An Experimental Evaluation and Guideline for Path Finding in Weighted Dynamic Network

Mengxuan Zhang
Lei Li
The University of Queensland, Australia
{mengxuan.zhang,l.li3}@uq.edu.au

Xiaofang Zhou
The Hong Kong University of Science and Technology, Hong Kong
zxf@cse.ust.hk

ABSTRACT

Shortest path computation is a building block of various network applications. Since real-life networks evolve as time passes, the *Dynamic Shortest Path (DSP)* problem has drawn lots of attention in recent years. However, as *DSP* has many factors related to network topology, update patterns, and query characteristics, existing works only test their algorithms on limited situations without sufficient comparisons with other approaches. Thus, it is still hard to choose the most suitable method in practice. To this end, we first identify the *determinant dimensions* and *constraint dimensions* of the *DSP* problem and create a complete problem space to cover all possible situations. Then we evaluate the state-of-the-art *DSP* methods under the same implementation standard and test them systematically under a set of synthetic dynamic networks. Furthermore, we propose the concept of *dynamic degree* to classify the dynamic environments and use *throughput* to evaluate their performance. These results can serve as a guideline to find the best solution for each situation during system implementation and also identify research opportunities. Finally, we validate our findings on real-life dynamic networks.

PVLDB Reference Format:
Mengxuan Zhang, Lei Li, and Xiaofang Zhou. An Experimental Evaluation and Guideline for Path Finding in Weighted Dynamic Network. PVLDB, 14(11): 2127-2140, 2021.
doi:10.14778/3476249.3476267

1 INTRODUCTION

The *Shortest Path (SP)* query is a fundamental operation on various network related applications such as *route planning* in road networks [5], *influential community search* and *privacy protection* in social networks [27], *link analysis* in web graphs and more. Given an *Origin and Destination (OD) pair*, a *SP* returns the path of minimum cost between them, where the cost can be distance, travel time, closeness, proximity, similarity, etc. Figure 1-(a) demonstrates an example of *SP* from A to C. *SP* is the building blocking of many other operations like *k Nearest Neighbors (kNN)* [66], *Top-k Shortest Path (KSP)* [59], *Shortest Path Counting (SPC)* [42], *Constraint Shortest Path (CSP)* [35] and so on. As a result of these applications, *SP* has been extensively studied [1, 2, 6, 11, 14, 16–19, 21–24, 37, 38, 43, 45–48, 51, 54, 60, 67].

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476267

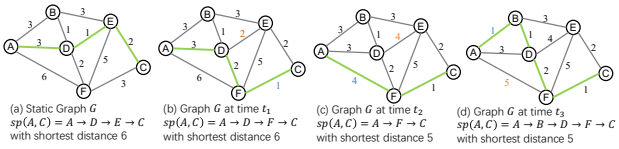


Figure 1: Example of *SP* and *DSP*

Real-life networks often evolve over time in terms of both topology and weight. For example, the proximity between two users may change in social networks because of their activities (likes, followers, tags, reposts, etc.), and the travel time changes in road networks because of traffic volumes or road construction. Therefore, *Dynamic Shortest Path (DSP)* computation is of great importance in practical applications. Figure 1-(b) to (d) shows an example of *DSP*. Although it is a special case of *SP*, it gives an example that many existing *SP* algorithms are hard to adapt to dynamic situations. Due to its significant practical value, *DSP* has been actively investigated recently. We classify the corresponding algorithms into four categories: 1) *Direct Search* methods, such as A^* [21] and *Dijkstra's* [11, 56], which calculate the shortest path by traversing graphs directly in a *Best-First Search (BFS)* manner. Their independence from using auxiliary information endows them with flexible adaption to dynamics, while on the other hand makes them inefficient in query processing because they have to find paths from scratch. 2) *Cache-based* approaches like *Local Cache* [31, 64] and *Global Cache* [50] that accelerate the query answering by caching previously answered shortest paths. Their boost on query processing efficiency comes with some extra overhead and could be vulnerable to query distribution since the query efficiency is proportional to both cache size and hit ratio; 3) *Contraction Hierarchy (CH)*-based algorithms and 4) *Hub Labeling (HL)*-based algorithms resort to index maintenance to adapt to dynamics. The adoption of an index guarantees efficient query answering, but the index maintenance is complicated and time-consuming.

Specifically, *CH*-based update methods include *vertex-centric* [16] and *shortcut-centric* [39, 55] techniques: the *vertex-centric* algorithm first identifies the affected vertex and then re-contracts them following the vertex order such that all the invalid shortcuts can be updated; the *shortcut-centric* algorithm decides the shortcut update order by exploring the shortcut priority and then maintains the shortcut value by keeping the corresponding property.

HL-based update methods contain both *search-based* and *propagation-based* approaches. [3, 10, 44] take *Pruned Landmark Labeling (PLL)* [2] as the underlying index and maintain correctness through the graph search, which is essentially maintaining multiple shortest

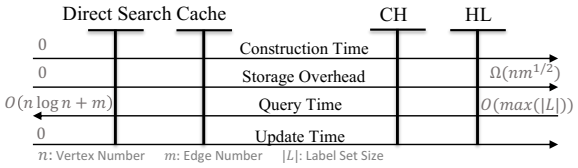


Figure 2: Comparison of *DSP* Algorithms

path trees. [62, 65] take *PLL* and *Hierarchical Labeling* (*H2H*) [38] as the underlying indexes and correct label values by propagating the affected labels to its neighbors. Even though these index maintenance methods perform well in query processing and can make the index "dynamic", they require additional memory because of index storage and time for index maintenance. In summary, these four categories of approaches have respective advantages and disadvantages. These approaches are compared in terms of their storage overhead, index construction time, query processing time and index update time in Figure 2.

Motivation and Contributions: Although these four types of algorithms are all applicable to dynamic scenarios, they all have their own pros and cons, such that we cannot find one method that is optimal for all circumstances. For instance, *HL* is fast for query answering but suffers from slow update while the *Direct Search* is the opposite. However, in each of the original papers, these methods were only tested in a very limited number of situations, such that the proposed one outperforms the others, while other scenarios are ignored. For example, *H2H* [38, 62] is very fast in road networks but is barely tested in other networks. Moreover, the traditional "construction time", "update time," and "query time" metrics can only reveal the performance without environment consideration. In other words, the existing papers are all algorithm-oriented but not problem-oriented. Consequently, without investigating the properties of dynamic networks, it is still unclear which method to use when facing complicated real-world scenarios. Finally, most of the *DSP* algorithms were proposed for *unweighted* graphs and claimed to be extended trivially to *weighted* graphs. However, their performance deteriorates dramatically during this conversion, and such phenomenon has been overlooked for a long time. To the best of our knowledge, the benchmark evaluations only exist in static networks [34, 56], so we aim to conduct a series of well-designed benchmark experiments to evaluate all *DSP* algorithms and provide insights and guidances towards real-world systems implementation.

We first provide a brief but insightful review of all the *DSP* algorithms to compare them theoretically and reveal their relations comprehensively. More importantly, we identify and explain the *Curse of Increase Update* that was hidden by either unweighted or small tree-width graphs in their existing works. It is this *curse* that causes the increase updates to either need more searches or more pre-computed information, which limits their scalability.

We then identify the following influential dimensions of dynamic networks: i) *Graph Topology* such as size, average degree, structure, and degree distribution are *determinate dimensions* of a network that have a profound influence on index construction, size and maintenance; ii) *Update Frequency* and *Update Volume* determine the index unavailability period, which index-free methods are immune to. We call them *constraint dimensions* that determines a networks dynamic environment; iii) *Query Amount* and *Query Frequency* are another type of *constraint dimensions* that determines if the current system can satisfy the application need. By combining these dimensions, we can obtain a complete problem space to describe all possible real-world scenarios.

After that, we delicately design a set of synthetic dynamic network environments according to our problem space under different parameters and systematically test the *DSP* algorithms. To have a deeper understanding of how each algorithm performs in different environments, we design a series of *dynamic degree*-based environments and compare their *throughputs*. In this way, we can identify the most suitable situation for each method. More importantly, for each situation, we can also obtain its current most suitable *DSP* method, which serves as a guideline for system implementation. Whilst the most suitable solution might still not be satisfactory for some situations, these results help to identify new problems and also serve as a guideline for future research. Finally, we use several real-life dynamic networks to test the above guidelines and validate their effectiveness.

Our contributions are summarized below:

- We identify the dimensions of the dynamic path finding problem space to model all real-life scenarios and propose a *dynamic degree*-based environment classification and *throughput*-based performance measurement to evaluate the actual performance.
- We present a comprehensive and insightful review of *DSP* algorithms to compare them in theory, and identify the *curse of increase update*.
- We conduct benchmark tests on all *DSP* algorithms to find the best method for each situation, which serves as the guidelines for system implementation and research opportunity identification.
- We conduct experiments on real-life dynamic environments to validate our guidelines. The source code of our implementation can be accessed online [1].

The remaining of this paper is organized as follows: We briefly define *DSP* problem in Section 2. In Section 3, we review all the existing *DSP* techniques. Section 4 discusses *DSP* problem dimensions and presents the experimental setups, followed by the experimental results, analysis and guidelines in Section 5. We discuss *DSP* related works in Section 6, followed by a conclusion in Section 7.

2 PRELIMINARIES

In this paper, we focus on a weighted network $G(V, E, W)$ where V is the vertex set, E is the edge set, and $W \rightarrow \mathbf{R}^+$ assigns a non-negative weight $e(u, v) \in W$ to each $(u, v) \in E$. For simplicity, we assume an undirected graph however all the mentioned methods can be extended to directed graph trivially. In a dynamic graph, since all the topological evolution can be generalized as edge weight updates [65], we ignore them in this paper for simplicity and define the dynamic network as follows:

[1] https://github.com/MengxuanZhang1/DynamicShortestPath_VLDB21

Definition 1. ***(Dynamic Network).*** *In a dynamic network $G_D(V, E, W)$, any $e(u, v) \in W$ can increase or decrease in the range of $[0, \infty]$ in ad-hoc.*

Although the above definition describes the real-life situation most precisely, we lose control of the dynamic such that we cannot study its influence in quantity. Therefore, we introduce the *Update Unit* $\Delta = (U, t)$ to discretize the dynamic information, where $U = \{e(u, v)\}$ is a set of new edge weights and t is the update time. The unit size $|U|$ determines the update volume, and the time between two units δt determines the update frequency. G_D describes the situation when the future Δ is unknown, while the *time-dependent network* is aware of all future Δ.

From now on, we refer to G_D as G if the context is clear. We denote the number of vertices and edges in G as $n = |V|$ and $m = |E|$ respectively. For each vertex $v \in V$, we represent its neighbors as $N(v) = \{u|(v, u) \in E\}$, and express the vertex degree as the number of neighbors via $deg(v) = |N(v)|$. Each vertex $v \in V$ is also associated with a vertex order $r(v)$ indicating its importance in G. A path $p = \langle v_0, v_1, \ldots, v_k \rangle$ $((v_i, v_{i+1}) \in E, 0 \leq i < k)$, is a sequence of vertices with length of $l(p) = \sum_{i=0}^{k-1} e(v_i, v_{i+1})$. We denote the shortest path and shortest distance between s, t as $p(s, t)$ and $d(s, t)$, respectively. Similar to the network update, we introduce a *Query Set* to control the quantity of queries. Given a starting vertex set S and a target vertex set T, a shortest path query set is denoted as $\hat{Q} = (Q, \tau)$, where $Q = \{q_i\} = \{(s_i, t_i)|s_i \in S, t_i \in T\} \subseteq S \times T$ is a set of shortest path query issued at time τ. $|Q|$ represents the query amount, and $\delta\tau$ reflects the query frequency.

Problem Definition. Given a dynamic network G with a stream of updates Δ and stream of queries $\hat{Q}$, the *DSP* problem aims to answer all shortest path queries Q_i in $\hat{Q}_i$ after applying the updates in Δ_i to G.

3 DSP ALGORITHMS

This section reviews 14 state-of-the-art *DSP* algorithms from three categories (*Cache* is discussed in Section 6.1 because it is a universal technique to boost all the other three categories). For the algorithms that were initially proposed for the unweighted graph, we extend them to the weighted version. We summarize and compare their complexities in Table 1. Due to the space limit, we omit the detailed complexity analysis and only show the results with their sources.

3.1 Direct Search Algorithms

The *Direct Search* algorithms involve no or little precomputed information so their construction time, size, and maintenance time are either 0 or very small. Therefore, they are immune to any dynamic but are slow at query processing.

3.1.1 Bi-Dijkstra's Algorithm. Given a shortest path query $q(s, t)$, *Bi-Dijkstra*'s [56] conducts a forward *Dijkstra*'s search from s on G and a backward *Dijkstra*'s search from t on the reversed G simultaneously, each traverses the vertices in increasing order of their distances from the source ($d_f(v)$ and $d_b(v)$ for forward and backward). Suppose $\overline{d}$ is the shortest distance ever found. Once a vertex v is visited from one search and its neighboring vertex w has already been visited reversely, $\overline{d}$ is updated if $\overline{d} > d_f(v) + e(v, w) + d_b(w)$. The algorithm terminates once a vertex has been visited in both searches, and then $d(s, t) = \overline{d}$ is the shortest distance. The *Bi-Dijkstra*'s search space consists of two smaller conceptual circles [5] with the radius sum slightly larger than $d(s, t)$. Therefore, it is more efficient compared with the *Dijkstra*'s big conceptual circle with a radius of $d(s, t)$.

3.1.2 A^* Algorithm. A^* [21] estimates the distance $h(v, t) \leq d(v, t)$ from the current vertex v to t heuristically and uses $d(s, v) + h(v, t)$ as the searching guidance. In this way, it reduces the *Dijkstra*'s search space to a smaller conceptual ellipse [5], and the closer $h(v, t)$ to $d(v, t)$, the smaller the search space. Although the *Euclidean Distance* is a widely used heuristic, it requires coordinate information which does not exist in non-spatial graphs. *ALT* [18] can be used on general graphs by pre-computing distances from some landmarks $L = \{l_i\}$ to all the other vertices and obtaining the heuristic distance $h(v, t) = max\{|d(l_i, v) - d(l_i, t)|\}$ with triangle-inequality. Consequently, it needs a set of *Dijkstra*'s to update the landmarks when the network changes.

3.2 Dynamic CH Algorithms

This category of methods add shortcuts through graph contraction and answer queries with the edges and shortcuts, so only the shortcuts need maintenance.

3.2.1 Graph Contraction, Shortcut, and Query. The vertices in G are contracted one by one in a pre-defined order (suppose lower to higher). During the contraction, the contracted vertex is removed, and path information through it is preserved by adding shortcuts among its neighbors. Specifically, there are two types of vertex contraction approaches depending on pruning or not:

Contraction with Pruning (CH-P). For the contracted vertex v, we go through all its neighbor pairs $u, w \in N_{G'}(v)$ in the partial contracted graph G' and compare the shortest distance $d_{G'}(u, w)$ with the sum of two edges $e(u, v) + e(v, w)$. If $d_{G'}(u, w)$ is shorter, then removing v would not affect the distance query, so the shortcut (u, w) is "pruned". Otherwise, we add (as a shortcut) or update $e(u, w)$ with weight $e(u, v) + e(v, w)$.

Contraction without Pruning (CH-W). Different from the previous method, we do not compute $d_{G'}(u, w)$ as it is time-consuming. Instead, we assign or update $e(u, w)$ with $min\{e(u, v) + e(v, w), e(u, w)\}$ directly. This method is distance-preserving as proved in [38]. Because it creates shortcuts between each vertex's all-pairs neighbors in the contracted graph G', it has the maximum shortcut number. Although adding a shortcut is extremely fast, the large shortcut number prohibits it from applicable to graphs with large treewidth. On the other hand, this densest shortcut set preserves all possible information, so it requires no search during maintenance.

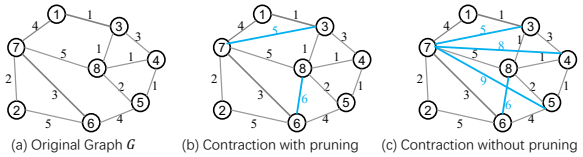


Figure 3: Graph Contraction Example

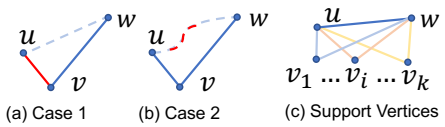
Figure 3-(a) is a graph with vertex in increasing order $< v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8 >$ and Figure 3-(b) is its intermediate *CH-P* result.

Table 1: Comparison of *DSP* Algorithms

Category	Algorithm		Construction Time	Size	Maintenance Time	Query Time
Direct Search	**Bi-Dijk** [56]		0	0	0	$O(nlogn+m)$
	A^* [18, 21, 63]		$O(nlogn+m)$	$O(n)$	$O(nlogn+m)$	$O(nlogn+m)$
Dynamic CH	**DCH-P** [15, 16]	+	$O(n^2logn+nm+\|P\|n)$	$O(nlog^2\sqrt{n}+m\|P\|)$	$O(\Delta h\cdot(nlogn+m))$	$O(w\log n)$
		-	$O(n^2logn+nm)$	$O(nlog^2\sqrt{n})$		
	DCH-W [39]		$O(n(w^2logn))$	$O(n\cdot w^2)$	$O(\delta w)$	$O(wlogn)$
	UE [55]		$O(n(w^2logn))$	$O(n\cdot w^2)$	$O(\delta w)$	$O(w\log n)$
Dynamic HL	**DPLL-S** [3, 10, 44]	+	$O(wmlogn+w^2nlog^3n)$	$O(wnlogn)$	$O(p'wm\log n+p'w^2nlog^2n)$	$O(wlogn)$
		-			$O(wm\log n+w^2nlog^2n)$	
	DPLL-P [65]	+	$O(wmlogn+w^2nlog^3n)$	$O(wn^2logn)$	$O(wmlogn)$	$O(wlogn)$
		-		$O(wnlogn)$		
	DH2H [62]	+	$O(n(logn+h\cdot w))$	$O(n\cdot w^2\cdot h)$	$O((\delta+\Delta h\cdot(\tau+w))w)$	$O(w)$
		-		$O(n\cdot h+n\cdot w^2)$	$O((\delta+\Delta h)w)$	

+: Edge weight increase case. −: Edge weight decrease case. *UE*'s complexities [55] were analysed on grid network with probability, so we use *DCH-W*'s complexity for general graphs as *UE* is the streaming version of *DCH-W*. Δh : Affected tree height. $|P|$: Witness path number. w: treewidth. δ: Affected shortcut number. τ: Affected tree node number. p': Affected label number / n.

When we contract v_3, (v_4, v_7) is one of its neighbor pairs with $d(v_4, v_7) = 6 < e(v_3, v_4) + e(v_3, v_7) = 5 + 3$, so we do not insert shortcut (v_4, v_7). However, as shown in (c), *CH-W* creates shortcut (v_4, v_7) as $e(v_4, v_7) = 8$ without comparing with $d(v_4, v_7)$. To answer a query $q(s, t)$, we only need to run a *Bi-Dijkstra*'s search upwardly (only visit the neighbors of higher order).



(a) Case 1 (b) Case 2 (c) Support Vertices

Figure 4: CH Update Examples

3.2.2 Shortcut Update. Updating shortcuts is essentially avoiding the follow lemma:

Lemma 1. ***(CH Incorrectness Condition)**: If $e(u, v) + e(v, w) < d_{G'}(u, w)$ on the partial contracted graph, but no shortcut (u, w) exists, then we cannot find the correct $d(u, w)$.*

Depending on which contraction type was used in construction, we have the following two types of updates:

Vertex-based Update. For the shortcuts added by *CH-P*, we need to keep the shortcuts satisfying the shortest distance constraint, so we have to re-contract the influenced vertices. *DCH-P* [16] first identifies the affected vertices by a *DFS*, then the pruned vertex contraction is applied to each affected vertex following the original order. Since it imitates the index construction procedure to update shortcuts, it suffers from high maintenance costs.

For the weight decrease update, there are two cases as shown in Figure 4-(a) and (b): a) $e(u, v)$ decreases could affect the shortcut (u, w). If $e(u, w)$ was created by (u, v) and (v, w), then we update $e(u, w)$ to the new value; If there is no shortcut (u, w), then a research from u to w is required to decide if a new shortcut is needed. b) The decrease could also make a shortest path $p(u, w)$ shorter. If it pruned the shortcut (u, w), then (u, w) is still not needed because $d_{G'}(u, w)$ is still shorter; If it did not prune (u, w) but now $d_{G'}(u, w) < e(u, w)$, we can just leave (u, w) as it is because it becomes a redundant shortcut like *CH-W*.

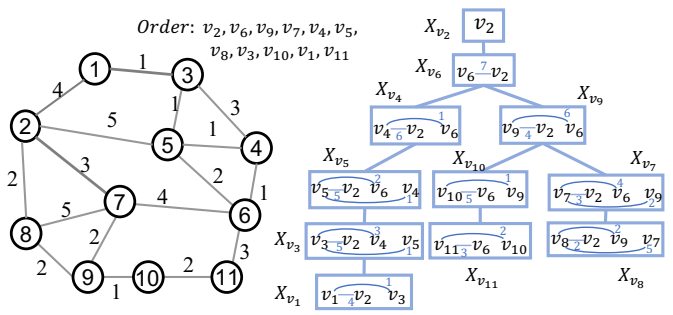
The weight increase case is more complicated because it may get caught by Lemma 1. Similar to the decrease, we also divide it in to two cases: a) If shortcut (u, w) does not exist, then the increase of $e(u, v)$ also could not create (u, w); If shortcut (u, w) was created by (u, v) and (v, w), then increasing the shortcut value won't affect the correctness. b) The increase could make a shortest path $d_{G'}(u, w)$ longer. If a shortcut (u, w) exists, then this increase would not affect it. If a shortcut (u, w) did not exist, then it was pruned by a shortest path $p_{G'}(u, w)$ because $e(u, v) + e(v, w) > d_{G'}(u, w)$, and $p_{G'}(u, w)$ is called a *witness path*. Now that $d_{G'}(u, w)$ increases, there is a chance that $e(u, v) + e(v, w)$ is shorter than the new $d_{G'}(u, w)$, so a shortcut (u, w) is needed to guarantee the correctness. However, we do not know which non-existing shortcut is affected so it may fall into the case of Lemma 1. To help identify the affected shortcuts, [16] stores the witness paths for all edges, which inflates the index size. We will discuss this phenomenon in Section 3.4.

Shortcut-based Update. For the shortcuts added by *CH-W*, although they have the densest shortcut set, they enjoy the *structure intactness* property, which avoids any shortcut insertion or deletion during maintenance, but only requires updating the shortcut values. In the decrease case when $e(u, v)$ decreases as shown in Figure 4-(a), we update $e(u, w)$ to $e(u, v) + e(w, v)$ if $e(u, v) + e(w, v) < e(u, w)$, and this procedure propagates recursively until no shortcut is updated. Because no pruning search occurs, it is much faster than *DCH-P*. In the increase case, suppose $e(u, w)$ was constructed from $e(u, v_1) + e(v_1, w)$ as shown in Figure 4-(c) and $e(u, v_1)$ increases. Then we need to find the new $e(u, w)$ from $min\{e(u, v_i)+e(v_i, w)|i \in [1, k]\}$, and this procedure repeats recursively. To store the support vertices, *DCH-W* [39] utilizes a *Shortcut Supporting Graph (SS-Graph)* and *UE* [55] uses *Occupant Mapping Table*, with *shortcut support* and *occupant* referring to the same support vertices concept. Another difference between them is that *DCH-W* can also process the edge weight change by batch, whereas *UE* can only process in stream, so *DCH-W* is more efficient than *UE* in maintenance. Nevertheless, they have the same query answering.

3.3 Dynamic HL Methods

This category of methods uses *labels* to answer the queries so maintaining the label correctness is at the center stage.

3.3.1 Label Construction and Query Answering. In hub labeling, each vertex $v \in V$ is assigned with a label set $L(v) = \{(u, d(u, v))\}$. The projection of $L(v)$ on the keys is called *hub nodes* $C(v)$ =



(a) Example Graph (b) Tree Decomposition

	Total Index
v_1	(1,0),(2,4),(3,1),(5,2),(4,3),(6,4)
v_2	(2,0)
v_3	(3,0),(5,1),(2,5),(4,2),(6,3)
v_4	(4,0),(6,1),(2,6)
v_5	(5,0),(2,5),(4,1),(6,2)
v_6	(6,0),(2,7)
v_7	(7,0),(2,3),(6,4),(9,2)
v_8	(8,0),(2,2),(9,2),(6,8)
v_9	(9,0),(2,4),(6,6)
v_{10}	(10,0),(9,1),(6,5),(2,5)
v_{11}	(11,0),(6,3),(10,2),(9,3),(2,7)

(c) PLL Hub Labeling

	Total Index
v_1	(1,0),(2,4),(6,4),(4,3),(5,2),(3,1)
v_2	(2,0)
v_3	(3,0),(2,5),(6,3),(4,2),(5,1)
v_4	(4,0),(2,6),(6,1)
v_5	(5,0),(2,5),(6,2),(4,1)
v_6	(6,0),(2,7)
v_7	(7,0),(2,3),(6,4),(9,2)
v_8	(8,0),(2,2),(6,8),(9,2),(7,4)
v_9	(9,0),(2,4),(6,6)
v_{10}	(10,0),(2,5),(6,5),(9,1)
v_{11}	(11,0),(2,7),(6,3),(9,3),(10,2)

(d) H2H Hub Labeling

Figure 5: HL Example. Label $(u, d(u, v))$

$\{u|(u, d(u, v)) \in L(v)\}$. We say the shortest distance can be correctly calculated through hub labeling L if it satisfies the *2-hop cover constraint*: $C(s) \cap C(t)$ shares at least one nodes in $p(s, t)$, $\forall s, t \in V$. Therefore, given one hub labeling satisfying the *2-hop cover constraint*, we can compute the shortest distance between s and t as $d(s, t) = \min_{c \in C(s) \cap C(t)} \{d(s, c) + d(c, t)\}$. It is faster than all the previously mentioned methods because it does not traverse the graph.

There are two main streams of hub labeling: *Pruned BFS-based (PBHL)* including *PLL* [2] and *PSL* [33], and *Tree Decomposition-based (TDHL)* including *H2H* [38], *Multi-Hop* [9], and *TEDI* [54]. We discuss the *PLL* and *H2H* here because they are the state-of-the-art of each category.

PLL Construction. *Pruned Landmark Labeling* got its name because its new labels are created through searches *pruned* by the previously created labels (*landmarks*). Specifically, it is constructed through the following two principles: *Principle 1) Dijkstra's search for label assignment:* We run a *Dijkstra*'s search from each vertex v in the decreasing order. When we have reached a vertex u, we can obtain the shortest distance $d(v, u)$ and then insert $(v, d(v, u))$ into u's label $L(u)$. In this way, the labels are added incrementally. *Principle 2) Query answering on partial label for search pruning:* When w is visited in the search from v and $q(v, w, L) \leq d(w, v)$, then $(v, d(w, v))$ will not be inserted into $L(w)$ and we do not visit w's neighors, where $q(v, w, L)$ is the shortest distance obtained from the existing labels L. An *PLL* example is shown in Figure 5-(c).

H2H Construction. *Hierarchical 2-Hop Labeling* got its name because it creates a *hierarchy* of the *2-hop* through tree decomposition. Specifically, it is built through the following three steps: *1) Tree Node Formation*: Following the same procedure as *CH-W*'s graph contraction, each contracted vertex v forms a tree node X_v that contains all its neighbors $N_{G'}(v)$ and the shortcuts between v and $u \in N_{G'}(v)$. *2) Tree Construction*: The tree nodes are connected by setting X_u as the parent of X_v, where u has the minimum order in X_v apart from v. A tree decomposition example is shown in Figure 5-(b). *3) Top-Down Label Assignment*: $L(v)$ contains the distance from v to all its ancestors, which is calculated as $\min\{e(v, u) + d(u, a_i)\}, \forall u \in X_v$. Because the assignment is conducted from root to leaves, $d(u, a_i)$ has already been obtained in the ancestors' labels. $q(s, t)$ can be answered by $min\{d(s, v_i) + d(v_i, t)|\forall v_i \in X_a\}$, where X_a is the *Lowest Common Ancestor (LCA)* tree node [9] of X_s and X_t. For example, $LCA(X_{v_1}, X_{v_{10}})$ is X_{v_6} with v_2 and v_6, then $d(v_1, v_{10}) = 4 + 5 = 9$.

3.3.2 Label Update. Depending on the affected area, the update algorithms can be categorized into the following three paradigms:

Search-Based Update. The first kind of *PLL* maintenance *DPLL-S* [3, 10, 44] is based on search because *PLL* was constructed by pruned search. For the weight decrease, we only need to re-run the pruned search from the lower vertex to update or add the labels. This is because the shortest paths that are affected by this edge decrease will get the new correct results, while the other paths can still be answered by other labels. This process may make some labels redundant, but it will not affect the query correctness. However, if an edge weight increases, similar to the *CH-P*'s increase case, it could invalidate a set of labels that pruned the searches during construction, so some labels are missing. Therefore, it needs to run the *Dijkstra*'s and label testings recursively to identify a set of affected vertices first, and then re-run pruned search again like the decrease case.

Affected Area Unbounded Propagation. To avoid the expensive search operations of the previous method, *DPLL-P* [65] propagates the update recursively until no new label is updated. Because the affected area cannot be determined beforehand, we call it *unbounded propagation*. Specifically, when v's label $(u, d(u, v))$ changes, we propagate this change to v's neighbors that have lower order than u and update the label values with the *pruning principle*. This procedure works for the decrease case due to the same reason of *DCH-P* and *search-based update*. However, for the increase case, the newly increased labels are invalid and the correct ones were pruned. Therefore, we store all the hub nodes that were pruned by each label and add back the ones that are smaller than the query distance now.

Affected Area Bounded Propagation. Because *H2H* has a hierarchical structure to organize all the labels, *DH2H* [62] can propagate the label changes with the bounded area. Since the labels are created from *CH-W* essentially, it first updates the shortcuts in the same way as *DCH-W*. For the label update, the sub-tree with the highest affected tree node is the possible affected area and should be updated with the same *top-down label assignment*. However, this is a big area so we can reduce it by the following pruning techniques: 1) Lowering the sub-tree root from the highest affected tree node

to the highest tree node that contains updated shortcut; 2) Only propagate the changes to those labels that were created by updated shortcuts and labels by storing which vertex contributed to each label; 3) Identify the affected area dominance relation such that several updates could be conducted in a single run.

3.4 The Curse of Increase Update

All the decrease index update is relatively easier because decrease means adding new shortest path with smaller distances, and such information can be derived directly by partial re-construction from the decreased edge. Because no pruning condition can stop a smaller-than-query value, the decrease update can be processed easily. However, the increase update will invalidate the current shortest path, which needs to be replaced by the correct one. For example in Figure 6, suppose the $p(s,t)$ is the old shortest path. If an edge $(a,b) \in p(s,t)$ increases and makes $l(p(s,t)) > d(s,t)$, then the new shortest path can fall into two categories: 1) The green path which $\exists c \in p'(s,t)$ such that $r(c) > max\{r(s), r(t)\}$. In *CH*, $p'(s,t)$ can exist as the new witness path for correct query answering; In *HL*, $d'(s,t)$ can be correctly computed by its highest c'. 2) The red path which $\forall c \in p''(s,t)$ such that $r(c) < max\{r(s), r(t)\}$. In *CH*, $p''(s,t)$ was pruned so we have lost it. In *HL*, $p''(s,t)$ was also pruned by $p(s,t)$'s the highest hop c. In other words, the information of $p''(s,t)$ cannot be recovered by just *CH* or *HL*, and this further leads to incorrectness of the index.

Figure 6: Increase Curse Example

To bring the actual shortest path $p''(s,t)$ back, the index-based *DSP* algorithm have to store extra information. Specifically, *DCH-P* stores all the *witness paths* of each shortcut, *DCH-W* and *UE* create shortcuts among all neighbor pairs and store their corresponding *support vertices*, *DPLL-S* runs several rounds of *Dijkstra*'s to identify the affected vertices, *DPLL-P* stores all the *pruning points* of each label, and *DH2H* stores all the information of *DCH-W*. Among these solutions, *DPLL-S* trades space with time, so it is the slowest to update. *DCH-P* and *DPLL-P* have to store a huge amount of extra data to support increase. Although *DCH-W*, *UE*, and *DH2H* seem to have smaller extra data sizes, they actually use the much larger index size (the densest *CH* with size proportional to tree-width square) to cover it. We call this high extra cost for the increase maintenance as the *Increase Update Curse*. This phenomenon was ignored in the previous works for the following reasons. Firstly, most of the maintenance works on the big treewidth graphs only consider the unweighted scenario [3, 10, 44], which has a very small diameter and the update search is fast with parallel bitwise *BFS*. Although they claimed the weighted version is easy to implement, the larger diameter and the slower *Dijkstra*'s search deteriorate the increase update performance dramatically. Secondly, the works on the small treewidth graphs [39, 55] use *CH-W* to cover a large amount of redundant information, whose complexity limits their applications only on small road networks. Therefore, how to maintain the weight increase case efficiently, especially in terms of space consumption, is still an open problem.

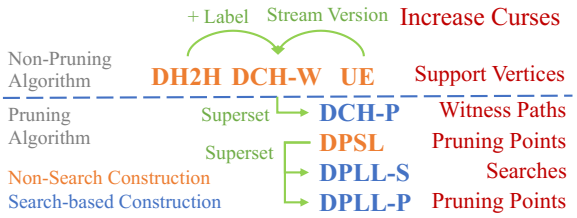


Figure 7: DSP Algorithm Relations

In summary, we analyze and demonstrate the relations of different *DSP* algorithms in Figure 7. Roughly, we categorize them into two types according to whether there is a pruning condition during index construction. Specifically, the *Non-Pruning Algorithms* include *DCH-W*, *UE* and *DH2H* and their relations are: the index update of *UE* is the stream version of *DCH-W*; *DH2H* is built from *DCH-W* with further propagation-based label construction; their constructions all rely on vertex contraction. Among the *Pruning Algorithms*, *DCH-P* prunes the unnecessary shortcuts using *Dijkstra's* search; *DPSL* [65] ignores and stops propagating the unnecessary labels; with the same underlying index, *DPLL-S* and *DPLL-P* prunes the labels based on *Dijkstra's* search. Besides, *DPSL* has been proved to be the superset of *DPLL* [65]. Since it is slower in both query processing and index update than *DPLL-P*, we do not include it in our experiments. Lastly, they all suffer from the *curse of increase update*: *Non-Pruning Algorithms* all require large storage of support vertices; *DCH-P* needs to make much space for witness paths; both *DPSL* and *DPLL-P* need to record a large number of pruning points. These auxiliaries play an important role in increase update as they preserve that information which cannot be easily retrieved unless index re-construction. Meanwhile, their sizes are astonishingly orders of magnitudes larger than the index size itself, which in turn dramatically drags down the update speed. Even though *DPLL-S* utilizes no extra auxiliary besides the index in index update, it still suffers from slow update because it relies on direct graph search.

3.5 Topology Update

In this section, we discuss the deletion and insertion of the edges and vertices that could change the index structure:

Edge Deletion. It can be reduced to the weight increase as it is equivalent to increasing the edge weight to ∞.

Vertex Deletion. It can be reduced to multiple weight increases as it is equivalent to deleting its adjacent edges.

Edge Insertion. It can be reduced to the weight decrease from ∞ to a smaller value. Edge insertion is easier for the *CH*-based and *PLL* index maintenance because they can propagate this change to the affected areas [65]. However, for *DH2H*, extra efforts are needed to modify the tree structure [62]. If the two ending vertices have an ancestor-descendant relation, it will not affect the tree structure. Otherwise, suppose edge (a,b) is inserted with X_c being the *LCA* of X_a and X_b, the tree decomposition is partially adjusted by merging the tree nodes from X_c to X_a and from X_c to X_b in the increasing

order of their representative vertex's order such that the two small branches are merged.

Vertex Insertion. To begin with, the inserted vertex is assigned with the lowest order. In this way, it would be the "first" vertex to contract in *CH*, and the last to assign label in *HL* (would not appear in other's label). Depending on the inserted edge number, the vertex insertion can be classified into the following scenarios: i) *No edge*: it is trivial and can be ignored because the vertex is isolated. ii) *One edge*: because the inserted vertex is a dead end, it would not introduce new shortcut in *CH* and cannot affect other labels in *HL* (label inherited from its neighbor). iii) *More than one edge*: it can be reduced to scenario ii) plus the multiple edge insertions.

In summary, all the topology update can be generalized to the edge weight updates.

3.6 Path Retrieval

In this section, we summarize the path retrieval of the above methods. For the search-related methods, we need to store which neighbor (*parent*) updated the current vertex (*Direct Search*), and further store it in the label (*PLL*). Then the path can be recovered by traversing the parents reversely from the destination. For the *CH*-based methods, the contracted vertex that created each shortcut should be stored. During the path retrieval, we first obtain a concise path mixing with shortcuts using the previous search-based method. Then the shortcuts are recovered to the actual edge recursively. For *H2H*, as it is essentially a hierarchical *CH-W*, we only need to recover the two *CH* shortcuts ($s \rightsquigarrow h$ and $h \rightsquigarrow t$). Because these methods only visit the vertices on the path, their complexities are all $O(|p|)$.

4 EXPERIMENTAL SETUP

The main objective of the experiments is to test the state-of-the-art *DSP* methods' performance systematically in different dynamic environments. Specifically, we aim to answer the following questions:
Q1: Which factors of the dynamic networks affect the *DSP* algorithms' performance?
Q2: Given a using scenario, which algorithm should I choose?
Q3: Are all the scenarios solved satisfactorily?

In this section, we first discuss the possible factors in Section 4.1. Then we present how to create the dynamic networks synthetically and categorize the real-life networks according to these factors in Section 4.2. Section 4.3 explains the performance indicators, and Section 4.4 describes how the algorithms are implemented.

4.1 DSP Problem Dimensions

From the dynamic network's perspective, we have a set of factors that determine each network's characteristics. From the *DSP* problem's perspective, these factors are the dimensions of the problem space. In the following, we introduce the dimensions from three different categories.

4.1.1 Update Volume and Update Frequency. *Update* $\Delta(U, t)$ is the essential feature that distinguishes the dynamic network from the static network. Specifically, it is the volume $|U|$ and frequency δt together that determine the degree of dynamic. For instance, a large $|U|$ happens every short period means the network changes very dramatically, while a small $|U|$ happens over a long period means the network is tending stable. Therefore, we combine these two parameters and define *dynamic degree* as followed to measure the severity of network change:

Definition 2. ***(Dynamic Degree).*** *We capture the dynamic degree by the number of weight changes within one unit time interval* $\mu = |U|/\delta t$.

Finally, Δ and the *DSP* algorithm's update efficiency together determine the system's unavailable period as illustrated in the red blocks of Figure 8.

4.1.2 Query Amount and Query Frequency. The *query stream* $\{\hat{Q}\}$ is another critical factor to consider when implementing a real-life system. The query amount $|Q|$, especially the peak amount $max\{|Q_i|\}$, determines the minimum number of queries a system should be capable of coping with during $\delta\tau$ (frequency). If an algorithm can only process a few queries, it needs more servers to meet the system requirement. Therefore, how many queries an algorithm can process is a crucial factor in determining the system size. For example, in Figure 8, suppose the height of the yellow block represents the query volume one algorithm can answer during the query processing period (green), and the whole yellow region represents the total number of queries issued, then the number of the blocks is how many servers are required to meet the system need.

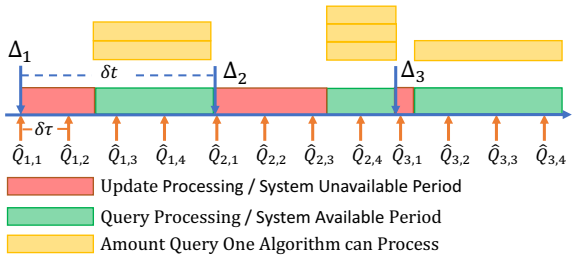


Figure 8: Update and Query Relation

4.1.3 Graph Topology. Different from the previous two categories, the *graph topology* is a set of *internal* factors that determine the algorithms' preprocessing time and space consumption (feasibility), maintenance time (unavailable time), and query time (available time). These factors can be roughly categorized into *vertex number* $|V|$, *average degree* D and graph structures. Specifically, $|V|$ affects the scalability, and D determines the density. As for the graph structures, *lattice graph* represents simple low-degree planar graphs like road networks, *regular graph* requires all the vertices to have the same degree, *small-world graph* has high clustering and short average path length, and *scale-free graph*'s degree has a power-law distribution. In addition, treewidth w is also an important factor used in many algorithms' complexity analysis, but it is derived from the previous factors, so we regard it as an indicator instead of a controllable variable.

4.2 Dynamic Network Design

In this section, we present the experiment settings. Some of the dimensions can influence the algorithm performance, so we call them *determinant dimensions* and consider them as part of settings, while

Table 2: DSP Problem Dimensions and Settings

Category	Dimension	Values
Determinant Dimension	Vertex Number	1k, 10k, 100k
	Average Degree	5,10,20,30
	Graph Structure	Lattice, Regular, Small-World, Scale-Free
Constraint Dimension	Update Volume	10, 100, 1k, 10k
	Update Frequency	1, 10, 100, 1k
	Query Amount	*Constraint*
	Query Frequency	*Constraint*

the others affect the system's usability, so we call them *constraint dimensions* and discuss their influence on real-life implementation. The classification and settings are summarized in Table 2.

4.2.1 Update and Query Set. Firstly, among the update dimensions, *update volume* $|U|$ determines the algorithm performance, so we test the volumes of 10^i with $i = 1$ to 4. The *update frequency* δt is a real-life constraint because a slow algorithm might still be tolerable as long as the updates seldom occur. Suppose we set the frequency evenly, then the *dynamic degree* μ can be represented by the volume. We distinguish the increase and decrease for some of the tests to show the *Increase Curse*. The updated edges are chosen randomly, with weights either decrease by 50% or increase by ten times.

Secondly, among the query dimensions, we treat the *Query Amount* as a constraint dimension and generate 10k queries randomly for each category. The average query time would be used to discuss the feasibility. *Query frequency* $\delta\tau$ is also a constraint dimension similar to the *update frequency* but with shorter periods. Because $\delta\tau$ only affects the server number and can be calculated manually, we do not test it.

4.2.2 Synthetic Networks. The network topology dimensions are all determinant dimensions so we generate the networks synthetically to fully test each dimension's influence. For the vertex number $|V|$, we set it to 1k, 10k and 100k. For the average degree D, we set it roughly to 5, 10, 20, and 30 for *small-world and scale-free graph*, 3, 4, 6 for *lattice graph* and 3, 4, 5, 6 for *regular graph*. Then for the network structure, we use *LightGraphs* [49] and *NetworkX* [20] to generate them under all the $|V|$ and D combinations randomly. Specifically, we use *Watts-Strogatz* model [53] for the small-world graph, and *Barabasi-Albert* model [4] for scale-free graph.

Table 3: Real-life Networks

Networks	Name	$\|V\|$	$\|E\|$	D
Road Network	Beijing (BJ) [1]	296,710	774,660	2.61
Social Network	Skitter (SKIT) [2]	1,696,415	21,990,934	12.96
Web Graph	Wikipedia (WIKI) [2]	3,333,397	200,943,616	60.28

[1] *NavInfo*: https://www.navinfo.com/en, private data;
[2] http://konect.uni-koblenz.de

4.2.3 Real-life Networks. We select several real-life networks of various graph topologies and different scales to test our claims obtained from the synthetic tests. The network description is shown in Table 3. For the *BJ* road network, we extract the traffic information from taxi trajectories collected on 1^{st} April 2015 [32, 61], which has 288 5-minutes snapshots of traveling time. Its number of updates between snapshots are shown as bars in Figure 15. The weights of *SKIT* and *WIKI* are generated randomly from 100 to 1000 to simulate the closeness, similarity, proximity, intimacy, information transmission, and contacts between the vertices. The update and query data follow the same rules used in Section 4.2.1.

4.3 Performance Metrics

Index Construction Time: We report the construction time in second and exclude the I/O time. This metric determines the static feasibility of a network.

Space Overhead: We use the 32-bit integer to save the vertex ID, weight value, or path ID in the index and demonstrate the index sizes. Specifically, *CH* stores the shortcut in form of (*vertexID, weight*), and *HL* stores the labels in form of (*vertexID, d*). This metric also determines the static feasibility in terms of memory usage.

Query Answering Time T_Q: We report the average answering time of randomly generated or real-life queries and use it to derive the query set answering time. This metric corresponds to the query amount and frequency constraints.

Index Maintenance Time T_U: We report the average index update time caused by randomly generated or real-life edge weight updates and use it to derive the update set processing time. This metric corresponds to the update volume and frequency constraints.

System Throughput θ: Given a specific dynamic environment with $|U|$, δt and $|Q|$, the system throughput $\theta = (\delta t - |T_U|)/T_Q$ is the maximum number of queries a system can process during δt after all U are updated. This metric corresponds to the query amount constraint.

4.4 Algorithm Implementation

We implement and evaluate the following 13 algorithms: i) *Direct Search Algorithms (DS)* including A^* and *Bi-Dijk*'s depending on if the coordinate is available; ii) *Dynamic CH* including *DCH-P* [16], *DCH-W* [39], and *UE* [55]; and iii) *Dynamic HL* containing *DPLL-S* [10], *DPLL-P* [65] and *DH2H* [62]. The algorithms ending with *-Inc* are their *increasing* version, while *-Dec* are their *decreasing* version.

We implement all the algorithms by ourselves to make sure they share the same base data structure, library, and optimization standard and do not involve any specific heuristic to make the comparison unfair. The algorithms that require ordering use the same order on different graphs. Specifically, for the regular and lattice graph, we use *Minimum Degree Elimination (MDE)* [8] order (computed from *CH-W*) as it has a good performance and fast to obtain. For the small-world and scale-free graph, we use degree order [22] as it represents the vertex importance. We do not discuss the influence of order since it is a different topic from this paper. All the algorithms are implemented in C++ with -O3 optimization and tested on a Dell R730 PowerEdge Server, which has two Xeon E5-2630 2.2GHz (each has 10 cores and 20 threads) and 378G memory.

5 EXPERIMENT RESULT ANALYSIS

This section discusses the experimental results comprehensively, followed by the guidance obtained for future research and validation on real-life networks.

5.1 Evaluation on Synthetic Network

5.1.1 Experimental Results. Figure 9 shows the index construction time and space consumption of the four synthetic graph types, and

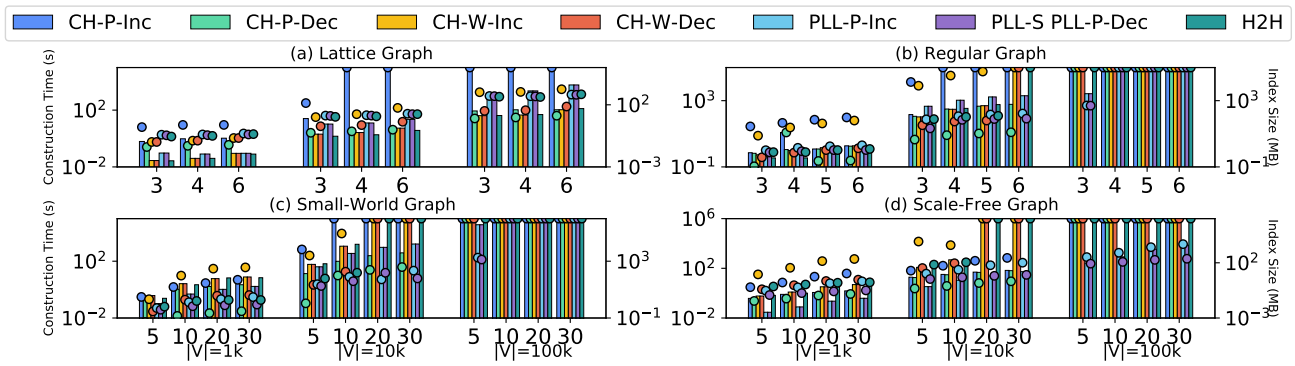


Figure 9: Synthetic Network Index Construction Time (Bar) and Size (Ball)

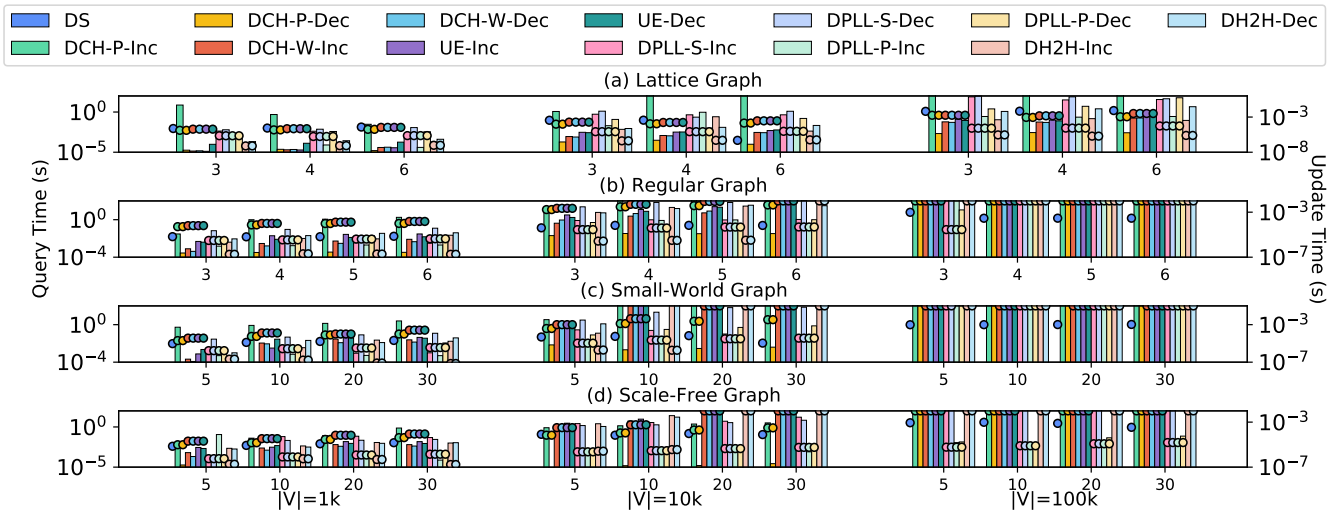


Figure 10: Synthetic Network Query Time (Ball) and Maintenance Time (Bar)

Figure 10 shows the corresponding maintenance and query processing time. The algorithms that exceed 200G memory or takes more than one day to finish construction are regarded as ∞ construction time and index space, and we draw them on the top of the y-axis. We first discuss from the perspective of the algorithms. For starters, *DS* is the only algorithm that works on every network because it has no index to construct or maintain. Among the *CH*-based methods, *DCH-W* always has the largest shortcut number because it is the densest scenario. As the graph becomes larger or denser, it exceeds the memory limit easily and takes longer time to construct, so it is only suitable for small lattice graphs. Due to the same reason, it is also the slowest to process queries. *UE* is worse at maintenance than *DCH-W* because it is essentially the streaming version of *DCH-W*. Moreover, their *supportive vertices* also increase dramatically because of the *increase curse*. *DCH-P-Dec* always has the smallest shortcut number because it applies pruning during construction. It can tolerate larger and denser graphs than *DCH-W* and fast to update. However, for the *DCH-P-Inc*, it has to store a huge number of witness paths, so it always performs the worst. In fact, its large space consumption, long construction time, and slow update time is a vivid example of the *increase curse*. Finally, among the *HL*-based methods, *DPLL* can apply to most networks. Although it is slow to construct because its weighted version cannot run in parallel, it takes much smaller memory than *DCH-W* and *DH2H*. Constraint by the performance of *DCH-W*, *DH2H* is more limited than *DCH-W* even though it has the fastest query time. *DPLL* has the second-fastest query time for most cases, and its propagation version *DPLL-P* also takes a shorter time to update. *DPLL-S* is always among the slowest to maintain, so we will not mention it in the future.

Next, we discuss from the perspective of the networks. Generally, as the graph becomes larger and denser, the indexes are harder to build and maintain. For the lattice graph, all algorithms except *DCH-P-Inc* work well. Because it has a smaller tree-width, the *CH-W*-based methods work best. For the regular graph, due to its randomness, all methods suffer even when the degree is small. The small-world graph has similar behavior but with larger degrees. Only the *pruning*-based methods survive while all *CH-W*-based methods explode. The scale-free graph is more friendly to the *pruning*-based methods as even *DCH-P-Inc* can scale and *DPLL* can

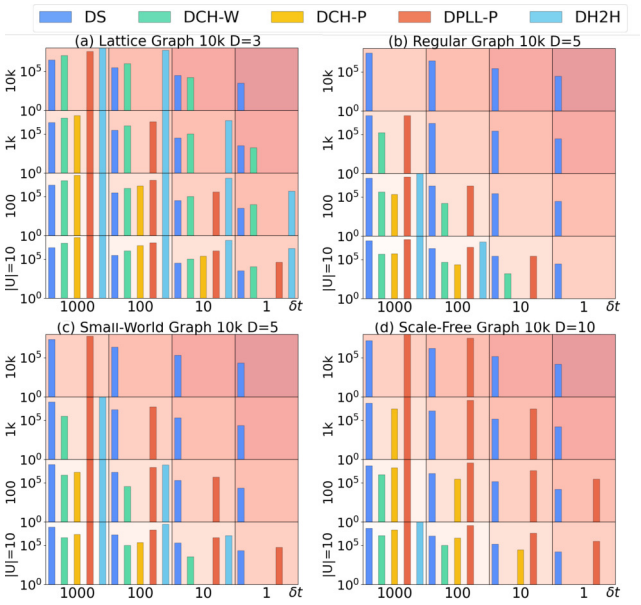


Figure 11: Synthetic Throughput Network View

further work in the 100k graphs. On the other hand, *CH-W*-based methods fail earlier than others.

In summary, the *CH-W*-based methods are only suitable for smaller and looser graphs with smaller tree-width but cannot scale due to their huge memory consumption. The *pruning*-based methods have longer construction time but require smaller memory, so they have the potential to scale to more graph types.

5.1.2 Throughput Analysis. The previous discussions focus on the traditional metrics. However, because they ignore the dynamic constraint dimensions, their results could be misleading. In this section, we divide the dynamic environments based on update volume $|U|$ (from 10 to 10k) and update interval δt (from 1000s to 1s) into 4×4 scenarios. For example, in Figure 11, the update becomes more frequent from left to right and becomes more from bottom to top. All the grids with the same color have the same *dynamic degree* $\mu = |U|/\delta t$, which increases from bottom-left to top-right (light to dark red). It should be noted all the throughput tests are undertaken in the mix update scenario with equal number of increase and decrease.

We test the following five representative algorithms: 1) *DS* that has no maintenance time; 2) *DCH-W* that represents *CH-W* because it has the same query efficiency as *UE* but faster in index maintenance; 3) *DCH-P*'s performance is limited by its increase version, so we choose the largest graph that *DCH-P-Inc* succeeded in construction; 4) *DPLL-P* represents *PLL* because it has the same query efficiency with *DPLL-S* but faster in maintenance; 5) *DH2H*.

Firstly, Figure 11 compares the throughput of different algorithms in four selected networks. In the first lattice graph, *DS* always shows the best durability, which indicates its capability of query processing in all dynamic scenarios, followed by *DCH-W*, *DH2H*, *DPLL-P*, and *DCH-P*. *DCH-W* fails only when $\mu = 10k$ and works in more scenarios than *DCH-P*, since *DCH-P* suffers from slow increase update. *DH2H* works up to $\mu = 100$ with the highest throughput. Besides, its throughput is always higher than *DPLL-P* since it has faster maintenance and query processing. In the second regular graph, most algorithms leave no time to process queries except *DS* because their index update can hardly catch up with the graph evolution. *DPLL-P* and *DCH-W* can only work when $\mu \leq 1$, with *DPLL-P* having higher throughput. In the third small-world graph, the algorithm performance is similar to those in the second graph, except for *DPLL-P* showing better durability. This tendency also appears in the forth scale-free graph with *DPLL-P* can further work in $\mu \leq 100$, whilst *DH2H* only survives $\mu \leq 0.01$, due to the large treewidth.

Secondly, Figure 12 shows the throughputs from the perspective of the algorithms, which can help to identify the best use scenarios of each algorithm. Specifically, *DS* can adapt to all dynamic scenarios but with relatively low throughput. *DCH-W* and *DH2H* are both mostly applicable on the lattice graph and have the worst performance on the scale-free network since they are more suitable on networks with low treewidth. *DCH-P* can only work when the dynamic degree is low because its increase update is seriously stumbled by the heavy auxiliary information (the support vertices and witness paths). Finally, *DPLL-P* is suitable for both small-world and scale-free networks.

5.1.3 Guidelines. Now we are ready to answer our questions and provide the following guidelines:

DSP Implementation Guidelines. **1) *DSP* Factors:** Internally, the performance of *DSP* algorithms can be affected by the *determinate dimensions* which include various elements like vertex number, average degree, and graph structure. Externally, their performance can be affected by the update volume and update frequency and constraint by query amount and query frequency.
2) *DSP* System Implementation: We first need to obtain the update and query performance of each candidate algorithm. Then we can draw a throughput plot to compare their performance under different dynamic scenarios. After that, the *dynamic degree* analysis can rule out the impossible algorithms. Depending on the *query constraints* of the application scenarios, we can use the algorithms that satisfy the constraints safely.
3) General *DSP* Using Scenarios: i) When μ is high, *DS* is almost the only choice. ii) In lattice graph (like road network), *DH2H* is the best choice when the μ is not too big. iii) In both small-world and scale-free graphs, *DPLL-P* is a good choice in most situations as long as μ is not too big. iv) *DCH* always has a lower throughput than *HL* in all dynamic scenario.

Open Challenges and Future Directions. **1) Highly Dynamic Low Throughput Problem:** For the scenarios with high dynamic degrees, the current index methods cannot scale so the overall system throughput suffers. Moreover, as the business increases, the query number also keeps increasing. Therefore, how to increase the scalability of the existing indexes in terms of both construction and maintenance is crucial for the real-life application.
2) The *Curse of Increase Update*: As it either requires a tremendous amount of searches during update or huge pre-computed information during construction, the index increase is still a severe

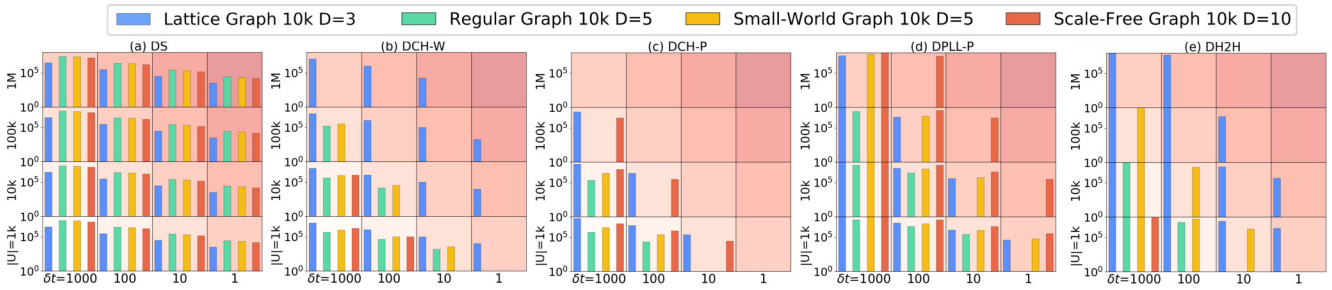


Figure 12: Synthetic System Throughput Algorithm Perspective

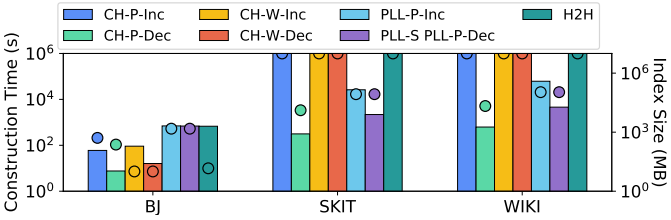


Figure 13: Real-life Network Index Construction Time (Ball) and Size (Bar)

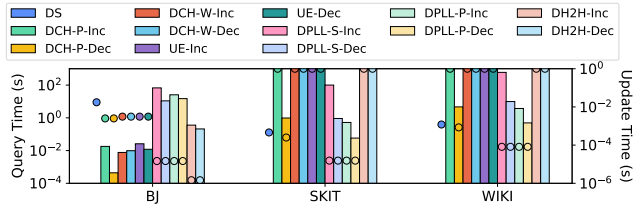


Figure 14: Real-life Network Query Time (Ball) and Maintenance Time (Bar)

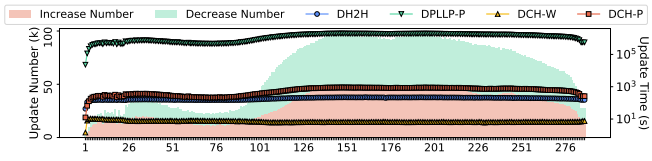


Figure 15: Beijing Real-Life Maintenance

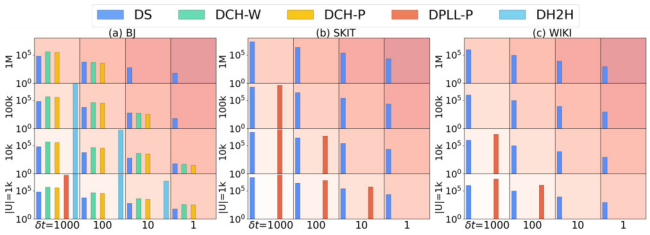


Figure 16: Real-Life Throughput Network View

open problem to solve. Theoretically hard, but a practical solution is in urgent need.

3) *PLL* Parallel Construction and Maintenance: *PLL* is a promising index structure that can apply to a wider range of network structures, but it is limited to its slow construction. Although a parallel [26] algorithm exists, it still suffers from the accuracy issue. *PSL* [33] can construct in parallel but it creates a superset of *PLL*. Meanwhile, the current *DPLL-P*'s maintenance efficiency is still slower than the others, and its throughput stands out only because of its scalability. Therefore, constructing and updating *PLL* efficiently would benefit the *DSP* problem a lot.

4) Hybrid *DSP*: As the real-life network keeps changing with different updates and query patterns at different times, how to utilize different algorithms together for different scenarios to keep the maximum system availability and throughput is also a significant practical problem.

5.2 Validation on Real-Life Network

In this section, we validate our results on real-life networks. Figure 13 shows the construction time and index size. Specifically, the *CH-W*-based methods work well on road network, but they cannot scale to social network and web graph. *CH-P* works well in decrease, but its increase version fails to construct because of the huge amount of *witness paths*. The *PLL*-based methods are generally slower in construction, but they can scale to larger and complex networks. Figure 14 compares the update and query performance. The index-based methods all have smaller query time but larger maintenance time, with *H2H* faster than *PLL*, and *PLL* faster than *CH*. For the maintenance time, *CH*-based methods are faster than *PLL*. But on *SKIT* and *WIKI*, only *DCH-P-Dec* and *PLL*-based methods survive, and *PLL-P* performs best. Moreover, we test with the real-life update data of *BJ* [32] as shown in Figure 15. It contains 287 sets of mix updates between 288 pieces of 5-min intervals of on 1^{st} Apr 2015. *DCH-W* and *DH2H* are the only two methods that can finish updating within each update period because they take advantage of the batch processing. *DPLL-P* is fast for the small-world network but slow in the road network, and the current sequential maintenance method cannot work in real life. *CH-P* is slightly slower than *DH2H* but its query is much slower. To further identify their performance under the dynamic environments, Figure 16 reveals the similar trends we got from the synthetic environments: although the *DS* algorithms are slow to run, they are the only choice when the μ is high. *CH* and *H2H* are suitable for road network, and *PLL* is suitable for other networks. Figure 17 further presents the

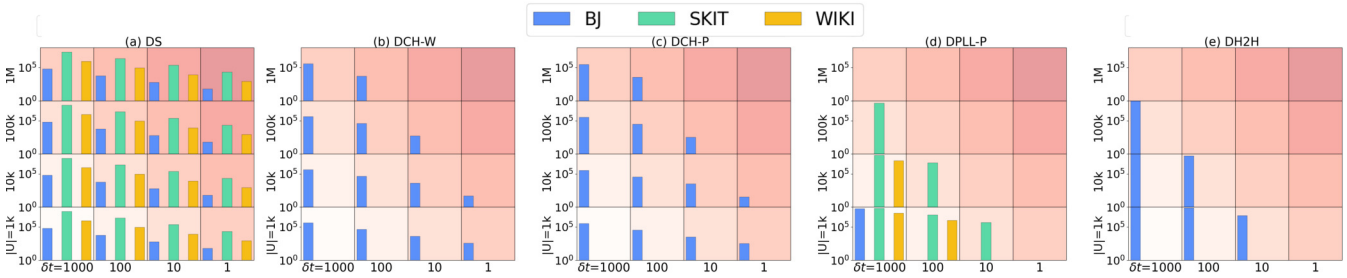


Figure 17: Real-Life System Throughput Algorithm Perspective

most suitable environments of each algorithm: *DS* has the broadest adaptability in terms of both network type and update frequency. *DCH* and *DH2H* are only available on *BJ* road network in case of no frequent update. *DPLL* is more suitable for small-world and scale-free networks under lower dynamic.

6 RELATED WORK

6.1 Static Shortest Path Algorithms

These algorithms work in the most basic scenarios where the networks remain static. To speed up query processing performance, different kinds of indexes are proposed. Specifically, the *Goal-Directed* approaches [18, 19, 37, 43, 51, 63] make use of the coordinate information in road networks and drag the search space towards the target based on the information in indexes (such as *Landmark* and *Flag*). The *Separator* approaches [24, 60] divide a road network into several small separators which are connected by boundary vertices or cuts, and queries can be processed in each separator. The *Hierarchical Techniques* [6, 17, 46, 67] construct a hierarchical structure for a network and prune the search space by only allowing graph traversal towards the upper layer. *Tree Decomposition* [38, 54] is another type of hierarchical structure using the cut property to reduce the search space. *Hop Labeling* [1, 2, 14, 22, 23] techniques associate each vertex with a distance label that contains coverage property to ensure that the shortest distance between any two vertices can be answered with these labels only. *Materialized Techniques* [45, 47, 48] store all-pair shortest path information to reduce index sizes by taking advantage of the path coherence property. To further improve query processing performance, *caching techniques* can also be used by reusing the results from previous answered queries. One way to improve hit ratios is to create a cache for each cluster of queries locally [31, 64] instead of globally [50]. However, its usability is limited as spatial information such as coordinates is required to cluster queries. Another way is to reduce the vertex number by mapping the vertices within a distance range ϵ to their central vertex, such that the paths in a cache become more concise. To deal with the dynamic environment, such caches have to be rebuilt often. It is a technique that can be applied to all other algorithms and but the performance depends highly on query distribution.

6.2 Dynamic Shortest Path Algorithms

Apart from the ones described in this paper, [55] also compare with *SILC-Adapt* [45], *AH-Adapt* [67] and *H2H-Adapt* [38]. These algorithms are essentially based on re-construction, so we do not test them in this paper. There also exist other attempts to solve *DSP* approximately. The *Region-to-Region* method [31] decomposes the query set into several subsets by referring to a given approximation error η and process each query subset together with bounded error η. A similar approach has been developed without error bound [36]. [61] strategically selects a set of snapshots and performs query answering by matching the current snapshot to the most similar one. It is also an approximation bound with no error bound.

The ***time-dependent*** model [12, 13, 25, 28, 32] uses a time-dependent function to tell the weight at different time. It can schedule the routes when the network weight change is periodically predicable. As for the index-based methods, *TCH* [7] extends *CH*, [52] extends *G-Tree*, and [29, 30] extend *PLL*. Nevertheless, as the time-dependent function is essentially a static function, this model cannot work in the ad-hoc dynamic environment.

Stochastic shortest path [40, 41, 57, 58] is another attempt to capture the dynamic by viewing the ever-changing travel time as uncertainty and use the probability distribution function to describe it. [58] first combines it with time-dependent and multiple costs to achieve eco-routing, and [41] improves efficiency with *CH*. [57] and [40] further improve the probability computation accuracy by computing the probability over paths instead of edges. Nevertheless, the probability distribution is still essentially static and cannot cope with the ad-hoc changes in the dynamic scenario.

7 CONCLUSION

In this work, we have thoroughly studied the *Dynamic Shortest Path* problem. We first review, analyze, compare, and provide relations for previously proposed *DSP* algorithms theoretically to give deep insights on the *DSP* problem. We also discuss the *curse of increase update* challenge which explains why all *DSP* problems are hard in terms of increased update costs, especially on weighted graphs. Then we identify and classify the *DSP* problem dimensions and use system throughput to evaluate the algorithm performance under different environments. With this benchmark test, we provide a guideline for system implementation and identify research opportunities. Finally, we validate our results on real-world networks.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*. Springer, 230–241.

[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 International Conference on Management of Data*. ACM, 349–360.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proceedings of the 23rd international conference on World wide web*. 237–248.

[4] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science* 286, 5439 (1999), 509–512.

[5] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. In *Algorithm engineering*. Springer, 19–80.

[6] Holger Bast, Stefan Funke, and Domagoj Matijević. 2006. Transit: ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge—Shortest Path*.

[7] G Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. 2009. Time-dependent contraction hierarchies. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 97–105.

[8] Anne Berry, Pinar Heggernes, and Genevieve Simonet. 2003. The minimum degree heuristic and the minimal triangulation process. In *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 58–70.

[9] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *The VLDB Journal* 21, 6 (2012), 869–888.

[10] Gianlorenzo D'angelo, Mattia D'emidio, and Daniele Frigioni. 2019. Fully dynamic 2-hop cover labeling. *Journal of Experimental Algorithmics (JEA)* 24, 1 (2019), 1–36.

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

[12] Bolin Ding, Jeffrey Xu Yu, and Lu Qin. 2008. Finding time-dependent shortest paths over large graphs. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. 205–216.

[13] Luca Foschini, John Hershberger, and Subhash Suri. 2011. On the complexity of time-dependent shortest paths. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 327–341.

[14] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *Proceedings of the VLDB Endowment* 6, 6 (2013), 457–468.

[15] Stefan Funke and Sabine Storandt. 2015. Provable efficiency of contraction hierarchies with randomized preprocessing. In *International Symposium on Algorithms and Computation*. Springer, 479–490.

[16] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46, 3 (2012), 388–404.

[17] Schultes D et al Geisberger R, Sanders P. 2008. Constraction Hierarchies: Faster and Simple Hierarchical Routing in Road Networks. *International Workshop on Experimental and Efficient Algorithms* (2008), 319–333.

[18] Andrew V Goldberg and Chris Harrelson. 2005. Computing the Shortest Path: A Search Meets Graph Theory. In *Proceedings of the annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 156–165.

[19] Ronald J Gutman. 2004. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. *Workshop on Algorithm Engineering and Experiments (ALENEX)* 4 (2004), 100–111.

[20] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.

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

[22] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. 2014. Hop Doubling Label Indexing for Point-to-Point Distance Querying on Scale-Free Networks. *Proceedings of the VLDB Endowment* 7, 12 (2014).

[23] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proceedings of the International Conference on Management of Data*. ACM, 445–456.

[24] Sungwon Jung and Sakti Pramanik. 2002. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering* 14, 5 (2002), 1029–1046.

[25] Evangelos Kanoulas, Yang Du, Tian Xia, and Donghui Zhang. 2006. Finding fastest paths on a road network with speed patterns. In *IEEE International Conference on Data Engineering*. IEEE, 10–10.

[26] Kartik Lakhotia, Rajgopal Kannan, Qing Dong, and Viktor Prasanna. 2019. Planting trees for scalable and efficient canonical hub labeling. *Proceedings of the VLDB Endowment* 13, 4 (2019), 492–505.

[27] Jianxin Li, Xinjue Wang, Ke Deng, Xiaochun Yang, Timos Sellis, and Jeffrey Xu Yu. 2017. Most influential community International Symposium on Experimental Algorithmsrch over large social networks. In *IEEE International Conference on Data Engineering*. IEEE, 871–882.

[28] Lei Li, Wen Hua, Xingzhong Du, and Xiaofang Zhou. 2017. Minimal on-road time route scheduling on time-dependent graphs. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1274–1285.

[29] Lei Li, Sibo Wang, and Xiaofang Zhou. 2019. Time-dependent hop labeling on road network. In *IEEE International Conference on Data Engineering*. IEEE, 902–913.

[30] Lei Li, Sibo Wang, and Xiaofang Zhou. 2020. Fastest path query answering using time-dependent hop-labeling in road network. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[31] Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2020. Fast query decomposition for batch shortest path processing in road networks. In *IEEE International Conference on Data Engineering*. IEEE, 1189–1200.

[32] Lei Li, Kai Zheng, Sibo Wang, Wen Hua, and Xiaofang Zhou. 2018. Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory. *The VLDB Journal* 27, 3 (2018), 321–345.

[33] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling distance labeling on small-world networks. In *Proceedings of the International Conference on Management of Data*. 1060–1077.

[34] Ye Li, Man Lung Yiu, Ngai Meng Kou, et al. 2017. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment* 11, 4 (2017), 445–457.

[35] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, Pingfu Chao, and Xiaofang Zhou. 2021. Efficient Constrained Shortest Path Query Answering with Forest Hop Labeling. In *IEEE International Conference on Data Engineering*. IEEE, 1763–1774.

[36] Hossain Mahmud, Ashfaq Mahmood Amin, Mohammed Eunus Ali, Tanzima Hashem, and Sarana Nutanong. 2013. A group based approach for path queries in road networks. In *International Symposium on Spatial and Temporal Databases*. Springer, 367–385.

[37] Rolf H Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. 2007. Partitioning graphs to speedup Dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)* 11 (2007), 2–8.

[38] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When Hierarchy Meets 2-Hop-Labeling: Efficient Shortest Distance Queries on Road Networks. In *Proceedings of the International Conference on Management of Data*. ACM, 709–724.

[39] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *Proceedings of the VLDB Endowment* 13, 5 (2020), 602–615.

[40] Simon Aagaard Pedersen, Bin Yang, and Christian S Jensen. 2020. Anytime stochastic routing with hybrid learning. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1555–1567.

[41] Simon Aagaard Pedersen, Bin Yang, and Christian S Jensen. 2020. Fast stochastic routing under time-varying uncertainty. *The VLDB Journal* 29, 4 (2020), 819–839.

[42] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Hop-constrained st Simple Path Enumeration: Towards Bridging Theory and Practice. *Proceedings of the VLDB Endowment* 13, 4 (2019), 463–476.

[43] Zichao Qi, Yanghua Xiao, Bin Shao, and Haixun Wang. 2013. Toward a distance oracle for billion-node graphs. *Proceedings of the VLDB Endowment* 7, 1 (2013), 61–72.

[44] Yongrui Qin, Quan Z Sheng, Nickolas JG Falkner, Lina Yao, and Simon Parkinson. 2017. Efficient computation of distance labeling for decremental updates in large dynamic graphs. *World Wide Web* 20, 5 (2017), 915–937.

[45] Alborzi H Samet H, Sankaranarayanan J. 2008. Scalable network distance browsing in spatial databases. *Proceedings of the International Conference on Management of Data* (2008), 43–54.

[46] Peter Sanders and Dominik Schultes. 2006. Engineering highway hierarchies. In *European Symposium on Algorithms*. Springer, 804–816.

[47] Jagan Sankaranarayanan, Houman Alborzi, and Hanan Samet. 2005. Efficient query processing on spatial networks. In *Proceedings of the 13th annual ACM international workshop on Geographic information systems*. ACM, 200–209.

[48] Jagan Sankaranarayanan, Hanan Samet, and Houman Alborzi. 2009. Path oracles for spatial networks. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1210–1221.

[49] James Fairbanks Seth Bromberger and other contributors. 2017. JuliaGraphs/LightGraphs.jl: LightGraphs. https://doi.org/10.5281/zenodo.889971

[50] Jeppe Rishede Thomsen, Man Lung Yiu, and Christian S Jensen. 2012. Effective caching of shortest paths for location-based services. In *Proceedings of the International Conference on Management of Data*. ACM, 313–324.

[51] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. 2005. Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics* 10 (2005), 1–3.

[52] Yong Wang, Guoliang Li, and Nan Tang. 2019. Querying shortest paths on time dependent road networks. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1249–1261.

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

[54] Fang Wei. 2012. TEDI: efficient shortest path query answering on graphs. In *Graph Data Management: Techniques and Applications*. IGI Global, 214–238.

[55] Victor Junqiu Wei, Raymond Chi-Wing Wong, and Cheng Long. 2020. Architecture-Intact Oracle for Fastest Path and Time Queries on Dynamic Spatial Networks. In *Proceedings of the International Conference on Management of Data*. 1841–1856.

[56] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. 2012. Shortest path and distance queries on road networks: an experimental evaluation. *Proceedings of the VLDB Endowment* 5, 5 (2012), 406–417.

[57] Bin Yang, Jian Dai, Chenjuan Guo, Christian S Jensen, and Jilin Hu. 2018. PACE: a PA th-CE ntric paradigm for stochastic path finding. *The VLDB Journal* 27, 2 (2018), 153–178.

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

[59] Ziqiang Yu, Xiaohui Yu, Nick Koudas, Yang Liu, Yifan Li, Yueting Chen, and Dingyu Yang. 2020. Distributed Processing of k Shortest Path Queries over Dynamic Road Networks. In *Proceedings of the International Conference on Management of Data*. 665–679.

[60] Dongxiang Zhang, Dingyu Yang, Yuan Wang, Kian-Lee Tan, Jian Cao, and Heng Tao Shen. 2017. Distributed shortest path query processing on dynamic road networks. *VLDB Journal* 26, 3 (2017), 399–419.

[61] Mengxuan Zhang, Lei Li, Pingfu Chao, Wen Hua, and Xiaofang Zhou. 2020. International Conference on Database Systems for Advanced Applications. In *International Conference on Database Systems for Advanced Applications*. Springer, 255–271.

[62] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic Hub Labeling for Road Networks. In *IEEE International Conference on Data Engineering*. IEEE, 336–347.

[63] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2019. Efficient Batch Processing of Shortest Path Queries in Road Networks. In *IEEE International Conference on Mobile Data Management*. IEEE, 100–105.

[64] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2020. Stream Processing of Shortest Path Query in Dynamic Road Networks. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[65] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-Hop Labeling Maintenance in Dynamic Small-World Networks. In *IEEE International Conference on Data Engineering*. IEEE, 133–144.

[66] Bolong Zheng, Kai Zheng, Xiaokui Xiao, Han Su, Hongzhi Yin, Xiaofang Zhou, and Guohui Li. 2016. Keyword-aware continuous knn query on road networks. In *IEEE International Conference on Data Engineering*. IEEE, 871–882.

[67] Andy Diwen Zhu, Hui Ma, Xiaokui Xiao, Siqiang Luo, Youze Tang, and Shuigeng Zhou. 2013. Shortest path and distance queries on road networks: towards bridging theory and practice. In *Proceedings of the International Conference on Management of Data*. ACM, 857–868.