Optimal Multi-Meeting-Point Route Search

Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao

Abstract—Real-time ride-sharing applications (e.g., Uber and Lyft) are very popular in recent years. Motivated by the ride-sharing application, we propose a new type of query in road networks, called the optimal multi-meeting-point route (OMMPR) query. Given a road network G, a source node s, a target node t, and a set of query nodes U, the OMMPR query aims at finding the best route starting from s and ending at t such that the weighted average cost between the cost of the route and the total cost of the shortest paths from every query node to the route is minimized. We show that the problem of computing the OMMPR query is NP-hard. To answer the OMMPR query efficiently, we propose two novel parameterized solutions based on dynamic programming (DP), with the number of query nodes l (i.e., $l = |U|$) as a parameter, which is typically very small in practice. The two proposed parameterized algorithms run in $O(3^l \cdot m + 2^l \cdot n \cdot (l + \log(n)))$ and $O(2^l \cdot (m + n \cdot (l + \log(n))))$ time, respectively, where n and m denote the number of nodes and edges in graph G, thus they are tractable in practice. To reduce the search space of the DP-based algorithms, we propose two novel optimized algorithms based on bidirectional DP and a carefully-designed lower bounding technique. We conduct extensive experimental studies on four large real-world road networks, and the results demonstrate the efficiency of the proposed algorithms.

Index Terms—Multi-meeting-point query, ride-sharing application, dynamic programming, *A** algorithm

1 Introduction

Real-time ride-sharing, also known as dynamic carpooling, is a promising way to lower the fuel usage and mitigate traffic congestion in modern transportation systems. Recently, many real-time ride-sharing applications, such as Uber (www.uber.com) and Lyft (www.lyft.com), become more and more popular for smart phone users to plan their trips. In a typical real-time ride-sharing system, there are two types of entities: the drivers and the passengers. The passengers are capable of using their smart phones, which are equipped with geo-locating devices, to book cars by providing their location information to the system, and then the system dynamically arranges drivers to serve the passengers with shared rides.

Building such a real-time ride-sharing system is a non-trivial task. The main technical challenges come from two directions. First, how to rapidly find the driver to serve the incoming passengers' requests with shared rides. Second, after matching the driver and passengers, how to quickly determine the best route for the driver to pickup all the matched passengers. In the literature, there are several studies that aim to overcome the first challenge [1], [2], [3]. For example, in [1], [2], Ma et al. proposed a system, called T-share, to support real-time matching between drivers and passengers for the Taxi ride-sharing application. In [3], Huang et al. presented an efficient kinetic tree algorithm to support real-time driver-passengers matching with service guarantee. Both of these work mainly focus on developing practical solutions to match the drivers and passengers efficiently. In this paper, we focus on overcoming the second challenge. We assume that the driver-passengers matching procedure has been completed (e.g., one can use the method proposed in [3] for this matching task), and our goal is to devise efficient algorithms to identify the best route for the driver to serve the matched passengers. To the best of our knowledge, this is the first work that attempts to develop practical solutions to tackle the second challenge in real-time ride-sharing applications.

Intuitively, the best route in our problem should consider the costs taken by both the driver and passengers. The reason is that if we only optimize the cost taken by the driver, the best route could be a detour route, because the passengers may be geographically dispersed and the driver has to arrive at all such scattered sites to pickup passengers. Clearly, a detour route is not economical, because the detour route not only wastes passengers and drivers' time, but it also wastes vehicle fuel. For example, in Fig. 1, assume that the source and target nodes are v_1 and v_{10} respectively, and there are two passengers who are located on nodes v_4 and v_6 respectively. When we only optimize the driver's cost, the best route is $(v_1, v_4, v_3, v_6, v_8, v_{10})$, the pickup points are v_4 and v_6, and the total cost of this route is 9. Clearly, such a route is a detour route between v_1 and v_{10}. However, if the passengers at v_4 and v_6 can take a relatively small cost to reach v_1 and v_3 respectively, then the best route becomes (v_1, v_3, v_7, v_{10}), the pickup points are v_1 and v_3, and the total cost taken by the driver and passengers is 7, which is smaller than the previous route. According to this observation, we propose a flexible cost model to measure the best route. In our cost model, we allow the passengers to take a relatively small cost to arrive at the best pickup points (not necessary the original locations of the passengers), and we use a parameter α ($\alpha \in (0, 1)$) to balance the tradeoff between the costs taken by the driver and passengers. Based on this cost model, we formulate the problem as a new route

- *R.-H. Li and R. Mao are with the Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen University, China. E-mail: {rhli, mao}@szu.edu.cn.*
- *L. Qin is with the QCIS, University of Technology, Sydney, New South Wales, Australia. E-mail: Lu.Qin@uts.edu.au.*
- *J.X. Yu is with the Systems Engineering and Engineering Management, Chinese University of Hong Kong, Hong Kong, China. E-mail: yu@se.cuhk.edu.hk.*

Manuscript received 16 Mar. 2015; revised 24 Sept. 2015; accepted 7 Oct. 2015. Date of publication 19 Oct. 2015; date of current version 2 Feb. 2016. Recommended for acceptance by G. Li.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2015.2492554

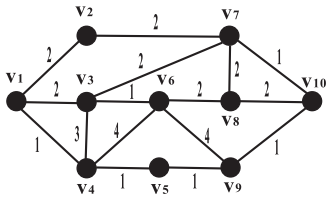


Fig. 1. Running example (the number on each edge denotes the weight of that edge).

search query problem in a road network, called the optimal multi-meeting-point route (OMMPR) query. The OMMPR query is defined on a road network G, and the input of the query contains four parameters, s, t, U, and α, where s and t denote the source node and the target node respectively, U denotes a set of query nodes (the locations of the passengers in the road network G), and $\alpha \in (0, 1)$ is a weight parameter. The OMMPR query returns a route P starting from s and ending at t, such that the weighted average cost (weighted by the parameter α) between the cost of the route P (the driver's cost) and the total cost of the shortest paths from every node in U to the route P (the total cost of the passengers) is minimized. Unlike many existing route search queries (e.g., shortest path query and its variants [4], [5], [6], [7]), the novel feature of the OMMPR query is that the optimal route not only considers the cost of the route, but it also takes the total cost of the shortest paths from the query nodes to the route into account. Moreover, the optimal route for the OMMPR query problem does not necessarily pass through the query nodes in U. The challenge of the problem is how to identify the optimal route among all the $s \sim t$ routes, the number of which can be exponentially large. Indeed, we show that computing the OMMPR query is NP-hard. To the best of our knowledge, there is no previous work on route search that can be adopted to answer the OMMPR query.

Due to the hardness of the problem, in this paper, we strive to devise practical solutions to answer the OMMPR query efficiently. Our main observation is that the number of query nodes denoted by l, i.e., $l = |U|$, is typically very small in real-time ride-sharing applications (e.g., $l \leq 7$ for a typical car), which enables us to design efficient parameterized solutions for the OMMPR query. Specifically, we first propose two novel parameterized algorithms based on dynamic programming (DP), with the number of query nodes l as a parameter. We refer to the two proposed algorithms as Basic and Grow respectively. Basic and Grow run in $O(3^l \cdot m + 2^l \cdot n \cdot (l + \log(n)))$ and $O(2^l \cdot (m + n \cdot (l + \log(n))))$ time respectively, where n and m are the number of nodes and edges in graph G. Therefore, both Basic and Grow are tractable when l is small. To further reduce the search space of our algorithms, we propose two new optimized algorithms, called Bidirect and Bidirect-Bounded, based on bidirectional DP and a carefully-designed lower bounding technique. We conduct extensive experiments on four large real-world road networks. The results show that when l is small (e.g., $l = 5$), all the proposed algorithms can answer the OMMPR query efficiently. The best algorithm Bidirect-Bounded can answer the OMMPR query with source-target distance no larger than 300 km in sub-second time in the whole USA road network, which consists of $23,947,347$ nodes and $58,333,344$ edges. This result indicates that our Bidirect-Bounded algorithm can be used for urban-scale real-time ride-sharing applications.

To summarize, our main contributions are threefold. First, we present the first study for the OMMPR query motivated by the real-time ride-sharing applications, and we prove that the problem of computing the OMMPR query is NP-hard. Second, we propose two novel DP-based parameterized solutions to answer the OMMPR query. We also devise two novel optimized algorithms to further improve the efficiency of the DP algorithms. The proposed solutions also demonstrate that the OMMPR query problem belongs to the class of fixed-parameter tractable algorithms [8], which is a particular complexity class and is tractable depending on some fixed parameters. Third, comprehensive experimental studies are conducted over four real-world datasets, and the results demonstrate the efficiency of the proposed algorithms.

The rest of the paper is organized as follows. Section 2 defines the problem and reviews the existing work related to ours. Section 3 introduces the proposed Basic and Grow algorithm. In Section 4, we explains the details of the two optimized algorithms Bidirect and Bidirect-Bounded. Section 5 reports the experiments results, and Section 6 concludes this work.

2 Problem Definition and Related Work

Below, we first formally define the OMMPR query problem and discuss the hardness of the problem. Then, we review the existing studies on the relevant problems.

Problem formulation. We model a road network as a weighted graph $G = (V, E, W)$, where V, E, and W denote the nodes set, the edges set, and the weights set respectively. Let $n = |V|$ and $m = |E|$ be the number of nodes and edges respectively. The weight $w(v_i, v_j)$ of an edge (v_i, v_j) denotes the distance between v_i and v_j.[1] In this paper, we consider G as an undirected graph. However, our major techniques can also be extended to handle directed graphs. Below, we use distance, cost, and weight interchangeably, if the context is obvious.

Denote by $P_{st} = (s, v_1, \ldots, v_{k-1}, t)$ an $s \sim t$ route (or $s \sim t$ path). Let $v_0 = s$, $v_k = t$, and $V_{P_{st}}$ be the set of nodes in P_{st}, i.e., $V_{P_{st}} = \{v_0, v_1, \ldots, v_k\}$. We define $c(P_{st}) = \sum_{(v_i, v_{i+1}) \in P_{st}} w(v_i, v_{i+1})$ as the total distance (or cost) between s and t along the route P_{st}. Note that here the route P_{st} is not necessarily a simple path. Let $U \subseteq V$ be a set of nodes. For each $u \in U$, we define

$$d(u, V_{P_{st}}) = \min_{v_i \in V_{P_{st}}} \{dist(u, v_i)\} \quad (1)$$

as the shortest-path distance from node u to the node set $V_{P_{st}}$, where $dist(u, v_i)$ denotes the distance of the shortest path from u to v_i. Clearly, $d(u, V_{P_{st}})$ signifies the shortest-path distance from node u to the route P_{st}. For convenience, we refer to $d(u, V_{P_{st}})$ as a node-route distance. Suppose that v is node in $V_{P_{st}}$ and $dist(u, v) = d(u, V_{P_{st}})$. Then, we refer to v as the meeting point (pickup point) for node u and route P_{st}, because it is the crossing node of the shortest path from u to the route P_{st}. Note that here we only consider the meeting point that is a node in the graph. Later, we will prove

1. Note that the weight of an edge can also denote the other quantities, such as travel time. Our proposed techniques are independent on how the edge weight is assigned. In addition, we assume that all the weights are nonnegative.

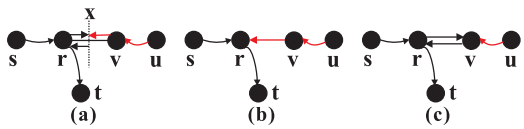


Fig. 2. Illustration of Theorem 2.3.

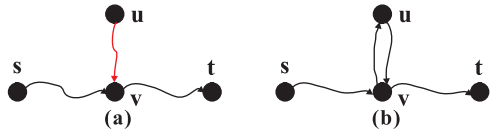


Fig. 3. Illustration of two equivalent routes when $\alpha = 1/3$.

that in our problem, the meeting point for a node u and the optimal route P_{st} must be a node in $V_{P_{st}}$, i.e., the meeting point cannot be located on any edge of the route P_{st}.

Let $d(U, V_{P_{st}})$ be the sum over all the node-route distances from the nodes in U to the route P_{st}, i.e., $d(U, V_{P_{st}}) = \sum_{u \in U} d(u, V_{P_{st}})$. Given s, t, U, and α, we define the weighted average cost function $f(P_{st})$ for a route P_{st} as

$$f(P_{st}) = \alpha \times c(P_{st}) + (1 - \alpha) \times d(U, V_{P_{st}}), \tag{2}$$

where the parameter $\alpha \in (0, 1)$ balances the tradeoff between the distance of the route P_{st} and the sum of all the node-route distances from the nodes in U to the route P_{st}. In this paper, we consider all nodes in U have the same weight $(1 - \alpha)$ in the cost function, however, our major techniques can also be extended to handle the situation when nodes in U have different weights. Clearly, in real-time ride-sharing applications, $c(P_{st})$ models the cost taken by the driver, while $d(U, V_{P_{st}})$ models the total cost taken by the passengers. The OMMPR query aims at finding the route P_{st} from a source s to a target t in a graph G such that $f(P_{st})$ is minimal. Formally, we define the OMMPR query as follows.

Definition 2.1. *Given a road network $G = (V, E, W)$, the* OMMPR *query $\mathcal{Q} = (s, t, U, \alpha)$ aims at finding the $s \sim t$ route P_{st} in G with minimum $f(P_{st})$, i.e.,*

$$\begin{aligned} &\min\ f(P_{st}) \\ &s.t. \quad P_{st} \in \mathcal{P}_{st}, \end{aligned} \tag{3}$$

where $\mathcal{P}_{st}$ denotes the set of all $s \sim t$ routes.

Example 2.2. Consider a graph shown in Fig. 1. Assume that $s = v_1$, $t = v_{10}$, $\alpha = 1/2$ and $U = \{v_6\}$. Then, the optimal route for the OMMPR query is the route $P_{st} = (v_1, v_3, v_7, v_{10})$. The weighted average cost of the optimal route P is 3, i.e., $f(P_{st}) = 3$. The meeting point for v_6 and route P_{st} is v_3, because the path (v_6, v_3) is the shortest path from v_6 to any node in $V_{P_{st}}$.

Remark 1. In real-world ride-sharing applications, the passengers may appear on the edges of the rode network G (i.e., some query points in U may not be the nodes in G). If that is the case, we can construct a new graph G' in which all query points in U are the nodes in G'. Specifically, for a query point $u \in U$ that is located on an edge (v, w), we split (v, w) into two edges which are (v, u) and (u, w) respectively. After processing all points in U, we obtain the new graph G'. Note that this procedure can be done in $O(|U|)$ time. For convenience, in this paper, we ignore such a preprocessing step, and we assume that all the query points in U appear on the nodes in G for the OMMPR query.

Below, we prove that the meeting point for any node $u \in U$ and the optimal route P_{st} must be a node in the graph.

Theorem 2.3. *Given a road network $G = (V, E, W)$ and a query $\mathcal{Q} = (s, t, U, \alpha)$. The meeting point for every node $u \in U$ and the optimal route P_{st} must be contained in V.*

Proof. We prove the theorem by contradiction. For each node $u \in U$ and the optimal $s \sim t$ route P_{st}, we assume without loss of generality that the meeting point of u and P_{st}, denoted by x, is located on an edge (r, v) as illustrated in Fig. 2a. Then, we have $f(P_{st}) = \alpha(c(P_{sr}) + c(P_{rt}) + 2w(r, x)) + (1 - \alpha)(dist(u, v) + w(v, x))$. To get a contradiction, we construct two alterative routes P^1_{st} and P^2_{st} which are illustrated in Figs. 2b and 2c respectively. The meeting point for u and P^1_{st} is the node r, and the meeting point for u and P^2_{st} is the node v. Then, we have $f(P^1_{st}) = \alpha(c(P_{sr}) + c(P_{rt})) + (1 - \alpha)(dist(u, v) + w(v, r))$, and $f(P^2_{st}) = \alpha(c(P_{sr}) + c(P_{rt}) + 2w(r, v)) + (1 - \alpha)(dist(u, v))$. We can derive that $f(P_{st}) - f(P^1_{st}) = (3\alpha - 1)w(r, x)$ and $f(P_{st}) - f(P^2_{st}) = (1 - 3\alpha)w(v, x)$. Thus, we have $f(P_{st}) - \min\{f(P^1_{st}), f(P^2_{st})\} = \max\{(3\alpha - 1)w(r, x), (1 - 3\alpha)w(v, x)\} \geq 0$. This result indicates that we can always construct a route (either P^1_{st} or P^2_{st}) that has no higher cost than that of the route P_{st}, which contradicts to that P_{st} is the optimal route. This completes the proof. □

Based on Theorem 2.3, we only need to consider the nodes in the graph to compute the OMMPR query. Below, we show the hardness of computing the OMMPR query.

Hardness. Intuitively, directly searching for the optimal route for the OMMPR query is impractical, because the number of $s \sim t$ routes is exponentially large. We show that computing the optimal route for the OMMPR query is NP-hard.

Theorem 2.4. *The problem of computing the* OMMPR *query is NP-hard.*

Proof. We show a reduction from the $s \sim t$ path Traveling Salesman Problem (TSP), which is known to be NP-hard [9], [10]. Given a complete graph $G = (V, E, W)$ with metric edge cost $W : E \rightarrow R^+$, a source node s, a target node t, and a set of nodes $U = V \backslash \{s, t\}$, the $s \sim t$ path TSP aims to find the shortest path that starts from s, ends at t, and passes through all the nodes in U [9]. It should be noted that the optimal path of the $s \sim t$ path TSP must visit all the nodes in U exactly once. Below, we show that given any $s \sim t$ path TSP query $\tilde{\mathcal{Q}} = (s, t, U)$, it is equivalent to the OMMPR query $\mathcal{Q} = (s, t, U, \alpha = 1/3)$. Specifically, for any $s \sim t$ route P_{st} and a node $u \in U$, we assume that $dist(u, v)$ is the shortest-path distance from node u to the route P_{st} as illustrated in Fig. 3a. Let $\tilde{P}_{st}(u)$ be the route $(s \sim v \sim u \sim v \sim t)$ as illustrated in Fig. 3b. Note that $\tilde{P}_{st}(u)$ is a $s \sim t$ route that passes through u. Clearly, the cost of the route $\tilde{P}_{st}(u)$ equals $c(P_{st}) + 2 \times dist(u, v)$. Recall that given $\alpha = 1/3$, the weighed average cost regarding to P_{st} and u, denoted by $f(P_{st}, u)$, is $\frac{1}{3}c(P_{st}) +$

$\frac{2}{3}dist(u,v)$ (see Eq. (2)). Obviously, when $\alpha = 1/3$, the cost of the route $\tilde{P}_{st}(u)$ is three times $f(P_{st},u)$. Moreover, it is easy to verify that this result also holds for any $s \sim t$ route P_{st} and all the nodes $u \in U$. That is to say, the cost of any $s \sim t$ route that passes through all the nodes in U, denoted by $\tilde{P}_{st}(U)$, is three times $f(P_{st},U)$, where $f(P_{st},U)$ is the weighed average cost regarding to P_{st} and U given that $\alpha = 1/3$. Since the $s \sim t$ path TSP query $\tilde{\mathcal{Q}}$ aims at finding the $s \sim t$ route $\tilde{P}_{st}(U)$ with minimum cost, it is equivalent to minimize $f(P_{st},U)$ $(\alpha = 1/3)$ over all $s \sim t$ routes, which is exactly the goal of the OMMPR query $\mathcal{Q} = (s,t,U,\alpha = 1/3)$. □

Related work. The OMMPR query is closely related to the optimal sequenced route (OSR) query in road networks which is independently proposed in [11] and [12], and generalized in [13], [14], [15]. As defined in [12], the OSR query aims to find a route of minimum distance starting from a source node and passing through several typed nodes in a specific sequence imposed on the types of the nodes, and then ending at a target node. The OSR query is different from the OMMPR query in three aspects. First, in OMMPR query, the nodes are without any type information. Second, unlike the OSR query, the OMMPR query does not impose a type-sequence constraint on the optimal route. Third, for the OSR query, the optimal route must pass through the specific typed nodes, whereas for the OMMPR query, the optimal route does not necessarily pass through the specific nodes. For example, reconsider the graph shown in Fig. 1. Assume that the source and target nodes are v_1 and v_{10} respectively, $\alpha = 1/2$, and the specific node is v_6 (i.e., $U = \{v_6\}$). Then, the answer for the OSR query is the route $(v_1, v_3, v_6, v_8, v_{10})$, whereas the optimal route for the OMMPR query is the route (v_1, v_3, v_7, v_{10}). Due to such dramatic differences, the previous techniques for the OSR query [11], [12], [16] cannot be used for the OMMPR query. Another related but different problem is the keyword-aware optimal route (KOR) search problem proposed in [17]. The KOR query aims at finding the best $s \sim t$ route such that the route passes through the nodes that covers all the given keywords, and it simultaneously satisfies some pre-defined constraints [17]. Clearly, by definition, the OMMPR query is fundamentally different from the KOR query, thus the techniques proposed in [17] cannot be applied to our problem.

Our work is also closely related to the so-called ride-sharing query problem studied in [18], [19]. The goal of the ride-sharing query in [18], [19] is to find the optimal $s \sim t$ detour route that includes a sub-route $s' \sim t'$ where s' and t' are specified in the query. Clearly, the optimal $s \sim t$ detour route for the ride-sharing query passes through the given nodes s' and t', whereas the answer for our problem does not necessarily pass through the query nodes. Owing to this fundamental difference, the techniques established in [18], [19] cannot be used for our problem.

Another related problem is the optimal meeting point (OMP) query problem in road networks [20], [21], where the query is a set of nodes and the goal is to find a gathering point such that the total cost of all the query nodes reaching the gathering point is minimized. The OMP query is also fundamentally different from our OMMPR query. On the one hand, the OMP query aims to find a gathering point, whereas the OMMPR query aims to find an $s \sim t$ route. On the other hand, in our problem, the objective cost function includes two parts which are the distance of the route and the total cost of all the node-route distances, whereas in OMP query, the cost function only depends on the distances of the query nodes to the gathering node. Due to these differences, the existing solutions for the OMP query cannot be used for the OMMPR query. Moreover, the authors in [21] also show that the gathering point of the OMP query must be a node in the road network by using a proof that is different from ours. In addition, it is worth mentioning that the shortest path between the source and the target is clearly not the optimal route for the OMMPR query, thus the Dijkstra algorithm [22] as well as many previous index-based solutions [4], [5], [6], [7] for the shortest path query cannot be applied to our problem.

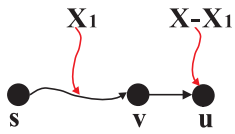


Fig. 4. Illustration of the definition of $f(u, X_1, X - X_1)$, where $s \sim v$ is the optimal route starting from s and ending at v, regarding to the set X_1.

3 Optimal Solution by DP

As shown in the previous section, the OMMPR query problem is NP-hard, thus it is impossible to devise a polynomial-time algorithm to solve it unless P=NP. In this section, we propose several novel fixed-parameter tractable [8] algorithms to solve our problem based on the technique of dynamic programming. Let l be the number of elements in U. We show that the time complexity of the proposed fixed-parameter tractable algorithms is $O(f(l)g(m,n))$, where $f(l)$ is an exponential function of l (e.g., $f(l) = 2^l$ or $f(l) = 3^l$) and $g(m,n)$ is a near-linear function with respect to the graph size (e.g., $g(m,n) = m + n(l + \log n)$). As a result, when the parameter l is small, our algorithm is very efficient and it can be scalable to handle very large graphs. It is important to note that for the OMMPR query problem, the size of the road network may be very large, whereas the number of elements in U is generally very small (e.g., $l = |U| \leq 7$). Thus, our fixed-parameter tractable algorithms are very efficient in practice, which is also confirmed in our experiments.

3.1 The Basic DP Algorithm

We propose a basic dynamic programming algorithm in which a state is represented by (u, X), where u is the end-node of a route and X is a subset of U. The DP algorithm finds the optimal route by expanding the end-node of a route and the subset X until the end-node reaches t and $X = U$. Let $f(u, X)$ be the weighted average cost of the optimal multi-meeting point route P_{su} starting from s and ending at u, with a query node set X, i.e., $f(u,X) = \alpha \times c(P_{su}) + (1-\alpha) \times d(X, V_{P_{su}})$. Then, we define $f(u, X_1, X - X_1)$ as the optimal weighted average cost of an $s \sim u$ route P_{su} such that all the meeting points for the nodes in X_1 and the route P_{su} are located in $V_{P_{su}} - \{u\}$ and all the meeting points for the nodes in $X - X_1$ and the route P_{su} coincide with u, where '$-$' denotes the set difference operator. Fig. 4 illustrates the definition of $f(u, X_1, X - X_1)$.

Formally, $f(u, X_1, X - X_1)$ can be computed by the following equation

$$f(u, X_1, X - X_1) = \min_{(v,u)\in E}\{f(v, X_1) + \alpha \times w(v, u) + (1 - \alpha) \times \sum_{x\in X - X_1} dist(x, u)\}. \quad (4)$$

We explain Eq. (4) as follows. For the partition $(X_1, X - X_1)$ of X, the optimal $s \sim u$ route can be obtained by expanding an edge (v, u) from the optimal sub-route $s \sim v$ and then taking the minimum cost over all $(v, u) \in E$ (Fig. 4). Based on $f(u, X_1, X - X_1)$, the weighted average cost of OMMPR starting from s and ending at u, regarding to X, is given by

$$f(u, X) = \min_{X_1\subseteq X}\{f(u, X_1, X - X_1)\}. \quad (5)$$

For convenience, in the rest of the paper, we use the cost of the OMMPR to denote its weighted average cost when the context is obvious. In Eq. (5), the cost of OMMPR is obtained by taking the minimum cost over all the partitions of X.

We set the initial state of (u, X) as $(s', \emptyset)$, where s' is a dummy node which links the source node s with a zero-weight edge (s', s). For the initial state $(s', \emptyset)$, we have $f(s', \emptyset) = 0$. To answer the OMMPR query, we can calculate $f(t, U)$ based on Eqs. (4) and (5) using dynamic programming with initial state $(s', \emptyset)$.

Algorithm 1. Basic(G, U, s, t)

Input: $G = (V, E, W)$, node set U, α, source node s, and target node t.
Output: the minimum cost.
1: $\mathcal{Q} \leftarrow \emptyset$; $\mathcal{D} \leftarrow \emptyset$;
2: Add a dummy node s' and an edge (s', s) with weight 0 into G.
3: $\mathcal{Q}.push((s', \emptyset), 0)$;
4: **while** $\mathcal{Q} \neq \emptyset$ **do**
5: $((v, X), cost) \leftarrow \mathcal{Q}.pop()$;
6: **if** $v = t$ **and** $X = U$ **then return** $cost$;
7: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(v, X)\}$;
8: **for all** $(v, u) \in E$ **do**
9: **for all** $X' \subseteq U - X$ **do**
10: $cost' \leftarrow cost + \alpha \times w(v, u) + (1 - \alpha) \times \sum_{x\in X'} dist(x, u)$;
11: update$(\mathcal{Q}, \mathcal{D}, (u, X \cup X'), cost')$;
12: **return** $+\infty$;
13: **Procedure** update$(\mathcal{Q}, \mathcal{D}, (v, X), cost)$
14: **if** $(v, X) \in \mathcal{D}$ **then return**;
15: **if** $(v, X) \notin \mathcal{Q}$ **then** $\mathcal{Q}.push((v, X), cost)$;
16: **if** $cost < \mathcal{Q}.cost((v, X))$ **then** $\mathcal{Q}.update((v, X), cost)$;

The basic algorithm. We present a Dynamic Programming algorithm based on the best-first strategy [22] to compute the OMMPR query. We refer to this DP algorithm as Basic, and outline it in Algorithm 1. Specifically, in Algorithm 1, we define (v, X) as a state, and represent it as a tuple $((v, X), cost)$, where $cost = f(v, X)$ denotes the cost of the OMMPR starting from s and ending at v, with query node set X. Algorithm 1 maintains a priority queue $\mathcal{Q}$, and each element of $\mathcal{Q}$ is a tuple $((v, X), cost)$. In $\mathcal{Q}$, the priority of each element $((v, X), cost)$ is $cost$, and the minimum $cost$ element is maintained as the top element of $\mathcal{Q}$. The queue $\mathcal{Q}$ has three operators which are pop, $push$, and $update$. The pop operator dequeues the element with minimum $cost$ from the queue. The $push$ operator inserts an element into the queue. The $update$ operator updates the $cost$ (i.e., priority) of an element in the queue, and then maintains the priority queue. In addition, Basic also maintains a set $\mathcal{D}$ to record all the tuples $((v, X), cost)$ such that the optimal route starting from s and ending at v, with query node set X, has been calculated.

The algorithm first initializes $\mathcal{Q}$ and $\mathcal{D}$ to be $\emptyset$ (line 1). Then, Basic adds a dummy node s' and an edge (s', s) with weight 0 into the graph G, and then inserts the tuple $((s', \emptyset), 0)$ into $\mathcal{Q}$ (lines 2-3). While $\mathcal{Q}$ is not empty, Basic repeatedly pops the top element $((v, X), cost)$ from $\mathcal{Q}$ based on the best-first strategy, and then expands the end-node of the current route and the set X until it finds the optimal $s \sim t$ route (lines 4-11). Note that in each expansion (lines 8-11), Basic updates the $cost$ of the expanded state $(u, X \cup X')$ by invoking the procedure update (lines 13-16). In line 16, $\mathcal{Q}.cost((v, X))$ gets the $cost$ of the element $((v, X), cost)$, and $\mathcal{Q}.update((v, set), cost)$ updates the $cost$ of the state (v, X). In line 6, when the top element of $\mathcal{Q}$ is (t, U), the algorithm terminates, because (t, U) cannot be expanded, and the $cost$ of (t, U) is minimum in the queue, thus it is the optimal cost for the OMMPR query. Theorem 3.1 shows the correctness of Algorithm 1.

Theorem 3.1. *Algorithm 1 correctly computes the optimal cost for the* OMMPR *query.*

Proof. The proof can be obtained by the induction on $|X|$. We omit the details due to space limit. □

The following example illustrates how Algorithm 1 works.

Example 3.2. Consider a graph shown in Fig. 1. Suppose that $s = v_1$, $t = v_{10}$, $U = \{v_6\}$, and $\alpha = 1/2$. First, by lines 2-3, the element $((s', \emptyset), 0)$ is pushed into the priority queue $\mathcal{Q}$. Then, in the first iteration, the element $((s', \emptyset), 0)$ is popped from the queue (line 5), as it has minimum $cost$. Subsequently, the algorithm expands the state $(s', \emptyset)$ in lines 8-11. After that, the algorithm generates two states which are $((v_1, \emptyset), 0)$ and $((v_1, \{v_6\}), 3/2)$, and pushes these states into $\mathcal{Q}$. In the second iteration, the algorithm pops the element $((v_1, \emptyset), 0)$ and expands the state $(v_1, \emptyset)$ in a similar way. When the algorithm pops the state $(v_{10}, \{v_6\})$, the algorithm terminates. We can get that the optimal cost of the state $(v_{10}, \{v_6\})$ is 3, and the optimal route is (v_1, v_3, v_7, v_{10}).

Discussion. Note that Algorithm 1 can be straightforwardly extended to find the optimal route. Specifically, we can maintain an array to record the previous state for each state (v, X) when the $cost$ of (v, X) is updated. Then, we can use this array to reversely output the optimal route. The same technique can be applied to all the algorithms proposed in this paper to output the optimal route. Therefore, in the rest of the paper, we only outline the algorithms to compute the optimal cost, and omit the details for finding the optimal routes. In addition, it is also very easy to determine all the meeting points based on all the recorded states in the optimal route, thus we omit the details for brevity.

Cost analysis. We analyze the time and space complexity of Algorithm 1 in the following theorem.

Theorem 3.3. *Algorithm 1 computes the* OMMPR *query using* $O(3^l \cdot m + 2^l \cdot n \cdot (l + \log(n)))$ *time and* $O(2^l n + m)$ *space, where* l *is the number of elements in* U.

Proof. First, there are n nodes and 2^l subsets of U, thus the total number of states is at most $2^l n$. Therefore, the length of the priority queue $\mathcal{Q}$ is at most $2^l n$, because each state is inserted into $\mathcal{Q}$ at most once. By using Fibonacci heap [22], the time costs of the *push* and *update* operators are $O(1)$, and the cost of the *pop* operator is $O(\log(2^l n))$. As a result, the time overhead for popping all the elements in $\mathcal{Q}$ is at most $O(2^l n(\log(2^l n)))$, and thus the total time cost of line 5 is bounded by $O(2^l n(\log(2^l n)))$. Second, for each node v, there are 2^l states. For each state (v, X), the time cost taken in lines 8-11 is $O(2^{l-|X|}|N_v|)$, where $|N_v|$ denotes the neighbor size of v. Therefore, for a node v, the total cost taken in lines 8-11 (the sum of the costs taken over all the states (v, X)) is $O(|N_v| \sum_{i=0}^{l} \binom{l}{i} 2^{l-i}) = O(3^l|N_v|)$. As a result, the total cost taken in lines 8-11 is at most $O(\sum_{v \in V} 3^l|N_v|) = O(3^l m)$. Note that all the shortest-path distances shown in line 10 can be pre-computed in $O(l(m + n\log n))$ time. Putting it all together, the time complexity of Algorithm 1 is $O(3^l \cdot m + 2^l \cdot n \cdot (l + \log(n)))$. For the space complexity, it is easy to derive that the space overhead is $O(2^l n + m)$. □

3.2 A New DP Solution

As shown in the previous section, the time complexity of Basic is dependent on $3^l m$ which is not very efficient. In this section, we propose a novel DP algorithm based on two growing rules, namely, edge growing and node growing, which is shown to be more efficient than Basic. The idea is that, for each state (v, X), by edge growing, we try to expand it to a new state (u, X) for an edge $(v, u) \in E$; and by node growing, we try to expand it to a new state $(v, X \cup \{x\})$ for the node $x \in U - X$. Specifically, the state transition equation is given by

$$f(u, X) = \min\{\min_{(v,u)\in E}\{f(v, X) + \alpha \times w(v, u)\}, \min_{x \in X}\{f(u, X - \{x\}) + (1 - \alpha) \times dist(x, u)\}\}. \quad (6)$$

We explain the Eq. (6) as follows. Let P_{su} be the optimal route starting from s and ending at u, with query node set X, and $f(u, X)$ be the cost of P_{su}. To obtain $f(u, X)$, we consider the following two cases.

Case-1: edge growing. Assume that all the meeting points for the query nodes in X and the optimal route P_{su} are located in $V_{P_{su}} - \{u\}$. In this case, we can get the optimal route P_{su} by expanding the optimal sub-route P_{sv} with set X, for $(v, u) \in E$. This is because all the meeting points in this case are located in the optimal sub-route P_{sv}. The idea of this case is illustrated in Fig. 5a. It is easy to show that in this case, $f(u, X) = \min_{(v,u)\in E}\{f(v, X) + \alpha \times w(v, u)\}$.

Case-2: node growing. Suppose that there is at least one node $x \in X$ whose meeting point with the optimal route P_{su} is u. In this case, the optimal route P_{su} with set X can be obtained by expanding the optimal route P_{su} with set $X - \{x\}$. The idea of this case is illustrated in Fig. 5b. Under this case, we can derive that $f(u, X) = \min_{x \in X}\{f(u, X - \{x\}) + (1 - \alpha) \times dist(x, u)\}$.

Clearly, $f(u, X)$ can be obtained by taking the minimum cost over these two cases (Eq. (6)). Note that the state transition equation of the new DP algorithm (Eq. (6)) is fundamentally different from that of the Basic algorithm (Eq. (5)). Let us consider a state (u, X). In Basic, the set X is split into two parts X_1 and $X - X_1$, where X_1 is a subset of X. To get the optimal substructures, two disjoint cases are considered in Basic: 1) all the meeting nodes for the nodes in $X - X_1$ and the optimal route P are u, and 2) all the meeting nodes for the nodes in X_1 and the optimal route P are located in $V_P - \{u\}$. However, in the new DP algorithm, to obtain the optimal substructures, we consider two new disjoint cases: 1) no meeting node is u, and 2) at least one meeting node is u.

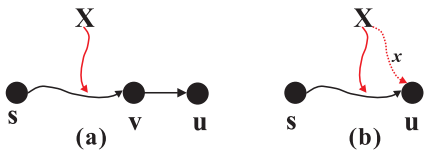


Fig. 5. Illustration of the idea of Eq. (6).

Algorithm 2. Grow(G, U, s, t)

Input: $G = (V, E, W)$, node set U, α, source node s, and target node t.
Output: the minimum cost.

```
1: Q ← ∅; D ← ∅;
2: Q.push((s, ∅), 0);
3: while Q ≠ ∅ do
4:     ((v, X), cost) ← Q.pop();
5:     if v = t and X = U then return cost;
6:     D ← D ∪ {(v, X)};
7:     for all (v, u) ∈ E do
8:         update(Q, D, (u, X), cost + α × w(v, u));
9:     for all x ∈ U − X do
10:        update(Q, D, (v, X ∪ {x}), cost + (1 − α) × dist(x, v));
11: return +∞;
```

The Grow algorithm. Based on Eq. (6), we can compute the OMMPR query by using the best-first dynamic programming. We refer to this DP algorithm as Grow, as it is based on two growing rules. The Grow algorithm is described in Algorithm 2. Similar to Basic, the Grow algorithm also repeatedly deletes the top element $((v, X), cost)$ in the priority queue $\mathcal{Q}$ based on the best-first strategy. Unlike Basic, in each expansion, the algorithm individually expands the end-node of the current route using edge growing (lines 7-8) and the set X with only one new node x using node growing (lines 9-10), and then invokes the same update procedure as Basic to refine the *cost* of the expanded element. The algorithm terminates until it finds the optimal route (line 5). We show the correctness of the algorithm in Theorem 3.4.

Theorem 3.4. *Algorithm 2 correctly computes the optimal cost for the* OMMPR *query.*

Proof. The proof can be obtained by the induction on $|X|$. We omit the details due to space limit. □

The following example illustrates how Algorithm 2 works.

Example 3.5. Reconsider the graph shown in Fig. 1. Assume that $s = v_1$, $t = v_{10}$, $U = \{v_6\}$, and $\alpha = 1/2$. First, the algorithm pushes the element $((s, \emptyset), 0)$ into the priority queue $\mathcal{Q}$ (line 2). Then, in the first iteration, the element

$((s,\emptyset),0)$ is popped from the queue (line 4). Subsequently, the algorithm expands the state $(s,\emptyset)$ following two growing rules: 1) (edge growing) expanding an edge (s,v) for all $(s,v) \in E$, and 2) (node growing) expanding a query node v_6. After that, the algorithm generates four states which are $((v_2,\emptyset),1)$, $((v_3,\emptyset),1)$, $((v_4,\emptyset),1/2)$, and $((v_1,\{v_6\}),3/2)$, and pushes these states into $\mathcal{Q}$. Note that after the first iteration, the priority queue in Algorithm 1 has seven elements, while the priority queue in Algorithm 2 has only four elements. This is because for each (v,X), Algorithm 1 generates $|N_v|2^{(l-|X|)}$ expanded states, whereas Algorithm 2 only generates $|N_v|+(l-|X|)$ states. Clearly, Algorithm 2 is much more efficient than Algorithm 1. Likewise, in the second iteration, the algorithm pops the element $((v_4,\emptyset),1/2)$ and expands the state $(v_4,\emptyset)$. When the state $(v_{10},\{v_6\})$ is popped, the algorithm terminates. We can obtain that the optimal cost for the query is 3, and the optimal route is (v_1,v_3,v_7,v_{10}).

Cost analysis. The time and space complexities of Algorithm 2 are analyzed in Theorem 3.6.

Theorem 3.6. *Algorithm 2 calculates the* OMMPR *query using* $O(2^l \cdot (m+n\cdot(l+\log(n))))$ *time and* $O(2^l n+m)$ *space, where* l *is the number of elements in* U.

Proof. Similar to the proof of Theorem 3.3, the length of the priority queue is at most $2^l n$. Thus, the total cost for popping all the top elements in $\mathcal{Q}$ is $O(2^l n \log(2^l n))$ by using Fibonacci heap. For each node v, there are 2^l states. For each state (v,X), lines 7-10 take $O(|N_v|+(l-|X|))$ time cost. Therefore, for each node v, the total cost taken in lines 7-10 is $O(2^l|N_v|+\sum_{i=0}^{l}\binom{l}{i}(l-i)) = O(2^l|N_v|+l2^{l-1})$. Then, for all the nodes, the total time cost is $O(\sum_{v\in V}(2^l|N_v|+l2^{l-1})) = O(2^l m+nl2^{l-1})$. Note that all the shortest-path distances shown in line 10 can be computed in advance in $O(l(m+n\log n))$ time. Putting it all together, the time complexity of Algorithm 2 is $O(2^l\cdot(m+n\cdot(l+\log(n))))$. In addition, it is easy to derive that the space complexity of Algorithm 2 is $O(2^l n+m)$. □

Remark 2. It is worth to remarking that the proposed DP solutions for the OMMPR query problem (Algorithms 1 and 2) are essentially different from the well-known DP solution, i.e., the Dreyfus-Wagner Algorithm, for the Steiner tree problem [23]. First, the Dreyfus-Wagner algorithm aims at computing the minimum cost Steiner tree, while our algorithms aims to find the optimal $s\sim t$ route. Therefore, to get the optimal Steiner tree, the Dreyfus-Wagner algorithm has to merge the partial results computed from two optimal subtrees, whereas our algorithms are clearly not based on such a subtree-merging idea. Second, from the parameterized complexity point of view, the time complexity of Dreyfus-Wagner algorithm is $O(3^l g(m,n))$ [23], [24], where l is the number of terminal nodes for the Steiner tree problem and $g(m,n)$ is a function over m and n. However, for the OMMPR query problem, the time complexity of our new DP algorithm (Algorithms 2) can achieve $O(2^l g(m,n))$, which is significantly lower than that of the Dreyfus-Wagner algorithm. On the other hand, for the Steiner tree problem, improving the exponential dependence on l from 3^l to 2^l is a well-known hard problem [25], [26], [27], [28]. The best known result for the Steiner tree problem takes $O(2^{l+\delta}g(m,n))$ time complexity, where δ is a constant depending on l and n.

4 Optimization Techniques

Although Grow is much more efficient than Basic, it may still produce a large number of states. To further improve the efficiency of the Grow algorithm, in this section, we propose two new optimized algorithms based on the idea of bidirectional DP and a carefully-designed lower bounding technique, which are shown to be very efficient in large graphs in our experiments.

4.1 Bidirectional DP

We note that in Grow, we can expand either from the source node or the target node to find the optimal route for the OMMPR query. This inspires us to further reduce the search space of Grow based on the idea of bidirectional search [29], [30]. We refer to the Grow process starting from the source node s as the forward Grow, and refer to the Grow process starting from the target node t as the backward Grow. The bidirectional DP algorithm, called Bidirect, works as follows. It alternatively performs the forward Grow and backward Grow. During the bidirectional state-expansion process, the algorithm maintains the cost of the best route that has been computed so far, denoted as $best$. Initially, $best$ is set to $+\infty$, and the states $(s,\emptyset)$ and $(t,\emptyset)$ are pushed into the priority queue for the forward Grow and backward Grow respectively. Then, the algorithm picks the top state (v,X) from the priority queue, and adopts the same rules as Grow to expand the current state. Assume that the optimal cost for the states (v,X) and $(u,U-X)$ are computed by the forward Grow and backward Grow respectively. Then, when the forward Grow scans the edge (v,u), we can obtain a candidate route with cost $c=f(v,X)+f(u,U-X)+\alpha\times w(v,u)$. If $best > c$, which means that we have found a better route than those found before, we can update $best$ to be c. For the backward Grow, we use a similar procedure. The bidirectional DP algorithm terminates when one direction pops a state (v,X) and the optimal cost for the state $(v,U-X)$ has already been computed in the other direction. The correctness of the bidirectional DP can be proved based on the theory of bidirectional search [29], [30]. Furthermore, similar to the bidirectional search algorithms [29], [30], one can show that the search space of the Bidirect algorithm is significantly smaller than that of the Grow algorithm, which is also confirmed in our experiments. The details of the bidirectional DP algorithm are shown in Algorithm 3.

In Algorithm 3 (Bidirect), for each state (v,X), we use the source node of the Grow algorithm as a label to distinguish the forward Grow and backward Grow. Specifically, if the state (v,X) is generated by the forward (backward) Grow, we label it by (s,v,X) $((t,v,X))$. The general procedure of Bidirect is very similar to Grow, thus we omit the details.

The major differences between Bidirect and Grow are the termination condition and the update procedure. In Bidirect, the termination condition is outlined in line 7. Here $(label', v, X') \in \mathcal{D}$ denotes that the optimal cost for the state $(v, U - X)$ has already been computed in another direction, implying that two Grow procedures meets at the node v. Moreover, by $\mathcal{D}.cost((label', v, X')) + cost = best$, we know that the current optimal cost cannot be improved. Based on the theory of bidirectional search [29], [30], we can derive that the optimal route has been found when these conditions are satisfied. Different from Grow, in the update procedure of Bidirect, it not only needs to update the cost of the expanded state (lines 20-21), but also needs to update the $best$ value when the algorithm finds a better route (lines 22-23). The following example shows how Algorithm 3 works.

Algorithm 3. Bidirect(G, U, s, t)

Input: $G = (V, E, W)$, node set U, α, source node s, and target node t.
Output: the minimum cost.

1: $\mathcal{Q} \leftarrow \emptyset$; $\mathcal{D} \leftarrow \emptyset$; $best \leftarrow +\infty$;
2: $\mathcal{Q}.push((s, s, \emptyset), 0)$; $\mathcal{Q}.push((t, t, \emptyset), 0)$;
3: **while** $\mathcal{Q} \neq \emptyset$ **do**
4: $((label, v, X), cost) \leftarrow \mathcal{Q}.pop()$;
5: $label' \leftarrow (label = t?s : t)$;
6: $X' \leftarrow U - X$;
7: **if** $(label', v, X') \in \mathcal{D}$ **and** $\mathcal{D}.cost((label', v, X')) + cost = best$ **then**
8: **return** $best$;
9: $\mathcal{D} \leftarrow \mathcal{D} \cup \{((label, v, X), cost)\}$;
10: **for all** $(v, u) \in E$ **do**
11: $cost' \leftarrow cost + \alpha \times w(v, u)$;
12: update$(\mathcal{Q}, \mathcal{D}, best, label', X', (label, u, X), cost')$;
13: **for all** $x \in U - X$ **do**
14: $cost' \leftarrow cost + (1 - \alpha) \times dist(x, v)$;
15: update$(\mathcal{Q}, \mathcal{D}, best, label', X' - \{x\}, (label, v, X \cup \{x\}), cost')$;
16: **return** $+\infty$;
17: **Procedure** update$(\mathcal{Q}, \mathcal{D}, best, label', X', (label, v, X), cost)$
18: **if** $(label, v, X) \in \mathcal{D}$ **then return**;
19: **if** $(label, v, X) \notin \mathcal{Q}$ **then** $\mathcal{Q}.push((label, v, X), cost)$;
20: **if** $cost < \mathcal{Q}.cost((label, v, X))$ **then**
21: $\mathcal{Q}.update((label, v, X), cost)$;
22: **if** $(label', v, X') \in \mathcal{D}$ **and** $\mathcal{D}.cost((label', v, X')) + cost < best$ **then**
23: $best \leftarrow \mathcal{D}.cost((label', v, X')) + cost$;

Example 4.1. Consider the graph shown in Fig. 1. Suppose that $s = v_1$, $t = v_{10}$, $U = \{v_6\}$, and $\alpha = 1/2$. Initially, Algorithm 3 pushes the elements $((s, s, \emptyset), 0)$ and $((t, t, \emptyset), 0)$ into the priority queue $\mathcal{Q}$, where the first s (t) in $((s, s, \emptyset), 0)$ ($((t, t, \emptyset), 0)$) is the label representing the forward (backward) Grow. Then, in the first iteration, the algorithm pops the element $((s, s, \emptyset), 0)$, and runs the forward Grow to expand the state $(s, \emptyset)$. In this process, four elements $((s, v_2, \emptyset), 1)$, $((s, v_3, \emptyset), 1)$, $((s, v_4, \emptyset), 1/2)$, and $((s, v_1, \{v_6\}), 3/2)$ are generated and pushed into $\mathcal{Q}$. In the second iteration, the algorithm pops the element $((t, t, \emptyset), 0)$ as it has minimum cost, and runs the backward Grow to expand the state $(t, \emptyset)$. After that, the algorithm creates four new elements which are $((t, v_7, \emptyset), 1/2)$, $((t, v_8, \emptyset), 1)$, $((t, v_9, \emptyset), 1/2)$, and $((t, t, \{v_6\}), 2)$. And then, in the third iteration, the algorithm pops the element $((s, v_4, \emptyset), 1/2)$, and performs the forward Grow to expand the state $(v_4, \emptyset)$. The procedures for other iterations are very similar, thus we omit the details. Finally we can get that the optimal cost returned by the algorithm is 3, which is consistent with the results of the previous examples.

4.2 Bidirectional Bounded DP

To further reduce the search space of our DP algorithms, we propose a bidirectional bounded DP algorithm, called Bidirect-Bounded, by integrating the bidirectional DP and a carefully designed lower bounding technique. Below, we first present a lower bound of the cost of the OMMPR query, and then introduce our algorithm.

Lower bound construction. First, we introduce a procedure to compute some useful quantities that are required to construct the lower bound. Such quantities are computed before applying the DP algorithm. Specifically, for all $X \subseteq U$ and $x, y \in X$, we refer to (x, y, X) as a state, denoting a route that starts from x, ends at y, and passes through all nodes in X. Let $C(x, y, X)$ be the optimal cost of the state (x, y, X). Then, it is easy to show that $C(x, y, X)$ can be computed using dynamic programming. In particular, for any $X \subseteq U$ and $x, v \in X$, the state transition equation of the DP algorithm is given by

$$C(x, y, X) = \min_{v \in X - \{y\}} \{C(x, v, X - \{y\}) + dist(v, y)\}. \quad (7)$$

Initially, we have $C(u, u, \{u\}) = 0$, for all $u \in U$. The idea of Eq. (7) is that the optimal route corresponding to the state (x, y, X) can be obtained by expanding the node v from the optimal sub-route $(x, v, X - \{y\})$ for each $v \in X - \{y\}$.

Based on Eq. (7), we can make use of the best-first DP algorithm to compute all the $C(x, y, X)$ for all $X \subseteq U$ and $x, y \in X$. The detailed description of the DP algorithm is outlined in Algorithm 4. We analyze the time and space complexity of Algorithm 4 in Theorem 4.2.

Theorem 4.2. *Algorithm 4 computes $C(x, y, X)$ for all $X \subseteq U$ and $x, y \in X$ in $O(2^l \cdot l^3 + (m + n \cdot \log(n)) \cdot l)$ time and $O(2^l \cdot l^2 + n \cdot l)$ space, where $l = |U|$.*

Proof. First, we can pre-compute the all pair shortest-path distances for the nodes in set U, which takes $O((m + n \cdot \log(n))l)$ time complexity. Second, there are $O(l^2 2^l)$ states in total, thus the length of the priority queue is bounded by $O(l^2 2^l)$. Therefore, the total cost for picking the top element from the priority queue is $O(l^2 2^l \cdot \log(l^2 2^l))$ by using Fibonacci heap. The total cost of lines 8-13 is bounded by $O(l^3 2^l)$ based on the pre-computed shortest-path distances. Putting it all together, we get that the time complexity of Algorithm 4 is $O(2^l l^3 + (m + n \cdot \log(n)) \cdot l)$. In addition, we can easily derive that the space complexity of Algorithm 4 is $O(2^l l^2 + n \cdot l)$. □

After computing $C(x, y, X)$ for all $X \subseteq U$ and $x, y \in X$, we develop a lower bound for the cost of the OMMPR query in the following lemma.

Lemma 4.3. *Given a query $Q = (s, t, U, \alpha)$ with $\alpha > 1/3$. Suppose that $C(x, y, X)$ for all $X \subseteq U$ and $x, y \in X$ have been*

pre-computed. Let $P' = ((v, X), cost)$ be a partial result of the DP algorithm (e.g., Algorithm 2), where (v, X) and cost denote a state of the DP algorithm and its corresponding cost respectively. Further, we let $X' = U - X$, and

$$c(\tilde{P}) = \min_{x,y \in X'} \{dist(v, x) + C(x, y, X') + dist(y, t)\}, \quad (8)$$

where $\tilde{P}$ denotes the optimal $v \sim t$ route that passes through all the nodes in X'. Then, for any partial result P', the lower bound of the optimal cost of the route expanded from the state (v, X) to the final state (t, U), denoted as $\underline{\text{cost}}(P')$, can be calculated by

$$\underline{\text{cost}}(P') = \begin{cases} cost + \frac{1-\alpha}{2} c(\tilde{P}) + \frac{3\alpha-1}{2} dist(v, t), & if \ \ X' \neq \emptyset \\ cost + \alpha \times dist(v, t), & otherwise. \end{cases} \quad (9)$$

Proof. Let P_{vt} be the optimal route starting from v and ending at t, with query node set X'. To prove the theorem, we need to prove that $cost + \alpha \times c(P_{vt}) + (1-\alpha) \sum_{u \in X'} dist(u, P_{vt}) \geq \underline{\text{cost}}(P')$. Clearly, when $X' = \emptyset$, it is easy to show that the above inequality holds. When $X' \neq \emptyset$, we have

$$\begin{aligned} &\alpha \times c(P_{vt}) + (1-\alpha) \sum\nolimits_{u \in X'} dist(u, P_{vt}) \\ &\quad - \frac{1-\alpha}{2} c(\tilde{P}) - \frac{3\alpha - 1}{2} dist(v, t) \\ &\quad = \alpha \times (c(P_{vt}) - dist(v, t)) \\ &\quad + \frac{1-\alpha}{2} (2 \sum\nolimits_{u \in X'} dist(u, P_{vt}) - c(\tilde{P}) + dist(v, t)) \\ &\quad \geqslant \frac{1-\alpha}{2} (c(P_{vt}) + 2 \sum\nolimits_{u \in X'} dist(u, P_{vt}) - c(\tilde{P})) \\ &\quad \geqslant 0, \end{aligned}$$

where the first inequality is due to $\alpha > (1-\alpha)/2$ when $\alpha > 1/3$. We explain the last inequality below. Note that $c(P_{vt}) + 2 \sum_{u \in X'} dist(u, P_{vt})$ can be deemed as the cost of a route that starts from v and ends at t, and passes through all the nodes in X'. By definition, we know that $\tilde{P}$ is the optimal $v \sim t$ route that passes through all the nodes in X', thus we have $c(P_{vt}) + 2 \sum_{u \in X'} dist(u, P_{vt}) \geq c(\tilde{P})$. □

It is straightforward to show that Lemma 4.3 also holds in the backward direction. Specifically, in the backward direction, the DP algorithm starts from the state $(t, \emptyset)$ and ends at the state (s, U). For the partial result $P' = ((v, X), cost)$, the lower bound in the backward direction can be computed by

$$\underline{\text{cost}}(P') = \begin{cases} cost + \frac{1-\alpha}{2} c(\tilde{P}) + \frac{3\alpha-1}{2} dist(s, v), & if \ \ X' \neq \emptyset \\ cost + \alpha \times dist(s, v), & otherwise, \end{cases} \quad (10)$$

where $X' = U - X$ and $c(\tilde{P}) = \min_{x,y \in X'} \{dist(s, x) + C(x, y, X') + dist(y, v)\}$.

The Bidirect-Bounded algorithm. Note that the lower bound shown in Lemma 4.3 is designed for the case $\alpha > 1/3$. To devise the Bidirect-Bounded algorithm, we need to consider two different cases: 1) $\alpha \leq 1/3$, and 2) $\alpha > 1/3$. For $\alpha \leq 1/3$, we prove that the optimal solution for the OMMPR query can be computed based on the following lemma.

Lemma 4.4. *Given a query $Q = (s, t, U, \alpha)$ with $\alpha \leq 1/3$, the optimal cost of the OMMPR query is $\alpha \times \min_{x \in U, y \in U} (dist(s, x) + C(x, y, U) + dist(y, t))$.*

Proof. For any $s \sim t$ route P_{st} and a node $u \in U$, let v be the meeting point for node u and route P_{st}. Then, the weighted average cost for P_{st} and node u, denoted by $f(P_{st}, u)$, is $\alpha \times c(P_{st}) + (1-\alpha) \times dist(u, v)$. Since $\alpha \leq 1/3$, $(1-\alpha) \times dist(u, v) \geq 2\alpha \times dist(u, v)$. Thus, we can construct an alternative route $(s \sim v \sim u \sim v \sim t)$ whose cost is $\alpha \times c(P_{st}) + 2\alpha \times dist(u, v)$, which is no larger than $f(P_{st}, u)$. Note that this route passes through the query node u. By this construction, we can easily derive that the optimal route must be a route that passes through all the query nodes. As a result, the optimal cost of the OMMPR query is $\alpha \times \min_{x \in U, y \in U} (dist(s, x) + C(x, y, U) + dist(y, t))$, which corresponds to the optimal route starting from s, ending at t, and passing through all nodes in U. □

Algorithm 4. All-Set-Paths(G, U)

```
1: Q ← ∅; D ← ∅;
2: for all u ∈ U do
3:     Q.push((u, u, {u}), 0);
4: while Q ≠ ∅ do
5:     ((x, y, X), cost) ← Q.pop();
6:     C(x, y, X) ← cost;
7:     D ← D ∪ {(x, y, X)};
8:     for all v ∈ U − X do
9:         X' ← X ∪ {v};
10:        cost' ← cost + dist(y, v);
11:        if (x, v, X') ∈ D then continue;
12:        if (x, v, X') ∉ Q then Q.push((x, v, X'), cost');
13:        if cost' < Q.cost((x, v, X') then
           Q.update((x, v, X'), cost');
```

Based on Lemma 4.4, when $\alpha \leq 1/3$, the Bidirect-Bounded algorithm computes the optimal solution of the OMMPR query as follows. First, for a given query $Q = (s, t, U, \alpha)$, the algorithm computes $dist(s, x)$ and $dist(y, t)$ for all $x, y \in U$ by using the Dijkstra algorithm. Second, the algorithm computes all $C(x, y, U)$ for every $x, y \in U$ by using Algorithm 4. Finally, the algorithm takes the minimum value over all $dist(s, x) + C(x, y, U) + dist(y, t)$ for all $x, y \in U$.

For the case of $\alpha > 1/3$, the Bidirect-Bounded algorithm integrates the bidirectional DP algorithm and the proposed lower bounding technique. Unlike our previous DP algorithms which use the best-first strategy, the Bidirect-Bounded algorithm adopts the bidirectional DP algorithm with an A^*-heuristic strategy [30], [31], [32] to expand the states, and simultaneously uses the lower bound developed in Lemma 4.3 to prune the unpromising states. Here the A^*-heuristic strategy expands the states based on the lower bound of the cost of the state. The key idea of the algorithm is as follows. The Bidirect-Bounded algorithm adopts the bidirectional DP with the A^*-heuristic strategy to find the optimal route, and maintains the cost of the best route, denoted as $best$, which has been calculated so far. In this procedure, the algorithm computes the lower bound for each state. If the lower bound cost of a state (v, X) is larger than $best$, then all the states expanded from the state (v, X) can be pruned. As a result, a large number of unpromising states can be pruned based on such a lower bound. In the experiments,

we will show that the Bidirect-Bounded algorithm is much more efficient than the other proposed DP algorithms. The Bidirect-Bounded algorithm terminates when one direction of the DP algorithm (either forward DP or backward DP) reaches the final state. The detailed implementation of the Bidirect-Bounded algorithm is depicted in Algorithm 5.

The Bidirect-Bounded algorithm first invokes Algorithm 4 to compute $C(x, y, X)$ for all $X \subseteq U$ and $x, y \in X$ (line 1). If $\alpha \leq 1/3$, the algorithm computes the optimal solution based on Lemma 4.4 (lines 2-3). Otherwise, the algorithm finds the shortest path P_{st} from s to t, and initializes $best$ to be $f(P_{st})$ (lines 4-5). Then, the algorithm initializes the priority queue, and performs the bidirectional DP with an A^*-heuristic strategy to find the optimal route (lines 6-20). Note that in Algorithm 5, each state (v, X) is represented by a tuple $((label, v, X), cost, lb)$, where $label$ is used to distinguish the forward and backward DP, v denotes a node, and X is a subset of U, $cost$ denotes the cost of the state (v, X), and lb denotes the lower bound of the state (v, X) which is calculated by Eq. (9) (or Eq. (10)). The general procedure of the algorithm is very similar to that of Bidirect, thus we omit the details. The main differences between Bidirect-Bounded and Bidirect are summarized below. First, Bidirect-Bounded uses the lower bound cost of the state as the priority by the A^*-heuristic strategy, whereas Bidirect uses the cost of the state as the priority. Thus in line 10, Bidirect-Bounded always pops the state that has the minimum lower bound cost. Second, the termination condition of Bidirect-Bounded (line 13) is different from that of Bidirect. In Bidirect-Bounded, the algorithm terminates when one direction of the DP reaches the final state, whereas in Bidirect, the algorithm terminates when the forward DP and backward DP meet at a certain node. Third, in the update procedure, the Bidirect-Bounded algorithm first computes the lower bound of the expanded state by invoking lb$((label, v, X), label', X', cost, \mathcal{D})$ (line 24), and determines whether the state can be pruned or not (line 25). If the state cannot be pruned, the algorithm needs to update the cost, the lower bound, as well as the $best$ value (lines 26-30). In addition, it is worth mentioning that in the procedure of computing the lower bound, when the optimal cost denoted by $cost'$ from the state (v, X) to the final state has been computed in the reverse direction, the lower bound of (v, X) is exactly equals to $cost + cost'$ (line 32).

Example 4.5. Let us consider a graph shown in Fig. 1. Suppose that $s = v_1$, $t = v_{10}$, $U = \{v_6\}$, and $\alpha = 1/2$. First, the algorithm pre-computes $C(v_6, v_6, \{v_6\})$ which equals 0 (line 1). Second, the algorithm finds the shortest path from v_1 to v_{10} which are $(v_1, v_4, v_5, v_9, v_{10})$, and initializes $best$ to be 7/2. Third, the algorithm computes the lower bounds of the sates $(s, s, \emptyset)$ and $(t, t, \emptyset)$ that are both 11/4. Then, the algorithm pushes the elements $((s, s, \emptyset), 0, 11/4)$ and $((t, t, \emptyset), 0, 11/4)$ into $\mathcal{Q}$ (lines 7-8). In the first iteration, the algorithm pops the element $((s, s, \emptyset), 0, 11/4)$, and performs the forward DP to expand the state $(s, s, \emptyset)$. Then, four elements $((s, v_2, \emptyset), 1, 4)$, $((s, v_3, \emptyset), 1, 3)$, $((s, v_4, \emptyset), 1/2, 13/4)$, and $((s, v_1, \{v_6\}), 3/2, 7/2)$ are generated. Since the lower bounds of the states $(s, v_2, \emptyset)$ and $(s, v_1, \{v_6\})$ are no less than $best$, these two states are pruned (line 25), and elements $((s, v_3, \emptyset), 1, 3)$ and $((s, v_4, \emptyset), 1/2, 13/4)$ are pushed into $\mathcal{Q}$. Comparing to the Bidirect algorithm, here the Bidirect-Bounded algorithm can prune two states, thus it is more efficient than Bidirect. In the second iteration, the algorithm pops the element $((t, t, \emptyset), 0, 11/4)$ as it has the smallest lower bound. The algorithm perform a similar way to expand the state. For other iterations, the processes are similar. When the algorithm terminates, we can get that the optimal cost returned by the algorithm is 3, which is consistent with the previous examples.

Algorithm 5. Bidirect-Bounded(G, U, s, t)

Input: $G = (V, E, W)$, node set U, α, source node s, and target node t.
Output: the minimum cost.

1: All-Set-Paths(G, U);
2: **if** $\alpha \leq \frac{1}{3}$ **then**
3: **return** $\alpha \times \min_{x \in U, y \in U}(dist(s, x) + C(x, y, U) + dist(y, t))$;
4: $P' \leftarrow$ shortest path from s to t;
5: $best \leftarrow \alpha \times c(P') + (1 - \alpha) \times \sum_{x \in U}(\min_{v \in V_{P'}} dist(x, v))$;
6: $\mathcal{Q} \leftarrow \emptyset$; $\mathcal{D} \leftarrow \emptyset$;
7: $\mathcal{Q}.push((s, s, \emptyset), 0,$ lb$((s, s, \emptyset), t, U, 0, \mathcal{D}))$; /* lb is the priority in $\mathcal{Q}$. */
8: $\mathcal{Q}.push((t, t, \emptyset), 0,$ lb$((t, t, \emptyset), s, U, 0, \mathcal{D}))$;
9: **while** $\mathcal{Q} \neq \emptyset$ **do**
10: $((label, v, X), cost, lb) \leftarrow \mathcal{Q}.pop()$; /* pop the minimum lb element. */
11: $label' \leftarrow (label = t ? s : t)$;
12: $X' \leftarrow U - X$;
13: **if** $v = label'$ **and** $X = U$ **then return** cost;
14: $\mathcal{D} \leftarrow \mathcal{D} \cup \{((label, v, X), cost)\}$;
15: **for all** $(v, u) \in E$ **do**
16: $cost' \leftarrow cost + \alpha \times w(v, u)$;
17: update$(\mathcal{Q}, \mathcal{D}, best, label', X', (label, u, X), cost')$;
18: **for all** $x \in U - X$ **do**
19: $cost' \leftarrow cost + (1 - \alpha) \times dist(x, v)$;
20: update$(\mathcal{Q}, \mathcal{D}, best, label', X' - \{x\}, (label, v, X \cup \{x\}), cost')$;
21: **return** $+\infty$;
22: **Procedure** update$(\mathcal{Q}, \mathcal{D}, best, label', X', (label, v, X), cost)$
23: **if** $(label, v, X) \in \mathcal{D}$ **then return**;
24: $lb \leftarrow$ lb$((label, v, X), label', X', cost, \mathcal{D})$;
25: **if** $lb >= best$ **then return**;
26: **if** $(label, v, X) \notin \mathcal{Q}$ **then** $\mathcal{Q}.push((label, v, X), cost, lb)$;
27: **if** $lb < \mathcal{Q}.lb((label, v, X))$ **then**
28: $\mathcal{Q}.update((label, v, X), cost, lb)$; /* update both $cost$ and lb */
29: **if** $(label', v, X') \in \mathcal{D}$ **and** $\mathcal{D}.cost((label', v, X')) + cost < best$ **then**
30: $best \leftarrow \mathcal{D}.cost((label', v, X')) + cost$;
31: **Procedure** lb$((label, v, X), label', X', cost, \mathcal{D})$
32: **if** $(label', v, X') \in \mathcal{D}$ **then return** $cost + \mathcal{D}.cost((label', v, X'))$;
33: **if** $label = s$ **then**
34: **if** $X' \neq \emptyset$ **then**
35: $c \leftarrow \min_{x \in X', y \in X'}\{dist(v, x) + C(x, y, X') + dist(y, t)\}$;
36: **return** $cost + \frac{1-\alpha}{2} \times c + \frac{3 \times \alpha - 1}{2} \times dist(v, t)$;
37: **return** $cost + \alpha \times dist(v, t)$;
38: **else**
39: **if** $X' \neq \emptyset$ **then**
40: $c \leftarrow \min_{x \in X', y \in X'}\{dist(s, x) + C(x, y, X') + dist(y, v)\}$;
41: **return** $cost + \frac{1-\alpha}{2} \times c + \frac{3 \times \alpha - 1}{2} \times dist(s, v)$;
42: **return** $cost + \alpha \times dist(s, v)$;

Correctness analysis. When $\alpha \leq 1/3$, the algorithm is obviously correct by Lemma 4.4. Below, we analyze the correctness of the Bidirect-Bounded algorithm when $\alpha > 1/3$.

Since Bidirect-Bounded terminates when one direction of the DP reaches the final state, we focus on analyzing the correctness of one direction of the DP. For simplicity, we consider the forward DP, and similar results can be obtained for the backward DP. Let $P_{vt}(X) = ((v, X), cost)$ be a partial result of the forward DP algorithm (forward Grow), and let $X' = U - X$. For the state (v, X) with $v \neq t$ and $X \neq U$, the forward DP algorithm has two different strategies to expand the state (v, X): 1) expanding an edge (v, u), and 2) expanding a query node $x \in X'$ (see Algorithm 5). For the first strategy, we can obtain a state (u, X), whereas for the second strategy, we can get a state $(v, X \cup \{x\})$. Let $\pi(P_{ut}(X)) = \underline{\text{cost}}(P_{ut}(X)) - cost$. Then, we have the following two lemmas.

Lemma 4.6. $\alpha \times w(v, u) + \pi(P_{ut}(X)) \geq \pi(P_{vt}(X))$.

Proof. Let $X' = U - X$. We consider two cases: 1) $X' = \emptyset$, and 2) $X' \neq \emptyset$. For the first case, we have $\alpha \times w(v, u) + \alpha \times dist(u, t) \geq \alpha \times dist(v, t)$, thus the lemma holds. For the second case, we let $c(\tilde{P}_{vt}(X))) = \min_{x,y \in X'} \{dist(v, x) + C(x, y, X') + dist(y, t)\}$, and $c(\tilde{P}_{ut}(X))) = \min_{x,y \in X'} \{dist(u, x) + C(x, y, X') + dist(y, t)\}$. Then, we have

$$\begin{aligned}
&\alpha \times w(v,u) + \frac{1-\alpha}{2} c(\tilde{P}_{ut}(X)) + \frac{3\alpha - 1}{2} dist(u,t) \\
&= \frac{1-\alpha}{2} (c(\tilde{P}_{ut}(X)) + w(v,u)) + \frac{3\alpha - 1}{2} (w(v,u) + dist(u,t)) \\
&\geqslant \frac{1-\alpha}{2} (c(\tilde{P}_{ut}(X)) + w(v,u)) + \frac{3\alpha - 1}{2} dist(v,t) \\
&\geqslant \frac{1-\alpha}{2} c(\tilde{P}_{vt}(X)) + \frac{3\alpha - 1}{2} dist(v,t),
\end{aligned}$$

where the last inequality can be derived by the definition of $c(\tilde{P}_{vt}(X))$. Putting it all together, the lemma is established. □

Lemma 4.7. *When* $X \neq U$, $(1 - \alpha) \times dist(x, v) + \pi(P_{vt}(X \cup \{x\})) \geq \pi(P_{vt}(X))$.

Proof. We consider two different cases: 1) $X \cup \{x\} = U$, and 2) $X \cup \{x\} \neq U$. For the first case, we have $\pi(P_{vt}(X \cup \{x\})) = \alpha \times dist(v, t)$ by Eq. (9). Then, we have

$$\begin{aligned}
&(1-\alpha) \times dist(x,v) + \alpha \times dist(v,t) \\
&= \frac{1-\alpha}{2} (2 \times dist(x,v) + dist(v,t)) + \frac{3\alpha - 1}{2} dist(v,t) \\
&\geqslant \frac{1-\alpha}{2} c(\tilde{P}_{vt}(X)) + \frac{3\alpha - 1}{2} dist(v,t),
\end{aligned}$$

where the last inequality can be derived by the definition of $c(\tilde{P}_{vt}(X))$. For the second case, we need to prove that

$$\begin{aligned}
&(1-\alpha) \times dist(x,v) \\
&\quad + \frac{1-\alpha}{2} c(\tilde{P}_{vt}(X \cup \{x\})) \geqslant \frac{1-\alpha}{2} c(\tilde{P}_{vt}(X)).
\end{aligned}$$

By the definition of $c(\tilde{P}_{vt}(X))$, it is easy to show that the above inequality holds. This completes the proof. □

Lemma 4.6 and Lemma 4.7 imply that the proposed lower bounding technique satisfies the so-called *consistent* condition defined in the A^* algorithms [30], [31], [33]. Based on the optimality of the A^*-search theory [31], [33], we can conclude that the Bidirect-Bounded algorithm finds the optimal route for the OMMPR query.

Cost analysis. It is easy to show that the worst-case time and space complexity of the Bidirect-Bounded algorithm are no higher than those of Algorithm 2 and Algorithm 3. In practice, we will show that the Bidirect-Bounded algorithm is much more efficient than the other proposed DP algorithms. The reason is as follows. Based on Lemmas 4.6 and 4.7, we can see that the lower bound will monotonously increase when the forward (backward) DP algorithm expands a state (see lines 16-17, lines 19-20, and line 24). Therefore, in the process of running Bidirect-Bounded, the lower bound (lb in line 24) keeps increasing, whereas the $best$ value keeps decreasing, and thus the pruning power of the algorithm is increasingly strong. When $best$ is close to optimal, most of unpromising states will be pruned by the Bidirect-Bounded algorithm. As a result, the algorithm only needs to generate a small number of states, and thus it is very efficient.

TABLE 1
Datasets

Dataset	Description	$\|V\|$	$\|E\|$
NW	Northwest USA	1,207,945	2,840,208
W	Western USA	6,262,104	15,248,146
CTR	Central USA	14,081,816	34,292,496
USA	Full USA	23,947,347	58,333,344

5 Performance Studies

We conduct extensive performance studies to evaluate the proposed algorithms. We implement four algorithms, namely, Basic (Algorithm 1), Grow (Algorithm 2), Bidirect (Algorithm 3), and Bidirect-Bounded (Algorithm 5). To the best of our knowledge, there is no algorithm in the literature that can answer the OMMPR query exactly. Thus, in our experiments, we use Basic as the baseline. All algorithms are implemented in C++, and the graph is stored in memory by using a adjacency list. We compare the query processing time and memory consumption for all the algorithms. In all the tests, for the Bidirect-Bounded algorithm, the time for computing the costs of all set paths (Algorithm 4) is included in the reported processing time. For memory consumption, we only report the memory allocated in query processing without including the memory used to store the graph. This is because for each dataset, the memory used to store the graph keeps unchanged for all queries. All experiments are conducted on a computer with 3.4 GHz Intel Xeon E5-2687W CPU and 32 GB memory running Red Hat Enterprise Linux 6.4 (64-bit).

Datasets. We use four large real-world datasets *NW*, *W*, *CTR*, and *USA*, each of which corresponds to a part of the road network in the United States (US). Among them, *USA* is the road network of the whole US. All datasets are downloaded from the DIMACS website (http://www.dis.uniroma1.it/challenge9/). The detailed statistics of the datasets are reported in Table 1.

Parameters and query generation. For each dataset, we vary five parameters, namely, $dist(s, t)$, $\bar{d}(U)$, $\bar{d}(U, s, t)$, $|U|$, and α. Here $dist(s, t)$ is the shortest-path distance from node s to

TABLE 2
Parameters

Parameter	Range	Default
$dist(s,t)$	$[100, 500]$ (km)	300 km
$\bar{d}(U)$	$[10\%, 50\%]$ ($\times dist(s,t)$)	30%
$\bar{d}(U,s,t)$	$[10\%, 50\%]$ ($\times dist(s,t)$)	30%
$\lvert U\rvert$	$[3, 7]$	5
α	$[0.2, 0.6]$	0.4

node t. $\bar{d}(U)$ is the average shortest-path distance for every pair of nodes in U. Let P_{st} be the shortest path from the source node s to the target node t. $\bar{d}(U,s,t)$ is average shortest-path distance from the nodes in U to the shortest path P_{st}, i.e., $\bar{d}(U,s,t) = \sum_{u \in U} d(u, V_{P_{st}})/|U|$. Given a certain set of parameters, our query is generated as follows. In the first step, we randomly select the source node s and target node t, such that the distance between s and t is $d(s,t)$. In the second step, we draw a square on the map. We control the center of the square such that the expected distance from any point in the square to the path from s to t is $\bar{d}(U,s,t)$. We also control the size of the square such that the expected distance between any two randomly selected points in the square is $\bar{d}(U,s,t)$. In the third step, we randomly select $|U|$ nodes in the square to be U. We keep doing the three steps until a valid query that satisfy all constraints is selected.

The range and the default values of the parameters are shown in Table 2. For $\bar{d}(U)$ and $\bar{d}(U,s,t)$, their values are represented in terms of the percentage of $d(s,t)$. When varying a certain parameter, the other parameters are set to their default values. For each test with a specific set of parameters, we randomly generate 20 queries with the corresponding parameters, and report the average results over all the 20 queries.

Exp-1: Vary $dist(s,t)$. In this experiment, we vary $dist(s,t)$ from 100 to 500 km. The query processing time for the four datasets is shown in Figs. 6a, 6b, 6c, and 6d respectively. Generally, when graph size increases, the query processing time for each algorithm increases. However, the increment of the query processing time is not proportional to the increment of graph size. The reason is that the total search space of all the proposed DP algorithms are confined by the parameters $dist(s,t)$, $\bar{d}(U)$, and $\bar{d}(U,s,t)$ which are fixed for all datasets. On average, Grow is 0.5 times faster than Basic; Bidirect is 6.6 times faster than Grow; and Bidirect-Bounded is 8 times faster than Bidirect in all datasets. Not surprisingly, the query processing time for each algorithm increases with increasing $dist(s,t)$, because the search space for all algorithms enlarge when $dist(s,t)$ increases. The results are consistent with the theoretical analysis shown in Sections 3 and 4. In addition, in *USA* dataset, we can see that our best algorithm (Bidirect-Bounded) takes less than one second to answer the OMMPR query when $dist(s,t) \leq 300$ km, and even when $dist(s,t) > 300$ km our best algorithm is still very efficient, which takes only a few seconds to answer the OMMPR query. This result indicate that Bidirect-Bounded can be used for real-time ridesharing applications.

We also report the memory consumption for each algorithm when varying $dist(s,t)$ from 100 to 500 km. The results are shown in Figs. 7a, 7b, 7c, and 7d respectively. For all algorithms, the curves for memory consumption in all datasets are similar to those for query processing time shown in Fig. 6. Bidirect-Bounded performs much better than all the other algorithms over all datasets. This is because Bidirect-Bounded prunes a large number of unpromising states, thus the space overhead for storing the states is smaller than that of the other algorithms. In the following, due to space limit, we only report the processing time for each query. The memory consumptions for all algorithms are consistent with those shown in Fig. 7.

Exp-2: Vary $\bar{d}(U)$. In this experiment, we vary $\bar{d}(U)$ from 10 to 50 percent of $d(s,t)$ (default 300 km), that is, from 30 to 150 km. The query processing time of all the algorithms over the four datasets are shown in Fig. 8. Similar to Exp-1, when the graph size increases, the query processing time for each algorithm increases. When $\bar{d}(U)$ increases, the query processing time for each algorithm increases. The reason is that, for a fixed $dist(s,t)$, when $\bar{d}(U)$ is large, the algorithm needs to find the meeting points for nodes in U that

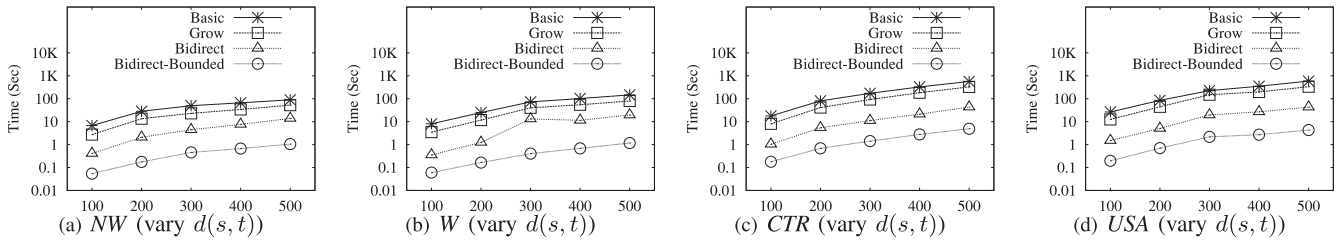


Fig. 6. Vary $d(s,t)$ (in km): processing time.

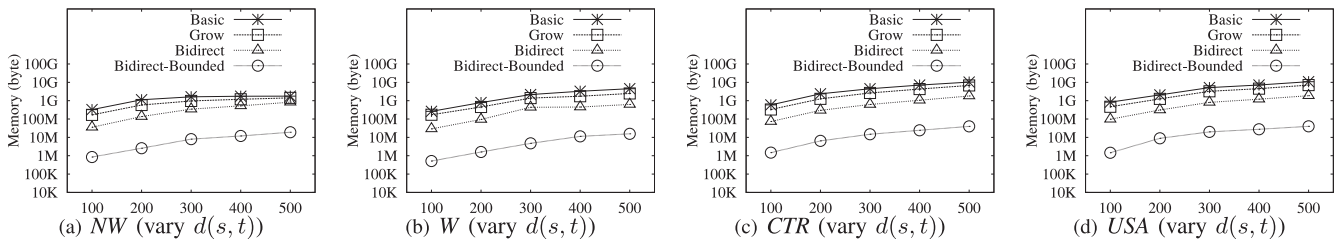


Fig. 7. Vary $d(s,t)$ (in km): memory consumption.

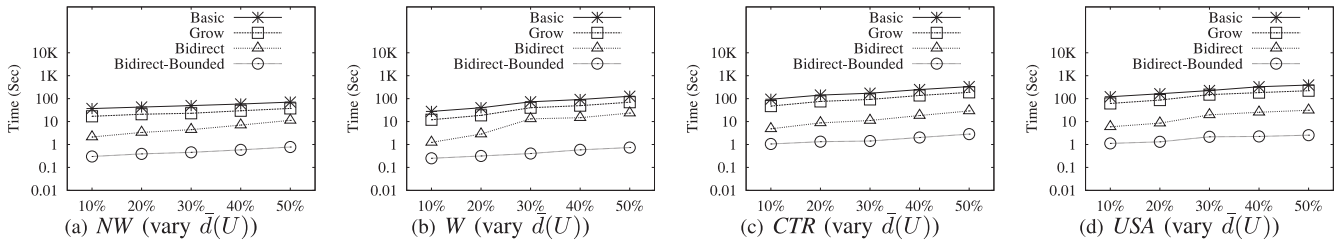


Fig. 8. Vary $\bar{d}(U)$ (in percentage of $d(s,t)$).

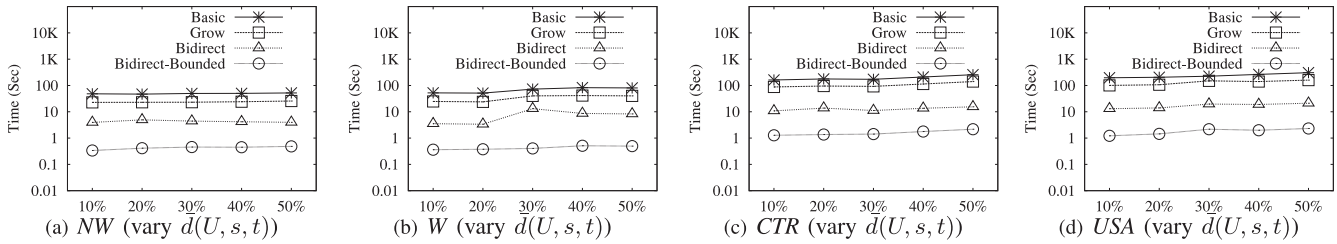


Fig. 9. Vary $\bar{d}(U,s,t)$ (in percentage of $d(s,t)$).

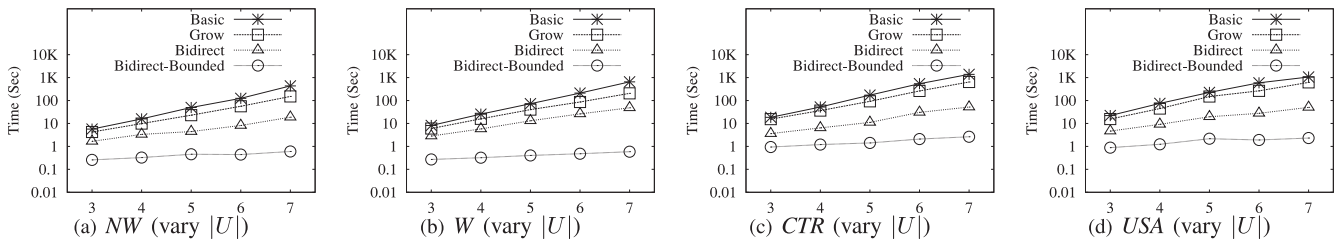


Fig. 10. Vary $|U|$.

are far away from each other, thus the time overhead for computing the optimal route increases. In addition, in all datasets, when $\bar{d}(U)$ increases, the processing time of Basic, Grow and Bidirect increases faster than that of Bidirect-Bounded. For example, in the *USA* dataset, when $\bar{d}(U)$ increases from 10 to 50 percent, the processing time of Basic, Grow, Bidirect, and Bidirect-Bounded increases by 2.2, 2.6, 4.3, and 1.3 times respectively. This is due to the strong pruning power of Bidirect-Bounded, which is consistent with the theoretical analysis shown in Section 4.2. This result also indicates the high scalability of the Bidirect-Bounded algorithm.

Exp-3: Vary $\bar{d}(U,s,t)$. In this experiment, we vary $\bar{d}(U,s,t)$ from 10 to 50 percent of $d(s,t)$ (default 300 km), that is, from 30 to 150 km. The results are depicted in Fig. 9. In all datasets, we can see that the query processing time of each algorithm slightly increases with increasing $\bar{d}(U,s,t)$. For example, in the *USA* dataset, when $\bar{d}(U,s,t)$ increases from 10 to 50 percent, the query processing time for Basic, Grow, Bidirect, and Bidirect-Bounded increases by 0.6, 0.6, 0.7, and 0.7 times respectively. This results indicate that for a fixed $dist(s,t)$ and $\bar{d}(U)$, the query processing time for all the algorithms is not sensitive to $\bar{d}(U,s,t)$. Similarly, Bidirect-Bounded performs much better than the other algorithms.

Exp-4: Vary $|U|$. In this experiment, we study how $|U|$ affects the performance of our algorithms. In particular, we vary $|U|$ from 3 to 7, as $|U|$ is generally very small in ride-sharing related applications (e.g., $|U| \leq 7$ for a typical car). The results are reported in Fig. 10. As can be seen, when $|U|$ increases, the query processing time for all the algorithms increase. The query processing time for Bidirect-Bounded increases slowly, whereas the query processing time for the other three algorithms increases much more sharply. For example, in the *USA* dataset, when $|U|$ increases from 3 to 7, the query processing time for Basic, Grow, Bidirect, and Bidirect-Bounded increases by 46.9, 38.5, 9.8, and 1.6 times respectively. When $|U| = 7$, Bidirect-Bounded is an order of magnitude faster than Bidirect, more than two orders of magnitude faster than Grow, and three orders of magnitude faster than Basic. The results show the high scalability of Bidirect-Bounded with increasing $|U|$.

Exp-5: Vary α. In this experiment, we vary α from 0.2 to 0.6, because α is typically not very large in practice. The results are shown in Fig. 11. We can find that when α increases, the processing time of the algorithms Basic, Grow, and Bidirect are relatively stable. For the Bidirect-Bounded algorithm, when $\alpha \leq 1/3$, the processing time keeps stable. This is because, when $\alpha \leq 1/3$, Bidirect-Bounded can output the result directly after invoking Algorithm 4. However, when $\alpha > 1/3$, the processing time of Bidirect-Bounded increases with increasing α. The reason is as follows. When $\alpha > \frac{1}{3}$ and it is close to $\frac{1}{3}$, by equ. (9), the lower bound used in Bidirect-Bounded is close to the optimal cost, resulting in a good pruning power for Bidirect-Bounded. Thus, a small α is more preferable for pruning. Note that even for a large α (e.g., $\alpha = 0.6$), Bidirect-Bounded is still much faster than all the other

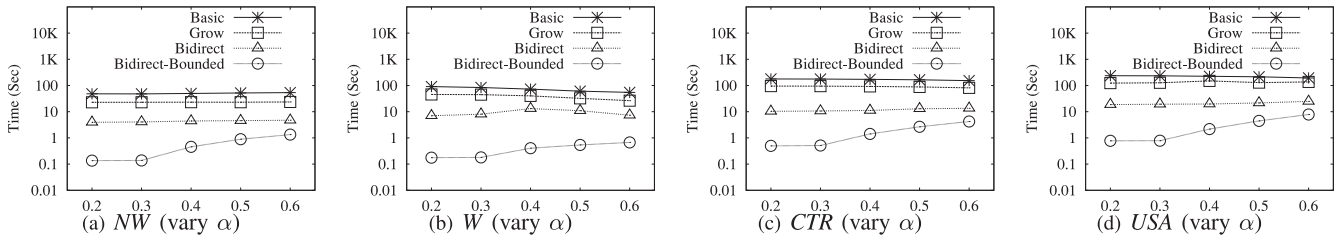


Fig. 11. Vary α.

algorithms in all the datasets. These results further confirm the theoretical analysis shown in the previous sections.

Exp-6: Testing walking distance. In this experiment, we study how the parameter α affects the ratio of the average walking distance of the query nodes in U (i.e., $d(U, V_{P_{st}})/|U|$) to the length of the optimal route. Let WR be the ratio of the average walking distance of the query nodes to the length of the optimal route. Fig. 12 reports the results in the USA dataset under the default parameter setting (for the parameters in Table 2 except α). For the other datasets and other parameter settings, we have similar observations and we omit the details due to space limitation. From Fig. 12, we can see that when $\alpha \leq 1/3$, we have $WR = 0$. This is because when $\alpha \leq 1/3$, the optimal route must pass through all the query nodes, and the walking distances of the query nodes are 0. On the other hand, we can see that when $\alpha > 1/3$, WR increases with increasing α as desired. Compared to the route distance, the average walking distance is very small, which is nearly two orders of magnitude smaller than the route distance. For example, when $\alpha = 0.4$, we have $WR = 0.022$. In this setting, the route distance is 45 times larger than the average walking distance. These results justify the motivation of our work. In addition, it should be noted that in real-world applications, we can always set an appropriate α value to make WR within a reasonable range. Based on the results shown in Fig. 12, we recommend to set $\alpha \in [1/3, 0.4]$, because in these cases, WR is within a reasonable range ($WR \in [0, 0.022]$) for many real-world applications.

6 Conclusion

This work presents a comprehensive study on optimal multi-meeting-point route query in road networks motivated by the real-time ride-sharing application, which aims at finding the best route starting from s and ending at t such that the weighted average cost between the cost of the route and the total cost of the shortest paths from the query nodes to the route is minimized. We prove that the problem of answering the OMMPR query is NP-hard. To solve the problem, we propose two fixed-parameter tractable algorithms, called Basic and Grow, based on dynamic programming. The time complexities of Basic and Grow rely on the number of query nodes, which is typically very small in practice. To further improve the efficiency of our algorithms, we propose two novel optimized algorithms based on bidirectional DP and a carefully-developed lower bounding technique. Extensive experiments over four large real-world road networks confirm the efficiency of the proposed algorithms.

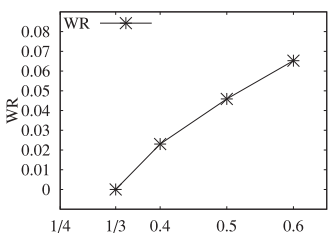


Fig. 12. α versus average walking distance ratio WR.

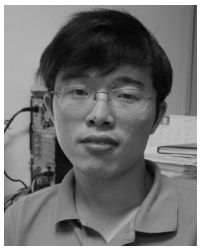
Acknowledgments

The work was supported in part by (i) NSFC Grants (61402292, 61170076, U1301252, 61033009), Natural Science Foundation of SZU (grant no. 201438), and Startup Grant (no. 827/000065) of Shenzhen Kongque Program; (ii) Research Grants Council of the Hong Kong SAR, China, 14209314 and 418512; (iii) China 863 (no. 2012AA010239) and Guangdong Key Laboratory Project (2012A061400024); (iv) ARC DE140100999; and (v) National Key Technology Research and Development Program of the Ministry of Science and Technology of China (2014BAH28F05). Rui Mao is a corresponding author.

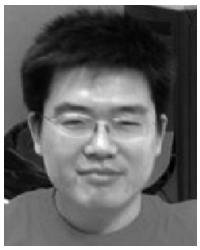
References

[1] S. Ma, Y. Zheng, and O. Wolfson, "T-share: A large-scale dynamic taxi ridesharing service," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 410–421.
[2] S. Ma and O. Wolfson, "Analysis and evaluation of the slugging form of ridesharing," in *Proc. 21st ACM SIGSPATIAL/GIS Int. Conf. Adv. Geographic Inf. Syst.*, 2013, pp. 64–73.
[3] Y. Huang, F. Bastani, R. Jin, and X. S. Wang, "Large scale real-time ridesharing with service guarantee on road networks," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 2017–2028, 2014.
[4] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transp. Sci.*, vol. 46, no. 3, pp. 388–404, 2012.
[5] R. Geisberger, M. N. Rice, P. Sanders, and V. J. Tsotras, "Route planning with flexible edge restrictions," *ACM J. Exp. Algorithmics*, vol. 17, no. 1, pp. 1–20, 2012.
[6] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou, "Shortest path and distance queries on road networks: Towards bridging theory and practice," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 857–868.
[7] C. Sommer, "Shortest-path queries in static networks," *ACM Comput. Surveys*, vol. 46, pp. 1–31, 2014.
[8] R. G. Downey and M. Fellows, *Parameterized Complexity (1999 edition)*. New York, NY, USA: Springer, 1999.
[9] H. An, R. Kleinberg, and D. B. Shmoys, "Improving christofides' algorithm for the s-t path TSP," in *Proc. 44th Annu. ACM Symp. Theory Comput.*, 2012, pp. 875–886.
[10] F. Lam and A. Newman, "Traveling salesman path problems," *Math. Program.*, vol. 113, no. 1, pp. 39–59, 2008.
[11] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng, "On trip planning queries in spatial databases," in *Proc. 9th Int. Conf. Adv. Spatial Temporal Databases*, 2005, pp. 273–290.
[12] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi, "The optimal sequenced route query," *VLDB J.*, vol. 17, no. 4, pp. 765–787, 2008.

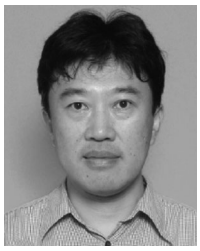
[13] R. Levin, Y. Kanza, E. Safra, and Y. Sagiv, "Interactive route search in the presence of order constraints," *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 117–128, 2010.
[14] H. Chen, W.-S. Ku, M.-T. Sun, and R. Zimmermann, "The partial sequenced route query with traveling rules in road networks," *GeoInformatica*, vol. 15, no. 3, pp. 541–569, 2011.
[15] R. Levin and Y. Kanza, "TARS: Traffic-aware route search," *GeoInformatica*, vol. 18, no. 3, pp. 461–500, 2014.
[16] M. Sharifzadeh and C. Shahabi, "Processing optimal sequenced route queries using voronoi diagrams," *GeoInformatica*, vol. 12, no. 4, pp. 411–433, 2008.
[17] X. Cao, L. Chen, G. Cong, and X. Xiao, "Keyword-aware optimal route search," *Proc. VLDB Endowment*, vol. 5, no. 11, pp. 1136–1147, 2012.
[18] R. Geisberger, D. Luxen, S. Neubauer, P. Sanders, and L. Völker, "Fast detour computation for ride sharing," in *Proc. ATMOS*, 2010, pp. 88–99.
[19] F. Drews and D. Luxen, "Multi-hop ride sharing," in *Proc. 6th Annu. Symp. Combinatorial Search*, 2013, pp. 71–79.
[20] Z. Xu and H.-A. Jacobsen, "Processing proximity relations in road networks," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 243–254.
[21] D. Yan, Z. Zhao, and W. Ng, "Efficient algorithms for finding optimal meeting point on road networks," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 968–979, 2011.
[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.
[23] S. E. Dreyfus and R. A. Wagner, "The steiner problem in graphs," *Networks*, vol. 1, no. 3, pp. 195–207, 1971.
[24] R. E. Erickson, C. L. Monma, and A. F. V. Jr., "Send-and-split method for Minimum-concave-cost network flows," *Math. Operations Res.*, vol. 12, no. 4, pp. 634–664, 1987.
[25] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto, "Fourier meets möbius: Fast subset convolution," in *Proc. 39th Annu. ACM Symp. Theory Comput.*, 2007, pp. 67–74.
[26] B. Fuchs, W. Kern, and X. Wang, "Speeding up the Dreyfus-wagner algorithm for minimum steiner trees," *Math. Meth. OR*, vol. 66, no. 1, pp. 117–125, 2007.
[27] B. Fuchs, W. Kern, D. Mölle, S. Richter, P. Rossmanith, and X. Wang, "Dynamic programming for minimum steiner trees," *Theory Comput. Syst.*, vol. 41, no. 3, pp. 493–500, 2007.
[28] J. Vygen, "Faster algorithm for optimum steiner trees," *Inf. Process. Lett.*, vol. 111, no. 21-22, pp. 1075–1079, 2011.
[29] I. Pohl, "Bi-directional search," *Mach. Intell.*, vol. 6, pp. 124–140, 1971.
[30] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A* search meets graph theory," in *Proc. 16th Annu. ACM-SIAM Symp. Discr. Algorithm*, 2005, pp. 156–165.
[31] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.
[32] T. Ikeda, M.-Y. Hsu, and H. Imai, "A fast algorithm for finding better routes by AI search techniques," in *Proc. Vehicle Navigat. Inf. Syst. Conf.*, 1994, pp. 291–296.
[33] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A*," *J. ACM*, vol. 32, no. 3, pp. 505–536, 1985.



Rong-Hua Li received the PhD degree from the Chinese University of Hong Kong in 2013. He is currently an assistant professor at Shenzhen University, China. His research interests include algorithmic aspects of social network analysis, graph data management and mining, as well as sequence data management and mining.



Lu Qin received the bachelor's degree from the Department of Computer Science and Technology, Renmin University of China in 2006, and the PhD degree from the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong, in 2010. He is currently a postdoc research fellow in the Centre of Quantum Computation and Intelligent Systems (QCIS), University of Technology, Sydney (UTS). His research interests include parallel big graph processing, I/O efficient algorithms on massive graphs, and keyword search in relational database.



Jeffrey Xu Yu received the BE, ME, and PhD degrees in computer science from the University of Tsukuba, Japan, in 1985, 1987, and 1990, respectively. He has held teaching positions in the Institute of Information Sciences and Electronics, University of Tsukuba, and in the Department of Computer Science, Australian National University, Australia. He is currently a professor in the Department of Systems Engineering and Engineering Management, the Chinese University of Hong Kong, Hong Kong. He is serving as a *VLDB Journal* Editorial Board Member. His current research interests include graph database, graph mining, keyword search in relational databases, and social network analysis.

Rui Mao received the PhD degree in computer science from the University of Texas at Austin, TX, in 2007. He is currently an associate professor at Shen Zhen University, China. His current research interests include big data analysis and management, content-based similarity query of multimedia and biological data, and data mining.