GeoInformatica (2024) 28:53–88
https://doi.org/10.1007/s10707-023-00497-2

Opportunistic package delivery as a service on road networks

Debajyoti Ghosh[1] · Jagan Sankaranarayanan[2] · Kiran Khatter[1] · Hanan Samet[3]

Received: 7 June 2022 / Revised: 8 February 2023 / Accepted: 7 March 2023 /
Published online: 3 June 2023

Abstract
In the new "gig" economy, a user plays the role of a consumer as well as a service provider. As a service provider, drivers travelling from a source to a destination may opportunistically pickup and drop-off packages along the way if that does not add significantly to their trip distance or time. This gives rise to a new business offering called *Package Delivery as a Service* (PDaaS) that brokers package pickups and deliveries at one end and connects them to drivers on the other end, thus creating an ecosystem of supply and demand. The dramatic cost savings of such a service model come from the fact that the driver is already en-route to their destination and the package delivery adds a small overhead to an already pre-planned trip. From a technical perspective, this problem introduces new technical challenges that are uncommon in the literature. The driver may want to optimise for distance or time. Furthermore, new packages arrive for delivery all the time and are assigned to various drivers continuously. This means that the algorithm has to work in an environment that is dynamic, thereby precluding most standard road network precomputation efforts. Furthermore, the number of packages that are available for delivery could be in the hundreds of thousands, which has to be quickly pruned down for the algorithm to scale. The paper proposes a variation called *dual* Dijkstra's that combines a forward and a backward scan in order to find delivery options that satisfy the constraints specified by the driver. The new dual heuristic improves the standard single Dijkstra's approach by narrowing down the search space, thus resulting in significant speed-ups over the standard algorithms. Furthermore, a combination of dual Dijkstra's with a heuristic landmark approach results in a dramatic speed-up compared to the baseline algorithms. Experimental results show that the proposed approach can offer drivers a choice of packages to deliver under specified constraints of time or distance, and providing sub-second response time despite the complexity of the problem involved. As the number of packages in the system increases, the matchmaking process becomes easier resulting in faster response times. The scalability of the PDaaS infrastructure is demonstrated using extensive experimental results.

✉ Debajyoti Ghosh
4u.debajyoti@gmail.com

[1] BML Munjal University, Haryana, India

[2] Google Inc., Sunnyvale, CA, USA

[3] University of Maryland, College Park MD, USA

Keywords Package delivery · Pickup · Drop-off · Spatial infrastructure · Road networks · Nearest neighbours · Shortest path finding algorithms · Roundtrip · Detours · Landmarks

1 Introduction

The proliferation of delivery apps has resulted in a democratisation of *package delivery as a service* (PDaaS) resulting in users playing the role of logistics companies in picking and delivering packages. PDaaS is opportunistic in the sense that drivers may pickup a package if it is en-route to their predetermined source and destination. The constraint here is the extra distance or time that is added to their trips due to the package pickup and drop-off since that denotes the added expenditure, which eats into the drivers' profits. In this sense, PDaaS is an opportunistic matchmaking service between packages and drivers while supporting driver specified constraints, and packages are delivered by drivers on the road who are on unrelated trips. The setup can be visualised by the two tables shown in Fig. 1 where it is to be noted that both tables are large. There can be hundreds of millions of drivers that are willing to deliver packages, and there are potentially hundreds of thousands of packages that need to be delivered.

The PDaaS provides a separate shipper and a driver user interface where they can independently configure the constraints. The goal of the PDaaS is to connect shippers with drivers in order to ensure that the continuous matchmaking of packages and drivers happens. In the shipper interface, the user provides the package specifications, including the pickup and drop-off locations, at which point the platform computes the shortest path and distance, both in terms of the road network distance as well as elapsed time. Once the shipper confirms the availability of the package at the pickup location, the platform looks for drivers that could opportunistically ship the package from the pickup to the drop-off location.

The driver interface requires the drivers to specify their source and destination addresses. The drivers also specify how much of a *detour* they could tolerate on their shortest path. Furthermore, they also specify how many results to display, which are ordered by the detour distance, and these displayed results are guaranteed to be less than the limit that was specified earlier.

A driver is presented with no choices if there are no packages satisfying the constraints. In our use-case with hundreds of millions of drivers and thousands of packages, the matchmaking

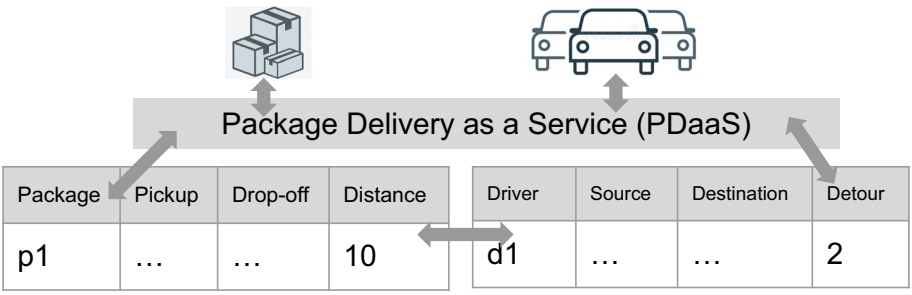


Fig. 1 PDaaS is a matchmaking service that delivers packages using drivers that can opportunistically pickup and deliver packages if they are en-route with bounded detour tolerance. The two tables are large in the sense that there are hundreds of millions of drivers and hundreds of thousands of packages requiring efficient matchmaking algorithms

is expected to be low. A driver who is interacting with a routing app may be opportunistically presented with a choice to deliver a package that has a low detour in relation to their current trip. However, if the driver is interested, they may explore more package options with increasing detours. This means the following aspects of the solution are more interesting.

1. The ability to quickly prune away queries that do not result in a match, which indicates the majority of the cases. Being able to do this quickly means that the platform can be used in conjunction with popular routing apps, which will generate a heavy workload on the PDaaS.
2. Complex cases involving multiple deliveries are not interesting in this domain since the assignment of one package denotes a rare enough outcome.
3. The driver is expected to pickup the package en-route and deliver it before reaching the destination. One can reason here that this is trivially always true since that denotes the smallest detour cost on road networks.

Our PDaaS problem happens in a dynamic environment where new packages are added to the system all the time. Similarly, new drivers are added to the system, and the assignment of packages to drivers happens continuously. Note that the road network can also be updated to reflect current road conditions [4] as well, although this seemingly happens less infrequently compared to the matchmaking between the packages and drivers. When this happens, the package distance or time estimates are also updated so that they reflect the current state of the road network. Given this dynamism, we decouple the drivers, the packages, and the road networks so that they can all be independently updated.

The problem in this paper has similarity to *detour* finding [33, 48] and *ridesharing* [8, 18, 31, 56] on road networks yet is different in important ways. Detour finding looks at the shortest paths that need to pass through a landmark of a certain type. Typically, a detour query would try to route through, say, a *flower shop*, with the least increase in the shortest path. In our case, packages have a pickup and drop-off location which make it similar to the detour problem except for that landmarks now have extents. Ridesharing can be viewed as an optimal route discovery and matching problem between drivers and riders. Given a set of drivers and riders, the ridesharing problem is both an assignment problem-assigning drivers to passengers as well as a sequencing problem that decides the order of pickup and drop-off. The main complexity of the problem comes from the sequencing of the drop-offs, which, if one is not careful, quickly explodes into a combinatorial problem. In our case, sequencing of sources and destinations is not necessary since the driver is expected to pickup and drop-off a single package before reaching the destination. Furthermore, the source of complexity is different. The complexity of our problem comes from the number of drivers, packages, and the constant matchmaking needed to keep the PDaaS operational. In other words, PDaaS has to work in a dynamic environment where there are hundreds of millions of drivers and hundreds of thousands of packages.

The contribution of the paper is the PDaaS infrastructure for efficient matchmaking between drivers and packages. The constraints and solutions we develop are intrinsically aligned with this problem domain. We show how a driver assignment happens in a space with thousands of packages. Furthermore, the solution we develop extends Dijkstra's algorithm [10, 14] to use a single priority queue where "*forward*" and "*backward*" scans as well as the packages found during the search are inserted in the same priority queue. In that sense, we have applied ideas from the *incremental nearest neighbour* algorithm [24] and applied it to road networks. We describe the following criteria for our package routing problem, which make it different from other approaches in the literature, while still being related.

- **[Throughput]** Since PDaaS has to support hundreds of millions of drivers, each search invocation should happen in a fraction of a second to minimise the number of machines needed to support the drivers.
- **[Scalability]** The system should scale with the number of packages, which could be hundreds of thousands. A desirable property we show in the experimental section later is that as the number of packages increases, the system is easily able to make the matchmaking efficiently, and thus the execution time is reduced.
- **[Caching]** Despite the dynamic nature of the PDaaS, we show that a sensible caching strategy can result in tremendous speed-up improvements and demonstrate experimental results to that effect.

The rest of this paper is organised as follows. Section 2 provides the basic concepts and problem definitions. Section 3 describes the existing approaches. Section 4 provides the proposed approach. Section 5 introduces the roundtrip variation of the problem where the driver starts and ends at the same location. Section 6 then introduces the problem in its entire generality. Next, experimental results are presented in Section 7, related work is discussed in Section 8, and finally, concluding remarks are provided in Section 9.

2 Preliminaries

In this section, we first develop the preliminary concepts before describing the problem setup. Our focus in this paper is on developing a PDaaS which is a spatial infrastructure that can assign drivers to packages. The required functionality of PDaaS is described in terms of Application Program Interfaces (APIs) that need to be developed. Next, we provide a high level description of the algorithms that implement key functionalities of PDaaS. We describe a few strawman solutions as a way of motivating the complexity of the problem.

2.1 Notations

Road Network: In our formulation, the road network [16, 17, 47] is modelled as a directed, and weighted graph, $G = (V, E, W)$, such that V corresponds to nodes which are road intersections, while E corresponds to edges which are directed road segments connecting two nodes. W corresponds to edge weights representing either the travel distance or the travel times between two adjacent road intersections v_i and v_j that have an edge between them. Edges are directed in the sense that given that two nodes v_i and v_j with an edge e_{ij} between them, $w(v_i, v_j)$ denotes the cost of the edge. Furthermore, note that $w(v_i, v_j)$ is strictly greater than zero. In other words, there are no negative or zero edge weights. Since G is directed, note that e_{ji} may not necessarily exist, and even if it does e_{ji} may not be equal to e_{ij}. Furthermore, $n = |V|$ and $m = |E|$ denote the number of nodes and edges in the road network.

Shortest Path: Given source s and destination t nodes, let $\pi(s, t)$ denote any simple path between s and t that is not necessarily the shortest. Among all such paths between s and t, let $\pi_N(s, t)$ be the shortest path between s and t formed by an ordered sequence of nodes $s, v_i, v_j, \cdots t$ along the road network. Similarly, $d(s, t)$ denotes the network distance that is obtained by summing up edges formed by node sequence in $\pi(s, t)$. The shortest distance $d_N(s, t)$ is obtained by summing up edges in $\pi_N(s, t)$. Since $w(.)$ denotes either distance or trip time, $d_N(s, t)$ denotes the shortest path using an appropriate unit. Note that G is fully

connected such that given any two nodes u and v, there is a path from u to v and vice-versa, although they may not have the same distance.

Detour: Given source s and destination t nodes, let $\pi(s,t)$ denote a simple path while $\pi_N(s,t)$ denotes a specific path that is shortest among all paths between s and t. The detour of an arbitrary path $\pi(s,t)$ is the additional distance of $\pi(s,t)$ compared to the shortest path $\pi_N(s,t)$. Furthermore, it is fairly trivial to see that the detour of any path is greater or equal to 0. Given k such paths, one can obtain an ordering of the shortest path based on the increasing detour values.

Driver: In our formulation, the driver starts from a source node s and is driving to the destination node t. The driver may specify two constraints on the problem. The driver may bound the detour distance as well as specify that they would like to be provided with k package options along with their detour distances.

Packages: Let P be the set of packages available for delivery such that $p_i \in P$ is denoted by a triple $(\text{PICK}_i, \text{DROP}_i, w_i = d_N(\text{PICK}_i, \text{DROP}_i))$ such that PICK_i is the pickup node while DROP_i is the drop-off node. w_i denotes the shortest distance or time between the pickup and drop-off nodes. Note that the pickups and drop-offs are quantized to nodes on the road network.

2.2 API definition

In the following, we describe the queries we need to support on our PDaaS framework. These would be rolled as APIs that our platform would support. PDaaS is stateful in the sense that the system operator would add and remove packages from the platform. As drivers come in, the system would respond to queries based on the currently available packages P. As packages are assigned, P would be transactionally modified. A package assignment request would work off the state at the start of the query and show results. This means that a package that is selected by the user therefore may not be available when it comes to confirmation time. This is fine since we would show the drivers multiple options, in which case they would move on to the next choice.

AddPackage(p): In this case, the platform would add p to the set of packages P available for delivery. Recall that P is represented by the triplet $(\text{PICK}_i, \text{DROP}_i, w_i = d_N(\text{PICK}_i, \text{DROP}_i))$ as mentioned above. Without loss of generality, the elements of p (i.e., $\text{PICK}_p, \text{DROP}_p$) also lie on the nodes of the road network. There are m packages that are available for delivery and in our case, m can be a very large number and in the order of thousands. The package is added to the system in an offline process which affords some opportunity to push some computations to this offline process. For example, when a package is inserted we can compute the distance between the pickup and drop-off locations of the packages, thus moving a potentially expensive operation offline. Furthermore, once a package is inserted it is associated with the nodes of G such that when at node v, one can pick out all packages that either use v as either the pickup or drop-off location. Note that moving such operations to an offline process is perfectly acceptable since there is no expectation that a package that is added to the system would be immediately scheduled. This means that a few minutes delay between when the package is received and before the system puts the package up for delivery is perfectly acceptable.

RemoveAssignPackage(p): This is a transactional layer where p is removed from P if it still exists. Although, in a busy platform that is assigning packages to drivers, it is possible that multiple drivers might select the same package at the same time, in which case the assignment will be made to one of them. One of the API calls will receive an error that p is not part of

P which denotes that the package has already been assigned to some other driver. It should be noted that package management is not an important part of the PDaaS platform and the algorithms will mainly focus on the package assignment API.

FindPackage(s, t, k, ϵ): This API is invoked by a driver that is driving from s to t on a road network G, where s and t are aligned with the nodes of the road network. There are a few additional constraints that one may specify here. k denotes the number of package options the driver would like to receive. Note that the results are ordered in an increasing detour distance which means that the first result has the smallest detour distance while the k^{th} has the maximum. If k is not specified it is assumed to be one and a single result is returned to the driver. Optionally, one can also provide ϵ which is the maximum allowed detour distance. Here, $d_N(s, t) + \epsilon$ is the farthest package, where $d_N(s, t)$ is not yet known.

The FindPackage() forms the heart of this work and the algorithm that supports this API is discussed in the rest of the paper.

2.3 Problem Definition

Given a set of packages, source and destination, detour tolerance, and the maximum number of packages in the results, find the subset of packages, if any, that satisfy these constraints. This naturally leads to some assumptions and limitations of our PDaaS approach which we have motivated earlier.

1. PDaaS will be on the critical path of a popular app that may issue requests at a high rate. Also, while it is expected that the query load may be high, the rate of matchmaking (the number of drivers agreeing to deliver the package) may be low.
2. The platform is dynamic with hundreds of thousands of packages available at any point. Many are added constantly and are deleted as they are assigned.
3. The focus here is one driver starting from a source and delivering a single package before reaching the destination. As noted above, it can be trivially shown that doing so is the cheapest cost approach when delivering a single package. Note that the delivery of multiple packages is not considered in this paper.
4. A key attribute that one wants here is the scalability of the approach as the number of packages increases. For instance, if the response time is largely independent of the number of packages, or if the algorithm becomes faster with more packages, those are good traits to look for in a solution.

2.4 Complexity

In this section, we first establish the complexity of the problem by proposing a few strawman techniques. Our example setup shown in Fig. 2 consists of a source s and destination t with 3 packages p_1, p_2, and p_3 are available for delivery. To keep the complexity of the example under check, we assume that the packages have a delivery distance of zero, which means that the PICK and DROP are the same, thus only requiring that the driver to navigate via that particular node to make the delivery possible. In this setup, a driver is travelling from source s and destination t and can opportunistically deliver packages. There are many ways of finding paths between s and t that pass through a package. We seek algorithms with a few desirable properties. The algorithm has to work in a setup where there may be hundreds of thousands of packages available for delivery. Furthermore, the driver is only interested in the nearest packages (or k-nearest) and there is no need to fetch the distance to all the packages. With

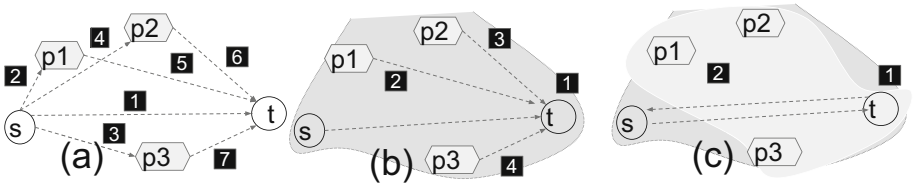


Fig. 2 The figure shows three variations of finding the nearest package among p_1, p_2, and p_3 for a driver travelling from s to t. For each of the variations, the shortest paths that are invoked are numbered in the dark square boxes. The following variations are shown in the figure: (a) Shortest paths between s and t, s to $\{p_1, p_2, p_3\}$ and $\{p_1, p_2, p_3\}$ to t resulting in 7 shortest path computations, (b) Forward scan from s and 3 shortest paths from $\{p_1, p_2, p_3\}$ to t resulting in 4 shortest path computations, and (c) Forward scan from s and backward scan from t resulting in two scans

these constraints in mind, we can examine a few strawmen approaches that fall short in one or more ways.

3 Existing Approaches

No Work Sharing: The simplest approach which is shown in Fig. 2(a) is where we first compute the shortest path between s and t. This is typically done using a best-first traversal algorithm which uses a priority queue. Then we compute the distance from s to p_1, p_2, and p_3 as well as compute the distance from the packages p_1, p_2, and p_3 to t. In other words, none of these best-first traversals share any work with one another. Once we have all the elements of the shortest paths, we can choose the package with the minimum detour which forms the result. The drawback of this approach is it issues 7 shortest path computations but it is conceivable that one can combine some of these traversals to improve efficiency.

Merging Forward Scans: We can reduce the number of shortest path computations by nearly half by using a best-first traversal from s which only terminates after it has visited t, p_1, p_2, and p_3. Once the algorithm has visited these four nodes, the "forward" distance from s has been established. Now, we invoke the shortest path algorithm from p_1, p_2, and p_3 to t as before to finish computing all the component distances needed to compute the nearest package from s and t. As shown in Fig. 2(b) this can be achieved using 4 shortest paths. Note that this approach still requires computing all the component shortest distances between s, t and p_1, p_2, and p_3, which makes this approach not scalable.

Using Three Priority Queues: We can further improve the previous algorithm by using the *bidirectional* Dijkstra's algorithm [3, 38] which uses two separate Dijkstra's algorithms, one from s and another from t. As shown in Fig. 2(c), we apply one best-first traversal from s that uses the outgoing edges incident on s and is called a *forward* scan. On the other hand, the best-first algorithm from t traverses using the incoming edges to t and hence is called the *backward* scan. When the forward scan and backward scans meet at a node for the first time, the road network distance between s and t is established. Similarly, when the forward and backward scans meet at a package node, then the shortest path via the package is also established. The algorithm has two priority queues, one for the forward scan and another for the backward scan. These two scans run independently and do not coordinate with each other. When the forward and backward distances to a package have been established, it is inserted into a third priority queue containing packages whose distances are known. Note that the algorithm can only terminate once the distance to all the packages has been established and all the packages

are in the third priority queue. In other words, there are very few opportunities for pruning of the packages or the early termination of the algorithm which makes it not suitable for our use-case.

4 Proposed approach

As discussed above the bidirectional Dijkstra reduces the invocation to two traversals, but it is still tricky to adapt it for our use-case. Recall that in our use-case we have hundreds to thousands of packages but the driver is only interested in a few choices (i.e., k). A real drawback of this approach is that it uses two independent traversals so it becomes very difficult to effectively prune the search. In the following, we will develop a variant of Dijkstra's algorithm called dual Dijkstra that will use a single min-priority queue Q to perform the traversal. In the rest of the paper, by Q, we denote the min-priority queue.

Q stores the *forward* and *backward* traversals as well as packages found during the scan. The use of a single priority queue means that we can establish an ordering of the packages by their detour where we can guarantee that all packages not yet retrieved from Q have a larger detour than the ones that have been already retrieved. This key insight in using a single priority queue is analogous to the *incremental nearest neighbour traversal* in [24] where the authors use a single priority queue to store objects and points which enabled an incremental traversal of the nearest neighbours. The mechanics of the dual Dijkstra's are subtle and we discuss these in detail in the next section.

5 Roundtrip package delivery problem

As a matter of exposition, we start with a simpler case of the problem where the driver starts and ends at the same location. In this case, since $s = t$, the driver would deliver the package from s and return back to s. This variation of the problem is called the "roundtrip" and is simpler to analyse than the "generalised" variant. This is the reason that we analyse the roundtrip variation first before considering the problem in its full generality.

In this section, we describe the mechanics of the dual Dijkstra's method, which forms the core of the techniques developed in the paper. In order to understand the nature of the problem, we first study the properties of a simpler variation of the problem where the source and destination nodes are the same. We represent this node as q. The roundtrip distance of a node v is the distance along the shortest path from q to v and back to q, which is the shortest among all the paths in G. In this variant of the problem, the driver is at q and needs to return back to q after delivering a single package. We are given a set of packages P where each package is represented by a triple $(\text{PICK}_i, \text{DROP}_i, \text{DIST}_i)$ denoting the pickup, drop-off, and distance of travel between pickup and drop-off node. In this formulation of the problem, we are interested in the nearest package whose delivery denotes the shortest roundtrip distance from q. Here, the roundtrip distance of the package i is given by $d_N(q, \text{PICK}_i) + \text{DIST}_i + d_N(\text{DROP}_i, q)$. Among all packages in P, we want to find the one with the shortest trip distance from q. While this problem seems simple, there are interesting algorithmic challenges exposed in this variant. In the following, we develop a few key concepts which form the core of our approach to solving the package delivery problem.

Definition 1 A forward best-first traversal of a graph from node q is a traversal that at any stage of the algorithm inserts the outgoing nodes into a priority queue. If a node v is retrieved from

the front of the priority queue, its outgoing nodes are inserted with weights corresponding to the distance at which v was found plus the corresponding edge weights of the outgoing nodes.

Definition 2 A backward best-first traversal is similar to the forward except that in every step the incoming nodes are inserted into the priority queue.

Lemma 1 *Once the forward and backward traversals from q meet at a node t, the shortest path from q (to q) that passes via t have been established.*

Proof We can establish the proof by contradiction. Suppose the meeting of the two traversals does not produce the shortest path, then either the forward or backward, or both traversals do not produce the shortest path from q to t, and t to q, respectively. This means that the best-first traversal from q did not produce the shortest path, which is a contradiction.

Now it is easy to see that the forward and backward traversals can establish the trip distance from q that delivers a package i. It is given by the following lemma, which follows from the previous one.

Lemma 2 *Once the forward traversal from q reaches* PICK_i *and the backward traversal from q reaches* DROP_i*, the trip distance of delivering a package is given by the forward distance from q to* PICK_i *plus* DIST_i *plus the backward distance of q to* DROP_i*.*

Proof The proof of this lemma follows from Lemma 1 It is trivial to see that if PICK_i is the same as DROP_i (in other words, DIST_i is 0), in which case this becomes a trivial application of the above Lemma. For the case where DIST_i is not 0, one can see that once PICK_i is reached by the forward traversal, the forward distance of DROP_i becomes the sum of the forward distance of PICK_i plus DIST_i. This comes from the property that any sub-path of a shortest path is also a shortest path [10]. At this point, the proof degenerates into a trivial application of Lemma 1.

Algorithm 1 Setup for finding the roundtrip nearest package from q.

```
1: q ⇐ query node
2: G(V, E) ⇐ road network with edge weights > 0
3: P is set of packages, each package is a triple (PICK_i, DROP_i, DIST_i)
4: Object O is a triple {v, dist, label} of node v, road network distance dist and label indicating forward or
   backward traversal
5: Q ⇐ min-priority queue of O ordered by dist.
6: Lookup table T(v_i, {FORWARD‖BACKWARD}) of either forward or backward distance from q.
7: VISITED(v_i, {FORWARD‖BACKWARD}) returns either true or false if a node is already visited.
8: VISITED(q, FORWARD) = TRUE
9: for each outgoing edges (q, v_i) ∈ E do
10:     Q.Insert({v_i, w(q, v_i), FORWARD})
11: end for
12: VISITED(q, BACKWARD) = TRUE
13: for each incoming edges (v_i, q) ∈ E do
14:     Q.Insert({v_i, w(v_i, q), BACKWARD})
15: end for
16: d_N ⇐ ∞                          ▷ Distance estimate to the roundtrip nearest package.
```

Armed with the basic properties of forward and backward traversals, we now describe the working of our dual Dijksta's approach which uses a single priority queue to find packages

in a best-first order. Algorithm 1 sets up the dual Dijkstra's implementation for finding the roundtrip nearest package from q. The goal is to find the roundtrip distance of a package with the smallest trip distance from q. The input to the algorithm consists of a road network G with positive non-zero edge weights and it outputs the smallest roundtrip distance from q. We are also given a set of packages P that need to be delivered. Each package p_i in the set is denoted by a pickup node PICK_i, a drop-off node DROP_i and the road network distance DIST_i between the corresponding pickup and drop-off nodes. In contrast to other approaches, our technique uses a single priority queue of objects O, which are made up of triples consisting of the node where the traversal is current is at, the nature of the traversal denoted by *label* (i.e., FORWARD or BACKWARD) and the distance from the starting node from which the traversal is being done (i.e., q in this case). Note that the priority queue also contains package objects (i.e., *label* is denoted by PACKAGE). These correspond to packages whose forward and backward distances are known but need to be inserted in Q before they can be part of the final result. As one can see that not only Q contain nodes but also candidate packages that can potentially form the answer. $O.v$, $O.dist$, and $O.label$ extract the node, road network distance, and label of the traversal, respectively. Here $O.label$ can either be a forward or backward scan, or a package. Note that the algorithmic elegance of the dual Dijkstra's approach is that it can put both the forward and backward scans as well as the packages on the same priority queue and thus can retrieve packages in a best-first manner. We make use of two data structures for storing the current state of the algorithm. T is a table that stores the forward and backward distance of a node from q. It is initially empty since the distance is only populated when the node is retrieved from the front of Q. Furthermore, we have a data structure VISITED that keeps track of which nodes have been visited by the algorithm, in other words, VISITED(.) data structure is used to avoid having to revisit already visited nodes. Note that since we have two traversals at the same time, this data structure needs to keep track of which traversal (i.e., FORWARD and/or BACKWARD) has visited a node. Finally, d_N keeps track of the distance to the roundtrip nearest package thus far and this estimate keeps decreasing as the algorithm progresses. Q is updated with the forward and backward traversals from q. In this case, the outgoing and incoming nodes are inserted into Q with their appropriate labels. Note that q is marked as visited in both the forward and backward traversals since the driver starts and ends at this node.

The rest of the algorithm is described in Algorithm 2. Line 1 captured the terminating condition of the main while loop of the algorithm. The algorithm terminates if Q is empty in which case the whole graph has been explored or if the front of Q is at a distance more than d_N. Since the correctness of the algorithm depends on setting the correct value of d_N, we will have a lemma later to show that we are correctly updating it. First, we examine the object O that has been retrieved from the front of Q. In lines 2-4, if a package is found at the front of Q, the loop terminates and the algorithm returns the identity of the package which is stored in $O.v$ and the roundtrip distance $O.dist$. If O is a node then in lines 5-9 we look in VISITED to see if those nodes have been traversed before. If so, we reject it and continue with the traversal, and else we mark those nodes as visited. Note here that every object has a label indicating if the traversal is forward or backward so care should be taken to ensure that the label is correctly reflected in the VISITED data structure.

At this point in line 10, we check to see if we need to update T corresponding to the node $O.v$ and the label $O.label$. Note that if the distance is already set, it cannot be improved further. This comes from the best-first nature of the forward and backward scans on G.

When the node found is a forward scan and $O.v$ equals the pickup node of the package i and the drop-off node DROP_i is already found, which means that the forward scan has found a detour path that involves dropping off a package i before returning to q. Similarly,

Algorithm 2 Fetch the roundtrip nearest package from q.

```
1: while !Q.empty() && O ⇐ Q.front() && O.dist < d_N do
2:     if O.label == PACKAGE then
3:         return package id stored in O.v and roundtrip distance in O.dist
4:     end if
5:     if VISITED(O.v, O.label) == TRUE then                    ▷ Indicates cycle
6:         continue
7:     else
8:         VISITED(O.v, O.label) = TRUE
9:     end if
10:    if Not EXISTS T(O.v, O.label) then
11:        Update T(O.v, O.label) with O.dist
12:        if exist T(PICK_i, FORWARD) and T(DROP_i, BACKWARD) then
13:            p ⇐ T(PICK_i, FORWARD) + DIST_i + T(DROP_i, BACKWARD)
14:            d_N ⇐ min(d_N, p)
15:            Q.Insert(i, p, PACKAGE)                          ▷ Insert i^th package
16:        end if
17:    end if
18:    if O.label == FORWARD then
19:        for outgoing nodes v_i of O.v s.t. !VISITED(v_i, O.label) do
20:            Q.Insert(v_i, O.dist + w(O.v, v_i), FORWARD)
21:        end for
22:    else
23:        for incoming nodes v_i of O.v s.t. !VISITED(v_i, O.label) do
24:            Q.Insert(v_i, O.dist + w(v_i, O.v), BACKWARD)
25:        end for
26:    end if
27: end while
28: return (−1, infty)                                           ▷ Package not found
```

in the backward scan, $O.v$ corresponds to the drop-off location of a package i, and the pickup location is already found by the forward scan, we have found a roundtrip path to package i. In this case, we insert an object into Q that stores the identity of the package i, the distance that equals the distance from q to PICK_i plus DIST_i plus the distance from DROP_i to q. The forward traversal captures the distance from q to PICK_i, DIST_i has been cached, while backward traversal captures the distance from DROP_i to q. This is captured in lines 12-16.

It is to be noted that packages found in lines 12-16 cannot be directly reported once they are "fully resolved" since they are fetched out of order. Here fully resolved means that the forward traversal has reached the pickup node and the backward traversal has reached the drop-off node. In other words, it is possible that the algorithm may find a package with a smaller detour later after resolving a package. This is a noteworthy property of this algorithm and is captured as in Lemma 3 A package can only be safely reported after it has been retrieved from the front of Q. This is the reason that we reinsert the fully resolved package into Q. Furthermore, once a package is resolved, we update d_N if this package has a smaller roundtrip distance. Again, it is not guaranteed that every subsequent package improves d_N, which is also captured by Lemma 4

Finally, depending on the nature of the traversal, in $O.label$, in the case of forward traversal, all the outgoing nodes of $O.v$ are inserted into Q. In the case of backward traversal, all the incoming nodes are inserted into Q. The algorithm loops till either an answer is returned or Q is empty. If Q is empty, it means that the algorithm did not find any package (line 28) in which case the algorithm returns an invalid package identifier to the calling function to indicate an answer was not found. At termination, the algorithm returns the package and the shortest roundtrip distance from q if one exists.

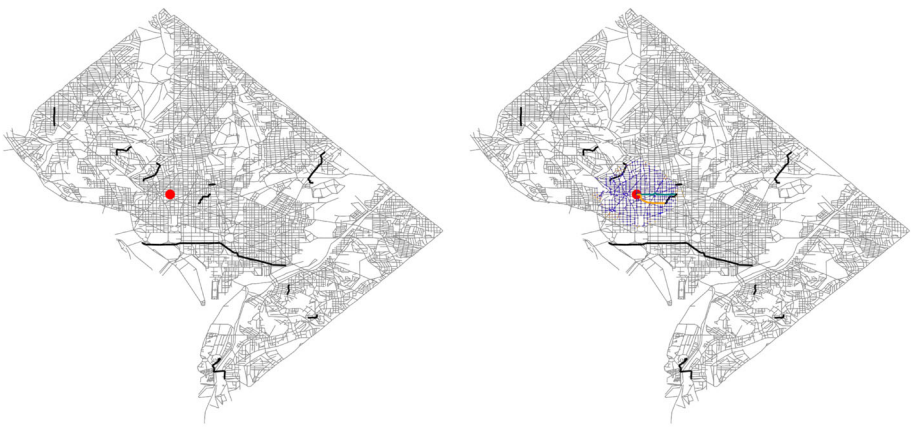
5.1 Working example

Figure 3 shows the mechanics of finding the roundtrip nearest neighbour. Given q and packages P that are available for delivery. q is represented by the *red* circle while the packages are marked using black lines. Note that the shortest path and distance between the pickup and drop-off nodes of each of the packages are cached and available before the start of the algorithm. Figure 3(b) shows the progression as the algorithm found the first package. In this case, the blue node colour roughly shows the progression of the forward and backward traversals. The package is now connected to q via a green path which denotes the shortest path from the forward traversal and an orange path from the backward traversal.

5.2 Properties of Algorithm

In the following, we showcase the property of the dual Dijkstra's algorithm using it to show interesting aspects of the package delivery algorithm. The dual Dijkstra uses a forward and a backward scan to speed-up the search process. When these scans meet at a node, the shortest roundtrip path is established. There is a subtlety that one needs to be careful of here since that affects the correctness of the algorithm. Note that while we can guarantee the shortest distance to a node, we cannot guarantee that the next node we visit will have a larger roundtrip distance. In other words, to provide the best-first property on the roundtrip distance we have to insert the node (or equivalently the package) into Q. Only fully resolved nodes (or packages) obtained from the front of Q are retrieved in a best-first manner by their roundtrip from q. The following lemmas are critical to the correctness of the package algorithm.

Lemma 3 *When a package is retrieved from the front of Q, it is the nearest package of q. In other words, the algorithm is a best-first retrieval of packages of q.*



(a) q and P (b) Nearest Package

Fig. 3 The figure shows the working of the roundtrip package delivery algorithm. Figure (a) shows the driver location q and the packages P available for delivery, while (b) show the roundtrip path that delivers a single package returning back to q

Proof Without loss of generality and for the sake of simplicity, let us assume that the package needs to be picked up and dropped off at the same place. In other words, for any package i, DIST_i is 0. The package distance is the sum of two best-first traversals, FORWARD and BACKWARD. This means that two components making up the roundtrips will, individually, will be non-decreasing as the algorithm proceeds. In other words, the traversal after finding this POI p will not find another one, say p' with a smaller FORWARD or BACKWARD. However, it is possible that Q contains a POI p'' node with a small FORWARD. When p'' is finally found, it would have a larger BACKWARD distance yet the roundtrip distance p'' of q can be smaller than p. This is the reason we cannot report packages once FORWARD and BACKWARD distances are established but rather have to reinsert them back into Q. Once, a package p is retrieved from the front of Q, we know that the other POIs in Q have a roundtrip of the same distance or greater due to the ordering property of the priority queue. In other words, the algorithm produces a best-first traversal of the nearest packages since it finds packages in non-decreasing order.

Lemma 4 *When a POI is found at the front of Q, the other POIs in Q are either at the same roundtrip distance from q or greater.*

Proof Follows from the best-first nature of package retrieval in Lemma 3. □

Once we have studied the nature of retrieval, we can now establish the stopping condition of the algorithm in the following lemma.

Lemma 5 *The algorithm can only stop for the object at the front of Q is at a distance that is greater than or equal to d_N. At that point, the algorithm returns the exact roundtrip nearest package to q.*

Proof From Lemma 3 we already established that the roundtrip distances are fetched in a best-first manner. Once d_N is established, it is conceivable that the algorithm will improve d_N in the subsequent iterations. When the front of Q is at a distance of d_N, it means that all subsequent roundtrip distances of the nodes in Q would be greater than or equal to d_N for directed graphs with non-negative edges. Hence, the lemma follows. □

6 Generalized package delivery problem

In the previous section, we discussed a delivery problem where the sources and the destinations were the same. In a generalised version, the source and destination are not the same which means that the driver is already travelling from the source to the destination and the package delivery is opportunistic in nature in that the driver delivers the package along the way.

Algorithm 3 captures the main changes needed to support the generalized package delivery problem. In this case, the driver is travelling from source s to destination t. So, we initialise the forward scans from s and the backward scans from t. Recall that in the roundtrip package delivery problem initialised the forward and backward scans from q, which have now been broken into s and t. Note that the following change is sufficient and produces packages that are best-first in the detour distance from s to t.

Algorithm 3 Setup for generalised package delivery from s to t.

1: $s, t \Leftarrow$ source and destination
2: ϵ is the allowed detour beyond the shortest distance between s and t
3: VISITED(s, FORWARD) = TRUE
4: **for** each outgoing edges $(s, v_i) \in E$ **do**
5: Q.Insert($\{v_i, w(s, v_i)$, FORWARD$\}$)
6: **end for**
7: VISITED(t, BACKWARD) = TRUE
8: **for** each incoming edges $(v_i, t) \in E$ **do**
9: Q.Insert($\{v_i, w(v_i, t)$, BACKWARD$\}$)
10: **end for**

Lemma 6 *Algorithm 3 can trivially support the case of general package delivery where the driver is already travelling from s to t by changing the initialization of forward scan to start from s and backward scan to start from t. No other changes are needed.*

Proof The correctness of the lemma can be trivially seen in that q is not even part of Algorithm 3. When the forward and backward scans meet at the pickup and drop-off locations, respectively, the package is inserted into Q. It does not matter where these scans started and the algorithm is agnostic to it. Hence, the proof.

In the generalised delivery problem, we extend the algorithm to allow drivers specify two restrictions on the kinds of packages that are shown to them. First, it would provide the driver with a fixed number of options. Here k denotes the number of options that are shown to the driver. The intuition behind showing k options is the driver may choose to deliver a package that does not necessarily have the smallest detour. For instance, the package delivery may be in the part of the town where the driver is familiar or the package pays more so the driver may choose the package in lieu of packages that are mostly along the way. Next, the driver may specify a constraint on the detour which restricts the choices shown. For instance, the driver may specify that they are only interested in packages involving less than say 5 km (or equivalently a restriction on the trip time). Note that the best-first nature of the algorithm is a desirable property since that allows the algorithm to terminate early without having to retrieve the detour involved in all the packages. In the following section, we will see that the prior algorithm developed for roundtrip deliveries can be trivially extended to this generalised problem with fairly minor changes.

Limiting the detour requires finding only those packages that are no more than $d_N(s, t)+\epsilon$ where ϵ is the user specified detour tolerance. In this case, note that $d_N(s, t)$ can only be established after the shortest distance between the s and t is first found. To do this seamlessly, it is necessary that the shortest path be established before the packages are found to avoid pruning once the packages are retrieved from the top of Q. Algorithm 4 retains most of the prior Algorithm 2 and the common lines between both these versions are not shown here. The main change comes when inserting incoming or outgoing nodes of the node $O.v$ which was retrieved out of Q. In this case, we additionally check that if the traversal is forward and if the outgoing node v_i already has a backward distance that is established, then d_N can be updated if it is greater than the just established forward distance of v_i (i.e., $O.dist + w(O.v, v_i)$) plus the already established backward distance from v_i (i.e., $T(v_i$, BACKWARD) plus ϵ. This is shown in lines 7-9. Similarly, we can do the same for the backward traversal which is shown in 14-16.

Algorithm 4 Fetch the nearest package from s to t along with the detour limit.

```
1: while !Q.empty() && O ⇐ Q.front() && O.dist < d_N do
2:     ···
3:     ··· 16 common lines
4:     ···
5:     if O.label == FORWARD then
6:         for outgoing nodes v_i of O.v s.t. !VISITED(v_i, O.label) do
7:             if VISITED(v_i, BACKWARD) == TRUE then
8:                 d_N = min(d_N, O.dist + w(O.v, v_i) + T(v_i, BACKWARD) + ϵ)
9:             end if
10:            Q.Insert(v_i, O.dist + w(O.v, v_i), FORWARD)
11:        end for
12:    else
13:        for incoming nodes v_i of O.v s.t. !VISITED(v_i, O.label) do
14:            if VISITED(v_i, FORWARD)) == TRUE then
15:                d_N = min(d_N, O.dist + w(v_i, O.v) + T(v_i, FORWARD) + ϵ)
16:            end if
17:            Q.Insert(v_i, O.dist + w(v_i, O.v), BACKWARD)
18:        end for
19:    end if
20: end while
21: return (−1, infty)                                   ▷ Package not found
```

6.1 Limiting detours

The correctness of this algorithm relies on the fact that the shortest path is guaranteed to be found before the packages are found. This is captured by the following lemma.

Lemma 7 *Detour is established before any package is retrieved from Q in Algorithm 4, which means that any package that is reported by the algorithm is within the detour tolerance limit.*

Proof The pathological case to discuss here is that of a package p whose pickup location is s and the drop-off location is t, with the driver specified detour tolerance ϵ set at 0. In other words, the driver does not want to deviate from the shortest path and the package exactly aligns with the source and destination. To prove the lemma, it would be sufficient to show that d_N is established before p is obtained from the front of Q. Now, p is inserted at the start of the algorithm at a distance equal to $d_N(s, t)$, and not retrieved from the front of Q right away. The key thing to show here is that d_N is set to $d_N(s, t)$ before p is fetched from the front of the q. This can be seen by observing that the d_N is established just before the forward and backward scans coincide. Recall that d_N is set when a node has already been visited by a forward node and its forward edge has already been visited by a backward edge. In other words, when d_N is set to the $d_N(s, t)$, the front of Q is strictly less than $d_N(s, t)$ if edge weights are greater than 0. This means that p is still in Q and the algorithm terminates after p is reported as the package that satisfies the driver's constraints.

6.2 *k*-nearest neighbours

As mentioned before the driver may be interested in obtaining k package delivery choices ordered by the detour distance. Typically k is small, although that is not a requirement. In the following, we discuss further changes needed to the algorithm to support this top k feature.

Lemma 8 *Algorithm 4 can be trivially made to return k packages by just keeping track of the number of packages retrieved from the front of Q.*

Proof Follows from Lemma 3. The best-first nature of the algorithm means that the algorithm can incrementally retrieve k neighbours or terminate when Q is empty.

6.3 Using landmarks

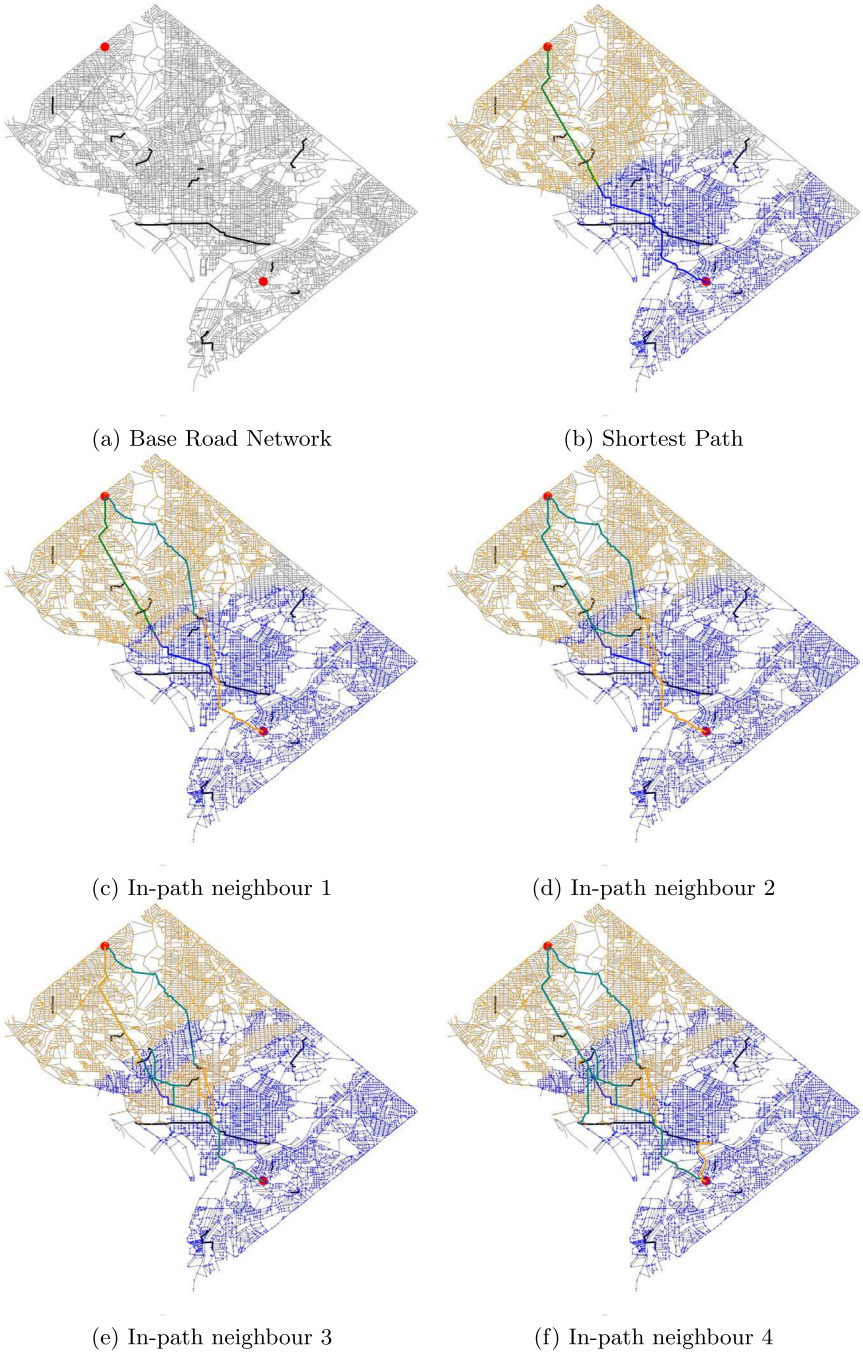
An interesting variation of the algorithm is that one can effectively incorporate caching techniques into the solution resulting in desirable performance characteristics. Recall that the packages are inserted into the system but there is no requirement that they are instantly made available to the drivers for matchmaking. One can leverage this property in order to ensure that the shortest distances to and from the package pickup and drop-off nodes are cached. In other words, for each package that is inserted into PDaaS, we cache the backward distance from the PICK and the forward distance from DROP. There are two considerations that one needs to be mindful of. First, the cost and the latency of computing the shortest distances are not trivial but they are now part of an offline process. Second, the size of the cache is $O(n)$ for each package. Note that even large cities and suburbs in the US have less than 500k nodes which means that such a cache can fit within 1 MB. Furthermore, one can either apply statistical compression or logical compression (e.g., SILC [42]) in order to make the representation smaller. These are all good strategies to implement but are not the main focus of this work.

We distinguish between the precomputation approaches such as [44] and our caching approach used in this paper. In a precomputation approach, a data structure is constructed using an offline process. The efficient working of the algorithm is reliant on the right data structure being chosen; otherwise, the algorithm will not run efficiently. For instance, in [44] if the path oracle is not precomputed efficiently or not available, path queries cannot be answered quickly. On the other hand, in this paper, we simply *cache* and preserve the forward and backward distances from each package pickup and drop-off location. This is the byproduct of computing the network distance between the pickup and delivery, which is needed to set up our problem. Note that if this information is missing, the algorithm still produces the correct answer albeit the run times will be slower. The opportunistic speed-up that caching of forward and backward distances distinguishes the caching approach here from other precomputation techniques.

The caching strategy proposed here is akin to landmarks [20, 21, 30, 37, 39, 40, 49] on the road network that is used to quickly prune the search space of shortest paths. Here, we cache the distance from the pickup and drop-off location such that during answering an actual query if the forward scan reaches the pickup node, we can compute the roundtrip distance to the destination by using the forward distance from pickup to the destination. Similarly, if the backward scan reaches the drop-off node, we can compute the distance from the pickup node. The advantage of the landmark method is that if either the forward or backward scan finds the pickup or drop-off of a package respectively, the package is fully resolved and can be inserted into Q. We provide some properties of the landmark algorithm using the following lemmas.

Lemma 9 *If the forward scan reaches the pickup location of a package or backward reaches drop-off, the package is fully resolved and can be inserted into Q.*

Proof When the forward scan reaches the pickup location, we know the forward distance to the pickup, the distance between pickup and drop-off (i.e., DIST), and the forward distance from drop-off to the destination using the landmarks.



(a) Base Road Network (b) Shortest Path

(c) In-path neighbour 1 (d) In-path neighbour 2

(e) In-path neighbour 3 (f) In-path neighbour 4

Fig. 4 Figures show the progression of dual Dijkstra's algorithm. Notice that the in-path neighbours are retrieved out of order, which is one reason they would have to be reinserted into Q to retrieve them in order of their shortest path distance

Lemma 10 *In the dual Dijkstra's algorithm with landmarks, efficiency improves with the number of packages.*

Proof In the following, the assumption is that the pickups and drop-offs are uniformly distributed across the road network. As the number of packages increases, it is likely that the forward and backward scans would more frequently encounter packages. Since we have the cached landmark distances, these would fully resolve the packages which are then inserted into Q. Note that if we did not have the landmarks, a package can only be inserted once the forward and backward scan reached the pickup and drop-off locations, respectively. Once a package is found by either of the scans, d_N can be updated since we now have the distance between s and t. As experimental results would show later, if k and detour limits are specified, the algorithm with cached landmark performance starts improving till it becomes as efficient as the shortest path finding between s and t. This is a desirable property for PDaaS which has to support hundreds to thousands of packages at any moment.

6.4 Working of the dual Dijkstra's algorithm

Figure 4 shows dual Dijkstra's algorithm execution progress. In the following, we describe the algorithm progression that is captured by the sequence of figures.

1. In Fig. 4(a) we can see the source and destination are shown by red dots while the packages are shown by dark black lines.
2. In Fig. 4(b) we can see the progression of the algorithm. The forward scan is shown by the blue dots and the backward shown by yellow dots. This figure captures the moment that these two traversals meet and the shortest path is established.
3. In Fig. 4(c) the first package is found when the forward scan visited the pickup and the backward visited the drop-off. Note that in this case, the pickup was visited earlier and when the backward traversals visited the drop-off the package's detour distance is established. Further note that the shortest distance that passes through the pickup and drop-off locations of this package for a path that is disjoint from the shortest path between s and t.
4. In Fig. 4(d–f) more packages are found in increasing detour distances. The algorithm terminates when the sufficient number of packages are found or the detour limit is exceeded.

7 Experimental results

The experiments were conducted on a GPU consisting of an AMD Ryzen Threadripper 3960 CPU, which has multi-processing power. It has 48 threads and 24 cores, having 2.2 GHz clock speed, 128GB RAM, 4TB HDD, and 512 KB cache memory. The implementation of proposed algorithms is done using Python programming language, version 3.10, running in 64-bit Ubuntu Linux operating system, version 20.04.3, Kernel version 5.4.0-125.

The road networks used in the evaluation were drawn from the 9th DIMACS Implementation Challenge [11]. In particular, we used three datasets of varying sizes to demonstrate the performance of our algorithms at varying scales. The sizes of these datasets are provided in Table 1. We use the following scheme to distinguish between the three datasets. The square symbol in the graphs refers to San Francisco SF, the circle symbol to US-NW and the diamond symbol refers to the CA-NV dataset. The weight of each edge in the datasets corresponds to the road network travel distances between two nodes on a directed edge.

Table 1 Dataset used in experimental evaluation

Dataset	Nodes	Edges
San Francisco Bay Area (SF)	3,21,270	8,00,172
Northwest USA (US-NW)	15, 24,453	38, 97,636
California and Nevada (CA-NV)	18,90,815	46, 57,742

We also consider two distribution models for packages on the road network. We first evaluate the case where the packages are evenly distributed over the road networks, and a number of parameters that affect the running time of the algorithms are examined and studied. We also consider the case where the pickup and drop-offs are clustered, creating a more challenging scenario for the algorithms under study.

Table 2 lists the parameters we vary to study the performance of the algorithms under various settings. To observe the effect of a parameter on the performance of the proposed algorithms, we vary the value of one parameter while fixing the others. For each experiment, we choose the source and destination at random and we take the average across 100 runs. For each experiment, we collect the following three metrics since they inform important properties of the algorithms: (1) Elapsed time of getting the program output (in seconds), (2) Maximum size of Q used during query execution, and (3) The number of packages inserted into Q. Elapsed time helps us to understand the performance of the algorithms and their suitability in supporting a real-time workload from the drivers. The size of Q informs the memory footprint required in supporting a single request to the PDaaS framework. This is needed to "size" the platform in support of the input workload. For instance, if the input request rate is known, we can now determine how many machines and their CPU-memory shapes are needed to support such a workload. Finally, the number of packages in Q informs the pruning power of the algorithms. In particular, a better algorithm is able to prune away a majority of the available packages focusing the search only on packages that can potentially be part of the answers.

The main variants we compare in this paper are single and dual Dijkstra's algorithms and their landmark variants. The existing algorithms [1, 5, 6, 25, 29, 31, 50] either precompute on the road network or assume a non-dynamic environment; both of these assumptions are not true for our problem setup. While the landmark variant discussed in the paper merely caches a byproduct of computing the distance between the pickup and drop-off locations. This cache aids in the best-first search for packages on the road network. Note that the algorithm works even if this cache is missing for a particular package, and hence is not germane to its correctness. In contrast, the above methods use the precomputation representation as the primary access structure or the main vehicle to apply the optimization. The correctness or proper functioning of the algorithm is contingent on the precomputed representation being available and current with respect to the present state of the system.

Table 2 Parameter setting used in experimental evaluation

Parameters	Values
Datasets	SF, US-NW, CA-NV
Desired k-nearest packages	1, 10, 50, 100, 500
Detour tolerance (in distance units)	10, 100, 1000, 10000, 100000
Number of packages	10, 100, 200, 500, 1k, 5k, 10k, 20k, 50k, 100k
Source-destination distance buckets	>10k, >50k, >100k

In the following, we use the single Dijkstra's as the baseline and show the improvements of the dual and landmark based approaches. In particular, we consider the following algorithm in our experimental study.

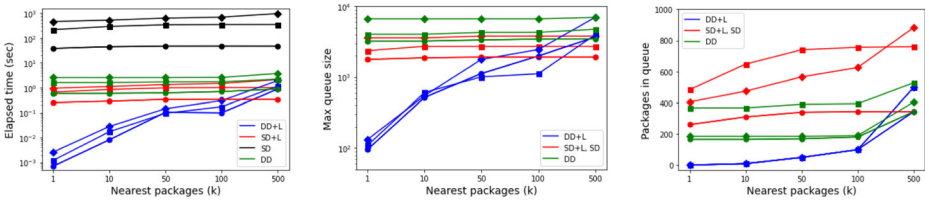
- **SD**: Single Dijkstra's algorithm that uses a single priority queue. In this case, when the forward traversal from the source reaches a package, the shortest path is involved from the drop-off location to the destination node.
- **DD**: Dual Dijkstra's algorithm where we apply a forward traversal from the source and a backward traversal from the destination. When the forward traversal visits a pickup location and the backward traversal arrives at the drop-off location of the same package, the roundtrip distance is established and the package is inserted into the priority queue.
- **SD+L**: This is the landmark variant of the SD algorithm where each package has cached the forward distance from the drop-off location to all the nodes in the road network. Note that when a package is first inserted into the system one can cache the distances to all the destination nodes in the road network. This becomes quite useful during query answering. When the forward traversal visits the pickup location of a package, we can immediately compute the round-trip distances to the actual destination, which makes such algorithms quite efficient.
- **DD+L**: This is the landmark variant of the DD algorithm. In this variant, we compute the backward distance from the pickup location and the forward distance from the drop-off location. If either the forward or the backward traversal visits the pickup or drop-off locations, then the shortest path involving the package delivery is established.
  Note that SD+L is strictly a latency improvement. It does not significantly change the pruning power of the algorithm other than the latency, so all the other metrics are unaffected by the landmarks. This is the reason that for all the metrics graphs other than latency we have merged the lines for SD and SD+L in the graphs in this section since they have otherwise identical performance.

At the high level, there are two interesting scenarios of this algorithm that are commonplace in our daily lives. The driver is at a source location, pickup and drops the package before returning back to the source location. This is referred to as the roundtrip query. We present experimental results for this scenario in Section 7.1. The more general version of the algorithm is where a driver is already travelling from a source to the destination location. We present experimental results in Section 7.2.

For each scenario, we examine the results of varying the number of nearest packages (i.e., k). If our application had a user interface, k would denote the number of choices we would present to the user. We examine the effect of using a detour limit tolerance. In this case, we would prune away any package delivery that would result in a detour more than what is provided by the user. Next, we look at the effect of varying the distance between the source and destination, the number of packages, their distribution across the road network and the distances between the pickup and drop-off locations. As one can see, we present a comprehensive algorithm and try to tease out the strengths and weaknesses of the various approaches.

7.1 Roundtrip package delivery queries

Varying Number of Neighbours This use-case corresponds to the scenario where the driver is at a source location and would return to the same location after delivering the package. We first vary the number of k-nearest packages to be fetched from 1 to 500 while the detour limit was kept fixed at 1M. The number of packages available for delivery is fixed at 1000.



(a) k with elapsed time

(b) k with maximum size of the priority queue

(c) k with number of packages inserted in the priority queue

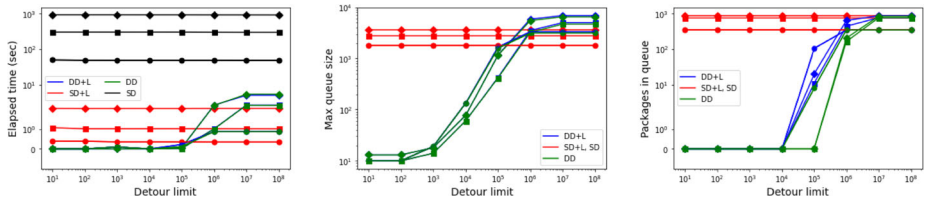
Fig. 5 The figure shows the effect of varying k on (a) elapsed time, (b) the maximum size of the priority queue, and (c) the number of packages inserted into the priority queue before the algorithm terminated. Note that square, circle, and diamond symbols refer to the SF, US-NW, and CA-NV datasets, respectively

Each experiment was run 100 times and we averaged over those runs. We measure the latency, maximum size of Q, and the number of packages that are found during the search process which are in turn inserted into Q.

The plot shown in Fig. 5(a) demonstrates that the running time increases with k. It can be seen from the graphs that DD is much better than SD, while SD+L and DD are similar in their performance. From $k = 100$ onwards, the elapsed time of DD, DD+L, and SD+L converge, indicating that these approaches start converging in terms of efficiency. Similarly, the queue size increases when the number of nearest packages increases shown in Fig. 5(b). Figure 5(c) shows that SD and SD+L have lesser pruning power compared to DD and DD+L which are able to prune away most of the packages explaining their superior run times.

Finally, note that our largest dataset, CA-NV is about 4X the size of the smallest one (i.e., SF). Yet DD and DD+L have fairly low latency, even for the larger road networks. For small values of k, the DD and DD+L methods are quickly able to give answers that are orders of magnitude faster than the strawman approaches. This means that in our use-case, where there is a low match between the drivers and packages, one can provide answers in near real-time and efficiently.

Varying Detour Tolerance For the same source and destination chosen at random, we vary the detour tolerance limit from 10 to 100M. Note that the units here are the same as the graph weights. For each of these runs, we set k to 1000 and the number of packages was fixed as 1000. Each experiment was run about 100 times and the average of elapsed time, maximum Q size, and the number of packages inserted into Q are recorded. The plot shown in Fig. 6(a) shows that the running time slightly increases when the detour tolerance limit increases. The



(a) detour tolerance limit vs elapsed time

(b) detour tolerance limit vs maximum size of the priority queue

(c) detour tolerance limit vs number of packages inserted into the priority queue

Fig. 6 The figure shows the effect of varying detour tolerance. The figure shows (a) elapsed time, (b) the maximum size of the priority queue, (c) the number of packages inserted into the priority queue, as the detour limit is increased from 10 to 10^8 for fixed values of $k = 1000$ and the number of packages fixed at 1000. Note that square, circle, and diamond symbols refer to the SF, US-NW, and CA-NV datasets, respectively. Also, note that DD+L and DD are overlapping with each other and have similar run-times

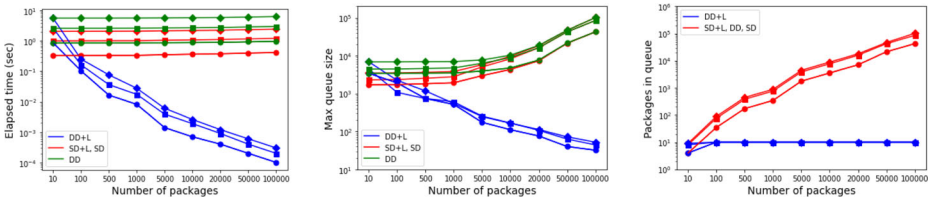
figures show that DD and DD+L are much more adept at pruning the search space compared to SD. The reason is that DD quickly establishes the shortest path much faster than SD and so is able to terminate the search once the shortest path is established. At very large detour values, DD and DD+L become slightly worse than SD+L since the detour has become large enough that it has become a scan of most of the road network. Even then, the execution times are fairly small, so the difference does not matter much. Note that the lack of pruning for SD and SD+L means that the performance is stable yet consistently worse than the competing variants.

Q size increases when the detour tolerance limit gets very large as we can see from Fig. 6(b) since that negates the pruning effect of DD and DD+L. Finally, from Fig. 6(c) we can see that the speed-up of DD also comes from the observation that fewer packages are inserted into Q. This shows that the search space of the DD and DD+L is much smaller compared to SD and SD+L. We did not see a significant difference in behaviour when comparing the SF, US-NW, and CA-NV datasets, indicating robustness across datasets of various sizes. For small detour tolerances, DD and its variants are good at quickly arriving at the answer. This is critical for our use-case, where there is a low match between the drivers and packages. In this domain, it is as important to provide negative answers (i.e., of no match) as quickly as a positive match.

Varying Number of Packages For the same source and destination chosen at random, we now vary the number of packages varying from 10 to 100,000, while not specifying a detour tolerance limit and keeping k at 10. Each experiment was run about 100 times and we obtained the three metrics of interest as in the previous experiments.

The plot shown in Fig. 7(a) shows that as the number of packages increases it becomes easier to find the 10 nearest neighbours without having to traverse too much of the road network. In some sense, the DD+L works much more efficiently as the packages increase since matching between drivers and packages becomes easier, while SD+L and DD have similar performance. The landmark variant improves the pruning power of the algorithms; hence, SD+L's performance becomes quite similar to that of DD. To explain the difference between DD and DD+L, there are two competing forces at play here.

1. The more the number of packages, the quicker the DD algorithm can find a fully resolved package, and hence the upper bound of the search is established. This is due to forward and backward scans steadily advancing in each iteration.
2. The more the number of packages there will be, the more packages would be found in the process of searching and needs to be tracked in the priority queue. Note that the package cannot be removed from the priority queue until it is fully resolved, reinserted, and then obtained from the front of the priority queue.



(a) #packages vs. elapsed time (b) #packages vs. max size of the priority queue (c) #packages vs. #packages in the priority queue

Fig. 7 The figure shows the effect of varying the number of packages. The figure shows (a) elapsed time, (b) the maximum size of the priority queue, (c) the number of packages inserted into the priority queue, as the number of packages increased from 10 to 100,000 for $k = 10$ with no detour distance limit. Note that square, circle, and diamond symbols refer to the SF, US-NW, and CA-NV datasets, respectively

DD gains from the first one but suffers from the second, resulting in a net increase as the number of delivery packages increases. On the other hand, DD+L is not affected by the second aspect since one can immediately resolve the package fully if the forward scan reaches either the pickup location or the backward scan reaches the drop location. Hence, there has been a clear improvement with an increase in the number of packages, as can be seen from Fig. 7.

As in the previous experiments, in Fig. 7(b, c) we can see that DD is much more efficient than SD and its variants in terms of visiting fewer nodes and inserting fewer packages in Q.

Note that the improved performance of DD+L as the number of packages increases is an important aspect of the algorithm from a platform scalability perspective. We did not see a significant difference in behaviour when comparing the SF, US-NW, and CA-NV datasets, indicating robustness across datasets of various sizes.

7.2 General package delivery queries

In this section, we examine the performance of the more general package delivery variant where the driver is travelling between a source and a destination and opportunistically delivering packages. To better understand the performance of the algorithms we break the algorithm into four cases depending on the distances between the pickup and drop-off location of the packages. In particular, we consider packages with drop-offs greater than 10k,

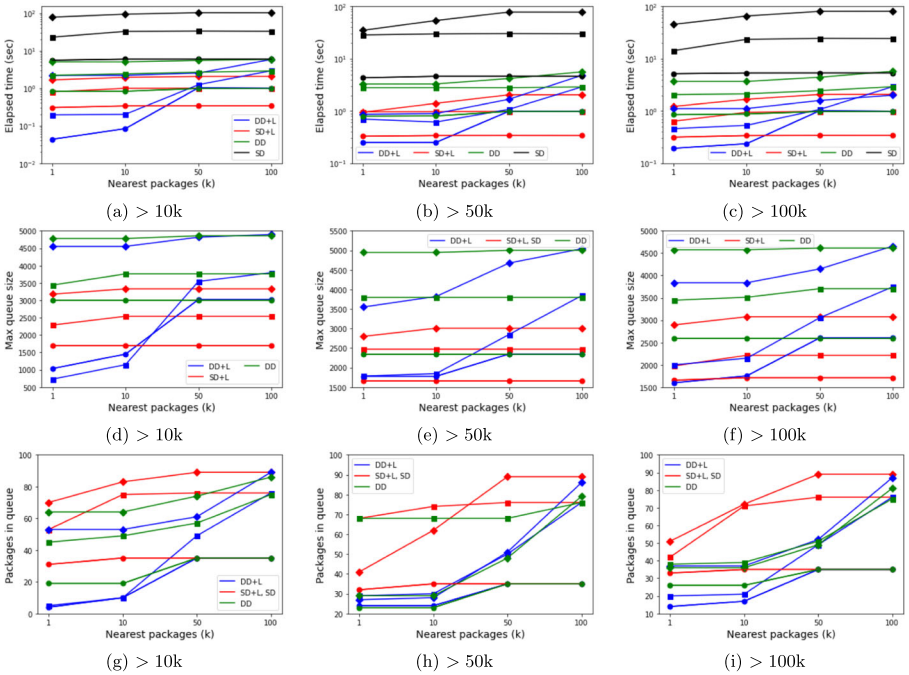
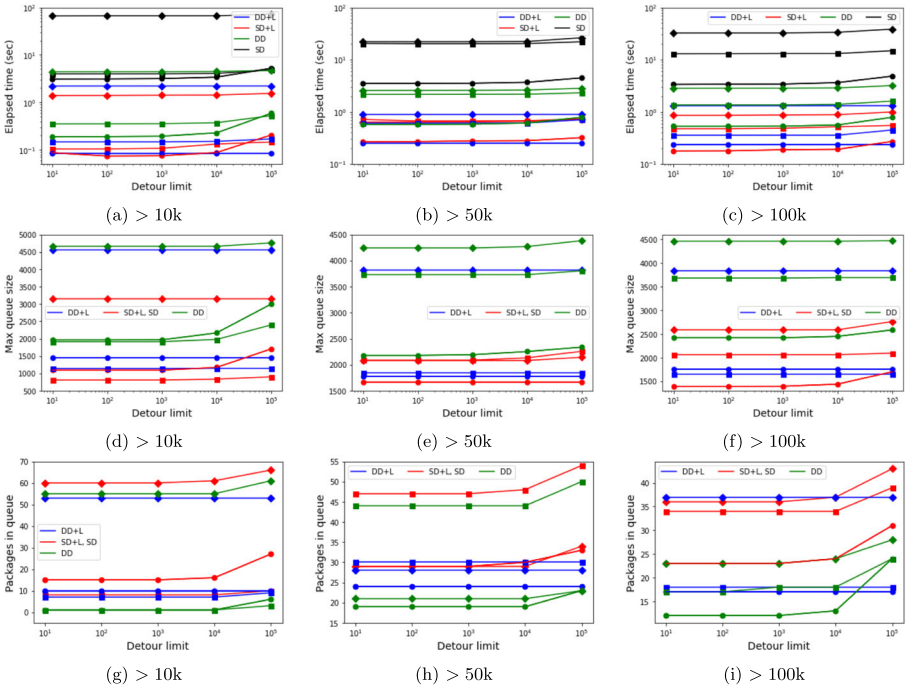


Fig. 8 The figure shows the effect of varying k-nearest packages where the sources and destinations are separated by greater than 10k, 50k, and 100k. Figure (a–c) shows the effect on elapsed time, (d–f) the maximum size of the priority queue, and (g–i) the number of packages inserted into the priority queue. Here we do not set a detour tolerance and the number of packages is kept at 100. Note that square, circle and diamond symbols refer to the SF, US-NW, and CA-NV datasets, respectively

50k and 100k distance units. These denote the relative hardness of matchmaking between the driver's shortest path and the package delivery requirements. Note that the distance between the pickup and drop-off locations also denotes an expansion of the search space resulting in larger run times.

Varying Number of Neighbours For different sources and destinations chosen at random, three different experiments were conducted for the varying distance between two packages $>$ 10k, $>$ 50k, and $>$ 100k. The number of k-nearest packages is varied from 1 to 10, to 50, and to 100 while the detour limit was kept fixed at 1M and the number of packages fixed at 100, run each experiment 100 times and get the average elapsed time, the maximum size of Q, and the number of packages inserted into Q. The plot shown in Fig. 8(a–c) illustrates that the running time increases when the number of nearest packages increases. As the distance between pickup and drop-off increases, DD+L, SD+L, and DD start having similar performance since the pruning power of the algorithm reduces significantly, yet DD+L produces consistently better performance. Similarly, Q size increases when the number of nearest packages increases shown in Fig. 8(d–f) but in this case single Dijkstra performs better than DD. The number of packages inserted into Q increases when the number of nearest packages increases shown in Fig. 8(g–i). In this case, the performance of both SD and DD are almost equal.

Varying Detour Tolerance For different sources and destinations chosen at random, three different experiments were conducted for the varying distance between two packages $>$ 10k,



(a) $>$ 10k (b) $>$ 50k (c) $>$ 100k

(d) $>$ 10k (e) $>$ 50k (f) $>$ 100k

(g) $>$ 10k (h) $>$ 50k (i) $>$ 100k

Fig. 9 The figure shows the effect of varying the detour limit where the pickup and drop-off locations are separated by greater than 10k, 50k, and 100k distance units. Figure (a–c) shows the effect on elapsed time, (d–f) the maximum size of the priority queue, and (g–i) the number of packages inserted into the priority queue. Here k is fixed at 10 and the number of packages is kept at 100. Note that square, circle, and diamond symbols refer to the SF, US-NW, and CA-NV datasets, respectively

$>$ 50k, and $>$ 100k. The detour tolerance limit is varied from 10 to 100k while keeping $k = 10$ and the number of packages fixed as 100, run each experiment 100 times, and get the average elapsed time, the maximum size of Q, and, the number of packages inserted into Q.

The plot shown in Fig. 9(a–c) illustrates that the running time increases when the number of nearest packages increases. Similarly, Q size increases when the number of nearest packages increases shown in Fig. 9(d–f) but in this case, SD performs better than DD. The number of packages inserted into Q increases when the number of nearest packages increases shown in Fig. 9(g-i). In this case, the performance of both SD+L, DD, and DD+L are similar but DD+L is superior overall.

Note that this set of experiments captures the negative use-case where, for small detour tolerances, there may be no packages that match the specifications. In our problem setup, this is a common case since the matchmaking rates are expected to be low. In these situations, the landmark variants and DD are quite efficient. This is an important result that suggests that our algorithm can be used when a driver is actively using a navigation app for the shortest route and our algorithm may be able to show in-route packages with a low detour tolerance in real-time.

Varying Number of Packages For different sources and destinations chosen at random, three different experiments were conducted for the varying distance between two packages $<$ 10k, $<$ 50k, and $<$ 100k. The number of packages varying from 10 to 100k while keeping no detour tolerance limit and fixing $k = 10$, and run each experiment 100 times and get

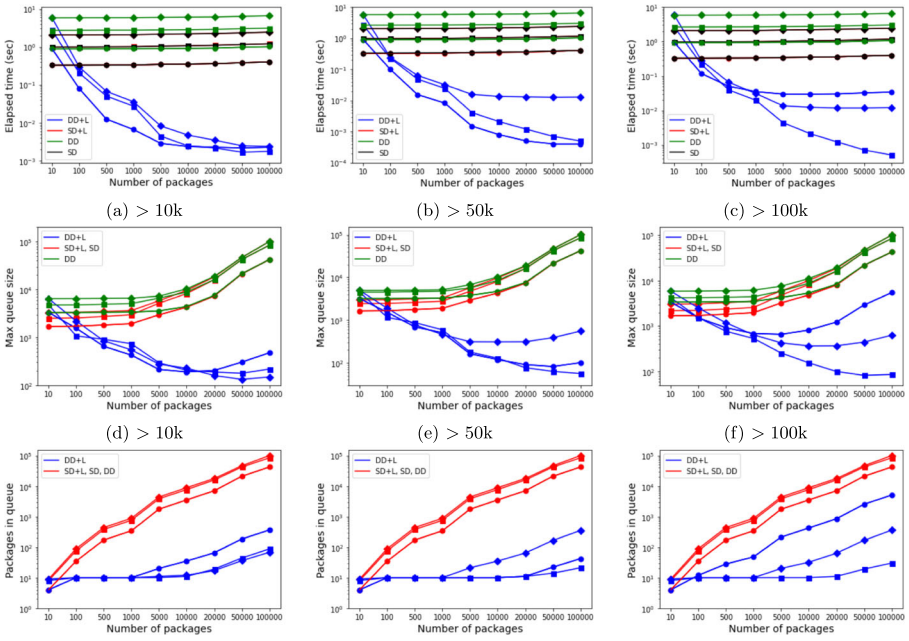


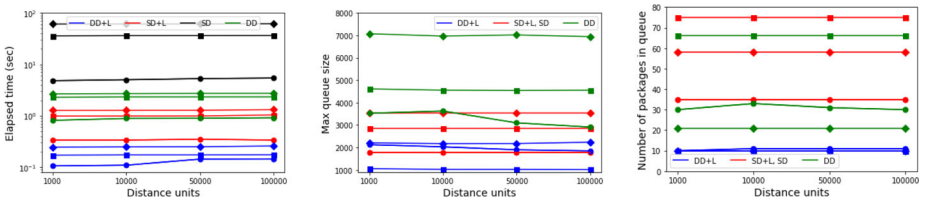
Fig. 10 The figure shows the effect of varying the number of packages where the pickup and drop-off locations are separated by greater than 10k, 50k, and 100k. Figure (a–c) shows the effect on elapsed time, (d–f) the maximum size of the priority queue, and (g–i) the number of packages inserted into the priority queue. Here k is fixed at 10 but no detour limit is specified. Note that square, circle, and diamond symbols refer to the SF, US-NW, and CA-NV datasets, respectively

the average elapsed time, the maximum size of Q, and the number of packages inserted into Q. The plot is shown in Fig. 10(a–c) illustrates that the running time increases when the number of nearest packages increases. Similarly, Q size increases when the number of nearest packages increases shown in Fig. 10(d–f). The number of packages inserted into Q increases when the number of nearest packages increases shown in Fig. 10(g–i).

A few things are worth mentioning here. The performance of the algorithms is fairly robust across the three datasets, SF, US-NW, and CA-NV. Furthermore, DD+L performance improves as the number of packages increases, which shows its pruning power. SD and SD+L seem not to be sensitive to the number of packages since their pruning power is limited compared to the DD variants. Note that DD and DD+L can establish the shortest path between the source and destination faster compared to the SD variants. Finally, queue sizes increase with the number of packages, as expected. However, DD+L manages to still buck that trend since it can terminate quickly.

7.3 Varying source-destination distance

In the final set of experiments, we vary the distance between the sources and destinations for the driver while keeping $k = 10$ and the number of packages fixed at 100. Note that at the onset, we kept the packages fairly sparse to illustrate a certain aspect of the algorithm. From the results, one can see in Fig. 11a that the runtime is weakly dependent on the distance between the source and the destination. The reason for it is that the algorithm consists of two searches - one to establish the shortest path between the source and destination, and another to find k packages. The former search becomes larger as the distance between the source and the destination increases. However, in our setup, the latter dominates the former since the number of packages in Q and the maximum queue sizes are fairly constant, as one can see in Fig. 11(b–c). It means that the effort of finding the k packages dominates the runtime here, thus weakening the effort of finding the distance between the source and destination. Hence, one can see that the runtimes are mostly the same, though DD+L still performs the best.



(a) distance vs elapsed time

(b) distance vs maximum size of the priority queue

(c) distance vs number of packages inserted into the priority queue

Fig. 11 The figure shows the effect of the varying source to destination distance from 1k to 100k. In particular, (a) elapsed time, (b) the maximum size of the priority queue, (c) the number of packages inserted into the priority queue are shown for fixed values of $k = 10$, and unlimited detour tolerance. The number of packages was 100. Note that square, circle, and diamond symbols refer to the SF, US-NW, and CA-NV datasets, respectively

7.4 Clustered packages

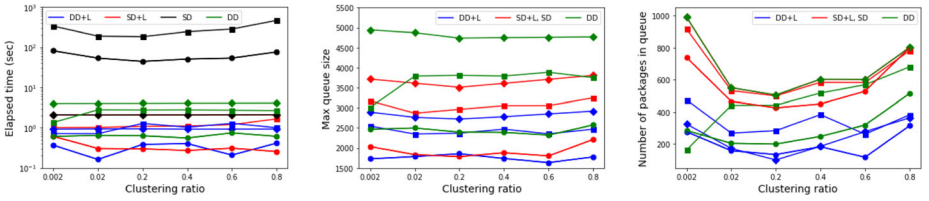
When it comes to delivering packages, the "clustering" of packages is a natural occurrence in this problem domain and manifests itself due to a few reasons. Packages may be collected in processing centres, from which they are dispatched to various destinations. In this case, the driver would have many packages to choose from if the search ever visited such a processing center. Similarly, clustering is also possible on the drop-off side of things. A prolific business may get packages from many locations so clustering in the drop-offs are also commonplace.

To understand the effect of clustering on the problem, we define the clustering ratio as the ratio of the number of packages available to be shipped (or received) at a location to the total number of packages. For instance, if there are 500 packages to deliver, a clustering ratio of 0.1 means that there are 50 packages per location and in 10 unique locations on the road network. Similarly, a clustering ratio of 0.02 means all the package pickup (or drop-off) locations are unique. Then, as discussed earlier, there is pickup or drop-off clustering, or possibly both, since any realistic scenario consists of a mixture of both processing centers and prolific businesses. Our experiment considers this mixed case, where the pickup and drop-off clustering are both equally likely.

Finally, one can study these effects as a function of the distance between the sources and destinations of the drivers. The problem becomes challenging if the sources and destinations are sufficiently far away, in which case they may end up visiting many such clusters during the search. So, we limit the experiments where the sources and destinations are greater than 100k units which denote far enough sources and destinations on the road network. Furthermore, the packages themselves have pickup and drop-off locations $> 10k$ distance units.

Figure 12(a–c) shows the result of the clustering experiments. The setup is as follows. For each of the datasets used in the evaluation, we fix the number of packages at 500, and the neighbours to find fixed at 100. We do not specify a detour limit for this experiment. We perform three sets of experiments denoting pickup, drop-off, and mixed clustering. In each case, the clustering ratio is varied between 0.002 and 0.8 indicating a spectrum of cases from no clustering to the case where half the pickup and drop-off locations are clustered.

There are two opposing forces to note in this experiment. When there are large clusters, then the packages are all found in one place. The complexity then shifts to ordering the



(a) clustering ratio vs elapsed time

(b) clustering ratio vs maximum size of the priority queue

(c) clustering ratio vs number of packages inserted into the priority queue

Fig. 12 The figure shows the effect of varying cluster sizes. The figure shows (a) elapsed time, (b) the maximum size of the priority queue, (c) the number of packages inserted into the priority queue, as the cluster size is increased from 0.002 to 0.8 for fixed values of $k = 100$ and the number of packages fixed at 500. Note that square, circle, and diamond symbols refer to the SF, US-NW, and CA-NV datasets, respectively

packages by their detour distances. When packages are not clustered, the complexity is in finding enough packages to satisfy k. Here, the initial search is more time consuming than ordering them. These two opposing forces cancel each other out, and the latencies are fairly unremarkable for most of the algorithms. If the algorithm during the search reaches one of the large clusters, while the landmark approaches can partially prune some of them, the other approaches add most of them to Q. This can be seen in Fig. 12c which has a u-shaped curve reflecting the above observation. In a real-world situation, where there may be a mix of clusters and sparse packages, the presence of clusters does not hurt the performance of the algorithms, which is a useful result to note.

7.5 Comparison with ridesharing approach

Figure 13 shows a comparison between our PDaaS algorithm and a modified version of a ridesharing algorithm. The ridesharing algorithm we chose is the algorithm by Geisberger et al. [18]. A reduction of this approach to the PDaaS problem is as follows. First, we compute the forward and backward distances of the packages and store them as we do in the DD+L algorithm. When the driver arrives at the system, we compute the shortest path between the source and the destination. Once the shortest path is known, we use the backward and forward distances of the source and destination from each of the packages to compute their induced detour. We insert those satisfying the constraints into a priority queue and choosing the k packages with the lowest detour distances. In this formulation, we gave the ridesharing algorithm the same benefits as we gave DD+L in the sense that the forward and backward

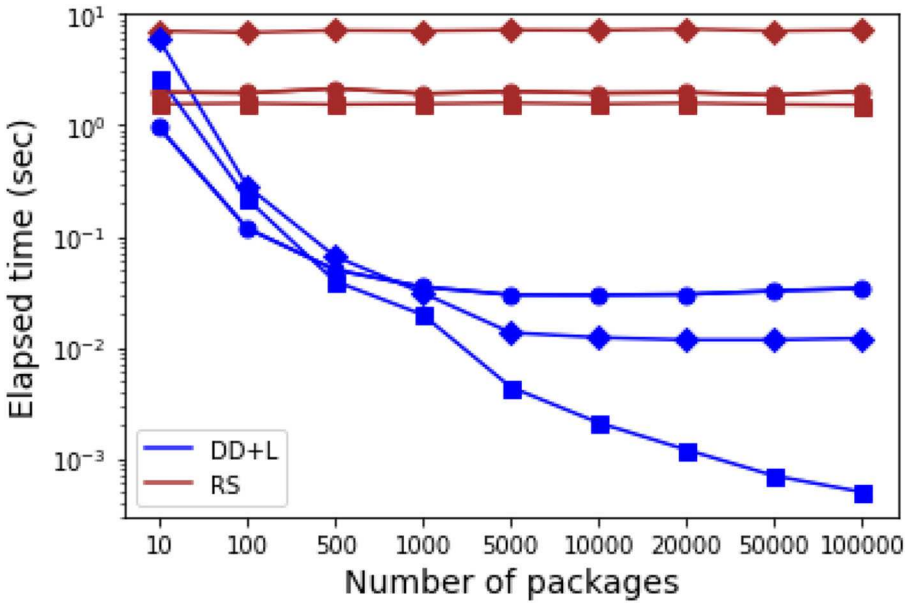


Fig. 13 The figure shows the effect of varying the number of packages where the pickup and drop-off locations are separated by greater than 100k. The figure shows the effect on elapsed time as the number of packages available for delivery increases. RS here refers to a ridesharing implementation that has been adapted for the PDaaS use-case. Here k is fixed at 10, but no detour limit is specified. Note that square, circle, and diamond symbols refer to the SF, US-NW, and CA-NV datasets, respectively

distances from each of the packages are computed in an offline process and available by the time the source and the destination of the driver are known.

We recreated the experiment in Fig. 10c, where the number of packages varied from 10 to 100k. Here, k is kept at 10 and a detour is not specified which means that k packages with the lowest detour are accepted. Furthermore, sources and destinations are kept $> 100k$ units apart indicating the case that they are quite far away from each other. The figure shows a comparison of dual Dijkstra's algorithm with landmarks (DD+L) with the ridesharing (RS) reduction. One can immediately see that the RS is much higher than DD+L and furthermore, does not decrease with an increasing number of packages.

The reason for the performance of RS being worse than DD+L is that in the case of RS the packages are only processed after the shortest path and distances between the sources and destinations are known. However, in the case of DD+L, the shortest path uses the packages as landmarks to prune away the shortest path resulting in a very unique and desirable property that the more packages we add to PDaaS the faster the answers can be retrieved. This is unique to our solution and as one can see is also very effective.

Having shown that our approach can hold its own against a ridesharing strawman, it is important to remember that the source of the complexity of the ridesharing is in choosing the subset of riders and in sequencing the pickups and drop-offs. PDaaS deals with a simpler problem of quickly pruning and ordering packages as drivers interact with a navigation app. These are different problems with widely different sources of complexity.

8 Related work

In this paper, we propose a PDaaS platform for delivering thousands of packages on the road network. The focus of our approach is to develop algorithms where matchmaking becomes more efficient as the number of packages in the system increases. Our work bears similarities to many prior works and we discuss them below.

On-the-fly Nearest Neighbour computations: One of the first studies on nearest neighbour query on road network distance was by Papadias et al. [34] which proposed a *spatial network database* (SNDB) architecture to process spatial queries on the directed road network. They proposed two pruning techniques, namely, INE and IER for computing the nearest neighbour in SNDB. To speed-up query processing they applied traditional R-tree based spatial access methods. IER is based on the fact that the lower-bound of road network distance between two locations can be established via the Euclidean or "crow-flying distance". INE traverses the road network to adjacent nodes similar to Dijkstra's algorithm and retrieves the nearest POI from the query point by looking up the spatial index for POIs that are associated with the node or edge. Many subsequent works [26, 27] have expanded on the IER and INE approaches. Hu et al. [25] proposed a distance signature based efficient precomputation method by indexing road network distances for POIs. Distance signature is an efficient alternative, a precomputation, and an index-based query processing scheme for distance computation. Abeywickrama et al. [1] optimised and open-sourced many nearest neighbour implementations and showed that when it comes to performance the quality of the implementation mattered. Compared to these approaches our technique is similar to INE except that we resort to forward and backward scans. The use of a priority queue to store the forward and backward nodes, as well as the packages makes our approach similar to the seminal incremental nearest neighbour finding [24] by Hjaltason and Samet.

Precomputation Techniques: Network Voronoi diagram based method [28] and regular 2D grid-based method [57] are few techniques in preprocessing based partitioning into regions of interest where a particular property holds (e.g., the region has the same neighbour in the case of Voronoi diagrams). Dickerson et al. [13] extended the concept of network Voronoi diagrams to roundtrips on undirected graphs using triangular inequality to prune the search space. SILC [42, 45] and distance oracles [43, 44, 46] precompute shortest paths in a road network to speed-up computations. They introduced the concepts of distance and path oracles in order to perform a variety of operations on road networks. Ghosh and Gupta [19] proposed roundtrip distance in a road network with respect to the nearest neighbour query and proposed a preprocessing based algorithm. Our work is different from these approaches in that our caching strategy is not a general purpose in the sense of aiding any arbitrary computations on the road network. In our case, the cache is tailored to deliver a package and discarded once the package is delivered.

Spatial Indexing Techniques: There is a rich body of work on range search queries and their variations in the areas of computational geometry [2, 32], spatial databases [34, 51], and spatial road networks [5, 6, 22, 25, 29, 50, 52]. In this context, Lee et al. [29] propose an efficient and flexible system framework named ROAD, after two index structures, namely, *Route Overlay* (RO) and *Association Directory* (AD). They further describe network expansion-based search algorithms that efficiently prune the search space for processing location-based spatial range queries on the ROAD framework. Sun et al. [50] propose a preprocessing based (grid) partition algorithm that is based on a spatial index called *Network Partition Index* (NPI) to process spatial range queries efficiently on undirected road networks. Xuan et al. [52] develop a variant that can only answer restricted range search queries based on the properties of network Voronoi diagrams on road networks. Bae et al. [6] examine algorithms for evaluating rectangular spatial range queries on the web data by using only *k-Nearest Neighbour* (k-NN) queries. An et al. [5] present a novel set of analysis techniques to estimate range query performance on spatial databases. They analyse point data sets and rectangular data sets. Bao et al. [7] propose an efficient algorithm for processing a k-range nearest neighbour query in road networks. The concept of the k-nearest neighbour query has been further extended to find the k-range nearest neighbour, which finds the k-nearest objects of interest to every road segment forming the shortest path. In contrast to these approaches, our technique is distinguished in a few ways. First, the object of interest has pickup and drop-off locations and due to the nature of the problem, we require the driver to drop-off the package before reaching the destination. Second, we do not index the road network but rather index the packages which make sense for our use-case.

In-route and Detour Queries: Yoo and Shekhar [53] propose the problem *In-route Nearest Neighbour* (IRNN) query. In their formulation, they gave a route with a current location and a destination. Their approach called IRNN finds a POI via which the detour from the query route on the way to the destination is the smallest. Saha et al. [41] propose *obstructed* detour queries and developed an efficient solution for processing such queries. Obstructed detour queries return the nearest POI with respect to the current location and the fixed destination in presence of obstacles like private property or a fence. The distance to a POI is measured as the summation of the obstructed distance from the traveller's current location to the POI and the obstructed distance from the POI to the destination. Shang et al. [48] devised optimization techniques for efficient processing of *best point* detour query in road networks. Given a preferred path P from a source location to a destination location, to be travelling on the

road network, the best point detour query aims to efficiently identify the best detour POI, i.e., detour with the minimum detour cost on the path to be travelling on along P. Nutanong et al. [33] study the problem of finding the shortest route between two locations that include a stopover (stopover should not introduce significant costs to the trip) of a given type, i.e., they are interested in minimising the total trip distance between two locations while covering a given type of stopover. While our algorithm is related to these papers, one key distinguishing aspect of our system is that drivers come with their own preferences, so indexing the road network does not work very well, so instead, we focus on indexing the package.

Ridesharing: Yuen et al. [56] investigate the shortest path for ridesharing queries, and propose an optimal route recommendation algorithm based on dynamic programming, which also reduces failure rates to find ridesharing partners. In the ridesharing environment, a permissible detour threshold is imposed by the car driver for multiple compatible ride-orders to the travellers who wish to ride in the same car. Mahin and Hashem [31] introduced and formulate a new type of ridesharing query in road networks that considers passengers' flexibility in selecting passengers, and ensures a complete ridesharing group trip of passengers. They call their problem *activity-aware ridesharing* which is different from a single driver trip. Geisberger et al. [18] develop a novel pruning strategy, based on distance table precomputation to efficiently compute detours and find a reasonable match for spontaneous ridesharing requests of prospective passengers and offers by drivers available on short calls in a dynamic environment for the large road network and for a large number of passengers each day. Furthermore, Chen et al. [9] investigate the problem of *path nearest neighbour (PNN) query* in road networks, where the inputs are the current location and the destination. Each path nearest neighbour query constructs a shortest path connecting the user's current location and the destination and then searches for the closest POI (with the minimum detour distance) with respect to the whole query path. Note that while ridesharing focuses on the order of picking up and dropping-off passengers, we are concerned with single package pickup and drop-off which makes our approach different. For instance, we are concerned with delivering thousands of packages which is typically not the problem space for ridesharing queries.

Spatial Computing Platforms: The current trend in spatial data literature is data platforms that are specialised for spatial operations. Yuan et al. [55] propose a two-stage routing algorithm to compute the fastest route to a destination at a given departure time in terms of taxi drivers' intelligence learned from a huge number of real-world historical GPS taxi trajectories. Eldawy and Mokbel [15] develop *SpatialHadoop*, which is a full-fledged MapReduce simplified programming framework designed specially to work with large distributed spatial data efficiently with native support for spatial data available as free open-source. SpatialHadoop was adopted as a solution for the scalable processing of huge datasets in many applications (e.g., machine learning, graph processing, etc). SpatialHadoop is a comprehensive extension to Hadoop that injects spatial data awareness into each of four main Hadoop layers, namely, the language, storage, MapReduce, and operations layers. The operations layer is equipped with three basic spatial operations, namely, k-nearest neighbour, range query, and spatial join. Yu et al. [54] presented a detailed design and development of GEOSPARK (extends the core engine of Apache Spark and SparkSQL) to support spatial indexes, data types, and geometric operations.

Peng et al. [36] propose a novel distance oracle on large road networks to address various spatial analytic queries with high throughput. Demiryurek et al. [12] presented a real-world data-driven system, named, Transportation Decision-Making (TransDec) that enables interactive and extensive spatiotemporal queries in transportation systems with dynamic, real-time, and historical datasets (e.g., massive traffic sensor data, trajectory data, transportation network data, and point-of-interest data). TransDec addresses the challenges in monitoring,

visualisation, querying, and analysis of dynamic and large-scale transportation data. Hendawi et al. [23] propose a spatial index structure, predictive tree (P-tree), for processing common types of predictive queries (e.g., predictive point, range, k-NN, aggregate queries) against moving objects on road networks based on the objects' expected future locations, with the help of user-defined functions. Peng et al. [35] present a framework called Spark and Distance Oracles (SPDO), which is an approximate distance oracle that implements an extremely fast distributed algorithm for computing road network distance queries on Apache Spark. Our work in this context can be seen as a spatial data platform that can perform matchmaking between the drivers and the packages. While many of the approaches above provide general purpose spatial processing, our focus is only on delivering and assigning packages to drivers.

9 Conclusion

In this paper, we envisioned a platform for package delivery, called PDaaS, that opportunistically performed the matchmaking between drivers that are already travelling between two locations. PDaaS matched a driver to a single package while respecting constraints imposed by detour limits and the number of packages to deliver. Through extensive experimental analysis, we showed that the algorithm scales to thousands of packages with the algorithm getting more efficient as the number of packages increases. We also noted that this was desirable from a platform scalability perspective.

The key contribution in this paper is dual Dijkstra's method, which uses a single priority queue for tracking the shortest path between the source and destination, as well as both the resolved and unresolved packages. By adopting a sensible caching strategy, one can further speed-up the query. Note that for a large number of packages, small k, and detour tolerance, the proposed approach becomes more and more efficient since the packages themselves act as landmarks to guide the search process. This aspect of the algorithm, we think, is a powerful validation that the proposed approach could be used in the critical path of a driver navigating with a popular navigation app. There are many possible venues for the extension here. One interesting problem we note is that some packages are easily matched because of their proximity to a major thoroughfare, while others are relatively difficult because of the choice of either the pickup or drop-off. It would be interesting to use this in the pricing model for packages, which would combine road network information with other package attributes (such as distance between pickup and drop-off, etc.).

Author Contributions All the authors contributed to the problem definitions, algorithms, experiments, and related work. Debajyoti Ghosh and Jagan Sankaranarayanan contributed to the main manuscript text and prepared figures. Kiran Khatter and Hanan Samet contributed by suggesting revisions, extensions, and commenting on experimental results.

Funding Hanan Samet is funded by NSF award 2114451.

Data Availability The datasets generated during and/or analysed during the current study are available.

Declarations

Conflicts of interest The authors declare that they have no competing interests.

References

1. Abeywickrama T, Cheema MA, Taniar D (2016) k-nearest neighbors on road networks: a journey in experimentation and in memory implementation. In: Proceedings of the VLDB Endowment, vol 9(6), pp 492–503
2. Agarwal PK, Erickson J (1999) Geometric range searching and its relatives. Advances in Discrete and Computational Geometry, Contemporary Mathematics 223:1–56
3. Ahuja R, Magnanti T, Orlin J (1993) Network Flows: Theory, Algorithms, and Applications, 1st edn. Pearson
4. Ali RY, Gunturi VM, Shekhar S, Eldawy A, Mokbel MF, Kotz AJ, Northrop WF (2015) Future connected vehicles: challenges and opportunities for spatio-temporal computing. In: Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, vol 14, pp 1–4
5. An N, Jin J, Sivasubramaniam A (2003) Toward an accurate analysis of range queries on spatial data. IEEE Transactions on Knowledge and Data Engineering 15(2):305–323
6. Bae WD, Alkobaisi S, Kim S, Narayanappa S, Shahabi C (2009) Supporting range queries on web data using k-nearest neighbour search. GeoInformatica 13(4):483–514
7. Bao J, Chow CY, Mokbel MF, Ku WS (2010) Efficient evaluation of k-range nearest neighbor queries in road networks. In: Proceedings of the Eleventh International Conference on Mobile Data Management, pp 115–124
8. Cao B, Alarabi L, Mokbel MF, Basalamah A (2015) SHAREK: a scalable dynamic ride sharing system. In: 16th IEEE International Conference on Mobile Data Management, vol 1, pp 4–13
9. Chen Z, Shen HT, Zhou X, Yu JX (2009) Monitoring path nearest neighbor in road networks. In: ACM SIGMOD International Conference on Management of data, pp 591–602
10. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to Algorithms, 3rd edn. MIT Press
11. dataset06 (2006) 9th DIMACS implementation challenge - shortest paths. http://users.diag.uniroma1.it/challenge9/download.shtml
12. Demiryurek U, Banaei-Kashani F, Shahabi C (2010) TransDec: a spatiotemporal query processing framework for transportation systems. In: 26th International Conference on Data Engineering, pp 1197–2000
13. Dickerson MT, Goodrich MT, Dickerson TD (2010) Roundtrip voronoi diagrams and doubling density in geographic networks. Transactions on Computational Science 6970:211–238
14. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numerische Mathematik 1(1):269–271
15. Eldawy A, Mokbel MF (2015) SpatialHadoop: a MapReduce framework for spatial data. In: IEEE 31st International Conference on Data Engineering, pp 1352–1363
16. Eppstein D, Goodrich MT (2008) Studying geometric graph properties of road networks through an algorithmic lens. In: Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems, vol 16, pp 1–10
17. Eppstein D, Gupta S (2017) Crossing patterns in nonplanar road networks. In: Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, vol 40, pp 1–9
18. Geisberger R, Luxen D, Neubauer S, Sanders P, Volker L (2010) Fast detour computation for ride sharing. 10th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems pp 88–99
19. Ghosh D, Gupta P (2016) Roundtrip nearest neighbors on road networks for location based services. In: IEEE International Conference on Computational Intelligence and Computing Research, pp 310–313
20. Goldberg AV, Harrelson C (2005) Computing the shortest path: A* search meets graph theory. In: Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms, pp 156–165
21. Goldberg AV, Werneck RF (2005) Computing point-to-point shortest paths from external memory. In: Proceedings of the SIAM Workshop on Algorithms Engineering and Experimentation, pp 26–40
22. Gupta P (2015) Algorithms for road network range queries in location lased services. Proceedings of International Conference on Computational Advancement in Communication Circuits and Systems 335:261–267
23. Hendawi AM, Bao J, Mokbel MF, Ali M (2015) Predictive tree: An efficient index for predictive queries on road networks. In: 31st International Conference on Data Engineering, pp 1215–1226
24. Hjaltason GR, Samet H (1995) Ranking in spatial databases. In: Egenhofer MJ, Herring JR (eds) Advances in Spatial Databases—4th International Symposium, SSD'95, Portland, ME, LNCS series 951, pp 83–95
25. Hu H, Lee D, Lee VCS (2006a) Distance indexing on road networks. In: Proceeding of 32nd International Conference on Very Large Databases, pp 894–905
26. Hu H, Lee DL, Xu J (2006) Fast nearest neighbour search on road networks. International Conference on Extending Database Technology, LNCS 3896:186–203

27. Ilarri S, Mena E, Illarramendi A (2010) Location-dependent query processing?: Where we are and where we are heading. ACM Computing Surveys 42(3):1–67
28. Kolahdouzan M, Shahabi C (2004) Voronoi-based K-nearest neighbour search for spatial network databases. In: Proceedings of the 13th international conference on very large databases, vol 30, pp 840–851
29. Lee KCK, Lee WC, Zheng B, Tian Y (2012) Fast object search on road networks. IEEE Transactions on Knowledge and Data Engineering 24(3):547–560
30. Ma S, Feng K, Wang H, Li J, Huai J (2014) Distance landmarks revisited for road graphs. arXiv:1401.2690
31. Mahin MT, Hashem T (2019) Activity-aware ridesharing group trip planning queries for flexible POIs. ACM Transactions on Spatial Algorithms and Systems 5(3):1–41
32. Matousek J (1994) Geometric range searching. ACM Computing Surveys 26(4):421–461
33. Nutanong S, Tanin E, Shao J, Zhang R, Ramamohanarao K (2012) Continuous detour queries in spatial networks. IEEE Transactions on Knowledge and Data Engineering 24(7):1201–1215
34. Papadias D, Zhang J, Mamoulis N, Tao Y (2003) Query processing in spatial network database. In: Proceedings of the 29th VLDB Conference, vol 29, pp 802–813
35. Peng S, Sankaranarayanan J, Samet H (2016) SPDO: High-throughput road distance computations on Spark using distance oracles. In: Proceedings of the 32nd IEEE International Conference on Data Engineering, Helsinki, Finland, pp 1239–1250
36. Peng S, Sankaranarayanan J, Samet H (2018) DOS: a spatial system offering extremely high-throughput road distance computations. In: Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp 199–208
37. Peque G, Urata J, Iryo T (2018) Preprocessing parallelization for the ALT-algorithm. In: Proceedings of the 18th International Conference on Computational Science, pp 89–101
38. Pohl I (1971) Bi-directional search. Machine Intelligence 6:127–140
39. Potamias M, Bonchi F, Castillo C, Gionis A (2009) Fast shortest path distance estimation in large networks. In: Proceedings of the 18th ACM conference on Information and knowledge management, pp 867–876
40. Qiao M, Cheng H, Chang L, Yu JX (2014) Approximate shortest distance computing: a query-dependent local landmark scheme. IEEE Transactions on Knowledge and Data Engineering 26(1):55–68
41. Saha RR, Hashem T, Shahriar T, Kulik L (2018) Continuous obstructed detour queries. In: 10th International Conference on Geographic Information Science, vol 114, pp 1–14
42. Samet H, Sankaranarayanan J, Alborzi H (2008) Scalable network distance browsing in spatial databases. In: Proceedings of the ACM SIGMOD Conference, Vancouver, Canada, pp 43–54
43. Sankaranarayanan J, Samet H (2009) Distance oracles for spatial networks. In: Proceedings of the 25th IEEE International Conference on Data Engineering, Shanghai, China, pp 652–663
44. Sankaranarayanan J, Samet H (2010) Roads belong in databases. IEEE Data Engineering Bulletin 33(2):4–11
45. Sankaranarayanan J, Alborzi H, Samet H (2005) Efficient query processing on spatial networks. In: Proceedings of the 13th ACM International Symposium on Advances in Geographic Information Systems, Bremen, Germany, pp 200–209
46. Sankaranarayanan J, Samet H, Alborzi H (2009) Path oracles for spatial networks. PVLDB 2(1):1210–1221
47. Schultes D (2008) Route planning in road networks. PhD thesis, Institut fur Theoretische Informatik
48. Shang S, Deng K, Xie K (2010) Best point detour query in road networks. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp 71–80
49. Sommer C (2014) Shortest-path queries in static networks. ACM Computing Surveys 46(4):1–31
50. Sun W, Chen C, Zheng B, Chen C, Liu P (2012) An air index for proximity query processing in road networks. IEEE Transactions on Knowledge and Data Engineering 27(2):382–395
51. Taniar D, Rahayu W (2015) A taxonomy for region queries in spatial databases. Journal of Computer and System Sciences 81(8):1508–1531
52. Xuan K, Zhao G, Taniar D, Srinivasan B, Safar MH, Gavrilova M (2009) Network voronoi diagram based range search. In: Proceedings of the IEEE International Conference on Advanced Information Networking and Applications, pp 741–748
53. Yoo JS, Shekhar S (2005) In-route nearest neighbor queries. GeoInformatica 9(2):117–137
54. Yu J, Zhang Z, Sarwat M (2018) Spatial data management in Apache Spark: the GeoSpark perspective and beyond. Geoinformatica 23(1):37–78
55. Yuan J, Zheng Y, Zhang C, Xie W, Xie X, Sun G, Huang Y (2010) T-Drive: driving directions based on taxi trajectories. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp 99–108

56. Yuen CF, Singh AP, Goyal S, Ranu S, Bagchi A (2019) Beyond shortest paths: route recommendations for ride-sharing. In: The World Wide Web, p 2258-2269
57. Zheng B, Xu J, Lee WC, Lee L (2006) Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services. International Journal of Very Large Data Bases 15(1):21–39

Debajyoti Ghosh is currently pursuing a Ph.D. degree at the Department of Computer Science, BML Munjal University, India. He has over 15 years of teaching experience. He received his M.Sc from the Department of Computer Science and Engineering, University of Calcutta, and his M.Tech from the Department of Computer Science and Engineering, University of Calcutta. His research interests include location-based spatial query algorithms on the road network, the design of efficient algorithms, data structures, complexity analysis, and graph algorithms.

Jagan Sankaranarayanan works for Google in their data warehouse team. He received a Doctorate Degree in Computer Science from the University of Maryland in 2008. He is the recipient of best paper awards at ACM SIGMOD 2008, (10-year best paper) ACM SIGSPATIAL GIS 2018, ACM SIGSPATIAL GIS 2008, Computers & Graphics Journal 2007, a best paper nomination at the ICDE 2009 conference, and an "Excellent Invention Award of 2014" by NEC.

Kiran Khatter has over 16 years of experience in academics and industry. She completed M.Tech from Punjabi University, Patiala and Ph.D. from Himachal Pradesh University, Shimla. Her Ph.D. work was focused on the identification and quantification of non-functional requirements. Her research interests span the analysis of software quality using fuzzy modeling, machine learning, and nature inspired algorithms.

Hanan Samet is a Distinguished University Professor at the Computer Science Department at the University of Maryland, College Park. He received the B.S. degree in engineering from UCLA, and the M.S. degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University.

He is a Fellow of the IEEE, ACM, AAAS, and IAPR (International Association for Pattern Recognition). He received the Walton Visitor Award from the Science Foundation of Ireland serving in the National University of Ireland at Maynooth, the 2010 University of Maryland College of Computer, Mathematical and Physical Sciences Board of Visitors Distinguished Faculty Award, the 2009 UCGIS research award, the 2011 ACM Paris Kanellakis theory and practice award, the 2014 IEEE Computer Society Wallace McDowell Award, best paper awards in the 2008 ACM SIGMOD and SIGSPATIAL Conferences, the 2012 SIGSPATIAL MobiGIS Workshop, and the 2007 Computer & Graphics Journal best paper. The 2008 ACM SIGSPATIAL best paper award winner also received the 2018 ACM SIGSPATIAL 10 Year Impact Award. He is an ACM Distinguished Speaker (2008-2015, and 2018-present).