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 (e.g., $r_3$) 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 \( p = (v_0, v_1, \ldots, v_{k-1}, v_k) \) 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 \( p = (v_0, v_1, \ldots, v_k) \), the cost of the path, denoted as \( \text{cost}(p) \), is calculated as \( \text{cost}(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \). Given two nodes \( u \) and \( v \) in \( G \), the shortest path from \( u \) to \( v \) is the path \( p_{\text{min}} \) with the minimum cost among all paths from \( u \) to \( v \) in \( G \). We denote the corresponding cost of \( p_{\text{min}} \) as \( \text{cost}(u, v) \).

**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 $t_c$ 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 $t_{\text{max}}$ 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 $\text{cost}(S_i)$, is calculated as:

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

where $\text{cost}(l(c_i), l(r_{i,1}))$ is the time spent from the current location of courier $c_i$ to the location of the first task in the schedule $S_i$; $\sum_{1 \leq j < m_i} \text{cost}(l(r_{i,j}), l(r_{i,j+1}))$ is the 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 $\text{cost}(l(r_{i,m_i}), ts(c_i))$ 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 $\text{cost}(S_i)$ can be simplified as follows:

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

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 $d(r_{i,m_i+1}) = ts(c_i) + t_{\text{max}}$, where $ts(c_i)$ 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.

### TABLE 1: List of Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w(u, v)$</td>
<td>The weight of edge $(u, v)$</td>
</tr>
<tr>
<td>$\text{cost}(u, v)$</td>
<td>The shortest traveling time from $u$ to $v$</td>
</tr>
<tr>
<td>$r$</td>
<td>A request for city express</td>
</tr>
<tr>
<td>$l(r)$</td>
<td>The location of the request $r$</td>
</tr>
<tr>
<td>$d(r)$</td>
<td>The deadline to serve the request $r$</td>
</tr>
<tr>
<td>$C = {c_1, c_2, \cdots, c_n}$</td>
<td>The set of couriers</td>
</tr>
<tr>
<td>$S_i = {r_{i,1}, \cdots, r_{i,m_i}}$</td>
<td>The schedule for courier $c_i$</td>
</tr>
<tr>
<td>$\text{cost}(S_i)$</td>
<td>The cost of the schedule $S_i$</td>
</tr>
<tr>
<td>$l(c_i)$</td>
<td>The current location of courier $c_i$</td>
</tr>
<tr>
<td>$l(r_{i,0})$, $l(r_{i,m_i+1})$</td>
<td>The location of the transit station of $c_i$</td>
</tr>
<tr>
<td>$t_{\text{set}}(c_i)$</td>
<td>The set-off time for courier $c_i$</td>
</tr>
<tr>
<td>$t_s$</td>
<td>Service time for a request</td>
</tr>
<tr>
<td>$t_{\text{max}}$</td>
<td>The maximum time spent for a trip</td>
</tr>
<tr>
<td>$l(r_{i,j})$, $d(r_{i,j})$</td>
<td>The latest time for courier $c_i$ to arrive $l(r_{i,j})$ in $S_i$</td>
</tr>
<tr>
<td>$\text{cand}(r)$</td>
<td>The candidate couriers to serve $r$</td>
</tr>
<tr>
<td>$\text{cand}(c_i)$</td>
<td>The candidate requests that $c_i$ can serve</td>
</tr>
<tr>
<td>$\text{cost}(u, v)$</td>
<td>A lower bound of cost $(u, v)$</td>
</tr>
<tr>
<td>$\text{dist}(r, S_i)$</td>
<td>The incurred distance of $r$ to segment $j$ of $S_i$</td>
</tr>
<tr>
<td>$\text{dist}(r, S_i)$</td>
<td>The incurred distance of $r$ to $S_i$</td>
</tr>
</tbody>
</table>

### 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 $t_{\text{max}}$ 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 \( \{r_1, r_2, \ldots, r_m\} \) 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 \( t_s \). 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 \( t_s(l) \) to denote the generator for the region that \( l \) lies in. For each node \( v \in V \), we precompute \( \text{cost}(v, t_s(v)) \) and \( \text{cost}(t_s(v), v) \). For any two generators \( t_s \) and \( t_s \), we precompute \( \text{cost}(t_s, t_s) \). We also precompute the radius of each generator \( t_s \), denoted as \( \text{radius}(t_s) \), which is the maximum cost from \( t_s \) to any node in the Voronoi region of \( t_s \). With the NVD index, given any location \( l \) in \( G \), suppose \( l \) lies on the edge \((u, v)\), the distance from \( l \) to \( t_s(l) \) can be calculated as:

\[
\text{cost}(l, t_s(l)) = \min\{w(l, u) + \text{cost}(u, t_s(u)), w(l, v) + \text{cost}(v, t_s(v))\}.
\]

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

\[
\text{cost}(l_i, l_j) = \max\{0, \text{cost}(l_i, l_j) - \text{cost}(t_s(l_i), l_i) - \text{cost}(l_j, t_s(l_j))\}.
\]

Obviously, with the NVD index, for any two locations \( l_i \) and \( l_j \) in \( G \), \( \text{cost}(l_i, l_j) \) 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.

![Fig. 2: The NVD Index](image)

**Example 3.1:** Fig. 2 (a) shows the NVD index for a road network \( G \) with three transit stations \( t_{s0}, t_{s1} \) and \( t_{s2} \) 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 \( \text{cost}(v_1, v_2) \) can be calculated as

\[
\text{cost}(v_1, v_2) = \max\{0, d_{02} - \text{cost}(t_{s0}, v_1) - \text{cost}(v_2, t_{s2})\}.
\]

\[\square\]

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 \( \text{cand}(r) \) for a request \( r \) as follows.

\[
\text{cand}(r) = \{c_i | \text{cost}(l(r_{i,0}), l(r)) \leq d(r) - t_{\text{cur}}\}.
\]

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

\[
\text{candts}(r) = \{t_{s_j} | \text{cost}(l(t_{s_j}), l(r)) - \text{radius}(t_{s_j}) \leq d(r) - t_{\text{cur}}\}.
\]

After computing \( \text{candts}(r) \), for each \( t_{s_j} \in \text{candts}(r) \), we enumerate all couriers \( c_i \) that lie in the Voronoi region of \( t_{s_j} \) and add \( c_i \) into \( \text{cand}(r) \) if \( \text{cost}(l(r_{i,0}), l(r)) \leq d(r) - t_{\text{cur}} \).
3.3 Incurred Distance Calculation

For each courier $c_i \in \text{cand}(r)$, we insert request $r$ into the schedule $S_i$ of $c_i$ and calculate the incurred distance of valid 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:

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

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:

$$a_j(S_i) + 2 \times t_s + \text{cost}(l(r_{i,j}), l(r)) + \text{cost}(l(r), l(r_{i,j+1})) \leq \overline{d}_{j+1}(S_i) \quad (\text{6})$$

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:

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

(\text{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 $\Delta \text{dist}(r, S_i)$ when inserting $r$ into the valid segment(s) of $S_i$. $\Delta \text{dist}(r, S_i)$ is defined as follows:

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

(\text{8})

where $\Delta \text{dist}_j(r, S_i)$ 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:

$$\Delta \text{dist}_j(r, S_i) = \text{cost}(l(r_{i,j}), l(r)) + \text{cost}(l(r), l(r_{i,j+1})) - \text{cost}(l(r_{i,j}), l(r_{i,j+1})).$$

(\text{9})

For each candidate courier in 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 $\Delta \text{dist}(r, S_i)$ 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 $\Delta \text{dist}_j(r, S_i)$ involves computing $\text{cost}(l(r_{i,j}), l(r))$ and $\text{cost}(l(r), l(r_{i,j+1}))$ 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 $\Delta \text{dist}_j(r, S_i)$ for any segment $j$ as:

$$\Delta \text{dist}_j(r, S_i) = \max\{0, \text{cost}(l(r_{i,j}), l(r)) + \text{cost}(l(r), l(r_{i,j+1})) - \text{cost}(l(r_{i,j}), l(r_{i,j+1}))\},$$

where $\text{cost}(l, l')$ for any two locations $l$ and $l'$ can be calculated by Eq. 3 using the NVD index. The rationale is that if the $\Delta \text{dist}$ of a segment is large, we can prune it without computing its $\Delta \text{dist}$. 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 $(c_i, r_{i,j}, \Delta \text{dist}, \text{flag})$ in $\mathcal{H}$ denotes segment $j$ in $S_i$ with incurred distance $\Delta \text{dist}$. flag is either true or false, denoting whether $\Delta \text{dist}$ is the exact $\Delta \text{dist}(r, S_i)$ or the lower bound $\Delta \text{dist}_j(r, S_i)$ respectively. $\mathcal{H}$ maintains the segment with the minimum $\Delta \text{dist}$, and is initialized to be $\emptyset$ (line 1). For each candidate $c_i$ in cand($r$), we insert into $\mathcal{H}$ every segment $j$ that satisfies conditions 1 (Eq. 5) and 2 (Eq. 6) by replacing cost with $\text{cost}$ (line 2-6). The incurred distance of each segment $j$ in $\mathcal{H}$ is the lower bound $\Delta \text{dist}_j(r, S_i)$. At this stage, no exact distance has been computed. Next, we iteratively pop out the entry $(c_i, r_{i,j}, \Delta \text{dist}, \text{flag})$ in $\mathcal{H}$ with the minimum $\Delta \text{dist}$. If $\Delta \text{dist}$ is the lower bound $\Delta \text{dist}_j(r, S_i)$ (flag is false), we compute the exact incurred distance $\Delta \text{dist}_j(r, S_i)$ using $\text{cost}(l(r_{i,j}), l(r))$ and $\text{cost}(l(r), l(r_{i,j+1}))$ (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 $\Delta \text{dist}_j(r, S_i)$ 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 cand($r$))

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

The overall algorithm is outlined in Algorithm 2. Suppose for each courier $c_{i}$, $a_{j}(S_{i})$ and $d_{j}(S_{i})$ for $0 \leq j \leq m_{i} + 1$ 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), $a_{j}(S_{i})$ for all $j + 1 \leq k \leq m_{i} + 1$ need to be updated in increasing order of $k$ (line 14-15), and $d_{k}(S_{i})$ for all $1 \leq k \leq j + 1$ 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 $c_{i} \in C$ do
2: \hspace{1em} compute $a_{j}(S_{i})$ and $d_{j}(S_{i})$ for $0 \leq j \leq m_{i} + 1$;
3: \hspace{1em} while there is a new request $r$ issued in real time do
4: \hspace{2em} load updated locations of all couriers;
5: \hspace{2em} Update($C$, $r$);
6: \hspace{2em} Procedure Update(courier set $C$, request $r$)
7: \hspace{3em} compute cand($r$);
8: \hspace{3em} if ($c_{i}$, $j$) = BestR($r$, cand($r$)) then
9: \hspace{4em} if $i = -1$ then
10: \hspace{5em} return;
11: \hspace{5em} Insert($S_{i}$, $j$, $r$);
12: Procedure Insert(schedule $S_{i}$, segment $j$, request $r$)
13: \hspace{1em} return $r$ between $r_{i,j}$ and $r_{i,j+1}$ of $S_{i}$;
14: \hspace{1em} for $k = j + 1$ to $m_{i} + 1$ do
15: \hspace{2em} $a_{k}(S_{i}) \leftarrow a_{k-1}(S_{i}) + t_{x} + \text{cost}(l(r_{k-1}, l(r_{k})));
16: \hspace{2em} for $k = j + 1$ down to 1 do
17: \hspace{3em} $d_{k}(S_{i}) \leftarrow \min(d(r_{i,k}), d_{k+1}(S_{i}) - a_{k+1}(S_{i}) + a_{k}(S_{i}))$

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 $r_{0} \rightarrow r_{1} \rightarrow r_{4} \rightarrow r_{5}$. The next step is to check $r_{3}$ and $r_{6}$. But neither of them can be served by courier $c$ due to the large incurred distance of $r_{3}$ 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 $r_{0} \rightarrow r_{1} \rightarrow r_{6} \rightarrow r_{2}$. 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 $d_{j}(S_{i})$ 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 $H$ to maintain the priorities of all requests. Each entry in $H$ 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 $r_{i,j} \rightarrow r_{i,j+1}$ 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 $H$ (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 \( H^g \) (line 18-19), we pop out the request \( r \) with the highest priority (minimum \( \Delta \text{dist} \)) from \( H^g \) 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 \( r' \in \text{cand}(c_i) \) needs to be updated in \( H^g \) (line 24-28). This is processed by removing \( r' \) from \( H^g \) (line 25), calculating the updated priority (line 26), and inserting it again into \( H^g \) if it is still valid (line 27-28). Here, after inserting \( r \) into segment \( j \) of schedule \( S_i \), for each request \( r' \in \text{cand}(c_i) \), 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 \( r' \in \text{cand}(c_i) \) 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 \( r' \in \text{cand}(c_i) \) may increase due to the insertion of two new segments.

3. **Segment updating.** The values of \( a^k_j(S_i) \) (for \( j + 1 \leq k \leq m_i + 1 \)) and \( d^k_j(S_i) \) (for \( 1 \leq k \leq j + 1 \)) are updated after the insertion of \( r \). Therefore, the priority of a request \( r' \in \text{cand}(c_i) \) needs to be updated accordingly because some segments become invalid w.r.t. \( r' \).

The algorithm terminates when all possible requests in \( H^g \) 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 \( c_i \in C \) do
2. compute \( a_{ij}(S_i) \) and \( d_{ij}(S_i) \) for \( 0 \leq i \leq m_i + 1 \);
3. for every \( t_i \) minutes do
4. load updated locations of all couriers;
5. \( R \leftarrow \) the set of all requests issued in the last \( t_i \) minutes;
6. ComputeCand\((C, R)\);
7. UpdateR\((C, R)\);
8. Procedure ComputeCand\((c_i, c_i, R)\);
9. cand\((c_i) \leftarrow H^g \) if \( c_i \in C \);
10. Procedure UpdateR\((c_i, C, R)\);
11. procedures for all requests in \( R \) do
12. for all \( c_i \in \text{cand}(r) \) do
13. cand\((c_i) \leftarrow \text{cand}(c_i) \cup \{r\} \);
14. UpdateR\((c_i, C, R)\);
15. for all \( r \in R \) do
16. for all \( (c_i, j) \in \text{Best}(r, \text{cand}(r)) \) do
17. if \( i \neq -1 \) then
18. if \( \Delta \text{dist}(r, S_i) \) for all \( j \in S_i \) do
19. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
20. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
21. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
22. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
23. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
24. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
25. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
26. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
27. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
28. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do
29. for all \( r' \in \text{cand}(c_i) \) and \( r' \in H^g \) do

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 BestR\((r', \text{cand}(c_i)) \) to compute a new assignment for every request \( r' \in \text{cand}(c_i) \) w.r.t. the schedule of every candidate courier \( c' \in \text{cand}(r') \). 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*. First, we observe that once a request \( r \) is assigned to the schedule \( S_i \) of a courier \( c_i \), only those requests in \( \text{cand}(c_i) \) 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 \( H_i \) that keeps all the candidate requests \( \text{cand}(c_i) \); and at the top level, we maintain a global priority queue \( H_g \) 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 \( H_i \) 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 \( H_i \) that keeps all the candidate requests \( \text{cand}(c_i) \). Each entry in the local priority queue \( H_i \) has the form:

\[
(r, c_i, r_{i,j}, r_{i,j}+1, \Delta \text{dist}, \text{flag})
\]

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 \( H^g \) of SIDF, a flag is added in each entry indicating the incurred distance \( \Delta \text{dist} \) is an exact value (if \( \text{flag} = \text{true} \)) or a lower bound (if \( \text{flag} = \text{false} \)).

The operations on the local priority queue include the following:

1. **Initialization.** In the initialization step, for each request \( r \in \text{cand}(c_i) \), each and valid segment \( j \) in \( S_i \), we insert an entry \( (r, c_i, r_{i,j}, r_{i,j}+1, \Delta \text{dist}, (r, S_i), \text{false}) \) to \( H_i \). 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 \( H_i \) and push it into the global priority queue \( H_g \). 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 \( H_i \).
(1) For a request \( r \) and a valid segment \( j \) in \( S_i \), once we calculate the exact incurred distance \( \Delta \text{dist}_i(r, S_i) \), we push the entry \( (r, c_i, r_{i,j}, r_{i,j+1}, \Delta \text{dist}_i(r, S_i), \text{true}) \) 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 \( (r, c_i, r_{i,j}, r_{i,j+1}, \Delta \text{dist}, \text{flag}) \) 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 \( \text{flag} = \text{true} \), we will assign the request into the concerned segment \( j \) and update courier \( c_i \)’s schedule \( S_i \).

   (2) If \( \text{flag} = \text{false} \), we will calculate the exact incurred distance \( \Delta \text{dist}_i(r, S_i) \) 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 \( \text{InitH}(c_i, R) \) (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 \( \Delta \text{dist} \).

**Step 2.** We pop the top element \( (r, c_i, r_{i,j}, r_{i,j+1}, \Delta \text{dist}, \text{flag}) \) (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., \( r \notin R \)) 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 \( \Delta \text{dist} \) is a lower bound (\( \text{flag} = \text{false} \)), or go to step 4 if we have exact \( \Delta \text{dist} \).

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

**Step 4.** We need to check \( \text{Valid}(S_i, j, r) \) due to segment updating (line 14). If \( \text{Valid}(S_i, j, r) \), we can assign \( r \) to...
segment $j$ of $S_i$ by invoking $\text{Assign}(S_i, j, r, R, H_i)$. In the $\text{Assign}(S_i, j, r, R, H)$ 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 $\text{InsertSeg}(H, S_i, j, R)$ and $\text{InsertSeg}(H, S_i, j + 1, R)$ to push new entries into $H_i$ 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 $H_i$ and push it into $H^g$ (line 16-18). Then we repeat step 2 to step 5 until $H^g$ 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 $H_i$. Other valid entries with exact incurred distance w.r.t. $c_i$ remain in $H_i$, 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 $H^g$, 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 $15\text{km} \times 5\text{km}$ 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.

**(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 $\lambda$, which denotes the average number of pickup requests arriving per hour on the node. We sample the $\lambda$ values for all nodes from a normal distribution $\mathcal{N}(\bar{\lambda}, \sigma^2)$, where the $\bar{\lambda}$ is the average request intensity on a node. We set $\bar{\lambda}$ 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 $50\text{km} \times 50\text{km}$ 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>The number of transit stations</td>
<td>7</td>
</tr>
<tr>
<td>$X$</td>
<td>The average request intensity on a node</td>
<td>0.6/\text{hr}</td>
</tr>
<tr>
<td>$n$</td>
<td>The number of couriers</td>
<td>500</td>
</tr>
<tr>
<td>$t_s$</td>
<td>The average service time</td>
<td>3 minutes</td>
</tr>
<tr>
<td>$t_c$</td>
<td>The maximum time to confirm a request</td>
<td>15 minutes</td>
</tr>
<tr>
<td>$t_{max}$</td>
<td>The maximum time for each trip</td>
<td>120 minutes</td>
</tr>
<tr>
<td>$n_0$</td>
<td>The number of delivery tasks in $t_{max}$</td>
<td>1,600</td>
</tr>
<tr>
<td>$n_1$</td>
<td>The number of pickup requests in $t_{max}$</td>
<td>10,800</td>
</tr>
<tr>
<td>$\rho$</td>
<td>The average speed of each courier</td>
<td>15 km/h</td>
</tr>
<tr>
<td>$d(r)$</td>
<td>The deadline for a pickup request</td>
<td>30 minutes</td>
</tr>
<tr>
<td>$\lambda_1, \lambda_2 \sim \mathcal{N}(\bar{\lambda}, \sigma^2)$</td>
<td>$\sigma^2 = 0.01$</td>
<td></td>
</tr>
<tr>
<td>$t_i \sim \text{Exp}(\lambda)$</td>
<td>$t_i \sim \text{Exp}(\lambda)$, $T_i = 1/\lambda$</td>
<td></td>
</tr>
<tr>
<td>TABLE 2: Parameters and Default Values</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5: Experimental Setup
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 \( t_{\text{max}} \) time and finish all the assigned delivery in this period. We set \( t_{\text{max}} \) 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 \( \bar{t}_r \), 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 \( \rho \). 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 \( \text{SR} \) 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 \( \text{AID} \) of different algorithms, which is computed as:

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

The average incurred distance \( \text{AID} \) 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) processess 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, \( \text{SR} \) and \( \text{AID} \) for all three algorithms Nearest, Basic, and SIDF* increase. When \( n \) is small, \( \text{SR} \) for all three algorithms increases fast, and when \( n \) is large, \( \text{SR} \) 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 \( \text{SR} \) by 20% on average compared to Nearest, and SIDF* increases \( \text{SR} \) 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 \( \text{SR} = 0.8 \)), 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., \( \text{SR} = 0.8 \)). Regarding \( \text{AID} \), Basic has a smaller \( \text{AID} \) than Nearest when \( n \) is less than 600 and a larger \( \text{AID} \) when \( n \) is larger than 600. It shows that trying to find an optimal schedule immediately after a request stream 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*.

**(Exp-2: Vary \( t_r \)).** We vary \( t_r \), the maximum time to confirm a request, from 1 to 24 minutes and compare \( \text{SR} \) and \( \text{AID} \) for the three algorithms Nearest, Basic, and SIDF*. The experimental results for \( \text{SR} \) and \( \text{AID} \) are shown in Fig. 7 (a) and Fig. 7 (b) respectively. Nearest and Basic are independent of \( t_r \). Thus, both \( \text{SR} \) and \( \text{AID} \) remain unchanged for Nearest and Basic when we vary \( t_r \). For SIDF*, \( \text{SR} \) is better than Nearest and Basic for all \( t_r \) values. When \( t_r \) increases from 1 to 12, \( \text{SR} \) 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 \( t_r \) increases further, \( \text{SR} \) 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 \( \text{SR} \) 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 \( \rho \)).** In this experiment, we vary the speed \( \rho \) of couriers from 15 to 25 (km/h). The trends of \( \text{SR} \) for the three algorithms Nearest, Basic, and SIDF* are shown in Fig. 8 (a). When \( \rho \) increases, \( \text{SR} \) 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,
which allows more requests to be served. When $\rho$ reaches 25, SR of Nearest, Basic and SIDF* is 0.67, 0.75 and 0.92, respectively. The curves for AID when we vary $\rho$ 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.

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 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.
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 cannot 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.

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 urban district of Beijing. The number of couriers, transit stations and delivery tasks are increased in proportion to 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.

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 pickup 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/km² per hour (i.e., 5,400/75km²), 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/km² (i.e., 500 couriers/75 km²) 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


