting the problem of query clustering. Specifically, we first propose three query decomposition methods to cluster queries: *Zigzag* that considers the *1-N* shared computation; *Search-Space Estimation* that further incorporates search space estimation; and *Co-Clustering* that considers the source and target's spatial locality. After that, we propose two batch algorithms that take advantage of the previously decomposed query sets for efficient query answering: *Local Cache* that improves the existing *Global Cache* with higher cache hit ratio, and *R2R* that finds a set of approximate shortest paths from one region to another with bounded error. Experiments on a large real-world query sets verify the effectiveness and efficiency of our decomposition methods compared with the state-of-the-art batch algorithms.**

I. Introduction

Computing the shortest path between two vertices is a fundamental operation in various applications such as social network analysis, knowledge graph question answering, road network routing and navigation, to name a few. In this work, we focus on shortest path queries in a dynamic road network denoted as $G(V, E)$, where vertex set V is a set of road intersections and edge set $E \subseteq V \times V$ is a set of road segments. Each vertex v has a spatial coordinate (e.g., longitude and latitude) and each edge (u, v) is associated with a non-negative numerical weight $w(u, v)$ that represents the cost (e.g., distance/travel time) to travel from u to v. Given two locations s and t in G, the shortest path query searches for a route from s to t with the minimum total cost.

Depending on if the edge weight stays the same or not, the graph can be classified into *static graph* and *non-static graph*. The static graph finds the path based on the distance, while the non-static graph uses the changing travel time. Furthermore, depending on whether the edge change is predictable or not, the non-static graph can be further classified into *time-dependent graph*, which uses a function to tell the travel time at different time points, and *dynamic graph* whose edge weight changes in ad-hoc. Although most of the time-related path-finding algorithms use the time-dependent graph to model the road network, it is unrealistic in real life because the traffic condition is unstable and the accident happens randomly. Therefore, we use the dynamic graph in this work by viewing it as a series of static snapshots and using the latest one to describe the current traffic condition.

Various approaches have been proposed to efficiently process shortest path queries, and most of them are index-based [4], [20], [10], [1], [24], [26], [27], [28], [15], [19], [17], [35], which pre-compute and store various auxiliary information in the index structures. Although these algorithms are quite efficient in query answering as the shortest path can be retrieved quickly from the index, they still have an obvious limitation: the index maintenance cost (i.e., construction and update) is very high when the network evolves. The traffic condition may have changed already (or switched to the next snapshot) before the index construction finishes. Such drawback becomes much worse on the time-dependent environment [18]. Therefore, the index-free algorithms (*Dijkstra*'s [7] and A^* [14]), which does not need any index construction or maintenance, are more appealing to the industry. Besides, they support the dynamic environment naturally because they can simply compute with new edge weights.

It is worth noting that all the aforementioned algorithms mainly focus on improving the efficiency of answering each shortest path query, while the scalability issue is largely ignored. Shortest path requests keep coming at a high speed in real applications. As an example, one of our collaborators which is a Uber-like company needs to handle over 100,000 shortest path queries per minute, or even over 1 million such queries in a peak hour. A direct solution is to deploy more servers to cope with a large number of concurrent queries which, however, is obviously resource-inefficient. Instead, we attempt to address the scalability issue algorithmically in this work by reusing shareable computation among queries.

Existing solutions in this line [29], [30] try to introduce caching technique to shortest path query processing. In particular, they store the most beneficial paths in a cache such that all the other queries whose origin and destination lie on the cached paths can be answered directly. However, the cache suffers from low hit ratio sometimes, and it is hard to determine the most beneficial way for cache refreshing in a dynamic environment. Obviously, if the concurrent queries can be rescheduled with similar queries issued together, we can

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00107

improve cache hit ratio and reduce cache refresh accordingly. Another approach, namely batch query processing [21], [25], adopts the *path coherence* phenomenon in road networks to utilize shared computation. Intuitively, when path queries are localized (i.e., origins and destinations are centralized in a small region such as airports, train stations, shopping malls, etc.), there could be a large proportion of common computations (or sub-paths) among these queries. Therefore, they attempt to group coherent queries together into several batches and answer each batch within a single run, so as to minimize total query cost. The idea is quite promising. Nevertheless, without a strict theorem, the existing batch algorithms can only determine query schedule indirectly by grouping queries based on some heuristics such as spatial closeness of origins and destinations, leading to poor query accuracy.

As we can see from the above analysis, a good query scheduling is indispensable to both cache-based and batch algorithms. Therefore, in this paper, we aim to improve the performance of batch shortest path query processing by revisiting the problem of query clustering[1]. In particular, we attempt to address the following challenges in this work: 1) how to efficiently determine the best partition of queries given a tremendous number of possible query clustering strategies? 2) how to answer queries in a batch within a single run and achieve accurate or error-bounded shortest paths? Our major contributions can be summarized as follows:

- We formally study the problem of batch shortest path query processing which addresses the scalability issue algorithmically by reusing shareable computation among concurrent queries.
- We propose three query set decomposition methods to efficiently cluster queries: *Zigzag* that considers the *1-N* shared computation; *Search-Space Estimation* that incorporates query and road directions; and *Co-Clustering* that considers the source and target's spatial locality.
- We propose two batch algorithms that answer the previously decomposed query sets: *Local Cache* that obtains accurate shortest paths and improves the existing *Global Cache* with higher cache hit ratio; and *R2R* that finds a set of approximate shortest paths from one region to another with bounded error.
- We conduct extensive evaluation on a large real-world query dataset, and the experimental results verify the superiority of our approaches compared with current state-of-the-art cache-based and batch shortest path algorithms.

The remaining of this paper is organized as follows: We summarize the current literature of shortest patch processing in Section II. Our problem is formally defined in Section III, and our solutions including query clustering and batch algorithms are described in detail in Section IV and Section V, respectively. Section VI reports the experimental results, followed by a brief conclusion in Section VII.

[1]In this paper, we use "query clustering", "query scheduling", and "query decomposition" interchangeably whenever it is clear.

II. Related Work

We divide the literature review into two parts: single shortest path algorithms which focus on improving the efficiency of each query; and batch shortest path algorithms that attempt to reduce total cost of answering a group of queries.

A. Single shortest path algorithms

In the past decades, various techniques have been proposed for the shortest path calculation in road networks. The fundamental algorithms are *Dijkstra*'s [7] and A^* [14] algorithms. The *Dijkstra*'s algorithm searches for the shortest path through an entire network traversal, which is inefficient due to some unnecessary node visits. A^* algorithm directs the graph traversal towards the destination by introducing the heuristic distance (e.g., Euclidean distance) so as to improve query efficiency. In addition, the bidirectional search technique [23] can further decrease the search space by traversing the graph from origin and destination simultaneously. These algorithms are index-free and can be easily applied dynamic environment.

Another line of research focuses on index-based shortest path algorithms which further reduce query cost by pre-calculating and maintaining an indexing structure. Particularly, algorithms such as *ALT* [13], *Geometric container* [31], *Arc-flag* [22], *Contraction hierarchy* [12], and *Arterial hierarchy* [36] prune the search space by referring to information stored in the index (e.g., landmarks, shortcuts, etc.). Other algorithms, such as *2-Hop labeling* [4], [20], [18], *IS-Labelling* [10], *Pruned Landmark Labelling* [1], *H2H-index* [24], *SILC* framework [26], *Distance Oracle* [27] and *Path Oracle* [28], etc., attempt to materialize all the pairwise shortest path results in a compressed manner such that a given shortest path query can be answered directly via a simple table-lookup or join. The index-based algorithms are usually more efficient than the non-index counterparts. However, the index structure is quite space-consuming and requires a long time to construct. Furthermore, road networks can change over time. Although some index maintenance methods (e.g., [8], [6], [2], [32]) have been proposed, they only deal with some special cases of network updates (e.g., edge weight decrease, vertex deletion or edge deletion) and cannot work in the dynamic environment.

B. Batch shortest path algorithms

All the aforementioned algorithms only target at improving the efficiency of answering a single shortest path query. They do not consider how to share computation among multiple queries. Query scalability becomes an issue when a large number of shortest path queries arrive simultaneously. Several techniques have been introduced to address this issue.

It is easy to prove that the sub-path of a shortest path is still a shortest path. Based on this theory, the *Global Cache* algorithm [29] uses a cache to store the most beneficial paths such that all the sub-path queries of the cached paths can be answered directly. [30] further introduces *cache refreshing* and *concise caching* to adapt to the dynamic road networks and increase cache hit ratio. Nevertheless, it can be expected that the cache has to be updated frequently due to the low hit ratio when queries come randomly. Therefore, a good query rescheduling, which is the focus of this work, is very important to the cache-based method as it can improve cache hit ratio and reduce cache refresh. Besides, there is another major limitation of the cache-based algorithm: it requires the endpoints of the shortest path query to exactly "hit" the cached paths in order to reuse previous calculation. However, it is obvious that queries with close origins or destinations can also share computations with each other.

Inspired by the *path coherence* property of road networks, several batch shortest path algorithms are proposed to process a group of queries together in a single run by sharing the intermediate results. For a group of queries with nearby sources and nearby destinations, [21] first calculates the shortest path between the exit point p_s of the source region and entry point p_t of the destination region. Then, the shortest path between s and t is retrieved by concatenating the shortest path from s to p_s, from p_s to p_t and from p_t to t. In [25], an A^*-like algorithm is proposed for batch shortest path processing. It keeps track of all the queries waiting to branch a certain vertex in a queue, and the average distance is recorded to generate approximate solutions to these queries. [33] introduces a generalized A^* algorithm to deal with a batch of queries with the same origin or destination. It uses a singe "representative node" to conduct A^*-search until all the target nodes are visited. To further improve query efficiency, [33] heuristically divides the widespread target set into several clusters and runs the generalized A^* algorithm for each query cluster.

There are still some major limitations of the existing batch shortest path algorithms. Firstly, query scheduling is determined based on simple heuristics such as the spatial closeness of query origins and destinations, which cannot generate a beneficial clustering result sometimes. Secondly, query clustering is regarded as a preprocessing step of query answering, whose time cost should also be carefully considered. It is however ignored in the existing work. Finally, some algorithms such as [21] only produce approximate solutions to shortest path queries without a bounded error.

III. Preliminary

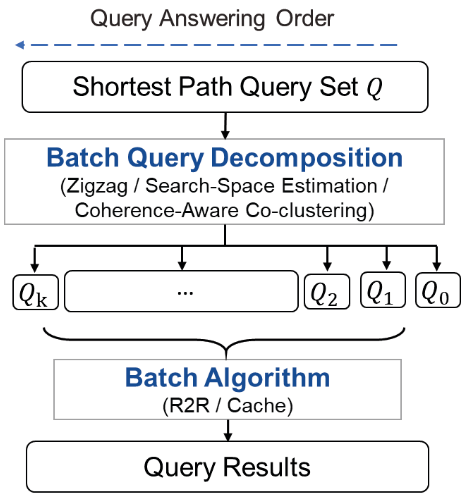
In this section, we will introduce and analyze our problem formally. In particular, Section III-A describes the problem input, namely the shortest path query set, and Section III-B defines and analyzes the query decomposition problem, and describes the overall procedure.

A. Shortest Path Query Set

Road Network is denoted as a directed graph $G(V, E)$, where V is a set of vertices representing the road intersections and $E = \{(u, v)\} \subseteq V \times V$ is a set of edges representing the road segments. Each vertex v has a spatial coordinate $(v.x, v.y)$ reflecting its longitude and latitude. Each edge (u, v) is associated with a non-negative numerical weight $w(u, v)$ that represents the cost to travel from u to v. The cost can be the length of the road segment, the time required to pass the road segment, or other costs such as fuel consumption and toll charge in different applications. A path from s to t is a sequence of vertices $p =< v_0, v_1, ..., v_k >$ with $v_0 = s$ and $v_k = t$. The length of a path $d(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$, and the shortest path is the one with the minimum $d(p)$. A batch shortest path query set is made up of a collection of shortest path queries issued within a short time period (e.g., 1 second), which is defined as below:

Definition 1: **(Shortest Path Query Set)**. Given a starting vertex set S and a target vertex set T, a shortest path query set is denoted as $Q = \{q_i\} \subseteq S \times T$, where $q_i = (s_i, t_i)$ is a shortest path query with $s_i \in S$ and $t_i \in T$. The size of Q satisfies: $max(|S|, |T|) \leq |Q| \leq |S| \times |T|$.

Normally, the queries in Q are answered one by one with a 1-1 shortest path algorithm. Suppose we have a function $C(q)$ to indicate how much it costs to answer query q (for example, *visited node number (VNN)*), then the total cost to answer Q is $C(Q) = \sum C(q_i)$. However, this approach ignores an important coherence phenomenon of the shortest paths: some of the paths have similar sub-paths that could be shared during computation. Therefore, $\sum C(q_i)$ is actually the highest cost we could get for $C(Q)$. If we can identify these similar queries beforehand and answer them in a batch, it is possible to reduce $C(Q)$ and achieve faster query answering time, as discussed in Section II-B.



Query Answering Order
Shortest Path Query Set Q
Batch Query Decomposition
(Zigzag / Search-Space Estimation / Coherence-Aware Co-clustering)
Q_k
...
Q_2
Q_1
Q_0
Batch Algorithm
(R2R / Cache)
Query Results

Fig. 1. Framework of Batch Query Answering

B. Shortest Path Query Decomposition Problem

Some batch algorithm such as *Global Cache* [29] takes the entire Q as a batch, but it suffers from the low hit ratio since all it can rely on are the historical query log and the network structure. If we could re-order the queries and answer the similar ones together, higher hit ratio could be achieved. If fact, all the other batch algorithms [21], [25], [33] decompose Q into subsets and process them separately. However, all of them are simple heuristic-based and little research has been done to improve the decomposition quality. This problem can be formulated as below:

Definition 2: (**Shortest Path Query Decomposition Problem**). Given a query set Q and a cost function C, decompose Q into several query subsets $\hat{Q} = \{Q_i\}$ such that $Q_i \subseteq Q, \cup Q_i = Q, Q_i \cap Q_j = \phi$, and $\sum_{Q_i \in \hat{Q}} C(Q_i)$ is minimum.

Figure 1 shows an example: Q is decomposed into k subsets. All the previous batch algorithms answer Q directly so they can hardly improve the efficiency, while we propose to add a decomposition layer to reduce the total query cost. However, it is usually unrealistic to achieve the optimal result because of the following reasons. First of all, the actual cost cannot be obtained until the shortest path search finishes. Without an accurate query cost estimation, the decomposition can only base on the approximate cost, which could not guarantee the result is minimum. Moreover, the batch cost is not necessarily smaller than answering queries individually. For example, it normally takes a larger cost to answer two queries with targets on the opposite directions of the source using generalized *1-N* A^* algorithm than answering them separately. In fact, [33] observes that 30° is already large enough to deteriorate batch performance. Furthermore, the query order inside each subset can also affect the query performance. In other words, $C(Q_i)$ is not a static value and there exists a permutation of queries in Q_i that has the minimum $C(Q_i)$. For example, in the cache-based method, different query orders could result in different cache content. If the more important query/path appears latter in the cache, the hit ratio would become lower. Therefore, the number of the possible decomposition results is

$$\sum_{k \in [1,|Q|]} \frac{|Q|! \times \binom{|Q|-1}{k-1}}{k!} \tag{1}$$

where $|Q|!$ is the total permutation of the queries, $\binom{|Q|-1}{k-1}$ is the number of possible k subsets (i.e. combination of $k-1$ possible cutting positions among the $|Q|$ queries), and $k!$ is the total permutation of the k subsets. Obviously, it is impossible to enumerate all the possible subsets, let alone the costs of them are not accurate. Last but not least, the decomposition phase is just the query preprocessing and the time it costs is also part of the query answering time. Although an optimal decomposition result could reduce the batch path algorithm running time, the decomposition itself could be time consuming and prolong the total query answering time. In fact, it is not a trade-off between the decomposition time and the decomposition quality, because the overall goal is reducing the total query answering time.

In summary, we aim to decompose Q as fast as possible while achieving a more beneficial decomposition result than the existing heuristic approaches. A framework of the total query processing procedure is illustrated in Figure 1. The query set is decomposed into subsets with three different methods, which will be introduced in Section IV. After that, these subsets will be answered separately using our revised batch algorithms, which will be described in Section V.

IV. Query Decomposition

In this section, we propose three query decomposition methods: *1-N Zigzag Decomposition*, *Search Space Estimation-based Decomposition*, and *Coherence-Aware Co-Clustering*. The first two methods produce results that are suitable for cache-based batch algorithms and create query set with a shape of cloud, while the third one helps the region-to-region batch algorithm to identify regions quickly and create query set in a shape of dumbbell ([25] demonstrates that the approximate shortest paths are more accurate when the starting region and target region are far away from each other). Specifically, Section IV-A presents the *1-N Zigzag* method that first decomposes the queries into several *1-N* subsets based on angle/distance thresholds and then merges the similar subsets into larger sets. In other words, it is a method based on one-way clustering. Section IV-B introduces a *Search Space Estimation* method which considers the properties of the underlying road network and puts the queries with similar search space into one query subset. Section IV-C describes the *Co-Clustering* method that clusters the queries directly using both of the source's and target's spatial coordinates as the feature, so it is based on two-way clustering. Besides, an approximate coherence is applied to guarantee the results are not far from the error bound of the region-to-region algorithm.

A. 1-N Zigzag Decomposition

Given a query set Q, the *1-N zigzag* approach first organizes the queries into a source query set $Q_s = \{Q_{s_i}\}$ and a target query set $Q_t = \{Q_{t_j}\}$, where Q_{s_i} is the set of queries that have the same starting vertex v_i and Q_{t_j} is the set of queries that have the same target vertex v_j. Obviously, a query $q_k = (s_i, t_j)$ appears in both of Q_{s_i} and Q_{t_j}, and $|Q_s| = |Q_t| = |Q|$. After that, each Q_{s_i} is further decomposed into several subsets $\{Q^k_{s_i}\}$, with targets of the same subset near to each other. Similar process is also applied on each Q_{t_j} to generate $\{Q^k_{t_j}\}$. Each $Q^k_{s_i}$ or $Q^k_{t_j}$ can be viewed as a *1-N* batch query set. Finally, the subsets are merged together into larger *M-N* query subset for the batch query algorithms. In the following sections, we will explain the *target set decomposition phase* and *zigzag merge phase*, respectively.

1) Target Set Decomposition Phase: Given a *1-N* query set Q_{s_i}, its targets form a target set T. The target set decomposition puts the targets into several smaller disjoint clusters $\{T^k_i\}$, where the targets in each T_i are close to each other. Because decomposing the sources of each Q_{t_j} is the same as decomposing the targets of each Q_{s_i}, we just explain target decomposition here.

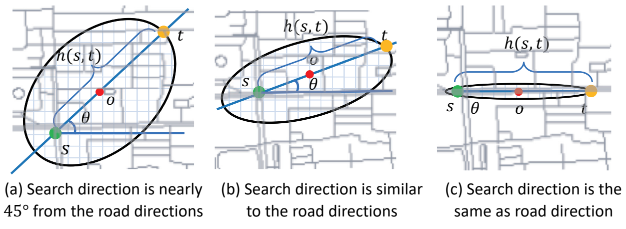
The most straightforward method is using *DBSCAN* [9], [11] to cluster the targets. However, the cluster it generates could be in any shape, which may not be suitable for the batch query processing. For example, a 180° arc-shape cluster is not suitable because little computation can be shared among these queries even if they share the same source. Therefore, more insights into the path coherence have to be taken into consideration. The first one is the direction from the source to the target. Intuitively, the targets in the similar direction could have higher possibility to share the same sub-paths. Thus, we set an angle threshold δ to put all the targets within δ into one cluster. The second insight is the distance to the targets. Because the farther targets usually have larger search space, they have a higher possibility to cover the unanswered shorter queries. Since the exact distance cannot be obtained before the query is answered, we use the Euclidean distance as the estimated distance. Therefore, we propose an *Angle/Distance (AD)*-based decomposition method that considers both of these two insights. Firstly, we select the farthest target as a cluster center and use its direction as the axis. After that, all the targets falling within $\pm\frac{\delta}{2}$ around the cluster axis are included in this cluster. This procedure is repeated on the remaining targets until every target has a cluster. As the sorting on the distance is inevitable, the time complexity of the *AD* decomposition is $O(n \log n)$ where n is the total number of queries.

2) Zigzag Merge Phase: After generating $\{Q^k_{s_i}\}$ and $\{Q^k_{t_j}\}$, we need to merge these *1-N* query sets into larger *M-N* sets. The *zigzag merge* approach got its name because we visit $\{Q^k_{s_i}\}$ and $\{Q^k_{t_j}\}$ alternatively to expand the query set. To begin with, all the $1-N$ clusters are sorted by their sizes, and we visit them in the descendant order. Suppose the first cluster is $Q^k_{s_i} = (s_i, T^k_i)$ and we are creating Q_0. Then for each $t_j \in T^k_i$, we retrieve its corresponding $Q^k_{t_j}$ that contains s_i and add its queries into Q_0. As a result, $Q_1 = \bigcup Q^k_{t_j} \cup Q^k_{s_i}$. After that, we remove all the queries in Q_0 from the remaining clusters and update their sizes. This process continues until all the clusters are processed. An inverted index from query to cluster is created for faster cluster finding, and cluster visiting order is maintained by a max-heap.

In addition, the above process can only combine the clusters with multi-sources or multi-targets but cannot handle the smaller 1-1 clusters. One straightforward approach is to simply treat them as normal query subsets and feed them to the batch algorithm. However, many 1-1 clusters might have very high path coherence with the existing query subsets, so ignoring them would reduce the batch processing benefit. Therefore, we also have to consider 1-1 clusters during the merge phase. Suppose we have obtained a query subset Q_i, which has a source set S_i and a target set T_i. We first find the convex hull H^s_i of S_i and H^t_i of T_i. Then we add the 1-1 clusters whose source and target falling within H^t_i and T_i into Q_i. To speed up the processing, we use a grid-based quad-tree to organize the remaining sources and targets approximately. The 1-1 clusters with source falling in the grids covered by H^s_i is selected as a candidate set, and the final result is validated by checking if their targets falling in the grids covered by H^t_i. After all the subsets are generated, the 1-1 clusters covered by them are also absorbed.

B. Search Space Estimation Decomposition

The essential idea of answering a set of queries in a batch is that they could have higher possibility of shared computation, or in other words, higher path coherence. Although the source/target location used in the previous method is an approximation of coherence, it does not take the actual search space into consideration. In fact, it is the search space that determines how much coherence the queries have. If one query's search space is covered by the others, or their search spaces have higher similarity, they have larger chance to share computation. Therefore, we dig deeper into the search space analysis to generate more coherent query subsets.



$h(s,t)$
t
o
θ
s
(a) Search direction is nearly 45° from the road directions
$h(s,t)$
t
o
θ
s
(b) Search direction is similar to the road directions
$h(s,t)$
s
θ
o
t
(c) Search direction is the same as road direction

Fig. 2. Generalized A^* Search Space Example

As the proposed batch query answering methods in this work use the *generalized* A^* [33] as the base path finding algorithm, our coherence estimation is based on A^* search space estimation. Basically, the *generalized* A^* can find the shortest paths from one starting vertex s to a set of targets T in one single run. Its query cost is determined by the searching space from the starting vertex s towards the *representative vertex* t, which is used to compute the heuristic distance $h(u,t)$ (using *Euclidean* distance or *Landmark* estimation), until all the targets are visited. Because the farthest target in T has the largest heuristic distance, it is selected as t in [33]. As illustrated in Figure 2, the estimated search space of a normal road network is an ellipse with s as one focus and t at the far end of the semi-major axis. Apart from the heuristic distance $h(s,t)$, the ellipse size is also determined by the angle θ between the search direction from s to t and the underlining roads directions. The maximum value of θ is 45° (When $\theta > 45°$, we can use $90° - \theta$ instead). The distance from s to the other focus is $\frac{2h(s,t)\cos\theta}{1+\cos\theta}$, and the constant distance is $\frac{2h(s,t)}{1+\cos\theta}$. Some examples of the search space with different θ are shown in Figure 2.

To estimate the search space of a query in near-constant time, we need a light-weight search space oracle to capture the road network's vertex number distribution and edge direction information. An outline of the estimation step for a given query is illustrated in Figure 3. Given a query from s to t, we first compare its direction with the underlying road directions to obtain a search difference angle θ. After that, an ellipse search space is determined and we need to return the queries it covers. To speed up the query finding, we partition road network into grids and only visit the approximately searched grids. Finally, the similar query sets are merged into larger sets.

In the following, we first explain the data structure of our search space oracle, and how to use it to summarize the edge directions. Then we describe the query set generation, which involves angle determination, search space determination, and query testing. Finally, the coverage estimation is provided to merge the similar query sets quickly.

1) Road Network Partition: Graph partitioning methods split the original graph into subgraphs with small number of edge cuts, and the *Natural Cut* [5] is the state-of-the-art partitioning method for road network. However, the subgraphs generated by these methods can have arbitrary sizes and shapes, which makes it hard to estimate the search space. Moreover, their optimization goal, the minimum edge cut number, has nothing to do with our estimation task. Therefore, it is not applicable to utilize the cut-minimizing partition methods here. In fact, a simple uniform strategy like the grid-based partition would be more suitable for our tasks.

We first describe our adaptive multi-level grid-based partition structure. Given a road network G, we split it into $2^n \times 2^n$ grids according to their latitudes and longitudes. These grids are indexed with a quad-tree-like hierarchical structure for faster regional summarization. To be specific, with the root being level 0, level i has 4^i grids. Level $i+1$ can be deemed as dividing each grid of level i into four equal-size quadrants. Each level's grids can be located by its boundary coordinates quickly, and the locality information is also captured by the neighboring coordinates. Each grid stores the number of vertices $g.n$ in it and the summarization of its road directions $g.\theta$.

Next, we explain how to summarize the directions of the roads in a grid. First of all, we need a reference line to determine the angle of the edges. As most of the roads in a small portion of the road network are either parallel or perpendicular to each other, using only one reference line will mix them up. In fact, the grid direction should reflect the offset degree of the road directions from the reference system. Therefore, we use both of the latitude line *lat* and the longitude line *lon* as the reference lines. For an edge e, its angles to the two reference lines are denoted as $\measuredangle(e, lat)$ and $\measuredangle(e, lon)$, whose values are between $0°$ and $90°$. The edge direction $e.\theta$ is the minimum of them: $e.\theta = min(\measuredangle(e, lat), \measuredangle(e, lon))$. The direction of a grid g is the weighted average directions of all the edges within it:

$$g.\theta = \frac{\sum_{e \in g} w(e) \times e.\theta}{\sum_{e \in g} w(e)} \tag{2}$$

The direction summarization of a set of grids $\hat{g} = \{g\}$ is the weighted average of each grid's direction:

$$\hat{g}.\theta = \frac{\sum_{g \in \hat{g}} w(g) \times g.\theta}{\sum_{g \in \hat{g}} w(g)} \tag{3}$$

where $w(g)$ is the weight sum of all the edges in grid g.

2) Search Space Estimation: As mentioned previously, we use ellipse to simulate the search space. The shape of an ellipse is determined by two foci and a distance sum. Given a query $Q(s,t)$, we only know s is one foci, Euclidean distance $h(s,t)$ from s to t, and an angle $q.\theta = min(\measuredangle(\vec{st}, lat), \measuredangle(\vec{st}, lon))$ from the reference line. The query's angle θ from the road network, which can determine the other foci f's position and

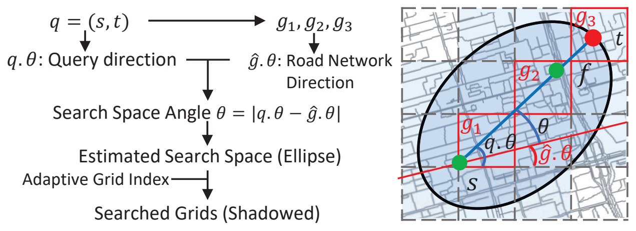

$q = (s,t)$
g_1, g_2, g_3
$q.\theta$: Query direction
$\hat{g}.\theta$: Road Network Direction
Search Space Angle $\theta = |q.\theta - \hat{g}.\theta|$
Estimated Search Space (Ellipse)
Adaptive Grid Index
Searched Grids (Shadowed)
g_3
t
g_2
f
g_1
θ
$q.\theta$
$\hat{g}.\theta$
s

Fig. 3. Search Space Estimation

the constant distance d_q, can be obtained by $|(q.\theta - \hat{g}.\theta)|$. $\hat{g}$ is the set of grids that $\vec{s,t}$ traverses through. For example, $\hat{g} = \{g_1, g_2, g_3\}$ in Figure 3. The distance from s to f can be computed by:

$$d(f,s) = \frac{2h(s,t)\cos\theta}{1+\cos\theta} \tag{4}$$

And the coordinates of f are:

$$\begin{cases} f.x = s.x \pm d(f,s)\cos q.\theta \\ f.y = s.y \pm d(f,s)\sin q.\theta \end{cases} \tag{5}$$

The $\pm$ depends on the relative position of s and t.

Although the estimated search space is conceptually an ellipse, we do not draw an actual ellipse to determine if one vertex or one grid is enclosed by it or not. In fact, we only need to check if the grid's endpoint is covered by the ellipse. To test if one endpoint p is in the ellipse, we compare its distance to the two foci d_p and the ellipse's constant distance $d_q = \frac{2h(s,t)}{1+\cos\theta}$. If more than two endpoints are inside the ellipse, we label this grid as a *covered grid*. Besides, the grids $\hat{g}$ that are used to determine the angle difference are naturally *covered grids*.

3) Query Decomposition: Similar to the *zigzag decomposition*, the search-space estimation decomposition algorithm also has two phases: the *generation phase* that creates a set of query clusters, and the *merge phase* that combines the clusters into larger query subsets.

Firstly, all the query sources and targets are indexed by the grids and the queries are sorted by their Euclidean distances. Then we process the queries in the distance decreasing order, because the longer ones tend to cover larger area. When we process a query q to generate a cluster Q_i, we can get its corresponding covered grids and store them with Q_i as $Q_i.g$. For all the queries whose source and target are all within $Q_i.g$, we further check their query direction. If the direction is smaller than $\frac{\delta}{2}$, we put them into Q_i and remove them from the grid. The cluster's direction is the same as q's direction. When all the queries are processed, we get a set of clusters.

During the merge phase, the generated clusters are sorted based on the directions, and we visit them in a directional sliding window. The window threshold is set to a small value like $\frac{\delta}{8}$ because larger direction difference would reduce the coherence and result in higher computation cost. For the clusters within the window, we use the *overlap coefficient* to represent the cluster similarity, which is computed as below:

$$overlap(Q_i, Q_j) = \frac{Q_i.g \bigcap Q_j.g}{min(|Q_i.g|, |Q_j.g|)} \tag{6}$$

If the overlap coefficient exceeds a pre-defined threshold, we combine Q_i and Q_j into a new query subset, and its direction is set to the weighted average $\frac{|Q_i| \times Q_i.\theta + |Q_j| \times Q_j.\theta}{|Q_i| + |Q_j|}$.

C. Coherence-Aware Co-Clustering Decomposition

1) Query Similarity: Unlike the previous methods, in this section, we consider two-way clustering strategy which groups shortest path queries directly by calculating query similarity based on both origins and destinations. Ideally, queries which have a high possibility of shared computation should be grouped together and answered in a batch. To support fast query clustering, we ignore the more complicated coherence estimation and introduce an alternative heuristics-based approach in this part. Our intuition is that queries with spatially close origins and destinations have a large chance of sharing their sub-paths, and hence should be grouped into one cluster.

Algorithm 1 shows the workflow of the co-clustering decomposition approach. For each query cluster Q_i, we regard the first query added into Q_i as its center (also called representative) denoted as C_i. Given a new query q, we determine to add q into Q_i if both the origins and destinations of q and C_i are spatially close enough, specifically, $d_{euc}(q.s, C_i.s) \leq r_i^*$ and $d_{euc}(q.t, C_i.t) \leq r_i^*$, where d_{euc} represents the Euclidean distance between two vertices and r_i^* is the radius of Q_i.

Algorithm 1: Co-Clustering Decomposition

```
   Input: $Q$: shortest path query set
   Output: $\{Q_i\}$: query clusters
1  $index = 0$;
2  foreach $q \in Q$ do
3      if $index == 0$ then
4          $Q_{index} = \{q\}$; $C_{index} = q$; $index$++;
5      else
6          $add = false$;
7          foreach $Q_i$ do
8              if $d_{euc}(q.s, C_i.s) \leq r_i^* \wedge d_{euc}(q.t, C_i.t) \leq r_i^*$ then
9                  $Q_i = Q_i \cup q$; $add = true$; $break$;
10         if $add == false$ then
11             $Q_{index} = \{q\}$; $C_{index} = q$; $index$++;
12 return $\{Q_i\}$
```

The most important problem now is how to determine an appropriate radius r_i^* for each query cluster Q_i. Intuitively, if the origin and destination of C_i is quite close to each other, other queries that can be added into Q_i should be very similar to C_i in order to share some sub-paths of C_i, which leads to a small radius r_i^*. Conversely, if $C_i.s$ and $C_i.t$ is quite far away from each other, we can have a large r_i^*. In the following subsection, we will introduce *η-Approximate Batch Shortest Path Algorithm*, based on which the most approximate radius r_i^* of each cluster can be precisely determined.

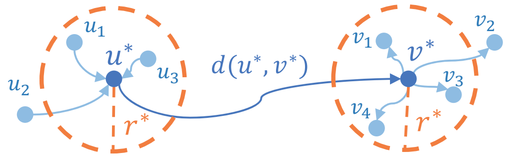


Fig. 4. η-Approximation

2) η-Approximate Batch Shortest Path: Well-separated pair decomposition [3] is the fundamental theory in construction of *Distance/Path Oracles* [27], [28]. Essentially, given two sets of vertices A and B, it provides an $\epsilon-$approximation of the distances $d(u, v)$, $\forall u \in A$ and $v \in B$:

$$(1 - \epsilon) \cdot d(u^*, v^*) \leq d(u, v) \leq (1 + \epsilon) \cdot d(u^*, v^*) \tag{7}$$
$$s.t. \quad d(u^*, v^*) \geq s \cdot r$$

where $u^* \in A$ and $v^* \in B$ are two representative vertices. We use $dia(A)$ to denote A's diameter, which is the maximum shortest distance among the vertices in it. We say A and B are *well-separated* if the minimum distance between them is at least $s \times r$, where $s > 0$ is a *separation factor* and $r = max(dia(A), dia(B))$. The relationship between the approximate ratio ϵ and s is $\epsilon = \frac{2}{s}$. The proof can be found in [27]. In this way, the approximation is converted to

$$(1 - \frac{2}{s}) \cdot d(u^*, v^*) \leq d(u, v) \leq (1 + \frac{2}{s}) \cdot d(u^*, v^*) \tag{8}$$

The diameter $r(A)$ is over-approximated by using the farthest distance from u^* to $u \in A$. Therefore, given any two vertex sets, we can determine its approximation error of using the shortest path between u^* and v^*.

However, in the above approach, the sets are predefined and the error is not globally bounded. Moreover, the approximation is on the shortest distance, but not the shortest path. Therefore, we further extend it to the path approximation and bound it by a global error η. Suppose we have a pair of vertex sets S and T. To find the η-approximate shortest paths between them, we first select $u^* \in S$ and $v^* \in T$, and find the shortest distance $d(u^*, v^*)$. For the remaining vertex pairs u and v, its approximate path is $u \rightarrow u^* \rightarrow v^* \rightarrow v$ with approximate distance of $d_{app}(u, v) = d(u, u^*) + d(u^*, v^*) + d(v^*, v)$. Because $d(u, u^*)$ and $d(v^*, v)$ are not larger than the diameter r, we have upper-bound below:

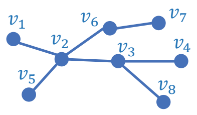
$$d_{app}(u, v) = d(u, u^*) + d(u^*, v^*) + d(v^*, v) \tag{9}$$
$$\leq d(u^*, v^*) + 2r$$
$$\leq d(u^*, v^*) + 2 \times \frac{d(u^*, v^*)}{s} \tag{10}$$
$$= d(u^*, v^*) \times (1 + \frac{2}{s}) \tag{11}$$
$$\leq \frac{d(u, v)}{1 - \frac{2}{s}} \times (1 + \frac{2}{s}) \tag{12}$$
$$= \frac{s + 2}{s - 2} \times d(u, v) \tag{13}$$

p_1	v_1, v_2, v_3, v_4
p_2	v_5, v_2, v_6, v_7
p_3	v_5, v_2, v_3, v_8

(a) Path List

v_1	p_1	v_5	p_2, p_3
v_2	p_1, p_2, p_3	v_6	p_2
v_3	p_1, p_3	v_7	p_2
v_4	p_1	v_8	p_3

(b) Inverted List



(c) Sub-Graph

Fig. 5. Cache Structure

(5) is because $r \leq d(u^*, v^*)/s$ and (7) is because $(1 - \frac{2}{s}) \cdot d(u^*, v^*) \leq d(u, v)$. As for the lower-bound, it is just $d(u, v)$. The error $\eta = \frac{d_{app}(u, v) - d(u, v)}{d(u, v)} = \frac{4}{s-2}$, and $s = \frac{4}{\eta} + 2$. In this way, when given an error bound η and $d(u^*, v^*)$, we can find two vertex sets around u^* and v^* that can approximate safely. The upper-bound of $d(u, u^*)$ is $r \leq d(u^*, v^*)/(s) \leq d(u^*, v^*)/(\frac{4}{\eta}+2) = \frac{\eta \cdot d(u^*, v^*)}{4 + 2\eta}$. Then all the u and v within the range of $r^* = \frac{r}{2}$ around u^* and v^* can be approximated with error of η.

Based on the above analysis, we can safely set the radius of each query cluster r_i^* as $\frac{\eta \cdot d(C_i.s, C_i.t)}{8 + 4\eta}$ for a given error bound η. Note that $d(C_i.s, C_i.t)$ denotes the shortest path distance between $C_i.s$ and $C_i.t$ which is unknown during the query set decomposition phase. Fortunately, based on our preliminary experimental analysis on a Beijing road network and the empirical results reported in [28] (Figure 8-b), most of the shortest path distances $d(u, v)$ are 1.2 times of the corresponding Euclidean distances $d_{euc}(u, v)$. Therefore, we set r_i^* as $\frac{1.2 \cdot \eta \cdot d_{euc}(C_i.s, C_i.t)}{8 + 4\eta}$ in this work. This also verifies our intuition that the radius of a query cluster r_i^* is positively proportional to the spatial distance between the origin and destination of the cluster center $d_{euc}(C_i.s, C_i.t)$.

V. Batch Query Answering

In this section, we present two batch shortest path algorithms, both of which are the extension of existing batch algorithms and could benefit from the decomposition results. Specifically, Section V-A describes the *Local Cache*, which uses the results from *Zigzag Decomposition* and *Search Space Estimation Decomposition*. It is more suitable to answer queries whose sources and targets are not too far away from each other. Section V-B introduces the error-bounded *Region-to-Region* batch algorithm which takes the results of *Coherence-Aware Co-Clustering* as input. It is suitable to answer the long queries.

A. Local Cache-based Query Answering

The cache-based query answering makes use of the sub-path property of shortest path: any sub-paths of a cached path can be answered directly. We extend the shortest path caching technique in [29] based on two observations: 1) one query q tends to hit the paths in or passing through the same local area with it; 2) the paths with longer distance are more likely to be hit by larger number of queries, and queries with shorter distance are more likely to hit the cached paths. Therefore, we aim to build cache locally rather than using the global cache in query answering. Moreover, we prioritize the longer queries and cache them under the cache size restriction. Finally, we present how to use the local cache in the dynamic environment.

1) Cache Structure and Query Processing: In the cache structure design, two problems need to be considered: 1) When there comes a query, how can we decide whether it can be hit or not? 2) If the query is hit, how to retrieve the path? To address the first problem, we build an inverted list from vertex to its cached paths. For example, we have three cached paths p_1, p_2, p_3 as shown in Figure 5. When path p_1 is inserted into the cache, we add its path number to the inverted list of the vertices v_1, v_2, v_3, v_4 respectively. When a new query $q(s, t)$ arrives, we first check whether there exists the same path number in s's and t's inverted lists. If there is no common path number, we process it using A^* directly and cache it as long as the current cache size dose not exceed the cache limit. If q can be answered with cache, we need to retrieve its shortest path next, where we use a subgraph for path retrieval. The subgraph is created by the cached paths. The subgraph formed by the three cached paths is shown in Figure 5. For query $q(v_1, v_3)$, the inverted lists of v_1 and v_3 contain the same path number p_1 and we start from the start node v_1 and search its adjacent nodes whose inverted list contain the path number p_1 and continue the search until we reach the target node v_3.

2) Local Cache vs Global Cache: The global cache [29] is based on query log analysis. It identifies some important paths and uses them as static cache. Our local cache extends it in the following aspects. Firstly, we build a cache for each cloud-shaped query set generated by the *1-N Zigzag Decomposition* or the *Search Space Estimation Decomposition*, because they have similar search area and have higher possibility to overlap. When a query set finishes answering, we clear the cache and build another one based on the next query set. In this way, although our cache is the same as the global cache's size $|GC|$, our total cache size is actually $|\hat{Q}| \times |GC|$. Secondly, for queries in each set, we answer them from the longest to the shortest, due to the second observation. Lastly, although the vertices in road networks are intersections, one intersection can be denoted by several vertices in real-life networks. For example, an intersection of two main roads may use four vertices express, and a roundabout may use tens of vertices to store. Obviously, these vertices are conceptually the same for a path query. Therefore, we use a super vertex to represent several nearby vertices. In this way, higher hit ratio and smaller cache size can be achieved.

3) Dynamic Batch Query Answering: Suppose the road weight w changes every T time and the queries come in several batches B during T. When the first B_1 comes, we decompose it and create caches to answer its queries. After that, we keep the caches because w remains the same at this stage. When B_i comes, we also decompose into subsets. However, before creating their caches, we first compute the similarity of them with the existing caches using the overlap coefficient introduced in Section IV-B3. It should be noted that the *Search Space* also needs to compare the cluster direction. The queries are answered with the most similar cache if its similarity is larger than a threshold. Otherwise, it creates a new cache. When w changes, the caches are destroyed and new caches are created by the coming batch.

B. Approximate Region-to-Region Shortest Path

This algorithm is based on the approximate bound described in Section IV-C2. Unlike Section IV-C2, which uses approximate distance of $d(u^*, v^*)$ to decompose the queries quickly, we have to use the accurate $d(u^*, v^*)$ to guarantee that the error is bounded by η. Due to the nature of the *path coherence*, it is suitable for the batch queries of regions far away from each other. Otherwise, the region diameter would be too small to cover enough queries. Although the guaranteed radius is r^* because the original *well-separated* theorem requires the diameter of the subgraphs smaller than $2r^*$, we can push the region radius to $2r^*$ with the following theorem:

Theorem 1: Given a pair of representative vertex u^* and v^*, and a computed diameter $2r^*$, all the queries (s, t) within the range of $2r^*$ of u^* and v^* can be approximated with error η.

Proof: We only need to prove one end of the query, so we use u^* as example. Given a set of starting vertices $S = \{s_i\}$ around u^*, with $d(s_i, u^*) \leq 2r^*$, we can view each pair of (s_i, u^*) as a subgraph. In this way, the $dia(s_i, u^*) \leq 2r^*$ holds, so we can use u^* to approximate each s_i safely. ■

Here is another way to view this proof: the original *well-separated* supports the approximation of the S using any $s_i \in S$, while we only use a pre-determined u^* to approximate other vertices. Therefore, there is no need to constrain the distance between any s_i and s_j.

Given a query set Q_i, we identify the region pairs greedily. Firstly, we select the longest query $q(u^*, v^*)$ in the remaining query set and run an A^* search to get its distance $d(u^*, v^*)$. After that, we search from u^* and v^* to collect the source set S and destination set T. The search stops when the distance is larger than $2r^*$. A query point u (or v) is considered as a query candidate and is put into a set C_s (or C_t). It should be noted that both searches from u^* and v^* should run forwardly and backwardly to satisfy the diameter definition. Next, we check the query points in C_s and C_t. A query is considered as a query candidate and put in the candidate set C_q only if its source and target appear in C_s and C_t, respectively. The pseudo-code is provided in Algorithm 2.

An example is shown in Figure 6. $q^* = (u^*, v^*)$ is first answered with an A^* search. After that, a pair of forward and backward searches are performed on u^*. Because both of the forward and backward distances are smaller than $2r^*$, u_1, u_2 and u_4 are put in C_st. u_3 is excluded because $d(u^*, u_3)$ exceeds the diameter even though $d(u_3, u^*) \leq 2r^*$. Similarly, two searches are performed from v^* and $C_t = (v_1, v_2, v_3)$ is obtained. After that, q_1 and q_2 can be answered because they appear in both of C_s and C_t. q_3 and q_4 will be settled in the next rounds.

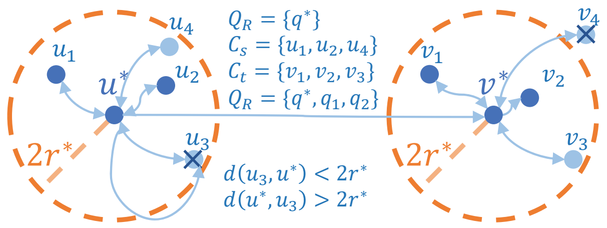


Fig. 6. Region Vertex Validation

Algorithm 2: Region-to-Region Shortest Path

```
Input: $Q$: shortest path query set
Output: $D = \{d_i\}$
1 foreach $q_i(u^*, v^*) \in Q$ do
2     $d(u^*, v^*) \leftarrow d(u^*, v^*); r^* \leftarrow \frac{\eta \cdot d(u^*, v^*)}{8 + 4\eta}; Q \leftarrow Q \backslash q_i$
3     $C_s \leftarrow \{Dij(u^*) < 2r^* \text{ and } rDij(u^*) < 2r^*\}$;
4     $C_t \leftarrow \{Dij(v^*) < 2r^* \text{ and } rDij(v^*) < 2r^*\}$;
5     $Q_R = \{d(u^*, v^*)\}; C_q(C_s, C_t) = S.q \cap T.q$;
6     foreach $q \in C_q$ do
7         $d(q) = d(q.s, u^*) + d(u^*, v^*) + d(v^*, q.t)$;
8         $Q \leftarrow Q \backslash q; D \leftarrow D \cup d(q)$
9 return $D$
```

VI. Experiment

In this section, we experimentally evaluate the proposed query decomposition methods and batch algorithms in real-life road network against the current state-of-the-art methods.

A. Experiment Setup

All the algorithms are implemented in C++, compiled with full optimizations, and tested on a Dell R730 PowerEdge Rack Mount Server which has two Xeon E5-2630 2.2GHz (each has 10 cores and 20 threads) and 378G memory. The data are stored on a 12 × 4TB Raid-50 disk.

1) Dataset and Query Sets: We obtain our road network of Beijing from *NavInfo*. It consists of 312,350 intersections and 403,228 roads, which covers a 184km × 185km spatial range.

The query data is sampled from Beijing taxi trajectory collected from 1 March 2016 to 31 March 2016. Totally, it contains more than 12M trajectories. Each pair of starting and ending location is regarded as a shortest path query. For the *Cache* test, we choose the queries with distance shorter than 50km randomly. For the *Region-to-Region* test, we choose the queries with distance longer than 30km and shorter than 80km randomly. We generate sets for both tests with size of 10k, 100k, 500k and 1M queries. To further test the effectiveness in real-life, we use 40 threads to simulate 40 servers.

2) Comparison Methods: For the first *Cache* experiment, we use *ZLC* to denote *Zigzag Decomposition* with *Local Cache*, *SLC* to denote *Search Space Estimation* with *Local Cache*, and compare them with the *Global Cache (GC)* [29], and A^*. *SLC* has its distance sorted version *SLC-S* to show the influence of query order, and the random version *SLC-R* for comparison. The decomposition time and query answering time are recorded separately. We use the first 20% queries in each test set to construct the cache. The size of the cache it generates is denoted as $|GC|$, and we use it as the size limit of each local cache. Apart from the query answering time, we also compare the *cache hit ratio* of each method. The hit ratio is computed as $R_h = \sum h_i/|Q_i|$, where $|h_i|$ is number of query hit by each local cache. Finally, we test the influence of cache size on the query answering time and hit ratio. The base cache size is $|GC|$, and we compare the performance with k times of $|GC|$, where $k = 70\%, 80\%, 90\%, 100\%$ of $|GC|$.

For the second *Region-to-Region* experiment, we denote our *Region-to-Region with Coherence-Aware Co-Clustering* as *R2R-S* and *R2R-R* with error bound $\eta = 0.05$. In *R2R-S*, we search the query pairs from the farthest to the nearest. In *R2R-R*, we search the queries randomly. We compare with *k-Path* [21], *Zigzag-Petal* [34], and A^*. *k-path* is slow when k is larger than 1 because it takes much longer time to find the $top - k$ path than the shortest path. Therefore, we choose $k = 1$ for *k-path* to test its fastest query time. In fact, because our *R2R* also only runs one shortest path between regions, it is fair to compare their query time and error. Moreover, *k-Path*'s original decomposition method is too slow to work on the whole graph, so we use our *Co-Clustering* to decompose the query set for it. We do not compare *Group* [25] here because it is essentially an inaccurate multi-run A^*, and its running time becomes much slower than A^* as the query number increases. Apart from the query time, we also compare the average error and maximum error of these approximate algorithms. The error is computed as $\epsilon = \sum |d_i^* - d_i|/d_i$, where d_i is the actual distance and d_i^* is the approximate distance. The average error is computed on all the approximate queries, excluding the accurate ones.

Finally, the multi-thread experiment compares total query answering time under 40 threads. Besides, we also present the construction time of the state-of-the-art index-based approaches *CH* [12] and *PLL* [1].

B. Decomposition Time

Figure 7-(a) shows the running time of our three decomposition methods. Naturally, all of their running time increases as the query size grows. However, even for the largest 1M query set, the slowest *Zigzag* takes 4.6s, while the fastest *Co-Clustering* takes only 1 second. Firstly, *Zigzag* is always the slowest because it creates lots of forward and backward *1-N* petals first. Since each query exists in both of forward and backward petals, it takes time to remove the redundancy as the petals merge. Moreover, it tends to generate several huge clusters and many small cluster, which also prolongs the merge time. Secondly, the *Search-Space* is always less than half of *Zigzag*'s. Although it has a road direction summarization phase, it takes less than 1ms. With the adaptive grid structure, the search space estimation only needs to find and check the related grids. Lastly, the *Co-Clustering* is the fastest as its search space is bounded by a radius. In other words, it has the smallest number of covered grids. In addition, the long queries actually have fewer clusters than the shorter queries.

TABLE I
CACHE SIZE (MB)

	10K	**100K**	**500K**	**1M**
20%$\|GC\|$	3.02	23.83	126.83	224.15

C. Cache-Based Batch Query Answering

1) Hit Ratio: As shown in Figure 7-(b), the hit ratio increases as the query size becomes larger. The hit ratio of *GC* is always the lowest while our local caches have higher hit ratio. The sorted version of *SLC-S* performs better than the unsorted *SLC-R* because it can cache more long paths. The *ZLC* is worse than *SLC* due to its clustering result. Therefore, we use *SLC-S* to further test the influence of cache size on hit ratio, as shown in Figure 7-(c). The highest line is the the same as the high in Figure 7-(b) because they are the same result. As the cache size decreases, the hit ratio decreases accordingly. The cache size of each query set is shown in Table I. It grows linearly as the query size increases.

2) Running Time: The query answering time is shown in Figure 7-(d). The running time of A^* is collected along with the cache query to avoid its performance over-boost after warming up. For each query size, the *SCL-S* always performs best and *SLC-R* follows it, and *ZLC* is just slightly better than *GC*, because *ZLC* tends to create some huge clusters, which is similar to *GC*. Besides, it also shows that the *Search Space Estimation* is a better decomposition method than *Zigzag*, in terms of both the decomposition time and the query cluster quality. The query becomes faster as the query number grows. The reason for the cache being faster is that larger number of queries are answered by cache. The reason for A^* being faster is just the algorithm warmed up. Figure 7-(e) shows the influence of cache size on running time. Within each query size, query time becomes longer as the cache size reduces, along with the hit ratio. In summary, our decomposition methods can improve the cache-based batch shortest path query by incorporating the local information into cache construction.

D. Approximate Region-to-Region Batch Query Answering

1) Query Time: Figure 7-(f) illustrates the average query answering time of each region-based algorithms under different query sizes.

First of all, A^*, as the baseline, is the slowest when the query set is larger than 10k. *Zigzag-Petal* is the slowest when the query set is 10k because this query set does not contain many *1-N* queries. Therefore, *Zigzag-Petal* spends lots of time on useless decomposing and redundant query removing. As the query size grows, *Zigzag-Petal* performs better and better because more *1-N* queries appear.

Secondly, the *k-path* is the fastest when the query set is 10k but becomes slower when the query set grows larger. This is due to its way of computation. For each dumbbell cluster,

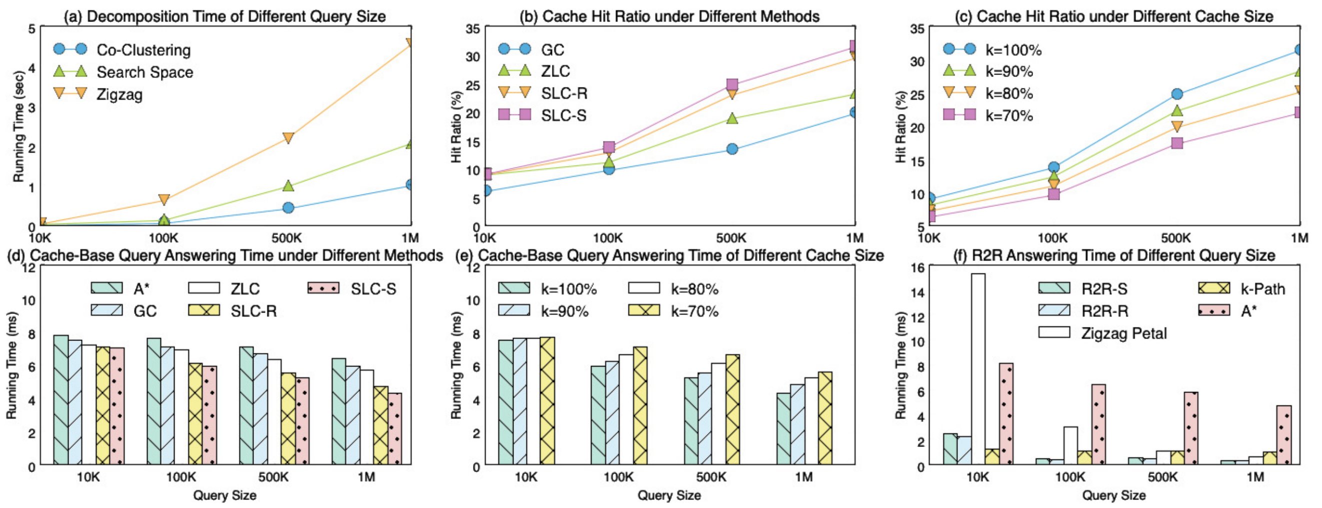


Fig. 7. Experimental Results of the Single Thread Test

TABLE II
REGION-BASED ERROR

	Average Error (%)		Max Error (%)	
	R2R	***k-Path***	***R2R***	***k-Path***
10K	0.99	3.26	4.59	30.15
100K	0.973	2.83	4.56	30.52
500K	1.33	2.56	4.79	30.93
1M	1.32	2.54	4.85	30.813

it only needs one long-range search to get the shortest path among the border node pairs. Then for each source and target, it has to run a *Dijkstra* to the borders to find the shortest paths. Although we choose *k=1* here, we still have to keep this all-border result because its original intention is to find the minimum of $s \to b_1 \to b_2 \to t$. Therefore, as the number of sources and targets in each region grows, it takes longer time to compute.

Finally, our *R2R* methods are faster when the query set is larger than 10k. Specifically, *R2R-R* is slightly faster than *R2R-S*. The reason for this might be the randomly selected center node can cover more query nodes than the farthest node. Even though the farthest node has the largest radius, it is not large enough to cover more nodes. If more query nodes are covered, fewer inner clusters are generated, then the fewer long-range search is needed.

2) Average Error: Table II shows the errors of each approximate region algorithms. The error value is in percentage. The average error of *R2R* is around 1%, while the *k-path*'s is 2% to 3%. As for the maximum error, *R2R* is bounded by the predefined 5%, while the *k-path*'s error is not bounded and can be as worse as 30%.

In summary, *R2R* can answer the batch long range queries quickly with approximate bound guaranteed.

E. Multi-Thread Evaluation

Figure 8 shows the result of multi-thread test. Apart from the query answering time, we also present the index construction

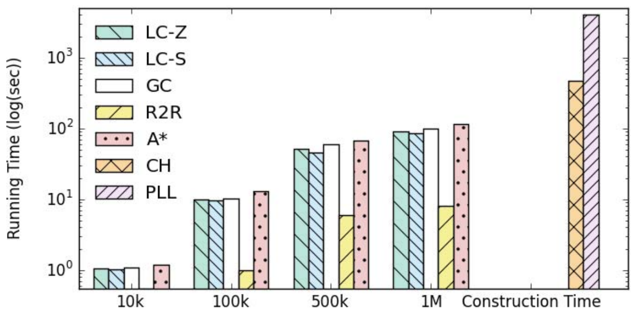


Fig. 8. Multi-Thread Running Time

time of *CH* and *2 Hop Labeling* (using *Pruned Landmark Labeling* [1]) in our network. Because their time is much longer than our query answering time, we show the experiment result in log second. It proves that they are not suitable to apply in dynamic environment no matter how fast they can answer a query, because the traffic condition could have already changed before their constructions finish.

Although it could take more than one minute to finish answering 1M queries, our cache approaches are still faster than A^*, which is actual best approach in practice. As for the *R2R*, it is much faster than the others especially when the query distance is large.

VII. CONCLUSION

In this paper, we study the problem of batch processing of shortest path queries. Although there exists many index structures to find the shortest path quickly, they cannot adapt to the dynamic environment because they all need a long preprocessing step. Therefore, batch computation is preferred as no preprocessing is needed. However, the existing algorithms have different limitations: cache-based method has low hit ratio, and approximate methods have no error bound and cannot decompose queries over the entire road network. Therefore, we improve them by proposing three query decomposition algorithms to cluster the query into smaller sets and two batch algorithms that benefit from them. Specifically, for the *cache-based* method, we propose two decomposition methods to generate cloud-shape query sets: *Zigzag Decomposition* that combines the existing *petal* clusters, and *Search Space Estimation* that takes the advantages of search behavior analysis. After that, *Local Cache* is proposed to provide cache with higher hit-ratio. For the approximate method, we first provide a fast *Coherence-Aware Co-Clustering* method to decompose the queries. Then we propose the *Region-to-Region* algorithm to answer the longer queries in batch with error-bounded. Our extensive experiments show the *Local Cache* has higher hit ratio than the global one, and *Region-to-Region* beats the state-of-the-art methods in terms of both query time and approximation ratio.

References

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2013.

[2] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 97–105. Society for Industrial and Applied Mathematics, 2009.

[3] P. B. Callahan. *Dealing with Higher Dimensions: The Well-separated Pair Decomposition and Its Applications*. PhD thesis, Baltimore, MD, USA, 1995. UMI Order No. GAX95-33229.

[4] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[5] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph partitioning with natural cuts. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1135–1146, 2011.

[6] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In *International Workshop on Experimental and Efficient Algorithms*, pages 52–65. Springer, 2007.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[8] G. D'Angelo, M. D'Emidio, D. Frigioni, and C. Vitale. Fully dynamic maintenance of arc-flags in road networks. In *International Symposium on Experimental Algorithms*, pages 135–147. Springer, 2012.

[9] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[10] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *Proceedings of the VLDB Endowment*, 6(6):457–468, 2013.

[11] J. Gan and Y. Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 519–530. ACM, 2015.

[12] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[13] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[15] L. Li, W. Hua, X. Du, and X. Zhou. Minimal on-road time route scheduling on time-dependent graphs. *Proceedings of the VLDB Endowment*, 10(11):1274–1285, 2017.

[16] L. Li, W. Hua, and X. Zhou. Hd-gdd: high dimensional graph dominance drawing approach for reachability query. *World Wide Web*, 20(4):677–696, 2017.

[17] L. Li, J. Kim, J. Xu, and X. Zhou. Time-dependent route scheduling on road networks. *SIGSPATIAL Special*, 10(1):10–14, 2018.

[18] L. Li, S. Wang, and X. Zhou. Time-dependent hop labeling on road network. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 902–913, April 2019.

[19] L. Li, K. Zheng, S. Wang, W. Hua, and X. Zhou. Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory. *The VLDB Journal—The International Journal on Very Large Data Bases*, 27(3):321–345, 2018.

[20] Y. Li, M. L. Yiu, N. M. Kou, et al. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11(4):445–457, 2017.

[21] H. Mahmud, A. M. Amin, M. E. Ali, T. Hashem, and S. Nutanong. A group based approach for path queries in road networks. In *International Symposium on Spatial and Temporal Databases*, pages 367–385. Springer, 2013.

[22] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)*, 11:2–8, 2007.

[23] T. A. J. Nicholson. Finding the shortest route between two points in a network. *The computer journal*, 9(3):275–280, 1966.

[24] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*, pages 709–724. ACM, 2018.

[25] R. M. Reza, M. E. Ali, and T. Hashem. Group processing of simultaneous shortest path queries in road networks. In *2015 16th IEEE International Conference on Mobile Data Management*, volume 1, pages 128–133. IEEE, 2015.

[26] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 43–54. ACM, 2008.

[27] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *2009 IEEE 25th International Conference on Data Engineering*, pages 652–663. IEEE, 2009.

[28] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *Proceedings of the VLDB Endowment*, 2(1):1210–1221, 2009.

[29] J. R. Thomsen, M. L. Yiu, and C. S. Jensen. Effective caching of shortest paths for location-based services. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 313–324. ACM, 2012.

[30] J. R. Thomsen, M. L. Yiu, and C. S. Jensen. Concise caching of driving instructions. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 23–32. ACM, 2014.

[31] D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics (JEA)*, 10:1–3, 2005.

[32] K. Xie, K. Deng, S. Shang, X. Zhou, and K. Zheng. Finding alternative shortest paths in spatial networks. *ACM Transactions on Database Systems (TODS)*, 37(4):29, 2012.

[33] M. Zhang, L. Li, W. Hua, and X. Zhou. Batch processing of shortest path queries in road networks. In *Australasian Database Conference*, pages 3–16. Springer, 2019.

[34] M. Zhang, L. Li, W. Hua, and X. Zhou. Efficient batch processing of shortest path queries in road networks. In *2019 20th IEEE International Conference on Mobile Data Management*. Springer, 2019.

[35] B. Zheng, H. Su, W. Hua, K. Zheng, X. Zhou, and G. Li. Efficient clue-based route search on road networks. *IEEE Trans. Knowl. Data Eng.*, 29(9):1846–1859, 2017.

[36] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 857–868. ACM, 2013.