The VLDB Journal (2017) 26:399–419
DOI 10.1007/s00778-017-0457-6

REGULAR PAPER

Distributed shortest path query processing on dynamic road networks

Dongxiang Zhang[1] · **Dingyu Yang**[2] · **Yuan Wang**[3] · **Kian-Lee Tan**[4] · **Jian Cao**[5] · **Heng Tao Shen**[1]

Received: 15 September 2016 / Revised: 31 December 2016 / Accepted: 2 February 2017 / Published online: 14 February 2017

Abstract Shortest path query processing on dynamic road networks is a fundamental component for real-time navigation systems. In the face of an enormous volume of customer demand from Uber and similar apps, it is desirable to study distributed shortest path query processing that can be deployed on elastic and fault-tolerant cloud platforms. In this paper, we combine the merits of distributed streaming computing systems and lightweight indexing to build an efficient shortest path query processing engine on top of Yahoo S4. We propose two types of asynchronous communication algorithms for early termination. One is first-in-first-out message propagation with certain optimizations, and the other is prioritized message propagation with the help of navigational intelligence. Extensive experiments were conducted on large-scale real road networks, and the results show that the query efficiency of our methods can meet the real-time requirement and is superior to Pregel and Pregel+. The source code of our system is publicly available at https://github.com/yangdingyu/cands.

Keywords Shortest path query · Dynamic road networks · Navigational intelligence · Yahoo S4

✉ Dingyu Yang
yangdy@sdju.edu.cn

Dongxiang Zhang
zhangdo@uestc.edu.cn

Yuan Wang
iseway@nus.edu.sg

Kian-Lee Tan
tankl@comp.nus.edu.sg

Jian Cao
cao-jian@sjtu.edu.cn

Heng Tao Shen
shenhengtao@hotmail.com

[1] School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China

[2] School of Electronics and Information, Shanghai Dian Ji University, Shanghai, China

[3] Department of Industrial System Engineering, National University of Singapore, Singapore, Singapore

[4] Department of Computer Science, National University of Singapore, Singapore, Singapore

[5] Department of Computer Science and Engineering, Shanghai Jiao Tong University , Shanghai, China

1 Introduction

With accelerated urbanization worldwide, the number of vehicles on the road and the need for transport is growing rapidly. The current transportation systems, with their potential inadequacy at handling fast changing traffic conditions and the optimal control of flows of vehicles, must be rectified to accommodate increasing transportation demands. In recent years, numerous attempts have been made to address these tough problems. For example, Mobile Millennium [17] was proposed as a smart traffic estimation and prediction system. The system processes millions of real-time GPS data using cloud computing and the Spark cluster computing framework. In another case, IBM InfoSphere Streams [5] demonstrates the capability of tackling the challenges of scalability, extensibility and user interaction in the domain of intelligent transportation services [16]. Both Mobile Millennium and IBM InfoSphere Streams provide traffic estimation based on real-time GPS data. With the traffic estimation and prediction models, it is still a challenging task to consider the dynamism of traffic patterns for real-time shortest path services.

Most of the previous work on index design to boost shortest path query processing in large road networks [2,6,18,32,40] is focused on accelerating query response time. They assume the network is static and adopt a carefully designed index to achieve good performance. However, these methods may not be suitable for dynamic road network as the weights of road segments are frequently updated. The family of customizable route planning (CRP) [7–9] takes care of query response time and traffic update at the same time. The index update cost is reduced to less than 1 second, and the queries can be answered promptly. However, the unprecedented success of Uber brings new challenges to these methods. On the one hand, the high query frequency requires the service system to provide good throughput. For example, the ride request of Didi in China (a service similar to Uber) has reached 14 million per day[1]. On the other hand, it is desirable for the system to handle larger-scale (e.g., world-scale) road networks. Since the customizable route planning algorithms require a considerable amount of memory and its index update process in response to traffic update will also consume considerable CPU resources, replicating the service among multiple servers may not be economical. In addition, it is trivial to handle the dynamic weight update in road network as all the replicated indexes in different servers have to be updated. Therefore, it is still of research interest and practical use to examine distributed solutions that can be deployed on elastic and fault-tolerant cloud platforms. There have been several distributed graph processing engines such as Pregel [23] and its variants [26,33,35] that can process shortest path query based on the bulk synchronous parallel (BSP) model. Although GraphLab [22] and PowerGraph [15] support asynchronous model, they were mainly designed for machine learning algorithms based on static graph and are not suitable for dynamic road networks. Hence, there is still a research gap in handling distributed shortest path queries in a large dynamic road network.

In this paper, we combine the merits of distributed streaming computing systems and lightweight indexing to support both efficient query processing and frequent edge update at the same time. We build our shortest path query processing engine on top of Yahoo S4[2], which is a distributed stream processing system. We first split the whole road network into small partitions and assign them to different machines. For each partition, we maintain shortcuts between each pair of border vertices. Each shortcut is the shortest path between the corresponding pair of border vertices within the same graph partition. Such lightweight index is easy to maintain because it only involves computation of vertices in a small local partition. The query processing starts from the partition containing the source vertex and propagates partial optimal results to the neighboring partitions until the destination vertex is reached. The partial results will also be aggregated in the master node which determines query completion.

Instead of using synchronization communication mechanism as in most graph processing systems, we propose two types of asynchronous algorithms for early termination. In the first type, the messages are sent to the neighboring graph partition in an first-in-first-out (FIFO) manner asynchronously. We set up a master processing element as a coordinator among all the workers to collect information and determine algorithm termination. Since there is no synchronization latency, the query processing is rather efficient. However, without any navigational intelligence, each partition blindly propagates messages to all the neighboring partitions, which incurs huge amounts of unnecessary communication cost. When there are multiple queries at the same time, the throughput degrades significantly.

To reduce the number of messages propagated in the network, we devise a prioritized communication mechanism with navigational intelligence to judiciously determine the order of message propagation. We propose two types of summary information as our navigational intelligence: one is a partition-level summary graph and the other consists of distance vectors to a group of sampled landmarks. The query processing is improved in two ways. First, the messages propagated to neighboring partitions that are close to the target vertex will be processed with higher priority. Second, the pruning power can be enhanced with the help of the maintained summary information. Consequently, the number of communication I/O can be dramatically reduced and the throughput of concurrent query processing is significantly improved.

Contributions. This paper is a journal extension of our previous conference paper [37]. The principle contributions in the conference paper and journal extension are summarized as follows:

1. We develop a distributed shortest path query processing engine on Yahoo S4 to support both efficient query processing and dynamic road traffic update (in conference paper).
2. We propose an asynchronous algorithm that processes and propagates messages in an FIFO manner. We devise a safe termination mechanism to guarantee the correctness and certain optimization techniques to reduce network I/O (in conference paper).
3. We propose a prioritized message propagation mechanism to replace the old FIFO one. Broadcast messages and promising intermediates results will be processed with higher priority (in journal version).
4. Inspired by the ideas of summary graph [24] and landmark techniques [13], we propose two types of naviga-

[1] http://www.recode.net/2016/6/1/11835620/didi-booking-china-apple.

[2] http://incubator.apache.org/s4/.

tional intelligence to judiciously determine the order of message propagation (in journal version).

5. We conducted extensive experiments on large-scale real road networks and compare with state-of-the-art distributed graph processing systems. In the journal version, we added more recent systems such as Pregel+ [35] in our comparison. We also added an evaluation on the index update efficiency to show that our index can support real-time traffic update.

The remainder of the paper is organized as follows. We first present the problem definition and system framework in Sect. 2. Literature review is conducted in Sect. 3. We present the graph partition scheme in Sects. 4. The FIFO-based asynchronous query processing algorithm is proposed in Sect. 5. We propose the navigational intelligence and the prioritized communication mechanism in Sect. 6. The related index update is discussed in 7. Extensive experiment results are reported in Sect. 8. Finally, Sect. 9 concludes the paper.

2 Problem statement and system framework

As a convention, we model a road network $G = (V, E, W)$ as a directed weighted graph. Each edge $e \in E$ is a road segment with a certain direction and is represented by $e = (v_i, v_j, w_e)$, where $v_i \in V$ and $v_j \in V$ are road junctions, and w_e is the average travel time to cross the edge. The average time changes over time subject to the traffic conditions. The goal of this paper is to support distributed single-source shortest path query processing over dynamic graphs.

Existing single-server index-based solutions to shortest path query processing require considerable index construction cost and are not suitable to handle large-scale and highly dynamic road network. In this paper, we study how to efficiently process time-dependent shortest path query in a distributed environment. Our system is implemented on Yahoo S4, which is a general-purpose distributed stream processing system. S4 provides friendly programming interfaces and supports an unbounded stream. The basic processing unit in S4 is called *processing element* (PE) that are customized with specific tasks and allowed to communicate with each other via asynchronous messaging. The system adopts actor model in which each PE makes local decisions in response to an incoming message (Table 1). It is worth noting most of the other streaming systems such as Twitter Storm[3] also provide such basic features and programming interface. Hence, our system can be naturally deployed on other streaming systems with the efforts of replacing the relevant processing and communication APIs.

When the system initializes, a collection of Processing Elements(PEs) are spawned in each node to handle different tasks. Two PEs can communicate in an inner-machine or intra-machine manner. In the former case, the communication is via shared memory as two related threads are located within the same machine. In the latter case, messages are sent via network. Obviously, the network I/O in the latter case is much more expensive and can easily become a bottleneck of system performance.

Table 1 Notations and symbols

G	A road network
V	Vertices in the road network
E	Edges in the road network
Q	A shortest path query
G_p	A graph partition
$\delta_{s \to t}$	The shortest distance from s to t ever found
G_s	The graph partition containing start vertex s
G_t	The graph partition containing target vertex v
$d(s, t)$	The distance of the shortest path from s to t
$d(s, G_p)$	The minimum network distance from s to any vertex in partition G_s

In our system framework, as shown in Fig. 1, we split the road network into smaller partitions (details presented in Sect. 4) and allocate as many adjacent partitions as possible to the same machine in order to save the network communication cost. The weight of the edge in the road network is the average travel time. Some shortcuts are also maintained as lightweight index to facilitate query processing. Our system accepts two types of incoming messages. The first one is a shortest path query, and the optimal path is calculated based on the current traffic conditions (details are presented in Sects. 5 and 6). The second one is the traffic update information with an edge id and the new weight. When receiving the message, our system needs to update the graph and the affected shortcuts within the same partition of the edge (details are presented in Sect. 7).

3 Related work

3.1 Shortest path query processing

Shortest path query processing has been intensively studied [1,3,6,11–13,18,25,27,32,40]. Most of the recent works focused on developing index structures to support query processing in a huge road network. For example, ALT [12] selects some vertices as *landmarks* and pre-computes the distance to these landmarks to accelerate query processing. RE [13] adds shortcuts between pairs of selected vertices to help pruning. TNR [3] uses distance tables on a subset of the vertices so that route planning can be conducted mostly by table lookup with reasonable space and preprocessing time.

[3] https://github.com/nathanmarz/storm/.

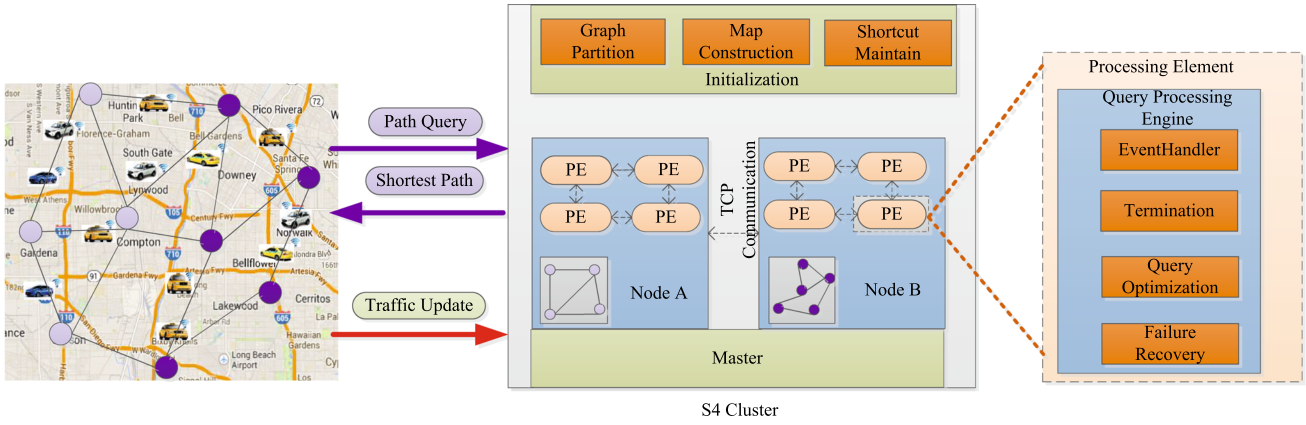


Fig. 1 System framework

In [1,6,11,18,25,40], the vertices are organized in a hierarchical structure and shortcuts between different levels of vertices are added. Although these indexes are efficient in answering shortest path queries, they assume the graph is relatively static as they require tremendous pre-computing cost to maintain the index and are not suitable for a dynamic road network with frequent traffic update. The family of customizable route planning [7–9] can attain prompt query response time and fast traffic update simultaneously. However, these algorithms are designed for a single server and not easy to be extended to a distributed environment. In [21], distributed implementation of contraction hierarchies was proposed to improve preprocessing time and support distributed query. However, the work cannot handle frequent network update. A comprehensive survey about route planning in transportation networks is available in [4].

3.2 Traffic mining

There are various mining algorithms proposed to predict travel time across a road segment. For example, T-drive [38] mines smart driving directions from the historical GPS trajectories of a large number of taxis and constructs a time-dependent landmark graph to recommend the best route. Bus scheduling based on real-time traffic and demand was studied in [30]. Another adaptive algorithm [14] is proposed to estimate traffic conditions from historical traffic data. $VTrack$ [28] can estimate the travel time along the route based on the noisy information from different sources such as WiFi. IBM also implemented a real-time traffic estimation system based on IBM InfoSphere Streams[4]. Mobile Millennium [17], developed in Spark[5], is another system supporting traffic estimation and prediction. It infers traffic conditions using GPS measurements from drivers running cell phone applications, taxicabs, and other mobile and static data sources.

3.3 Distributed graph processing systems

Distributed graph processing systems such as Pregel [23], GPS [26], Giraph[6], Blogel [33], Pregel+ [35] and Quegel [36] were developed to efficiently process large-scale web graphs and various social networks. These systems require much effort in synchronization and are designed for general graph processing algorithms. Therefore, our customized method can achieve two orders of magnitude improved performance. Although GraphLab [22] and PowerGraph [15] also adopt asynchronous methods to support scalable graph mining in an asynchronous manner, their current implementation assumes the input is a static graph and hence is not suitable for the scenario we consider in this paper. Recently, Blogel [33] was proposed to make the graph processing system “think-like-a-block” instead of “thinking-like-a-vertex.” It utilizes better graph partitioning algorithms to address the issues of skewed degree distribution, large diameter, and high density in real-world graphs. Pregel+ [34,35] is an extended work over Pregel to reduce communication cost and eliminate skewness in communication. Quegel [36] is a more recent system developed from the same research group of Pregel+. It implemented the Hub^2-Labeling approach [19] as a distributed index to support point-to-point shortest path queries. Since it requires considerable index maintenance overhead, we consider it not suitable for frequent network update and treat Pregel+ as the state-of-the-art work.

4 Graph initialization

In our model, the road network is modeled as a time-dependent graph. The edges are road segments and vertices

[4] http://www-01.ibm.com/software/data/infosphere/streams/.

[5] http://spark.incubator.apache.org/.

[6] http://giraph.apache.org/.

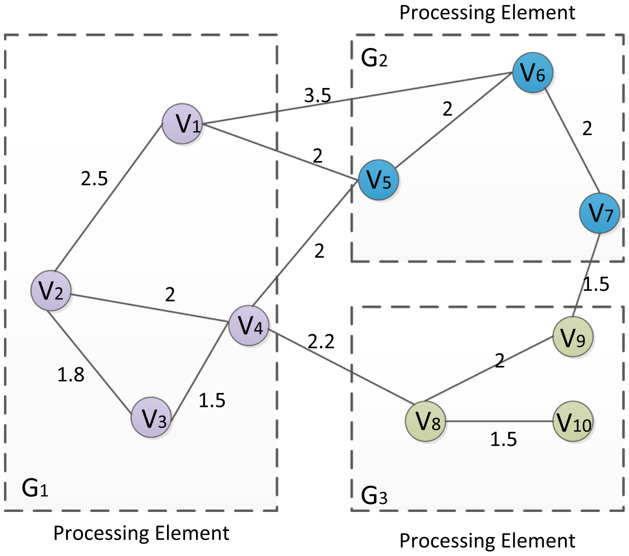


Fig. 2 An example graph with three partitions

are road junctions. The weight of each graph edge is estimated as the average travel time to cross the segment. In the following, we present some graph initialization steps before query processing.

4.1 Graph partitioning

In graph partitioning, the whole road network is first partitioned into M subgraphs using a METIS-balanced graph partitioning algorithm [20][7], where M is the number of machines in a cluster. This step is critical to performance improvement. First, the query processing time is normally determined by the slowest task and a balanced partitioning can eliminate the performance bottleneck. Second, communication between vertices in the same subgraph is done in the same machine and network I/O can be significantly reduced. In the second level, each subgraph is further split into P smaller partitions and each partition is assigned to one *processing element* so that the computing resources in each machine can be fully utilized. The number P is a user-defined parameter and is estimated in Sect. 8. The border edges crossing two partitions are stored in both partitions because our query processing algorithm requires communication among neighboring partitions.

Example 1 Figure 2 shows an example of a graph split into three partitions and assigned to different PEs. For each partition G_p, we maintain a list of pairs $L_p = \{\langle v_i, v_j, G'\rangle\}$ to store the connecting edges from G to its neighboring partitions G'. For example, the border vertices in partition G_1 are $\{v_1, v_4\}$ and we maintain for G_1 a list of pairs $L_1 = \{\langle v_1, v_6, G_2\rangle, \langle v_1, v_5, G_2\rangle, \langle v_4, v_8, G_3\rangle, \langle v_4, v_5, G_2\rangle\}$.

4.2 Graph shortcut

We maintain a collection of shortcuts to facilitate query processing. The shortcut is the shortest path of two border vertices within each partition. For any two border vertices b_1 and b_2, we pre-compute their shortest paths using Dijkstra's algorithm and store the results. In the above example, the shortest path $v_1 \rightarrow v_2 \rightarrow v_4$ with distance 4.5 is maintained in G_1. Note that this path only guarantees local optimality instead of global optimality. A better result may be found to contain vertices from other partitions. In this example, $v_1 \rightarrow v_5 \rightarrow v_4$ with distance 4 is the real shortest path from v_1 to v_4. When there is an update in the road status in this partition, the shortcuts will be refreshed to ensure the correctness of local optimality. The index update algorithm is presented in Sect. 7 and evaluated in the experimental study.

5 Dynamic SSSP query processing

Existing solutions on shortest path query processing form two extremes in terms of the index maintenance cost. The index-based approaches [3,11,40] are very efficient. However, they are difficult to adapt in a dynamic graph due to their prohibitive pre-computation costs. On the other hand, shortest path algorithm is adopted in several distributed graph processing systems [15,23,26], and it does not have any index construction cost. But these systems cannot efficiently support shortest path queries in a real-time manner for a huge road network.

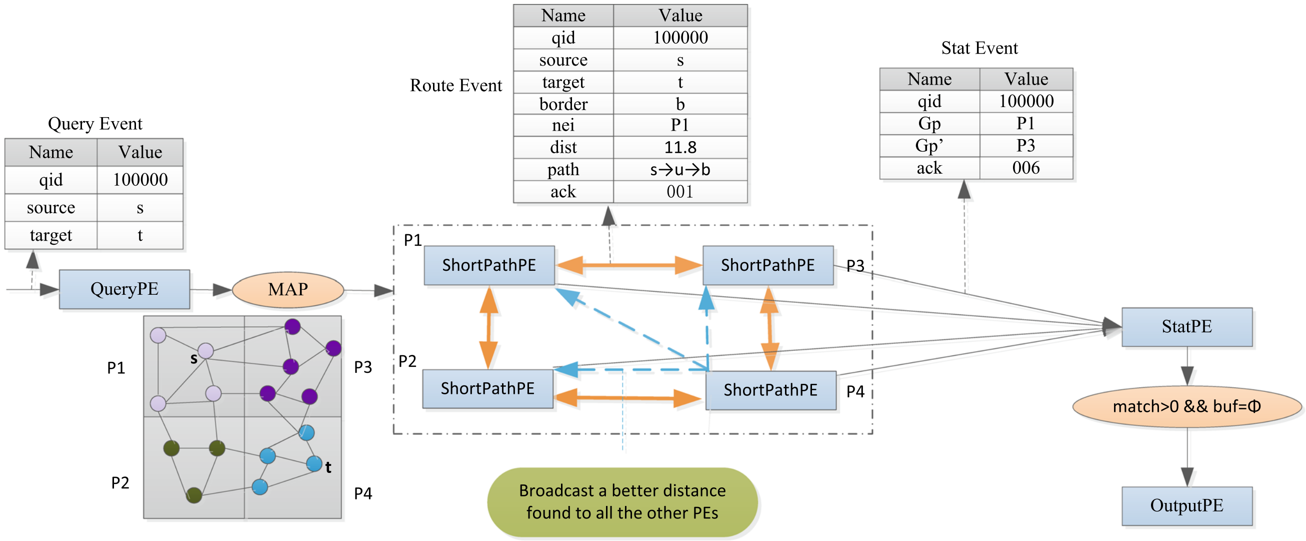
The solution proposed in this paper is a hybrid scheme between these two extremes. The road network is split into partitions that are assigned to different nodes. In each node, a number of PEs are deployed, each in charge of one partition. In each partition, we maintain some shortcuts between border vertices to facilitate query processing. The shortcuts can be used directly to pass through one partition instead of traversing all the vertices in this partition.

The query processing engine shown in Fig. 1 has the following functionalities. *EventHandler* is responsible for receiving events, emitting events to downstream PEs. A termination algorithm (*Termination*) is designed to determine whether the resultant shortest path is optimal. In each PE, query processing optimization algorithms (*Optimization*) are further applied to improve the performance.

5.1 FIFO-based query processing algorithm

The query processing strategy, depicted in Fig. 3, relies on the coordination of four types of PEs. *QueryPE* accepts all the navigation requests from vehicles. It stores the graph partition information and knows which partition contains the source vertex of the query. A new *RouteEvent* is created and

[7] Other edge-balanced graph partitioning methods can also be applied.



Query Event

Name	Value
qid	100000
source	s
target	t

Route Event

Name	Value
qid	100000
source	s
target	t
border	b
nei	P1
dist	11.8
path	s→u→b
ack	001

Stat Event

Name	Value
qid	100000
Gp	P1
Gp’	P3
ack	006

Fig. 3 PEs handle a shortest path query

Table 2 The fields contained in a *RouteEvent*

qid	A query id to uniquely identify the query
s	The source vertex of the query
t	The target vertex of the query
b	The border node receiving the message
nei	The neighboring partition which sends the message
dist	The partial best distance from *s* to *b*
path	The shortest path from *s* to *b*
ack	An acknowledgment sequence from the neighboring partition

sent to *ShortPathPE*. The event contains eight fields, as listed in Table 2. The qid, s and t are derived from the navigation request. The border node b is initialized to be the same as s, and the neighboring partition is initialized as the one containing the source vertex. The remaining fields are left empty. When the *ShortPathPE* receives the message, it starts cooperating with other *ShortPathPE*s (graph partitions) in an asynchronous manner to find the shortest path. Meanwhile, acknowledgments are sent from *ShortPathPE* to a *StatPE*. The *StatPE* collects the information, determines the termination of the query and sends the shortest path to *OutputPE* after the termination. The *OutputPE* then returns the result to the querying vehicle.

ShortPathPE is the core PE. It cooperates with neighboring partitions to find the shortest path. The idea is similar to Dijkstra’s algorithm, but without synchronization in each iteration to check which is the best vertex to visit. Our algorithm starts from the graph partition containing the source vertex, denoted by G_s. The *ShortPathPE* spreads the shortest path from s to all the border nodes in G_s to its neighboring partitions. The neighbors receive the events and process these messages *in an FIFO manner*. They improve the partial results and further disseminate them to neighbors. Finally, all these partial results will arrive at the graph partition containing the target vertex, denoted by G_t. When there is no message propagating in the network, our algorithm can terminate with the correct shortest path.

The pseudo-code of event processing in a *ShortPathPE* with regard to a graph partition G_p is illustrated in Algorithm 1. In the following, we explicitly explain how a *ShortPathPE* handles an arrival *RouteEvent*. When an event is received from a neighboring partition, the *ShortPathPE* needs to check whether the source vertex s or the target vertex t in this event is contained in G_p, which leads to four cases:

- **Case 1:** $s \in G_p \ \wedge \ t \in G_p$ (lines 5-19). If both nodes are within G_p, we can directly calculate the shortest path from s to t in the subgraph G_p. The resultant path is sent to *StatPE*. However, the algorithm cannot be terminated at this point because the true shortest path may be missed. For example, as shown in Fig. 2, the shortest path from v_1 to v_4 is $v_1 \rightarrow v_5 \rightarrow v_4$, while the Dijkstra’s algorithm in partition G_1 returns $v_1 \rightarrow v_2 \rightarrow v_4$. Therefore, we still need to send the query and partial result to the neighboring partitions even though we have obtained an initial candidate path.
- **Case 2:** $s \in G_p \ \wedge \ t \notin G_p$ (lines 14-19). If only s is in G_p, we call Dijkstra’s algorithm to calculate the shortest path from s to all the border nodes. Then, all these partial results are sent to neighboring partitions of G_p.
- **Case 3:** $s \notin G_p \ \wedge \ t \notin G_p$ (lines 20-28). If the current partition is just a bridge between s and t, the

message is updated by taking account of the shortcuts between border nodes and then forwarded to the neighbors. Such a forwarding process is efficient as the shortest path between border nodes has been pre-computed and is available in the local memory. Meanwhile, the partial distance from s to the incoming border node, denoted by $\delta_{s \to b}(G_p)$, is cached for future pruning. Another advantage is that it can be used to avoid message looping between two neighboring partitions, which generates partial results with loops.

- **Case 4:** $s \notin G_p \ \wedge \ t \in G_p$ (lines 29-39). If messages containing partial results arrive at the target graph partition, we call Dijkstra's algorithm to calculate the path and notify *StatPE* of the result.

Algorithm 1 *RouteEvent* handling algorithm in *ShortPathPE*

```
1. $s \leftarrow event.source$; $t \leftarrow event.target$; $b \leftarrow event.border$;
2. $dist(s, b) \leftarrow event$.dist;$ack = nextAck(event.qid)$;
3. initialize a StatEvents $sEvent$; $sEvent$.rev[$G_p$] $\leftarrow 1$;
4. if $s \in G_p$ then
5.   if $t \in G_p$ then
6.     dist($s$,$t$) $\leftarrow$ Dijkstra($s$,$t$)
7.     if dist($s$,$t$)< $\delta_{s \to t}$ then
8.       initialize a PathEvent $pEvent$
9.       $pEvent$.path $\leftarrow path(s, t)$
10.      $\delta_{s \to t} \leftarrow dist(s, t)$
11.      broadcast $\delta_{s \to t}$ to all the other ShortPathPE
12.      emit $pEvent$ to StatPE
13.    $T \leftarrow T \bigcup \langle qid, G_p, G_p, ack\rangle$
14.  for each $b' \in L_n.keySet()$ do
15.    run Dijkstra($s$,$b'$)
16.    initialize a $rEvent$
17.    $rEvent \leftarrow \langle qid, dist(s, b'), b', G_p, path(s, b'), ack\rangle$
18.    $combiner[G'_p][b] \leftarrow rEvent$
19.  $T \leftarrow CombineByPartition(combiner)$
20. else if $s \notin G_p$ && $t \notin G_p$ then
21.  if $dist(s, b) < distCache(s, b)$ then
22.    for each pair $\langle b', G'_p\rangle \in L_n$ do
23.      $dist(s, b') \leftarrow dist(s, b) + dist(b, b')$
24.      if $dist(s, b') < \delta_{s \to t}$ then
25.        initialize a $rEvent$
26.        $rEvent \leftarrow \langle qid, dist(s, b'), b', G_p, path(s, b')\rangle$
27.        $combiner[G'_p][b] \leftarrow rEvent$
28.    $T \leftarrow CombineByPartition(combiner)$
29. else if $s \notin G_p$ && $t \in G_p$ then
30.  if $dist(s, b) < distCache(s, b)$ then
31.    dist($b$,$t$) $\leftarrow$ Dijkstra($b$,$t$)
32.    if dist(s,b)+dist(b,t)< $\delta_{s \to t}$ then
33.      initialize a PathEvent $pEvent$
34.      $pEvent$.path $\leftarrow path(s, b) \bowtie path(b, t)$
35.      $\delta_{s \to t} \leftarrow dist(s, b) + dist(b, t)$
36.      broadcast $\delta_{s \to t}$ to all the other ShortPathPE
37.      emit $pEvent$ to StatPE
38.  $T \leftarrow T \bigcup \langle qid, G_p, G_p, ack\rangle$
39. $T \leftarrow T \bigcup \langle qid, event.nei, G_p, event.ack\rangle$
40. $sEvent.tuples \leftarrow T$
41. emit $sEvent$ to StatPE
```

5.2 Query processing optimization

Algorithm 1 not only introduces four cases of processing an arrival *ShortPathPE*, but also incorporates two optimization strategies *Message Combiner* and *Message Broadcast* to improve the processing performance.

5.2.1 Message combiner

After traversing one partition, messages are sent to neighbor partitions through border edges. For each border edge, one message will be generated and sent to the connecting neighbor partition. It is possible that a partition has multiple border vertices connecting to the same neighborhood. For example, in Fig. 2 there are two border vertices v_1 and v_4 in partition $G1$. v_1 has to send two messages to partition G_2 and v_4 sends one message to G_2 and one message to G_3. This message dissemination mechanism incurs too much communication cost in the network. To tackle this problem, we propose a *Message Combiner* technique to combine some messages together. We add a message combiner before emitting the messages to the neighboring *ShortPathPE*. If the messages in one partition are sent to the same neighbor partition, they will be merged together as one message (lines 19 and 28 in Algorithm 1). In Fig. 2, we combine the *RouteEvent*s between G_1 and G_2 as one message and then send it to G_2.

5.2.2 Message broadcast

Another optimization technique to avoid message flooding is to broadcast the shortest path to all the *ShortPathPE*s whenever a better result is found at the target partition G_t. In this way, each partition G_p maintains a variable $\delta_{s \to t}$, indicating the current best distance. The purpose is that the partitions far away from s can learn this information before receiving messages from neighbors. Then, the message propagation process can stop earlier without involving too many graph partitions. We define the distance from s to a graph partition as follows:

$$d(s, G_p) = \min_{v \in G_p.\text{border}} d(s, v) \tag{1}$$

If $d(s, G_p) > \delta_{s \to t}$, this propagation can stop. Although this condition may not be satisfied as the message transmission in the network is asynchronous, it can avoid spreading partial worse results from the source partition G_s to every other partition. The partition far away from s can dismiss the events if the partial result is found to be worse than the broadcasted result (lines 11 and 36 in Algorithm 1). Although the broadcast overhead is not cheap (the number of messages is the same as that of graph partitions), it still gains an advantage compared to the overhead caused by the communication between any two neighboring *ShortPathPE*s far from s.

If a graph partition G_p does not receive any *RouteEvent* from its neighbors, we have $d(s, G_p) \geq d(s, t)$. That means this partition is far away from source vertex s and the shortest path between s and t must not pass through this partition. In the following, we show that if a graph partition G_p does not receive any `RouteEvent` from its neighbors, we have $d(s, G_p) \geq d(s, t)$.

Lemma 1 *For any partition G_p, its cached $\delta_{s \to t}$ is at least $d(s, t)$.*

Proof If the partition does not receive a broadcast message from the destination PE, $\delta_{s \to t}$ is initialized to ∞ and thus $\delta_{s \to t} \geq d(s, t)$. Otherwise, we have found a path from s to t with distance $\delta_{s \to t}$. Since $d(s, t)$ is the distance of the shortest path, $\delta_{s \to t} \geq d(s, t)$. □

Lemma 2 *For any partition G_p, if $d(s, G_p) < d(s, t)$, G_p receives at least one `RouteEvent` from its neighbors.*

Proof Assume that the shortest path from s to G_p is $s \to \ldots u \to v$ where $u \in G'_p$ and $v \in G_p$. In other words, G'_p is the neighboring partition of G_p with a connecting edge (u, v). We know that in G'_p, we have $dist(s, u) + dist(u, v) = dist(s, G_p) < d(s, t)$. From Lemma 1, we have $d(s, t) \leq \delta_{s \to t}$. Therefore, $dist(s, u) + dist(u, v) < \delta_{s \to t}$ and a `RouteEvent` will be sent from G'_p to G_p based on Algorithm 1. □

This naturally leads to the following lemma.

Lemma 3 *If a graph partition G_p does not receive a message w.r.t. to $Q_{s \to t}$ from its neighbor, we have $d(s, G_p) \geq d(s, t)$.*

5.3 Algorithm termination mechanism

Algorithm termination is a critical issue in distributed query processing. For example, Pregel terminates when every vertex `votes to halt` in each superstep. In this paper, a solution is proposed to determine algorithm termination. We create a special PE, named *StatPE*, to receive statistics of messages sent and received (which are encapsulated in *StatEvent*) from *ShortPathPE* and determine when query processing can be terminated. A *StatEvent* contains multiple tuples in the form of $\langle qid, G_p, G'_p, ack \rangle$, each acknowledges the communication with its neighboring partition. It means the message is sent from G_p to G'_p with acknowledgment sequence ack. Two tuples T_1 and T_2 match each other if the following condition is satisfied:

$$T_1.G_p = T_2.G_p \ \wedge \ T_1.G'_p = T_2.G'_p \ \wedge \ T_1.ack = T_2.ack$$

In *StatPE*, we maintain a buffer and a match counter for each query. When a *StatEvent* arrives, we scan the tuples in the event. For each tuple, we scan the buffer to find if there is a match. If a match is found, the tuple is removed from the buffer and the match counter increases by 1. Otherwise, we insert the tuple into the buffer. The algorithm is shown in Algorithm 2. All PEs can fully utilize the CPU resources without being idle. The algorithm can be terminated immediately when the buffer is empty and $match > 0$. There is no need to vote for halt.

Note that when there is message loss, the termination condition will not be satisfied. To avoid endless waiting, we set a threshold for maximum waiting time for early termination of a failed query processing.

Algorithm 2 *StatEvent* handling algorithm in *StatPE*

```
for each tuple $T$ in StatEvent do
  for each tuple $T'$ in $buf$ w.r.t to $Q_{s \to t}$ do
    if $T$ matches $T'$ then
      remove $T'$ from $buf$
      $match \leftarrow match + 1$
  if $buf$ is empty and $match > 0$ then
    emit result to OutputPE
```

This leads to the following lemma:

Lemma 4 *Given a message propagation chain from $G_1 \to G_2 \to \ldots \to G_n$, if a set of acknowledgments from $\{G_{j_1}, G_{j_2}, \ldots\}$ arrive earlier than G_i where $j_k > i$, we can find at least one acknowledgment from G_{j_k} $(j_k > i)$ pending in the buffer.*

Proof Let $u = \min\{j_1, j_2, \ldots\}$, i.e., $u > i$ and G_u is the partition closest to G_i in the propagation chain $G_1 \to G_2 \to \ldots \to G_n$ whose acknowledgment has arrived. Since the acknowledgment of G_{u-1} has not arrived, the acknowledgment from G_u, which is $\langle qid, G_{u-1}, G_u, ack \rangle$, cannot find a match and it will be kept in the buffer as in Algorithm 2.□

Lemma 5 *If $match > 0$ and $buf = \emptyset$, all the RouteEvents have been processed.*

Proof Suppose at timepoint t, we have $match > 0$ and $buf = \emptyset$ in *StatPE* and there still exists one *RouteEvent* sending from G_p to G'_p that has not been processed. The event could be either under the transmission in network or queueing in the event buffer in G'_p. We assume that the partial result from s to G'_p in this message is $s \to v_1 \to v_2 \to \ldots \to u \to v$ and this path crosses a set of partitions $G_1 \to G_2 \to \ldots \to G_n$. In the following, we prove the lemma by contradiction.

Case 1: If there exists no partition G_i from which *StatPE* has received a tuple $\langle qid, G_i, G_{i+1}, ack \rangle$ used to acknowledge its sending a *RouteEvent* to the neighbor G_{i+1} in this path. Since $G_1 = G_s$, if there is no other message propogated except $G_1 \rightarrow G_2 \rightarrow \ldots \rightarrow G_n$, no acknowledgment has ever arrived at *StatPE* and $match = 0$. Otherwise, we can find a chain forking from one of G_i in $G_1 \rightarrow G_2 \rightarrow \ldots \rightarrow G_n$. According to Lemma 4, the buffer will not be empty.

Case 2: *StatPE* has received an acknowledgment from G_i. Suppose G_i is closest to G_n in this chain and the acknowledgment G_i has been sent to *StatPE*. Since the acknowledgment from G_{i+1} has not arrived, the one from G_i cannot be matched and it will be pending in the buffer. This contradicts with the fact that the buffer is empty. □

Theorem 1 *If* $match > 0$ *and* $buf = \emptyset$ *in StatPE,the algorithm can terminate with the correct shortest path from* s *to* t.

Proof From Lemma 5, we know that when $match > 0$ and the message buffer in *StatPE* is empty, all the *RouteEvents* have been processed and acknowledged and there is no necessary for the *StatPE* to continue waiting for message coming from *ShortPathPE*. The algorithm can terminate.

Next we prove that when the algorithm terminates, the found result is the shortest path. Let P be the real shortest path and $P = (s \rightarrow v_{s_1} \ldots) \rightarrow (v_{11} \rightarrow v_{12} \ldots) \rightarrow (v_{21} \rightarrow v_{22} \ldots) \rightarrow \ldots \rightarrow (v_{m1} \rightarrow \ldots \rightarrow t)$ which passes a sequence of m graph partitions $(G_1) \rightarrow (G_2) \rightarrow \ldots \rightarrow (G_m)$. Since all the nodes v_{ij} are on the shortest path, we have $d(s, v_{(i-1)1}) + d(v_{(i-1)1}, v_{i1}) = d(s, v_{i1}) < d(s, t) < \delta_{s \rightarrow t}$. Based on Algorithm 1, we know that each partition G_i will receive a *RouteEvent* along this path.

We know that the target partition G_m does not receive the real shortest distance $d(s, v_{m1})$ from its neighbor. Otherwise, P will be returned by our algorithm when Dijkstra's algorithm is called to compute the remaining path from v_{m1} to t. Suppose G_i $(G_i \neq G_s)$ is the first graph partition in the path such that the partial result d'_i sent from G_{i-1} to G_i is not equal to $d(s, v_{i1})$. Since in G_{i-1}, we have $dist(s, v_{i1}) < d(s, t) \leq \delta_{s \rightarrow t}$. It will send the partial result to G_i which leads to a contradiction as G_i does not receive the partial result containing the best distance from s to v_{i1}. □

5.4 Event loss and failure recovery

In S4, event loss may occur from time to time. When the event arriving rate is too high for PE to handle, the event receiving buffer will become full and some events may have to be discarded. If a *RouteEvent* is lost, the termination condition cannot be satisfied because it will always wait for the acknowledgment of this event. Similarly, the event loss in *StatPE* will also lead to the same consequence. We set a threshold for maximum waiting time P_T for a tuple in the buffer to be matched. If a tuple of acknowledgment stays in the buffer for longer than P_T, it is considered there is an event loss for this query and a message is broadcasted to notify all the graph partitions of dismissing all the events related to this query. Then, a new query with the same source and target vertex is re-submitted. If the system is overloaded at the moment, an alternative is to adopt the caching technique to find an approximate shortest path to the vehicles [29].

6 Prioritized communication mechanism

The communication mechanism for each processing element proposed in Sect. 5 is essentially in an FIFO fashion. The processing element associated with a graph partition maintains a buffer as an FIFO queue. The incoming messages are queued in the buffer and sorted by the arrival timestamp. Each time the PE picks the message on the top of the queue, processes it and moves to the next one. There are two extreme cases for the buffer: 1) When the buffer is empty, the PE turns into the state of idle, waiting for the next incoming event, and 2) when the buffer is full, the local graph partition becomes a hot spot. It means the arrival speed of incoming messages has become too high to process. Thus, the system will directly discard some of the messages without processing them. In consequence, the returned shortest path may not be accurate and this is the scenario we want to avoid in the system design.

In this section, we propose a prioritized communication mechanism to significantly reduce the incurred network I/O and improve the success rate of query processing as well as the throughput of concurrent query processing. The idea is based on the following two observations. First, there are two types of messages involved in the query processing. One is the partial results propagated from neighboring partitions. The other is the broadcast messages sent from the global $StatPE$ to notify all the PEs about a better shortest path of a query. Obviously, the broadcast message is more useful in terms of reducing the number of network I/O. If the local PE processes this type of message with higher priority, some of the partial results in the buffer w.r.t. to the same query may be pruned. Second, there could be multiple queries issued at the same time and the buffer contains partial results w.r.t. different queries. These messages should be processed in the order of their potential for a better result. The promising partial results should be extended with local path in the current partition and propagated to neighboring partitions urgently. The less promising ones can be delayed for processing in case a broadcast message with a better shortest path arrives. Then, these less promising messages may be able to be pruned directly.

Therefore, in the prioritized communication mechanism, we treat the broadcast message with the highest priority. Each time a better result of the shortest path query is broadcast, we process it immediately by updating the best distance for that query stored in the local cache and scan all the partial results in the buffer. As long as the minimum distance of complete path derived from the partial results in the buffer is already larger than the newly received distance for the same query, we can safely prune that message from the buffer. For the remaining messages, we propose a novel scoring criterion to measure its potential to lead to a better result such that more promising partial results will be handled with higher priority. In the following, we first present how to construct navigational intelligence offline for online decision making in Sect. 6.1. Based on the navigational intelligence, we propose our prioritized communication mechanism in Sect. 6.2.

6.1 Navigational Intelligence

The high-level query processing algorithm proposed in Sect. 5 can be considered as asynchronous breadth-first search by starting from the source vertex and gradually propagating to the target vertex. Without relying on any external intelligence, the propagation is essentially *blind* because it has to calculate all the possible partial results that may lead to the target node and send them to its neighbors. This incurs a significant amount of network I/O especially since all the partitions within the radius of $\delta_{s\to t}$ have to be involved in the message propagation even though some of them are far away from the target node. Our solution to this issue is to maintain navigational intelligence such that the message propagation in the network can become smarter. It has an estimated knowledge of the distance from its neighboring partitions to the target node such that the messages can be sent to those neighbors which are more likely to find a better result.

A straightforward method to estimate the distance from a graph partition to a target node is to use the Euclidean distance. In several previous work, the Euclidean distance provides a lower bound of the real network distance. Unfortunately, the geo-coordinate information of each vertex in the road network may not be always available. Many published road network datasets only provide the graph topological structure without spatial information. In this paper, we adopt the ideas of low bound estimation in [24] and [13] in the context of vertex-to-vertex distance calculation and extend them in our context of partition-to-partition distance estimation, which serves as the navigational intelligence to guide message propagation between partitions.

6.1.1 Partition-level summary graph

Our first attempt to capture the navigational intelligence is to build a partition-level summary graph based on the lower bound of the original graph G, denoted by G_L. The idea is similar to PCD [24], in which the graph is partitioned into clusters and the shortest distance between each pair of clusters is pre-computed to facilitate pruning. In G_L, the weight for each edge is the minimum possible travel time at any timestamp. Hence, it is easy to infer that the shortest path distance from s to t in G_L is the lower bound for all the snapshots of temporal graphs G_T.

Figure 4 shows an example of a partition-level summary graph. The whole network is split into seven partitions G_1, $G_2, \ldots, G_7$, each representing a vertex in the summary graph. There is a directed edge from G_i to G_j if we can find a connected edge (v_i, v_j) in the original network such that $v_i \in G_i$ and $v_j \in G_j$. The weight of (G_i, G_j) in the summary graph is set to the minimum travel time among all the connecting edges. We can use the partition-to-partition weight in the summary graph to estimate the vertex-to-vertex distance in the original network. For example, if s is located in G_1 and t is located in G_5, it is likely that the shortest path from s to t should go through partitions $G_1 \to G_2 \to G_5$ based on the knowledge of the summary graph. Hence, in the query processing, we can put higher priority to the communication message propagation along this path.

The space cost of the summary graph is affordable. The storage complexity is at most $O(M^2P^2)$ as there are $M \cdot P$ partitions as presented in Sect. 4. In addition, we can use the summary graph to improve the pruning effect. In our proposed query processing algorithm in Algorithm 1, we compare the distance in the partial result with the $\delta_{s\to t}$. If the partial distance is already no smaller than $\delta_{s\to t}$, the message can be pruned without incurring further propagation to neighboring partitions. In the improved version with summary graph, we can obtain the minimum distance from each neighboring partition to the target partition. The pruning condition in line 24 of Algorithm 1 can be replaced by

$$dist(s, b') + dist(G'_p, G_t) < \delta_{s\to t} \qquad (2)$$

where $dist(G'_p, G_t)$ is the minimum distance from G'_p to G_t in the summary graph.

Lemma 6 *Algorithm 1 is correct with the new pruning rule in Eqn. 2.*

Proof Given a partial result sent to the current partition G_p via border node b, we have $dist(s, b') = dist(s, b) + dist(b, b')$, where b' is the border node in the neighboring partition G'_p. Since $dist(G'_p, G_t)$ is the minimum distance from any node in G'_p to target partition G_t in the lower-bound network, we have $dist(b', G_t) \geq dist(G'_p, G_t)$. Since $dist(b', t) \geq dist(b', G_t)$, we know the shortest distance from b' to t is at least $dist(G'_p, G_t)$. Hence, it is impossible to derive a better shortest path from the partial result $s \to b \to b'$ if $dist(s, b') + dist(G'_p, G_t) \geq \delta_{s\to t}$. In other words, the message will only be handled when $dist(s, b') + dist(G'_p, G_t) < \delta_{s\to t}$. □

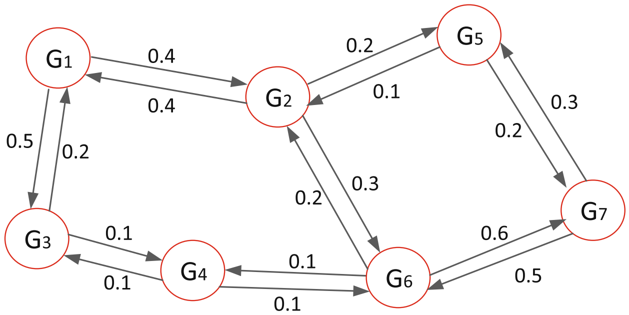
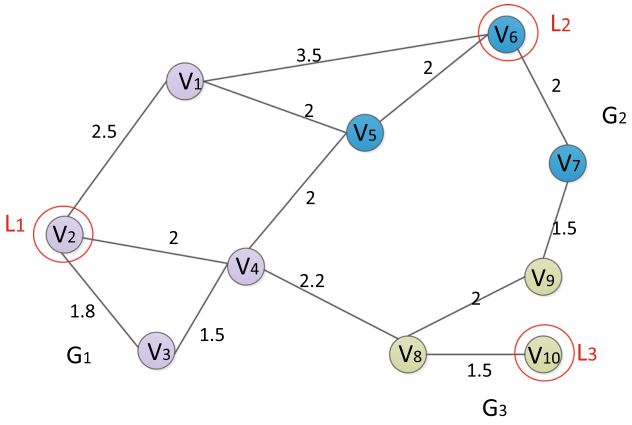


Fig. 4 An example of a partition-level summary graph



V1 : [2.5, 3.5, 7.7] V2 : [0.0, 6.0, 5.7] V3 : [1.8, 5.5, 5.2]

V4 : [2.0, 4.0, 3.7] V5 : [4.0, 2.0, 5.7] V6 : [6.0, 0.0, 7.0]

V7 : [7.7, 2.0, 5.0] V8 : [4.2, 5.5, 1.5] V9 : [6.2, 3.5, 3.5]

V10 : [5.7, 7.0, 0.0]

Fig. 5 An example of landmark-based distance vector

6.1.2 Graph landmark

Inspired by [13], an alternative representation of navigational intelligence is to randomly sample N vertices as landmarks or reference points in the lower-bound graph G_L. Given a vertex u in G_L, we maintain a N-dimensional vector in which each entry stores the distance of the shortest path from u to the associated landmark. We can use the distance vector to estimate node-to-node distance: intuitively, if the distance vectors of u and v are similar, it is likely that these two vertices are close to each other in the network. We use the previous partitioning example in Fig. 2 as a lower-bound graph to illustrate the idea of landmark-based distance vector. As shown in Fig. 5, vertices v_2, v_6 and v_{10} from different partitions are selected as landmarks, denoted by L_1, L_2 and L_3, respectively. Then, we can calculate the shortest network distance in the lower-bound graph from any vertex to these landmarks. Take v_3 as an example. Its minimum distance to landmark v_2 is 1.8. Hence, we set its first dimension in the distance vector to be 1.8. Similarly, we can calculate its shortest distance to the other landmarks and fill the distance vector. From the vector results, we can see that the three-dimensional vector of v_1 is more similar to v_2 than v_9. It verifies that the similarity between two distance vectors is positively correlated with their network distance in the graph.

Given the vectors preserving node-to-landmark distance, we can aggregate them to obtain the partition-to-landmark distance. The distance vector for a graph partition G_i is also N-dimensional. Suppose G_i contains nodes $\{v_1, v_2, \ldots, v_m\}$. The N-dimensional distance vector for G_i is $[d_1, d_2, \ldots, d_N]$, where d_j is the shortest distance from any $v \in G_i$ to the landmark L_j. We still use Fig. 5 as an example. The distance vectors for the three partitions G_1, G_2 and G_3 become

$$G_1 : [0.0,\ 3.5,\ 3.7]$$
$$G_2 : [4.0,\ 0.0,\ 5.0]$$
$$G_3 : [4.2,\ 3.5,\ 0.0]$$

We can see that the kth entry in the vector of G_i is essentially the minimum value of the kth entry among the distance vectors of vertices belonging to G_i. Then, we use the distance vector as an approximation of the network distance. Given an incoming message whose target node is t, we can estimate the similarity between each neighboring partition and the target partition containing t. The most similar neighboring partition is considered to be closer to the routing destination, and we can assign higher priority to the messages propagated to this neighbor. For example, we extend the example in Fig. 5 by adding an edge v_{10} to v_{11} which belongs to a new graph partition G_4 and suppose the edge weight is 1.0. Then, the distance vector of G_4 is [6.7, 8.0, 1.0]. Given a shortest path query from $v_2 \in G_1$ to $v_{11} \in G_4$, we know that G_1 has two neighboring partitions leading to the destination, namely $G_1 \rightarrow G_2 \rightarrow G_4$ and $G_1 \rightarrow G_3 \rightarrow G_4$. Since the distance between distance vectors (G_3, G_4) is smaller than (G_2, G_4), we consider $G_1 \rightarrow G_3 \rightarrow G_4$ as a more promising communication path which can reach the destination earlier.

In our implementation, we set N to be the same with the number of partitions $M \cdot P$. In other words, we randomly pick a vertex in each partition to form a group of landmarks. For each landmark, we run one round of Dijkstra's algorithm to find single-source shortest paths for all the vertices and estimate the average distance from the landmark to all the partitions. When a node is popped from the priority queue, we get the partition id of that node, compare the distance with the preserved minimum distance, and update it if necessary. In addition, we store the maximum distance from the landmark to any other node within the same partition to facilitate pruning. The detailed algorithm is shown in Algorithm 3. Since we call Dijkstra's algorithm P times, the running time complexity is $O(M \cdot P(|E|+|V|\log|V|))$. The space requirement for the distance vectors is $O(M^2P^2 + MP)$ because we need to maintain $M \cdot P$ distance vectors, each with dimension $M \cdot P$. In addition, we will maintain additional $O(M \cdot P)$ values to store the maximum local distance of landmarks. These values can facilitate pruning, as will be presented in the following.

Algorithm 3 Distance vector construction

```
for each partition $G_i$ do
  randomly sample a vertex $L_i$ as a landmark
for each landmark $L_i$ do
  $Q \leftarrow \{\}$; $max_i \leftarrow 0$; $dist[L_i] \leftarrow 0$
  for each vertex in graph $G_L$ do
    if $v \neq L_i$ then
      add $v$ to $Q$
      $dist[v] \leftarrow \infty$
  while $Q$ is not empty do
    pop $u$ with the minimum distance $dist[u]$
    find the partition $G_j$ containing $u$
    if $dist[u]$ is smaller than the $i$-dimension of distance vector of $G_j$ then
      update the distance vector of $G_j$
    if $j = i$ && $dist[u] > max_i$ then
      $max_i \leftarrow dist[u]$
    for each neighbor $v$ of $u$ do
      if $dist[u] + dist(u, v) < dist[v]$ then
        $dist[v] \leftarrow dist[u] + dist(u, v)$
```

Similar to the pruning rule in Eq. 2, we propose an alternative pruning condition based on the landmark distance vectors. The main idea is about how to infer the lower-bound distance from any partition G'_p to the target partition G_t. Let $[d_1, d_2, \ldots, d_N]$ and $[d'_1, d'_2, \ldots, d'_N]$ denote their distance vectors, respectively. Since we pick a landmark from each partition, let L_t be the landmark selected in partition G_t. Then, we know the minimum distance from G_p to L_t is located in the tth dimension of the distance vector and $d'_t = 0$. In Algorithm 3, we have maintained the maximum distance from a landmark to any other node within the same partition. Let max_t denote the maximum distance from L_t to any node in G_t, we have $d_t - max_t$ as the minimum distance from partition G_p to G_t in the lower-bound graph.

Lemma 7 *Let $[d_1, d_2, \ldots, d_N]$ be the distance vector for a partition G'_p, we have $dist(G'_p, G_t) \geq d_t - max_t$.*

Proof Suppose the shortest path from any node in G'_p to reach a node in G_t is from $u \in G'_p$ to $v \in G_t$ and we have $dist(G'_p, G_t) = dist(u, v)$. Since d_t is the minimum distance from any node in G'_p to t, we have $d_t \leq dist(u, t) \leq dist(u, v) + dist(v, t)$. Since $dist(u, v) = dist(G'_p, G_t)$ and $dist(v, t) \leq max_t$, we get the lower bound for the inter-partition distance $dist(G'_p, G_t)$. □

Based on Lemma 7, we can replace the pruning condition in line 24 of Algorithm 1 by

$$dist(s, b') + d_t - max_t < \delta_{s \to t} \tag{3}$$

and similarly, we can easily prove that the algorithm remains correct.

6.2 Prioritized communication mechanism

With the navigational intelligence that can be built offline, we are ready to present our new event processing algorithm for the case $s \notin G_p$ and $t \notin G_p$, which is the most common scenario among the four cases. Our objective is to significantly reduce the amount of network I/O and improve the success rate and throughput of concurrent query processing.

Instead of using a FIFO message buffer, we create m buffers where m is the number of neighboring partitions of G_p. Each buffer B_i is in charge of the message propagation to a specific neighbor N_i. The messages sent to N_i will only be buffered in B_i. On one hand, we want to process the messages with a new order to replace FIFO such that more important messages can be handled earlier. On the other hand, we need to set a waiting time limit to avoid keeping a message waiting all the time. Thus, we propose a new sorting scheme to determine the order of the messages to be processed. When a new message containing the partial result w.r.t query Q arrives, we know the destination in Q. Then, we can estimate the distance from each neighboring partition to the destination based on our stored navigational intelligence. If a summary graph is maintained, we use the partition distance in the summary graph as an estimation. If distance vectors to the selected landmarks are maintained, we calculate the similarity between the distance vectors of each neighboring partition and target partition. Eventually, we can spawn each incoming message into m messages. Each message is associated with one particular neighbor and is assigned with a ranked order. The messages with higher rank are considered to be closer to the target node and should be processed earlier. To guarantee that the messages with lower rank can also be processed in time, we split the buffer into time-based windows. Each message is hashed into a unique window based on its arrival time. The messages in a window are sorted by the ranked order, and they are processed in a window-by-window manner.

Figure 6 illustrates our data structure to support prioritized communication mechanism. Suppose the current partition is G_2 and it has three neighboring partitions G_1, G_5 and G_6 as in Fig. 4. We maintain a buffer for each partition, and the buffers are organized into time-based windows. For example, partial results about Q_1, Q_2 and Q_3 arrive in the first time window. For each query, we can sort the partitions based on their estimated distance to the target partition. The

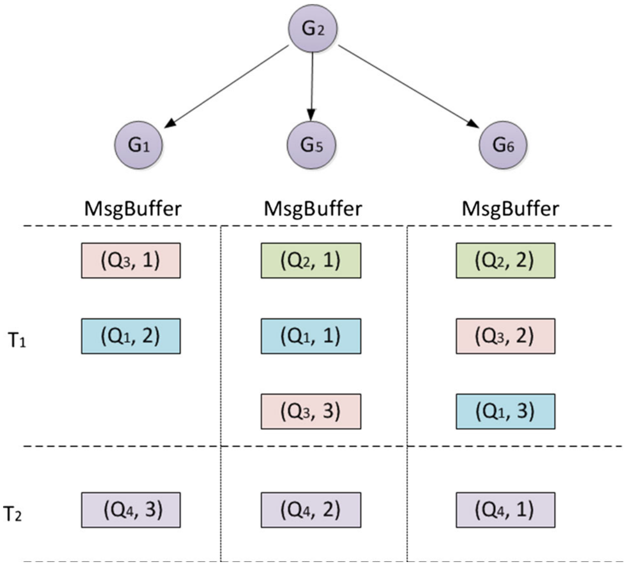


Fig. 6 An example of prioritized communication mechanism

message with higher rank will be put on top of the window buffer so as to be processed earlier. There is no $(Q_2, 3)$ in this example because the partial result of Q_3 cannot lead to a better distance through G_1 based on our pruning rules. Hence, there is no need to put the message in the buffer of G_1. Q_4 is put into the next window because its arrival time is larger than the boundary of the first window. In this case, we can guarantee that the messages $(Q_3, 3)$ and $(Q_1, 3)$ in the buffers of G_5 and G_6, respectively, can be processed earlier than messages about Q_4.

Based on the new message buffer, we propose a round-robin message processing algorithm. We start from the buffer for G_1 and pop the top message $(Q_3, 1)$ in the window. The partial result about Q_3 is further aggregated with the shortest path between local border nodes and sent to G_1. Then, we move the cursor to the buffer of G_5 and G_6 and process the top messages similarly. The procedure continues until all the messages in the first window are processed and we proceed to the next window. Such a round-robin processing algorithm exhibits two major advantages. First, the messages with more promising partial results will be processed earlier. The window-based mechanism also guarantees all the messages can be processed in time. Second, it provides better load balance by making the workload of network communication evenly distributed among the partitions. Consequently, a partition is less likely to become a hot spot, and as long as all the messages can be processed in time, we can guarantee that the success rate is close to 100 percent.

We also observed that most of the network I/O is incurred in the message propagation stage before receiving the first broadcast message containing an initial result $\delta_{s\to t}$. After that, each partition can use $\delta_{s\to t}$ for pruning and the number of messages drop dramatically. In Fig. 7, we plot the figure about how the amount of network I/O varies in different timestamps when handling a shortest path query. The x-axis is the timestamp in milliseconds, and the y-axis is the amount of network I/O. The timings for the broadcast messages are also plotted in the vertical dashing lines. We can see that the majority of the network I/O is triggered before the first broadcast. When the partitions receive the broadcast message with an initial distance for pruning, many messages with non-promising partial results are pruned, which leads to a sharp drop in network I/O.

Based on the observation, we propose an improved message propagation strategy with two stages to process each query Q. The first stage is the period before receiving the first broadcast message that contains an initial result of Q. Then, the messages are handled only when its associated rank is smaller than k. k is normally set to a very small number such as 3. The other messages will be delayed until a broadcast message about Q is received or the maximum waiting time is reached. In other words, we only process the messages that can lead to a near-optimal path. Such strategy can help reduce the network I/O incurred before the first broadcast shown in Fig. 7. In the second stage, at least a broadcast message about Q has been received and we follow the prioritized message propagation mechanism shown in Fig. 6.

It is worth noting that the aforementioned optimization mechanisms about message combiner (in Sect. 5.2.1) and message broadcast (in Sect. 5.2.2) are applicable in the context of prioritized communication mechanism. For instance, one of the improvements is that the new approach still relies on the message broadcast strategy, but sets the highest priority to the broadcast message for early termination. In consequence, many partial results in the buffer can be pruned at an earlier stage. Another improvement is that we maintain a message buffer for each neighboring partition individually. Hence, the message combiner strategy can still be applied to group the messages sent to the same neighboring partition across multiple border edges.

7 Index update

When the state of a road segment e at partition G_e changes, we need to update the shortcuts maintained for border nodes in G_e. A naive solution is to re-calculate the shortest path for each pair of border nodes, which is quite expensive. To reduce the maintenance overhead of shortcuts, different strategies are adopted in terms of the weight change and whether e appears in the original shortest path between border nodes b and b'. The pseudocode is presented in Algorithm 4 that incorporates the following cases:

– **Case 1**: $w_e \uparrow \quad \wedge \quad e \in P_{b\to b'}$ (lines 4-6). The shortest path between b and b' may not be optimal, and we call Dijkstra's algorithm to re-compute the shortest path.

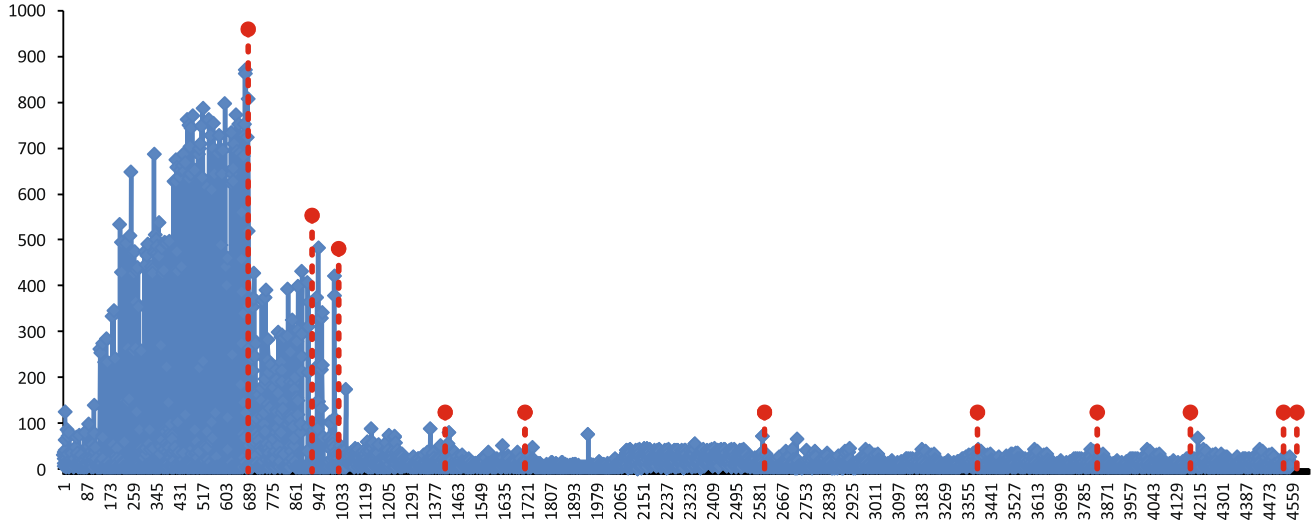


Fig. 7 Network I/O visualization for a query at different timestamps

- **Case 2**: $w_e \uparrow \;\; \wedge \;\; e \notin P_{b \to b'}$. The original shortcut is not affected by the update and is still optimal. No action is required.
- **Case 3**: $w_e \downarrow \;\; \wedge \;\; e \in P_{b \to b'}$ (lines 8-9). The shortcut between b and b' is still optimal. We simply need to update its distance because w_e decreases.
- **Case 4**: $w_e \downarrow \;\; \wedge \;\; e \notin P_{b \to b'}$ (lines 10-11). In this case, it is possible for border nodes b and b' to find a better result which passes edge e. Thus, Dijkstra's algorithm is called to guarantee the shortcut is optimal.

Algorithm 4 onEvent(*event*) to update shortcut

```
1.  for each $b \in L_n.keySet()$ do
2.    for each $b' \in L_n.keySet()$ do
3.      if $b \neq b'$ then
4.        if $event$.weight increases then
5.          if $event.edge \in P_{b \to b'}$ then
6.            $P_{b \to b'}$ =Dijkstra($b, b'$)
7.        else
8.          if $event.edge \in P_{b \to b'}$ then
9.            update $dist(b, b')$ only
10.         else
11.           $P_{b \to b'}$ =Dijkstra($b, b'$)
```

8 Experiments

We implement our shortest path processing engine on S4, which is a general-purpose distributed stream processing system under Apache incubation. S4 provides friendly programming interfaces and supports an unbounded stream. This system is event driven and query processing in each PE can be triggered periodically or by an incoming event. All of our experiments are run on a cluster with ten nodes. Each node has one Xeon E5607 Quad Core CPU (2.27GHz), 32GB memory, running CentOS 6.2. We select one node for Zookeeper and data source adapter in the customization of S4.

8.1 Comparison methods

We compare variants of our proposed asynchronous approaches with state-of-the-art distributed graph processing systems such as GPS [26] and Pregel+ [35] that can handle distributed shortest path query. The comparison with various index-based solutions including Quegel [36] is beyond the scope of the paper because these methods cannot support frequent network traffic update. In summary, the methods compared in the following experiments include:

- **GPS** [26], which is the first open-source implementation of Google's Pregel by the Stanford team.
- **Pregel+** [35], which is a more recent system developed by the CUHK team to reduce communication cost and eliminate skewness in Pregel.
- **Async**, which is the asynchronous algorithm supporting FIFO message propagation but without any optimizations.
- **Async+MB**, which is the **Async** algorithm with the optimization of *Message Broadcast* in Sect. 5.2.
- **Async+MC**, which is the **Async** algorithm with the optimization of *Message Combiner* in Sect. 5.2.
- **Aysnc+MB+MC**, which is the **Async** algorithm using both optimization techniques.
- **Summary**, which is the asynchronous algorithm supporting prioritized message propagation and using partition-level summary graph as the navigation intelligence.

- **Landmark**, which uses distance vectors to a group of selected landmarks as the navigation intelligence.

8.2 Performance metric

In the experimental setup, we issue 10 queries per second and measure the performance of shortest path query processing from the following four aspects:

- **Average query latency**. The latency of an SSSP query is the time interval between the time it arrives at the system and the time it has been processed. If a query fails, we do not count it when calculating the average query latency.
- **Throughput**. Throughput is measured by the number of queries processed in a time unit. In our setting, the time unit is one minute.
- **Network I/O**. The network I/O is measured by the amount of bytes of messages transferred in the network.
- **Success rate**. Since messages may be lost in S4 when the message arrival rate is too high, we use the query success rate as a measurement of system availability.

8.3 Experiments with synthetic workloads

Our first set of experiments are conducted in real road networks, but with synthetic traffic estimation. We use datasets derived from US road network[8] and pick 5 representative road networks of different sizes. The statistics of these networks are summarized in Table 3. The smallest dataset contains only 320K vertices, and the largest one contains more than 14 million vertices. By default, we use the CAL dataset with moderate size to test the performance.

We evaluate the performance with varying numbers of nodes in a cluster, length of path route and number of graph partitions. The parameters are listed in Table 4, with default settings in bold. We set $P_T = 60$ seconds, i.e., the query processing is considered as a failure if the result is not returned within 60 seconds.

Table 3 Road network datasets

Name	Region	Vertex number	Edge number
BAY	San Francisco Bay Area	321,270	800,172
FLA	Florida	1,070,376	2,712,798
CAL	California and Nevada	1,890,815	4,657,742
E	Eastern USA	3,598,623	8,778,114
CTR	Central USA	14,081,816	34,292,496

Table 4 Experiment Parameters

Cluster nodes	1, 2, 3, 4, 5, 6, 7, 8, **9**
Path length	[1,100], **[100,200]**, [200,300], [300,400], [400,500], [500,600]
Graph partition	10, 100, 500, **1000**, 5000, 10000

8.3.1 Construction cost of summary graph and landmark

We first present the construction cost of two types of navigational intelligence, i.e., summary graph and landmarks, before query processing. This stage is considered as preprocessing, and the construction cost with varying number of graph partitions (from 10 to 5000) is presented in Table 5. It takes around 3 minutes to build the summary graph because we only need to scan the border edges between each pair of partitions to find the lower-bound cost. In contrast, the construction time of landmark grows dramatically with increasing number of partitions. This is because for each partition, we need to calculate its shortest distance to all the sampled landmarks. If there are N partitions, the size of distance matrix is $N \times N$. Thus, it takes around an hour to build these distance vectors in the default dataset with 1000 partitions. Fortunately, the computations of distance vectors are highly independent and can be executed in parallel. With 10 parallel processes, we can reduce the construction cost of landmark to only a few minutes.

8.3.2 Increasing number of nodes in a cluster

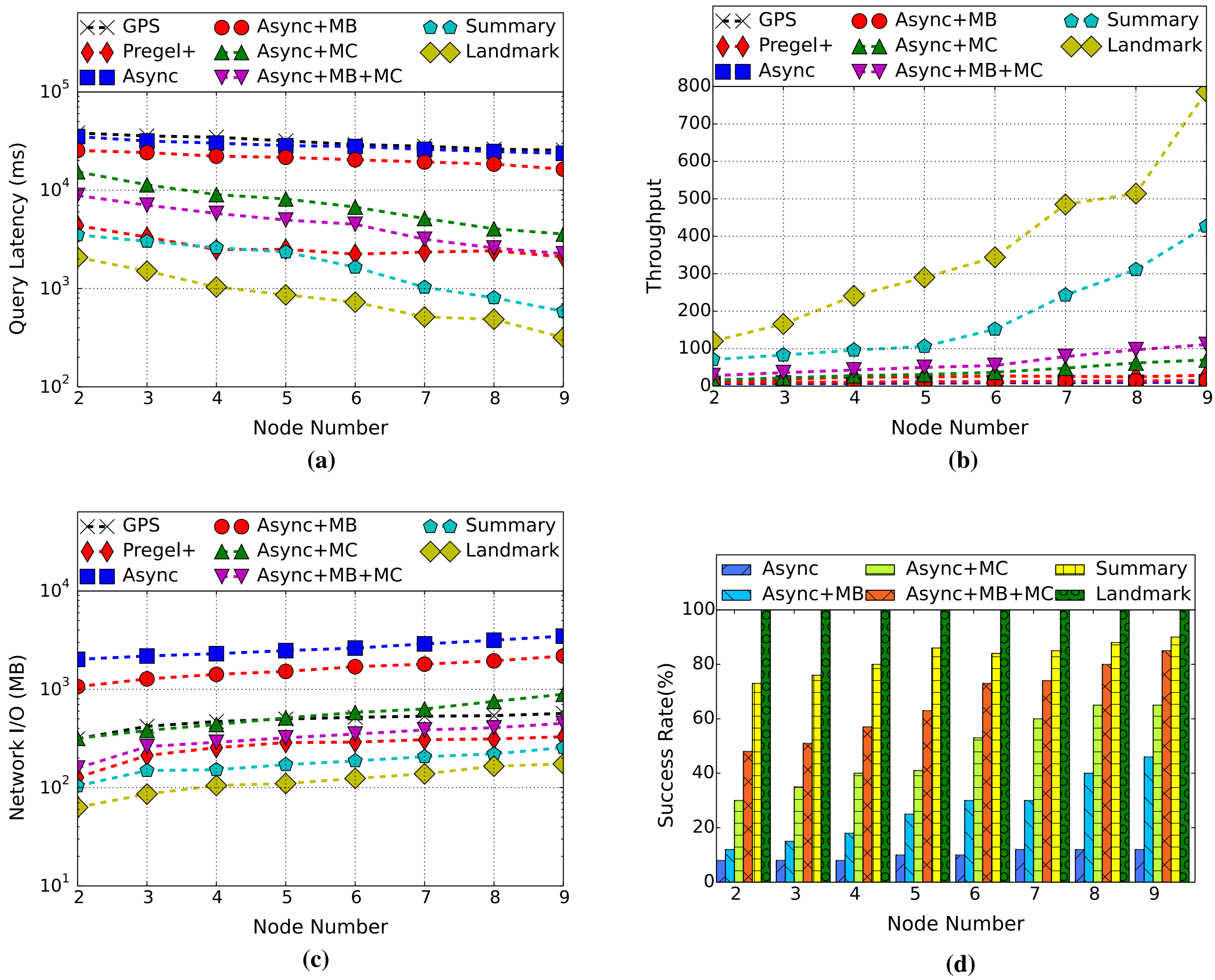
As to the performance of query processing, We first examine the performance in terms of increasing number of nodes in a cluster. The query latency, throughput, network I/O and success rate are reported in Fig. 8, leading to the following observations: 1) The running time of our Landmark solution is two orders of magnitude superior to GPS. Our Landmark method adopts asynchronous communication mechanism, and with the help of navigational intelligence, the performance can be further boosted. GPS uses bulk synchronous parallel model, and in each superstep, a vertex handles messages coming from its neighboring vertices. If there is no incoming message in the current iteration, it votes to halt. Therefore, given a query with length L (meaning the route has L edges), it requires at least L supersteps to finish a query. 2) Pregel+ significantly improves the performance of GPS because it was implemented with C++ and has a very small constant overhead per superstep. However, its running time and throughput are still inferior to the Landmark approach. 3) Our method demonstrates much better scalability than GPS and Pregel+ with increasing number of cluster nodes. The average running time drops drastically with more computing resources. When there are 9 nodes, it takes around 300ms to answer an SSSP query and supports up to 800 queries in one minute without any failure. 4) Message broadcast (MB) method can reduce the communication cost, but the effect is

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

Table 5 Construction Cost of Summary Graph and Landmark (in seconds)

	10	100	500	1000	5000
Summary graph (non-parallel)	46	118	140	169	187
Landmark (non-parallel)	175	1071	3842	6030	20097
Landmark (parallel)	30	156	415	660	2479

limited. The message combiner (MC) plays a more important role than MB in the optimization of query processing. It is very effective in reducing network I/O and query latency. It improves system throughput by 5 times. 5) The methods with navigational intelligence can further improve the performance by a wide margin, meaning our prioritized message propagation strategies are highly effective. 6) Under our high-workload experimental environment (10 queries per second), the asynchronous methods incur a large amount of network I/O within a short time and Yahoo S4 may discard messages if their incoming speed is too high to process. We can see from Fig. 8d that when facing very high workload, only the Landmark method can guarantee successful query processing because it incurs the least amount of network I/O and its communication strategy is more load balancing. The other asynchronous would fail in certain queries. GPS and Pregel+ have no such issues and always achieve 100 percent success rate.



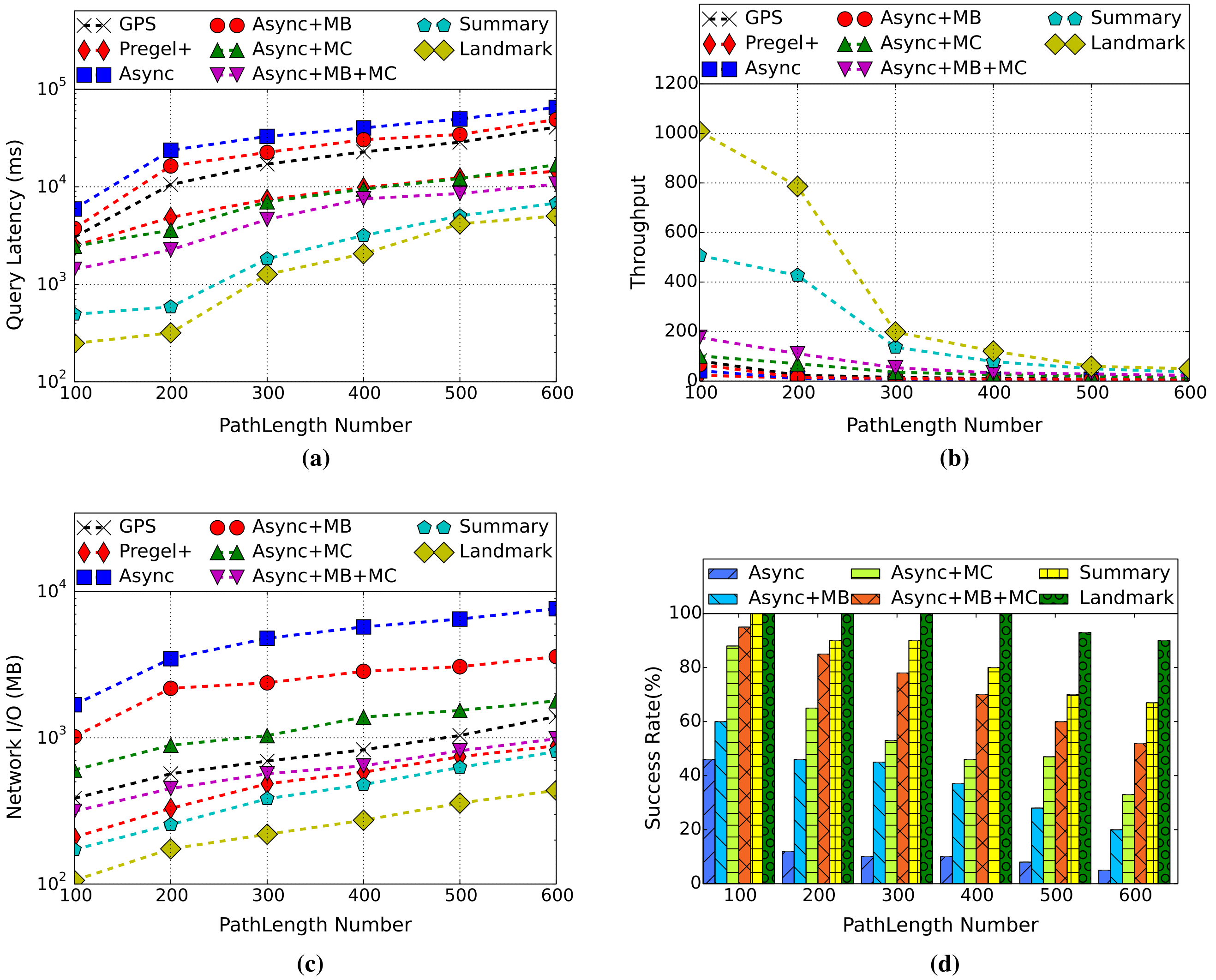
(a)

(b)

(c)

(d)

Fig. 8 Performance w.r.t. increasing nodes in a cluster. **a** Avg. query latency, **b** throughput, **c** network I/O, **d** success rate



(a)

(b)

(c)

(d)

Fig. 9 Performance w.r.t. increasing length of query result. **a** Avg. query latency, **b** throughput, **c** network I/O, **d** success rate

8.3.3 Increasing query result length

The performance of query processing is also sensitive to the query result length. We increase the length from [1, 100] to [500, 600] and show the performance in Fig. 9. The query latency grows with increasing path length and the throughput drops sharply. With longer queries, the search radius from the source node becomes larger and it is more difficult for the termination condition to take effect, resulting in a dramatically increasing number of messages propagated in the network. We can see that even the Landmark method starts to fail when the result length of each query exceeds 500.

8.3.4 Increasing number of graph partitions

The query latency and network I/O of query processing with increasing number of graph partitions are estimated in Fig. 10. The communication I/O grows significantly when the number of partitions increases from 10 to 10, 000. The query processing time first drops when the graph is split into more partitions. When the partition number is very small, it is computationally expensive to employ Dijkstra algorithm to find the distance from source vertex s to border vertices or from border vertices to target vertex t. However, when the partition number grows to thousands, the performance degrades because it incurs high communication cost and much effort is spent in sending and receiving messages. The Landmark solution always achieves the best performance as the partition number grows.

8.3.5 Performance in various datasets

We also report the average query latency and success rate in different road networks in Fig. 11. When the graph size increases, the PEs need to take charge of a larger graph partition and the performance of query processing degrades. It

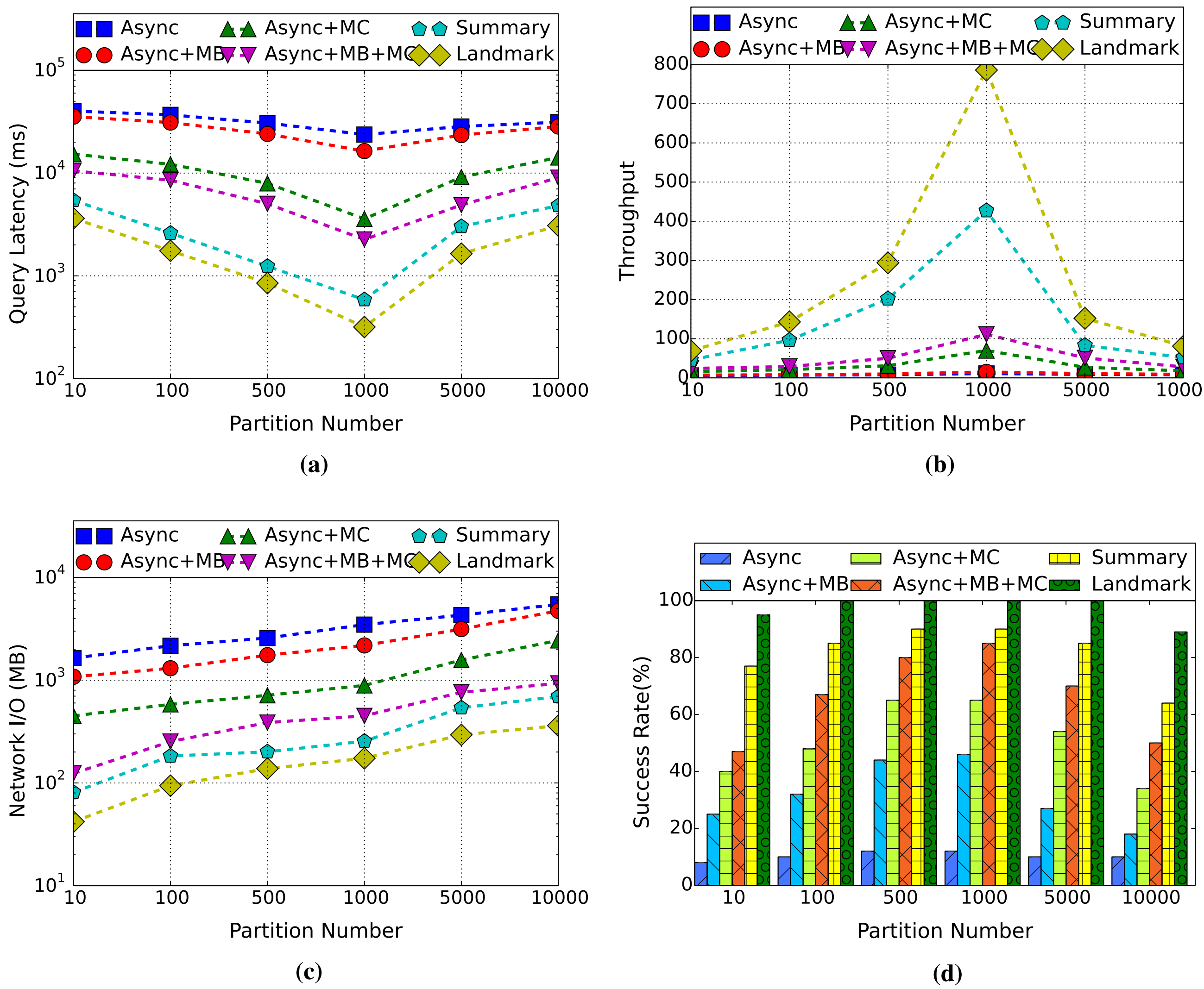


Fig. 10 Performance w.r.t. increasing number of partitions. **a** Avg. query latency, **b** throughput, **c** network I/O, **d** success rate

takes longer query processing time and the success rate also drops. However, the Landmark solution still achieves promising performance in the CTR dataset, which contains 14 million vertices. With a large number of concurrent queries, its elapsed time for one query is slightly over 1 second and the success rate is close to 100 percent.

8.4 Index update overhead

Since dynamic road network is of primary concern in this paper, we are interested to investigate the performance w.r.t. to traffic update. We report the index construction time *per partition* and index update time w.r.t. each edge weight update in Table 6. The results show that with more partitions, the partition size becomes smaller and the associated index construction time decreases as well. In our default setting, we have 1000 partitions and the response time to update the index for each traffic weight change is only 0.4 ms, which means our solution is able to support frequent traffic update in real time.

8.5 Experiments with real workloads

The real trace dataset contains the GPS trajectories of 10,357 taxis during the period of February 2 to February 8, 2008, within Beijing [10,39]. There are 154, 662 vertices and 337, 662 road segments in the Beijing road network. Each trajectory is a sequence of GPS records with timestamp and status information. These GPS records are mapped to the corresponding network edges to infer the original travel path [31]. We split a day into multiple time intervals, and for each interval, we estimate the weight of road network by the average travel time of all the taxies passing through the edge in that period. We extract the boarding and alighting

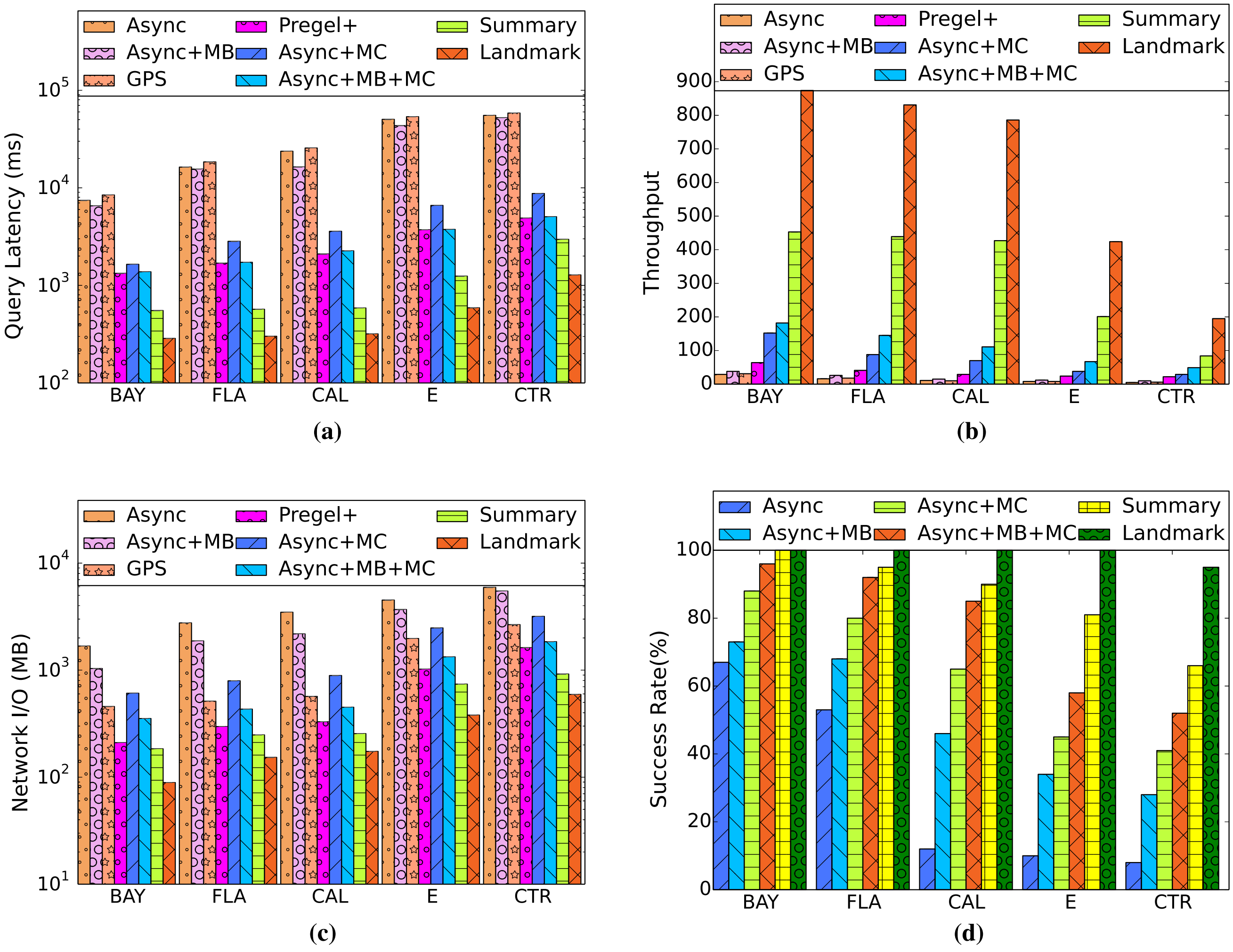


Fig. 11 Performance in different road networks. **a** Avg. query latency, **b** throughput, **c** network I/O, **d** success rate

Table 6 Index update cost w.r.t. increasing number of partitions

	10	100	500	1000	5000
Index construction time (s)	6.46	3.07	0.55	0.2	0.1
Index update time (ms)	4.5	1.7	0.4	0.1	0.08

points of passengers as s and t in query $Q_{s \to t|T}$. In this way, we generated 200 queries. The trace of these queries has an average of around 200 path lengths.

The performance of query processing is evaluated with increasing number of graph partitions and cluster nodes. The query latency and success rate are reported in Fig. 12. The best partition number is 100 and has higher success rate. The query latency decreases with more nodes in all query methods, which is similar to that for *Synthetic Workloads*. The success rate of Landmark is much higher than the other methods, exhibiting robustness in the high-workload environment.

In real navigation systems, success rate is a critical issue to guarantee a high standard of service of quality. In the synthetic experiments with 10 queries per second, the success rate of the Landmark algorithm cannot reach 100 percent when handling queries with very long path (with more than 400 road segments) or very large networks (with more than 14 million vertices). This is because the workload has exceed the processing capacity of S4 systems when deployed in a cluster with 10 machines. There are two alternative solutions to handle such extreme cases. First, we can increase the number of machines in a cluster to improve the processing power. As depicted in Figs. 8d and 12d, the success rate increases with more machines deployed. Second, we can apply the Landmark algorithm to estimate the distance of a query. If this is a very long query, approximate methods can be used [29]. In practice, the system is normally deployed to handle city-scale applications. From the experimental results of Beijing City, we can see that when the network size is moderate (around 154K vertices) and there are rarely very long queries, there

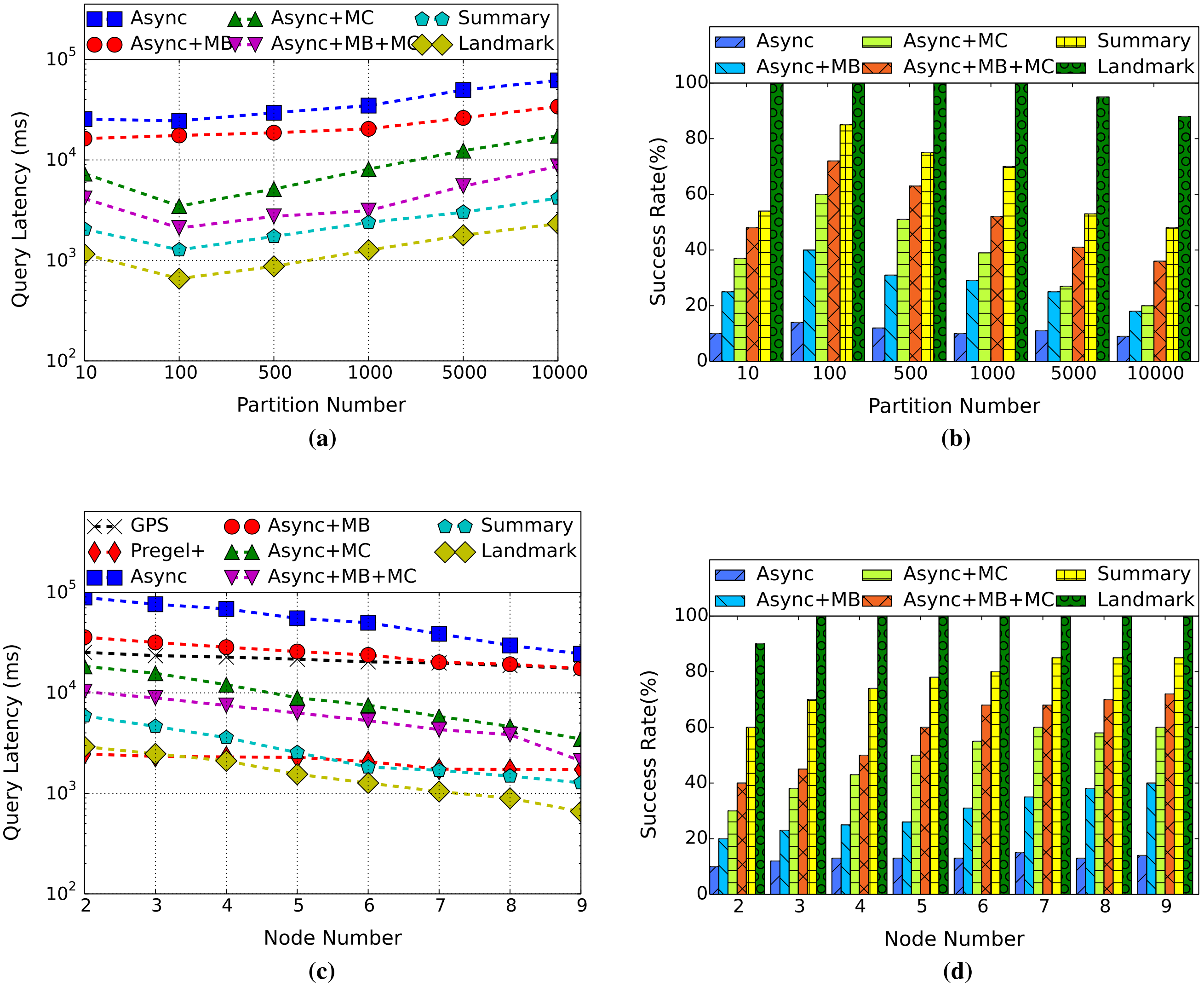


Fig. 12 Performance Evaluation of query processing on real traffic conditions. **a** Avg. query latency, **b** success rate, **c** Avg. query latency, **d** Success rate

is hardly any failure of query processing for the Landmark algorithm.

9 Conclusion and future work

In this paper, we studied the distributed shortest path query processing over dynamic road networks. In the baseline solution, we proposed an asynchronous framework that uses traditional FIFO message propagation strategy, together with message broadcast and message combiner as two optimization techniques. As improved solutions, we propose prioritized message propagation mechanism with the aid of navigation intelligence, in the form of summary graph or landmark, to further reduce communication cost. Promising query processing time and throughput have been achieved when we deploy our system on Yahoo S4 and the system is able to handle up to 800 requests within one minute.

To further improve the performance, our future work can incorporate the merits of customizable route planning and use more advanced routing algorithms. In addition, the static partitioning may not work particularly well in a dynamic and skew workload distribution. For instance, in the morning peak hours, a navigation request is likely to be from other areas to a few hot spots, and vice versa in the evening peak hours. Therefore, an adaptive load-balancing strategy is an interesting research topic worth further exploration.

Acknowledgements This work is supported in part by the National Nature Science Foundation of China under grants No. 61602087, No. 61632007 and No.61472253. It is supported by Academic Discipline Project of Shanghai Dianji University, Project Number: 16YSXK04.

References

1. Abraham, I., Fiat, A., Goldberg, AV., Werneck, RF.: Highway dimension, shortest paths, and provably efficient algorithms. In: SODA, pp. 782–793 (2010)
2. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: SIGMOD, pp. 349–360 (2013)
3. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast routing in road networks with transit nodes. Science **316**(5824), 566–566 (2007)
4. Bast, H., Delling, D., Goldberg, AV., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., Werneck, RF.: Route planning in transportation networks. arXiv:1504.05140v1 [cs.DS] (2015)
5. Biem, A., Bouillet, E., Feng, H., Ranganathan, A., Riabov, A., Verscheure, O., Koutsopoulos, H., Moran, C.: Ibm infosphere streams for scalable, real-time, intelligent transportation services. In: SIGMOD, ACM, pp. 1093–1104 (2010)
6. Cheng, J., Ke, Y., Chu, S., Cheng, C.: Efficient processing of distance queries in large graphs: a vertex cover approach. In: SIGMOD, pp. 457–468 (2012)
7. Delling, D., Werneck, RF.: Faster customization of road networks. In: Experimental Algorithms, Springer, Berlin, pp. 30–42 (2013)
8. Delling, D., Goldberg, AV., Pajor, T., Werneck, RF.: Customizable Route Planning. In: Pardalos, PM., Rebennack, S., (Eds.) Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11), Springer, Lecture Notes in Computer Science, vol. 6630, pp. 376–387 (2011)
9. Delling, D., Goldberg, AV., Pajor, T., Werneck, RF.: Customizable route planning in road networks. Transportation Science (2015). doi:10.1287/trsc.2014.0579
10. Fan, Q., Zhang, D., Wu, H., Tan, K.: A general and parallel platform for mining co-movement patterns over large-scale trajectories. PVLDB **10**(4), 313–324 (2016)
11. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: Experimental Algorithms, pp. 319–333. Springer, Berlin (2008)
12. Goldberg, AV., Harrelson, C.: Computing the shortest path: A search meets graph theory. In: SODA, pp. 156–165 (2005)
13. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Reach for a*: Efficient point-to-point shortest path algorithms. ALENEX **6**, 129–143 (2006)
14. Gonzalez, H., Han, J., Li, X., Myslinska, M., Sondag, JP.: Adaptive fastest path computation on a road network: a traffic mining approach. In: VLDB, VLDB Endowment, pp. 794–805 (2007)
15. Gonzalez, JE., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: OSDI, pp. 17–30 (2012)
16. Guerrero-Ibáñez, A., Flores-Cortés, C., Damián-Reyes, P., Andrade-Aréchiga, M., Pulido, J.: Emerging technologies for urban traffic management. Tech. rep. (2012)
17. Hunter, T., Moldovan, TM., Zaharia, M., Merzgui, S., Ma, J., Franklin, MJ., Abbeel, P., Bayen, AM.: Scaling the mobile millennium system in the cloud. In: SOCC, p. 28 (2011)
18. Jin, R., Ruan, N., Xiang, Y., Lee, VE.: A highway-centric labeling approach for answering distance queries on large sparse graphs. In: SIGMOD, pp. 445–456 (2012)
19. Jin, R., Ruan, N., You, B., Wang, H.: Hub-accelerator: Fast and exact shortest path computation in large social networks. arXiv:1305.0507v1 [cs.SI] (2013)
20. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
21. Kieritz, T., Luxen, D., Sanders, P., Vetter, C.: Distributed time-dependent contraction hierarchies. ISEA, LNCS **6049**, 83–93 (2010)
22. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning in the cloud. PVLDB **5**(8), 716–727 (2012)
23. Malewicz, G., Austern, MH., Bik, AJ., Dehnert, JC., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD, ACM, pp. 135–146 (2010)
24. Maue, J., Sanders, P., Matijevic, D.: Goal directed shortest path queries using precomputed cluster distances. ACM J. Exp. Algorithmics (2007)
25. Rice, M., Tsotras, V.J.: Graph indexing of road networks for shortest path queries with label restrictions. VLDB **4**(2), 69–80 (2010)
26. Salihoglu, S., Widom, J.: GPS: a graph processing system. In: SSDBM, pp. 22:1–22:12 (2013)
27. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: ESA, pp. 568–579, Springer, Berlin (2005)
28. Thiagarajan, A., Ravindranath, L., LaCurts, K., Madden, S., Balakrishnan, H., Toledo, S., Eriksson, J.: Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones. In: SenSys, pp. 85–98, ACM (2009)
29. Thomsen, JR., Yiu, ML., Jensen, CS.: Effective caching of shortest paths for location-based services. In: SIGMOD, pp. 313–324 (2012)
30. Wang, Y., Zhang, D., Hu, L., Yang, Y., Lee, LH.: A data-driven and optimal bus scheduling model with time-dependent traffic and demand. IEEE Trans. Intell. Transp. Syst. (99):1–10, (2017) doi:10.1109/TITS.2016.2644725
31. Wei, H., Wang, Y., Forman, G., Zhu, Y., Guan, H.: Fast Viterbi map matching with tunable weight functions. In: SIGSPATIAL GIS, pp. 613–616, ACM (2012)
32. Wu, L., Xiao, X., Deng, D., Cong, G., Zhu, A.D., Zhou, S.: Shortest path and distance queries on road networks: an experimental evaluation. PVLDB **5**(5), 406–417 (2012)
33. Yan, D., Cheng, J., Lu, Y., Ng, W.: Blogel: a block-centric framework for distributed computation on real-world graphs. PVLDB **7**(14), 1981–1992 (2014)
34. Yan, D., Cheng, J., Xing, K., Lu, Y., Ng, W., Bu, Y.: Pregel algorithms for graph connectivity problems with performance guarantees. PVLDB **7**(14), 1821–1832 (2014)
35. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In: WWW, pp. 1307–1317 (2015)
36. Yan, D., Cheng, J., Özsu, MT., Yang, F., Lu, Y., Lui, JCS., Zhang, Q., Ng, W.: Quegel: A general-purpose query-centric framework for querying big graphs. arXiv:1601.06497v1 [cs.DC] (2016)
37. Yang, D., Zhang, D., Tan, K., Cao, J., Mouël, F.L.: CANDS: continuous optimal navigation via distributed stream processing. PVLDB **8**(2), 137–148 (2014)
38. Yuan, J., Zheng, Y., Zhang, C., Xie, W., Xie, X., Sun, G., Huang, Y.: T-drive: driving directions based on taxi trajectories. In: SIGSPATIAL GIS, pp. 99–108 , ACM (2010)
39. Zheng, Y., Liu, Y., Yuan, J., Xie, X.: Urban computing with taxicabs. In: Ubicomp, pp. 89–98 (2011)
40. Zhu, AD., Ma, H., Xiao, X., Luo, S., Tang, Y., Zhou, S.: Shortest path and distance queries on road networks: towards bridging theory and practice. In: SIGMOD, pp. 857–868 (2013)