Effective and Efficient: Large-scale Dynamic City Express

Siyuan Zhang, Lu Qin, Yu Zheng, *Senior Member, IEEE,* and Hong Cheng

Abstract—Due to the large number of requirements for city express services in recent years, the current city express system is found to be unsatisfactory for both the service providers and customers. In this paper, we are the first to systematically study the large-scale dynamic city express problem. We aim to increase both the effectiveness and the efficiency of the scheduling algorithm. The challenges of the problem stem from the highly dynamic environment, the NP-completeness with respect to the number of requests, and real-time demands for the scheduling result. We introduce a basic algorithm to assign a request to a courier on a first-come, first-served basis. To improve the effectiveness of the basic algorithm, we adopt a batch assignment strategy that computes the pickup-delivery routes for a group of requests received in a short period rather than dealing with each request individually. To improve the efficiency of the algorithm, we further design a two-level priority queue structure to reduce redundant shortest distance calculation and repeated candidate generation. We develop a simulation system and conduct extensive performance studies on the real road network of Beijing city. The experimental results demonstrate the high effectiveness and efficiency of our algorithms. Remarkably, our system can achieve much better service quality and largely reduce the operation cost of a city express company simultaneously.

Index Terms—City express service, logistics, batch assignment.

1 Introduction

With the development of logistics industry and the rise of E-commerce, city express services have become increasingly popular in recent years [1]. We demonstrate how current city express systems work in Fig. 1: A city is divided into several regions (e.g., R_1 and R_2), each of which covers some streets and neighborhoods. A transit station is built in a region to temporarily store the parcels received in the region (e.g., ts_1 in R_1). The received parcels in a transit station are further organized into groups according to their destinations. Each group of parcels will be sent to a corresponding transit station by trucks regularly (e.g., from ts_1 to ts_2). In each region, there are a team of couriers delivering parcels to and receiving parcels from specific locations in the region. When a truck carrying parcels arrives at a transit station, each courier will send a portion of these parcels to their final destinations by a small delivery van, or a bike, or a motorcycle, which has a limited capacity. Before departing from the transit station, they will pre-compute the delivery routes (e.g., the blue lines in Fig. 1) usually based on their own knowledge. During the delivery, each courier could receive pickup requests (e.g., r_5, r_6 and r_7) from a central dispatch system or directly from end users. Each pickup request is associated with a location and a deadline of pickup time. A courier may change the originally planned route to fetch the new parcels, or decline the pickup request due to the constraints in their schedule or their vehicle capacity. All the couriers are required to return to their own transit station by some specific time (so as to fit the schedule of trucks that travel between transit stations regularly), or when fully loaded.

The service quality and operational efficiency of current express services have never been found satisfactory due to the following three reasons. *First*, current central dispatch systems process each pickup request individually without a global optimization. For example, a new pickup request

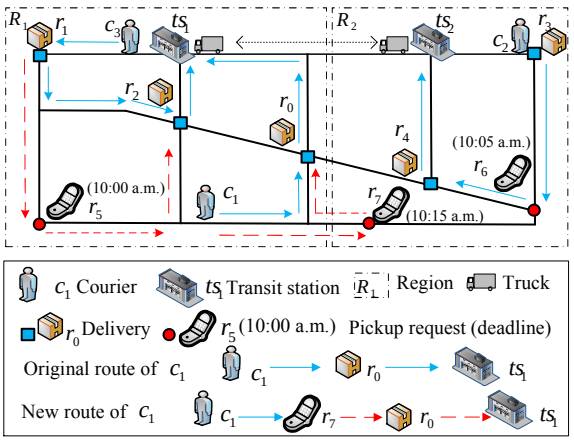


Fig. 1: Dynamic City Express

(e.g., r_5) is usually assigned to the nearest courier (e.g., c_1) to the pickup location. In the meantime, each courier makes a decision on whether to pick up a new parcel solely based on his own situation without knowing the status of other couriers in the same region (e.g., c_3 can pick up a parcel at r_5 instead of c_1). *Second*, the dispatch systems do not know the current status of a courier either, e.g., the remaining capacity, the number of parcels that have not been delivered, and the following pickup-delivery route, before assigning a new pickup request. Note that these statuses would change dynamically due to the new pickup requests. *Third*, requests near the boundary of a region (e.g., r_7) are ignored by couriers (e.g., c_1) in other regions because couriers only pick up parcels in their own regions.

Motivated by the huge number of requirements and drawbacks in current city express systems, in this paper, we study the dynamic city express problem and aim to design a central dispatch system with an effective scheduling algorithm for couriers to deliver and pick up parcels in real time.

Each courier in the system carries a handheld device recording his location, uploading his status when he finishes a delivery or pickup task, and receiving new pickup requests. The central dispatch system receives information of couriers from their handheld devices and manages their schedules consisting of pickup and delivery time and routes. After collecting pickup requests from customers in a short period, the system processes the requests in a batch according to the minimum incurred distance. Finally, the system sends the updated schedules to all the couriers and the confirmation or decline messages to the customers.

Challenges. In the literature, related problems such as vehicle routing with time window [2], [3], [4], [5], [6], and taxi ridesharing [7], [8], [9] have been studied. However, the city express problem is challenging due to the following reasons.

(1) Real-time demand and dynamic scheduling for a large number of requests and couriers. The real-time property of the city express problem requires the couriers to adjust their route dynamically when new pickup requests are issued. Meanwhile, the schedules of the couriers change dynamically as couriers finish a delivery or pickup task. However, most existing solutions for vehicle routing problem with time window are based on the static assumption [2], [3] where all requests are given in advance. Moreover, existing solutions for dynamic vehicle routing with time window [4], [5], [6] can only handle a small number of requests per hour (less than 100). In fact, thousands of pickup requests can be issued within an hour in real time. A city express system should work out the schedule of all couriers in a short time before the gathered information of couriers and requests is outdated.

(2) Exponential search space to find a feasible schedule. As shown in Section 2.2, the city express problem is an NP-complete problem even if all requests are given in advance. The optimal algorithm is usually with time complexity exponential to the capacity of a courier, i.e., the maximum number of parcels that can be carried by a courier. On the other hand, the spatio-temporal constraints of the city express problem are looser than those of the taxi ridesharing problem [7], [8], [9], which studies the routing problem of taxis under specific time windows. More candidate couriers for a pickup request and more feasible scheduling possibilities of a courier exist in the city express problem. It reduces the pruning power of existing spatio-temporal index [7], [8] and search space index [9].

Contributions. In this paper, we tackle the above challenges and make the following contributions.

(1) Systematic study of large-scale dynamic city express problem. We formally define the dynamic city express problem which involves the scheduling of multiple couriers to serve pickup requests and delivery tasks in real time. To the best of our knowledge, this is the first work that systematically studies the large-scale dynamic city express problem on road networks.

(2) High effectiveness using incurred distance and batch assignment. We design a basic algorithm that handles pickup requests on a first-come, first-served basis, and assigns a new request to the courier with the minimum incurred distance (the increased cost for a courier to serve the request). Such a strategy has been proven to be effective for the Travelling Salesman Problem (TSP) [10], [11], [12], which is a relaxation of the city express problem. We further observe that the short response time from issuing the request to the confirmation of the request allows the system to gather multiple requests and assign the requests in a batch mode other than individually. Therefore, we propose a batch assignment algorithm to improve the effectiveness of the basic solution.

(3) High efficiency using space reduction strategies. We further improve the computational efficiency of our solution based on a two-level priority queue structure. The proposed algorithm, SIDF$*$, can reduce redundant shortest distance computation by utilizing a global priority queue, and avoid repeated candidate generation by maintaining a local priority queue for each courier.

(4) Extensive performance studies. We conduct extensive performance studies with a city express simulation system on the real Beijing city road network. The experimental results show that our proposed algorithms can achieve both high effectiveness and efficiency.

Outline. Section 2 provides the preliminaries, formally defines the dynamic city express problem, and proves the complexity of the problem. Section 3 introduces our basic solution based on the incurred distance with a simple lazy path computation strategy to reduce the computational cost. Section 4 presents a new solution to improve the effectiveness of the algorithm using batch assignment. Section 5 designs a two-level priority queue to improve the efficiency. Section 6 presents extensive experimental results on a real city road network. Section 7 reviews related work, and Section 8 concludes the paper.

2 Overview

2.1 Preliminaries

We model a road network as a directed weighted graph $G(V, E)$, where V is a set of nodes (road intersections), and E is a set of edges (roads). An edge $(u, v) \in E$ connects two nodes u and v in V. We also define I to be the set of intermediate nodes (with degree 2) which lie in edges. Each edge $(u, v) \in E$ is associated with a positive weight $w(u, v)$ denoting the time to travel along the edge. Note that travel distance can be easily converted to travel time when the average travel speed is given. Thus, for the simplicity of presentation, we use travel time to denote travel cost in the rest of the paper.

For any two nodes v_0 and v_k in $V \cup I$, a path from v_0 to v_k in G is a sequence $p = (v_0, v_1, v_2, \cdots, v_{k-1}, v_k)$ such that $v_i \in V$ for any $1 \leq i \leq k-1$, and (v_i, v_{i+1}) is an edge in G for any $0 \leq i \leq k-1$. Note that (v_0, v_1) (or (v_{k-1}, v_k)) can be a partial edge when $v_0 \in I$ (or $v_k \in I$). Given a path $p = (v_0, v_1, \cdots, v_k)$, the cost of the path, denoted as $\mathsf{cost}(p)$, is calculated as $\mathsf{cost}(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$. Given two nodes u and v in G, the shortest path from u to v is the path p_{min} with the minimum cost among all paths from u to v in G. We denote the corresponding cost of p_{min} as $\mathsf{cost}(u, v)$.

Definition 2.1: (Request) Given a road network $G(V, E)$, a *request* is denoted as $r = (l, d)$, where $l(r)$ is the location of

the request that lies in the road network G, and $d(r)$ is the deadline to pick up the parcel in the request. □

We suppose that each request has the same service time t_s, which is the time spent on serving a customer (e.g., filling the forms and wrapping up the parcels) for a courier. However, our proposed algorithms can easily handle requests with different service time. After a request is issued, a staff will contact the customer within a short period t_r to confirm or decline the request. If the request is accepted, it will be assigned to a courier.

Let $C = \{c_1, c_2, \cdots, c_n\}$ be the set of couriers. As described in introduction, each courier also has parcels for delivery on board. We use the same form for pickup request to represent a delivery task. The delivery tasks in each trip are assigned to the courier before the trip, and the pickup requests are assigned to the courier in real time during the trip. All the delivery tasks are required to be finished before their deadlines, whereas a new pickup request may or may not be satisfied. If it is not satisfied by any courier, the request is declined or asked to modify its deadline for consideration.

Definition 2.2: (Schedule) For a courier $c_i \in C$, a schedule for c_i, denoted as $S_i = (r_{i,1}, r_{i,2}, \cdots, r_{i,m_i})$, is a sequence of unserved tasks, such that if following the sequence to pick up/deliver the parcels, the courier can (1) arrive at the location $l(r_{i,j})$ before the deadline $d(r_{i,j})$ for every $1 \leq j \leq m_i$, and (2) return to the transit station after serving r_{i,m_i} within t_{max} time after setting off. □

Note that S_i only keeps the unserved tasks. Once a task is served by courier c_i, it is removed from the schedule of c_i. Suppose $l(c_i)$ is the current location of courier c_i and $ts(c_i)$ is the location of the transit station where c_i sets off. Given the schedule S_i for courier c_i, the cost of S_i, denoted as $\mathsf{cost}(S_i)$, is calculated as:

$$\mathsf{cost}(S_i) = \mathsf{cost}(l(c_i), l(r_{i,1})) + \sum_{1 \leq j < m_i} \mathsf{cost}(l(r_{i,j}), l(r_{i,j+1})) + t_s \times m_i + \mathsf{cost}(l(r_{i,m_i}), ts(c_i)),$$

where $\mathsf{cost}(l(c_i), l(r_{i,1}))$ is the time spent on traveling from the current location of courier c_i to the location of the first task in the schedule S_i; $\sum_{1 \leq j < m_i} \mathsf{cost}(l(r_{i,j}), l(r_{i,j+1}))$ is time spent on traveling to the locations of all the tasks in S_i following the order in the schedule; $t_s \times m_i$ is the total service time for all the m_i tasks in S_i; and $\mathsf{cost}(l(r_{i,m_i}), ts(c_i))$ is the time to travel from the location of the last task to the transit station. For ease of representation, we use $l(r_{i,0})$ to denote $l(c_i)$ and use $l(r_{i,m_i+1})$ to denote $ts(c_i)$. Then $\mathsf{cost}(S_i)$ can be simplified as follows:

$$\mathsf{cost}(S_i) = \sum_{0 \leq j \leq m_i} \mathsf{cost}(l(r_{i,j}), l(r_{i,j+1})) + t_s \times m_i. \quad (1)$$

In other words, we treat the current location of c_i and the transit station of c_i as the locations of two special tasks $r_{i,0}$ and r_{i,m_i+1} without any service time. The deadline of the special task r_{i,m_i+1} can be calculated as $d(r_{i,m_i+1}) = t_{set}(c_i) + t_{\mathsf{max}}$, where $t_{set}(c_i)$ is the set-off time for courier c_i from the transit station $ts(c_i)$.

A list of notations used in this paper is summarized in Table 1.

Notation	Definition
$w(u, v)$	The weight of edge (u, v)
$\mathsf{cost}(u, v)$	The shortest traveling time from u to v
$r = (l, d)$	A request for city express
$l(r)$	The location of the request r
$d(r)$	The deadline to serve the request r
$C = \{c_1, ..., c_n\}$	The set of couriers
$S_i = \{r_{i,1}, ..., r_{i,m_i}\}$	The schedule for courier c_i
$\mathsf{cost}(S_i)$	The cost of the schedule S_i
$l(c_i)$ or $l(r_{i,0})$	The current location of courier c_i
$ts(c_i)$ or $l(r_{i,m_i+1})$	The location of the transit station of c_i
$t_{set}(c_i)$	The set-off time for c_i from $ts(c_i)$
t_s	Service time for a request
t_r	The maximum time to confirm/decline the request
t_{max}	The maximum time spent for a trip
t_{cur}	The current time
$a_j(S_i)$	The time for c_i to arrive $l(r_{i,j})$ in S_i
$\overline{d}_j(S_i)$	The latest time for c_i to arrive $l(r_{i,j})$ in S_i
$\mathsf{cand}(r)$	The candidate couriers to serve r
$\mathsf{cand}(c_i)$	The candidate requests that c_i can serve
$\underline{\mathsf{cost}}(u, v)$	A lower bound of $\mathsf{cost}(u, v)$
$\Delta\mathsf{dist}_j(r, S_i)$	The incurred distance of r to segment j of S_i
$\Delta\mathsf{dist}(r, S_i)$	The incurred distance of r to S_i
$\underline{\Delta\mathsf{dist}}_j(r, S_i)$	A lower bound of $\Delta\mathsf{dist}_j(r, S_i)$

TABLE 1: List of Notations

2.2 The Dynamic City Express Problem

In this paper, we study the dynamic city express problem (DCEP), which is defined as follows: *Given a set of n couriers $C = \{c_1, c_2, \cdots, c_n\}$, a set of delivery tasks at different transit stations, and a stream of pickup requests,* DCEP *aims to update the schedule for each courier dynamically as new pickup requests stream in, such that (1) all delivery tasks are satisfied, and (2) the pickup requests are satisfied as many as possible.*

The following lemma shows the complexity of the problem when all requests are given in advance. In the dynamic case, the problem becomes even more difficult to handle.

Lemma 2.1: *Given the set of couriers C and all requests in advance, the problem to decide whether a certain ratio P of requests can be satisfied by the couriers is an NP-complete problem.* □

Proof Sketch: Given the set of couriers C and all requests, we show that the problem to decide whether a certain percentage P of requests can be satisfied by the couriers is a generalization of the decision version of the Traveling Salesman Problem (TSP) which is proven to be NP-complete [10]. Given a set of m nodes and the travel cost between each pair of the m nodes, the problem is to decide whether it is possible to find a route shorter than L that visits every node once and returns to the original node. Given an instance of TSP, we can construct a corresponding instance of DCEP as follows.

(1) We construct a graph G of m nodes each of which corresponds to a node in TSP. For each pair of nodes in G, we add an edge in G, and the weight of the edge is the corresponding travel cost between the pair of nodes in TSP.
(2) We add m pickup requests on the m nodes respectively, and each request sets its deadline to be infinity.
(3) We set the number of couriers to be 1, P to be 100%, t_s to be 0, and t_{max} to be L. After the construction, solving the instance of TSP is exactly equivalent to solving the corresponding constructed instance of DCEP. This completes the proof. □

2.3 Solution Overview

We first introduce a basic solution in Section 3. After receiving a request r_1, the system identifies candidate couriers

that can possibly handle it using the courier index. Among the candidate couriers, the basic algorithm finds the one with the minimum incurred distance, and then updates the courier's schedule.

In contrast to the basic solution that handles requests on a first-come, first-served basis, our proposed solution in Section 4 collects a set of requests $\{r_1, r_2, ..., r_m\}$ that are issued within a short time, and processes them in a batch. The batch strategy allows more flexibility in assignment, thus can find more cost-effective pickup-delivery routes and potentially serve more requests. To improve the computational efficiency, we use a two-level priority queue structure described in Section 5 to reduce redundant shortest distance calculations and avoid repeated candidate generation.

In real practice, an express company may decline a request either because it is impossible to serve it or the service cost is too high [2]. Existing solutions [5], [7], [8], [9] consider accepting or declining a request once after it arrives, which share a similar idea of our basic solution. But in this work we will show that a batch processing mode can satisfy more requests than the basic solution.

3 The Basic Solution

In this section we first introduce a basic solution for DCEP. Note that our dynamic city express problem shares similar spatio-temporal constraints with the dynamic taxi ridesharing problem [7], [8], where a taxi ridesharing query asks if there exists a taxi in a city that can take the customer from his/her current location to a specific destination under time window and capacity constraints. We modify the existing method for taxi ridesharing service [8] to solve our problem and introduce a basic solution in this section. Our basic solution adopts the same framework of taxi ridesharing, which is processed as:

(1) A list of candidate taxis that can satisfy pickup and delivery time window constraints is first returned by a grid spatio-temporal index.
(2) The pickup and delivery locations are inserted into the schedule of a candidate taxi with the minimum incurred distance.

Based on the same framework, in this section, we first introduce the spatio-temporal index we use for candidate generation. Then we describe the basic solution to process a new request r in three steps: (1) identifying candidate couriers that can possibly serve r; (2) finding the courier that can serve r with the minimum incurred distance; and (3) updating the schedule of that courier. These components also serve as basic building blocks in our improved algorithms introduced in Sections 4 and 5. These components and the related notations are first introduced in this section for the ease of understanding.

3.1 Indexing

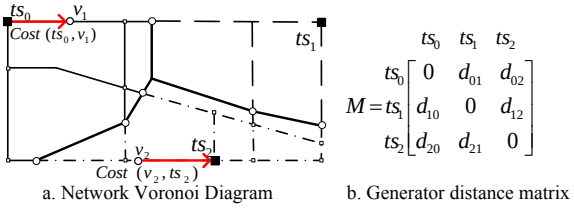
In order to compute the candidate couriers to serve a pickup request and avoid unnecessary shortest path computations, we build a Network Voronoi Diagram (NVD) to index all the couriers in the road network G. Specifically, we select the set of transit stations as the generators and build a NVD on the road network using the algorithm introduced in [13]. NVD partitions G into a set of Voronoi regions each of which is represented by a generator ts_i. Note that an edge $e \in E$ may belong to two different Voronoi regions. In this case, we add a new node in V that splits e into two edges, to ensure that each edge in E belongs to only one Voronoi region. For each location l in G, we use $ts(l)$ to denote the generator for the region that l lies in. For each node $v \in V$, we precompute $\mathsf{cost}(v, ts(v))$ and $\mathsf{cost}(ts(v), v)$. For any two generators ts_i and ts_j, we precompute $\mathsf{cost}(ts_i, ts_j)$. We also precompute the radius of each generator ts_i, denoted as $\mathsf{radius}(ts_i)$, which is the maximum cost from ts_i to any node in the Voronoi region of ts_i. With the NVD index, given any location l in G, suppose l lies on the edge (u, v), the distance from l to $ts(l)$ can be calculated as:

$$\begin{aligned}\mathsf{cost}(l, ts(l)) = \min\{&w(l,u) + \mathsf{cost}(u, ts(u)),\\ &w(l,v) + \mathsf{cost}(v, ts(v))\}.\end{aligned} \tag{2}$$

For any two locations l_i and l_j in G, the lower bound of $\mathsf{cost}(l_i, l_j)$ can be calculated as:

$$\begin{aligned}\underline{\mathsf{cost}}(l_i, l_j) = \max\{0, &\mathsf{cost}(ts(l_i), ts(l_j))\\ &- \mathsf{cost}(ts(l_i), l_i) - \mathsf{cost}(l_j, ts(l_j))\}.\end{aligned} \tag{3}$$

Obviously, with the NVD index, for any two locations l_i and l_j in G, $\underline{\mathsf{cost}}(l_i, l_j)$ can be calculated in constant time. Note that we can update the NVD index periodically to handle the dynamic update of travel cost in road network.



a. Network Voronoi Diagram b. Generator distance matrix

Fig. 2: The NVD Index

Example 3.1: Fig. 2 (a) shows the NVD index for a road network G with three transit stations ts_0, ts_1 and ts_2 as generators. Accordingly, G is partitioned into three regions by the bold lines. Roads in different Voronoi regions are depicted in different line styles. The distance matrix for generators is shown in Fig. 2 (b). The lower bound of $\mathsf{cost}(v_1, v_2)$ can be calculated as

$$\underline{\mathsf{cost}}(v_1, v_2) = \max\{0, d_{02} - \mathsf{cost}(ts_0, v_1) - \mathsf{cost}(v_2, ts_2)\}.$$

□

3.2 Candidate Courier Generation

Given a new request r, as the first step, we compute the set of candidate couriers for r, which is the set of couriers that can possibly arrive at the location $l(r)$ before the deadline $d(r)$ from their current locations. We define the candidate set $\mathsf{cand}(r)$ for a request r as follows.

$$\mathsf{cand}(r) = \{c_i | \underline{\mathsf{cost}}(l(r_{i,0}), l(r)) \le d(r) - t_{cur}\}. \tag{4}$$

Here, $\underline{\mathsf{cost}}(l(r_{i,0}), l(r))$ is calculated by Eq. 3 using the NVD index. In order to efficiently identify $\mathsf{cand}(r)$, we first compute the set of candidate Voronoi regions $\mathsf{candts}(r)$ as:

$$\mathsf{candts}(r) = \{ts_j | \underline{\mathsf{cost}}(l(ts_j), l(r)) - \mathsf{radius}(ts_j) \le d(r) - t_{cur}\}.$$

After computing $\mathsf{candts}(r)$, for each $ts_j \in \mathsf{candts}(r)$, we enumerate all couriers c_i that lie in the Voronoi region of ts_j and add c_i into $\mathsf{cand}(r)$ if $\underline{\mathsf{cost}}(l(r_{i,0}), l(r)) \le d(r) - t_{cur}$.

3.3 Incurred Distance Calculation

For each courier $c_i \in \mathsf{cand}(r)$, we insert request r into the schedule S_i of c_i and calculate the incurred distance for the insertion. Due to the spatio-temporal constraints of our problem, we only calculate the incurred distance of valid insertion. In the rest of this paper, we denote any two consecutive requests $r_{i,j}$ and $r_{i,j+1}$ as *segment j of S_i* ($0 \leq j \leq m_i$). We call a segment j in S_i a *valid* segment w.r.t. request r iff after inserting r between $r_{i,j}$ and $r_{i,j+1}$, the deadlines of all requests in the new S_i can be satisfied. Specifically, two conditions need to be satisfied:

(1) *Condition 1:* After inserting r into segment j of S_i, courier c_i can serve r before its deadline $d(r)$, which is formalized as:

$$a_j(S_i) + t_s + \mathsf{cost}(l(r_{i,j}), l(r)) \leq d(r) \tag{5}$$

where $a_j(S_i)$ is the courier's arrival time at $l(r_{i,j})$, and t_s is the service time for $r_{i,j}$.

(2) *Condition 2:* After inserting r into segment j of S_i, other requests $r_{i,k}$ with $j \leq k \leq m_i + 1$ on S_i can still be served on time, which can be formalized as:

$$\begin{aligned} a_j(S_i) + 2 \times t_s + \mathsf{cost}(l(r_{i,j}), l(r)) \\ + \mathsf{cost}(l(r), l(r_{i,j+1})) \leq \overline{d}_{j+1}(S_i) \end{aligned} \tag{6}$$

where we define $\overline{d}_{j+1}(S_i)$ as the latest time for c_i to arrive at $l(r_{i,j})$ in S_i, such that each request $r_{i,k}$ with $j \leq k \leq m_i + 1$ can be served before its deadline $d(r_{i,k})$. For each $0 \leq j \leq m_i + 1$, we can compute $\overline{d}_j(S_i)$ recursively as follows:

$$\overline{d}_j(S_i) = \begin{cases} d(r_{i,m_i+1}) & j = m_i + 1 \\ \min\{d(r_{i,j}), \overline{d}_{j+1}(S_i) & \\ -a_{j+1}(S_i) + a_j(S_i)\} & \text{otherwise} \end{cases} \tag{7}$$

Note that there is another condition that the number of parcels carried by a courier cannot exceed the capacity of the courier. Since such a condition is trivial to handle, we omit it for the ease of discussion in the rest of the paper.

After we find the valid segment(s) in S_i of courier c_i, we calculate the shortest incurred distance $\Delta\mathsf{dist}(r, S_i)$ when inserting r into the valid segment(s) of S_i. $\Delta\mathsf{dist}(r, S_i)$ is defined as follows:

$$\Delta\mathsf{dist}(r, S_i) = \min_{0 \leq j \leq m_i;\ \text{segment } j \text{ of } S_i \text{ is valid w.r.t. } r} \Delta\mathsf{dist}_j(r, S_i), \tag{8}$$

where $\Delta\mathsf{dist}_j(r, S_i)$ is the incurred distance if we insert request r between $r_{i,j}$ and $r_{i,j+1}$ in S_i, and is computed as:

$$\begin{aligned} \Delta\mathsf{dist}_j(r, S_i) = &\mathsf{cost}(l(r_{i,j}), l(r)) + \mathsf{cost}(l(r), l(r_{i,j+1})) \\ &- \mathsf{cost}(l(r_{i,j}), l(r_{i,j+1})). \end{aligned} \tag{9}$$

For each candidate courier in $\mathsf{cand}(r)$, we calculate the shortest incurred distance when inserting r into the valid segment(s) of his schedule. Among all candidates, we find the courier c_i with the minimum incurred distance $\Delta\mathsf{dist}(r, S_i)$ and update the schedule S_i by inserting r into the corresponding valid segment j. The values $a_j(S_i)$ and $\overline{d}_j(S_i)$ for $0 \leq j \leq m_i$ should be updated due to the insertion of r in S_i.

3.4 Lazy Path Computation

According to Eq. 9, calculating the incurred distance $\Delta\mathsf{dist}_j(r, S_i)$ involves calculating $\mathsf{cost}(l(r_{i,j}), l(r))$ and $\mathsf{cost}(l(r), l(r_{i,j+1}))$ using the costly Dijkstra's algorithm (or its variant A*) [14], with time complexity $O(m + n \cdot \log(n))$, on a road network G with n nodes and m edges. In order to minimize the number of exact shortest path computations, we compute a lower bound of $\Delta\mathsf{dist}_j(r, S_i)$ for any segment j as:

$$\begin{aligned} \underline{\Delta\mathsf{dist}}_j(r, S_i) = \max\{0, &\underline{\mathsf{cost}}(l(r_{i,j}), l(r)) + \underline{\mathsf{cost}}(l(r), l(r_{i,j+1})) \\ &- \mathsf{cost}(l(r_{i,j}), l(r_{i,j+1}))\}, \end{aligned}$$

where $\underline{\mathsf{cost}}(l, l')$ for any two locations l and l' can be calculated by Eq. 3 using the NVD index. The rationale is that if the $\underline{\Delta\mathsf{dist}}$ of a segment is large, we can prune it without computing its $\Delta\mathsf{dist}$. We call this strategy *lazy shortest path computation*. Accordingly, we design an algorithm BestR to find the best courier for a new request r using lazy shortest path computation, which is shown in Algorithm 1. We use a priority queue $\mathcal{H}$ to maintain the candidate segments. Each entry $(c_i, r_{i,j}, \Delta\mathsf{dist}, \mathsf{flag})$ in $\mathcal{H}$ denotes segment j in S_i with incurred distance $\Delta\mathsf{dist}$. flag is either true or false, denoting whether $\Delta\mathsf{dist}$ is the exact $\Delta\mathsf{dist}_j(r, S_i)$ or the lower bound $\underline{\Delta\mathsf{dist}}_j(r, S_i)$ respectively. $\mathcal{H}$ maintains the segment with the minimum $\Delta\mathsf{dist}$, and is initialized to be $\emptyset$ (line 1). For each candidate c_i in $\mathsf{cand}(r)$, we insert into $\mathcal{H}$ every segment j that satisfies conditions 1 (Eq. 5) and 2 (Eq. 6) by replacing cost with $\underline{\mathsf{cost}}$ (line 2-6). The incurred distance of each segment j in $\mathcal{H}$ is the lower bound $\underline{\Delta\mathsf{dist}}_j(r, S_i)$. At this stage, no exact distance has been computed. Next, we iteratively pop out the entry $(c_i, r_{i,j}, \Delta\mathsf{dist}, \mathsf{flag})$ in $\mathcal{H}$ with the minimum $\Delta\mathsf{dist}$. If $\Delta\mathsf{dist}$ is the lower bound $\underline{\Delta\mathsf{dist}}_j(r, S_i)$ (flag is false), we compute the exact incurred distance $\Delta\mathsf{dist}_j(r, S_i)$ using $\mathsf{cost}(l(r_{i,j}), l(r))$ and $\mathsf{cost}(l(r), l(r_{i,j+1}))$ (line 11), check the conditions 1 (Eq. 5) and 2 (Eq. 6) using the exact cost (line 12), and push the entry with the exact incurred distance into $\mathcal{H}$ if both conditions are satisfied (line 13). Otherwise, the segment (i, j) is the one with the minimum $\Delta\mathsf{dist}_j(r, S_i)$ and is returned by the algorithm (line 9-10). If no segment is returned when $\mathcal{H}$ becomes $\emptyset$ (line 7), we return $(-1, -1)$ indicating that request r cannot be satisfied.

Algorithm 1 BestR(request r, candidate set $\mathsf{cand}(r)$)

1: $\mathcal{H} \leftarrow \emptyset$;
2: **for all** $c_i \in \mathsf{cand}(r)$ **do**
3: compute $\underline{\mathsf{cost}}(l(r), l(r_{i,j}))$ for all $r_{i,j} \in S_i$;
4: **for all** segment j in S_i **do**
5: **if** ValidLB(S_i, j, r) **then**
6: $\mathcal{H}$.Push($(c_i, r_{i,j}, \underline{\Delta\mathsf{dist}}_j(r, S_i), \text{false})$);
7: **while** $\mathcal{H} \neq \emptyset$ **do**
8: $(c_i, r_{i,j}, \Delta\mathsf{dist}, \mathsf{flag}) \leftarrow \mathcal{H}$.Pop();
9: **if** flag $=$ true **then**
10: **return** (c_i, j);
11: compute $\mathsf{cost}(l(r_{i,j}), l(r))$ and $\mathsf{cost}(l(r), l(r_{i,j+1}))$;
12: **if** Valid(S_i, j, r) **then**
13: $\mathcal{H}$.Push($(c_i, r_{i,j}, \Delta\mathsf{dist}_j(r, S_i), \text{true})$);
14: **return** $(-1, -1)$;
15: **Procedure** ValidLB(schedule S_i, segment j, request r)
16: **return** $a_j(S_i) + t_s + \underline{\mathsf{cost}}(l(r_{i,j}), l(r)) \leq d(r)$ **and** $a_j(S_i) + 2t_s + \underline{\mathsf{cost}}(l(r_{i,j}), l(r)) + \underline{\mathsf{cost}}(l(r), l(r_{i,j+1})) \leq \overline{d}_{j+1}(S_i)$;
17: **Procedure** Valid(schedule S_i, segment j, request r)
18: **return** $a_j(S_i) + t_s + \mathsf{cost}(l(r_{i,j}), l(r)) \leq d(r)$ **and** $a_j(S_i) + 2t_s + \mathsf{cost}(l(r_{i,j}), l(r)) + \mathsf{cost}(l(r), l(r_{i,j+1})) \leq \overline{d}_{j+1}(S_i)$;

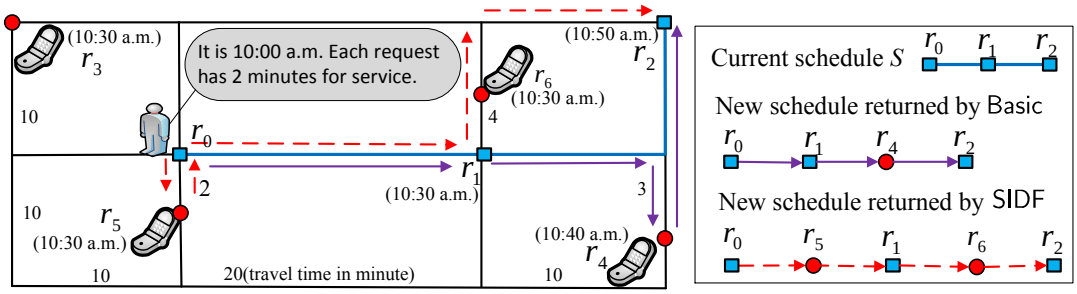


Fig. 3: An Example with One Courier and Four New Requests

3.5 Basic Algorithm

The overall algorithm is outlined in Algorithm 2. Suppose for each courier c_i, $a_j(S_i)$ and $\overline{d}_j(S_i)$ for $0 \le j \le m_i + 1$ have been computed respectively (line 1-2). When a new request r is received, the algorithm loads the updated locations of all couriers (line 4), and invokes Update(C, r) to find the courier $c_i \in C$ to serve r and update the corresponding schedule S_i (line 5). In Update(C, r) (line 6-11), as step 1, the algorithm computes cand(r) (line 7); as step 2, it finds the courier c_i and segment j in S_i with the minimum incurred distance by invoking procedure BestR (line 8) using lazy path computation; and as step 3, it either declines r if no valid segment is found (line 9-10) or updates the corresponding schedule using the Insert procedure (line 11). In the Insert(S_i, j, r) procedure (line 12-17), after inserting r between $r_{i,j}$ and $r_{i,j+1}$ (line 13), $a_k(S_i)$ for all $j + 1 \le k \le m_i + 1$ need to be updated in increasing order of k (line 14-15), and $\overline{d}_k(S_i)$ for all $1 \le k \le j + 1$ need to be updated in decreasing order of k (line 16-17).

Algorithm 2 Basic(couriers C and their current schedules)

```
1: for all courier c_i ∈ C do
2:     compute a_j(S_i) and d̄_j(S_i) for 0 ≤ j ≤ m_i + 1;
3: while there is a new request r issued in real time do
4:     load updated locations of all couriers;
5:     Update(C, r);
6: Procedure Update(courier set C, request r)
7: compute cand(r);
8: (c_i, j) ← BestR(r, cand(r));
9: if i = −1 then
10:    decline request r; return;
11: Insert(S_i, j, r);
12: Procedure Insert(schedule S_i, segment j, request r)
13: insert r between r_{i,j} and r_{i,j+1} of S_i;
14: for k = j + 1 to m_i + 1 do
15:    a_k(S_i) ← a_{k−1}(S_i) + t_s + cost(l(r_{i,k−1}), l(r_{i,k}));
16: for k = j + 1 down to 1 do
17:    d̄_k(S_i) ← min{d(r_{i,k}), d̄_{k+1}(S_i) − a_{k+1}(S_i) + a_k(S_i)};
```

4 Effectiveness: Batch Assignment

The Basic solution (Algorithm 2) adopts a first-come, first-served strategy in which early requests have high priority to be assigned. However, using request arrival time as the first priority may result in inferior scheduling result. We use an example depicted in Fig. 3 for explanation.

Example 4.1: A courier c has two tasks r_1 and r_2 in his schedule S. Four new requests r_3, r_4, r_5, r_6 arrive in sequence. The basic solution first considers r_3, but cannot satisfy it. Then it processes r_4 and updates the courier's schedule to be $r_0 \to r_1 \to r_4 \to r_2$. The next step is to check r_5 and r_6. But neither of them can be served by courier c due to the large incurred distance of r_4 to schedule S.

In contrast, if we process the four requests according to their incurred distance in a batch model, we can first satisfy r_6 because it has the minimum incurred distance among the four requests. Thus we update the schedule to be $r_0 \to r_1 \to r_6 \to r_2$. We perform another search and insert r_5 into the new schedule. After that there is no satisfiable request on the map. It is obvious that with incurred distance as the first priority, we can satisfy more requests with less incurred distance comparing to the basic solution. □

This example motivates us to consider processing a batch of requests using incurred distance as the first priority. Batch assignment is feasible in practice because the response period t_r allows the algorithm to collect multiple requests and assign them in any order. Moreover, as shown in [12], using the minimum incurred distance as node insertion order for TSP is a 2-approximation algorithm. By adopting this strategy, we propose a new algorithm SIDF, which stands for *shortest incurred distance first*.

SIDF is outlined in Algorithm 3. After computing $a_j(S_i)$ and $\overline{d}_j(S_i)$ for the current schedule S_i of each courier $c_i \in C$ (line 1-2), the algorithm loads the updated information of all couriers, collects and processes the set of requests R issued in every t_r minutes (line 3-4). Two steps, candidate computation (line 6) and schedule updating (line 7), are used to process the requests in R. In candidate computation, the set of candidate requests for each courier and the set of candidate couriers for each request are computed. In schedule updating, the requests are processed in the increasing order of their minimum incurred distances to couriers.

Candidate Computation. The procedure ComputeCand to compute the candidates is shown in line 8-13 of Algorithm 3. We first compute the candidates cand(r) for each request $r \in R$ (line 10-11) using the same method in Algorithm 2. Then we compute the candidate set cand(c_i), which is a set of requests that courier c_i can reach before their deadlines, for each $c_i \in C$. cand(c_i) can be considered as an inverse set of cand(r), thus, for each $c_i \in$ cand(r), we simply add r into cand(c_i) (line 12-13).

Schedule Updating. In the schedule updating procedure UpdateR (line 14-29 of Algorithm 3), we use a global priority queue $\mathcal{H}^g$ to maintain the priorities of all requests. Each entry in $\mathcal{H}^g$ has the form:

$$(r, c_i, r_{i,j}, r_{i,j+1}, \Delta\text{dist}_j(r, S_i)),$$

which means that the minimum incurred distance for the request r is $\Delta\text{dist}_j(r, S_i)$ when we insert r into segment $r_{i,j} \to r_{i,j+1}$ of c_i's schedule S_i. We compute the initial priorities of requests in R (line 16-19), and assign the requests according to their priorities in $\mathcal{H}^g$ (line 20-28). The priority of a request r is the minimum $\Delta\text{dist}(r, S_i)$ among the schedules of all candidate couriers, which can be computed using the procedure BestR$(r, \text{cand}(r))$ (line 17).

After computing the priorities of all requests and adding them into $\mathcal{H}^g$ (line 18-19), we pop out the request r with the highest priority (minimum Δdist) from $\mathcal{H}^g$ iteratively, assign r to courier c_i by inserting it into the corresponding segment j of S_i (line 22), and remove r from the unprocessed request set R (line 23). After a request r is assigned to segment j of S_i, the priority of each request $r' \in \mathsf{cand}(c_i)$ needs to be updated in $\mathcal{H}^g$ (line 24-28). This is processed by removing r' from $\mathcal{H}^g$ (line 25), calculating the updated priority (line 26), and inserting it again into $\mathcal{H}^g$ if it is still valid (line 27-28). Here, after inserting r into segment j of schedule S_i, for each request $r' \in \mathsf{cand}(c_i)$, the priority of r' needs to be recalculated due to the following three factors:

(1) *Segment deletion.* The original segment j is removed from S_i. Therefore, the priority of a request $r' \in \mathsf{cand}(c_i)$ may decrease due to the removal.
(2) *Segment insertion.* Two new segments j and $j+1$ are created in S_i. Therefore, the priority of a request $r' \in \mathsf{cand}(c_i)$ may increase due to the insertion of two new segments.
(3) *Segment updating.* The values of $a_k(S_i)$ (for $j+1 \leq k \leq m_i+1$) and $\overleftarrow{d}_k(S_i)$ (for $1 \leq k \leq j+1$) are updated after the insertion of r. Therefore, the priority of a request $r' \in \mathsf{cand}(c_i)$ needs to be updated accordingly because some segments become invalid w.r.t. r'.

The algorithm terminates when all possible requests in $\mathcal{H}^g$ are assigned. The remaining unassigned requests in R are declined (line 29).

Algorithm 3 SIDF(couriers C and their current schedules)

```
1: for all courier c_i ∈ C do
2:     compute a_j(S_i) and d←_j(S_i) for 0 ≤ i ≤ m_i + 1;
3: for every t_r minutes do
4:     load updated locations of all couriers;
5:     R ← the set of all requests issued in the last t_r minutes;
6:     ComputeCand(C, R);
7:     UpdateR(C, R);
8: Procedure ComputeCand(courier set C, request set R)
9: cand(c_i) ← ∅ for all c_i ∈ C;
10: for all r ∈ R do
11:     compute cand(r);
12:     for all c_i ∈ cand(r) do
13:         cand(c_i) ← cand(c_i) ∪ {r};
14: Procedure UpdateR(courier set C, request set R)
15: H^g ← ∅;
16: for all r ∈ R do
17:     (c_i, j) ← BestR(r, cand(r));
18:     if i ≠ −1 then
19:         H^g.Push((r, c_i, r_{i,j}, r_{i,j+1}, Δdist_j(r, S_i)));
20: while H^g ≠ ∅ do
21:     (r, c_i, r_{i,j}, r_{i,j+1}, Δdist) ← H^g.Pop();
22:     Insert(S_i, j, r);
23:     R ← R \ {r};
24:     for all r' ∈ cand(c_i) and r' ∈ H^g do
25:         H^g.Remove(r');
26:         (c_{i'}, j') ← BestR(r', cand(r'));
27:         if i' ≠ −1 then
28:             H^g.Push((r', c_{i'}, r_{i',j'}, r_{i',j'+1}, Δdist_{j'}(r', S_{i'})));
29: decline all requests in R;
```

5 Efficiency: Two-Level Priority Queue Structure

SIDF improves the effectiveness of request assignment with a batch mode, but it is computationally expensive due to redundant shortest distance computations. Specifically, the redundant computations come from two aspects. *First*, in the UpdateR procedure, when a request r is assigned to courier c_i (line 22), its schedule S_i is updated. Then it invokes $\mathsf{BestR}(r', \mathsf{cand}(r'))$ to compute a new assignment for every request $r' \in \mathsf{cand}(c_i)$ from scratch, which involves the costly shortest distance computation for every $r' \in \mathsf{cand}(c_i)$ w.r.t. the schedule of every candidate courier $c' \in \mathsf{cand}(r')$. *Second*, SIDF only makes a new assignment after all the exact incurred distances of all requests are computed, which may not be necessary as some requests with very large distances to some couriers can be easily pruned.

In this section, we aim to improve the efficiency of our solution with a new algorithm SIDF$*$ from two aspects. *First*, we observe that once a request r is assigned to the schedule S_i of a courier c_i, only those requests in $\mathsf{cand}(c_i)$ may be affected in terms of their incurred distance to S_i. The schedules of other couriers are not affected by this assignment. Thus we can confine the update to courier c_i. *Second*, we design a mechanism to delay and reduce the exact shortest distance computation as much as possible, while at the same time, still maintain the same effectiveness of SIDF. With these considerations, we design a two-level priority queue structure, which is illustrated in Fig. 4. At the bottom level, for each courier c_i, we maintain a local priority queue $\mathcal{H}_i$ that keeps all the candidate requests $\mathsf{cand}(c_i)$; and at the top level, we maintain a global priority queue $\mathcal{H}_g$ that keeps the topmost entries popped from each local priority queue. With the two-level priority queue structure, after a new request r is assigned to a courier c_i, we only need to update the local priority queue $\mathcal{H}_i$ and the global priority queue incrementally. For other couriers that also contain r in their local priority queues, they can discard r in a lazy manner. In this way, the total number of shortest distance computation can be largely reduced.

In the following, we first introduce the components of the two-level priority queue before describing the new scheduling algorithm SIDF$*$.

5.1 Local Priority Queue for Courier

For each courier c_i, we maintain a local priority queue $\mathcal{H}_i$ that keeps all the candidate requests $\mathsf{cand}(c_i)$. Each entry in the local priority queue $\mathcal{H}_i$ has the form:

$$(r, c_i, r_{i,j}, r_{i,j+1}, \Delta\mathsf{dist}, \mathsf{flag})$$

It represents a possible assignment that assigns a request r into segment $r_{i,j} \rightarrow r_{i,j+1}$ of c_i's schedule. Different from the entry in $\mathcal{H}^g$ of SIDF, a flag is added in each entry indicating the incurred distance Δdist is an exact value (if flag $=$ true) or a lower bound (if flag $=$ false).

The operations on the local priority queue include the following.

1. Initialization. In the initialization step, for each request $r \in \mathsf{cand}(c_i)$, and each valid segment j in S_i, we insert an entry $(r, c_i, r_{i,j}, r_{i,j+1}, \underline{\Delta\mathsf{dist}}_j(r, S_i), \text{false})$ to $\mathcal{H}_i$. Here we only calculate the lower bound of the incurred distance, but not the exact incurred distance.

2. Pop. We pop the topmost entry in $\mathcal{H}_i$ and push it into the global priority queue $\mathcal{H}_g$. The rationale is, the request in the top entry of a local priority queue is more likely to have a short incurred distance. Thus it will be first pushed into the global priority queue for further verification.

3. Push. There are two cases that we will push an entry into the local priority queue $\mathcal{H}_i$.

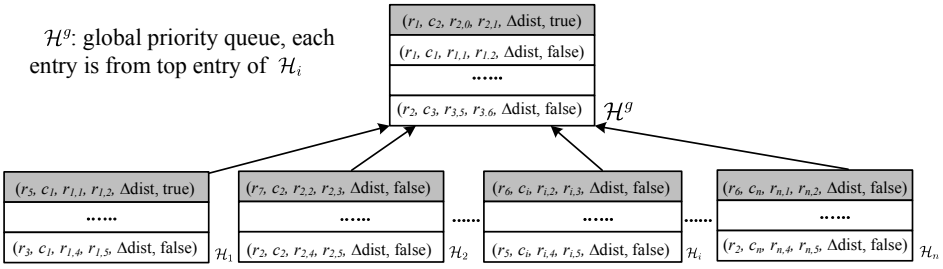


Fig. 4: Global Priority Queue and Local Priority Queues

(1) For a request r and a valid segment j in S_i, once we calculate the exact incurred distance $\Delta\mathsf{dist}_j(r, S_i)$, we push the entry $(r, c_i, r_{i,j}, r_{i,j+1}, \Delta\mathsf{dist}_j(r, S_i), \text{true})$ back into $\mathcal{H}_i$ for further comparison with other candidate requests in the queue.
(2) When S_i is updated with a new request inserted, there may be some requests that can possibly be served by the courier c_i due to this insertion. Accordingly, we push the entries for those new candidates with the lower bound of their incurred distance into the queue $\mathcal{H}_i$.

5.2 Global Priority Queue

On top of the local priority queues for couriers, we maintain a global priority queue $\mathcal{H}^g$ that keeps the topmost entries popped from each local priority queue. The entries in $\mathcal{H}^g$ have the same representation as those in local priority queues. The operations on the global priority queue include the following.

1. Initialization. In the initialization step, the topmost entry from each local priority queue is popped out and pushed into $\mathcal{H}^g$.

2. Pop. The top entry $(r, c_i, r_{i,j}, r_{i,j+1}, \Delta\mathsf{dist}, \mathsf{flag})$ is popped out from $\mathcal{H}^g$. We will first check whether the request r is not assigned yet and whether the concerned segment $r_{i,j} \rightarrow r_{i,j+1}$ is still valid (i.e., not altered by any earlier assignment). If the entry is still valid, we process it in one of the following two cases.

(1) If $\mathsf{flag} = \text{true}$, we will assign the request into the concerned segment j and update courier c_i's schedule S_i.
(2) If $\mathsf{flag} = \text{false}$, we will calculate the exact incurred distance $\Delta\mathsf{dist}_j(r, S_i)$ and push this entry back to the local priority queue $\mathcal{H}^i$ for further comparison.

3. Push. After the top entry is popped out from $\mathcal{H}^g$, a new entry related to the same courier is popped out from the corresponding local priority queue and pushed into $\mathcal{H}^g$.

5.3 Improved Algorithm SIDF∗

The new scheduling algorithm SIDF∗ is shown in Algorithm 4. We describe how it works in 5 steps.

Step 1. We initialize the global priority queue $\mathcal{H}^g$ and the local priority queue $\mathcal{H}_i$ for each courier $c_i \in C$ (line 1-6). The initialization of $\mathcal{H}_i$ is done in procedure InitH (line 20-27), which computes the candidate entries for c_i and pushes each candidate along with the lower bound of its incurred distance into the priority queue. To initialize $\mathcal{H}^g$, we simply pop out the top element from each $\mathcal{H}_i$ (line 5) and push it into $\mathcal{H}^g$ (line 6). We add a flag in each entry of $\mathcal{H}^g$ to mark whether the entry keeps the exact Δdist.

Algorithm 4 SIDF∗ (courier set C, request set R)

1: $\mathcal{H}^g \leftarrow \emptyset$;
2: **for all** $c_i \in C$ **do**
3: $\quad\mathcal{H}_i \leftarrow \mathsf{InitH}(c_i, R)$;
4: $\quad$**if** $\mathcal{H}_i \neq \emptyset$ **then**
5: $\quad\quad(r, c_i, r_{i,j}, r_{i,j+1}, \Delta\mathsf{dist}_j(r, S_i), \mathsf{flag}) \leftarrow \mathcal{H}_i.\mathsf{Pop}()$;
6: $\quad\quad\mathcal{H}^g.\mathsf{Push}((r, c_i, r_{i,j}, r_{i,j+1}, \Delta\mathsf{dist}_j(r, S_i), \mathsf{flag}))$;
7: **while** $\mathcal{H}^g \neq \emptyset$ **do**
8: $\quad(r, c_i, r_{i,j}, r_{i,j+1}, \Delta\mathsf{dist}, \mathsf{flag}) \leftarrow \mathcal{H}^g.\mathsf{Pop}()$;
9: $\quad$**if** $r \in R$ **and** $r_{i,j} \rightarrow r_{i,j+1}$ is still a segment in S_i **then**
10: $\quad\quad$**if** $\mathsf{flag} = \text{false}$ **and** $\mathsf{ValidLB}(S_i, j, r)$ **then**
11: $\quad\quad\quad$compute $\mathsf{cost}(l(r_{i,j}), l(r))$ and $\mathsf{cost}(l(r), l(r_{i,j+1}))$;
12: $\quad\quad\quad$**if** $\mathsf{Valid}(S_i, j, r)$ **then**
13: $\quad\quad\quad\quad\mathcal{H}_i.\mathsf{Push}((r, c_i, r_{i,j}, r_{i,j+1}, \Delta\mathsf{dist}_j(r, S_i), \text{true}))$;
14: $\quad\quad$**if** $\mathsf{flag} = \text{true}$ **and** $\mathsf{Valid}(S_i, j, r)$ **then**
15: $\quad\quad\quad\mathsf{Assign}(S_i, j, r, R, \mathcal{H}_i)$;
16: $\quad$**if** $\mathcal{H}_i \neq \emptyset$ **then**
17: $\quad\quad(r', c_i, r_{i,j'}, r_{i,j'+1}, \Delta\mathsf{dist}_{j'}(r', S_i), \mathsf{flag}') \leftarrow \mathcal{H}_i.\mathsf{Pop}()$;
18: $\quad\quad\mathcal{H}^g.\mathsf{Push}(r', c_i, r_{i,j'}, r_{i,j'+1}, \Delta\mathsf{dist}_{j'}(r', S_i), \mathsf{flag}')$;
19: decline all requests in R;
20: **Procedure** InitH(courier c_i)
21: $\mathcal{H} \leftarrow \emptyset$;
22: **for all** $r \in \mathsf{cand}(c_i)$ **do**
23: $\quad$compute $\underline{\mathsf{cost}}(l(r), l(r_{i,j}))$ for all $r_{i,j} \in S_i$;
24: $\quad$**for all** segment j in S_i **do**
25: $\quad\quad$**if** $\mathsf{ValidLB}(S_i, j, r)$ **then**
26: $\quad\quad\quad\mathcal{H}.\mathsf{Push}((r, c_i, r_{i,j}, r_{i,j+1}, \underline{\Delta\mathsf{dist}}_j(r, S_i), \text{false}))$;
27: **return** $\mathcal{H}$;
28: **Procedure** Assign(schedule S_i, segment j, request r, request set R, priority queue $\mathcal{H}$)
29: $\mathsf{Insert}(S_i, j, r)$;
30: $R \leftarrow R \setminus \{r\}$;
31: $\mathsf{InsertSeg}(\mathcal{H}, S_i, j, R)$;
32: $\mathsf{InsertSeg}(\mathcal{H}, S_i, j+1, R)$;
33: **Procedure** InsertSeg(priority queue $\mathcal{H}$,schedule S_i,segment j,request set R)
34: **for all** $r \in \mathsf{cand}(c_i) \cap R$ **do**
35: $\quad$compute $\underline{\mathsf{cost}}(l(r_{i,j}), l(r))$ and $\underline{\mathsf{cost}}(l(r), l(r_{i,j+1}))$;
36: $\quad$**if** $\mathsf{ValidLB}(S_i, j, r)$ **then**
37: $\quad\quad\mathcal{H}.\mathsf{Push}((r, c_i, r_{i,j}, r_{i,j+1}, \underline{\Delta\mathsf{dist}}_j(r, S_i), \text{false}))$;

Step 2. We pop the top element $(r, c_i, r_{i,j}, r_{i,j+1}, \Delta\mathsf{dist}, \mathsf{flag})$ (line 8) from the global priority queue $\mathcal{H}^g$ and attempt to assign request r to segment j in the schedule of courier c_i. Before the assignment, we first check whether r has not been assigned yet (i.e., $r \in R$) and segment $r_{i,j} \rightarrow r_{i,j+1}$ is still valid. If the entry is not valid at this step (line 9 returns false), we go to step 5. Otherwise, we go to step 3 if Δdist is a lower bound ($\mathsf{flag} = \text{false}$), or go to step 4 if we have exact Δdist.

Step 3. We need to check $\mathsf{ValidLB}(S_i, j, r)$ due to segment updating (line 10). If $\mathsf{ValidLB}(S_i, j, r)$, we compute the exact $\mathsf{cost}(l(r_{i,j}), l(r))$ and $\mathsf{cost}(l(r), l(r_{i,j+1}))$ (line 11), and push the entry with exact $\Delta\mathsf{dist}_j(r, S_i)$ into $\mathcal{H}_i$ if $\mathsf{Valid}(S_i, j, r)$ (line 12-13), as described in case 1 of the push operation of the local priority queue before.

Step 4. We need to check $\mathsf{Valid}(S_i, j, r)$ due to segment updating (line 14). If $\mathsf{Valid}(S_i, j, r)$, we can assign r to

segment j of S_i by invoking Assign$(S_i, j, r, R, \mathcal{H}_i)$. In the Assign$(S_i, j, r, R, \mathcal{H})$ procedure (line 28-32), in addition to inserting r into segment j of S_i (line 29) and removing r from R (line 30), we invoke InsertSeg$(\mathcal{H}, S_i, j, R)$ and InsertSeg$(\mathcal{H}, S_i, j+1, R)$ to push new entries into $\mathcal{H}_i$ due to the insertion of r (line 31-32), as described in case 2 of the push operation of the local priority queue before.

Step 5. We simply pop out the top entry from the corresponding local priority queue $\mathcal{H}_i$ and push it into $\mathcal{H}^g$ (line 16-18). Then we repeat step 2 to step 5 until $\mathcal{H}^g$ becomes $\emptyset$. Finally, all the remaining requests in R are declined (line 19).

5.4 Analysis of SIDF$*$

The benefits of keeping a local priority queue for each courier include:

(1) All the updating entries after we make a new assignment are contained in one local priority queue. Using a local priority queue avoids computing the new incurred distance of every affected request, which considers a lot of repeated possible entries due to the large number of candidate couriers.

(2) The local priority queue enables incremental entry updating. Once we make an assignment related to courier c_i, only new possible entries related to the two new segments in S_i are pushed into $\mathcal{H}_i$. Other valid entries with exact incurred distance w.r.t. c_i remain in $\mathcal{H}_i$, which avoids repeated exact incurred distance computation. No exact distance computation is performed in the local priority queue.

The benefit of keeping a global priority queue is, it supports lazy path computation among all assignments. We only compute the exact incurred distance of an assignment once it is popped out of $\mathcal{H}^g$, which reduces a lot of exact incurred distance computations.

Thus with the two-level priority queue structure, we can substantially improve the efficiency of the scheduling algorithm. Note that, SIDF$*$ can achieve the same effectiveness as SIDF, since it still adopts the batch assignment strategy and the *shortest incurred distance first* criterion for request assignment.

6 Performance Studies

Data set. We use a real road network in Beijing city for experimental study. We mainly focus on a 15km×5km area that lies between northeastern 4th ring road and 5th ring road of Beijing. The road network contains $8,840$ nodes and $11,331$ edges. In the scalability test, we use a series of road networks with different sizes.

Simulation with Parameter Settings. The set of parameters, their meanings, as well as the default values of all parameters are shown in Table 2, and illustrated in Fig. 5. In our experiments, when we vary a certain parameter, all other parameters are set to be their default values if not otherwise specified. Given a certain set of parameter setting, we develop a simulation system to simulate the city express process as follows.

(Transit Station and Courier Generation): We choose K nodes as transit stations by performing the k-medoids algorithm on all the nodes in the road network. Then we build a network Voronoi diagram with the transit stations as centers. The courier number in a transit station is proportional to the number of nodes in the corresponding Voronoi region, and the total number of couriers is n. All couriers are located at their corresponding transit stations at the beginning of the simulation.

Parameter	Meaning	Default
K	The number of transit stations	7
$\overline{\lambda}$	The average request intensity on a node	$0.6/hr$
n	The number of couriers	500
$\overline{t}_s$	The average service time	3 minutes
t_r	The maximum time to confirm a request	15 minutes
$t_{\max}$	The maximum time for each trip	120 minutes
m_d	The number of delivery tasks in $t_{\max}$	$1,600$
m_p	The number of pickup requests in $t_{\max}$	$10,800$
ρ	The average speed of each courier	15 km/h
$d(r)$	The deadline for a pickup request r	30 minutes

TABLE 2: Parameters and Default Values

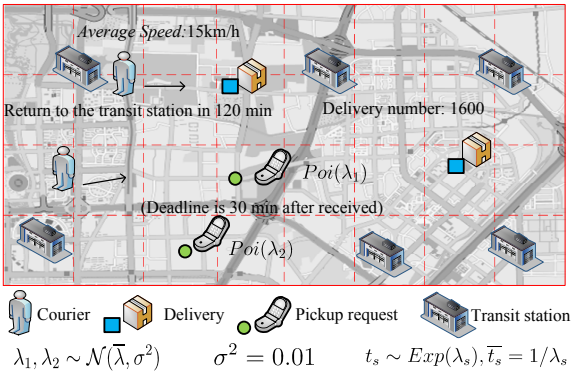


Fig. 5: Experimental Setup

(Request Generation): We assume that requests are generated on all nodes in our experiments. Similar to most queuing systems [15], we consider the arrival of a pickup request on a node as a Poisson process with intensity λ, which denotes the average number of pickup requests arriving per hour on the node. We sample the λ values for all nodes from a normal distribution $\mathcal{N}(\overline{\lambda}, \sigma^2)$, where the $\overline{\lambda}$ is the average request intensity on a node. We set $\overline{\lambda}$ to be 0.6 by default, and as a result, the average number of pickup requests in an hour in the area is $5,400$ by default. Such a setting is reasonable, since according to the statistics of the State Post Bureau of China (http://www.chinapost.gov.cn/), the total number of parcels including pickup and delivery in Beijing in the first quarter of 2015 is 283 million. Under the assumption that all parcels are uniformly distributed in a region of 50km × 50km every day from 6:00 a.m. to 11:59 p.m., the expected number of pickup requests per hour in our experiment region should be $5,185$, which is close to our default value. The deadline for a pickup request is set to be 30 minutes after the request is issued. Compared with the current city express company such as S.F. Express (http://www.sf-express.com/us/en/), which requires to reach the location of a pickup request in an hour after it is issued, our setting imposes a stricter requirement on the service quality.

(Initial Delivery Assignment): The set of delivery tasks should be generated at each transit station at the beginning. Thus, we use our request generation algorithm described above to generate the delivery in a period of 20 minutes and assign them to their corresponding transit stations. The delivery

tasks are randomly assigned to the couriers in each transit station and the initial schedule of each courier is calculated based on Algorithm 3 by processing all delivery tasks together. After setting off, each courier should return to his transit station within $t_{\max}$ time and finish all the assigned delivery in this period. We set $t_{\max}$ to be 2 hours. Since there is no strict temporal constraint on each delivery task, we can easily calculate the initial delivery routes at the beginning of the simulation.

(Service Process Simulation): We assume that the service time of each task follows an exponential distribution with mean $\bar{t}_s$, which is set to be 3 minutes by default. We update the locations of all couriers every 1 minute according to their schedules and average speed ρ. Once the schedule of a courier is changed, we change the route of the courier immediately.

Measurements. We test both the effectiveness and the efficiency of our proposed algorithms. All the experimental results are based on a simulation for 2 hours of the city express process using our simulation system. For effectiveness testing, we compare the satisfaction ratio SR of different algorithms, which is computed as:

$$\text{SR} = \frac{\text{\# accepted pickup requests}}{\text{\# issued pickup requests}}$$

We also compare the average incurred distance AID of different algorithms, which is computed as:

$$\text{AID} = \sum_{r \text{ is satisfied by } c_i} \Delta\text{dist}(r, S_i)/\text{\# accepted pickup requests}$$

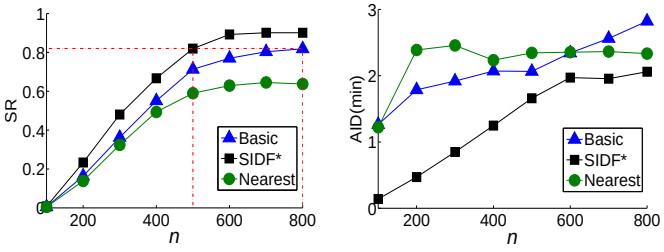
The average incurred distance AID is used to measure the average increased cost to serve an accepted pickup request.

For efficiency testing, we compare the average processing time per request, which is defined as the time spent on computing the schedules for all requests within the 2 hours divided by the number of issued requests. We also compare the average number of access nodes per request, which is the average number of node access operations involved to process a new request within the 2 hours.

Evaluation Algorithms. For effectiveness testing, we compare three algorithms, namely, Nearest, Basic and SIDF$*$. Nearest is a simple baseline that assigns each request r to its nearest courier c_i that can possibly serve r. If no such courier is found, the request is rejected. Basic (Algorithm 2) processes requests on a first-come, first-served basis, and SIDF$*$ (Algorithm 4) is the efficient version of SIDF (Algorithm 3) with the same effectiveness. For efficiency testing, we first compare Nearest, Basic, and SIDF$*$. Then we compare SIDF and SIDF$*$.

6.1 Effectiveness Testing

(Exp-1: Vary n). In this experiment, we vary the number of couriers n from 100 to 800. The experimental results for SR and AID are shown in Fig. 6 (a) and Fig. 6 (b) respectively. When n increases, SR and AID for all three algorithms Nearest, Basic, and SIDF$*$ increase. When n is small, SR for all three algorithms increase fast, and when n is large, SR for all algorithms increase slowly. The reason is that, when n is large, the potential for all three algorithms to satisfy more requests becomes small since most requests that can result in small incurred distance can still be satisfied by the algorithms with small n. Basic increases SR by 20% on average compared to Nearest, and SIDF$*$ increases SR by 10% on average compared to Basic. Remarkably, as illustrated by the dashed line in Fig. 6 (a), to satisfy the same number of requests (with SR $= 0.8$), Basic requires 800 couriers while SIDF$*$ only requires 500 couriers. In other words, our algorithm can potentially help the express company to save substantial cost (i.e., 37.5%) while achieving the same satisfaction ratio (i.e., SR $= 0.8$). Regarding AID, Basic has a smaller AID than Nearest when n is less than 600 and a larger AID when n is larger than 600. It shows that trying to find an optimal schedule immediately after a request streams in may not lead to a smaller average incurred distance. The AID of SIDF$*$ is the shortest among the three algorithms under all n values, which demonstrates the high effectiveness of our proposed batch algorithm SIDF$*$.

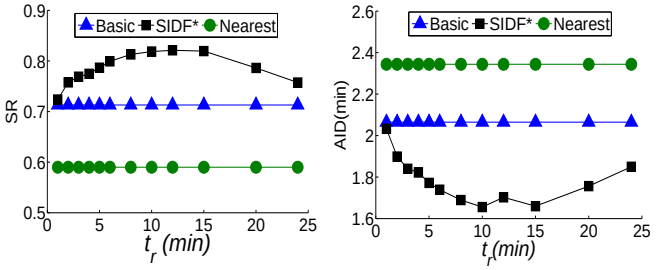


(a) SR (b) AID

Fig. 6: Vary n (Effectiveness)

(Exp-2: Vary t_r). We vary t_r, the maximum time to confirm a request, from 1 to 24 minutes and compare SR and AID for the three algorithms Nearest, Basic, and SIDF$*$. The experimental results for SR and AID are shown in Fig. 7 (a) and Fig. 7 (b) respectively. Nearest and Basic are independent of t_r. Thus, both SR and AID remain unchanged for Nearest and Basic when we vary t_r. For SR, SIDF$*$ is better than Nearest and Basic for all t_r values. When t_r increases from 1 to 12, SR for SIDF$*$ increases. This is because the batch assignment strategy in SIDF$*$ can have a large room for improvement when t_r is large. However, when t_r further increases, SR for SIDF$*$ starts to decrease. The reason is as follows. When t_r approaches $d(r)$ (30 minutes by default), the time left for couriers to satisfy early arrival requests becomes short. For example, if request r arrives at the beginning of the t_r period when t_r is 24 minutes, only 6 minutes are left for couriers to satisfy it. Therefore, the overall SR decreases when t_r increases in this case. Based on this result, a city express company can choose the best t_r using our simulation system given a certain parameter setting. For AID, SIDF$*$ has the shortest average incurred distance among the three algorithms under all t_r values. The AID of SIDF$*$ decreases when t_r increases from 1 to 10, but begins to increase when t_r is larger than 15. This is because when t_r is too large, SIDF$*$ has to reject some requests that have a small incurred distance with a strict deadline constraint (i.e., the requests issued at the beginning of t_r period).

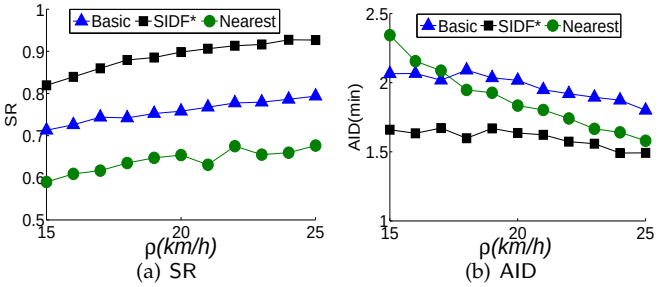
(Exp-3: Vary ρ). In this experiment, we vary the speed ρ of couriers from 15 to 25 (km/h). The trends of SR for the three algorithms Nearest, Basic, and SIDF$*$ are shown in Fig. 8 (a). When ρ increases, SR for all three algorithms increase. This is because the increasing of speed means the decreasing of cost to travel between any two nodes in the network,



(a) SR (b) AID

Fig. 7: Vary t_r in minutes (Effectiveness)

which allows more requests to be served. When ρ reaches 25, SR of Nearest, Basic and SIDF$*$ is 0.67, 0.75 and 0.92, respectively. The curves for AID when we vary ρ are shown in Fig. 8 (b). SIDF$*$ has the shortest AID under all speeds, while Basic has a smaller AID than Nearest only when the speed is lower than 17 km/h. When the speed is larger than 17 km/h, both Basic and Nearest can satisfy more requests with small incurred distance, but Basic can also satisfy those requests with larger incurred distance that are rejected by Nearest, which leads to the higher AID as shown in Fig. 8 (b). The results are consistent with our analysis that SIDF$*$ can choose a schedule with less incurred distance than Basic.



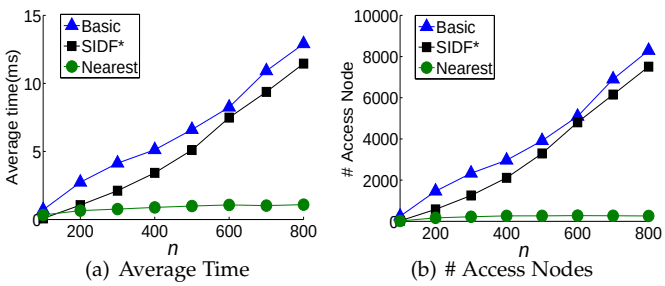
(a) SR (b) AID

Fig. 8: Vary ρ in km/h (Effectiveness)

6.2 Efficiency Testing

(Exp-4: Vary n). In this experiment, we vary the number of couriers n and test the average processing time and the number of access nodes per request by the three algorithms. The results are shown in Fig. 9 (a) and Fig. 9 (b) respectively. When the number of couriers n increases, the total processing time as well as the number of access nodes increase for all algorithms, because more couriers result in a larger search space. All the algorithms can process a request in less than 15ms and thus are suitable for realtime applications. Remarkably, SIDF$*$ has a smaller average processing time per request than Basic. This is because the average incurred distance per request by SIDF$*$ is less than Basic, which means finding a suitable schedule in SIDF$*$ needs less node access than Basic.

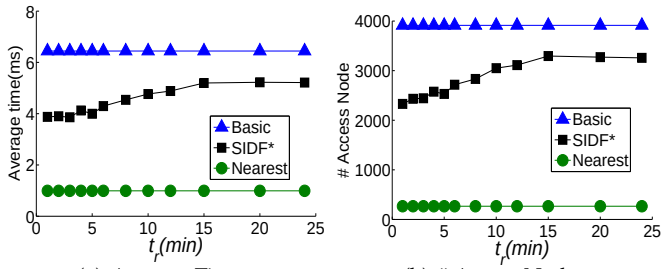
(Exp-5: Vary t_r). We vary t_r, the maximum time to confirm a request, from 1 to 24 minutes and test the efficiency of Nearest, Basic and SIDF$*$. The results for the average processing time and the number of access nodes are shown in Fig. 10 (a) and Fig. 10 (b) respectively. When t_r increases, the average time and the number of access nodes increase for SIDF$*$ because more search space incurs for processing more requests received in a longer t_r period. The efficiency of two streaming algorithms Basic and Nearest is not affected by the window size t_r. Nearest is the fastest among the three



(a) Average Time (b) # Access Nodes

Fig. 9: Vary n (Efficiency)

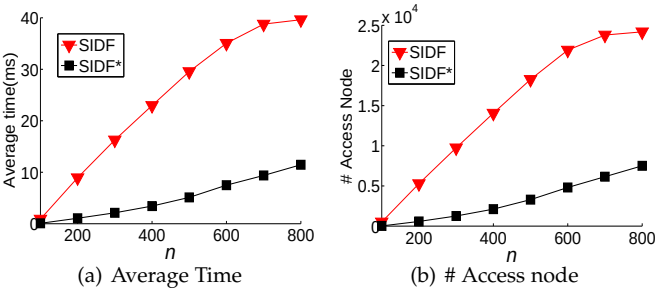
algorithms while Basic needs more processing time for a request than SIDF$*$. Remarkably, SIDF$*$ can process about 190 requests per second on average when $t_r = 15$ minutes and only 8 seconds are needed to process all requests received in 15 minutes.



(a) Average Time (b) # Access Nodes

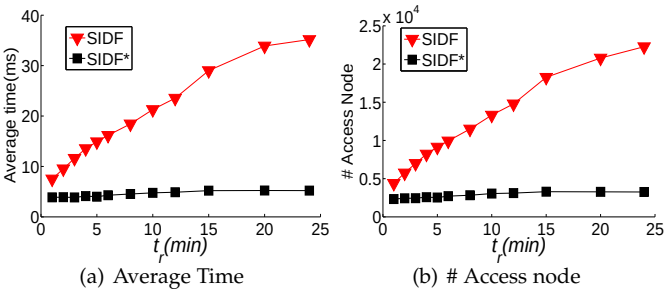
Fig. 10: Vary t_r in minutes (Efficiency)

(Exp-6: Comparing SIDF and SIDF$*$). We compare the average processing time per request by SIDF and SIDF$*$ to show the efficiency improvement by the two-level priority queue. The results for varying n and t_r are shown in Fig. 11 and Fig. 12 respectively. SIDF$*$ is six times faster than SIDF on average, which demonstrates the advantage of the global priority queue in reducing exact incurred distance calculation, as well as the advantage of the local priority queues in maintaining the information of candidate entries. The results for the number of access nodes are similar to the average processing time.



(a) Average Time (b) # Access node

Fig. 11: Efficiency improvement by SIDF$*$ (vary n)



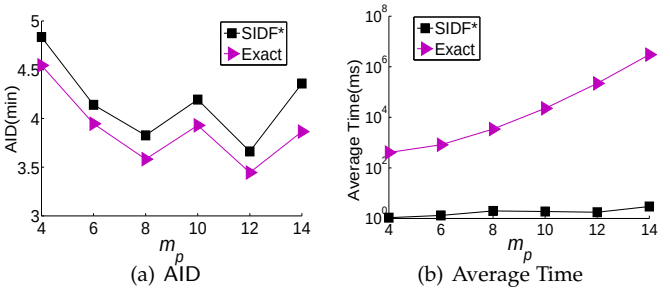
(a) Average Time (b) # Access node

Fig. 12: Efficiency improvement by SIDF$*$ (vary t_r)

6.3 Performance Comparison with the Exact Algorithm

Since SIDF∗ is an approximate algorithm, it is interesting to compare its performance with an exact algorithm which performs exhaustive search to find the optimal schedule to serve the pickup requests. Given a set of pickup requests and a set of n couriers, the exact algorithm first uses a brute force search strategy to explore all possible ways of request assignments to the n couriers (without considering where to insert them into the schedule). For every assignment scheme, the exact algorithm uses dynamic programming to obtain the optimal schedule of each courier with the minimum travel cost. We assume that the couriers can satisfy all the requests and compare the AID and average processing time for SIDF∗ and the exact algorithm Exact.

(Exp-7: Varying the number of request m_p when the courier number n=2). We fix the number of couriers $n = 2$. The results are shown in Fig. 13 (a) and Fig. 13 (b) respectively. It is obvious that SIDF∗ has very close AID to the exact algorithm while running at least two orders of magnitude faster than the exact algorithm when the number of request varies from 4 to 14. When the number of request is larger than 14, the exact algorithm can not finish in 24 hours, thus we omit the results when m_p is larger than 14. This results show that: (1) the exact algorithm is computationally infeasible due to its exponential cost, and (2) our method SIDF∗ can compute the request assignments with very close AID to that of the exact algorithm, and is orders of magnitude faster.

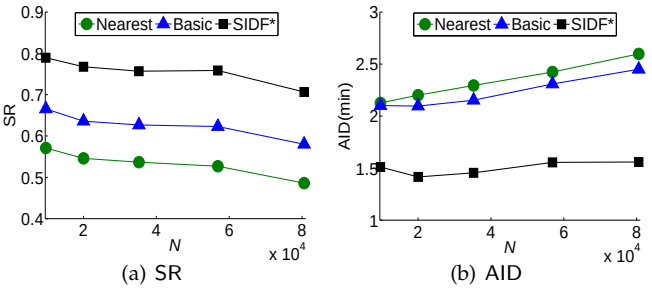


(a) AID (b) Average Time

Fig. 13: Varying m_p (Effectiveness and Efficiency)

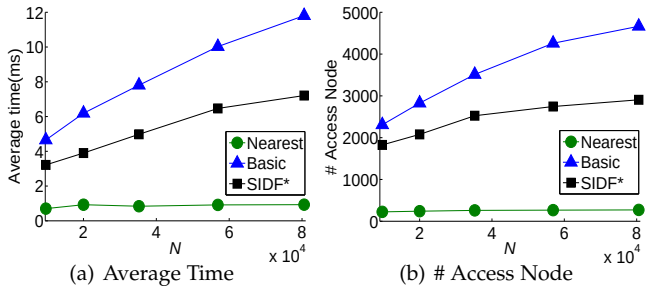
6.4 Scalability Test

(Exp-8: Varying the number of nodes N in the road networks). To evaluate the scalability of SIDF∗, we compare the performance of the three algorithms, SIDF∗, Basic and Nearest on road networks of increasing size. Specifically, we run the three algorithms on road networks corresponding to the second ring, third ring, fourth ring, fifth ring and sixth ring of Beijing. The largest road network consists of 81,000 nodes and 104,000 edges, and covers the main urban district of Beijing. The number of couriers, transit stations and delivery tasks are increased in proportion to the node number of the road network. For instance, the largest road network contains 5,000 couriers, 70 transit stations and 16,000 delivery tasks. Other parameters are the same as the default value. The results for SR, AID, the average processing time and the number of access nodes are shown in Fig. 14 and Fig. 15 respectively. The results show that when the number of nodes N increases from 8,400 to 81,000, SIDF∗ has the highest SR and the smallest AID among the three algorithms. Besides, it uses less average processing time than the streaming algorithm Basic according to Fig. 15. It is noted that the SR of the three algorithms decrease when the road network contains 81,000 nodes. This is because the density of the nodes is different in different regions of Beijing, and it takes more time to travel between nodes outside the fifth ring region of Beijing. Overall, our method SIDF∗ scales well with the road network size in terms of both service quality and computational efficiency.



(a) SR (b) AID

Fig. 14: Varying N (Effectiveness)



(a) Average Time (b) # Access Node

Fig. 15: Varying N (Efficiency)

7 Related Works

Query Processing on Spatial Networks. A lot of work has been done on spatial network query processing. The most related works include shortest path computation, kNN search, and best point detour on a road network.

Shortest path and distance queries on road networks have been experimentally evaluated by Wu et al. [16]. In [16], it is shown that the vertex importance based approach CH [17] is the most space-economic technique and spatial coherence based method SILC [18] is the most efficient method. Inspired by CH, Zhu et al. [19] proposed the Arterial Hierarchy (AH) method that can outperform CH in terms of both asymptotic and practical performance.

The kNN query in a spatial network was first studied by Papadias et al. [20]. In [20], two classical methods were proposed: INE (Incremental Network Expansion) and IER (Incremental Euclidean Restriction). The LBC method proposed by Deng et al. [21] improved the INE method by maintaining a path distance lower bound for each candidate data point and searching toward the candidates with minimum path distance lower bound. Nutanong et al. [22] further improved LBC by devising a novel heuristic function that considered all the candidates for a query point as a single unit.

The best point detour problem is proposed by Shang et al. [23], which aims to find a detour that can pass a certain type of node with the minimum additional traveling cost. The best point detour problem is different from our problem studied in this paper since we aim at optimizing the route for multiple requests and multiple couriers in a dynamic environment.

Dynamic Vehicle Routing. Our problem is a restricted variant of the dynamic vehicle routing problem [2], [24], [25], [26], [27], where each request that includes both pickup and delivery location constraints are satisfied by a set of vehicles. The dynamic vehicle routing problem has been studied extensively in the Operational Research literature. Our work is different from existing studies in three aspects:

(1) To the best of our knowledge, there is no existing work that studies the dynamic vehicle routing problem on a real road network, where it is costly to compute the shortest distance between two nodes, and is impractical to precompute and store all-pair shortest distances in the large road network. Our solution exploits a spatio-temporal index to estimate the travel cost lower bound and reduces the number of exact shortest path computations. Therefore, we are able to handle real-world pick-up requests workloads (e.g., 5400 requests per hour with 800 couriers in a road network of Beijing). In contrast, the largest simulation instance on dynamic vehicle routing problem from the OR literature is 60 requests per hour with 100 couriers in Euclidean space [28].
(2) Most works on dynamic vehicle routing problem assume that all the requests can be satisfied by the fixed numbers of couriers and their objective is to minimize the operational cost [4], [29], [30], [31], [32], [33], [34], [35] (e.g., the sum of the total travel cost for couriers and the delayed cost for all requests). Our objective is to satisfy incoming pickup requests as many as possible under the temporal constraints on pickup requests and couriers (i.e., each request must be satisfied on time once assigned to a courier and couriers must come back to their transit stations on time). This setting is more realistic, because in practice, a city express company may not be able to satisfy all the pickup requests arriving in real-time due to limited manpower.
(3) Compared with existing works [5], [28] that reject requests immediately due to hard time constraints, we adopt a batch solution that considers the scheduling problem for requests received in a short period instead of making a decision immediately after a new request arrives. As we demonstrate in our experiments, such a strategy can satisfy more requests than a basic streaming algorithm as we gather short term requests information for further optimization.

Dynamic Taxi Ridesharing. Recently, dynamic taxi ridesharing has been studied in the database community. A taxi ridesharing query asks if there exists a taxi in a city that can take the customer from his/her current location to a specific destination under time window constraints. Our dynamic city express problem is different from the taxi ridesharing problem in three aspects. *First*, though the number of taxis is larger than the couriers in our problems, the capacity of a courier is much larger than a taxi so that finding the optimal route of a courier using kinetic tree structure [9] proposed for taxi ridesharing is impractical. *Second*, the deadline of processing a pickup request (e.g., 30 minutes after issuing) is larger than the maximal waiting time of a customer (e.g., 5 minutes) in ridesharing service. Moreover, there is a tight time window for a taxi to reach customer's destination but a parcel can usually wait a longer time at the transit station before being delivered to its destination. Thus the spatial-temporal index proposed in [7], [8] is less effective in solving city express problem resulting in more candidate couriers for a pickup request. *Third*, there is response time for a request in city express service while a taxi ridesharing request needs instant response. Once a customer issues a pickup request on the website or through a smartphone, a staff from the express company will contact the customer to confirm or decline the request within a short period (e.g., 10 minutes). Such a period allows the system to gather multiple requests for better scheduling. On the other hand, we propose a new algorithm that can accelerate the request assignment on a two-level priority queue structure, which can benefit the research in urban computing [1] such as the ridesharing system.

8 Conclusion

In this paper, we propose a solution to batch process requests in dynamic city express. We also develop a simulation platform to confirm both the effectiveness and efficiency of our solution. The new city express system has three advantages: *First*, when the intensity of pickup requests is $72/km^2$ per hour (i.e., 5,400/75km^2), our solution can increase the satisfaction ratio of pickup requests by more than 10% compared to a basic solution and 30% compared to an existing straightforward solution. On the other hand, our solution can save 37.5% manpower to achieve 80% satisfaction ratio compared to the basic solution as shown in the experiment. *Second*, our solution enjoys high efficiency with the help of the two-level priority queue structure. On average, we can process about 190 pickup requests per second under default settings, which is 15% faster than the basic streaming algorithm. *Third*, the developed simulation platform can be used to estimate the number of couriers a city express company needs to achieve a certain satisfaction ratio in urban area. For instance, by estimating the satisfaction ratio under different courier numbers in the simulation, we can find that at least 7 couriers/km^2 (i.e., 500 couriers/75 km^2) are needed to keep the satisfaction ratio above 80%, or equivalently, serve 8,640 pickup requests in 2 hours. In the future, we plan to consider more factors that can affect the service quality of city express service such as the time-dependent travel cost of road segments and the stochastic information of requests' locations.

Acknowledgments

This work is supported by a Microsoft Research grant on urban informatics, the National Natural Science Foundation of China (Grant No. 61672399), CUHK Direct Grant No. 4055015 and 4055048. Lu Qin is supported by ARC DE140100999 and ARC DP160101513.

References

[1] Y. Zheng, L. Capra, O. Wolfson, and H. Yang, "Urban computing: concepts, methodologies, and applications," *ACM Transactions on Intelligent Systems and Technology*, vol. 5, no. 3, pp. 38–55, 2014.
[2] V. Pillac, M. Gendreau, C. Guéret, and A. L. Medaglia, "A review of dynamic vehicle routing problems," *European Journal of Operational Research*, vol. 225, no. 1, 2013.
[3] O. Bräysy and M. Gendreau, "Vehicle routing problem with time windows, part i: Route construction and local search algorithms," *Transportation science*, vol. 39, no. 1, 2005.

[4] Z.-L. Chen and H. Xu, "Dynamic column generation for dynamic vehicle routing with time windows," *Transportation Science*, vol. 40, no. 1, 2006.

[5] M. Gendreau, F. Guertin, J.-Y. Potvin, and E. Taillard, "Parallel tabu search for real-time vehicle routing and dispatching," *Transportation science*, vol. 33, no. 4, 1999.

[6] M. M. Solomon, "Algorithms for the vehicle routing and scheduling problems with time window constraints," *Operations research*, vol. 35, no. 2, 1987.

[7] S. Ma, Y. Zheng, and O. Wolfson, "T-share: A large-scale dynamic taxi ridesharing service," in *Proc. of ICDE'13*, 2013.

[8] ——, "Real-time city-scale taxi ridesharing," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1782–1795, 2015.

[9] Y. Huang, F. Bastani, R. Jin, and X. S. Wang, "Large scale real-time ridesharing with service guarantee on road networks," *PVLDB*, vol. 7, no. 14, 2014.

[10] M. M. Flood, "The traveling-salesman problem," *Operations Research*, vol. 4, no. 1, 1956.

[11] M. Jsnger, S. Thienel, and G. Reinelt, "Provably good solutions for the traveling salesman problem," *Zeitschrift Operations Research*, vol. 40, no. 2, 1994.

[12] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II, "An analysis of several heuristics for the traveling salesman problem," *SIAM journal on computing*, vol. 6, no. 3, 1977.

[13] M. R. Kolahdouzan and C. Shahabi, "Voronoi-based K nearest neighbor search for spatial network databases," in *Proc. of VLDB'04*, 2004.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[15] S. M. Ross *et al.*, *Stochastic processes*. John Wiley & Sons New York, 1996, vol. 2.

[16] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou, "Shortest path and distance queries on road networks: An experimental evaluation," *PVLDB*, vol. 5, no. 5, 2012.

[17] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Proc. of WEA'08*, 2008.

[18] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases," in *Proc. of SIGMOD'08*, 2008.

[19] 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. of SIGMOD'13*, 2013.

[20] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *Proc. of VLDB'03*, 2003.

[21] K. Deng, X. Zhou, H. T. Shen, S. W. Sadiq, and X. Li, "Instance optimal query processing in spatial networks," *VLDB J.*, vol. 18, no. 3, 2009.

[22] S. Nutanong and H. Samet, "Memory-efficient algorithms for spatial network queries," in *Proc. of ICDE'13*, 2013.

[23] S. Shang, K. Deng, and K. Xie, "Best point detour query in road networks," in *Proc. of GIS'10*, 2010.

[24] G. Laporte, "Fifty years of vehicle routing," *Transportation Science*, vol. 43, no. 4, pp. 408–416, 2009.

[25] J.-F. Cordeau and G. Laporte, "The dial-a-ride problem: models and algorithms," *Annals of Operations Research*, vol. 153, no. 1, pp. 29–46, 2007.

[26] D. Sáez, C. E. Cortés, and A. Núñez, "Hybrid adaptive predictive control for the multi-vehicle dynamic pick-up and delivery problem based on genetic algorithms and fuzzy clustering," *Computers & Operations Research*, vol. 35, no. 11, pp. 3412–3438, 2008.

[27] C. E. Cortés, A. Núnez, and D. Sáez, "Hybrid adaptive predictive control for a dynamic pickup and delivery problem including traffic congestion," *International Journal of Adaptive Control and Signal Processing*, vol. 22, no. 2, pp. 103–123, 2008.

[28] A. Fabri and P. Recht, "On dynamic pickup and delivery vehicle routing with several time windows and waiting times," *Transportation Research Part B: Methodological*, vol. 40, no. 4, pp. 335–350, 2006.

[29] M. Savelsbergh and M. Sol, "Drive: Dynamic routing of independent vehicles," *Operations Research*, vol. 46, no. 4, pp. 474–490, 1998.

[30] O. B. Madsen, H. F. Ravn, and J. M. Rygaard, "A heuristic algorithm for a dial-a-ride problem with time windows, multiple capacities, and multiple objectives," *Annals of operations Research*, vol. 60, no. 1, pp. 193–208, 1995.

[31] D. Teodorovic and G. Radivojevic, "A fuzzy logic approach to dynamic dial-a-ride problem," *Fuzzy sets and systems*, vol. 116, no. 1, pp. 23–33, 2000.

[32] M. Gendreau, F. Guertin, J.-Y. Potvin, and R. Séguin, "Neighborhood search heuristics for a dynamic vehicle dispatching problem with pick-ups and deliveries," *Transportation Research Part C: Emerging Technologies*, vol. 14, no. 3, pp. 157–174, 2006.

[33] S. Mitrović-Minić, R. Krishnamurti, and G. Laporte, "Double-horizon based heuristics for the dynamic pickup and delivery problem with time windows," *Transportation Research Part B: Methodological*, vol. 38, no. 8, pp. 669–685, 2004.

[34] S. Mitrović-Minić and G. Laporte, "Waiting strategies for the dynamic pickup and delivery problem with time windows," *Transportation Research Part B: Methodological*, vol. 38, no. 7, pp. 635–655, 2004.

[35] L. Coslovich, R. Pesenti, and W. Ukovich, "A two-phase insertion technique of unexpected customers for a dynamic dial-a-ride problem," *European Journal of Operational Research*, vol. 175, no. 3, pp. 1605–1615, 2006.

Siyuan Zhang Siyuan Zhang is a Ph.D. student in the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. His major research interests include spatial database and geographic information systems.

Lu Qin Lu Qin is now a senior lecturer in the Centre of Quantum Computation and Intelligent Systems (QCIS) in University of Technology Sydney (UTS). He received his bachelor degree from Renmin University of China (RUC) in 2006 and his Ph.D. degree from department of Systems Engineering and Engineering Management in the Chinese University of Hong Kong (CUHK) in 2010.

Yu Zheng Dr. Yu Zheng is a research manager from Microsoft Research, passionate about using big data to tackle urban challenges. He currently serves as the Editor-in-Chief of ACM Transactions on Intelligent Systems and Technology. He is also the founding Secretary of SIGKDD China Chapter and has served as chair on over 10 prestigious international conferences, e.g. as the program co-chair of ICDE 2014 (Industrial Track). Zheng received five best paper awards from ICDE13 and ACM SIGSPATIAL10, etc. His book, titled Computing with Spatial Trajectories, has been used as a text book in universities world-widely and awarded the Top 10 Most Popular Computer Science Book authored by Chinese at Springer. In 2013, he was named one of the Top Innovators under 35 by MIT Technology Review (TR35) and featured by Time Magazine for his research on urban computing. In 2014, he was named one of the Top 40 Business Elites under 40 in China by Fortune Magazine, because of the business impact of urban computing he has been advocating since 2008. Zheng is also a visiting Chair Professor at Shanghai Jiao Tong University and an Adjunct Professor at Hong Kong University of Science and Technology.

Hong Cheng Hong Cheng is an Associate Professor in the Department of Systems Engineering and Engineering Management at the Chinese University of Hong Kong. She received her Ph.D. degree from University of Illinois at Urbana-Champaign in 2008. Her research interests include data mining, database systems, and machine learning. She received research paper awards at ICDE'07, SIGKDD'06 and SIGKDD'05, and the certificate of recognition for the 2009 SIGKDD Doctoral Dissertation Award. She is a recipient of the 2010 Vice-Chancellor's Exemplary Teaching Award at the Chinese University of Hong Kong.