Backbone Index to Support Skyline Path Queries over Multi-cost Road Networks

Qixu Gong
Computer Science, New Mexico State University
Las Cruces, NM, USA
qixugong@nmsu.edu

Huiping Cao
Computer Science, New Mexico State University
Las Cruces, New Mexico, USA
hcao@nmsu.edu

ABSTRACT

Skyline path queries (SPQs) extend skyline queries to multi-dimensional networks, such as multi-cost road networks (MCRNs). Such queries return a set of non-dominated paths between two given network nodes. Despite the existence of extensive works on evaluating different SPQ variants, SPQ evaluation is still very inefficient due to the nonexistence of efficient index structures to support such queries. Existing index building approaches for supporting shortest-path query execution, when directly extended to support SPQs, use unreasonable amount of space and time to build, making them impractical for processing large graphs. In this paper, we propose a novel index structure, *backbone index*, and a corresponding index construction method that condenses an initial MCRN to multiple smaller summarized graphs with different granularity. We also present efficient approaches to find approximate solutions to SPQs. Our extensive experiments on *nine* real-world large road networks show that our approaches can efficiently find meaningful approximate SPQ solutions by utilizing the compact index. The backbone index can be constructed with reasonable time, which dramatically outperforms the construction of other types of indexes for road networks. As far as we know, this is the first compact index structure that can support efficient approximate SPQ evaluation on large MCRNs.

1 INTRODUCTION

Skyline path queries (SPQs) extend skyline queries to multi-dimensional networks (MDNs) [29]. They generalize shortest-path queries over single-cost graphs. Given an MDN, SPQs return a set of non-dominated paths between two given graph nodes. In this paper, we study SPQs on multi-cost road networks (**MCRNs**), which are the most widely studied MDNs while considering SPQs [17, 20, 29, 44, 46]. In real applications, the multiple edge costs of MCRNs can represent different things such as distance, travel time, the number of traffic lights, gas consumption, etc. Consider an application of utilizing a public transportation system, the walking distance, the time traveled using the public transportation system, and the number of transitions between different transportation lines can be the different weights. SPQs over a public transportation system find Pareto optimal solutions of bus routes that can take a user from a given bus stop to a target bus stop, where the expense and travel time of those routes do not dominate each other. In this scenario, a user may not like the path (say p_{minE}) with the lowest expense but a long travel time or the path (say p_{minT}) with the shortest travel time and a higher expense. Instead, the user may want to use another path, which either (a) has a slightly higher expense and much less travel time than p_{minE}, or (b) has a slightly longer travel time and much lower expense than p_{minT}.

The evaluation of SPQs is very time consuming due to the large number of solutions [20, 29] and the vast search space. Many works attempt to accelerate the query process by reducing the search space. In [29], the landmark index [28] is utilized to stop growing a path when its upper-bound cost is dominated by the cost of at least another result. To address the cold-start problem in [29], Yang et al. [45] use the shortest path found for each dimension as the initial results. Other works define different variations of SPQs and propose specialized query processing approaches by utilizing the properties of their SPQs to reduce the search space [7, 12, 17, 20, 44].

A general idea to speed up query evaluation is to utilize indexes. The *major challenge* of designing index structures for SPQs is the large number of skyline paths that need to be pre-calculated. Multiple skyline paths (not just one shortest path) exist between two nodes on an MCRN. Traditional indexes that are used to support location-based queries (e.g., shortest path queries) [18, 26, 30, 32, 50], if directly adopted to solve SPQs, either incur expensive index building and use much space (partition-based method), or increase node degrees and the number of edges. As a consequence, the query performance deteriorates. To the best of our knowledge, no compact index structures exist to support efficient SPQs.

We conduct an *extensive analysis* [19] of an improved SPQ evaluation method of [29] on two real-world MCRNs to understand how the characteristics of road networks (e.g., high node-degree distribution) and queries (e.g., long paths between the query nodes) affect query performance. The study shows that the existing methods (even with improvements) are too inefficient to evaluate SPQs even on small MCRNs.

Considering the above situations, this paper proposes a hierarchical index structure to support getting approximate answers for SPQs. The design utilizes the *concept of backbone*, which captures the core graph topology, to abstract the original graph. The idea is similar to intuitive human behavior when navigating from a source to a destination in a road network. Let us consider a scenario that a student needs to drive from his/her university in city A to a hotel in city B. He/she first finds the paths to the main street from the university's district. Then, the routes from the main street to highway entrances of city A are identified. Highways between the cities are utilized to lead him from city A to city B. Then, a similar idea is adopted to find the paths from freeway ramps to the hotel in city B. As Figure 1 illustrates, the search involves three levels: the district level (paths to the main street), the intra-city level (routes to highways' entrances), and the inter-city level (highways from city A to city B).

The idea of highway entrances is also utilized in partition-based approaches [26, 30, 50] as border nodes between partitions. These methods divide the original graph into non-overlapping partitions and store extra information (e.g. the shortest path weight) between every pair of border nodes for the partitions. The goal of their design is to minimize the number of border nodes. **Our design is different** in that we do not minimize the number of entrance

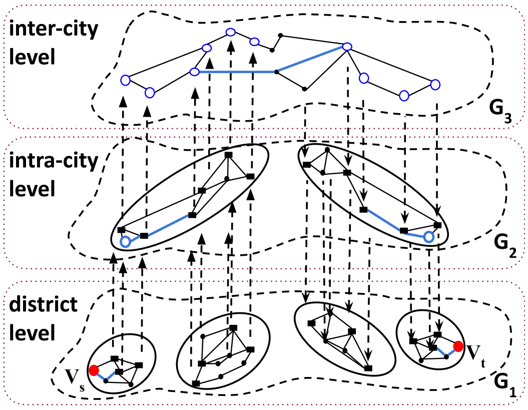


Figure 1: Example of a backbone index

nodes, instead the entrance nodes are used to preserve the overall topology of the original network while conducting network summarization.

Our proposed backbone index is a hierarchical structure that tries to preserve the topology of the original graph by condensing/summarizing dense local graph units level by level. The abstracted graphs at higher levels are more abstract than the lower-level graphs, while maintaining the topological structure.

The main **contributions** of our work are as follows.

- We propose a novel hierarchical index based on the concept of backbone and clustering to abstract the original graph to several summarized graphs with different summarization granularity. The index is utilized to find approximate answers to SPQs.
- We present an efficient index building algorithm and several variations. The index construction algorithm summarizes a graph by reducing the density of its dense local units (or clusters).
- A query evaluation algorithm is proposed to get approximate answers of SPQs. The algorithm combines a dynamic-programming search strategy at lower index levels and an optimized many-to-many landmark-based skyline search algorithm at the most abstracted graph level. The approximate answers are more succinct than the exact answers and enable users to focus on choosing from fewer good results.
- We analyze the quality of the approximate solutions and the complexity of our proposed methods.
- We conduct extensive experiments using *nine* real-world datasets, including large road networks with *millions of nodes and edges*.

The rest of the paper is organized as follows. Section 2 discusses existing works that are related to our study. Section 3 defines the research problem, related concepts, and notations. Our proposed index structure and the query algorithm are presented in Sections 4 and 5. Experimental results are reported in Section 6.

2 RELATED WORKS

2.1 Skyline queries on road networks

The SPQ problem over an MCRN is first proposed and studied in [29, 39]. Kriegel et al. [29] propose to use landmark index to calculate lower bounds of paths and reduce the search space of SPQs. Tian et al. [39] utilize the partial path dominance test to prune search space. Yang et al. [45] define a stochastic dominance relationship. Instead of using the landmark index, the lower bound of the cost on each dimension is calculated using a reverse Dijkstra [15] search.

More recent works evaluate different SPQ variants. The work [17, 44] conducts SPQs over moving objects on single-dimensional road networks with multi-attributed points of interest (PoIs). Gong et al. [20] propose a Constrained Skyline Queries problem assuming that PoIs can be off an MCRN. The work [31] proposes a new concept of skyline groups by considering the strength of social ties and the spatial distance in a single-dimensional road network.

The previous techniques (except [29]) answer skyline queries without the support of any index structures. Although using the landmark index [29] and finding shortest paths on each dimension [45] are efficient ways to prune the search space, the query process using these techniques are still very inefficient when node degrees are high or the number of hops between query nodes is large. In addition, constructing landmark index on a large graph is expensive.

The work [47] is most similar to ours. It proposes a partition-based single-level index. However, their index supports the optimal path finding problem instead of SPQs. The query performance decreases dramatically as the degree of border nodes grows because one border node in a partition connects to multiple border nodes (or entrances) of its neighbor partitions.

2.2 Location-based queries on road networks

The shortest-path query is one type of fundamental location-based queries for graph structured data. The Dijkstra [15] and the A* [23] algorithms are the most successful and widely used methods. These traditional search methods are not practical to work for the large graphs collected in recent years. The design and use of an index structure to keep pre-calculated path information is inevitable.

For road networks, graph-partition [26, 30, 32, 50] and shortcut-based [18, 43] approaches are two typical ways to design indexes to support location-based queries. When such approaches are directly utilized to process SPQs, the partition-based methods find enormous number of skyline paths when the length of paths between partitions is long, which leads to expensive index construction and large disk use. The shortcut-based approaches create shortcuts between two graph nodes. The number of shortcuts grows exponentially with the increase of node degrees and the length of paths between graph nodes. The huge number of shortcuts does not improve the query performance, but deteriorates the query evaluation. Our preliminary analysis [19] has verified the statements about both types of methods. Several partition-based methods [26, 30, 50] minimize the number of border nodes so that fewer shortest paths need to be found in a partition. This does not work to process SPQs because the number of skyline paths and search time increase dramatically in dense partitions, which has nothing to do with the number of border nodes.

Recent graph-partition based attempts [13, 35, 49] utilize tree decomposition as the pre-process step for building hubs or shortcuts among tree nodes. These methods either (i) face the issue of huge disk use and high computational cost while storing the skyline path information from each tree node to its ancestor tree nodes [13, 35] or (ii) generate large number of shortcuts from each tree node to its neighbors in the SPQ setting. Other approaches [6, 22, 37] to answer shortest-path queries apply Breadth-First Search (*BFS*)-based methods with specially designed pruning conditions. They run slowly if directly adopted to answer SPQs for graphs with high node degrees. Different from all the existing approaches, our proposed approach condenses *local dense units* of a graph (i.e., inside a partition) and utilizes such condensed partitions to support SPQ evaluation.

2.3 Finding backbones on graphs

Graph backbone extraction identifies critical nodes and edges to preserve the topology and other essential information of a graph. Recent works [10, 21, 25, 36, 38] study the backbone extraction problem for different networks with specialized research interests. In [36], the authors identify a network's backbone that consists of a set of paths maximizing the Bimodal Markovian Model likelihood. The work [21] finds a tree-like backbone structure utilizing both the node attribute and the graph topology in geo-social attribute graphs. Graph backbone can also be extracted using the graph structure. The work [25] merges nodes and edges by creating shortcuts with the intention to preserve the topology of the original graph. The works in [10, 38] define a criterion to examine the importance or relevance to a network, and adopt strategies for edge sampling [8] or edge filtering (or pruning) [10, 14] to create backbone structures.

The above methods either conduct high-cost inference that is not practical on large graphs, or dramatically increase the graph size that causes the degradation of queries, or define specific criteria [11, 14, 33] for specialized MCRNs. Thus, they cannot be directly applied to build indexes to support SPQs over general MCRNs. Moreover, most of the existing methods [14] cannot guarantee the connectivity of the extracted backbone graph.

3 PROBLEM STATEMENT

A multi-cost road network (**MCRN**) is represented as an undirected graph $G=(V,E,W)$ where V is the set of nodes, E is the set of edges where $E \subseteq V \times V$, and $W \in \mathbb{R}^{|G.E| \times d}$ is a weight tensor. Let $|G.V|$ and $|G.E|$ be the number of graph nodes and edges respectively. Each edge $e \in E$ is associated with a d-dimensional cost vector w, where w_i is the value of the i-th cost of edge e. Roads have directions. Two roads with opposite directions generally connect two same nodes, and the costs of the two opposite directed roads do not differ much. Given these, we model a road network as an undirected graph. When road networks are modeled as directed graphs, our method can be easily extended to work (more discussions see the end of Section 4.3.1).

A **path** p between a node v_s and another node v_t is denoted as $p(v_s \rightsquigarrow v_t)$. The **cost of a path** p, cost(p), is the summation of the weights of the edges of p on each dimension. The cost(p) is d-dimensional. The **length of a path** is the number of edges in the path. Given two nodes, the **path hop** is defined to be the average length of all the shortest paths when different single dimension is utilized. Given two paths p_i and p_j where the ending node of p_i is the same to the starting node of p_j, p_i and p_j can be **concatenated** as $p_i||p_j$, where $||$ denotes the concatenation of two paths.

3.1 Path domination and skyline path queries

For multiple paths with d-dimensional cost, we adopt their domination relationship from [20, 29] and define it below.

Definition 3.1 (Path domination). Given two paths p and p' with multi-dimensional costs, the path p dominates another path p', denoted as $p \prec p'$, if and only if $\forall i \in [0,d]$, cost(p)[i] $\leq$ cost(p')[i] and $\exists i \in [0,d]$, cost(p)[i] $<$ cost(p')[i].

Intuitively, p dominates p' when $cost(p)$ is not worse than $cost(p')$ on each dimension, and is strictly better than $cost(p')$ on at least one dimension.

Definition 3.2 (Skyline Path Query (SPQ)). Given a graph G representing an MCRN, a skyline path query (SPQ) is denoted with a starting node v_s and a target node v_t. The answer to a SPQ is a set of paths $\mathbb{P}$ satisfying (1) $\forall p \in \mathbb{P}$, p is from v_s to v_t, (2) $\forall p' \notin \mathbb{P}, \exists p \in \mathbb{P}$ s.t. $p \prec p'$, and (3) $\forall p \in \mathbb{P}, \nexists p' \in \mathbb{P}$ s.t. $p' \prec p$.

A path $p(v_s \rightsquigarrow v_t) \in \mathbb{P}$ is called a **skyline path** from v_s to v_t. Where there is no ambiguity in the context, we use p to represent $p(v_s \rightsquigarrow v_t)$. Given two nodes, one SPQ returns a set of paths between the nodes while such paths do not dominate each other.

3.2 Degree pairs and single segments

Our approach utilizes graph density information. To better capture and describe the density of subgraphs in a graph, we introduce several concepts: degree pairs, degree-1 edges, and single segments.

Definition 3.3 (Degree Pair). Given an edge e with its two end nodes s_e and t_e, the degree pair of e, $DP(e) = \langle e.first, e.second \rangle$, is defined as follows.

$$DP(e) = \begin{cases} \langle deg(s_e), deg(t_e) \rangle & deg(s_e) \leq deg(t_e) \\ \langle deg(t_e), deg(s_e) \rangle & Otherwise \end{cases} \quad (1)$$

where $deg(v)$ is the degree of the node v. As the definition shows, the elements in the degree-pair tuple are ordered where the first element $e.first$ is always smaller than or equal to the second element $e.second$. An edge that has a degree pair $\langle 1, x \rangle$ $(x \geq 1)$ is called a **degree-1 edge**.

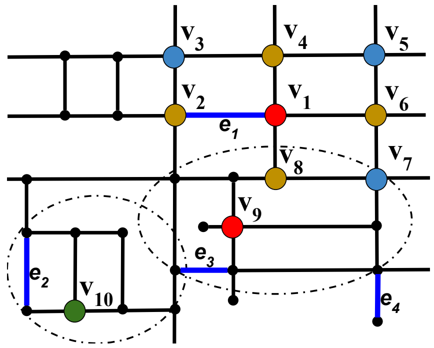


Figure 2: Degree pair example, where $DP(e_1)$=$\langle 4,4 \rangle$, $DP(e_2) = \langle 2,3 \rangle$, $DP(e_3) = \langle 3,4 \rangle$, and $DP(e_4) = \langle 1,4 \rangle$.

Example 3.4. Let use Figure 2 to demonstrate the concept of degree pairs. For e_1, whose two end nodes are v_1 and v_2, the degree pair $DP(e_1)$ is $\langle 4,4 \rangle$ because both nodes v_1 and v_2 have degree 4. Similarly, we can get that $DP(e_2) = \langle 2,3 \rangle$, $DP(e_3) = \langle 3,4 \rangle$, and $DP(e_4) = \langle 1,4 \rangle$. e_4 is a *degree-1* edge because $e_4.first$ is 1.

Definition 3.5 (Single Segment). A single segment is a path consisting of consecutive $\langle 2,2 \rangle$ degree-pair edges except the first and the last edges for which one end-node's degree is greater than 2.

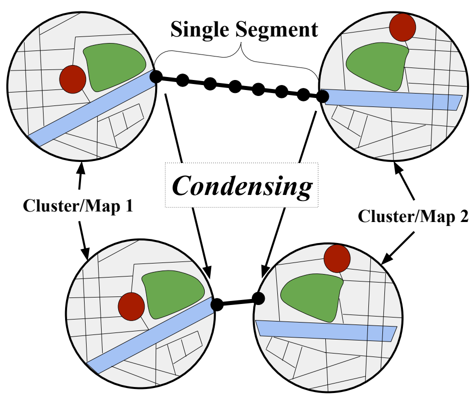


Figure 3: Single segment example

Example 3.6. Figure 3 shows an example of a single segment that connects two sub-graphs/maps with consecutive edges whose degree pairs are $\langle 2, 2 \rangle$.

Single segments are utilized to condense graphs (Section 4.3.1).

4 THE BACKBONE INDEX

The core idea for building the *backbone* index structure is summarizing the dense local units (clusters) of the original graph.

4.1 Hierarchical summarization

Before we present the index structure, we first introduce several *major factors where the design idea emerges from.*

First, the effectiveness of an index for graphs is highly related to the efficiency in the pre-calculation. For single-cost networks, pre-calculating shortest paths and using them to answer shortest path queries is a commonly used strategy. On MCRNs, multiple skyline paths exist between two nodes. Compared with pre-calculating shortest paths from single-cost networks, it is much more expensive to pre-calculate skyline paths because the number of skyline paths for a given query is highly impacted by node degrees and the distance between two nodes [19]. To leverage this, we identify local units to be dense graph components with nodes having more neighbors (or neighbors of neighbors). The abstraction occurs on each dense local unit by removing less critical nodes and edges. The abstraction leads to a smaller index size and a shorter construction time according to [19]. After the abstraction, we expect that the degree distribution of the graph nodes does not change much, which then can help us find useful results without missing too much information.

Second, too much information may be missing when directly summarizing the original graph to a very abstracted graph. Aggressive abstraction strategy may not be able to effectively support queries whose two query nodes are relatively close to each other. Considering this, we design our index structure to consist of a hierarchy of multiple abstracted graphs $G_0, G_1, \cdots, G_{L-1}, G_L$ with different granularity, where G_0 is the original graph, G_L is the most abstracted graph, and G_{i+1} ($0 \leq i < L$) directly summarizes G_i.

Third, to compensate the information loss caused by the removal of nodes and edges in dense clusters when summarizing a graph G_i, a facilitating structure I_i is introduced to keep the skyline paths from graph G_i to G_{i+1}. In particular, it stores the skyline paths from each node in a dense cluster to all the nodes that are still in G_{i+1}.

Based upon the design of the backbone index considering the above three factors, our query method returns informative approximate solutions instead of exact solutions by searching the summarized graphs from the finest granularity to the coarsest granularity. When we cannot find a path to connect two nodes in a lower-level graph G_i, the search has to be conducted on its summarized graph G_{i+1} which generates approximate skyline paths since G_{i+1} does not keep all the detailed information from its lower-level graph G_i.

4.2 Dense local units/clusters at each level

We introduce an important concept, *dense clusters*, in our backbone index. Intuitively, dense clusters represent local units or subgraphs of a graph. The nodes in the dense clusters generally have more neighbors (i.e., denser) than other subgraphs. We use dense clusters and local units exchange-ably in this paper.

4.2.1 Dense clusters and node clustering coefficient. DBSCAN [16] is one classical algorithm to find dense clusters. Density based clustering on road networks [41, 48] adopts the shortest path distance as the distance measurement. This is not suitable for MCRNs. Without extra information such as user pattern data [34], POIs [41], and trajectory location data [9], we need to formally define the measurements that can be used to calculate node density to conduct density based clustering on MCRNs. The well-known local clustering coefficient [42] is designed for general graphs where a node degree is usually more than hundreds. For MCRNs, where a node degree is generally no more than 5, the local clustering coefficient cannot be used to distinguish dense nodes from others. The cluster-coefficient concept should not only reflect the degree of a node, but also consider its neighbors. In Figure 2, node v_1 and node v_9 have the same number of neighbors, but intuitively, v_1 is more likely the center of its neighbors than v_9. Considering nodes v_{10} and v_9, based on their different degrees ($deg(v_{10}) = 3$ and $deg(v_9) = 4$), it seems v_9 is denser. However, v_{10} connects tighter with its neighbors in a local community than v_9 when examining the structure of the graph. Removing v_{10} and the edges connecting to it greatly reduces topological information of the graph. Overall, it is difficult to differentiate the density of a node by considering only node degrees.

We define a node's cluster coefficient to capture the density information of graph nodes. Let $\mathcal{N}_{1st}(v)$ be the set of neighbors of the node v and $\mathcal{N}_{2nd}(v)$ be the set of nodes that are two hops away from v (which are also denoted as *two-hop neighbors* of v) except the nodes in $\mathcal{N}_{1st}(v)$. We consider the node clustering coefficient of a node v is proportional to the number of connections between $\mathcal{N}_{1st}(v)$ and $\mathcal{N}_{2nd}(v)$. Following this idea, we introduce the concept of cluster coefficient on road networks.

Definition 4.1 (A node's cluster coefficient). The cluster coefficient of a node v is defined as

$$\text{cluster\_coefficient}(v) = \frac{|\mathcal{N}^v_{com}|}{|\mathcal{N}_{1st}(v)| * (|\mathcal{N}_{1st}(v)| - 1)} \quad (2)$$

where N^v_{com} is the set of node pairs (u, w) where $u \in \mathcal{N}_{1st}(v)$ and $w \in \mathcal{N}_{1st}(v)$ connect to a same node $v_{com} \in \mathcal{N}_{2nd}(v)$.

Example 4.2 (Node's cluster coefficients). In Figure 2, the cluster coefficient of node v_1 equals to $\frac{3}{4*3} = \frac{1}{4}$ since v_1 has *4* neighbors (v_2, v_4, v_6, and v_8) and those neighbors share 3 common nodes (v_3, v_5 and v_7) in $\mathcal{N}_{2nd}(v_1)$. For node v_9, the cluster coefficient is $\frac{1}{4*3} = \frac{1}{12}$ because the nodes in $N_{1st}(v_9)$ share one common node. For node v_{10}, $cluster\_coefficient(v_{10})$ is $\frac{2}{3*2} = \frac{1}{3}$.

If more second-order neighbors of v are connected through v's first-order neighbors (e.g., the center of a district), v has a higher probability to be in a dense area. Our approach thus clusters the nodes with bigger cluster coefficient first.

4.2.2 Condensing threshold. Our graph summarization is to keep the topology (thus the reachability) of the graph while condensing a graph. We discuss the rationale behind our design.

Motivation of defining condensing threshold. There are sparse components in real-world networks, such as secluded roads that connect business areas in a city. These sparse components are treated as noise clusters. Such noise clusters should not be completely condensed in the summarization stage. Otherwise, the nodes in these clusters cannot be reached from other graph nodes.

A node v can be categorized as a noise node or non-noise node using its node degree (i.e., the number of its first-order neighbors $|N_{1st}(v)|$) or its cluster coefficient (*cluster_coefficient*(v)). We observe that using either measurement is not sufficient to decide whether a node should be condensed or not. This is because the node degree (i.e., $|N_{1st}|$) and the cluster coefficient (decided by $|N_{1st}|$ or $|N_{2nd}|$) of different nodes on road networks have very similar values. I.e., the value ranges of node degrees and cluster coefficients are small. For instance, most nodes have degrees 2 and 3, and most nodes' neighbors share no or few common N_{2nd} neighbors. This makes the cluster coefficient values very small. E.g., in Figure 2, $cluster\_coefficient(v_9) = \frac{1}{12}$ and $cluster\_coefficient(v_{10}) = \frac{1}{3}$.

We need to investigate other measurements to decide whether a node can be condensed. That measurement should have a larger range and should capture the neighbor information so that a smaller value indicates a less important node.

We observe that $|N_{1st}(v) + N_{2nd}(v)|$ has a much bigger value range. Figure 2, $|N_{1st}(v_{10})+N_{2nd}(v_{10})| = 7$ is less than $|N_{1st}(v_9)+N_{2nd}(v_9)| = 10$. The node v_{10} is a less important node because it is connected with less other nodes. Thus, the cluster that v_{10} belongs to can be condensed later than the cluster that v_9 belongs to since v_9's cluster is denser than v_{10}'s cluster. Based on $|N_{1st}(v) + N_{2nd}(v)|$, we introduce another parameter, *condensing threshold percentage* p_{ind}, to help identify nodes that can be condensed.

Given a graph G, we can find the two-hop neighbors of all the nodes and calculate the cardinality of such neighbor sets. For each distinct two-hop neighbor cardinality k, we can find the number of nodes having this cardinality (denoted as $freq(k)$). I.e., $freq(k) = |\{v\}|$ s.t. $|N_{1st}(v) + N_{2nd}(v)| = k$. Let $\vec{L}(G)$ be the list of sorted frequency values calculated from a graph G, and $\vec{L}[j]$ be the frequency value at the j-th position in $\vec{L}(G)$, where j starts with 0. We define the condensing threshold as follows.

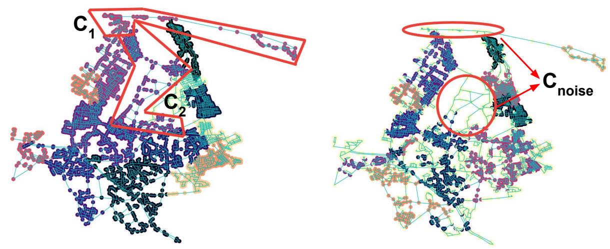
Definition 4.3 (Condensing threshold). Given G, the sorted frequency list $\vec{L}(G)$, a percentage $p_{ind} \in (0, 1)$, the condensing threshold $noise\_val$ is the cardinality value with frequency $\vec{L}[pos]$ s.t.

$$\sum_{i=0}^{pos-1} \vec{L}[i] \le p_{ind} * |G.V| < \sum_{i=0}^{pos} \vec{L}[i]$$

Example 4.4 (Condensing threshold). Given a graph G with 10 nodes, let the cardinality of the two-hop neighbor sets of the nodes be $\{8, 3, 6, 3, 6, 4, 4, 8, 2, 8\}$. The distinct cardinality values are 2, 3, 4, 6, and 8. Then, $\vec{L}(G) = (1, 2, 2, 2, 3)$ because $freq(2)$=1, $freq(3)$=2, $freq(4)$=2, $freq(6)$=2, and $freq(8)$=3. Let p_{ind} = 0.3, then $p_{ind} * |G.V| = 3$. $\vec{L}[0] + \vec{L}[1] = 3 \le 3$ and $3 < \vec{L}[0] + \vec{L}[1] + \vec{L}[2] = 5$. The $noise\_val$ of G is the cardinality value with frequency $\vec{L}[1]$. Since $\vec{L}[1] = 2 = freq(3)$, $noise\_val$ of G is *3*.

A node v is treated as a noise node if $|N_{1st}(v) + N_{2nd}(v)| < noise\_val$. The clustering procedure sets low-density nodes as noises when the condensing threshold is used. For example, two clusters, C_1 and C_2, in Figure 4(a) contain low-density nodes. These two clusters are condensed in the index construction process. However, using the condensing threshold, these low-density nodes are identified as noise nodes (Figure 4(b)). The noise nodes are not condensed when creating the index to preserve the topology structure that connects the low-density nodes.

4.2.3 Condensing dense clusters. Nodes on a map are always connected. We desire that the connectivity of a graph is preserved after condensing. We propose to use a spanning tree to condense a dense cluster because all the nodes in a spanning tree are connected. Minimum spanning trees (MSTs) are generated for optimization purposes on single-cost graphs. It is not possible to find MSTs from MCRNs because of the multiple edge weights. When using spanning trees to summarize a dense cluster, we build a spanning tree from the perspective of preserving the graph's topology as much as possible. In particular, we keep higher degree-pair edges because they can keep more information in the original graph, which is consistent with [40].



(a) clusters found without using condensing threshold **(b) clusters found using condensing threshold**

Figure 4: Example of dense clusters on C9_NY_5K

4.2.4 Details to process dense clusters of G_i. A graph G_i can be abstracted to a more summarized graph G_{i+1} by removing its nodes and edges. The removed node and edge information needs to be saved as labels (Definition 4.7) to support future query processing. This section discusses the process of condensing a graph G_i by utilizing its dense clusters. The detailed steps are described in Algorithm 1.

The condensing process contains two steps: (i) finding dense clusters of nodes (Lines 7-35) and (ii) abstracting each dense cluster (Lines 36-39). The cluster finding process grows the node with the highest cluster-coefficient value (the seed node) to the first cluster (details see below), then grows the node (as seed node), which has the highest cluster-coefficient value among all the nodes not belonging to any clusters, to the second cluster. This process of growing a seed node to a dense cluster stops until all the nodes are marked either as belonging to one cluster or as a noise node. After all the clusters are formed, small clusters (constrained by a parameter m_{min} defined in Definition 4.8) are merged to avoid cluster fragmentation (Line 35).

The details of growing a seed node v to a dense cluster $C_{i,j}$ are as follows. First, we calculate the threshold $noise\_val$ using the parameter p_{ind} (Line 2) and create a cluster list C that stores dense clusters of G_i (Lines 3-5). We designate a special set (C_{noise}) to keep all the noise nodes and add this noise-node set to C (Lines 4-5).

Then, a priority queue q is created to manage the growing process (Lines 21-33). Initially, q has a seed node v. While q is not empty, the node v_{pop} with the highest cluster-coefficient value in q is popped out. If v_{pop} is not a noise node or has not been visited yet, v_{pop} is put into the cluster $C_{i,j}$ (Line 30). Then, all the neighbors v' of v_{pop} are checked to see whether they need to be added to q to grow the cluster $C_{i,j}$ (Lines 31-33). When the cluster $C_{i,j}$ already contains m_{max} nodes or when v' is a noise node, we do not need to add v' to q. Once q is empty, the dense cluster $C_{i,j}$ is added to the cluster list C (Line 34).

The second step of condensing G_i is to condense each cluster. We form a spanning tree of G_i using a similar procedure as the Kruskal's algorithm with a different strategy on choosing edges. Our method first chooses the edges (not a random edge) with higher degree-pair values. Then degree-1 edges on the tree are recursively removed to guarantee the road network to be a 2-core graph after the removal. The removed nodes ΔV_i and edges ΔE_i are kept to create the index structure later (Details see Section 4.3).

Algorithm 1: Creation of dense clusters

Input : Graph G_i at the i-th level, maximum cluster size m_{max}, minimum cluster size m_{min}, p_{ind} for the condensing threshold, removed nodes ΔV_i, removed edges ΔE_i

Output : Updated ΔV_i, updated ΔE_i, and a list of clusters C

begin
  noise_val = findNosieIndicator(p_{ind});
  Set the set of clusters $C = \emptyset$;
  Create a noise-node cluster $C_{noise} = \emptyset$;
  C.put(C_{noise});
  /* Nodes in $G_i.V$ are sorted in the descending order of their *cluster_coefficient* values */
  foreach $v \in G_i.V$ **do**
    /* If v is visited, skip it */
    if *v.isVisited* **then**
      continue;
    /* If the number of v's two-hop neighbors in $\mathcal{N}_{1st}(v) \cup \mathcal{N}_{2nd}(v)$ is less than the condensing threshold, v is a noise node, skip it */
    if $|\mathcal{N}_{1st}(v) + \mathcal{N}_{2nd}(v)| <$ *noise_val* **then**
      C_{noise}.add(v);
      v.isVisited = true ;
      continue;
    /* Nodes in the queue are sorted by their cluster_coefficient values */
    j=size(C)+1 /* The j-th cluster for level i */;
    $C_{i,j}$= new cluster();
    q = new priority queue();
    q.add(v);
    while *!q.empty()* **do**
      v_{pop} = q.pop() /* v_{pop} has the highest cluster coefficient */;
      if v_{pop}*.isVisited* **then**
        continue;
      else if $v_{pop} \in C_{noise}$ **then**
        C_{noise}.remove(v_{pop});
        $C_{i,j}$.add(v_{pop});
      else
        v_{pop}.isVisited = true;
        $C_{i,j}$.add(v_{pop});

      foreach $v' \in v_{pop}$*.neighbors* **do**
        if $|C_{i,j}.V| \leq m_{max}$ & $|\mathcal{N}_{1st}(v') + \mathcal{N}_{2nd}(v')| \geq$ *noise_val* **then**
          q.add(v');
    C.add($C_{i,j}$);
  C.mergeSmallCluster(m_{min});
  foreach $C_{i,j} \in C$ **do**
    SpanningTree t = $C_{i,j}$.findSpanningTree();
    $\Delta V_i = \Delta V_i \cup$ t.removeNode();
    $\Delta E_i = \Delta E_i \cup$ t.removeEdges();
  return C, ΔV_i, ΔE_i

4.3 Backbone index

We introduce more terminologies and concepts. A given graph G_i may have multiple dense clusters, e.g., $C_{i,1}, C_{i,2}, \cdots, C_{i,c}$. Let $C_{i,j}.V$ denote the nodes in the dense cluster $C_{i,j}$ and use $C_{i,j}.\tilde{V}$ to denote the remaining nodes after removal.

Definition 4.5 (Highway Entrance Set). Given G_i, its dense clusters $\{C_{i,1}, C_{i,2}, \cdots, C_{i,c}\}$, and its abstracted graph G_{i+1}, the highway entrances of any $v \in C_{i,j}.V$ from G_i to G_{i+1} are $C_{i,j}.\tilde{V}$ and are denoted as H_v^{i+1}. Correspondingly, the overall highway entrances to G_{i+1} from G_i, denoted as H_{i+1}, form a set of nodes $\cup_{j=1}^{c} C_{i,j}.\tilde{V}$.

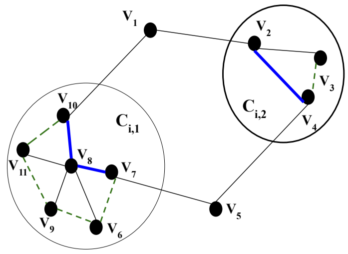


Figure 5: Example of highway entrances

Example 4.6 (condense process and highway entrances). In Figure 5, the given graph has two dense clusters $C_{i,1}$ and $C_{i,2}$, and two noise nodes v_1 and v_5. The edges are shown in lines (solid and dash lines). Initially, we find the spanning tree with higher degree-pair edges in each cluster (solid lines). Then the degree-1 edges on the trees are removed. Finally, thicker solid blue lines are the summary of dense clusters and are kept in G_{i+1}. This gives us $C_{i,1}.\tilde{V} = \{v_7, v_8, v_{10}\}$ and $C_{i,2}.\tilde{V} = \{v_2, v_4\}$. G_{i+1} consists of the noise nodes (v_1, v_5) and nodes in $C_{i,1}.\tilde{V}$ and $C_{i,2}.\tilde{V}$. The nodes in $C_{i,1}.\tilde{V}$ and $C_{i,2}.\tilde{V}$ are the highway entrances of the nodes in $C_{i,1}$ and in $C_{i,2}$ to G_{i+1} respectively. $H_{i+1} = C_{i,1}.\tilde{V} \cup C_{i,2}.\tilde{V} = \{v_7, v_8, v_{10}, v_2, v_4\}$ is the highway entrance set from G_i to G_{i+1}.

We use a facilitating structure I_i to store the skyline paths from each node v in $C_{i,j}$ to its highway entrance set H_v^{i+1}. An element of I_i, denoted as $label(v)$, is defined below.

Definition 4.7 (label(v)). Given a graph G_i, its dense clusters $\{C_{i,1}, C_{i,2}, \cdots, C_{i,c}\}$, and its abstracted graph G_{i+1}, the label of a node $v \in C_{i,j}.V$ is defined to be a triple $(v, H_v^{i+1}, \mathbb{P}_v^{H_v^{i+1}})$. Here, H_v^{i+1} is the set of highway entrances from v to G_{i+1} and $\mathbb{P}_v^{H_v^{i+1}} = \cup_{h \in H_v^{i+1}} \mathbb{P}_v^h$, where $\mathbb{P}_v^h$ is the set of skyline paths from v to a highway entrance $h \in H_v^{i+1}$.

A structure I_i keeps labels for all the nodes in each cluster $C_{i,j}.V$ no matter whether the node is removed from G_{i+i} or preserved in G_{i+1}. I.e., $I_i = \cup_{v \in C_{i,j}.V} label(v)$. For example, in Figure 5, the label of the highway entrance v_7, $label(v_7)$, needs to be created if the path $(v_7, v_6, v_9, v_{11}, v_{10})$ is a skyline path from node v_7 to v_{10}, which uses the removed edges $(v_7, v_6), (v_6, v_9), (v_9, v_{11})$, and (v_{11}, v_{10}).

Definition 4.8 (Backbone Index). Given a graph G, two integer thresholds m_{max} and m_{min}, and a percentage p, the backbone index of G consists of (i) a list of graph summarization structures $(0, I_0), (1, I_1) \cdots, (L-1, I_{L-1})$, and (ii) the most abstracted graph G_L. Here, m_{max} and m_{min} are the maximum and minimum number of nodes of a dense cluster, and p is the minimum percentage of edges that must be condensed in each level.

For example, if we set the parameters to be $m_{min} = 30$, $m_{max} = 200$, and $p = 0.01$, we expect (i) at most 200 nodes exist in each cluster, (ii) clusters containing less than 30 nodes are merged, and (iii) at least 1% of the edges need to be removed in the process of index construction at each level to avoid generating too many summarization structures. The parameter p decides the number of edges that must be removed, thus controls the index level L.

Figure 6 shows a backbone index with three layers (i.e., $L = 3$). The index provides a multi-level view of the original graph with different abstraction power. For instance, G_1 is a summarized view

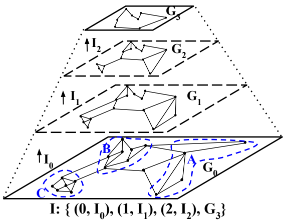


Figure 6: Index example

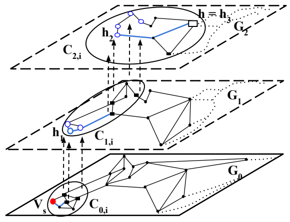


Figure 7: Paths in index

of the original graph G_0 by condensing three dense clusters (local units) A, B, and C. I_0 keeps the labels of the nodes in G_0. The highest level graph $G_L(G_3)$ is the most abstracted view of G_0.

4.3.1 Index construction.

Algorithm 2 outlines the framework of the index construction process. Initially, the backbone index takes the original graph G_0 as the root. Then, the index is construed recursively. This summarization works in two steps: (1) regular summarization and (2) aggressive summarization if needed.

Regular summarization. We first remove the degree-1 edges from graph G_i. This action leads to the removal of paths consisting of consecutive degree-1 edges. All the degree-1 edges are removed until every remaining node in G_i has a degree of 2 or higher.

Then, we identify dense clusters (i.e., $C_{i,1}, \ldots, C_{i,j}, \ldots, C_{i,c}$) of G_i (Algorithm 1). A more abstracted graph is formed after the condensation. The removed nodes ΔV_i and edges ΔE_i are returned to create $label(v)$ of each node v in $C_{i,j}$. In $label(v)$, the skyline paths from v to its highway entrances H_v^{i+1} are generated using only the deleted edges E_r^i of $C_{i,j}$ where $E_r^i \subseteq \Delta E_i$ by applying a single source skyline path query algorithm (e.g., *BBS* mentioned in Section 6s). This strategy not only preserves the deleted edge information in the skyline paths, but also speeds up the query process.

The index height L increases rapidly if G_i is only condensed in one iteration to form G_{i+1}. To prevent the rapid increase of the index height, we keep abstracting G_i until both of the following two conditions are met: (i) some nodes and edges are left after the current iteration (i.e., $|G_{i+1}.V| \neq 0$), and (ii) sufficient number of edges are removed from G_i (i.e., $|\Delta E_i| \geq p * |G_0.E|$). When these conditions are met, the abstracted graph is considered as G_{i+1} and used as the input of the summarization to the next level.

Aggressive summarization. While trying to maintain the graph's topology, it is possible that the regular summarization function cannot remove sufficient nodes and edges (Line 9), with the construction terminating with a large G_L, which leads to high computational cost during the query process. To address this issue, we deploy a more aggressive strategy that condenses a special type of paths, single segments (Definition 3.5), in G_{i+1}. In particular, it builds shortcuts to replace single segments and creates labels for the deleted nodes in the single segments.

The aggressive summarization strategy is simple, but when to apply it is not trivial. The graph's topology is destroyed if the strategy is used during the regular summarization step. If it is not applied, G_L can still be very large, thus cannot help support efficient query processing. If this strategy is called too frequently, numerous short single segments are merged, which increases the node degrees of the graph. This goes against our design principle of reducing the graph's node degrees and incurs longer index-building process.

Example 4.9 (Condensing single segments.). Given a single segment s=$(u, v_0, v_1, \cdots, v_{j-1}, v_j, w)$, the aggressive strategy condenses it to an edge $e = (u, w)$ by removing all the nodes v_0, v_1, $\cdots$, and v_j. The cost of e is the summation of the edge weights of s. The labels are created for each v to its highway entrances $\{u, w\}$. Figure 3 shows an example of condensing a single segment.

Algorithm 2: Framework of index construction

Input : Graph G, percentage p, maximum and minimum cluster sizes m_{max} and m_{min}

Output: Backbone index I_{list}: (0, I_0), $\cdots$, (L-1, I_{L-1}) and the highest graph G_L

```
1  begin
2      i = 0;
3      Create index I_list = ∅;
4      do
5          /* Step 1: Regular Summarization of G_i */
6          (ΔE_i, I_i, G_{i+1}) = GraphSummarization(G_i, p, m_max, m_min);
7          I_list.put(i, I_i);
8          /* Step 2: Aggressive Summarization of G_{i+1} */
9          if |G_{i+1}.V| ≠ 0 & ΔE_i ≤ p * |G_0.E| then
10             ΔE_new, I_new = AggressiveGraphSummarization(G_{i+1});
11             if |ΔE_new| ≠ 0 then
12                 Update I_i using I_new;
13                 ΔE_i = ΔE_i ∪ ΔE_new;
14         L=i, i=i+1 ;
15     while |G_{i+1}.V| ≠ 0 and ΔE_i ≥ p * |G_0.E|;
16     landmark(G_L);
17     return I_list, G_L
```

The index element I_{new}, which is generated in the aggressive graph summarization process, is used to update the existing index item I_i. In particular, every path $p \in \mathbb{P}_v^{v'}$ (where $label(v) \in I_i$) is concatenated with every path $p' \in \mathbb{P}_{v'}^{h}$ (where $label(v') \in I_{new}$) where v' is a highway entrance of v (i.e., $v' \in H_v^{i+1}$ and H_v^{i+1} is in $label(v)$). Finally, the landmark index [28] is built over the highest level graph G_L.

Index maintenance. The backbone index can be dynamically maintained when there are changes in the underlying road networks (e.g., addition or removal of nodes and edges). The basic idea is to recalculate the skyline path information for the cluster nodes that are involved in graph updates. We omit the details and the experimental results due to space limitation, which can be found from [19].

Extended to directed graphs. When road networks are modeled as directed graphs, the index just needs to include the extra information from highway entrances to each node in dense clusters. Getting such information is straightforward because skyline path information between all pairs of nodes in each dense cluster has been calculated in the regular summarization process.

5 QUERY PROCESSING ALGORITHM

This section explains the query processing algorithm over a graph G to get approximate solutions for a SPQ. A SPQ is denoted by two nodes v_s and v_t. The query is processed on the backbone index I=$\{(0, I_0), (1, I_1), \cdots, (L-1, I_{L-1}), G_L\}$.

Given a node $v_s \in G_0.V$, let us use $\mathbb{P}_{v_s}^{h_i}$ to denote the set of skyline paths from v_s to a highway entrance $h_i \in H_v^i$ in G_i. A path in $\mathbb{P}_{v_s}^{h_i}$ concatenates multiple skyline paths $p(v_s \leftrightsquigarrow h_1), p(h_1 \leftrightsquigarrow$

$h_2), \cdots, p(h_{i-1} \leftrightsquigarrow h_i)$ where h_i is a highway entrance at G_i. Figure 7 shows an example of one path p in $\mathbb{P}^{h}_{v_s}$ on subgraphs of G_0, G_1, and G_2 where blue hollow circles in G_1 and G_2 are the highway entrances. p consists of three sub-paths $p(v_s \leftrightsquigarrow h_1)$ (in G_0), $p(h_1 \leftrightsquigarrow h_2)$ (in G_1), and $p(h_2 \leftrightsquigarrow h_3)$ (in G_3).

A node v can directly or indirectly reach a highway entrance node h at different index levels through a path $p(v \leftrightsquigarrow h)$. We call the set of highway entrance nodes at different index levels that v can reach as v's *successor nodes* and denote them as $succ(v)$. For example, all the nodes represented as blue hollow circles in Figure 7 are successor nodes of the node v_s.

Given a query with two nodes v_s and v_t, *the backbone paths* are formed as two types: (1) when two sets $\mathbb{P}^{h}_{v_s}$ and $\mathbb{P}^{h}_{v_t}$ reach a common highway node $h \in H_k$ where $k < L$ is an intermediate index level (the first type), or (2) when both nodes v_s and v_t reach the most abstracted graph G_L through the highway nodes h_s and h_t in H_L, which means that $\mathbb{P}^{h_s}_{v_s}$ and $\mathbb{P}^{h_t}_{v_t}$ are connected using paths $p(h_s \leftrightsquigarrow h_t)$ in G_L, where h_s and h_t are successor nodes of v_s and v_t respectively (the second type).

Algorithm 3 describes the process to find the first (Lines 6-28) and the second type (Lines 29-32) of backbone paths between v_s and v_t. Given a node v, the function *ReadLabel*(v) reads the index label of v and extracts the highway entrance nodes H^{i}_{v} that v can reach G_i from G_{i-1} directly. When v does not exist in G_{i-1}, then H^{i}_{v} is empty. The function *addToSkyline* adds paths to the result set $\mathcal{R}$ while guaranteeing all the paths in $\mathcal{R}$ do not dominate each other.

To find the first type of skyline paths, the algorithm grows skyline paths from v_s and v_t to their successor nodes. If the paths from v_s and v_t meet at a common successor node, such paths are skyline candidates. To manage the skyline path growing process, two map structures, $\mathbb{S}$ and $\mathbb{D}$, are created (Lines 3 and 4) to store the skyline paths from v_s and v_t to their *successor nodes* respectively. In $\mathbb{S}$, a key is the ending node of a path from v_s and the corresponding value for the key is a list of skyline paths from v_s to the ending node. The initial key-value pair in $\mathbb{S}$ is $(v_s, \{p_{v_s} = \{v_s\}\})$ (Line 3). Similarly, $\mathbb{D}$ is constructed to manage skyline paths from v_t.

Lines 6-15 grow the skyline paths from v_s using the index structure at each level i by utilizing the ending node sh of a path in $\mathbb{S}$. The algorithm finds all the paths $\mathbb{P}^{h}_{sh}$ from sh to each highway entrance node h at level i (i.e., $h \in H^{i}_{sh}$), which can be extracted from $label(sh)$ (Line 10) and concatenates them with the skyline paths in $\mathbb{P}^{sh}_{v_s}$ (which can be found from $\mathbb{S}$ with key sh (Line 11). If the highway entrance node h is another query node v_t, the formed skyline paths are used to update the result set $\mathcal{R}$ (Line 13). Otherwise, the formed skyline paths are added to the intermediate skyline path set $\mathbb{S}$. This path growing process may reach level G_L.

A similar procedure is used to calculate backbone paths from v_t to its successor nodes (Lines 16-28). The difference is that one more condition is added to form new candidate paths, when one successor $h \in succ(v_t)$ is also in $\mathbb{S}$ (Lines 24-26).

The second type of skyline paths are found when the paths in $\mathbb{S}$ and $\mathbb{D}$ reach G_L but cannot be concatenated. A many-to-many method, *m_BBS*, is conducted (Line 32) to find the skyline paths $p(v_s \leftrightsquigarrow v_t) = p(v_s \leftrightsquigarrow h_s)||p(h_s \leftrightsquigarrow h_t)||p(h_t \leftrightsquigarrow v_t)$. $p(h_s \leftrightsquigarrow h_t)$ represents any skyline path from h_s to h_t where h_s and h_t are successor nodes of v_s and v_t in G_L respectively. The *m_BBS* method is a modified version of *BBS* by accepting multiple nodes as input and estimating the lower bounds of a path to all the possible destination (not one destination in the original algorithm). The proposed *m_BBS* just needs to be executed once, instead of multiple times, for each pair of nodes in $\mathbb{S}.keys$ and $\mathbb{D}.keys$.

Algorithm 3: Query processing algorithm

Input : Query nodes v_s and v_t, the most abstracted graph G_L, backbone index I

Output : The set of backbone skyline paths $\mathcal{R}$

1 **begin**
2 Initialize the result set $\mathcal{R} = \emptyset$;
3 Create a new map $\mathbb{S}$ initialized with (v_s, p_{v_s});
4 Create a new map $\mathbb{D}$ initialized with (v_t, p_{v_t});
5 /* Find the first type of skyline paths */
6 **foreach** $0 \leq i \leq L$ **do**
7 **foreach** $sh \in \mathbb{S}.keys$ **do**
8 $ReadLabel(sh)$ and extract the highway entrances H^{i}_{sh};
9 **foreach** $h \in H^{i}_{sh}$ **do**
10 Get the set of skyline paths $\mathbb{P}^{h}_{sh}$ from sh to h ($ReadLabel(sh)$);
11 $\mathbb{P}^{h}_{v_s}$ = combine all the paths in $\mathbb{P}^{sh}_{v_s}$ with all the paths in $\mathbb{P}^{h}_{sh}$;
12 **if** $h = v_t$ **then**
13 $\mathcal{R}$.addToSkyline($\mathbb{P}^{h}_{v_s}$);
14 **else**
15 $\mathbb{S}$.put(h, $\mathbb{P}^{h}_{v_s}$) ;
16 **foreach** $0 \leq i \leq L$ **do**
17 **foreach** $dh \in \mathbb{D}.keys$ **do**
18 $ReadLabel(dh)$ and extract the highway entrances H^{i}_{dh};
19 **foreach** $h \in H^{i}_{dh}$ **do**
20 Get the set of skyline paths $\mathbb{P}^{h}_{dh}$ from dh to h ($ReadLabel(dh)$);
21 $\mathbb{P}^{h}_{v_t}$ = combine all the paths in $\mathbb{P}^{dh}_{v_t}$ with all the paths in $\mathbb{P}^{h}_{dh}$;
22 **if** $h = v_s$ **then**
23 $\mathcal{R}$.addToSkyline($\mathbb{P}^{h}_{v_t}$);
24 **else if** $h \in \mathbb{S}$ **then**
25 $\mathbb{P}^{v_t}_{v_s}$ = new paths combining $\mathbb{P}^{h}_{v_t}$ with $\mathbb{S}$.get(h).$\mathbb{P}^{h}_{v_s}$;
26 $\mathcal{R}$.addToSkyline($\mathbb{P}^{v_t}_{v_s}$);
27 **else**
28 $\mathbb{D}$.put(h, $\mathbb{P}^{h}_{v_t}$) ;
29 /* *BBS* on G_L to find the second type of skyline paths */
30 $S_{possible} = G_L.V \cap \mathbb{S}.keys$;
31 $D_{possible} = G_L.V \cap \mathbb{D}.keys$;
32 $\mathcal{R}$.addToSkyline(m_BBS(G_L, v_s, v_t, $S_{possible}$, $D_{possible}$))
33 **return** $\mathcal{R}$ // Return the results

Support to other types of queries. The backbone index can be used to support one-to-all SPQs to return approximate skyline paths to all other nodes from a given query node. The details and experimental results can be found in [19].

Solution bound. Given a graph G, its backbone index, a query (v_s, v_t), the upper bound of an approximate solution path's weight is $O((F_{val})^L)$. Here, L is the height of the index, and F_{val} is the expected summation of the weights for all the edges in the minimum spanning tree over a complete graph with a very large number of nodes.

Complexity. The complexities of index construction time and index size are $O(|G.V|log(|G.V|))$ and $O(|G.V|m_{max}S_n d)$ respectively. Here, d is the number of dimensions of edge cost, and S_n is the average number of skyline paths between every node to its highway entrance in each dense cluster and is almost constant

Table 1: Statistics of road networks

	description	vertex #	edge #	raw data size
C9_NY	New York	254,346	365,050	16.2 MB
C9_BAY	San Francisco Bay Area	321,270	397,415	18.9 MB
C9_COL	Colorado	435,666	521,200	38.9 MB
C9_FLA	Florida	1,070,376	1,343,951	98.4 MB
C9_E	East USA	3,598,623	4,354,029	337.7 MB
C9_CTR	Center USA	14,081,816	16,933,413	1304.0 MB
L_CAL	California	21,048	21,693	1.3 MB
L_SF	San Francisco	174,956	221,802	12.2 MB
L_NA	USA	175,813	179,102	11.0 MB

when m_{max} is small. S_n is no more than 10 when m_{max} is 200 in our experiments.

The detailed complexity analysis for *the upper bound of an approximate solution, the index construction time, and the index space* is omitted here due to space limit and can be found at [19].

6 EXPERIMENTS

6.1 Experimental settings

Our experiments are conducted on a desktop with an Intel(R) 3.60 GHz CPU, 32 GB main memory, and 2 TB HDD, running Ubuntu 18.04. All the algorithms are implemented using Java 13 [3]. We use Neo4j [4], the most popular graph database [2], to store all the graphs. The page size and cache size of Neo4j are set to 2 KB and 2 GB respectively. The native JAVA APIs of Neo4j are used to access neighbor nodes. Our backbone index is not stored in Neo4j.

Default parameter setting. The condensing threshold p_{ind} (Definition 4.3) is set to 30%, the minimum and maximum cluster sizes m_{min} and m_{max} (Definition 4.8) are set to be 30 and 200 respectively, and the percentage p used to decide whether sufficient number of edges are removed (Definition 4.8) is 0.01. More discussions about the effect of these parameters are in Section 6.2.5.

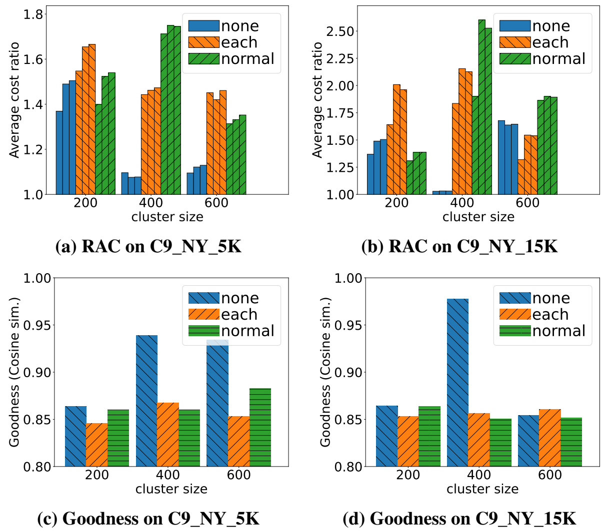
Parameter value selection. To set values of different parameters, users can take a strategy that is widely adopted in using machine learning libraries: starting with the default setting and fine-tuning the parameters. For any dataset, users can use the above default setting to get query results with similar accuracy that we report. If users accept query results with less accuracy guarantee, they can increase m_{max} and/or p. Otherwise, they need to decrease m_{max} and/or p. Users need to be aware that the index construction time for larger/smaller datasets is longer/shorter. Generally, m_{min} and p_{ind} do not need to be changed. Or, users can follow the analysis in Section 6.2.5 to fine tune them.

Data. Our experiments use *nine* real-world road networks [1, 5] (details see Table 1). The original networks contain the coordinates of nodes and one-dimensional edge weights (the spatial length of road segments). We generate two extra synthetic edge weights by sampling them from a uniform distribution in the range of [1,100] following the practice in [12, 29]. A detailed comparison of different ways to generate synthetic costs is in Section 6.3. When smaller subgraphs with a specific number of nodes are needed in the experiments, we generate such subgraphs by conducting *BFS* from a random node on the real-world networks.

Approximation quality measurements. To evaluate the quality of an approximate result set, we apply the following measurements.

(1) The ratio of average cost on each dimension (RAC). We introduce RAC_i to measure the similarity between the approximate results and the exact solutions on the ith dimension. It is defined as $\text{RAC}_i = \frac{(\sum_{p' \in \mathbb{P}'} w'_i | w'_i \in cost(p'))/|\mathbb{P}'|}{(\sum_{p \in \mathbb{P}} w_i | w_i \in cost(p))/|\mathbb{P}|}$ where $\mathbb{P}'$ and $\mathbb{P}$ are the set of approximate skyline paths and the exact SPQ solutions respectively. A RAC_i value that is closer to 1 is better.

(2) Goodness. We modify the *goodness* measurement [20] to make it suitable for SPQs, which are different from the queries in [20]. Given the exact solution set $\mathbb{P}$ and an approximate solution set $\mathbb{P}'$ for an SPQ, the goodness score of $\mathbb{P}'$ is defined as: $goodness(\mathbb{P}') = \frac{\sum_{p \in \mathbb{P}} \{\text{argmax}_{p' \in \mathbb{P}'} \, sim(p,p')\}}{|\mathbb{P}|}$ where $sim(p, p')$ is the similarity function between the cost of two paths. We use the *cosine similarity* (the higher the better) to calculate $sim(p, p')$.



(a) RAC on C9_NY_5K **(b) RAC on C9_NY_15K**

(c) Goodness on C9_NY_5K **(d) Goodness on C9_NY_15K**

Figure 8: Comparison of approximation quality

Exact method. We implement the SPQ method in [29] and speed up the query by initializing the result set with the shortest path on each dimension. We call this implementation the Baseline Best-first Search method (abbreviated as **BBS**). *BBS* returns exact SPQ solutions that are used to verify the quality of the approximate solutions.

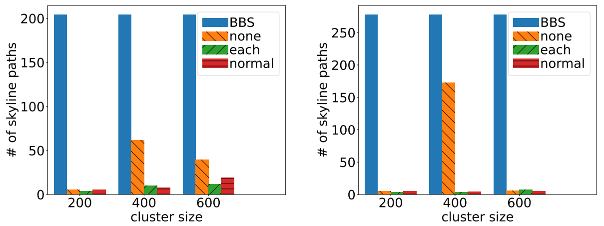
Comparison methods. Since no existing index structure is particularly designed to support SPQs, to demonstrate the effectiveness of our proposed index construction strategy **backbone_normal** (Algorithm 2), we modify two representative shortest path indexes, **GTree** [50] and **CH** [37], to compare with our index structures. The index construction process of *GTree* and *CH* follows their original contracting process. The difference is that we use skyline paths (instead of shortest paths) as the new edges. We also implement two more variations (**backbone_none** and **backbone_each**) of our index construction methods by varying the implementation of triggering the aggressive graph summarization (Section 4.3.1). The *backbone_none* only conducts regular graph summarization. The *backbone_each* triggers the aggressive summarization at each level.

6.2 Experimental results

6.2.1 Effectiveness of the proposed index structure and query method. We compare the query results with the exact solutions returned by *BBS*. The *BBS* method does not work well on large graphs [19]. Thus, we use small subgraphs of C9_NY with 5K and 15K nodes. On both C9_NY_5K and C9_NY_15K, we randomly generate 300 queries (i.e., pairs of starting and ending nodes of the queries). For these random queries, we run both the *BBS* method and our methods to get exact and approximate solutions for comparisons.

We examine how good the approximate results are. Figures 8(a-b) show the RAC values. Three consecutive bars in the same color and shape represent results from one method. The ratio for each dimension is shown from left to right. Figures 8(c-d) plot the goodness values. We can see that *backbone_none* has the best (smallest) average approximation in most cases among the three variations. This is because the *backbone_none* variation keeps much more nodes and edges in G_L while building the index. One exception is that *backbone_none* is slightly worse than *backbone_each* on C9_NY_15K when m_{max}=600 . This is because the level L of the index generated by the *backbone_none* (L=6) is larger than the level of index generated by *backbone_each* (L=4). This is consistent with our analysis about the index structure: an index with a larger L (meaning a higher index) loses more information.

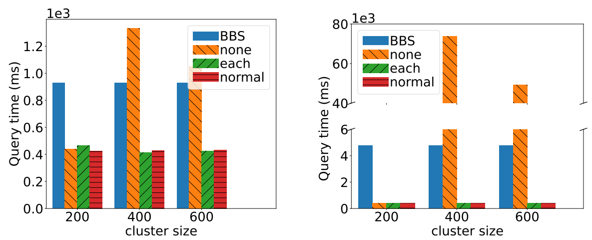
The *backbone_each* and *backbone_normal* variations perform similarly because they all trigger the aggressive strategy. They provide rough 1.5-approximation solutions (RAC) and get ~0.85 goodness scores. The approximation of *backbone_normal* is slightly better than that of *backbone_each* for three settings (m_{max}=200 for both graphs, and m_{max}=600 for C9_NY_5K) because the indexes generated using *backbone_normal* are larger than those generated using *backbone_each* in these three settings. On the other hand, *backbone_each* slightly outperforms *backbone_normal* for the remaining three settings (m_{max}=400 for both graphs, and m_{max}=600 for C9_NY_15K) because of a similar reason.



(a) C9_NY_5K **(b) C9_NY_15K**

Figure 9: Comparison of result set size (# of skyline paths)

Figures 9 shows that all three variations can hugely reduce the result-set sizes. When more nodes and edges are kept in G_L, more skyline paths are found on G_L, which leads to a larger result set. When cluster size increases, the *backbone_none* variation generates larger G_L compared with the other two variations, which slows down the *m_BBS* significantly. Figure 10 reports the averaged query time for the 300 queries. The *backbone_none* variation even needs more time than *BBS* in most situations because of the large G_L. The query time of *backbone_each* and *backbone_normal* is stably small because of a smaller G_L (Figure 10). In summary, our proposed index construction approach can achieve a good trade off in preserving the graph information and effectively supporting queries.



(a) C9_NY_5K **(b) C9_NY_15K**

Figure 10: Comparison of query time

6.2.2 Efficiency of index construction. We conduct experiments to measure the index size and building time by comparing our index structure with *GTree* and *CH*. We use subgraphs of C9_NY with 5K, 10K, and 15K nodes. For the *GTree* method, the fan out is set to be 4 and the number of vertices in a leaf node is set to 64. These parameter values are used to generate the best results in the original paper. The experimental results are reported in Table 2.

The results show that the index size of *GTree* is comparable to our proposed method. However, the construction time of *GTree* is much more than our method. The main reason is that the graph contracting process of *GTree* increases the graph size, which grows exponentially in the number of nodes and edges. Such graph-size increase slows down the performance of SPQs. For example, the root node in the *GTree* contains 74794 and 169623 edges for C9_NY_5K and C9_NY_15K respectively. The index on C9_NY_10K cannot be created in one day while processing a non-leaf node with 2,754,341 edges. Given these, we can observe that *GTree* index structure is not practical in supporting SPQs on large graphs.

Table 2: Comparison of index construction

		C9_NY_5K	C9_NY_10K	C9_NY_15K
Construction time (sec.)	Backbone	99	251	216
	GTree	23,896 (6 hours)	-	39,781 (11 hours)
	CH	12,000	42,184	26,340
Index size (MB)	Backbone	27	89	68
	GTree	27.5	-	41.6
Size of the most abstracted graph	CH node #	4,071	9,654	13,499
	CH edge #	22,627	30,894	83,302

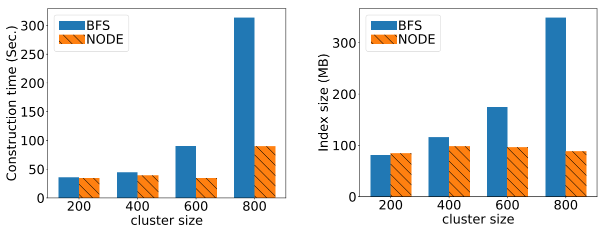
For the *CH* index, we report the graph size instead of the index size because the final graph of the *CH* is used to speed up online shortest path queries. The result shows that the number of nodes does not change much after summarization. However, the number of edges is at least 5 times more than that in the initial graph. The huge final graph causes the deterioration of query processing. The underlying reason is that multiple skyline paths (instead of one shortest path) exist between two nodes. Furthermore, the index building time also becomes impractical when the graph size increases.

6.2.3 Effectiveness of using dense clusters to condense G_i. We evaluate the effectiveness of our approach of using dense clusters to condense G_i (Section 4.2). For comparison purpose, we implement another approach to partition the nodes in G_i to different connected components by using *BFS*. Other partition methods [24, 27] used in [26, 30, 32] that merely consider the connectivity between partitions but not the density of the partitions get similar results as the *BFS* partitioning method. Our method is labeled as *NODE* and the alternative partition method is labeled as *BFS*. We measure the index size and the time to construct the backbone index from the partitions discovered using our dense-cluster based method and from the partitions found using *BFS* method.

Figure 11 shows the results on dataset C9_NY_15K. When the cluster size increases, building indexes using the partitions found from the *BFS* method requires longer time and uses more space (can be more than three times for m_{max}=800), compared with creating indexes using graph partitions discovered from our method.

This result demonstrates that *our design of using dense clusters to condense a graph* is more appropriate than using partitions which do not consider graph density.

6.2.4 Scalability test of query algorithms. We test the scalability of our approach by comparing it with *BBS* on subgraphs of C9_BAY with different number of nodes (from 10K to 100K). We generate ten random queries for different datasets. To control the randomness of queries, we constrain the distributions of the number of hops between the starting and ending query nodes to be similar for all the datasets. In particular, for each dataset, two



(a) Construction time **(b) Index size**

Figure 11: Effectiveness of cluster-based condensing

queries have less than 50 hops, three queries have between 50 to 100 number of hops, and five queries have larger than 100 hops. We also constrain that these queries can be finished in fifteen minutes using the *BBS* method so that comparisons can be done with reasonable time. We run these queries using our approach and the *BBS* method, and report the averaged running results in Table 3.

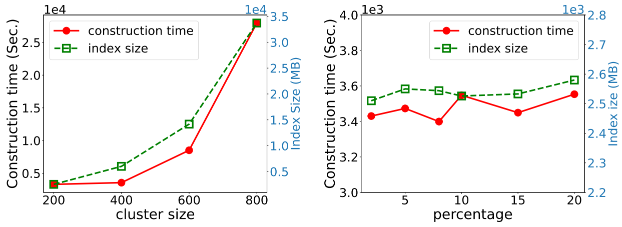
Table 3: Scalability of query algorithms (subgraphs of C9_BAY)

# of nodes	10K	40K	70K	100K
RAC	1.41, 1.67, 1.63	1.48, 1.79, 1.68	1.85, 1.90, 1.93	1.56, 1.80, 1.71
Goodness (Cosine similarity)	0.88	0.85	0.87	0.87
BBS method query time (ms)	34,154	63,557	101,470	30,789
Backbone index query time (ms)	461	410	437	470
Speed-up ratio	74	155	232	65
Construction time (ms)	126,450	429,488	815,771	930,892

The first observation is that our proposed algorithm achieves reasonable *RAC* and *goodness* score in these different graphs. Second, although the construction time grows as the graph size increases, the improvement of query time is significant. Our method speeds up the *BBS* method dramatically (more than 65 times in all subgraphs). We are aware that the construction time of our backbone index is not less than the average query time of *BBS*. This is because the index construction needs to pre-calculate skyline path information for all the node pairs in each cluster. We need to emphasize that the backbone index just needs to be built once and can support any ad-hoc SPQs efficiently. To speed up the index construction process, we need to improve the component of pre-calculating skyline paths. A reasonable idea is to pre-calculate less (but still good) skyline paths for the node pairs in clusters utilizing strategies in [20].

The query time of both the *BBS* method and our method does not show a steady trend with the increase of node numbers. This is because the performance of the *BBS* method is more affected by node degrees and the number of hops of queries according to our preliminary study [19]. On the 100K subgraph, the *BBS* method has abnormally low running time because of the lower average node degree of this graph compared with other graphs and the smaller average number of hops for queries on this subgraph. Our proposed method takes a relatively constant time on different subgraphs (vary from 410 ms to 470 ms). The queries over the 10K graph have larger query time because its index has more levels (i.e., a larger L).

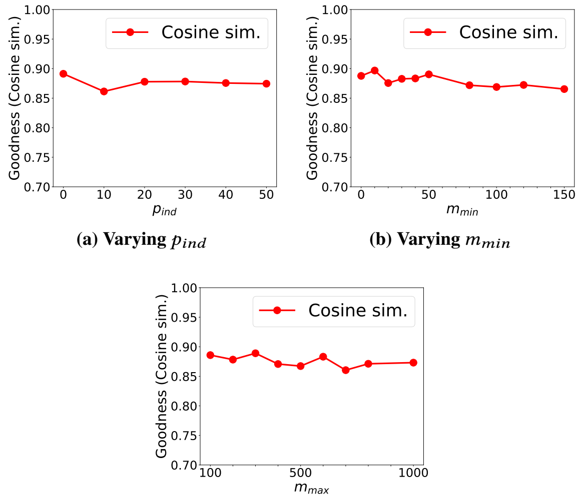
6.2.5 Effect of parameters. Figure 12 shows the impact of the parameters p and m_{max} on the performance of index construction. The index construction process is sensitive to the cluster



(a) Varying m_{max} **(b) Varying p**

Figure 12: Index building time and index size for C9_NY

size as shown in Figure 12(a). Both the time of finding skyline paths and the number of skyline paths in each cluster grow with the increase of m_{max}. The results indicate that it is practical to set m_{max} to be 200 and 400 to get reasonable building time and index size. When m_{max} reaches 800, the algorithm can take 6 hours to build the index and the index size is 3.5 times of G's size, which is not workable. On the contrary, the building time and the index size are almost constant when p changes (Figure 12(b) because p only affects the levels of the indexes L, which are almost the same for different p values.



(a) Varying p_{ind} **(b) Varying m_{min}**

(c) Varying m_{max}

Figure 13: Goodness comparison on C9_NY_15K

We further examine the effect of three parameters, condensing threshold p_{ind}, minimum cluster size m_{min}, and maximum cluster size m_{max} on the quality of approximation results using a small graph with 15K nodes (C9_NY_15K) because *BBS* is inefficient on large graphs. The reported numbers in Figure 13 are averaged from results of 100 random queries over C9_NY_15K. For the parameter p_{ind}, the overall trend is that its effect fluctuates before reaching a value (20 for this test) and slightly decreases after that. In this test, the best performance is achieved with zero. This is because the dataset is obtained using *BFS* and it has less low-density nodes. This is not the general conclusion for all the datasets. For m_{min}, a similar overall trend is observed: its effect fluctuates before reaching a value (m_{min}=50) and slightly decreases after that. This is because the approximation is worse when we do not sufficiently merge small clusters (smaller m_{min}) or merge big clusters (larger m_{min}). For the parameter m_{max}, the goodness score shows fluctuations with an overall trend of decreasing performance with the increase of m_{max}. Given these, smaller m_{max} should be used to achieve better query accuracy. However, very small m_{max} (the extreme case is m_{max}=1) should not be used because of much longer query time.

6.2.6 Performance on larger graphs. We apply our index construction approach to large real-world graphs. The results are shown in Table 4. The size of the highest graph row shows the

Table 4: Scalability of backbone index construction

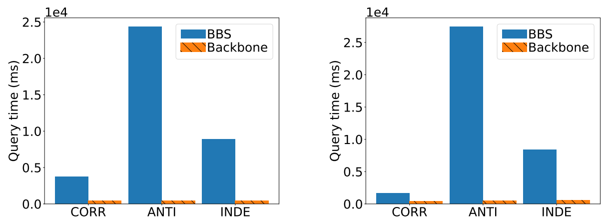
	C9_NY	C9_BAY	C9_COL	C9_FLA	C9_E	C9_CTR
Construction time (sec.)	3,305	3,056	4,331	12,082	61,471	532,456
Index size (MB)	2,526	1,954	2,535	6,531	21,484	81,196
Size of the (node #)	193	152	4	219	97	167
highest graph (edge #)	193	152	6	306	131	217
Query time (ms)	419	426	414	505	526	516

(a)

	L_CAL	L_SF	L_NA			
Construction time (sec.)	270	3,056	1,472			
Index size (MB)	86	1954	709			
Size of the (node #)	173	152	56			
highest graph (edge #)	248	152	87			
Query time (ms)	479	424	418			

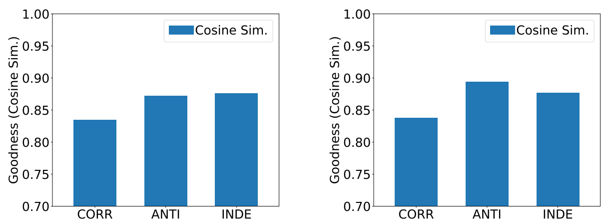
(b)

number of nodes (top number) and edges (bottom number) in the most abstracted graph G_L. Table 4(a) shows the results on the graphs [1] that have higher node degrees. Table 4(b) shows the results on graphs with lower average node degrees [5]. Our proposed algorithm scales well as the number of graph nodes increases from 0.01 million (C9_NY_10) to 14 million (C9_CTR). On the graph C9_CTR, the average search time is only 0.5 seconds. A huge jump on the index construction time occurs on C9_CTR. This is because the graph has higher node degrees, which make the pre-calculation of skyline paths in dense clusters more expensive than in other graphs.



(a) C9_NY_20K **(b) C9_BAY_20K**

Figure 14: Query time (different edge-cost distributions)



(a) C9_NY_20K **(b) C9_BAY_20K**

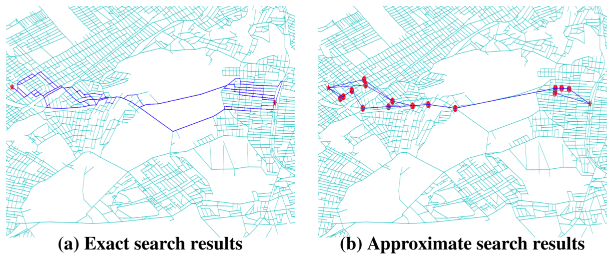
Figure 15: Goodness scores (different edge-cost distributions)

6.3 Effect of edge-cost distribution

We examine the effect of the distribution of edge cost on the query time and the goodness score. We generate subgraphs with *20K* nodes from the C9_NY and C9_BAY datasets. For these subgraphs, we generate synthetic edge cost that are *correlated (CORR) with*, or *anti-correlated (ANTI) with*, or *independent (INDE) from* the distance between two nodes. Over these subgraphs, 150 random queries have been generated and executed. The average query time is reported in Figure 14. The correlated edge cost leads to the shortest *BBS* query time. Among the three types of edge cost, *BBS* method has the longest query time when edges have anti-correlated cost. On the contrary, the performance of our proposed algorithm is relatively constant to the edge-cost distributions and is much faster than the *BBS* method (Figure 14). Figure 15 shows the similar performance of queries over the backbone index on graphs with different types of edge-cost distributions. It is interesting to note that our proposed approach works even slightly better on graphs with anti-correlated or random edge cost than on graphs with correlated edge cost. This shows the potential of applying our methods to networks other than road networks because road-network cost are generally correlated to the distance between two nodes.

6.4 Case study

To illustrate the usefulness of the query results returned by our method, we visualize the result sets returned by our method and *BBS* from C9_NY_10K for a randomly picked query. Figure 16(a) plots all the 293 exact skyline paths, which differ from each other with only a tiny portion of the nodes/edges. When plotted, it looks like there are only very few alternative routes. Thus, the visualization cannot clearly show the many different routes. Figure 16(b) shows the five approximate skyline paths returned from our method, where only the highway entrances and the abstracted paths are drawn. The results returned by our method are more representative and succinct than the large number of exact solutions that share a large portion of nodes and edges, thus can better support decision making.



(a) Exact search results **(b) Approximate search results**

Figure 16: Use case demonstration

7 CONCLUSIONS

This paper introduces a new index structure (denoted as *backbone index*) to support efficient processing of SPQs over MCRNs. This index structure organizes the summarized graphs of the original graph with different summarization granularity in a hierarchical structure. Higher-level graphs summarize lower-level graphs by reducing the graph density. We implement a practical index construction approach that utilizes the idea of finding dense clusters to condense graphs. A corresponding query processing method is introduced to find approximate skyline paths by using our proposed index. Extensive experiments are conducted on nine real-world road networks. Our introduced query method can find reasonable approximate results efficiently, which are comparable to the results found by an exact SPQ query algorithm. The results also show that our backbone index has more efficient index size and building time than two other index structures adopted from the shortest-path-query supporting indexes.

REFERENCES

[1] 9th DIMACS Implementation Challenge. http://users.diag.uniroma1.it/challenge9/download.shtml.

[2] DB-Engines Ranking of Graph DBMS. https://db-engines.com/en/ranking/graph+dbms.

[3] GitHub Repository for this work. https://github.com/gongwolf/BackbonIndex.

[4] Neo4j Graph Platform. https://neo4j.com/.

[5] Real Datasets for Spatial Databases: Road Networks and Points of Interest. https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm.

[6] Takuya Akiba, Yoichi Iwata, and Yuichi 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, 2013.

[7] Saad Aljubayrin, Bin Yang, Christian S. Jensen, and Rui Zhang. Finding non-dominated paths in uncertain road networks. *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016.

[8] Neli Blagus, Lovro Šubelj, and Marko Bajec. Assessing the effectiveness of real-world network simplification. *Physica A: Statistical Mechanics and its Applications*, 413:134–146, 2014.

[9] G. Borruso. Network density estimation: A gis approach for analysing point patterns in a network space. *Trans. GIS*, 12:377–402, 2008.

[10] Zhan Bu, Zhiang Wu, Liqiang Qian, Jie Cao, and Guandong Xu. A backbone extraction method with local search for complex weighted networks. *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*, pages 85–88, 2014.

[11] S. Chawla, Venkata Rama Kiran Garimella, A. Gionis, and Dominic Tsang. Backbone discovery in traffic networks. *International Journal of Data Science and Analytics*, 1:215–227, 2016.

[12] Yi-Chung Chen and Chiang Lee. Skyline path queries with aggregate attributes. *IEEE Access*, 4:4690–4706, 2016.

[13] Zitong Chen, A. Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. P2h: Efficient distance querying on road networks by projected vertex separators. *Proceedings of the 2021 International Conference on Management of Data*, 2021.

[14] Liang Dai, Ben Derudder, and Xingjian Liu. Transport network backbone extraction: A comparison of techniques. *Journal of Transport Geography*, 69:271–281, 2018.

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

[16] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[17] Xiaoyi Fu, Xiaoye Miao, Jianliang Xu, and Yunjun Gao. Continuous range-based skyline queries in road networks. *World Wide Web*, 20(6):1443–1467, 2017.

[18] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[19] Qixu Gong and Huiping Cao. Technical report, TR-CS-NMSU-2022-0223, Supplementary Materials. https://computerscience.nmsu.edu/research/technical-reports.html.

[20] Qixu Gong, Huiping Cao, and Parth Nagarkar. Skyline queries constrained by multi-cost transportation networks. *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 926–937, 2019.

[21] Sheng Guan, Hanchao Ma, and Yinghui Wu. Attribute-driven backbone discovery. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 187–195, 2019.

[22] Andrey Gubichev, Srikanta J. Bedathur, Stephan Seufert, and Gerhard Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM '10*, 2010.

[23] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4:100–107, 1968.

[24] Y. Huang, N. Jing, and Elke A. Rundensteiner. Effective graph clustering for path queries in digital map databases. In *CIKM '96*, 1996.

[25] Shalev Itzkovitz, Reuven Levitt, Nadav Kashtan, Ron Milo, Michael Itzkovitz, and Uri Alon. Coarse-graining and self-dissimilarity of complex networks. *Phys. Rev. E*, 71:016127, Jan 2005.

[26] N. Jing, Y. Huang, and Elke A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Trans. Knowl. Data Eng.*, 10:409–432, 1998.

[27] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distributed Comput.*, 48:96–129, 1998.

[28] Hans-Peter Kriegel, Peer Kröger, Peter Kunath, Matthias Renz, and Tim Schmidt. Proximity queries in large traffic networks. In *GIS*, 2007.

[29] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert. Route skyline queries: A multi-preference path planning approach. *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 261–272, 2010.

[30] K. Lee, W. Lee, B. Zheng, and Yuan Tian. Road: A new spatial object search framework for road networks. *IEEE Transactions on Knowledge and Data Engineering*, 24:547–560, 2012.

[31] Qiyan Li, Yuanyuan Zhu, and J. X. Yu. Skyline cohesive group queries in large road-social networks. *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 397–408, 2020.

[32] Zijian Li, Lei Chen, and Yue Wang. G*-tree: An efficient spatial index on road networks. *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 268–279, 2019.

[33] A. Maratea, A. Petrosino, and Mario Manzo. Extended graph backbone for motif analysis. *Proceedings of the 18th International Conference on Computer Systems and Technologies*, 2017.

[34] D. Orellana and M. Guerrero. Exploring the influence of road network structure on the spatial behaviour of cyclists using crowdsourced data. *Environment and Planning B: Urban Analytics and City Science*, 46:1314 – 1330, 2019.

[35] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Y. Zhang, and Xuemin Lin. Progressive top-k nearest neighbors search in large road networks. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[36] Ning Ruan, Ruoming Jin, Guan Wang, and Kun Huang. Network backbone discovery using edge clustering. *arXiv preprint arXiv:1202.1842*, 2012.

[37] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, 2005.

[38] M Ángeles Serrano, Marián Boguná, and Alessandro Vespignani. Extracting the multiscale backbone of complex weighted networks. *Proceedings of the national academy of sciences*, 106(16):6483–6488, 2009.

[39] Yuan Tian, K. Lee, and W. Lee. Finding skyline paths in road networks. In *GIS '09*, 2009.

[40] Ulrike von Luxburg, Agnes Radl, and Matthias Hein. Hitting and commute times in large graphs are often misleading. 2010.

[41] T. Wang, C. Ren, Y. Luo, and J. Tian. Ns-dbscan: A density-based clustering algorithm in network space. *ISPRS Int. J. Geo Inf.*, 8:218, 2019.

[42] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[43] Victor Junqiu Wei, R. C. Wong, and Cheng Long. Architecture-intact oracle for fastest path and time queries on dynamic spatial networks. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[44] Bin Xu, Jun Feng, and Jiamin Lu. Continuous skyline queries for moving objects in road network based on mso. In *Proc. of the 12th Intl. Conf. on Ubiquitous Information Management and Communication*, IMCOM, pages 53:1–53:6. ACM, 2018.

[45] Bin Yang, Chenjuan Guo, Christian S. Jensen, Manohar Kaul, and Shuo Shang. Multi-cost optimal route planning under time-varying uncertainty. 2013.

[46] Bin Yang, Chenjuan Guo, Christian S. Jensen, Manohar Kaul, and Shuo Shang. Stochastic skyline route planning under time-varying uncertainty. *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, pages 136–147, 2014.

[47] Yajun Yang, Hang Zhang, Hong Gao, Qing hua Hu, and Xin Wang. An efficient index method for the optimal route query over multi-cost networks. *ArXiv*, abs/2004.12424, 2020.

[48] Man Lung Yiu and N. Mamoulis. Clustering objects on a spatial network. In *SIGMOD '04*, 2004.

[49] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. Dynamic hub labeling for road networks. *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 336–347, 2021.

[50] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. G-tree: An efficient index for knn search on road networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 39–48, 2013.