Backbone Index and GNN Models for Skyline Path Query Evaluation over Multi-cost Road Networks

QIXU GONG, New Mexico State University, Las Cruces, USA
HUIYING CHEN, New Mexico State University, Las Cruces, USA
HUIPING CAO, New Mexico State University, Las Cruces, USA
JIEFEI LIU, New Mexico State University, Las Cruces, USA

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 an 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 present efficient approaches to find approximate solutions to SPQs by utilizing the backbone index structure. Furthermore, considering making good use of historical query and query results, we propose two models, **S**kyline **P**ath **G**raph **N**eural **N**etwork (SP-GNN) and **T**ransfer SP-GNN (TSP-GNN), to support effective SPQ processing. Our extensive experiments on real-world large road networks show that the backbone index can support finding meaningful approximate SPQ solutions efficiently. The backbone index can be constructed in a 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. The results on the SP-GNN and TSP-GNN models also show that both models can help get approximate SPQ answers efficiently.

CCS Concepts: • **Computing methodologies** → *Supervised learning by classification*; *Instance-based learning*; • **Information systems** → **Query optimization**; **Graph-based database models**.

Additional Key Words and Phrases: Multi-cost Road Networks, Graph Neural Networks, Index, Skyline Path Queries, Data Augmentation

1 INTRODUCTION

Skyline path queries (SPQs) extend skyline queries to multi-dimensional networks (MDNs) [33]. 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, 33, 67, 69]. 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.

Example 1.1. (Motivation example) Consider that Alice needs to transport from a location A to destination B. Her concerns encompass various factors: travel time, expenses, and the frequency of traffic lights and stop signs along the route. Alice desires a swift arrival. At the same time, she is mindful of keeping expenses reasonable (for

Authors' Contact Information: Qixu Gong, Computer Science, New Mexico State University, Las Cruces, NM, USA, gongwolf@gmail.com; Huiying Chen, Computer Science, New Mexico State University, Las Cruces, NM, USA, hchen@nmsu.edu; Huiping Cao, Computer Science, New Mexico State University, Las Cruces, New Mexico, USA, hcao@nmsu.edu; Jiefei Liu, Computer Science, New Mexico State University, Las Cruces, New Mexico, USA, jiefei@nmsu.edu.

ACM 2374-0353/2024/4-ART
https://doi.org/10.1145/3660632

instance, avoiding toll roads that could spike costs). Additionally, Alice has a strong driving preference, avoiding roads heavily dotted with red lights or stop signs. Given these, she may not favor the path with the lowest expense (say p_{minE}) which might involve a long travel time and numerous traffic interruptions, nor the path with the shortest travel time (say p_{minT}), which could come with higher expenses. Instead, Alice may opt for (a) a path with a slightly higher expense but significantly less travel time than p_{minE} and a reasonable number of traffic lights, or (b) a path with a slightly longer travel time but considerably lower expense than p_{minT} and a small number of traffic lights. Alice's choice may not optimize in a singular dimension, but seeks a balance that aligns with her overall preference.

The evaluation of SPQs is very time-consuming due to a large number of solutions [20, 33] and the vast search space. Many works attempt to accelerate the query process by reducing the search space. In [33], the landmark index [32] 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 [33], Yang et al. [68] 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 [3, 10, 17, 20, 67].

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, 28, 34, 38, 78], 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 [33] 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 the 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).

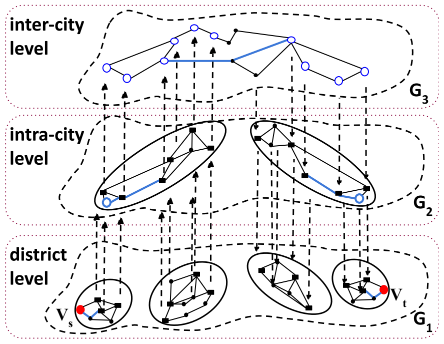


Fig. 1. Example of a backbone index

The idea of highway entrances is also utilized in partition-based approaches [28, 34, 78] 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 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.

All the approaches that utilize index techniques to support the evaluation of SPQs do not make use of historical results. There may be a large amount of historical results when a system supports running such queries for a long time. This work further explores strategies of utilizing historical results to accelerate query evaluation. It is well known that most neural network (NN) models can well capture the non-linear patterns hidden in the data and the NN models are much smaller than the data. Our work thus adopts the use of one type of NN models, Graph neural networks (GNNs), to take advantage of historical query results to efficiently support SPQ evaluation. We propose two models, **S**kyline **P**ath **G**raph **N**eural **N**etwork (SP-GNN) model and **T**ransfer SP-GNN (TSP-GNN) model, to support the search. These two models are properly designed to encode historical query results together with the historical queries) as training instances. In the SPQ query processing stage, these two models serve as a filter to limit the search space of the query, thus to process the query more efficiently. SP-GNN can effectively support queries over smaller graphs, while TSP-GNN can support SPQs over larger graphs more effectively. We note that the novelty of SP-GNN and TSP-GNN is not the design of a new GNN architecture for SPQ processing. Instead, most GNN architectures can be adopted in our approaches. This makes the design more general and easily to be utilized.

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. Thus, it can also be utilized to support GNN-based models by generating training data more efficiently.
- 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 to 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 propose two GNN-based models that take advantage of historical query results to further support efficient SPQ evaluation. These two models can adopt most GNN architectures as the building blocks, which makes them general to be utilized. The idea of utilizing NN models (or machine learning models in general) to summarize data and facilitate query processing can be applied to process other types of queries.
- 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. Our proposed GNN-based approaches are presented in Sections 6. Experimental results are reported in Section 7. Section 8 provides an overall discussion of the proposed techniques. Section 9 concludes the work and discusses possible future works.

2 RELATED WORKS

2.1 Skyline queries on road networks

The SPQ problem over an MCRN is first proposed and studied in [33, 56]. Kriegel et al. [33] propose to use landmark index to calculate lower bounds of paths and reduce the search space of SPQs. Tian et al. [56] utilize the partial path dominance test to prune search space. Yang et al. [68] 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 [14] search. Shekelyan et al. [52] present a different type of SPQ, linear skyline path queries, which use a linear combination of multiple cost values to define the optimality of paths.

More recent works evaluate different SPQ variants. The work [17, 67] 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 [37] 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 [33]) answer skyline queries without the support of any index structures. Although using the landmark index [33] and finding shortest paths on each dimension [68] are efficient ways to prune the search space, the query process using these techniques is 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 [73] 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.

In summary, existing studies on skyline query processing either do not use index structures or use index structures with only one level. Our work is different in that we introduce a novel hierarchical index structure with different levels of abstraction to the graphs. Our GNN-based methods further create highly summarized models by using historical results to improve query efficiency.

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 [14] and the A* [23] algorithms are the most successful and widely used search methods. These traditional search methods are not practical to work for the large graphs collected in recent years. Wagner et al. [60] investigated reducing the search space of Dijkstra's algorithm by using geometric containers that encapsulate pre-computed shortest-path information. Despite the improvement, the graphs that this technique applies to are still relatively small. Yang et al. [71] addressed the problem of finding the shortest path passing through a set of user-specified vertices in large graphs. It proposes novel exact and approximate heuristic algorithms to improve search efficiency. However, their proposed approach cannot be easily extended to answer SPQ over MCRNs.

The design and use of an index structure to keep pre-calculated path information is inevitable. For road networks, graph-partition [28, 34, 38, 78] and shortcut-based [4, 18, 31, 66] 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 an 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. Contraction hierarchies (CH) [4, 31] are built using such shortcut-based method while one [4] focused on answering queries from graphs with time-dependent edges and the other [31] targeted at improving the pre-processing and memory efficiency through the introduction of time-dependent CH. For shortest path queries or routing problems, the number of shortcuts is still manageable. However, for SPQ queries, 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 [28, 34, 78] 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 [11, 45, 76] 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 [11, 45] or (ii) generate large number of shortcuts from each tree node to its neighbors in the SPQ setting. Other approaches [2, 22, 50] 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.

On MCTN, people also work on answering other types of constrained queries including constrained route planning (e.g., [62]) and Multi-Constraint Shortest Path (MCSP) (e.g., [40]). Constrained route planning finds the optimal solution from a source to a destination with a weight constraint. MCSP are still *shortest paths* w.r.t. one dimension of the edge weight in an MCRN, while their costs on the other edge weight dimensions are constrained. These query problems are different from our work because they still use one criterion (e.g., path length, or one edge weight) to define the optimal/shortest solution.

Traveling salesman path (TSP) problem is another type of location-based queries. Rice et al. [47] woked on solving the Generalized Traveling Salesman Path Problem (GT-SPP). Their algorithms find the shortest path from a location to another location that passes through at least one location from each of a set of generalized location categories.

To summarize, strategies to process existing location-based queries focus on finding shortest paths. Their direct utilization to process SPQs is prohibitively expensive. For non-shortest path query processing, different constraints are used to improve their search algorithm. Our research problem does not impose any special constraints, which makes the existing methods not directly applicable to process SPQs.

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 [7, 21, 27, 48, 51] study the backbone extraction problem for different networks with specialized research interests. In [48], 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 [27] merges nodes and edges by creating shortcuts with the intention to preserve the topology of the original graph. The works in [7, 51] define a criterion to examine the importance or relevance to a network, and adopt strategies for edge sampling [5] or edge filtering (or pruning) [7, 13] 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 [9, 13, 42] for specialized MCRNs. Thus, they cannot be directly applied to build indexes to support SPQs over general MCRNs. Moreover, most of the existing methods [13] cannot guarantee the connectivity of the extracted backbone graph.

2.4 Using machine learning methods in query processing

Observing the success of the machine learning techniques, the database community utilizes such techniques for different types of database tasks [35]. Given that searching for useful information from database is the cornerstone of many database applications, much work has utilized machine learning techniques to improve the efficiency of query evaluation and optimization (e.g.,[15, 24, 29, 41, 53, 58]).

Increasing amount of work has utilized machine learning techniques to manage and analyze spatial data [49]. For example, deep generative models are used to detect anomalous trajectory [39]. Deep reinforcement learning is utilized to facilitate similar subtrajectory search [64] and trajectory simplification [63]. Attention-network based models are designed for approximate trajectory similarity calculation [70]. There are also efforts trying to improve the efficiency of model training. For example, Zeng et al. [75] introduce a GNN framework COSAL to avoid aggregating all the modes in a graph when the analysis task is related to a much smaller number of target nodes. This strategy cannot be directly applied to our work because our trained model is supposed to support future ad-hoc queries where every node has the potential to be accessed.

Some recent work has utilized GNN models to solve the shortest path searching problem. SPAGAN [72] used path-based attention to explore the graph structure and aggregate information from distant neighbors more effectively. Despite their success in improving the shortest path finding problem, these methods unfortunately only work for unweighted/no-cost graphs. Shortest paths are used to improve the building of neural network (NN) models. Abboud et al. [1] proposed a novel message passing neural network (NN) that utilizes shortest paths between nodes. With this message passing NN, a node can directly communicate with its direct neighbors and shortest-path neighbors.

On general graphs (which may not be road networks and do not have spatial information), NN models are also utilized to facilitate query processing. Nishad et al. [43] has worked on reachability estimation over graphs by introducing a position-aware inductive GNN. This GNN can represent both node attributes and node positions in graphs.

The work proposed by Chang et al. [8] is most related to this work. It proposed to use graph contrastive learning to learn a task-agnostic road network embedding. However, it is not trivial to directly apply such embedding on multi-cost road networks.

In a word, no approach applies machine learning models to process SPQs despite different models are utilized to solve different types of queries including general shortest path queries and specialized queries.

2.5 Learned indexes

Learned indexes are a novel approach that leverages machine learning models to enhance query-processing efficiency and reduce index size [36, 55, 57, 77]. Different from traditional index techniques such as B+-trees, learned indexes offer improved query latency and storage optimization. Existing learned indexes have been developed for various scenarios including key lookup and insertion, concurrency control, and bulk loading, aiming to address practical challenges in different settings. Researchers have explored spatial learned indexes to handle spatial data efficiently, with recent advancements focusing on multi-dimensional interpolation functions and dynamic encoding techniques to enhance prediction accuracy, building time, and storage overhead. These innovative learned index models, such as DILI [36], demonstrate superior performance in key search time, query execution time, and write performance compared to traditional index structures. LIMS [57] is proposed to use data clustering and pivot-based data transformation techniques to build learned indexes for efficient similarity query processing in metric spaces.
Different from the learned indexes technique, this work uses GNN-based models to accelerate query efficiency without changes to indexes, the backbone index.

3 TERMINOLOGY AND PROBLEM STATEMENT

This section presents the terminology used in our problem statement and solution. Table 1 lists the symbols utilized in the definitions and our methodology.

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, where d is the dimension of the

Table 1. Notations used in problem definition and methodology

Symbol	Meaning	Definition section
G	Graph	Sec. 3
v, V, $\|V\|$	One graph node, graph node set, number of graph nodes	Sec. 3
e, E, $\|E\|$	One graph edge, graph edge set, number of graph edges	Sec. 3
$DP(e)$	Degree pair of edge e	Sec. 3.2
$deg(v)$	Degree of a node v	Sec. 3.2
d	Number/dimension of edge cost	Sec. 3
w, W	d-dimensional cost of one edge, edge weight tensor	Sec. 3
p, $cost(p)$	one path, the cost of a path	Sec. 3
$p_i\|\|p_j$	concatenation of two paths p_i and p_j	Sec. 3
$p \prec p'$	path p dominates path p'	Sec. 3.1
$G_0, G_1, \cdots, G_L$	Different levels of abstracted graph in the backbone index where $G_0 = G$	Sec. 4.1
$\mathcal{N}_{1st}(v)$, $\mathcal{N}_{2nd}(v)$	The 1-hop and 2-hop neighbors of the node v	Sec. 4.2
cluster_coefficient(v)	A node's cluster coefficient	Sec. 4.2
p_{ind}	Cluster condensing threshold	Sec. 4.2
$C_{i,j}$	The j-th dense cluster for level i abstract graph G_i	Sec. 4.2.4
$\mathcal{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_{\|G.V\|}]^T$	Feature matrix (spatial coordinates of nodes) of graph G; $\mathcal{X} \in \mathbb{R}^{\|G.V\| \times 2}$	Sec. 6.1
B_e, B_{gnn}, B_{fc}	# of hidden features in GNN models after the embedding layer, GNN layers, and fully connected layers	Sec. 6.2
$\mathbf{H}_{embed}$, $\mathbf{H}_{G_i}$, $\mathbf{H}_o$	Hidden features in GNN models after the embedding layer, GNN layers, and fully connected layers	Sec. 6.2
$\mathbf{H}_{v_s}$, $\mathbf{H}_{v_t}$, $\mathbf{H}_{query}$	Hidden features learned from the GNN model representing the nodes v_s, v_t, and query (v_s, v_t)	Sec. 6.2

edge costs. Let $|G.V|$ and $|G.E|$ be the number of graph nodes and edges respectively. Each edge $e \in E$ representing a road segment is associated with a d-dimensional cost vector w, where w_i is the value of the i-th cost of edge e. Examples of edge weights include the length of the edge, the speed limit of the edge, and whether there is construction work going on the road segment. Edge weight can be the estimated travel time, although it can be calculated from the distance and the speed limit of the edge. All the edge weights in the graphs are static. We will explore strategies to deal with graphs with dynamic weights as our future work. 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 $p(v_s \leftrightsquigarrow v_t)$ or the list of nodes on the path. For example, (v_3, v_4, v_1, v_8) in Figure 2 is a path $p(v_3 \leftrightsquigarrow v_8)$. 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 a different single dimension is utilized. Given two paths p_i and p_j where the ending node of p_i is the same as 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, 33] 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 an 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 $p(v_s \leftrightsquigarrow v_t) \in \mathbb{P}$ is called a **skyline path** from v_s to v_t. Where there is no ambiguity in the context, we use p to represent $p(v_s \leftrightsquigarrow v_t)$. 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 v_{s_e} and v_{t_e}, the degree pair of e, $DP(e) = \langle e.first, \langle e.second \rangle$, is defined as follows.

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

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**.

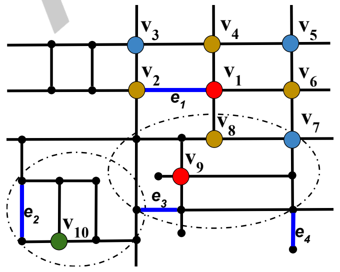


Fig. 2. Degree pair example, where $DP(e_1)$=$\langle 4, 4 \rangle$, $DP(e_2) = \langle 2, 3 \rangle$, $DP(e_3) = \langle 3, 4 \rangle$, and $DP(e_4) = \langle 1, 4 \rangle$.

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.

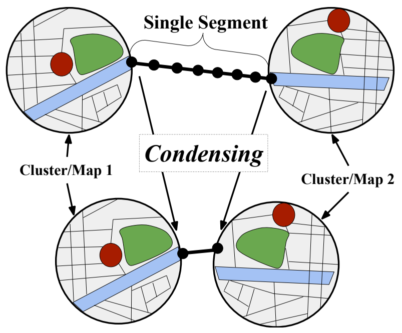


Fig. 3. Single segment example

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 into 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 for 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 exchangeably 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 [61, 74] adopts the shortest path distance as the distance measurement. This is not suitable for MCRNs. Without extra information such as user pattern data [44], POIs [61], and trajectory location data [6], 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 [65] 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 the 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

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

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 a 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 $\mathcal{N}_{2nd}$ 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 fewer 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.

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

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 $p_{ind} * |G.V| = 3$. $\vec{L}[0] + \vec{L}[1] = 3 \leq 3$ 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*.

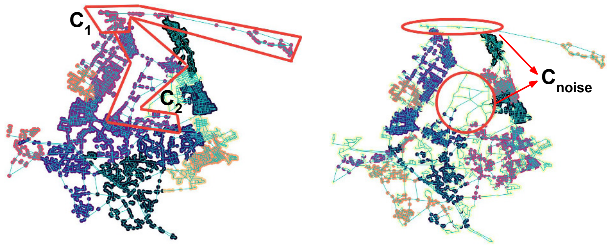
A node v is treated as a noise node if $|N_{1st}(v) + N_{2nd}(v)| < noise\_val$. The clustering procedure sets low-density nodes as noises when the condensing threshold is used. For example, two clusters, C_1 and C_2, in Figure 4(a) contain low-density nodes. These two clusters are condensed in the index 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 [59].

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.



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

Fig. 4. Example of dense clusters on **C9_NY_5K**

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 for choosing edges. Our method first chooses the edges (not random edges) 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).

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 $\cup_{j=1}^{c} C_{i,j}.\tilde{V}$.

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

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 $\Delta V_i$, removed edges $\Delta E_i$
Output : Updated $\Delta V_i$, updated $\Delta E_i$, 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 $C$, $\Delta V_i$, $\Delta E_i$
```

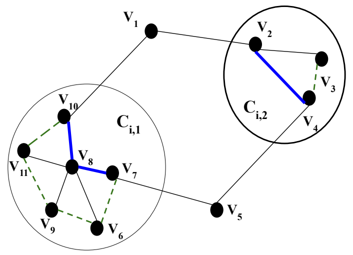


Fig. 5. Example of highway entrances

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 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

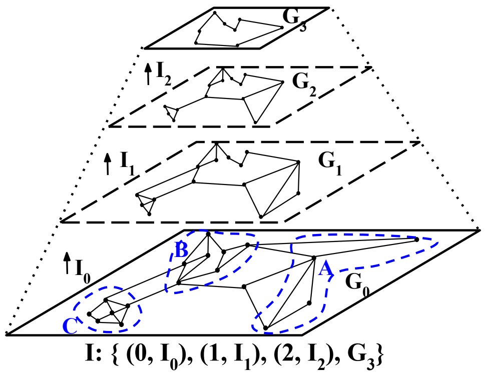


Fig. 6. Index example

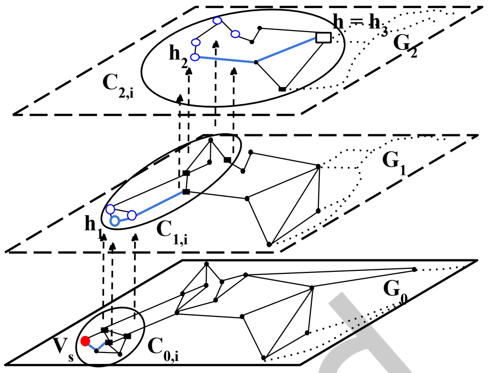


Fig. 7. Paths in index

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 $I_{list}$ =$\emptyset$;
4      do
5          /* Step 1: Regular Summarization of $G_i$ */
6          ($\Delta E_i$, $I_i$, $G_{i+1}$) = 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 $|G_{i+1}.V| \neq 0$ & $\Delta E_i \leq p * |G_0.E|$ then
10             $\Delta E_{new}, I_{new}$ = AggressiveGraphSummarization( $G_{i+1}$);
11             if $|\Delta E_{new}| \neq 0$ then
12                 Update $I_i$ using $I_{new}$;
13                 $\Delta E_i = \Delta E_i \cup \Delta E_{new}$;
14         L=i, i=i+1 ;
15     while $|G_{i+1}.V| \neq 0$ and $\Delta E_i \geq p * |G_0.E|$;
16     landmark($G_L$);
17     return $I_{list}$, $G_L$
```

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., BFS mentioned in Section 7s). 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) a 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 a 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.

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 [32] 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 $\mathbb{P}_{v_s}^h$ 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 $\mathbb{P}_{v_s}^h$ and $\mathbb{P}_{v_t}^h$ 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 $\mathbb{P}_{v_s}^{h_s}$ and $\mathbb{P}_{v_t}^{h_t}$ 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_v^i that v can reach G_i from G_{i-1} directly. When v does not exist in G_{i-1}, then H_v^i 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.

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 $0 \le i \le L$ 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 $0 \le i \le L$ 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     /* BFS 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
```

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 *m_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 $\mathbb{D}.keys$.

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 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 GNN-BASED APPROACHES

The Backbone index together with the query processing algorithm presented in the preceding sections establish a robust foundation for efficient approximate skyline path query (SPQ) evaluation on Multi-Cost Road Networks (MCRNs). The hierarchical index, rooted in the concepts of backbone and clustering, is a typical representative of traditional query processing approach.

The Backbone index, like other existing index techniques, does not utilize any historical query information. Historical queries and their solutions can provide very useful information for future queries when the search space of future queries overlaps with historical query results. However, all the historical results cannot be directly stored to support future queries because their size may be huge (even larger than the original graph).

In this section, we propose a strategy to complement the index-based query processing technique. It utilizes historical query results to support SPQ evaluation through the design of graph neural network (GNN) models. The ideal case of the historical query results are exact skyline paths (returned by an exact SPQ processing algorithm). However, for large graphs, it is almost unreasonable and impractical to get exact solutions due to the long running time and large memory use. Thus, our GNN-based approaches also accept the approximate skyline paths returned by the Backbone index as historical results to train the model. Despite approximate, the abstract graphs in the Backbone index capture essential graph characteristics, they can still guide the GNNs to focus on relevant features during training and searching. By incorporating the distilled knowledge from the results generated from the Backbone index into our GNN-based approaches, we enhance the ability of our approaches to leverage historical approximate query results.

It is worth noting that not all historic results need to be stored. When we get a query's results, we can use them to train our GNN model. When the train stage converges, these results can be discarded. Alternatively, a small portion of the results can be stored for future training. Our experiments demonstrate that not all the historical query results are needed to train a GNN model for effective approximate query processing.

Machine learning models have been known to be able to capture the patterns hidden in large amounts of data. For our problem, we consider designing GNN models (instead of other machine learning models) because the result of a query is a set of skyline paths, which form a subgraph. GNNs are a good candidate to capture connectivity information among nodes. With the GNN models learned from historical query results, we expect to use the GNN models to help reduce the search space for a new coming SPQ.

We build one GNN model for each MCRN G from all the training instances, which can be accumulated historical results or manually generated query results. For any new SPQ issued to this MCRN G, the GNN model is used to find the search space when evaluating an SPQ query, which is much smaller than the whole graph as the search space.

We propose two GNN models. The first model, Skyline Path Graph Neural Network (SP-GNN), is directly built by utilizing the exact query solutions from the original graph. The second model, Transfer SP-GNN (TSP-GNN), is built using path solutions from the summarized graphs of the original graph. It particularly works on large graphs. Note that the novelty of the GNN-based approaches is not the design of a new GNN architecture. Instead, most existing GNN architectures can be adopted in our models. In this paper, we implemented the transformer model [54], which has shown great success. However, other GNN architectures can be utilized.

In what follows, we explain the details of training-instance construction (Section 6.1), the architecture of SP-GNN, the loss function, the search process using SP-GNN, and the design of TSP-GNN.

6.1 Construct training instances

The first step of creating an SP-GNN model is to prepare its training instances (or samples). Each training sample consists of one historical SPQ and its solution (i.e., all the skyline paths for this SPQ).

For a query, based on its solution set of skyline paths, we can categorize all the graph nodes to be three types. The first type of nodes does not occur in any skyline paths. The nodes in the second type form the skyline paths. The nodes in the third type of nodes are the direct neighbors of the second type of nodes. Let use 0, 1, and 2 to denote these three types of nodes respectively. Then, for any given query, we can define its corresponding output (or ground truth) as a length-$|G.V|$ vector Y. The value $\mathrm{Y}[i]$ represents the type of a node v_i.

Formally, we define a training instance as follows.

Definition 6.1 (Training sample). For a graph G, a training sample of its corresponding SP-GNN model is a query together with the set of skyline paths from G to answer this query. The output of this training sample is a length-$|G.V|$ vector Y where each $\mathrm{Y}[i] \in \{0, 1, 2\}$.

A training instance provides the approximate search space when evaluating an SPQ. We limit the search space of a query to be the set of nodes with types 1 or 2. Including nodes with type 2 increases the search space to find more answers. We also note that the search space increase incurred due to the inclusion of type 2 nodes is not dramatic, thus it does not increase the query overhead significantly. If we want to further enlarge the search space, we can label nodes that are 2-hops or 3-hops away from the type-1 nodes as node type 2.

Example 6.2. Figure 8 shows an example of a training instance for a query which are represented with the star nodes. Assume there are only two skyline paths between them, $(v_s, v_1, v_2, v_3, v_4, v_5, v_t)$ and $(v_s, v_6, v_2, v_3, v_7, v_5, v_t)$.

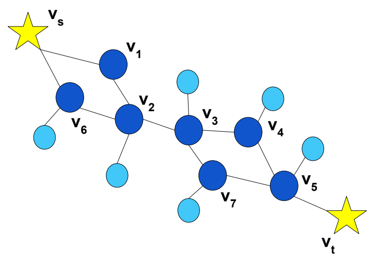


Fig. 8. Example of a training instance and the approximate search space of a query represented with star nodes (dark blue nodes are labeled with type 1, and light blue nodes are labeled with type 2)

The nodes on the paths are labeled as type 1 and marked using dark blue. Note that nodes v_2, v_3, and v_5 occur in both paths, which is commonly observed from the answer set. The 1-hop neighbors of these nodes are the type-2 nodes and are marked using light blue in the figure. These nodes form the minimum search space of this query. During the query stage, an ideal GNN model predicts all these nodes to be in the search space although an actual GNN model may not accurately predict all these nodes to be in the search space.

The coordinates of the nodes in G are normalized to be in the range of [0,1] using min-max normalization. The normalization is conducted to make the values in the same range. Normalization is a common process in utilizing gradient descent type of training algorithms to learn a model so that the training can converge faster. Each node v_i in the graph is associated with the normalized 2-dimensional coordinates, which are stored as a feature vector $\mathbf{x}_i \in \mathbb{R}^2$. The entire feature matrix for a graph G is $\mathcal{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_{|G.V|}]^T \in \mathbb{R}^{|G.V| \times 2}$.

6.2 SP-GNN architecture and its training stage

The architecture of the SP-GNN model is shown in Figure 9. SP-GNN consists of a node embedding layer, two GNN layers, a layer to incorporate query information, two fully connected (FC) layers, and the last output layer.s

The node embedding layer converts the two features of each node (which are the two normalized spatial coordinates) to a node embedding with B_e hidden features. This transformation can be represented as

$$\mathbf{H}_{embed} = \mathcal{X}\mathbf{W}_e + \mathbf{b}_e$$

where $\mathbf{W}_e \in \mathbb{R}^{2 \times B_e}$ keeps the weight parameters and $\mathbf{b}_e \in \mathbb{R}^{B_e}$ is the bias vector.

Then, the embedded node features $\mathbf{H}_{embed}$ and the adjacency matrix $\mathbf{A}$ of the graph G are passed to the GNN layers in the following form.

$$\mathbf{H}_{G1} = \mathcal{G}_{upd}(\mathcal{G}_{agg}(\mathbf{H}_{embed}, \mathbf{A}), \mathbf{A})$$

$\mathcal{G}_{upd}$ and $\mathcal{G}_{agg}$ are differentiable and permutation invariant functions (e.g., element-wise sum, mean, or max). They are also called the *update* and *message* functions respectively. The architecture can include multiple GNN layers. The features in the next GNN layer are calculated as $\mathbf{H}_{G_{i+1}} = \mathcal{G}_{upd}(\mathcal{G}_{agg}(\mathbf{H}_{G_i}, \mathbf{A}), \mathbf{A})$. The last GNN layer outputs the representation of each node that incorporates the node's neighbor information. Let $\mathbf{H}_G$ ($\mathbf{H}_G = \mathbf{H}_{G_2}$ in this architecture) represent this representation. For the query's starting and ending nodes v_s and v_t, we can extract their corresponding hidden features $\mathbf{H}_{v_s}$ and $\mathbf{H}_{v_t}$ from $\mathbf{H}_G$.

Next, the first layer after the GNN layers concatenates the hidden features $\mathbf{H}_{v_s}$ and $\mathbf{H}_{v_t}$ to $\mathbf{H}_G$ as

$$\mathbf{H}_{query} = CONCATENATE(\mathbf{H}_G, \mathbf{H}_{v_s}, \mathbf{H}_{v_t}).$$

$\mathbf{H}_{query}$ captures the information that this representation is for the query (v_s, v_t).

Finally, the hidden features $\mathbf{H}_{query}$ that incorporate the query information are passed to two fully connected layers to get $\mathbf{H}_o$ as follows.

$$\mathbf{H}_o = (\mathbf{H}_{query}\mathbf{W}_{B_{fc1}} + \mathbf{b}_{B_{fc1}})\mathbf{W}_{B_{fc2}} + \mathbf{b}_{B_{fc2}}$$

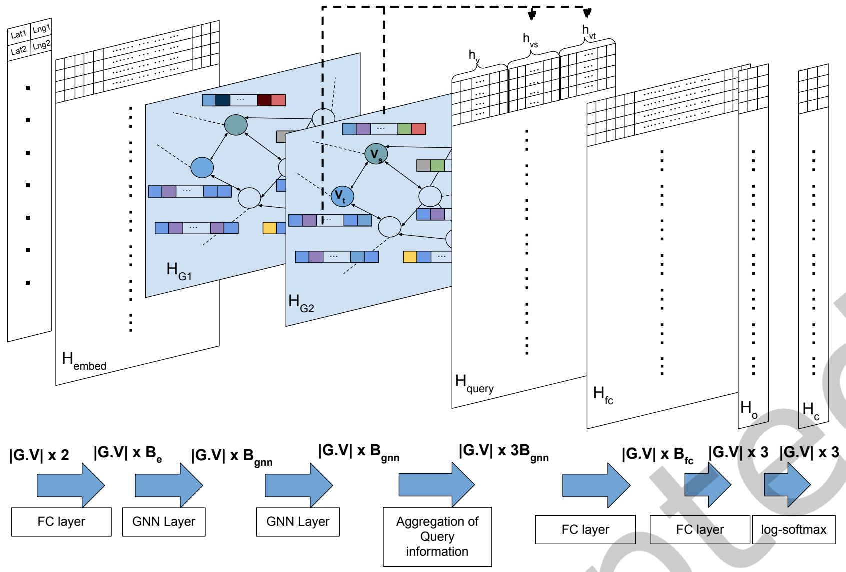


Fig. 9. Architecture of SP-GNN (The number of hidden features used in the two GNN layers is set to be the same B_{gnn} to simply the model representation)

Here, $\mathbf{W}_{B_{fc1}} \in \mathbb{R}^{3B_{gnn} \times B_{fc}}$ and $\mathbf{W}_{B_{fc2}} \in \mathbb{R}^{B_{fc} \times 3}$ are the weight parameter matrices, where B_{gnn} and B_{fc} are the number of features in the hidden layers and the fully-connected layers respectively, and $\mathbf{b}_{B_{fc1}}$ and $\mathbf{b}_{B_{fc2}}$ are bias vectors. The *Softmax* function can be applied to the output tensor $\mathbf{H}_o$ to calculate the probabilities of nodes belonging to the three different classes (node types in this case, which are 0, 1, or 2).

6.3 Loss function

The loss function is defined to be the Negative Log-Likelihood (NLL) function, which is closely utilized together with the Softmax function.

Let C represent the set of different class labels that a node can belong to (in our case, C={0, 1, 2}) and let $|C|$ be its cardinality. In our model, for each training instance, the output is a tensor $\mathbf{H}_o \in [0,1]^{|G.V|*|C|}$, which represents the predicted probability for all the graph nodes belonging to each class type. Given $\mathbf{H}_o$, each node v has a corresponding $|C|$-dimensional vector output $\mathbf{h_v} = \mathbf{H}_o[v]$. The value $\mathbf{h}_v[c]$ denotes the probability that the node v belongs to the class c and is defined below using the Softmax function.

$$Softmax(\mathbf{h}_v[c]) = \frac{exp(\mathbf{h}_v[c])}{\sum_{j=0}^{|C|} exp(\mathbf{h}_v[j])}$$

Recall that $\mathrm{Y}[v]$ is the ground truth node type of the node v, the NLL loss of the node v is calculated as

$$NLL(v) = -log(Softmax(\mathbf{h}_v[\mathrm{Y}[v]])).$$

A node v that has a higher probability of being the ground-truth node type $\mathrm{Y}[v]$ should have a lower $NLL(v)$ value.

Given all the training instances in the set Q_{train}, the optimization loss function is formulated as below.

$$\mathcal{L}_{min} = \sum_{q \in Q_{train}} \frac{1}{|G.V|} \sum_{i=0}^{|G.V|} NLL(v_i)$$

6.4 Search using GNN

This section presents the details that an SPQ query is evaluated after a GNN model M is trained. Algorithm 4 outlines the major steps.

Algorithm 4: SPQ query evaluation over SP-GNN model

```
Input  : Graph $G$, SPQ $q = (v_s, v_t)$, SP-GNN model $M$
Output : skyline paths $\mathbb{P}$ for $q$
1  begin
2      Form a query instance with $(G, q)$ ;
3      Make predictions $V_{label} = M$.predict($G$, $q$) ;
4      $V_{sub}$ = All the nodes that have a non-zero label in $V_{label}$ ;
5      foreach node $v \in V_{sub}$ do
6          $p_s$ = Find the shortest path in G from $v$ to $v_s$ on all the cost dimensions;
7          Add all the nodes in $p_s$ to $V_{sub}$;
8          $p_{tgt}$ = Find the shortest path in G from $v$ to $v_t$ on all the cost dimensions;
9          Add all the nodes in $p_t$ to $V_{sub}$;
10     $\mathbb{P}$ = $BBS(G, q, V_{sub})$ /* Conduct BBS query over $G$ by limiting the search space to be $V_{sub}$*/;
11     Return $\mathbb{P}$;
```

For a given new query q, we first form a testing instance for the model (Step 2). For this testing instance, the model M predicts the nodes' types (which can be 0, or 1, or 2) (Step 3). The nodes with a non-zero node type are considered in the search space of q (Step 4).

Note that the nodes that are predicted in the search space V_{sub} may not form a connected subgraph. Directly utilizing these nodes, we are not guaranteed to get a solution. To make these nodes connected, and also to make sure that from v_s we can reach v_t by going through some of these nodes, we enlarge the set V_{sub}. We conduct the shortest path search from each node $v \in V_{sub}$ to the starting and target query nodes on all the cost dimensions. The nodes in these shortest paths are added to V_{sub}. Then, we conduct the baseline SPQ approach, *BBS* method, on the original graph using V_{sub} to constrain the search space. The algorithm disallows any node in V_{sub} to be expanded. The returned paths are approximations of the exact skyline paths because of the reduced search space. This method leads to a good approximation and a very efficient search, which can be verified through our experiments.

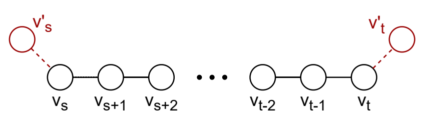
6.5 Transfer SP-GNN model and data augmentation

SP-GNN works well on constraining the search space to answer SPQ on small graphs. However, it has two major issues when applied to large graphs. First, the training sample construction stage becomes very time-consuming or impossible for large graphs. Even for graphs with just 30K nodes, it can take multiple days to generate the exact solutions for just one thousand queries whose answers can be found within fifteen minutes [46]. That is the very reason that we design the backbone index to facilitate the search process. Second, to get a well-trained GNN, a large number of training instances are needed, which requires generating query solutions for a large number of SPQs.

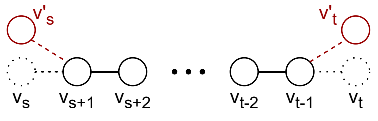
To solve the above mentioned issues, this section introduces Transfer SP-GNN (TSP-GNN) to support evaluating SPQs on large graphs. The idea is to generate training samples from backbone abstracts, which are much smaller than the original graph. We explain how TSP-GNN works. Given a graph G, we first generate backbone abstracts of G, which can be done through the construction of backbone index. Then, we generate training samples by calculating the skyline paths from the backbone abstracts (instead of the original graph) for a set of queries. One TSP-GNN model $\mathcal{M}_s$ can be trained using all the training samples generated from the backbone abstracts. Note that all the coordinates of nodes in the graph abstracts are also normalized using the min-max normalization.

When a new query q is issued on G, we use the model $\mathcal{M}_s$ that is learned from the training samples calculated using the backbone abstracts to embed the original graph G. The query q is used in the stage of generating $\mathbf{H}_{query}$ from the model $\mathcal{M}_s$. After we get the output from the Softmax layer of the $\mathcal{M}_s$ model, the same process in Section 6.4 is applied to generate the search space V_{sub} to finish the SPQ solutions.

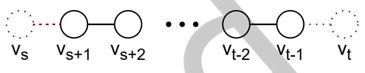
6.5.1 Data augmentation. Even with TSP-GNN, generating a large number of training instances still takes a lot of time. We further propose a strategy to alleviate the issue of training instance generation by introducing several basic ways to augment training instances. The data augmentation generates more training synthetic instances.



(a) Augmentation Strategies 1&2



(b) Augmentation Strategy 3



(c) Augmentation Strategy 4

Fig. 10. Data Augmentation Strategies

Our data augmentation strategies utilize a small number of skyline paths, which can be either exact or approximate, and extend such paths to create synthetic training instances for new queries. For a given skyline path $p(v_s \leftrightsquigarrow v_t)$: $(v_s, v_{s+1}, v_{s+2}, ..., v_{t-2}, v_{t-1}, v_t)$, we augment it to get training instances of other queries using four strategies. The first data augmentation method finds the direct neighbors of both the starting and the destination nodes v_s and v_t to create new queries. For example, if v'_s (and v'_t) is the direct neighbor that does not exist in the path p, this method creates an instance $(v'_s, p(v_s \leftrightsquigarrow v_t), v_{t'})$ for a new query (v'_s, v'_t). This method is demonstrated in Figure 10(a). If there are multiple v'_ss (or v'_ts), we choose the one with the least value in the first cost dimension to make sure the augmented training instance for a new query is still a skyline path. This rule also applies to the augmentation strategies 2&3. If either v'_s or v'_t does not exist, p is augmented using the second strategy which creates new queries by only replacing either the starting or the ending nodes of the query. One example of instance augmented using the second strategy from Figure 10(a) is $(v'_s, p(v_s \leftrightsquigarrow v_t))$ for query (v'_s, v_t).

The third strategy, which is demonstrated in Figure 10(b), utilizes the direct neighbors of the second path node v_{s+1} and the second-to-the-last path node v_{t-1}. Suppose that v'_s and v'_t are the direct neighbors of v_{s+1} and v_{t-1} respectively and they are not on the path p. The new augmented instance is $(v'_s, v_{s+1}, v_{s+2}, ..., v_{t-2}, v_{t-1}, v'_t)$. I.e., replacing v_s and v_t in p with v'_s and v'_t simultaneously. If either of v'_s and v'_t does not exist, this augmentation strategy is not utilized.

The fourth strategy, which is demonstrated in Figure 10(c), extracts a sub-path of p: $(v_{s+1}, v_{s+2}, ..., v_{t-2}, v_{t-1})$ and uses it as a skyline path for the new query (v_{s+1}, v_{t-1}). These four strategies can be extended to create more augmented training instances for the TSP-GNN model.

6.6 Complexity analysis

A general GNN model aggregates the features of neighbor nodes by utilizing simple functions, such as mean, summation, and concatenate. The time complexity of these functions is constant. The complexity of the aggregation operation of a GNN model is $\sum_{i=0}^{n} d_i$ where d_i is the degree of the node v_i. In real applications, the average degree of a road network is generally no more than 4. Given a graph G with $|G.V|$ nodes and $|G.E|$ edges, the complexity of a general GNN model is $O(|G.E|)$ since $\sum_{i=0}^{|G.V|} d_i \approx C_1|G.E|$, where C_1 is a constant.

Our SP-GNN model consists of a linear embedding of node coordinates, two GNN layers, one linear layer of concatenating the node embedding of two query nodes, and two fully connected layers. The computation cost is $C_2|B_e| \cdot |B_{gnn}| + 2 \cdot C_3 \cdot |G.E| + 2 \cdot C_4|B_{fc}|^2$, which is of complexity $O(|G.E| + B^2)$ where C_2, C_3 and C_4 are constant factors and $B = max(|B_e|, |B_{gnn}|, |B_{fc}|)$ is the maximum number of hidden features (Figure 9). For the TSP-GNN

Table 2. 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

model, the time complexity for constructing the backbone abstracts is $O(|G.V|log(|G.V|))$ and the complexity of the model is also $O(|G.E| + B^2)$. In general, the complexity of TSP-GNN is $O(|G.V|log(|G.V|) + |G.E| + B^2)$.

7 EXPERIMENTS

7.1 Experimental settings

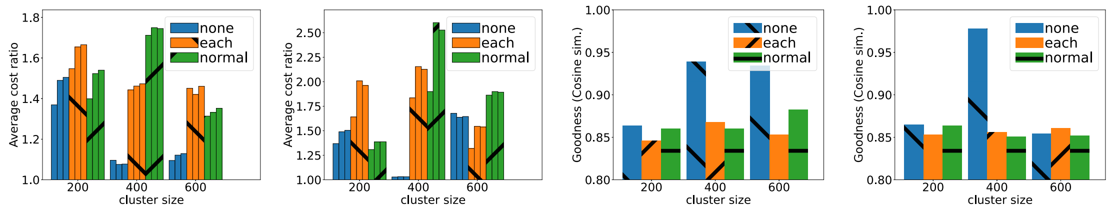
Our experiments are conducted on a desktop with an Intel(R) 3.60 GHz CPU, 32 GB main memory, and 2 TB HDD, running Ubuntu 18.04. All the algorithms are implemented using Java 13. We use Neo4j[1], the most popular graph database (according to DB-Engines ranking[2]), to store all the graphs. The page size 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. The proposed GNN models are trained on a server with an NVIDIA A100 80G GPU. All the models are implemented using PyTorch Geometric[3] which is a well-known library and can be easily used to train graph neural networks for a wide range of applications. The source code of this paper can be found from here[4].

Default parameter setting. For the backbone index, 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 a sufficient number of edges are removed (Definition 4.8) is 0.01. More discussions about the effect of these parameters are in Section 7.2.5. For the GNN-based models, two GNN layers are included as shown in Figure 9. By default, B_e, B_{gnn}, and B_{fc} are set to be 128. The training of each GNN model uses 8100 instances in the training set and 900 instances in the validation set. Each model is trained with 100 epochs.

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 7.2.5 to fine-tune them. For the GNN-based methods, the default parameters can be used. Then, users can adjust them based on the effect of the GNN model parameters, which is analyzed in Section 7.3.2.

Data. Our experiments use *nine* real-world road networks, where six datasets including New York city (C9_NY), Bay Area (C9_BAY), Colorado (C9_COL), Florida (C9_FLA), Eastern USA (C9_E), and Central USA (C9_CTR),

[1]https://neo4j.com/
[2]https://db-engines.com/en/ranking
[3]https://pytorch-geometric.readthedocs.io/en/latest/index.html
[4]https://github.com/huipingcao/ACMTSAS2024



(a) RAC on **C9_NY_5K**
(b) RAC on **C9_NY_15K**
(c) Goodness on **C9_NY_5K**
(d) Goodness on **C9_NY_15K**

Fig. 11. Comparison of approximation quality

and Full USA (C9_USA) are from the 9th DIMACS Implementation Challenge[5] and three datasets including California (L_CAL), San Francisco (L_SF), and USA (L_NA) are from a real spatial benchmark database[6]. The details of these datasets are presented in Table 2. 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 [10, 33]. A detailed comparison of different ways to generate synthetic costs is in Section 7.2.7. 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 $\mathrm{RAC}_i = \frac{(\sum_{p'\in\mathbb{P}'} w'_i | w'_i \in cost(p'))/|\mathbb{P}'|}{(\sum_{p\in\mathbb{P}} w_i | w_i \in cost(p))/|\mathbb{P}|}$ 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. When the paths in the approximate solution set $\mathbb{P}'$ are longer than the exact paths, the RAC values are larger than 1. This is because a longer path generally has a larger per-dimension cost. On the other hand, when the paths $\mathbb{P}'$ are shorter than the exact paths, the RAC values are shorter than 1. The RAC value can measure the overall quality of all the paths in $\mathbb{P}'$.

(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: $goodness(\mathbb{P}') = \frac{\sum_{p\in\mathbb{P}}\{\mathrm{argmax} D_{p'\in\mathbb{P}'} sim(p,p')\}}{|\mathbb{P}|}$ 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 $sim(p, p')$. The definition of the goodness metric shows that it measures the *best* similarity of paths in the approximate answer set and the exact paths.

(3) Ratio of exact solutions. The ratio of exact solutions pertains to the possibility of returned skyline paths in the approximate solution set being exact skyline paths for a given SPQ. Given the exact solution set $\mathbb{P}$ and an approximate solution set $\mathbb{P}'$ for an SPQ, we define the ratio of exact skyline paths to measure the quantity of exact skyline paths present within the approximate solution set. $exact\_sp\_ratio(\mathbb{P}') = \frac{|\{p' | p'\in\mathbb{P}' \wedge \exists p\in\mathbb{P}\ s.t.\ p'=p\}|}{|\mathbb{P}'|}$. This ratio tells us how many exact skyline paths exist in an approximate skyline path set.

Exact method. We implement the SPQ method in [33] 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.

[5]http://users.diag.uniroma1.it/challenge9/download.shtml
[6]https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm

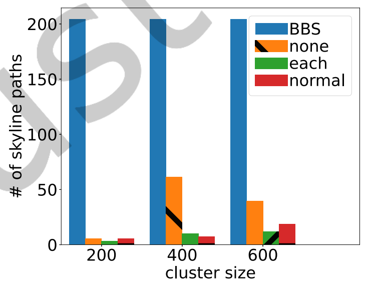
Comparison methods. Since no existing index structure is particularly designed to support SPQs, to demonstrate the effectiveness of our proposed index construction strategy **backbone_normal** (Algorithm 2), we modify two representative shortest path indexes, **GTree** [78] and **CH** [50], to compare 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.

7.2 Experimental results of SQP evaluation based on the backbone index

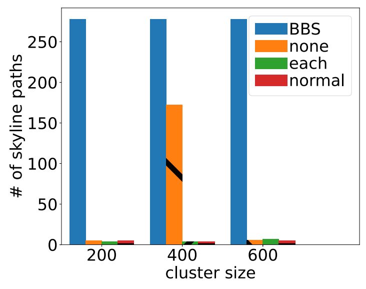
7.2.1 Effectiveness of the proposed index structure and query method. We compare the query results with the exact solutions returned by *BBS*. The *BBS* method does not work well on large graphs [19]. Thus, we use small subgraphs of C9_NY with 5K and 15K nodes. On both C9_NY_5K and C9_NY_15K, we randomly 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 11(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 11(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 many 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 of 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**



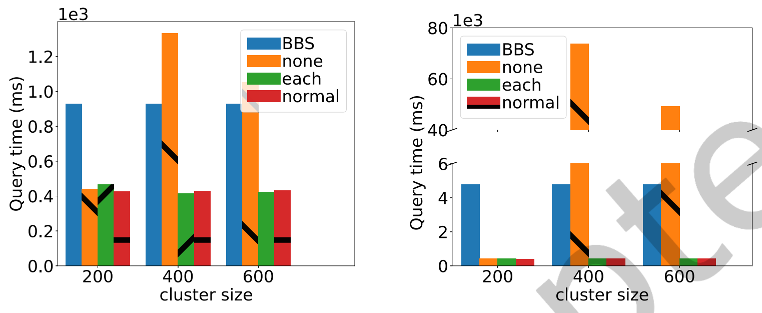
(b) **C9_NY_15K**

Fig. 12. Comparison of result set size (# of skyline paths)

Figures 12 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 13 reports the average 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 13).

We would like to report the ratio of exact skyline paths in the approximate solution set returned by our approach *backbone_normal* (with cluster size 200). For C9_NY_5K and C9_NY_15K, the ratio are 0.62 and 0.46 respectively. The larger graph has a lower ratio because the larger graph's index compresses more information (thus loses more data) in the abstracted graph when their levels are the same.

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**

Fig. 13. Comparison of query time

7.2.2 Efficiency of index construction. We conduct experiments to measure the index size and building time by comparing our index structure *backbone_normal* (Algorithm 2) 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 3.

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 3. Comparison of index construction

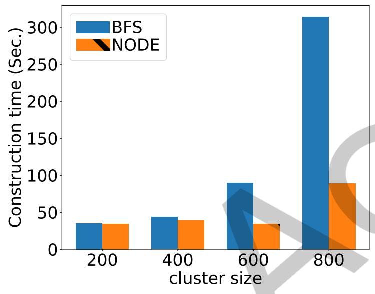
		C9_NY_5K	**C9_NY_10K**	**C9_NY_15K**
Construction time (sec.)	Backbone	99	251	216
	GTree	23,896 (6 hours)	-	39,781 (11 hours)
	CH	12,000	42,184	26,340
Index size (MB)	Backbone	27	89	68
	GTree	27.5	-	41.6
Size of the most abstracted graph	CH node #	4,071	9,654	13,499
	CH edge #	22,627	30,894	83,302

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.

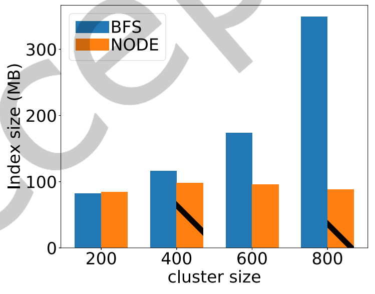
7.2.3 Effectiveness of using dense clusters to condense G_i. We evaluate the effectiveness of our approach of using dense clusters to condense G_i (Section 4.2). For comparison purpose, we implement another approach to partition the nodes in G_i to different connected components by using *BFS*. Other partition methods [26, 30] used in [28, 34, 38] 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 14 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 that does not consider graph density.



(a) Construction time



(b) Index size

Fig. 14. Effectiveness of cluster-based condensing

7.2.4 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 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 4.

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

Table 4. 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

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).

7.2.5 Effect of parameters. Figure 15 shows the impact of the parameters p and m_{max} on the performance of index construction.

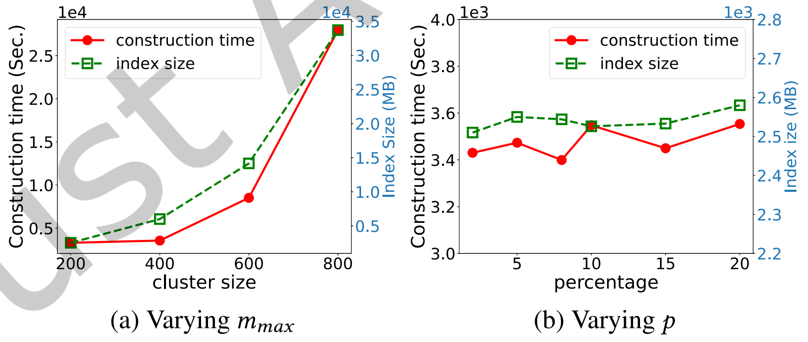


Fig. 15. Index building time and index size for **C9_NY**

The index construction process is sensitive to the cluster size as shown in Figure 15(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 15(b) because p only affects the levels of the indexes L, which are almost the same for different p values.

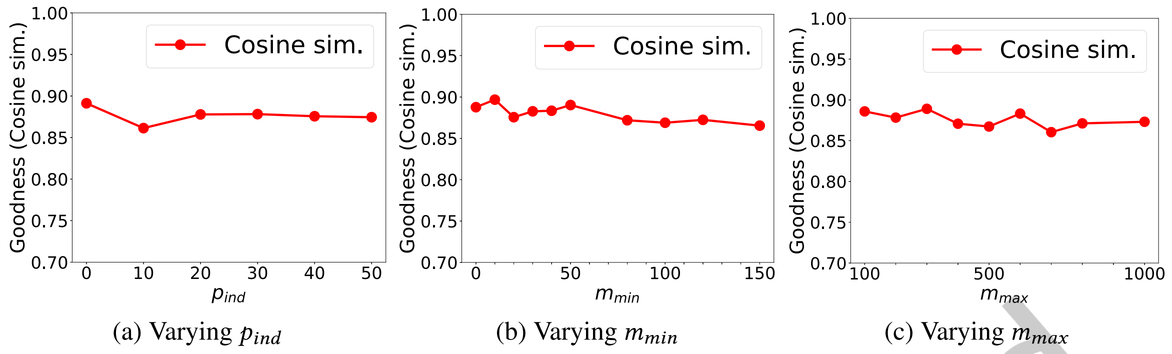


Fig. 16. Goodness comparison on **C9_NY_15K**

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 16 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* with fewer 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.

7.2.6 Performance on larger graphs. We apply our index construction approach to large real-world graphs. The results are shown in Table 5. The size of the highest graph row shows the number of nodes (top number)

Table 5. 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)						

and edges (bottom number) in the most abstracted graph G_L. Table 5(a) shows the results on the C9_* datasets that have higher node degrees. Table 5(b) shows the results on graphs with lower average node degrees. 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.

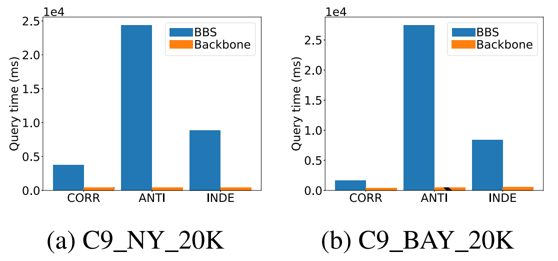


Fig. 17. Query time (different edge-cost distributions)

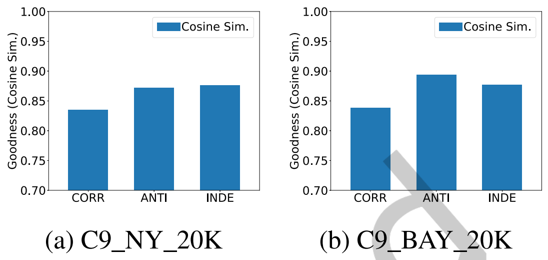


Fig. 18. Goodness scores (different edge-cost distributions)

7.2.7 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 costs 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 17. 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 17). Figure 18 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.

7.3 Experimental results of SPQ evaluation based on the GNN models

7.3.1 Effectiveness of SP-GNN and TSP-GNN.

This experiment examines the quality of the approximate results returned by the query methods on the backbone index, SP-GNN, and TSP-GNN by comparing them with the exact solutions returned by the baseline *BBS* method. Because the *BBS* method takes extremely long time [19] to get exact solutions, this experiment uses two small subgraphs of C9_NY with 5K and 15K nodes and only one original graph L_CAL (which is relatively larger). For the Backbone index, *backbone-normal* is used with the default parameter setting. For both the SP-GNN and TSP-GNN models, 9000 instances (default setting) are used to train the models where 10% (i.e., 900 instances) are used as the validation set. For the TSP-GNN model, these instances are generated using the backbone abstracts either in levels 4-7 (L4-7) or in levels 1-7 (L1-7), where the number of instances in each level is equal. The choice of utilizing TSP-GNN with both L4-7 and L1-7 abstractions is intentional. We apply these two settings for the L_CAL dataset, which is the smallest among the real graphs, aiming to understand the effectiveness of the transfer learning approach across different levels of abstraction. For larger real graphs (L_SF, C9_BAY, L_NA), a reasonable proportion of lower levels' backbone skyline results need to be in the training set to ensure that the model captures information from the original graph. We use the instances from levels 1-7 for these larger real graphs. We do not test the effect of SP-GNN on these large graphs because, in reality, it takes too long time (days to weeks) to generate the exact skyline paths for model training.

To get stable query results from the different methods we report the averaged query performance from 100 randomly generated queries (i.e., pairs of starting and ending nodes of the queries) on all these datasets.

Table 6. Comparing different methods using metrics RAC, Goodness, exact_sp_ratio (L4-7/L1-7: instances are generated using the backbone abstracts levels 4-7 or in levels 1-7)

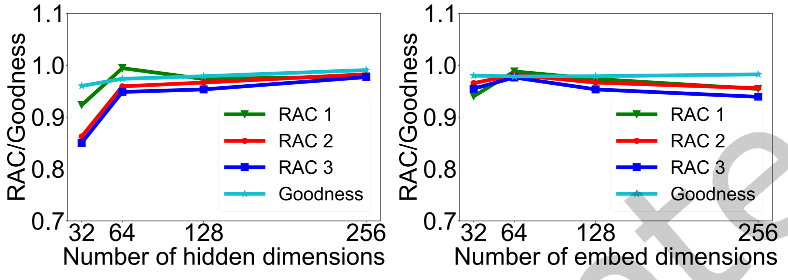
Dataset	method	RAC_1	RAC_2	RAC_3	average $\lvert RAC_i\text{-}1 \rvert$	Goodness	exact_sp_ratio
C9_NY_5K	Backbone	1.063	1.09	1.1071	0.09	0.93	0.62
	SP-GNN	0.9531	0.9453	0.9292	**0.06**	**0.98**	0.86
	TSP-GNN (L4-7)	0.9042	0.8379	0.8297	0.14	0.96	0.92
C9_NY_15K	Backbone	1.0544	1.0028	1.0219	0.03	**0.97**	0.46
	SP-GNN	1.0245	0.9667	0.9834	**0.02**	**0.97**	0.76
	TSP-GNN (L4-7)	0.9605	0.8529	0.8805	0.10	0.95	0.88
L_CAL	Backbone	1.0100	1.0391	1.0476	**0.03**	0.94	0.62
	SP-GNN	1.0787	1.0386	1.0364	0.05	**0.98**	0.87
	TSP-GNN (L4-7)	0.8955	0.8901	0.9015	0.10	**0.98**	0.86
	TSP-GNN (L1-7)	0.9480	0.9262	0.9357	0.06	0.97	0.87
L_SF	Backbone	1.1048	1.0778	1.0772	0.09	0.97	0.49
	TSP-GNN (L1-7)	0.9249	0.8266	0.8352	0.14	0.95	0.97
C9_BAY	Backbone	1.1477	1.0827	1.0856	0.11	0.97	0.49
	TSP-GNN (L1-7)	1.0035	0.9111	0.9221	0.06	0.95	0.94
L_NA	Backbone	1.0201	0.9977	0.9952	0.01	0.95	0.64
	TSP-GNN (L1-7)	1.0626	1.0718	1.0761	0.07	0.95	0.99

Table 6 shows the RAC values, the goodness scores, and the ratio of the exact skyline paths of the approximate results from the different methods on the three datasets. Note that when a RAC value is closer to one, it is better. Thus, we also report the average $\lvert RAC_i\text{-}1 \rvert$ values in the second to the last column. The results show that SP-GNN can get the best approximate RAC results (smallest $\lvert RAC_i\text{-}1 \rvert$ values) for C9_NY_5K and C9_NY_15K. This is because SP-GNN is directly trained using the exact solutions from the original graph. However, for L_CAL, the RAC results from SP-GNN are not as good as the results calculated from the backbone index. This is because L_CAL is a larger graph, which requires more training instances. The approximate results from TSP-GNN(L4-7) are the worst on the RAC metric. This is consistent with the design that TSP-GNN utilizes many approximate solutions from the higher-level backbone abstracts to train the model. For large graphs (L_SF, L_NA, and C9_BAY), there is no deterministic superior performer between the Backbone and the TSP-GNN method because multiple factors affect the results including the number and the length of the returned approximate paths. On the L_SF graph, TSP-GNN's worse performance is directly related to its less number of returned paths (3.16 on average), compared with the slightly more returned number of paths (3.94 and 4.14 on average) for the C9_BAY and the L_NA datasets.

Regarding the Goodness metric, SP-GNN achieves the best performance. TSP-GNN model and the Backbone index have similar results. This is because the Goodness metric measures the largest similarity of the exact paths and the approximate paths. The close-to-1 goodness score shows that paths from the TSP-GNN method can be very close to the exact paths although on average they are not as good as the paths returned by the backbone method.

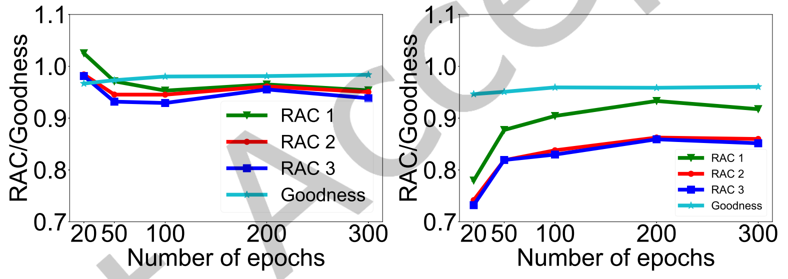
For the exact_sp_ratio, surprisingly, the results of Backbone method are worse than those from the TSP-GNN and SP-GNN. This is because Backbone results are generally approximate when it involves higher level index structure. SP-GNN utilizes exact skyline paths as training instances, thus their results are better than those from the Backbone index. The results from TSP-GNN and SP-GNN are similar on L_CAL. It indicates that the TSP-GNN model is able to catch the skyline path representations as that of the SP-GNN model. On small graphs (C9_NY_5K and C9_NY_15K), the ratio of the exact skyline paths is higher because fewer paths are returned from TSP-GNN than from SP-GNN.

7.3.2 Sensitivity test when varying model training parameters. This experiment tests how the results' quality is affected by the different model parameters using the C9_NY_5K dataset. We investigate the effect of the number of hidden dimensions B_{gnn} and the number of embedding dimensions B_e. Figure 20(a) shows the results for varying the number of hidden dimensions (from 32 to 256) while fixing the embedding dimension number to be the default value of 128. It shows that when increasing the number of hidden dimensions, the performance improves gradually. However, the improvement is not much when it reaches 128 dimensions.



(a) Vary # of hidden dimensions (b) Vary # of embedding dimensions

Fig. 19. RAC and goodness vs. model hidden dimensions (SP-GNN)



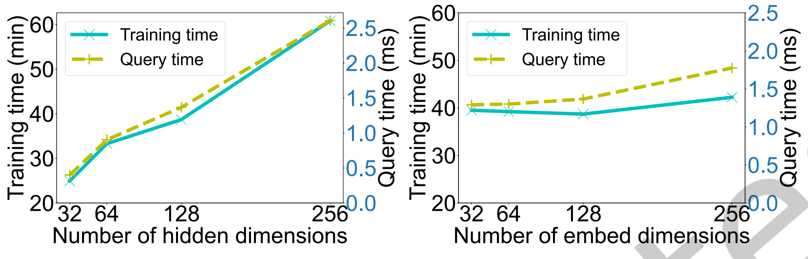
(a) SP-GNN (b) TSP-GNN

Fig. 20. RAC and goodness vs. # of epochs

We also vary the number of embedding dimensions when fix the number of hidden dimensions (default value 128). Fig. 20(b) reports the results. It shows that increasing the embedding dimensions may not increase the performance. Our analysis is that the graph's overall node degree is low (round 4), thus embedding too many dimensions may not be helpful.

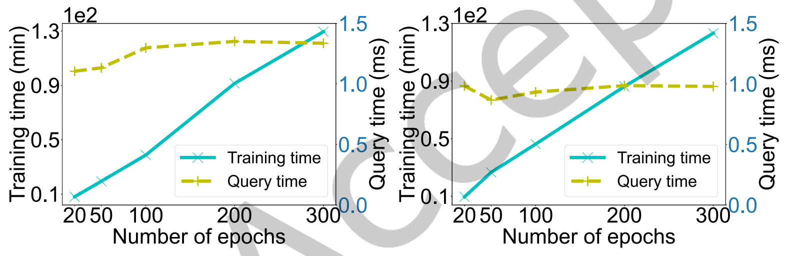
The number of training epochs also affects the performance of a model. We report the RAC values and the goodness scores for both the SP-GNN and TSP-GNN models when changing the number of epochs. Figure 20 shows that, when the number of epochs increases, the quality of the query results generally improves (except epoch number 20 for SP-GNN). The quality does not increase much after 100 epochs. It means that we do not need to train the GNN models with a huge number of iterations when we are good with the approximation quality. The abnormal behavior is observed from the SP-GNN model when the epoch number is 20. Its RAC and goodness scores are even better than those for epoch number 50. This is because the model can only support answering a much small number of queries (74 out of 100 queries) when the epoch number is 20. These returned results still show good approximation quality.

7.3.3 GNN model construction time and query time. The construction of a GNN model requires generating the training instances and training a model. We do not include the time for training instance generation because the GNN models are expected to be built upon historical data (queries and query results) that are accumulated. We also note that, when historical data is not available, generating instances for large graphs may become super time consuming [19] (e.g., utilizing several days) because it depends on the inefficient *BBS* method to find the exact solution. The training time is largely affected by the model parameters (embedding dimensions, hidden dimensions) and the number of training epochs. This section looks into the training time and query time.



(a) Vary # of hidden dimensions (b) Vary # of embedding dimensions

Fig. 21. Time vs. model parameters



(a) SP-GNN (b) TSP-GNN

Fig. 22. Time vs. # of epochs

We plot the training time and the query time when varying the model parameters. Figure 21(a) and (b) show the training time for 100 epochs and the query time per query when varying the number of hidden dimensions (while fixing the embedding dimension to be 128) and changing the number of embedding dimensions (when fixing the number of hidden dimensions to be 128). The results show that both the training time (in minutes) and the query time (in milliseconds) grow linearly to the increase of these hidden dimensions. This is better than the worst case analysis result. Increasing the number of training epochs also linearly increases the training time, but not the query time, as shown in Figure 22. The query time is almost constant and negligible (several milliseconds) to answer a query.

We also report the training time of both the SP-GNN and TSP-GNN models for the default number of instances (i.e., 9000) and the default number of epochs (i.e., 100). The time is reported in Table 7. The training time for each instance and every epoch is similar. Thus, the reported training time (from the 2nd column) divided by 9000 represents the time required to process one query result, i.e., one training instance.

7.4 Effect of data augmentation

This section examines the effect of the amounts of augmented data on the quality of query results. We run the TSP-GNN model utilizing the instances from the L_CAL dataset. While fixing the total number of training instances

Table 7. Comparing training time for different datasets (L4-7/L1-7: instances are generated using the backbone abstracts levels 4-7 or in levels 1-7)

Dataset	training time (min.)
C9_NY_5K (SP-GNN)	38.698
C9_NY_5K (TSP-GNN (L4-7))	46.203
C9_NY_15K (SP-GNN)	109.391
C9_NY_15K (TSP-GNN (L4-7))	112.321
L_CAL (SP-GNN)	141.267
L_CAL (TSP-GNN (L4-7))	142.044
L_CAL (TSP-GNN (L1-7))	154.510
L_SF (TSP-GNN (L1-7))	2793.849
L_NA (TSP-GNN (L1-7))	2593.774
C9_BAY (TSP-GNN (L1-7))	5707.043

to be 9K (the default setting), we vary the proportion of the exact skyline-path instances from 30% to 70%, while the corresponding augmented skyline-path instances are from 70% to 30%. Table 8 shows the results.

Table 8. Comparison of different augmentation ratios in TSP-GNN on graph L_CAL

augmentation percentage	average $\lvert RAC_i\text{-}1\rvert$	Goodness	exact_sp_ratio	avg # of paths in result set
0	0.06	0.97	0.87	8.73
30	0.11	0.97	0.88	6.18
40	0.10	0.97	0.89	6.25
50	0.08	0.97	0.89	7.42
60	0.11	0.97	0.90	6.14
70	0.08	0.97	0.89	7.95
80	0.07	0.97	0.89	9.06
90	0.08	0.97	0.90	7.75

The RAC results show variations for different augmentation percentages, which is introduced by the augmentation approach. We also observe that there is no clear trend with the RAC values as the augmentation ratio varies. This can be attributed to the intricate relationship between this metric and the average number of paths in the result set. When there are more paths in the result set (last column), there is less variability in the paths, the $\lvert RAC_i\text{-}1\rvert$ is smaller.

The Goodness scores across different augmentation percentages are the same. This is because it measures the highest similarity between exact solutions and approximate result sets. This similarity in exct_sp_ratio is also observed across all augmentation percentages, affirming the model's ability to return skyline paths that align closely with the exact solutions.

Despite the inherent randomness in the augmentation process, the overall performance of TSP-GNN remains consistently high across varying augmentation ratios. This study showcases the models' competence even when a smaller portion of the original training set is utilized. The use of naive augmentation strategies proves effective in mitigating the need for an exhaustive generation of exact skyline paths for training. It is possible to design advanced techniques to augment the training set, which is not the main focus of this work.

7.5 Effect of alternative GNN models

This section examines the effect of different graph operators over the query results and the training/inference efficiency. Besides utilizing the graph Transformer model [54] to implement both SP-GNN and TSP-GNN, we have implemented two other recent graph neural network models, LightGCN [25] and ClusterGCN [12]. The results are reported in Table 9.

All three GNN models show similar results on the query results' quality measured Goodness scores and exact_sp_ratio. For the RAC value, LightGCN model returns slightly better results thanks to its complex operators, which also make it suffer from longer training time. The difference of model inference to find the query search space is negligible.

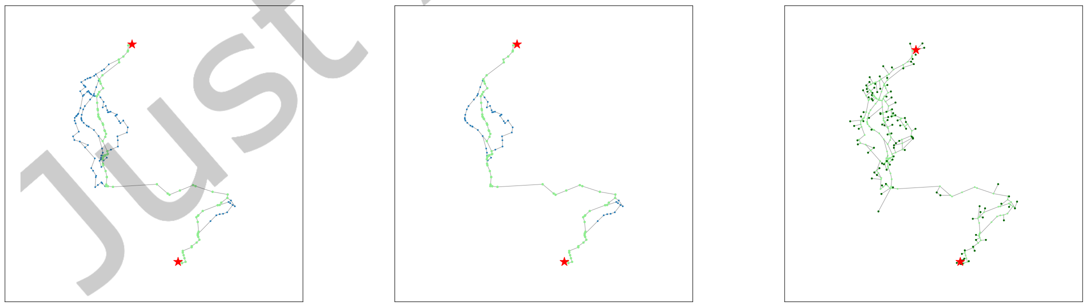
Our findings suggest that different GNN architectures can be used in SP-GNN or TSP-GNN. When more efficient GNN architectures are available, they can be adopted to improve training efficiency while retaining reasonable performance. Note that developing more efficient GNN models is not the focus of this paper.

Table 9. Comparison of different graph operators in TSP-GNN

GNN model	RAC_1	RAC_2,	RAC_3	average $\|RAC_i\text{-}1\|$	Goodness	exact_sp_ratio	Time (min) Training	Time (sec) Inference
Transformer	0.948	0.9262	0.9357	0.06	0.97	0.87	151	0.062
ClusterGCN	0.9641	0.937	0.9491	0.05	0.97	0.87	107	0.059
LightGCN	0.9635	0.9408	0.9509	0.05	0.97	0.86	165	0.056

7.6 Use case study

We conduct a case study to provide a more in-depth understanding of the approximate skyline paths returned by TSP-GNN. The approximate results returned by Backbone and SP-GNN methods show similar characteristics, thus are not included here. On two real graphs (C9_NY_5K and L_CAL), we randomly pick two queries and plot the exact solutions, the approximate solutions, and their training instances in Figs. 23 and 24. From Fig. 23 (a-b) and Fig. 24(a-b), we can see that there are much more exact skyline paths than approximate paths. Accordingly, the total number of nodes on the exact skyline paths are more than that in the approximate solutions. However, the approximate paths still look very similar to the exact paths. The smaller number of approximate skyline paths can give users a reasonable choice of paths to choose from. We also show the training instances for these two queries in Fig. 23(c) and Fig. 24(c). The instances consist of type-1 (highlighted in green) and type-2 (highlighted in dark green) nodes, which together form the search space to return the approximate paths.

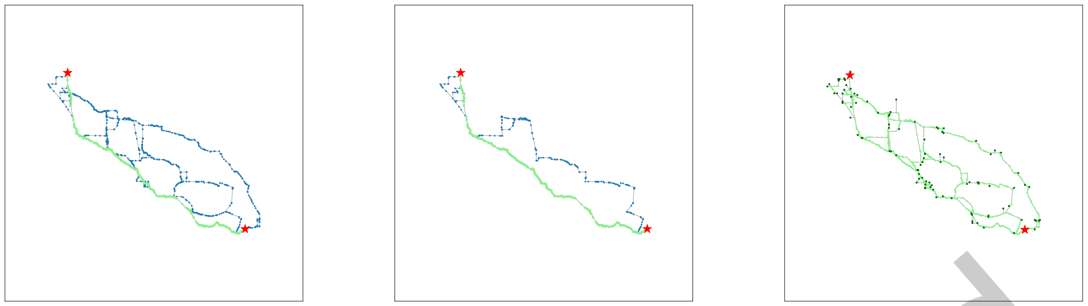


(a) exact solution (24 paths) one path is highlighted in green

(b) approximate solution returned by TSP-GNN (3 paths) one path is highlighted in green

(c) the training instance

Fig. 23. Skyline path results on C9_NY_5K from source node 56 to destination node 2830



(a) exact solution (193 paths) one path is highlighted in green

(b) approximate solution returned by TSP-GNN (4 paths) one path is highlighted in green

(c) the training instance

Fig. 24. Skyline path results on L_CAL from source node 19426 to destination node 13307

8 DISCUSSIONS

This section provides a summarized discussion about the advantages, the shortcomings, and the use scenarios of the backbone index and the two GNN models.

Table 10. Summary of the performance and the overhead of three approaches

	Backbone	SP-GNN	TSP-GNN
Result quality	Second best on RAC; Worst on Goodness; Worse on exact_sp_ratio	Best on RAC and Goodness; Second best on exact_sp_ratio	Second best on Goodness; Worst on RAC; Best on exact_sp_ratio
Query time	hundreds of ms	tens of ms	tens of ms
Overhead	(1) build backbone index	(1) build landmark index over the original graph, (2) run *BBS* alg. using landmark index & data augmentation, and (3) train GNN models	(1) build backbone index, (2) build landmark index over backbone abstracts, (3) run *BBS* alg. over backbone & data augmentation, and (4) train GNN models

Table 10 summarizes the result quality with respect to RAC, goodness, and exact_sp_ratio scores, query time, and the overhead of the three methods. When historical query results are available, which means that there is no overhead for building the landmark index and running *BBS* method, the only overhead for SP-GNN is model training, SP-GNN is the best choice for searching over small graphs. For large graphs, the backbone index needs to be utilized. It can either directly support queries or support query evaluation through the use of the TSP-GNN.

9 CONCLUSIONS AND FUTURE WORK

This paper introduces a new index structure (denoted as Backbone index) and two GNN models (SP-GNN and TSP-GNN) 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. We further present two GNN-based models that can be trained utilizing historical query results to support more efficient query processing. Extensive experiments are conducted on multiple 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 a more efficient index size and building time than two other index structures adopted from the shortest-path-query supporting indexes.

This work assumes that all the graph edge weights are static. However, in many applications, parameters such as traffic flow/density vary over time and are measured by road-side sensors. Query processing over graphs with dynamic edge weights will be explored by leveraging temporal features as future work.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support from the National Science Foundation (NSF) under Awards No. 1914635 and 2151254, and from the United States Department of Agriculture - National Institute of Food and Agriculture (USDA-NIFA) under Award No. 20196901229853. These funds have been instrumental in advancing the research presented in this work.

REFERENCES

[1] Ralph Abboud, Radoslav Dimitrov, and Ismail Ilkan Ceylan. 2022. Shortest path networks for graph property prediction. In *Learning on Graphs Conference*. PMLR, 5–1.

[2] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. 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*. 349–360.

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

[4] G Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. 2009. Time-dependent contraction hierarchies. In *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 97–105.

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

[6] G. Borruso. 2008. Network Density Estimation: A GIS Approach for Analysing Point Patterns in a Network Space. *Trans. GIS* 12 (2008), 377–402.

[7] Zhan Bu, Zhiang Wu, Liqiang Qian, Jie Cao, and Guandong Xu. 2014. 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)* (2014), 85–88.

[8] Yanchuan Chang, Egemen Tanin, Xin Cao, and Jianzhong Qi. 2023. Spatial Structure-Aware Road Network Embedding via Graph Contrastive Learning. In *EDBT*. OpenProceedings.org, 144–156.

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

[10] Yi-Chung Chen and Chiang Lee. 2016. Skyline Path Queries With Aggregate Attributes. *IEEE Access* 4 (2016), 4690–4706.

[11] Zitong Chen, A. Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. 2021. P2H: Efficient Distance Querying on Road Networks by Projected Vertex Separators. *Proceedings of the 2021 International Conference on Management of Data* (2021).

[12] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 257–266. https://doi.org/10.1145/3292500.3330925

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

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

[15] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD Conference*. ACM, 1241–1258.

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

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

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

[19] Qixu Gong and Huiping Cao. [n. d.]. 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. 2019. Skyline Queries Constrained by Multi-cost Transportation Networks. *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), 926–937.

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

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

[23] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.* 4 (1968), 100–107.

[24] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *SIGMOD Conference*. ACM, 1035–1050.

[25] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yong-Dong Zhang, and Meng Wang. 2020. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. In *Proceedings of International ACM SIGIR conference on research and development in Information Retrieval, SIGIR*, Jimmy X. Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen, and Yiqun Liu (Eds.). ACM, 639–648. https://doi.org/10.1145/3397271.3401063

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

[27] Shalev Itzkovitz, Reuven Levitt, Nadav Kashtan, Ron Milo, Michael Itzkovitz, and Uri Alon. 2005. Coarse-graining and self-dissimilarity of complex networks. *Phys. Rev. E* 71 (Jan 2005), 016127. Issue 1. https://doi.org/10.1103/PhysRevE.71.016127

[28] N. Jing, Y. Huang, and Elke A. Rundensteiner. 1998. Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation. *IEEE Trans. Knowl. Data Eng.* 10 (1998), 409–432.

[29] Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Bertty Contreras-Rojas, Rodrigo Pardo-Meza, Anis Troudi, and Sanjay Chawla. 2020. ML-based Cross-Platform Query Optimization. In *ICDE*. IEEE, 1489–1500.

[30] G. Karypis and V. Kumar. 1998. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distributed Comput.* 48 (1998), 96–129.

[31] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. 2010. Distributed time-dependent contraction hierarchies. In *Experimental Algorithms: 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings 9*. Springer, 83–93.

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

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

[34] K. Lee, W. Lee, B. Zheng, and Yuan Tian. 2012. ROAD: A New Spatial Object Search Framework for Road Networks. *IEEE Transactions on Knowledge and Data Engineering* 24 (2012), 547–560.

[35] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. Machine Learning for Databases. *Proc. VLDB Endow.* 14, 12 (2021), 3190–3193.

[36] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A distribution-driven learned index. *arXiv preprint arXiv:2304.08817* (2023).

[37] Qiyan Li, Yuanyuan Zhu, and J. X. Yu. 2020. Skyline Cohesive Group Queries in Large Road-social Networks. *2020 IEEE 36th International Conference on Data Engineering (ICDE)* (2020), 397–408.

[38] Zijian Li, Lei Chen, and Yue Wang. 2019. G*-Tree: An Efficient Spatial Index on Road Networks. *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), 268–279.

[39] Yiding Liu, Kaiqi Zhao, Gao Cong, and Zhifeng Bao. 2020. Online Anomalous Trajectory Detection with Deep Generative Sequence Modeling. In *ICDE*. IEEE, 949–960.

[40] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2023. Multi-constraint shortest path using forest hop labeling. *The VLDB Journal* 32, 3 (2023), 595–621.

[41] Qingzhi Ma and Peter Triantafillou. 2019. DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models. In *SIGMOD Conference*. ACM, 1553–1570.

[42] A. Maratea, A. Petrosino, and Mario Manzo. 2017. Extended Graph Backbone for Motif Analysis. *Proceedings of the 18th International Conference on Computer Systems and Technologies* (2017).

[43] Sunil Nishad, Shubhangi Agarwal, Arnab Bhattacharya, and Sayan Ranu. 2020. GraphReach: Position-aware graph neural network using reachability estimations. *arXiv preprint arXiv:2008.09657* (2020).

[44] D. Orellana and M. Guerrero. 2019. 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 (2019), 1314 – 1330.
[45] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Y. Zhang, and Xuemin Lin. 2020. Progressive Top-K Nearest Neighbors Search in Large Road Networks. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).
[46] Huiping Cao Qixu Gong and Parth Nagarkar. 2018. *Skyline Queries Constrained by Multi-Cost Transportation Networks*. Technical Report TR-CS-NMSU-2018-09-02. Department of Computer Science, New Mexico State University, Las Cruces, New Mexico. https://www.cs.nmsu.edu/wp/wp-content/uploads/2018/09/constrainedSkyline_TR.pdf
[47] Michael N Rice and Vassilis J Tsotras. 2012. Exact graph search algorithms for generalized traveling salesman path problems. In *International Symposium on Experimental Algorithms*. Springer, 344–355.
[48] Ning Ruan, Ruoming Jin, Guan Wang, and Kun Huang. 2012. Network backbone discovery using edge clustering. *arXiv preprint arXiv:1202.1842* (2012).
[49] Ibrahim Sabek and Mohamed F. Mokbel. 2020. Machine Learning Meets Big Spatial Data. In *ICDE*. IEEE, 1782–1785.
[50] Peter Sanders and Dominik Schultes. 2005. Highway Hierarchies Hasten Exact Shortest Path Queries. In *ESA*.
[51] M Ángeles Serrano, Marián Boguná, and Alessandro Vespignani. 2009. Extracting the multiscale backbone of complex weighted networks. *Proceedings of the national academy of sciences* 106, 16 (2009), 6483–6488.
[52] Michael Shekelyan, Gregor Jossé, and Matthias Schubert. 2015. Linear path skylines in multicriteria networks. In *ICDE*. IEEE, 459–470.
[53] Jiachen Shi, Gao Cong, and Xiaoli Li. 2022. Learned Index Benefits: Machine Learning Based Index Performance Estimation. *Proc. VLDB Endow.* 15, 13 (2022), 3950–3962.
[54] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjing Wang, and Yu Sun. 2021. Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification. In *Proceedings of International Joint Conference on Artificial Intelligence, IJCAI 2021*, Zhi-Hua Zhou (Ed.). ijcai.org, 1548–1554. https://doi.org/10.24963/IJCAI.2021/214
[55] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned index: A comprehensive experimental evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.
[56] Yuan Tian, K. Lee, and W. Lee. 2009. Finding skyline paths in road networks. In *GIS '09*.
[57] Yao Tian, Tingyun Yan, Xi Zhao, Kai Huang, and Xiaofang Zhou. 2022. A learned index for exact similarity search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering* (2022).
[58] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD Conference*. ACM, 1153–1170.
[59] Ulrike von Luxburg, Agnes Radl, and Matthias Hein. 2010. Hitting and commute times in large graphs are often misleading. *arXiv preprint arXiv:1003.1266*.
[60] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. 2005. Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics (JEA)* 10 (2005), 1–3.
[61] T. Wang, C. Ren, Y. Luo, and J. Tian. 2019. NS-DBSCAN: A Density-Based Clustering Algorithm in Network Space. *ISPRS Int. J. Geo Inf.* 8 (2019), 218.
[62] Yishu Wang, Ye Yuan, Hao Wang, Xiangmin Zhou, Congcong Mu, and Guoren Wang. 2021. Constrained Route Planning over Large Multi-Modal Time-Dependent Networks. In *ICDE*. IEEE, 313–324.
[63] Zheng Wang, Cheng Long, and Gao Cong. 2021. Trajectory Simplification with Reinforcement Learning. In *ICDE*. IEEE, 684–695.
[64] Zheng Wang, Cheng Long, Gao Cong, and Yiding Liu. 2020. Efficient and Effective Similar Subtrajectory Search with Deep Reinforcement Learning. *Proc. VLDB Endow.* 13, 11 (2020), 2312–2325.
[65] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.
[66] Victor Junqiu Wei, R. C. Wong, and Cheng Long. 2020. 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).
[67] Bin Xu, Jun Feng, and Jiamin Lu. 2018. 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* (Langkawi, Malaysia) *(IMCOM)*. ACM, Article 53, 6 pages. https://doi.org/10.1145/3164541.3164634
[68] Bin Yang, Chenjuan Guo, Christian S. Jensen, Manohar Kaul, and Shuo Shang. 2013. Multi-Cost Optimal Route Planning Under Time-Varying Uncertainty.
[69] Bin Yang, Chenjuan Guo, Christian S. Jensen, Manohar Kaul, and Shuo Shang. 2014. Stochastic skyline route planning under time-varying uncertainty. *2014 IEEE 30th International Conference on Data Engineering (ICDE)* (2014), 136–147.
[70] Peilun Yang, Hanchen Wang, Defu Lian, Ying Zhang, Lu Qin, and Wenjie Zhang. 2022. TMN: Trajectory Matching Networks for Predicting Similarity. In *ICDE*. IEEE, 1700–1713.
[71] Yajun Yang, Zhongfei Li, Xin Wang, Qinghua Hu, et al. 2019. Finding the shortest path with vertex constraint over large graphs. *Complexity* 2019 (2019).
[72] Yiding Yang, Xinchao Wang, Mingli Song, Junsong Yuan, and Dacheng Tao. 2021. SPAGAN: Shortest path graph attention network. *arXiv preprint arXiv:2101.03464* (2021).

[73] Yajun Yang, Hang Zhang, Hong Gao, Qing hua Hu, and Xin Wang. 2020. An Efficient Index Method for the Optimal Route Query over Multi-Cost Networks. *ArXiv* abs/2004.12424 (2020).
[74] Man Lung Yiu and N. Mamoulis. 2004. Clustering objects on a spatial network. In *SIGMOD '04*.
[75] Juxiang Zeng, Pinghui Wang, Lin Lan, Junzhou Zhao, Feiyang Sun, Jing Tao, Junlan Feng, Min Hu, and Xiaohong Guan. 2022. Accurate and Scalable Graph Neural Networks for Billion-Scale Graphs. In *ICDE*. IEEE, 110–122.
[76] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic Hub Labeling for Road Networks. *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), 336–347.
[77] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2022. Efficient learned spatial index with interpolation function based learned model. *IEEE Transactions on Big Data* 9, 2 (2022), 733–745.
[78] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. 2013. G-tree: An efficient index for knn search on road networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 39–48.

APPENDIX - DETAILED ANALYSIS OF FACTORS AFFECTING SPQ EVALUATION

To get a better understanding of the factors that affect the performance of Skyline Path Query (SPQ) evaluation, we conduct a detailed analysis of the relationships between SPQ answers and the characteristics of the underlying graphs using two real-world road networks, California (L_CAL) with 21,048 nodes and 21,693 edges and a subgraph of the road network in New York City, (C9_NY_22K) containing 22,000 nodes and 30,900 edges. The L_CAL and the C9_NY_22K datasets are used in previous works [33] and [38, 78] respectively. These two networks have a similar size from the perspective of the number of nodes and edges. There is only one cost dimension for the edges in the original graphs. We generate two more cost dimensions for each edge by letting the cost follow a uniform distribution in the range of [1,100], which is a commonly used strategy when the evaluation is conducted on multi-cost road networks [10, 20, 33]. In total, three cost weights are associated with each edge. The detailed node degree distributions of the two networks are listed in Table 11.

Table 11. Node-degree distribution of two graphs

L_CAL		C9_NY_22K	
Dregree	**# of nodes**	**Dregree**	**# of nodes**
1	182	1	2,836
2	**19,683**	2	3,611
3	915	3	**10,595**
4	255	4	4,843
5	7	5	105
6	5	6	10
8	1	-	-

With these two similar size graphs, we conduct a detailed analysis to understand how a SPQ algorithm performances differently by running the *BBS* method. We randomly generate 300 queries on each road network. We summarize basic statistics of the query results and show them in Table 12.

Table 12. SPQ query performance comparison

	L_CAL	**C9_NY_22K**
avg. query time (ms)	2,967	68,056
avg. # of hops of the shortest path	287	92
avg. # of skyline path results	82	1,097
avg. coverage	26%	24%
avg. # of nodes in the results	3%	1.5%
# of unfinished queries in 30 min.	0	42

Despite that the two road networks have similar numbers of nodes and edges, the results show a vast difference in query performance. The query time on C9_NY_22K is almost 23 times more than that for queries on L_CAL. The number of the skyline paths returned from C9_NY_22K is 10 times more than that from L_CAL. Even worse, 42 queries cannot finish in half an hour on the C9_NY_22K graph. A major reason behind this performance difference is that the degree distribution is different. Most of the nodes have degree 2 on the L_CAL, but 3 on the C9_NY_22K dataset.

We further analyze the average number of nodes that are visited during the query process (*avg. coverage*) and the number of distinct nodes showing in the returned paths. The results on these two characteristics are similar. These results indicate that the query process has similar exploration behavior, and the returned paths consist of approximately the same sets of nodes. However, when more nodes have a higher degree (even one more), there are more ways that paths can be constructed because more edges can be selected when a candidate path reaches a high-degree node. It means the probability that other paths do not dominate the candidate path is high when the node's degree is high.

The average length of the shortest paths on all the dimensions is 287 on the L_CAL, which is much higher than 92 on the C9_NY_22K graph. Interestingly, we note that the number of the skylines paths found on L_CAL is 82, which is much smaller than 1097 on C9_NY_22K. It further confirms that more skyline paths are found when the nodes have a higher degree, despite that these skyline paths have fewer hops. This detailed analysis shows that the performance of an SPQ algorithm is highly sensitive to the degree distribution of a network.

Table 13. Analysis of running *BBS* algorithm on C9_NY_22K

# of hops of shortest paths	**Query time (ms)**	**# of skyline paths**	**coverage (%)**	**# of unfinished queries**
10	4.25	12.88	0.57	-
20	17.69	38.31	1.76	-
30	116.00	64.07	3.37	-
40	455.23	202.59	5.11	-
50	640.65	202.83	8.69	-
60	2459.07	415.63	11.61	-
70	6544.39	613.65	19.28	-
80	11825.90	529.80	23.32	-
90	31577.19	1039.26	29.01	-
100	77220.57	1641.71	36.64	-
110	132296.29	1750.67	46.17	-
120	189357.00	2551.67	46.19	1
130	395374.64	3764.79	59.24	3
140	118297.20	1574.20	48.92	7
150	353904.25	5249.13	62.89	5

We further examine the effect of path hops on the performance of SPQ evaluations by running the *BBS* method over C9_NY_22K. The results are reported in Table 13. The results show that the number of path hops profoundly influences the number of results and the query time. Even when the number of hops increases with 10 more hops, the query time and the size of the result set can be doubled. For example, when the number of hops increases from 50 to 60, the result size and the query time are increased from 202.83 skyline paths and 640.64 ms to 415.63 paths and 2459.07 ms (on average) respectively. The number of queries that cannot finish in 30 minutes increases when the number of hops reaches 140. The query time and the number of skyline paths have a sudden drop at 140 and jump back at 150. This is due to the increasing number of unfinished queries that lead to more missing results.

We also examine the effect of the node coverage on the query performance. Although *BBS* uses strategies and auxiliary structures to reduce the search space [33], its exploration space can still reach up to more than half of the network nodes. When the number of hops reaches 150, the nodes that are visited during the query process are more than 60% of the nodes in the network. These statistics show that the number of path hops plays a critical role in SPQ evaluation. It further confirms the difficulty of improving SPQ algorithms due to high node coverage.